SASE HANDBOOK

Structured Agentic Software Engineering

A practical handbook for durable coding-agent workflows, tracked handoffs, reviewable changes, and provider-portable automation.

Site https://sase.sh/

Source https://github.com/sase-org/sase

CONTENTS

Handbook Chapters

1. Structured Agentic Software Engineering
2. SASE Blog
3. SASE Blog Series
4. ACE TUI User Guide
5. Axe Background Automation Daemon
6. Spec-Driven Development
7. XPrompts
8. ChangeSpecs
9. Beads
10. Workflow Specs
11. Workspaces
12. Mentors
13. Commit Workflows
14. Notifications
15. Mobile Gateway
16. Mobile MVP Runbook
17. Performance Runbook
18. Telemetry
19. Integration APIs
20. Plugins
21. LLM Providers
22. VCS Providers
23. Rust Backend
24. Configuration Reference
25. Query Language
26. ProjectSpec Reference
27. Agent Attachments and Image Previews

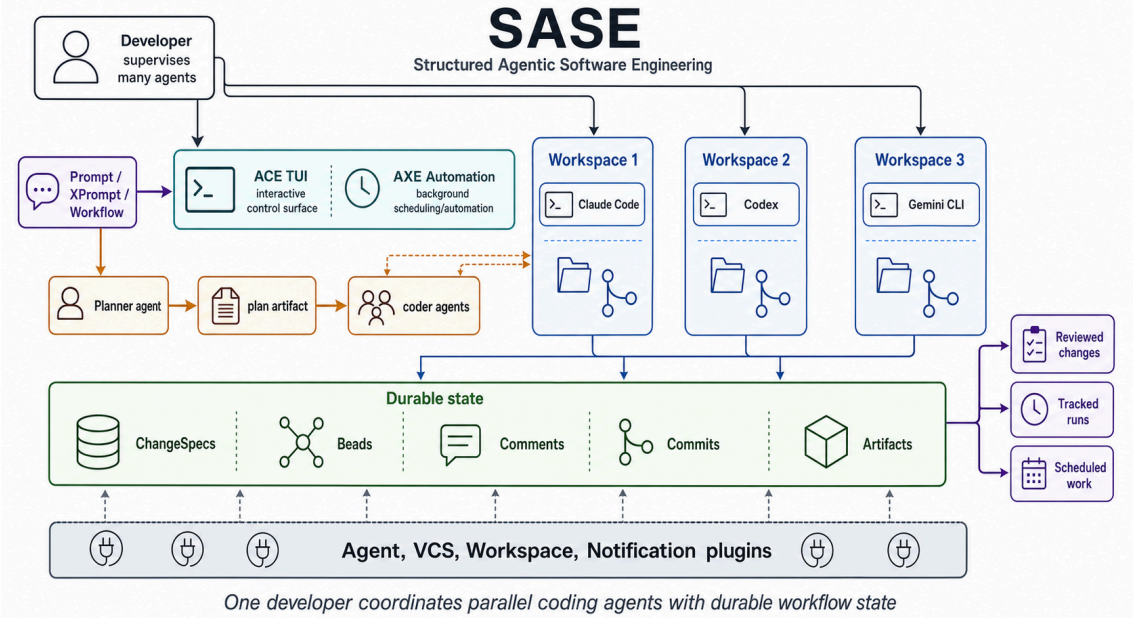
SASE

1 Structured Agentic Software Engineering

SASE is a Python toolkit for coordinating coding-agent work: durable plans, tracked handoffs, reviewable changes, resumable runs, and automation that can move across model and version-control providers.

15-Minute Quickstart | Download PDF | View on GitHub

SASE Blog Series



One developer coordinates parallel coding agents with durable workflow state

WHY SASE EXISTS

1.1 One prompt is not an engineering system

A single coding-agent run can produce a patch. Real projects also need a place to store intent, pass work between agents, order dependencies, track review state, retry failed runs, run background automation, and record what happened. SASE provides that coordination layer without tying the workflow to one model provider or one terminal app.

START BY ROLE

1.2 Pick the surface that matches your work

1.2.1 I am setting up a repo

Run explicit initialization subcommands to write agent memory, refresh generated SDD guide files, and inspect, preview, or deploy optional provider skill files before handing work to agents.

Open initialization (https://sase.sh/init/)

1.2.2 I want a TUI for agent work

Use ACE, the Agentic ChangeSpec Explorer TUI, to navigate ChangeSpecs, live agents, notifications, and automation state from one terminal interface.

Open the ACE guide (https://sase.sh/ace/)

1.2.3 I want durable work units

Use ChangeSpecs for CL/PR-sized review state and Beads for plan, epic, and phase dependencies that can drive multi-agent execution.

Learn the SDD flow (https://sase.sh/sdd/)

1.2.4 I need shared agent memory

Use instruction memory loaded through AGENTS.md, audited long-term reads, and reviewed proposals for agent-suggested updates.

Open memory (https://sase.sh/memory/)

1.2.5 I want reusable agent workflows

Use XPrompts for reusable prompt templates and workflow specs for repeatable multi-step automation.

Build with XPrompts (https://sase.sh/xprompt/)

1.2.6 I want implementation context

Use the architecture overview, command index, and development guide to understand the CLI surface, source layout, provider boundaries, and docs workflow.

Read the architecture map (https://sase.sh/architecture/)

1.2.7 I want editor completions

Use the xprompt LSP and editor helper bridge for prompt completion, snippets, hover, diagnostics, and jump-to-definition.

Open editor integration (https://sase.sh/editor/)

CORE PRIMITIVES

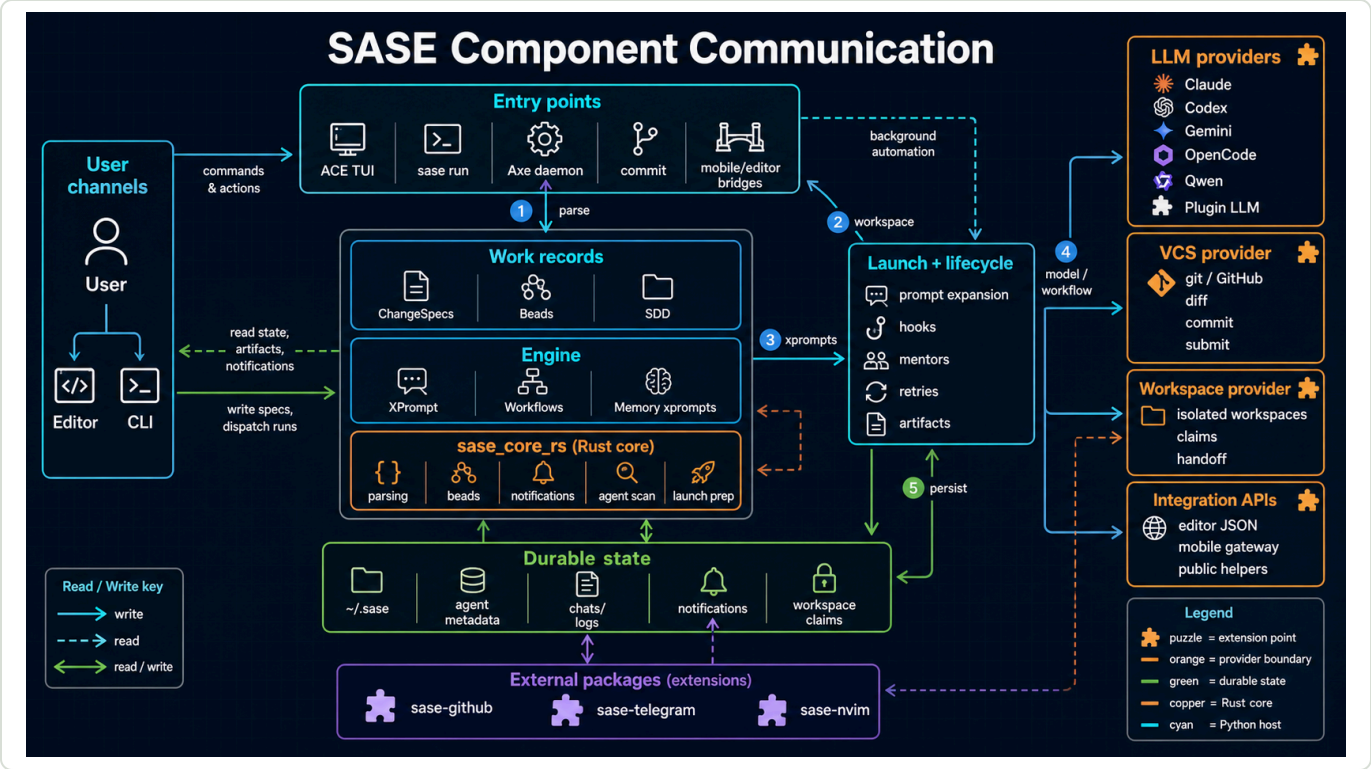
1.3 The coordination model

SASE keeps the work state outside the chat transcript. Plans, ChangeSpecs, beads, agent artifacts, and workflow records let agents be scheduled, resumed, reviewed, retried, or handed off without relying on one session's context window.

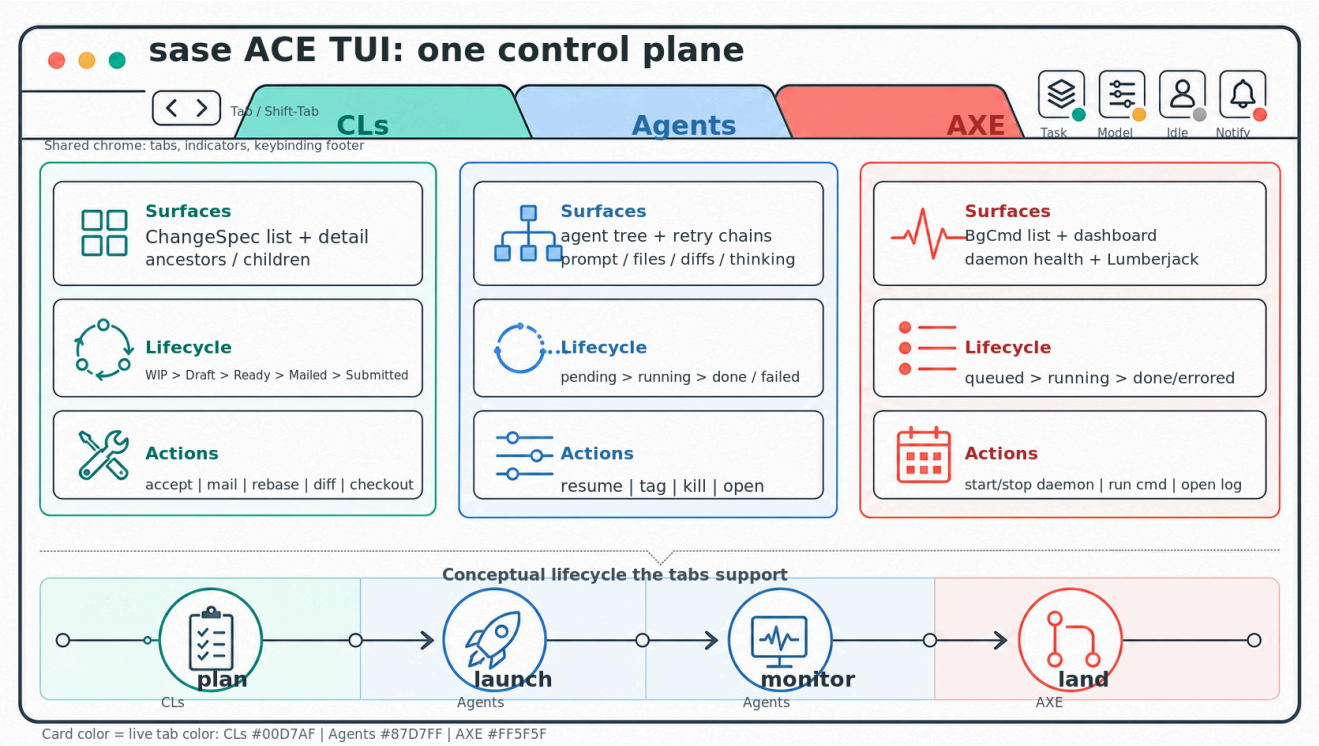
- **ProjectSpecs and ChangeSpecs** track project lifecycle, CL/PR-sized work, commits, review state, comments, mentors, and lifecycle transitions.
- **Beads** provide git-native issue tracking for plans, executable epics, phase dependencies, and agent handoff.
- **XPrompts** turn prompt templates into reusable workflows with reference expansion and typed inputs.
- **ACE** is the interactive control surface for daily work.
- **Axe Automation** runs background hooks, mentors, maintenance jobs, and scheduled workflows.
- **Provider and workspace abstractions** route agent launches, VCS operations, and workspace setup through plugin-backed boundaries.

HOW THE PIECES CONNECT

1.4 Designed around real agent handoffs



Component boundaries keep TUI, daemon, workflow, and provider concerns explicit.



ACE gives operators one place to inspect agents, changes, notifications, and automation state.

NEXT CLICKS

1.5 Move from overview to practice

1.5.1 Find a command

Use the CLI index to route from a command to its detailed owner page.

Open the CLI reference (https://sase.sh/cli/)

1.5.2 Understand the system

See how CLI, ACE, axe, workflows, providers, and the Rust core fit together.

Open architecture (https://sase.sh/architecture/)

1.5.3 Contribute locally

Review setup, verification commands, source layout, and docs deployment.

Open development (https://sase.sh/development/)

1.5.4 Read the launch essay

Why coding agents need orchestration above individual provider CLIs.

Read the essay (https://sase.sh/blog/posts/why-coding-agents-need-orchestration/)

1.5.5 Start with ACE

Learn the terminal interface for day-to-day SASE work.

Open ACE (https://sase.sh/ace/)

1.5.6 Explore the series

Read the full agentic software engineering series from one hub.

Explore the series (https://sase.sh/series/agentic-software-engineering/)

1.5.7 Download the handbook

Keep the current public docs and launch articles in one static PDF.

Download PDF (https://sase.sh/downloads/sase-handbook.pdf)

1.5.8 View the code

Inspect the implementation, issues, and project direction.

Open GitHub (https://github.com/sase-org/sase)

Initialization

SASE initialization commands create or refresh durable files that agents and companion tools rely on. Bare `sase init` checks the current project and home setup first, then either reports that everything is current or shows the initializers that need attention:

```
sase init -c       # report drift without writing
sase init          # prompt before each needed initializer
sase init --yes    # run every needed initializer in order
```

The coordinator plans in registry order: AMD, memory, SDD, then skills. Planning is read-only. In non-interactive shells, bare `sase init` reports drift and exits non-zero instead of prompting; use `sase init --yes` when you want an unattended apply run. Apply runs can write project files, deploy home files through chezmoi when configured, and use each initializer's normal commit/push behavior. Bare `sase init` only lets AMD generate managed project `AGENTS.md` content when the current project's own `./sase.yml` sets `amd_h1_title`; explicit AMD commands still repair provider shims and handle legacy one-file migrations when that field is unset.

Explicit subcommands are still available when you need narrower control:

```
sase amd init --check
sase amd init
sase amd list
sase init amd --check
sase memory init --no-commit
sase memory init --check
sase memory list
sase memory review --list
sase memory log
sase memory log --include proposals
sase memory log --path generated_skills.md
sase memory log --id <read-id>
sase init sdd
sase init sdd --check
sase skill list
sase skill init --dry-run
sase skill log
sase skill log --runtime codex

# Agent-side audited operations, normally run from a SASE-launched agent:
sase memory read generated_skills.md --reason "Need generated skill context"
sase skill use sase_plan --reason "Need to prepare an implementation plan"
sase memory write --title "Generated skills" --slug generated_skills --evidence chat:abc123 --body "Durable memory body" --notify
```

Start with `sase init -c`, `sase amd init --check`, or `sase memory init --check` when you only want a drift report. After that, `sase memory init --no-commit` is the usual first apply run for memory because it writes the generated files but skips the project git commit/pull/push path. It is not a dry run: it can still write project files, write home memory, and follow home-level `use_chezmoi` deployment. `sase init amd` remains a compatibility alias for `sase amd init`, `sase init memory` remains a compatibility alias for `sase memory init`, and `sase init skills` remains a compatibility alias for `sase skill init`.

Commands

Command	Purpose
`sase init`	Check AMD, memory, SDD, and skills; prompt once per needed initializer in interactive shells.
`sase init -c, --check`	Report initialization drift without writing and exit non-zero when changes are needed.
`sase init --yes`	Run every needed initializer in AMD, memory, SDD, skills order without prompting.
`sase amd`	Alias for `sase amd list`.
`sase amd list`	Inspect project, home, and chezmoi `AGENTS.md` files and nearby provider shims.
`sase amd init`	Create or refresh `AGENTS.md` files and provider shims for the selected AMD root or roots.
`sase amd init --check`	Report AMD initialization drift without writing files.
`sase init amd`	Compatibility alias for `sase amd init`.

`sase memory`	Alias for `sase memory list`.
`sase memory list`	Inspect loaded, referenced, available, and missing memory files for the current root.
`sase memory read <path>`	Agent-side read of one long-term memory file with an attributable audit event.
`sase memory write`	Create an attributable long-term memory proposal for human review.
`sase memory review`	List, inspect, approve, edit, or reject pending memory proposals.
`sase memory log`	Summarize audited long-term memory reads.
`sase memory log --include proposals`	Include proposal and review events in the memory audit surface.
`sase memory log --path <path>`	Show a path-level summary and matching individual read events.
`sase memory log --id <read-id>`	Show one full audited read event by id or unambiguous id prefix.
`sase memory init`	Create or refresh project/home memory roots and AGENTS memory references.
`sase memory init --check`	Report memory initialization drift without writing files.
`sase memory init -C`	Write memory files but skip the project git commit/pull/push path.
`sase init memory`	Compatibility alias for `sase memory init`.
`sase init sdd`	Alias for `sase sdd init`; enables version-controlled SDD and refreshes generated guides.
`sase init sdd --check`	Report SDD config and generated-file drift without writing files.
`sase skill`	Alias for `sase skill list`.
`sase skill list`	Inspect generated skill sources, provider targets, and deployed-file drift without writing.
`sase skill init`	Generate skill files; existing files require confirmation or `--force`.
`sase skill init --dry-run`	Preview generated skill target paths without writing files.
`sase skill init --force`	Generate and overwrite deployed skill files without confirmation.
`sase skill init -p <provider>`	Deploy only one provider's generated skill files.
`sase skill log`	Summarize or inspect audited generated skill-use events.
`sase skill use <name>`	Agent-side audit event recording that a generated skill was used.
`sase init skills`	Compatibility alias for `sase skill init`.

Advanced deploy controls such as `--no-commit`, `--no-push`, and `--no-apply` live on explicit subcommands rather than the bare coordinator. Scoped `--check` flags also live on explicit subcommands when you want to validate only memory or only SDD generated files.

Agent Markdown Documents

AMD is the initialization surface for agent markdown documents: root `AGENTS.md` plus provider shims such as `CLAUDE.md`, `GEMINI.md`, `QWEN.md`, and `OPENCODE.md`.

```
sase amd list
sase amd init --check
sase amd init
sase init amd --check
```

With no subcommand, `sase amd` defaults to `sase amd list`. The inventory shows project, subdirectory, home, and chezmoi-source `AGENTS.md` files, their H1 titles, whether they look AMD-managed, short/long memory reference counts, and nearby provider shim status.

`sase amd init` plans against the current directory unless `use_chezmoi: true` adds or redirects to the chezmoi home source root. It always creates or repairs provider shims for each selected root. Project-local provider shims contain `@AGENTS.md`; direct live-home provider shims contain an absolute import such as `@/home/bryan/AGENTS.md`. Chezmoi home source shims are managed as `CLAUDE.md.tmpl`, `GEMINI.md.tmpl`, `QWEN.md.tmpl`, and `OPENCODE.md.tmpl` files containing `@{{ .chezmoi.homeDir }}/AGENTS.md`, which render to absolute imports on each machine. When a selected ordinary root has a local `./sase.yml` with `amd_h1_title`, AMD writes a managed `AGENTS.md` with marker-delimited short-memory and long-memory sections. When the selected root is the live home root, a user config value from `~/.config/sase/sase.yml` or `~/.config/sase/sase_*.yml` can provide the home `AGENTS.md` title. When the current directory is the chezmoi home source root, AMD reads the source-side `dot_config/sase/sase.yml` and `dot_config/sase/sase_*.yml` files instead, so edited dotfile source config can drive source `AGENTS.md` generation before `chezmoi apply`.

With `use_chezmoi: true`, running `sase amd init` from the live home root initializes the chezmoi home source root instead of writing `~/AGENTS.md` directly. Running it from any other directory initializes that directory and the chezmoi home source root, deduplicating roots that resolve to the same path. A global or user-level `amd_h1_title` still does not opt ordinary roots into managed `AGENTS.md`; each ordinary root must set the title in its own `./sase.yml`.

When no applicable title is configured and `AGENTS.md` is missing, explicit AMD init can migrate exactly one custom provider instruction file into `AGENTS.md`; multiple custom provider files block so content is not guessed.

Bare `sase init` runs AMD before memory so memory validation can see the `AGENTS.md` that AMD would create. To avoid surprising existing repositories, bare `sase init` only lets AMD generate managed `AGENTS.md` when the current project's own `./sase.yml` opts in with `amd_h1_title`; global config values are ignored for this generator.

Memory Initialization

`sase memory init` initializes both project-local and home-level memory surfaces:

- Project memory under `./memory/`, including `memory/README.md` and flat note files with `type`/`parent` frontmatter.
- Home memory under `~/memory/`, or under `~/.local/share/chezmoi/home/` when `use_chezmoi: true`.
- A minimal `AGENTS.md` when one does not already exist and the repo is not opted into AMD-managed instructions.
- Provider shims `CLAUDE.md`, `GEMINI.md`, `QWEN.md`, and `OPENCODE.md`; project roots contain `@AGENTS.md`, while direct live-home roots contain absolute `@/path/to/home/AGENTS.md` imports. Chezmoi home source roots use `*.md.tmpl` provider shim sources containing `@{{ .chezmoi.homeDir }}/AGENTS.md`.

When the project-local `./sase.yml` sets `amd_h1_title`, `sase memory init` synchronizes the AMD-managed memory blocks inside `AGENTS.md`, adds missing canonical frontmatter to memory files, and renders the long-memory list from `description` values before reachability validation runs.

When `use_chezmoi: true`, the home files are written to the chezmoi source tree. The command can then commit those home changes and run `chezmoi apply --force`; `--no-commit` does not disable that home deployment path.

The generated `memory/sase.md` summarizes workspace naming and linked repositories. Project memory reads linked-repo descriptions from the project-local `./sase.yml`; home memory reads them from the global config `~/.config/sase/sase.yml`, or from the chezmoi-managed config path when `use_chezmoi: true`. Generated memory distinguishes static-path linked repos (`workspace.strategy: none`) from numbered-workspace linked repos, lists the direct path for static linked repos, and includes `sase workspace open` instructions only when at least one configured linked repo uses numbered workspace resolution.

Every configured `linked_repos` entry (or its deprecated `sibling_repos` alias) must have a non-empty `description`. Initialization fails instead of generating ambiguous memory when a description is missing.

By default, project memory initialization runs the configured precommit command, stages generated project files, commits them with the standard memory-init commit message, pulls with rebase, and pushes. Use `sase memory init --check` for a read-only drift check, or `sase memory init --no-commit` when you want to review generated project files before committing. `--no-commit` only skips the project deploy path; home memory deployment still follows `use_chezmoi` when it is enabled.

Memory validation is reachability-based: Markdown files under `memory/` must be reachable from `AGENTS.md` directly or through transitive `@memory/...` or `memory/...` references. Unreferenced memory files make the command fail so important agent context is not silently ignored.

Memory Context List

`sase memory list`, or bare `sase memory`, renders a read-only dashboard for the current directory. It reports:

- `loaded` files reached by transitive `@...` references from `AGENTS.md` in the project or home context.
- `referenced` files mentioned by plain `memory/...` text from loaded context or by audited `sase memory read` instructions. These are visible in the dashboard, but their contents are not loaded unless another `@...` edge reaches them.
- `available` files present under project or home `memory/` that the current launch context does not reach.
- `missing` referenced memory paths that do not exist.

The dashboard includes approximate local token estimates for loaded memory context.

For day-to-day read/write operations, including audited reads and reviewed long-term memory proposals, see Memory (https://sase.sh/memory/).

Memory Read Audit Log

`sase memory read <memory-relative-path> -r <reason>` is the audited path for agent-initiated long-term memory reads. The path is relative to `memory/`; the command allows `type: long` Markdown notes and rejects `type: short` notes because short-term memory is expected to arrive through instruction loading. The command strips one leading YAML frontmatter block from stdout and appends `## Children` when nested long notes exist, but the audit log records only metadata such as path, agent name, timestamp, cwd, byte count, and reason.

Every read must include a non-empty reason via `-r` or `--reason`. The command also requires agent attribution from `SASE_AGENT_NAME`, `SASE_AGENT`, or `SASE_ARTIFACTS_DIR/agent_meta.json`; unattributed reads fail instead of writing a log row. Human shell users normally inspect files directly and use `sase memory review` for promotion decisions.

`sase memory write` creates an attributable proposal under `~/.sase/projects/<project>/` and never writes canonical memory files directly. It uses the same agent-attribution rules as `read`; `--manual-author` is intended for tests and demos. Pass `--notify` when you want a best-effort `memory.proposed` notification in the SASE inbox. `sase memory review` is the human promotion path for listing, showing, approving, editing, or rejecting those proposals.

`sase memory log` reads the project-scoped audit log from SASE state under `~/.sase/projects/<project>/`, not from the repo. Use `--path` or `--agent` to drill down to matching read events, `--id <read-id>` to inspect one event, and `--json` for deterministic machine-readable output. Add `--include proposals` to include proposal and review ledger events alongside read-log summaries.

```
# read requires SASE agent identity; write requires agent identity unless --manual-author is used for demos
sase memory read generated_skills.md --reason "Need generated skill context"
sase memory write --title "Generated skills" --slug generated_skills --evidence chat:abc123 --body "Durable memory body"
--notify
sase memory review --list
sase memory log
sase memory log --include proposals
sase memory log --path generated_skills.md
sase memory log --id <read-id>
```

SDD Initialization

`sase init sdd` is an alias for `sase sdd init`. It creates or updates the project-local `sase.yml` with `sdd.version_controlled: true`, then creates or refreshes generated SDD guide files and `sdd/assets/sdd-directory-map.png` for either a project root or an SDD root:

```
sase init sdd
sase init sdd --check
sase init sdd --path ./sdd
```

Keep conceptual SDD documentation in docs/sdd.md (https://sase.sh/sdd/). The files generated by `sase init sdd` are intentionally short project-local guides and are safe to overwrite. Use `--check` to compare the project config and generated files without rewriting them.

Built-in bare-git projects run this same generated-file refresh automatically during first-use `#git:<project>` initialization, existing bare-repo registration, workspace materialization, and the first version-controlled SDD write. Manual `sase init sdd` is still useful when you want an explicit refresh or a drift check.

Skill Initialization

Generated skills start as xprompt sources marked with a `skill` frontmatter field. `sase skill list` is the read-only inventory: it shows loaded skill sources, the providers they target, and whether generated `SKILL.md` files are current, stale, or missing. Bare `sase skill` shows the same dashboard.

`sase skill init` renders those sources into provider-specific `SKILL.md` files. Sources include bundled skill xprompts and user/runtime xprompt catalog entries. By default, generated skill files include a first-step `sase skill use <name> --reason ...` directive so agent skill usage is attributable in the same project audit surface as memory reads; `sase skill log` summarizes and inspects those recorded skill-use rows. A source can set `log_skill_use: false` to omit that directive. The usual workflow is to inspect first, preview writes, then deploy:

```
sase skill list
sase skill init --dry-run
sase skill init --force
```

Without `use_chezmoi`, generated skill files are written directly under the provider's home-directory skill targets. When `use_chezmoi: true`, skill initialization writes through the chezmoi-managed home tree and can commit, push, and apply those dotfile changes. The `--no-commit`, `--no-push`, and `--no-apply` flags only affect that chezmoi deployment sequence. `sase init skills` still works as a compatibility alias for `sase skill init`.

See XPrompt Skill Field (https://sase.sh/xprompt/#skill-field) for the skill-source contract and bundled skill list.

Memory

SASE memory is durable context that survives individual agent chats. Notes live as Markdown files directly under `memory/`; each non-README note declares its tier in YAML frontmatter:

- **Short-term memory** uses `type: short`. It is instruction context, loaded only when `AGENTS.md` reaches it through `@memory/...` references. `sase memory init` creates that wiring for generated defaults and synchronizes AMD-managed `AGENTS.md` memory blocks when the project opts in with `amd_h1_title`.
- **Long-term memory** uses `type: long`. It is reference context, requires `description` frontmatter, and can set `parent: memory/<note>.md` to appear under another long note's `## Children` section.
- **Audited memory operations** live under the project state directory and record agent reads plus proposed writes and human review decisions.

Use initialization (https://sase.sh/init/#memory-initialization) to create or refresh the files. Use `sase amd list` (https://sase.sh/init/#agent-markdown-documents) to inspect `AGENTS.md` and provider shim status. Day to day, the usual order is: inspect loaded context with `sase memory list`, have agents use `sase memory read` for audited long-term reads, have agents use `sase memory write` only to create proposals, then have a human approve or reject those proposals with `sase memory review`.

Inspect Context

`sase memory` and `sase memory list` render the memory files visible from the current directory:

```
sase memory
sase memory list
```

The dashboard separates:

- `loaded` files reached by transitive `@...` references from `AGENTS.md` in the project or home context. Provider shims normally point at `AGENTS.md`; they are reported as instruction roots but are not separate traversal roots for this dashboard.
- `referenced` files mentioned by plain `memory/...` text or by audited `sase memory read` instructions, but not loaded.
- `available` files present under project or home `memory/` but unreachable from the current launch context.
- `missing` referenced files that do not exist.

Approximate token counts are included so large instruction surfaces are visible before an agent launch.

Audited Reads

Agents should read long-term memory through `sase memory read` so the access is attributable:

```
sase memory read generated_skills.md --reason "Need generated skill context"
sase memory log
sase memory log --include proposals
sase memory log --path generated_skills.md
sase memory log --agent agent-a
sase memory log --id <read-id>
```

The read path is relative to `memory/` and accepts long-term notes (`type: long`). Short-term notes are excluded because they are intended to arrive through instruction loading rather than ad hoc reads. The command strips one leading YAML frontmatter block from stdout and appends a `## Children` section when the note has nested long-term children. The audit event records metadata such as path, agent name, timestamp, cwd, byte count, and reason.

Every read requires a non-empty reason via `-r` or `--reason` and agent attribution from `SASE_AGENT_NAME`, `SASE_AGENT`, or `SASE_ARTIFACTS_DIR/agent_meta.json` (`name`, `workflow_name`, or `agent_name`). Unattributed reads fail instead of writing the log. In a normal human shell, use regular file reads instead of this audited command unless you are intentionally simulating an agent identity.

Pass `--include proposals` to include memory proposal and review ledger events in the same audit dashboard. Path and agent filters also apply to proposal target paths and proposal/review actors.

Propose Memory

Agents do not write canonical long-term memory files directly. They create proposals:

```
sase memory write \
  --title "Generated skills" \
  --slug generated_skills \
  --keyword "commit skill" \
  --evidence sdd/research/skills.md \
  --body "Durable memory body" \
  --notify

cat draft.md | sase memory write \
  --title "Generated skills" \
  --target generated_skills.md \
  --from-chat abc123 \
  --keyword "commit skill"
```

`sase memory write` is the agent-side authoring path. It writes proposal state only under `~/.sase/projects/<project>/`; it never modifies canonical memory files. A proposal needs:

- `--title`
- exactly one of `--slug <slug>` or `--target <slug>.md`
- at least one non-note evidence item
- body content from `--body`, `--file <path>`, `--file -`, or piped stdin when neither `--body` nor `--file` is supplied

Use `--file -` when a wrapper needs the explicit `--file` form but should still pass the body on stdin.

Targets must be one-level long-memory paths such as `generated_skills.md`; slugs must match `[a-z0-9][a-z0-9_-]*`. Evidence can be a path, `chat:<id>`, `--from-chat <id>`, `url:<url>`, a bare HTTP(S) URL, or a supplemental `note:<text>`. Note-only evidence is rejected.

Proposal bodies must be non-empty UTF-8 and at most 256 KiB. Bodies above 16 KiB produce a warning unless `--allow-large` is passed. Prompt-injection-like text is also recorded as a warning for the reviewer.

Proposal authors are attributed from the same agent identity sources as audited reads. `--manual-author` exists for tests and demos; normal agent writes should rely on the SASE-provided identity.

Use `--notify` to best-effort append a `memory.proposed` notification after proposal creation. The notification carries the `memory` tag, attaches any evidence paths that resolved to local files, and opens the interactive memory review TUI at that proposal when selected in ACE. The notification is only a prompt to review; it does not approve, reject, or edit the proposal by itself. Notification delivery is reported in the human output and as `notification_id` in JSON output.

Use `--json` for deterministic machine-readable output.

Review Proposals

Humans review proposals with `sase memory review`:

```
sase memory review                          # interactive TUI on a TTY
sase memory review --list
sase memory review --list --all --json
sase memory review <proposal-id> --show
sase memory review <proposal-id> --approve
sase memory review <proposal-id> --edit
sase memory review <proposal-id> --approve --edited-file edited.md
sase memory review <proposal-id> --reject --reason "Too speculative"
```

A bare `sase memory review` opens the Textual review app when stdin/stdout are TTYs. In non-interactive shells it prints the pending list instead. `--list` and `--show` are inspection commands; `--approve`, `--edit`, and `--reject` are the human promotion decisions. Proposal ids can be abbreviated when the prefix is unambiguous.

Agents cannot approve, edit-approve, or reject proposals: those actions fail when agent identity is present in `SASE_AGENT_NAME`, `SASE_AGENT`, or `SASE_ARTIFACTS_DIR/agent_meta.json`. Human review events record the local user and hostname. `--edit` opens `$VISUAL` or `$EDITOR`, then approves the edited body.

Approval writes the canonical file under the current repo's `memory/` path and prepends frontmatter:

```
---
type: long
parent: AGENTS.md
description: Generated skills
source_candidate: mem-20260523-142233-a1b2c3d4
keywords:
  - "commit skill"
---
```

Approval refuses to overwrite an existing target. Use `--target <slug>.md` to approve into a different unused one-level target, `--edit` to open `$VISUAL` / `$EDITOR` before approving, or `--edited-file` for non-interactive edited approval.

If approved memory should be loaded every time, add an explicit `@memory/<note>.md` reference from the appropriate instruction file.

Review TUI

The interactive review app shows pending proposals, evidence, target status, diffs against existing files, warnings, and audit events. Keybindings:

Key	Action
`j` / `k`	Move through pending proposals
`Down` / `Up`	Move through pending proposals
`g` / `G`	Jump to first / last proposal
`/`	Filter by id, title, author, target, keyword, status
`Enter` / `d`	Toggle detail view

`Esc`	Return from detail view
`a`	Approve as-is
`e`	Edit in `$VISUAL` / `$EDITOR`, then approve
`r`	Reject with a required reason
`t`	Override the approval target
`y`	Copy the proposal id
`q`	Quit

The proposal ledger is append-only JSONL with a lock companion. Malformed rows are skipped when reading, and every review action appends a new event rather than mutating previous events.

4 ACE TUI User Guide

4.1 Overview

ACE (Agentic ChangeSpec Explorer) is the primary TUI for the SASE toolkit. It provides an interactive interface for navigating, managing, and operating on ChangeSpecs, agents, and the Axe daemon.

4.2 Launching

```
sase ace [QUERY] [options]
```

If no query is provided, ACE loads the last used query, then the first saved query, then falls back to `!!!` for error suffixes.

4.2.1 CLI Options

Option	Description
`QUERY` (positional)	Query string for filtering ChangeSpecs
`-m`, `--model-tier`	Override model tier for all LLM providers (`large` or `small`)
`-M`, `--model-size`	Deprecated alias for `--model-tier` (`big` or `little`)
`-p`, `--profile [PATH]`	Profile the TUI session with pyinstrument; optional output path
`-r`, `--refresh-interval`	Auto-refresh interval in seconds (default: 10, 0 to disable)
`-x`, `--no-axe`	Disable auto-starting the axe daemon on startup
`-v`, `--vcs-provider`	Override VCS provider (`git`, `hg`, or `auto`)
`-R`, `--restart-axe`	Restart the axe daemon on startup (shows RESTARTING indicator)
`-t`, `--tab`	Tab to focus on startup (`changespecs`, `agents`, `axe`; default: `agents`)
`-T`, `--tmux`	Launch ACE in a new tmux window and print the target for external control

When profiling is enabled, ACE writes text output to `PATH` or `$SASE_TMPDIR/ace_profile_<ts>.txt`, prints the shortened path on exit, and copies that path when a clipboard tool is available.

4.2.2 Examples

```
sase ace                          # Last query, first saved query, or "!!!"
sase ace '"feature" AND "Drafted"'    # Filter by name and status
sase ace '+myproject'                 # Filter by project
sase ace -m small -r 30 '!!! OR @@@' # Small model, 30s refresh
```

When `--profile` is enabled, ACE prints a shortened profile-output path after the TUI exits and tries to copy that shortened path to the system clipboard (`pbcopy`, `wl-copy`, `xclip`, or `xsel` when available).

4.3 Tab System

ACE has three tabs, cycled with `Tab` and `Shift+Tab`:

Tab	Description
PRs (ChangeSpecs)	Browse and act on ChangeSpecs matching the current query
Agents	View running and completed agents, their files and prompts
Axe	Monitor the Axe daemon and background commands

4.4 Keybindings: PRs Tab

4.4.1 Navigation

Key	Action
`j` / `k`	Move to next / previous visible row (banner at fold `< L2`, PR at the leaf level)
`<` / `>` / `~`	Navigate to ancestor / child / sibling PR
`'`	Jump to entry by hint character (current tab); hints land on collapsed banners too
`Ctrl+O`	Fast jump: jump back if possible, otherwise jump to the first current-tab hint
`` ` ``	Jump to entry across all tabs (see Jump All Modal)
`Ctrl+R` / `Ctrl+K`	Jump back / forward in PR history
`o` / `O`	Cycle PR grouping mode forward / reverse (`BY_PROJECT` ↔ `BY_DATE` ↔ `BY_STATUS`)
`g` / `G`	Scroll detail panel to top / bottom
`Ctrl+D` / `Ctrl+U`	Scroll detail panel down / up (half page)

> **Note:** `o` / `O` ("organize") cycles the L0 grouping bucket forward / reverse on the Agents and PRs tabs (each tab keeps its own in-session mode). On the AXE tab it is a silent no-op. See PR Grouping and Folding and the Agents-tab Grouping Modes below.

4.4.2 PR Actions

Key	Action
`a`	Accept proposal (`!` = spec only, `@` = mark ready to mail)
`b`	Rebase PR onto parent
`C` / `c1` - `c9`	Checkout PR (primary / workspace 1-9)
`d`	Show diff
`e`	Edit spec file
`f`	Edit hooks (re-run / delete via hint input)
`M`	Mail PR
`m`	Mark / unmark current PR (auto-advances to next)
`n`	Rename PR (non-Sub/Rev PRs only)
`R`	Rewind to previous commit (`!` suffix skips VCS operations)
`s`	Change status (opens status modal)
`S`	Bulk status change for all marked PRs
`T`	Checkout + tmux (opens workspace input modal for number)
`u`	Clear all marks
`v`	View files (hint mode)
`w`	Reword PR description

`W`	Add tag to PR description
`Y`	Sync workspace

4.4.3 PR Grouping and Folding

The PRs tab is always grouped — the renderer walks one of `BY_PROJECT`, `BY_DATE`, or `BY_STATUS` and emits a banner row above each bucket. `BY_PROJECT` is the startup default; `o` cycles `BY_PROJECT → BY_DATE → BY_STATUS` for the current session.

Mode	L0 buckets	Notes
`BY_PROJECT`	Project name	Adds an L1 sibling-root sub-banner shared by `foobar_1` / `foobar_2` style suffixed siblings. Singletons suppress their L1 banner.
`BY_DATE`	`Today` / `Yesterday` / `This Week` / `Earlier`	Bucket from the latest TIMESTAMPS entry. Today/Yesterday add 4-hour L1 windows; hourly L2 headings appear only inside 4-hour windows with 2+ PRs. This Week adds day headings; Earlier adds week headings plus `(no timestamp)`.
`BY_STATUS`	`Mailed` / `Ready` / `WIP` / `Draft` / `Submitted` / `Reverted` / `Archived`	Bucket from the literal `status` field; actionable buckets first (`Mailed` = awaiting response, `Ready` = next to mail), terminal states last. Adds an L1 sibling-root sub-banner shared by `foobar_1` / `foobar_2` style suffixed siblings inside each status bucket. Singletons suppress their L1 banner.

In `BY_DATE` mode, PRs sort newest-first within each date bucket. `Today` and `Yesterday` are grouped first by compact 4-hour windows (`8AM-12PM`); one-hour headings (`09:00`) appear only when that 4-hour window contains at least two PRs. `This Week` uses calendar-day subgroups; `Earlier` uses Monday-start week ranges. PRs without a parseable TIMESTAMPS entry fall into `(no timestamp)` under `Earlier`.

The active grouping mode is shown in the PRs-tab info-panel header as a `[group: <label>]` badge.

Key	Action
`l`	Expand the focused banner one level (or peel one layer of the visible tree)
`h`	Collapse the focused banner; on a collapsed L1 banner, escalate to its parent. With agent focus, collapse the deepest enclosing group.
`L`	Snap to fully expanded — all banners and PR rows visible
`H`	Snap to fully collapsed — collapse every visible banner

Collapsed banner rows are first-class navigation stops: `j` / `k` step through them just like PR rows, and `'` jump-hints land on them too. After a fold change that hides the focused CL, focus snaps to the deepest collapsed ancestor banner so the cursor always sits on a row the user can see.

4.4.4 Fold Mode (`z` prefix)

Key	Action
`z c`	Cycle commits section (expand → collapse)
`z d`	Cycle deltas section (folded ↔ unfolded)

`z h`	Cycle hooks section (expand → collapse)
`z m`	Cycle mentors section (expand → collapse)
`z t`	Cycle timestamps section (expand → collapse)
`z C`	Toggle commits section (collapsed ↔ fully expanded)
`z D`	Toggle deltas section (folded ↔ unfolded)
`z H`	Toggle hooks section (collapsed ↔ fully expanded)
`z M`	Toggle mentors section (collapsed ↔ fully expanded)
`z T`	Toggle timestamps section (collapsed ↔ fully expanded)
`z z`	Cycle all sections
`z Z`	Toggle all sections (expand ↔ collapse)

COMMITS, HOOKS, MENTORS, and TIMESTAMPS sections each cycle through three fold levels:

Level	Behavior
Collapsed	Notes truncated to fit; multi-line body shown as `[+N lines]`; only latest drawers
Expanded	Full notes; body shown in dimmed text; all CHAT/DIFF/PLAN drawers visible
Fully Expanded	Everything visible including rejected proposals

The lowercase cycle keys (`z c`, `z h`, `z m`, `z t`) step through all three levels in order. The uppercase toggle keys (`z C`, `z H`, `z M`, `z T`) skip the intermediate **Expanded** state, jumping directly between **Collapsed** and **Fully Expanded**.

When collapsed, a `[folded: CHAT + DIFF + PLAN + N proposals]` indicator appears on COMMITS entries with hidden content. The indicator width is pre-calculated so that note truncation accounts for it. TIMESTAMPS shows a `[folded: N]` indicator inline with the header and displays the most recent timestamp entry when collapsed, giving a quick view of the last lifecycle event.

The DELTAS section uses two semantic states. When **folded**, the section renders a one-line file and line-count summary such as `DELTAS:  +3 (+428) ~6 (+91 ~37 -14) -1 (-22) (10 files)`. When **unfolded**, the alphabetical entry list is shown with colored glyphs (green `+`, gold `~`, red `-`) and inline line-count tokens. Binary files display `binary`; zero-count entries display `0 lines`. The section is omitted entirely when the ChangeSpec has no deltas.

4.4.5 Workflows and Agents

Key	Action
`r`	Run workflow on current PR
`@`	Run a custom agent (opens project/ChangeSpec selection)
`Space`	Run agent from current PR

If ACE cannot detect a workspace provider for the selected ChangeSpec or agent, the quick-launch actions show an error toast instead of opening a prompt with a broken VCS prefix.

4.4.6 Bang Mode (`!` prefix)

Key	Action
`!!`	Run background command (opens hook history modal)
`!x`	Start / stop axe (or select process)

4.4.7 Hook History Modal

Pressing `!!` opens the hook history modal showing previously run background commands:

Key	Action
`j` / `k`	Navigate through hook history
`Enter`	Select and execute highlighted hook
`Ctrl+D`	Delete highlighted hook from history
`Ctrl+G`	Edit first — select hook and open in input
`Esc` / `q`	Cancel and close modal

The modal supports live filtering as you type in the search box and displays last-used timestamps for each hook.

4.4.8 Leader Mode (`,` prefix)

Key	Action
`,,`	Repeat the last leader command
`,!`	Run command using current PR context
`,A`	Open the Agent Run Log modal for the current PR
`,c`	Clear COMMENTS field (kills CRS agents, deletes CRS proposals)
`,h`	Run agent from home prompt context; bare prompts default to `#git:home`
`,m`	Review mentors (opens Mentor Review modal)
`,M`	Kill running mentors
`,p`	Open project lifecycle management (see Project Management Panel)
`,o`	Open model overrides (see Model Overrides)
`,R`	Show runners info
`,t`	Open task queue modal (see Task Queue Modal)
`,<space>`	Run agent from current PR (skips project selection)
`,.`	Open prompt history modal
`,>`	Open prompt history modal with cancelled prompts visible

The `,h` shortcut opens a home-context prompt directly. Project and PR launch pickers use lifecycle-aware discovery: project entries, including `home` when it appears in picker lists, must have active and launchable ProjectSpecs; PR choices come from active ProjectSpecs. Inactive projects do not appear in normal launch pickers until they are reactivated with `sase project activate <project>`.

Project launch pickers also support `Ctrl+D` for cleanup of empty project entries. This deletes only the highlighted project's active/archive ProjectSpec files, refuses entries whose ProjectSpec files still contain ChangeSpecs, and does not delete workspace checkouts or other SASE state. For lifecycle changes, bulk operations, ProjectSpec editing, or deleting the whole SASE project directory, use the `,p` Project Management panel.

The repeat binding is the leader prefix followed by the configured `repeat_last` key. With the defaults both are comma, so the sequence is `,,`; if the leader prefix is changed but `repeat_last` is not, the second key remains comma. Repeat re-dispatches the last recognized leader subkey against the current tab and selection. If no leader command has been run yet, ACE shows a toast and does nothing.

> **Note:** `,x` (kill & edit) is only available on the Agents tab — see Agents Tab Leader Mode.

4.4.9 Mentor Review Modal

Press `,m` to open the Mentor Review modal, which lets you navigate mentor comments, accept or reject suggestions, and apply accepted changes. See docs/mentors.md (https://sase.sh/mentors/) for the full mentor system reference.

Key	Action
`j` / `k`	Navigate between mentors
`n` / `p`	Navigate between comments within a mentor
`N` / `P`	Navigate between accepted comments only
`Ctrl+D` / `Ctrl+U`	Scroll comment details down / up
`Space`	Toggle acceptance of the current comment
`Enter`	Apply all accepted comments (launches agent)
`a`	Apply accepted comments and propose (amend with propose)
`A`	Apply accepted comments and commit
`r`	Run a mentor profile (opens profile picker)
`y`	Copy the current comment to the clipboard
`Shift+K`	Kill a running mentor
`Esc` / `q`	Close modal

4.4.10 Copy Mode (`%` prefix)

Key	Action
`%%`	Copy ChangeSpec

`%!`	Copy ChangeSpec + snapshot
`%b`	Copy bug number
`%c`	Copy PR number
`%n`	Copy PR name
`%p`	Copy project spec file
`%s`	Copy sase ace snapshot

4.5 Keybindings: Agents Tab

4.5.1 Navigation

Key	Action
`j` / `k`	Move to next / previous visible row (banner at fold `< L3`, agent at `L3`)
`J` / `K`	Cycle focus across tag side panels (forward / reverse)
`'`	Jump to entry by hint character (current tab); on the Agents tab, hints land on collapsed banners too
`Ctrl+O`	Fast jump: jump back if possible, otherwise jump to the first current-tab hint
`` ` ``	Jump to entry across all tabs (see Jump All Modal)
`o` / `O`	Cycle grouping mode forward / reverse (`STANDARD` ↔ `BY_DATE` ↔ `BY_STATUS`)
`g`	Scroll to top (file, tools, or metadata panel)
`G`	Scroll to bottom (file, tools, or metadata panel)
`Ctrl+D` / `Ctrl+U`	Scroll file panel down / up
`Ctrl+F` / `Ctrl+B`	Scroll prompt panel down / up

> **Note:** `o` / `O` ("organize") cycles the grouping mode forward / reverse on the Agents and PRs tabs (each tab keeps its own in-session selection independently); on the AXE tab it is a silent no-op. `g` / `G` keep their conventional vim-style scroll-to-top/bottom meaning on every tab. See Grouping Modes below.

4.5.2 Agent Actions

Key	Action
`R`	Revive a previously dismissed agent
`A`	Open completion artifacts for the focused agent; in tmux, press again to close the viewer pane
`@`	Run custom agent
`a`	Cycle auto-approve state / answer HITL
`f`	Fork agent (by name if running, by chat file if completed)
`n`	Name agent
`r`	Edit prompt and relaunch agent (retry without killing)
`v`	View files (hint mode)

`V`	Open the Agent Run Log modal for the focused agent
`w`	Wait/unwait agent (opens WaitModal — see below)
`W`	Wait for agent (populate prompt with `%w`); with marks, fans out to `%w:a,b,c`
`m`	Mark / unmark current agent (auto-advances to next)
`s`	Save and dismiss marked agents as a revivable group (opens optional group-name modal)
`U`	Toggle the focused agent's unread marker
`u`	Clear all agent marks
`x`	Kill / dismiss agent, every marked agent, or every agent in the focused group
`X`	Open the cleanup panel for panel, global, tag, marked, group, or custom cleanup
`Enter` / `L`	Jump to PR (for agents with `meta_new_cl` / `meta_new_pr`)
`e`	Edit chat in editor; with marks, open all editable marked transcripts in one editor invocation
`E`	Edit panel content in editor
`t`	Open the focused agent's tmux target; agents with opened linked-workspace context show a workspace chooser
`T`	Open tmux window in the agent's primary project workspace
`N`	Open the agent tag/untag modal (input is pre-seeded with `pinned` for untagged agents; submit empty to clear)
`]` / `[`	Cycle panels: file → tools → metadata (forward / reverse)
`p`	Toggle file / prompt layout
`Ctrl+N` / `Ctrl+P`	Next / previous file in panel
`-`	Reset file trim to default
`=`	Show all file lines

ACE separates fast visible-inbox loads from full-history scans. The visible inbox is the normal Agents-tab working set: active rows plus recent completed, non-hidden rows. Startup, manual refresh (`y`), and active agent search use that path through the persistent artifact index when it is available.

If the index is missing or unhealthy, ACE falls back to a bounded source-artifact scan for the first paint and shows a repair warning with the reason. That repair state can arm a deferred full-history reconcile after the TUI is idle, but normal `y` refreshes still stay on the visible-inbox path. Use `sase agent index status --json` for a lightweight check that does not scan source artifacts, `sase agent index verify` to compare the index with source artifacts, and `sase agent index gc` to rebuild the index and dismissed projection. Use the Agents-tab leader command `,y` when you want an immediate full-history refresh from source artifacts.

The dismissed projection that hides agents from the visible inbox is rebuilt from the in-memory dismissed set *unioned with every dismissed-bundle summary*. Reviving an agent now purges its dismissed bundle, so a revived agent stays visible. For archives that accumulated stale bundles before that fix, plain `sase agent index gc` is **not** a repair on its own -- it rebuilds the projection *from* those lingering bundles and re-hides the revived agents. Run `sase agent index gc --purge-revived-bundles` (`-r`) to first delete dismissed-bundle files and summary rows for suffixes that are no longer present in `dismissed_agents.json`, then rebuild the corrected projection.

When one or more agents are marked, `e` edits the marked set instead of only the focused row. ACE opens editable completed transcripts in visible row order, deduplicates repeated paths, skips live marked rows that are still running or have no chat file, and reports that live skip count. Stale marks are ignored for this action, and marks remain in

place after the editor exits.

4.5.3 Linked Workspace Context

Configured `linked_repos` are recorded in agent metadata at launch time. For non-terminal agents, ACE can include dirty, existing suffix-strategy linked repo workspaces in the agent detail `DELTAS` section. The folded summary counts primary and linked-repo changes together; the unfolded view groups linked entries under the workspace glyph and linked-repo name, and file hints resolve paths relative to the linked workspace directory. Static linked repos (`workspace.strategy: none`), missing workspace directories, clean repos, and completed/failed agents are not part of this live linked-delta display.

When a SASE-launched agent opens a configured linked repo with `sase workspace open -p <linked_repo> -r "<reason>" <workspace_num>`, the run records an opened-workspace marker. Here `-p/--project` is the workspace CLI's project selector; configured linked repos are backed by hidden `PROJECT_STATE: sibling` project records so their names can be used there. ACE shows recorded markers in the prompt/detail `SASE CONTEXT` section as a `WORKSPACES` lane with the repo name, resolved path, open time, and reason. If the selected agent has cached opened-linked-workspace context, pressing `t` opens a keyboard-first tmux chooser with `CURRENT` first and one `LINKED` target per unique opened repository workspace. Selecting `CURRENT` uses the normal agent-workspace tmux path; selecting a `LINKED` row opens or switches to a tmux window named after the linked workspace directory. If no opened-workspace context is cached, `t` opens the normal agent tmux target directly. `T` always opens the primary project workspace.

4.5.4 Wait Modal

Press `w` on the Agents tab to open the WaitModal. Behavior depends on the agent's status:

- **WAITING agent**: Enter another agent's name to wait for, or leave empty and press Enter to run immediately (unwait).
- **RUNNING agent**: Enter an agent name to kill the current agent and restart it with a `%w` (wait) directive. This is useful for redirecting an agent to wait on a different dependency.

The modal supports readline-style keybindings (`Ctrl+F` / `Ctrl+B` / `Ctrl+A` / `Ctrl+E`) for cursor movement.

4.5.5 VCS Tag Resolution in Fork/Wait

When forking or waiting on an agent, VCS tags in the prompt (e.g., `#git(ref)`, `#gh:ref`) are automatically updated to point to the correct branch. For non-project agents, the ref is replaced with the agent's CL name (branch). For project agents using `#pr`, the ref is replaced with `@<name>` which resolves to the agent's branch. HITL suffixes (`!!`, `??`) are stripped during replacement since fork scenarios should not carry over HITL overrides.

4.5.6 Workflow Visibility

Workflows launched via `sase run` are visible in the Agents tab alongside ACE-launched workflows. The TUI scans `artifacts/run/*` directories in addition to `workflow-*` and `ace-run` directories, and writes an initial `workflow_state.json` before execution so that step data appears immediately rather than showing a bare

RUNNING entry. Anonymous `tmp_*` workflows are included in the normal visible-inbox index when their workflow state has `appears_as_agent: true` and does not set `hidden: true`; explicitly hidden workflow rows are omitted from the default view. Specialized review runners launched by axe (mentor, CRS, fix-hook, and summarize-hook review agents) are also visible and are automatically grouped under the `@review` tag, matching the behavior of a `%group:review` prompt launch.

4.5.7 Agent Artifacts

Press `A` on a focused agent to open the artifact panel whenever artifacts are associated with that agent. The list can include chat transcripts, plan files, generated Markdown PDFs, generated images, prompt-referenced images from saved prompt artifacts, and explicit files saved with `sase artifact create -p <path> [-n <label>] [-k <kind>]`. ACE always opens the panel, even for a single artifact, so the label, kind, and path are visible before launching the terminal viewer.

The prompt/detail header includes an `ARTIFACTS` section for non-chat entries from the same list. Paths are made workspace-relative when possible, and hint mode assigns numbers to those paths so they can be opened with the normal file-hint flow.

Artifact panel controls:

Key	Action
selector	Open the artifact with that one-key selector (`1-0`, then letters)
`j` / `k`	Move through artifact rows
`m`	Mark / unmark the highlighted artifact and advance to the next row
`y`	Copy highlighted Markdown artifact contents
`Y`	Copy the highlighted artifact path, workspace-relative when possible
`Enter`	Open marked artifacts in list order, or the highlighted row if unmarked
`A`	Open all artifacts in list order, ignoring marks
`q` / `Esc`	Close the panel

When ACE is running inside tmux, artifact viewing opens in a right-side tmux pane so the TUI remains visible. The Agents list collapses while the tracked pane is live, row-changing navigation shows a warning instead of moving to a different agent, `l` focuses the tracked pane, and `A` closes it. If the pane was already closed, `A` opens the artifact panel normally. Outside tmux, ACE suspends while the terminal viewer runs in the current pane. The viewer supports image, Markdown, and PDF artifacts: images are displayed directly with `kitten icat`; Markdown is first rendered to PDF; PDFs are converted to PNG pages for paging. The viewer needs `kitten` for display, `pdftoppm` for PDF/Markdown paging, and `pandoc` plus a supported PDF engine for Markdown rendering. Missing tools produce a warning instead of failing the TUI.

Viewer controls:

Key	Action
`j`	Next page; wraps from the last page to the first
`k`	Previous page; wraps from the first page to last

`n`	Next artifact when viewing an artifact sequence
`p`	Previous artifact when viewing an artifact sequence
`r`	Refresh the current page
`q`	Close the viewer

Only one plan artifact is shown for an agent. When both an archived plan and an SDD tale path are present, ACE prefers the committed SDD plan; otherwise it keeps the path that best matches the run metadata.

During successful-agent finalization, Markdown-to-PDF rendering updates `workflow_state.json.pdf_status` and a compact activity label. ACE renders that label on the agent row and in the prompt/detail header, so long conversions show progress such as `PDF 2/4 <path>` or `PDFs done 3/4 (1 skipped)` instead of looking idle.

4.5.8 Tag Side Panels

The Agents tab is laid out as a series of vertically-stacked side panels, one per agent **tag**. Untagged agents live in their own `(untagged)` panel; each tagged group renders as `#<tag>` with an agent count in the panel title. Each panel title can also show compact scoped metrics in the form `[H1 R2 W1 F1 U1 D3]`: `H` is human-in-the-loop, `R` is running, `W` is waiting, `F` is failed, `U` is unread terminal work, and `D` is done/read terminal work. Zero-count metrics are omitted. Panel heights are sized to their content and separated by a one-row gap. When the panels fit, the first panel grows to absorb leftover vertical space while later panels stay pinned to their natural height; when the panels overflow, space is weighted by each panel's rendered row count.

Use `J` / `K` to move focus across panels (forward / reverse). `J` lands on the first selectable row in the new panel; `K` lands on the last selectable row, including collapsed group banners when those are visible. Per-panel actions (kill, dismiss, expand, etc.) operate on whichever panel currently holds focus. Press `X` to open the cleanup panel: `d` dismisses completed agents in the focused panel, `D` dismisses completed agents across loaded panels, `k` cleans the focused panel, `K` cleans all loaded panels, `m` cleans marked agents, `g` cleans the focused group, `t` chooses a tag, and `c` opens the custom selector.

Tags are set or cleared with `N` (see Agent Actions). When opening the modal on an untagged agent the input is pre-seeded with `pinned` so a single Enter promotes the agent into the standard "pinned" panel; that default makes tag removal discoverable too — opening the modal on a tagged agent and submitting an empty string clears the tag. The `%group <name>` directive in a prompt assigns the tag at launch.

4.5.9 Group Banners and Folding

Within each tag side panel, agents are grouped into either a 2-level or 3-level banner hierarchy depending on whether any agent in the panel targets a ChangeSpec:

- **3-level layout** (panel contains at least one ChangeSpec-scoped agent): **project** → **ChangeSpec** → **name-root**. Project-scoped agents and agents with no `cl_name` fall into a synthetic `(no ChangeSpec)` bucket that sorts last.
- **2-level layout** (no ChangeSpec anywhere in the panel): **project** → **name-root**.

Banners are rendered between agent rows and carry a summary chip (`N agents · K running · M failed`). Workflow children inherit grouping identity from their parent agent so banners never appear between a parent and its workflow steps.

A single global fold level controls how much of the hierarchy is visible:

Level	What's visible (3-level layout)	What's visible (2-level layout)
`L0`	Project banners only	Project banners only
`L1`	Project + ChangeSpec banners	Project + name-root banners
`L2`	Project + ChangeSpec + name-root banners	All banners and agent rows
`L3`	All banners and agent rows (and per-workflow folds apply)	(same as `L2`)

Key	Action
`l`	Step the focused group's fold one level up (`L0` → `L1` → `L2` → `L3`); at `L3`, expand the focused workflow
`h`	Collapse the focused workflow; once it's collapsed (or no workflow is focused), step the group fold one level down
`L`	Snap to fully expanded — every banner, every agent row, every workflow step visible
`H`	Snap to fully collapsed — every per-workflow fold collapsed, then group fold to `L0` (only top-level banners)

Banners at fold levels `< 3` are selectable rows. When a banner is focused, `x` performs a bulk kill/dismiss on every top-level agent in that group (single confirmation modal). Marks take priority over the group, so a non-empty mark set always drives the bulk action regardless of banner focus. When a fold change hides the previously focused agent, focus snaps to the nearest visible ancestor banner so navigation context is never lost.

Visual treatment: every row carries a fixed-width tier-guide gutter built from one `│` segment per ancestor L0/L1 banner (in the parent tier's dim accent — project blue or ChangeSpec cooler accent), so nesting reads as a tree at a glance. L0 project / bucket banners use a sky-blue `▌` left bar and a heavy `━` rule; level-2 visual headings (ChangeSpec banners and `BY_DATE` 4-hour windows) get a cooler accent with a `┃` bar and a lighter `─` rule. Level-3 visual headings (name-root banners and conditional `BY_DATE` hourly windows) use a `▸` branch glyph with a teal label. Singleton name-root groups suppress their banner entirely to reduce visual noise.

The currently-focused side-panel row is marked with a thick accent-colored left bar, **bold** text, and a translucent accent tint applied to the row background. The tint is intentionally light so per-token status colors (running cyan, failed red, waiting yellow, etc.) remain readable through the highlight — the bar and bold weight do most of the work of marking the selection.

After a kill or dismiss, focus re-anchors on the visually-next row (rather than the next row in input order) so the selection always lands somewhere meaningful in the rendered tree.

4.5.10 Grouping Modes

Press `o` on the Agents tab to cycle the L0 grouping bucket through three modes. The Agents tab shows a brief toast (`Grouping: by project` / `by date` / `by status`) on each cycle:

Mode	L0 buckets	Notes

`STANDARD`	Project (with optional ChangeSpec sub-level)	The "by project" default. Uses the 2-/3-level layout described above.
`BY_DATE`	`Today` / `Yesterday` / `This Week` / `Earlier`	Date bucket at L0, then a date-aware L1 subgroup. Sorted newest-first within each bucket.
`BY_STATUS`	`Stopped` / `Failed` / `Running` / `Waiting` / `Done` / `Starting`	Bucketed by shared status semantics; name-root and name-prefix subgroups appear only when useful.

In `BY_DATE` mode, ACE chooses one L1 subgroup style from the L0 date bucket: one-hour windows (`09:00`) for `Today` and `Yesterday`, calendar-day labels for `This Week`, and Monday-start week ranges for `Earlier`. The time anchor is `stop_time` for terminal agents and `start_time` otherwise; both buckets and their subgroups sort newest-first. Workflow children inherit the parent's anchor so they stay adjacent regardless of their own start time, and agents with no usable timestamp fall into a `(no time)` subgroup that sorts last.

In `BY_STATUS` mode the L0 banner is the status bucket and L1 is the name-root, with the same singleton-suppression rule as `STANDARD`. The bucket order is fixed: Stopped, Failed, Running, Waiting, Done, Starting. The `Starting` bucket sorts last so startup-only rows do not displace active work during daemon or launch refreshes. Each mode keeps its own per-group fold registry, so collapsing buckets in `BY_STATUS` doesn't affect the project layout you had in `STANDARD`. `BY_STATUS` banners are prefixed with semantic glyphs (`▲`, `✗`, `▶`, `⏳`, `✓`, `◐`) so the bucket title still leads visually.

The active grouping strategy is also surfaced in the Agents tab header via a `[group: <label> (o)]` badge so the current session mode is always visible after the cycle toast fades. The same header starts with a visible top-level agent metric strip in the form `N Agents [S stopped · R running · W waiting · F failed · U unread · D done]`, with numeric counts in place of the letters and zero-count metrics omitted. `stopped` counts agents paused for plan approval, questions, or workflow human-input steps; `running` excludes waiting, failed, and stopped rows; `waiting` is the blocked/queued subset; `failed` is terminal failed work; `unread` counts terminal rows that still need acknowledgement; and `done` is completed visible work that has already been acknowledged. During startup the metric strip renders `Agents: …` until the first agent scan has loaded, avoiding a misleading zero-agent count. Each TUI launch starts in by-project grouping; cycling only changes the current session. **Waiting** holds agents that are blocked but progressing on their own — `WAITING` with a `wait_until` timer (`%wait:5m`, `%wait:1430`) or a non-empty `waiting_for` dependency. **Stopped** keeps the strict "you need to act" semantics: a `WAITING` agent with neither a timer nor a dependency stays there because it's parked waiting on the user.

4.5.11 Agent Row Glyphs

To keep rows compact, agent statuses and types are rendered as one- or two-character badges instead of verbose text:

Glyph	Meaning
`▶`	RUNNING
`✓`	DONE
`✓P`	PLAN DONE
`▶P`	PLAN APPROVED
`★E`	EPIC CREATED

✎	PLAN
✗	FAILED
⏳	WAITING
?	QUESTION
↻	RETRYING (followed by attempt count, e.g. `↻2`)
≡	Workflow row (top-level)
❑	ChangeSpec / CL row (top-level)
⚡	Autonomous (`%approve`) agent
◌	Hidden agent (visible only when `.` toggles them in)

Agents launched by `sase bead work` also show a gold `◆ <bead_id>` badge between the status glyph and the tag/name. A phase agent named `<epic_id>.<N>` displays that phase bead ID; the final `<epic_id>` land agent displays the parent epic bead ID. Legacy `<epic_id>.land` land agents keep the same badge. Dismissed agents keep the badge by stripping only the date-prefix used for dismissal.

Each agent row also carries a per-provider emoji badge before the display name so the LLM provider behind a row is readable at a glance without scanning the right-hand model suffix:

Badge	Provider
🎭	Claude
🪐	Antigravity (agy)
🤖	Codex
🐼	Qwen
🐙	OpenCode

The same provider palette also colors the `<PROVIDER>(<model>)` suffix on the right edge of the row — the provider name, the parentheses, and the model name each render in a distinct shade from that provider's palette so multi-model fan-outs are easy to scan. Providers without a dedicated palette (anything outside the table above) fall back to a neutral purple palette and render no emoji badge.

Workflow child rows for `python` and `bash` steps render a leading 🐍 / 🐚 glyph after the `N/M` step number, styled with the matching step-type accent. The glyph is a stronger signal than the step-type color alone for colorblind users and for rapid scanning. Agent, parallel, and `prompt_part` step rows are left unchanged — agent rows already carry a meaningful display name, parallel rows fan out into structural children, and `prompt_part` rows are invisible by default.

The right-hand edge of each row carries a runtime suffix (`<start-timestamp> · <elapsed>`) right-aligned within the panel. Active rows that have actually started include a 🏃 marker before the ticking elapsed duration; unread completed rows use a ✅ marker in the same suffix slot, or ❌ when the agent finished in a `FAILED` state; and user-paused rows (`PLAN`, `QUESTION`, `WAITING INPUT`) use a ✋ marker while waiting for a human response. Pre-run `WAITING` rows with no `BEGIN` time hide the suffix so queued waits do not look like live runtime. For finished agents, the start-timestamp half is rendered as a humanized `(date_prefix, time)` pair sized to fit the existing 15-cell slot:

- **Same day**: `HH:MM:SS`
- **Prior day, same year**: `Mon DD HH:MM` (drops seconds — they're noise once a row finished hours ago)
- **Different year**: `Mon DD 'YY` (date only)

The elapsed duration starts at `BEGIN` when a row recorded wait-before-run metadata, otherwise at the row start time. Completed `DONE` / `PLAN DONE` / `TALE DONE` workflow rows use the terminal agent stop time when one exists; plan-step rows that finish without a subprocess stop time anchor to the latest recorded plan submission time so completed planning rows do not keep ticking. `PLAN APPROVED` rows with a running follow-up show active elapsed time for the planner segment plus the coder segment, excluding the idle approval gap between plan submission and code launch. The date prefix uses a softer `dim #8787AF` while the time half keeps the standard `#8787AF`, giving the column internal hierarchy without inflating the palette. Statuses not in the table fall back to `(STATUS)` text for forwards compatibility.

4.5.12 Agent Search

Press `/` on the Agents tab to open the query editor. The query language is a **structured Boolean expression** — parallel to the ChangeSpec query language but with a property-key allowlist tailored to agents. Bare words are substring-matched against an agent's `cl_name`, `display_name`, `agent_name`, and `status`, plus its **xprompt, live reply/response, chat transcript, and prior attempt replies**.

Property keys (closed allowlist):

Key	Form	Notes
`status`, `cl`, `project`, `name`, `model`, `provider`, `tag`, `text`, `type`, `source`	`key:value` (substring match)	`source` is `axe` (workflow / step) or `manual`.
`pinned`, `hidden`, `attention`, `needs`	`key:true` / `key:false`	Boolean keys.
`age`	`age<5m`, `age>=2h`, `age:1d`, etc.	`:` is sugar for `>=`. Suffixes: `s` / `m` / `h` / `d`.

Boolean operators: juxtaposition is implicit `AND`; explicit `AND`, `OR`, and `NOT` (with parentheses) are honored. Precedence is `NOT > AND > OR`. The help modal carries an **Agent Query Syntax** section listing the same grammar.

Parse failures are non-fatal: the loader falls back to "no filter" for that render and surfaces a transient toast; the query-edit modal re-validates on Apply, keeping itself open and rendering the error inline (in red) on failure.

Transcript files are read lazily (only while a query is active) and cached by `(path, mtime_ns)` so auto-refresh stays cheap. Per-file reads are capped at 512 KB; missing or unreadable files are skipped silently. Parsed ASTs are also cached by raw query string so re-renders skip the parse.

4.5.13 Leader Mode (`,` prefix)

On the Agents tab, leader mode exposes layout and notification shortcuts for the currently loaded agent list. Unread-completed actions operate on terminal rows that are loaded in the tab; `,j` only targets visible rows.

Key	Action

`,,`	Repeat the last leader command
`,h`	Run agent from home prompt context; bare prompts default to `#git:home`
`,I`	Toggle manual idle (shows IDLE indicator; any keypress re-activates)
`,g`	Toggle between tag-split panels and one merged agent panel
`,j`	Jump to the next visible unread completed agent, newest first, and mark it read
`,J`	Jump to the next visible stopped/terminal agent, newest first, without changing unread state
`,y`	Refresh the Agents tab from full artifact history
`,u`	Mark all loaded unread completed agents as read
`,n`	Jump to agent notification (plan or question; auto-unhides if needed)
`,p`	Open project lifecycle management (see Project Management Panel)
`,o`	Open model overrides (see Model Overrides)
`,C`	Capture an Agents-tab reproduction bundle for debugging row disappearance or duplication
`,T`	Toggle continuous Agents-tab repro invariant checks and auto-capture on violation
`,x`	Kill focused or marked agent(s) and edit their prompt(s)
`,<space>`	Run agent from current agent's PR (skips selection)
`,.`	Open prompt history modal
`,>`	Open prompt history modal with cancelled prompts visible

Here, "stopped" means a dismissable terminal row such as `DONE`, `FAILED`, `PLAN DONE`, `TALE DONE`, `PLAN REJECTED`, `PLAN COMMITTED`, or `EPIC CREATED`; it is separate from the Agents header's "stopped" attention bucket for rows paused on user action.

If any agents are marked, `,x` acts on that marked set instead of the focused row. Stale marks are ignored; if any remaining marked agent has no recoverable prompt, ACE warns and leaves the set untouched. After confirmation, ACE kills or dismisses the marked agents and opens a prompt stack with one editable pane per original prompt in mark order. Embedded `---` inside an individual agent prompt stays inside that agent's pane.

4.5.14 Agents Tab Reproduction Bundles

Agents-tab reproduction bundles capture the loader/apply sequence that determines which rows are visible. Use them when the Agents tab briefly drops historical rows, re-adds them, or shows duplicate workflow parents.

When you see one of these bugs in a live ACE session, switch to the Agents tab and press `,C` before refreshing again. ACE writes a commit-safe bundle to `~/.sase/repros/<timestamp>-manual-.../agents_tab_repro.json` and shows a toast with the path. "Commit-safe" means local names and paths are redacted, and prompt, response, chat, and diff bodies are omitted. The bundle keeps the row identities, loader state, app projection state, screen text, and an SVG screenshot needed to replay the row-list behavior.

Replay a bundle from a checkout of this repository:

```
sase repro replay tests/ace/tui/repro/fixtures/agents_tab_disappear_reappear_v1.json --assert-stable --json
```

The current expected verdict for the checked-in fixture is:

```
{
  "result": "passed",
  "failed_invariants": [],
  "verdict": "current code fixed for the captured Agents-tab bug class"
}
```

Add `--write-artifacts /tmp/sase-agents-tab-repro-artifacts` to write one `.txt` screen dump and one `.svg` screenshot per replay step. The replay JSON lists those paths in `screen_paths` and `screenshot_paths`.

Use out-of-band capture only when you need a filesystem baseline and did not have the live TUI capture running:

```
sase repro capture agents-tab --output /tmp/sase-agents-tab-capture --commit-safe --json
```

Out-of-band capture is labeled `capture_mode=out_of_band` because it loads the current filesystem state and cannot reconstruct transient refreshes that already passed through the running TUI. The replay harness is scoped to the known Agents-tab disappearance/reappearance and duplicate-parent bug class; it is not a general proof for arbitrary rendering races.

For continuous diagnosis, press `,T` on the Agents tab to enable invariant checks after each load/apply cycle. On the first violation in a burst, ACE auto-captures one bundle under `~/.sase/repros/<timestamp>-auto-.../` and shows a warning toast. It does not write a new bundle every refresh while the same violation remains active.

4.5.15 Bang Mode (`!` prefix)

Key	Action
`!!`	Run background command
`!x`	Start / stop axe (or select process)

4.5.16 Copy Mode (`%` prefix)

Key	Action
`%c`	Copy chat file path
`%E`	Copy file path
`%n`	Copy the focused agent's `agent_name` (falls back to `display_name`; toast indicates which)
`%p`	Copy agent prompt
`%s`	Copy sase ace snapshot

4.6 Keybindings: Axe Tab

4.6.1 Sidebar Row Taxonomy

The Axe sidebar renders three row types so the operational tree reads at a glance:

- **Lumberjack** rows are top-level sections with a solid left accent bar (`▌`) in the lumberjack hue, a `[*]` / `[!]` / `[·]` running/error/idle marker, the lumberjack name, and an optional compact `Nc / Ne` cycles/errors chip at the end.
- **Chop** rows are child rows indented under their parent with a `└─` tree connector, a per-run status icon (`✓` success, `!` failure/timeout, `?` missing script, `●` running, `*` agent-launched, `·` no runs), and the chop name in a dim-gold child hue.
- **Background command** rows (run via `!!`) live below the lumberjack tree, separated by a dim divider line when both groups are present, and use a distinct command/slot badge so they cannot be mistaken for scheduled AXE work.

4.6.2 Dynamic Sidebar Width and No-Wrap Rows

Every sidebar row is rendered as single-line Rich `Text` with `no_wrap=True` and `overflow="ellipsis"`. After each refresh the widget computes the widest formatted row and emits a `WidthChanged` message; the AXE container resizes between a 35-cell minimum and an 80-cell maximum, clamped further so the right-hand dashboard always keeps at least 40 cells. On terminals too narrow to fit a label even at the clamped width, the row ellipsizes rather than wrapping onto a second line.

4.6.3 Controlled-Output Highlighting and ANSI Fallback

Output in the dashboard right panel uses a semantic highlighter for sources whose shape is controlled by sase, and falls back to ANSI rendering for everything else:

- **Lumberjack aggregate logs** (`[YYYY-MM-DD HH:MM:SS] [lumberjack] message`) get timestamp, lumberjack name, status words (`success`, `failure`, `timeout`, `running`, `error`, …), PIDs, durations, exit codes, and counts colored by severity and consistent with the sidebar taxonomy.
- **Controlled chop output** — the agent-launch line `Launched agent chop '<name>' (PID <pid>)` emitted for `agent_launched` runs — is highlighted with the chop name and PID coherent with sidebar colors.
- **External chop scripts** and **background command output** are arbitrary text and stay on the ANSI fallback (`Text.from_ansi`) with the existing capping and tail-biased caching behavior.

Render cache slots are keyed on `(source_id, source_type)` so the semantic and ANSI paths cannot collide for the same numerical identity.

4.6.4 Navigation

Key	Action
`j` / `k`	Move to next / previous sidebar row (lumberjack, chop, or background command)
`Ctrl+N` / `Ctrl+P`	Page through the focused chop's run history (newer / older)
`g`	Scroll to top
`G`	Scroll to bottom (pins auto-scroll)

4.6.5 Commands

Key	Action
`@`	Run agent
`r`	Run selected chop manually, or re-run the focused completed background command (`!!`) row
`x`	Start / stop axe (or kill the focused background command)
`X`	Clear output

4.6.6 Leader Mode (`,` prefix)

Key	Action
`,,`	Repeat the last leader command
`,h`	Run agent from home prompt context; bare prompts default to `#git:home`
`,p`	Open project lifecycle management (see Project Management Panel)
`,o`	Open model overrides (see Model Overrides)
`,R`	Show runners info

4.6.7 Bang Mode (`!` prefix)

Key	Action
`!!`	Run background command
`!x`	Start / stop axe (or select process)

4.6.8 Copy Mode (`%` prefix)

Key	Action
`%o`	Copy visible output
`%O`	Copy full output
`%s`	Copy sase ace snapshot

4.6.9 Axe Control

Key	Action
`Q`	Stop axe and quit

4.7 Query System

4.7.1 Editing Queries

Press `/` to open the query editor. The current canonical query is pre-filled.

To save a query, prefix with `#`:

- `#3 "myproject"` -- save to slot 3
- `# "myproject"` -- save to next available slot
- `#3` (no query) -- delete slot 3

4.7.2 Saved Queries

Press `1`-`9` or `0` to instantly load a saved query. These also work from within the help modal (`?`).

4.7.3 Query History

Key	Action
`^`	Navigate to previous query in history
`_`	Navigate to next query in history

Query history is available on the PRs tab and tracks queries as you switch between them.

See `docs/query_language.md` (https://sase.sh/query_language/) for the full query syntax reference, including boolean expressions, status shorthands, property filters, and searchable fields.

4.8 Global Keybindings

These work on all tabs:

Key	Action
`Tab` / `Shift+Tab`	Switch between PRs, Agents, and Axe tabs
`#`	Open XPrompt Browser (see XPrompt Browser below)
`.`	Toggle visibility of hidden items (reverted PRs, non-run agents, or axe commands)
`:` / `;`	Open the context-aware Command Palette
`,i`	Open Activity Dashboard modal
`i`	Show notifications inbox
`I`	Pin idle mode (IDLE stays until `I` is pressed again; keypresses don't clear it)
`Ctrl+G`	Open the agent editor pre-filled with the most recent VCS xprompt prefix
`Ctrl+L`	Dismiss all currently-visible toast notifications

`Q`	Stop axe daemon and quit
`y`	Refresh current tab
`q`	Quit
`?`	Show help modal

4.8.1 Quit Confirmation

When quitting (`q` or `Q`) while background tasks are still running (task queue workers or background command slots), a confirmation dialog appears showing the count of active tasks and asking whether to kill them and quit. Declining the dialog cancels the quit and returns to the TUI.

4.9 Command Palette

Press `:` or `;` from any tab to open the **Command Palette** — a context-aware modal listing every keymapped action that is currently runnable. The palette is the discovery surface for the TUI: rather than memorizing every chord, you can search by command label, key sequence (e.g. `%n` , `,t` , `zc`), category, or alias.

Behavior:

- Only commands applicable to the current tab and selected entry are shown by default. For example, PR diff appears only when a PR is selected; AXE start/stop appears only on the AXE tab; agent-specific actions appear only when an agent row (not a group banner) is focused.
- Each row shows the keybinding, the command label, and a category badge such as `Navigation` , `PR Actions` , `Agent Actions` , `Copy` , or `Leader` .
- A title-bar badge (`PRs` , `Agents` , or `AXE`) reflects the current tab.

Keybindings inside the palette:

Key	Action
`Type`	Filter commands (case-insensitive substring)
`↑` / `↓`	Move highlight
`Ctrl+P` / `Ctrl+N`	Move highlight
`Enter`	Run the highlighted command
`Esc`	Close without running anything

The palette delegates execution to the same handlers that the keybindings use, so behavior matches pressing the chord directly. Selecting a built-in mode subcommand (e.g. `%n` to copy an agent name) runs the action without forcing you through the transient prefix mode. Custom modes defined in `sase.yml` are also represented per-command.

The `:` / `;` binding follows your configured keymap. To rebind it, set `ace.keymaps.app.open_command_palette` in `~/.config/sase/sase.yml` ; comma-separated keys in that setting are treated as alternate bindings for the same action.

4.10 Project Management Panel

Press `,p` from any tab to open the **Project Management** panel. It lists non-system projects across `active`, `sibling`, and `inactive` lifecycle states, so hidden projects remain reachable even though normal launch and discovery views omit them. Rows include workspace path, active claim count, launchability, and lifecycle or workspace warnings.

Key	Action
`j` / `k`	Move selection
`/`	Filter projects by text
`Tab`	Cycle lifecycle filter: active, sibling, inactive, all
`Enter`	Activate the highlighted project when it is not active
`m`	Mark or unmark the highlighted project
`u`	Clear all project marks
`e`	Open the highlighted ProjectSpec in `$EDITOR`
`a`	Activate highlighted project, or all marked projects
`d`	Deactivate highlighted project, or all marked projects
`Ctrl+D`	Delete highlighted SASE project directory, or all marked directories
`F`	Force the last blocked deactivate after confirming live-work checks
`R`	Reload project records
`q` / `Esc`	Close the panel

When one or more projects are marked, `a`, `d`, and `Ctrl+D` target the marked set instead of only the highlighted row. Successful lifecycle changes clear the affected marks; blocked or failed rows stay marked so you can inspect or retry them. Marks survive filtering and are pruned on reload when their project records disappear.

Deactivating uses the same locked mutation path as `sase project deactivate`. If a project still has `RUNNING` claims or live artifact markers, ACE shows the blocked reason and lets you retry with `F` when the force action is intentional. Activating a project from this panel makes it available again in normal launch pickers.

`e` suspends ACE, opens the selected ProjectSpec in `$EDITOR` (falling back to `nvim`), holds the ProjectSpec edit lock for the editor session, then reloads project records. In this panel, `Ctrl+D` asks for confirmation before deleting the entire SASE project directory: ProjectSpecs, project-local config, artifacts, and related state under `~/.sase/projects/<project>/`. Deletion is refused while the project still has `RUNNING` claims or live artifact markers. It does not delete workspace checkouts, and system-managed projects such as `home` are excluded from the panel.

4.11 Model Overrides

Press `,o` from any tab to open the **Model Overrides** modal. It manages temporary primary and worker provider/model overrides for new agent launches without editing `~/.config/sase/sase.yml`.

The modal shows two lanes:

- **Primary** — the normal default lane for launches without an explicit `%model` directive.
- **Worker** — the secondary lane used by delegated work such as `sase bead work` phase agents that do not have an explicit per-bead model.

Each row shows the current effective model and its source: `override`, `config <key>` (the matched `llm_provider.worker_models` key, e.g. `config claude/opus` or `config codex`), `follows primary`, or `default`. Set or change the primary override with `s` / `c`, clear it with `x`, set or change the worker override with `w`, and clear the worker override with `W`. Both lanes use the same provider-grouped picker and duration choices: `15m`, `30m`, `1h`, `2h`, `4h`, `Until cleared`, or a custom duration like `45m`, `1h30m`, `90m`.

When a worker override is active, the top bar includes a compact `W ...` chip next to the primary override indicator. Permanent `llm_provider.worker_models` config is visible in the modal, not the top bar.

4.11.1 Behavior

- The overrides apply only to default lane selection. Explicit prompt directives (`%model:codex/o3`, `%model:opencode/anthropic/claude-sonnet-4-5`) and an explicit `provider_name` argument always win. `%model:worker` explicitly opts into the worker lane.
- Already-running agents keep their current provider/model. Only **new** launches use the override.
- The primary override is persisted to `~/.sase/llm_override.json`; the worker override is persisted to `~/.sase/llm_worker_override.json`. All sase processes on the same machine see those files. Reads are best-effort self-cleaning: expired or malformed state files are deleted on next access.
- `Until cleared` is a no-expiry mode — convenient, but still a *temporary* state, not a permanent config edit.
- The temporary override is independent of `SASE_MODEL_TIER_OVERRIDE`. A concrete temporary model override takes the full provider/model path; the tier override only applies when no concrete override is active.

4.11.2 Examples

- `codex/o3` for `1h` — switch to Codex `o3` for the next hour, then revert to the configured default.
- `opencode/anthropic/claude-sonnet-4-5` for `1h` — switch to an OpenCode provider/model pair.
- `sonnet` for `30m` — known bare model name; the provider is inferred from plugin metadata.
- Worker `codex/gpt-5.5` for `1h` — route `%model:worker` launches through Codex for the next hour.
- `Until cleared` — leave the override active across sessions; clear it later from the same `,o` modal.

See docs/llms.md (https://sase.sh/llms/#temporary-default-override) for the resolution order and state-file format.

4.12 Notifications Modal

Press `i` (or the `,n` leader chord to jump straight to an agent's notification) to open the notifications modal. See `docs/notifications.md` (https://sase.sh/notifications/) for the full keybinding reference, tag tabs, the priority/inbox/muted section taxonomy, and the per-notification snooze and mute affordances.

The top-bar notification indicator color reflects the highest-priority unread bucket: orange for **PRIORITY** notifications (plan approvals, user questions, mentor reviews, axe errors, CRS results, agent error reports), gold for regular **INBOX** notifications, and cyan when only **MUTED** notifications remain.

4.13 Notification Actions

Some notifications carry an `action` field that triggers a handler when the notification is selected. The following notification action types are supported:

Action	Source	Behavior
`HITL`	Workflow	Opens the workflow human-in-the-loop response modal
`JumpToAgent`	Agent/workflow	Jumps to the matching Agents-tab row
`JumpToChangeSpec`	Sync/workflow	Jumps to the referenced ChangeSpec on the PRs tab
`JumpToMentorReview`	Mentors	Jumps to the ChangeSpec and opens mentor review output when available
`PlanApproval`	Agent	Opens the plan approval modal
`Tmux`	External bridge	Runs `tm <workspace-name>` for the notification's `action_data.workspace_dir`
`UserQuestion`	Agent	Opens the structured user-question response modal
`ViewErrorReport`	Axe/agent	Opens `action_data.error_report_path`, or the first attached file, in `$EDITOR`
`memory_review`	Memory	Suspends ACE and opens the memory proposal review TUI at that proposal

The axe `error_digest` chop creates `ViewErrorReport` notifications whose digest files live under `~/.sase/axe/error_digests/digest_<timestamp>.txt`; user-agent failures can use the same action for their own attached error reports. Memory proposal notifications created by `sase memory write --notify` use `memory_review` with `action_data.proposal_id`. Selecting one opens the same review UI as `sase memory review`, preselected on that proposal; approval or rejection still happens inside the review UI.

4.13.1 Toast Notifications

Each newly-arrived notification produces a short toast in the TUI. The toast text is derived per-action type (plan, question, HITL, axe error, ChangeSpec sync, agent update) so the message previews the actual event rather than a generic "N new notification(s)" line. Severity is also picked per type: plans, questions, and HITL render as warnings; axe errors (and sync failures) render as errors; everything else renders as information.

When more than 3 notifications arrive in the same poll tick, per-notification toasts are consolidated into one grouped toast per severity bucket (e.g., `2 warnings: 1 plan, 1 question`). Ordering is urgency-first: errors, then warnings, then information. Silent notifications are excluded from this pipeline entirely.

Agent completion and failure toasts include the `%name`-set agent name with an `@` prefix when present (e.g., `CLAUDE(opus) @sase-q.land completed: ace(run)-...`); anonymous agents (no `agent_name`) keep the prior format.

4.14 XPrompt Browser

Press `#` on any tab to open the XPrompt Browser modal. It displays all discovered xprompts in a two-panel layout: a filterable list on the left and a syntax-highlighted preview on the right.

Xprompts are grouped by source (CWD `.xprompts/`, CWD `xprompts/`, Home `~/.xprompts/`, Home `~/xprompts/`, project-specific, config `sase.yml`, plugins, built-in). Workflow xprompts (multi-step YAML) are marked with a gear icon; standalone workflows are displayed with the `#!name` insertion syntax. Project-local xprompts defined in each project's `sase.yml` file are also included, even though the TUI's normal config loading does not read project-local config files.

The list rows and preview metadata show the same insertion form and visible input metadata used by `Ctrl+T` completion. Step-only inputs are hidden from this user-facing surface because they are supplied by workflow execution rather than typed by the user.

4.14.1 Keybindings

Key	Action
`Ctrl+N`	Navigate to next xprompt
`Ctrl+P`	Navigate to previous xprompt
`Ctrl+D`	Scroll preview panel down
`Ctrl+U`	Scroll preview panel up / clear input
`Enter`	Edit highlighted xprompt in `$EDITOR`
`Ctrl+O`	Add a new xprompt
`Esc`	Close browser

Type in the filter input to narrow the list in real time.

4.14.2 Editing XPrompts

Press `Enter` on any xprompt to edit it in `$EDITOR`. All xprompts are editable, including plugin and built-in sources — these are copied to the highest-priority user directory (`~/.xprompts/`) before opening, so edits create an override rather than modifying the original. After saving, the browser offers to commit and push changes to git if applicable.

4.14.3 Creating XPrompts

Press `Ctrl+O` to start the guided creation flow:

1. **Location modal** — Choose where to save the new xprompt (CWD `.xprompts/`, CWD `xprompts/`, Home `~/.xprompts/`, Home `~/xprompts/`, or a config file). Press `Ctrl+G` to open the selected config file in `$EDITOR` instead of proceeding with creation.
2. **Filename modal** — Enter a filename (`.md` for prompt parts, `.yml` for workflows). Workflow files are pre-filled with a YAML template containing the workflow scaffold.

3. **Editor** — The file opens in `$EDITOR` for editing.
4. **Git commit** — After saving, the browser offers to commit and push changes.

4.15 Idle Detection

ACE tracks user activity and displays an orange **IDLE** badge in the top bar when the user has been inactive for longer than the configured threshold (`ace.inactive_seconds`, default: 600 seconds). The badge is also shown when the user presses `,I` (leader chord) to manually mark themselves as inactive. Any keypress re-activates the user and hides the badge.

Pressing `I` (capital) activates **pinned idle** mode, shown as a red ■ **IDLE** badge. Pinned idle stays active regardless of keypresses — only pressing `I` again clears it. This is useful when you want to remain marked as idle while still interacting with the TUI. Pinned idle state is persisted to `~/.sase/tui_pinned_idle` and automatically restored when the TUI restarts, so the user remains marked as idle across sessions.

External tools (e.g., chop scripts) can call `is_idle()` from `sase.ace.tui_activity` to check idle status programmatically.

4.16 Jump All Modal

Press `` ` `` (backtick) on any tab to open the Jump All Modal. It displays all entries across PRs, Agents, and Axe tabs with single-keypress hint characters for instant navigation. Selecting an entry switches to the appropriate tab and focuses it.

Hint characters are drawn from an extended alphabet — lowercase `a`–`z` first, then uppercase `A`–`Z` — so modals with many entries can still fit a unique single-keypress hint per row without resorting to multi-character hints.

Key	Action
Hint char	Jump to the corresponding entry
`Esc` / `q`	Close modal

The modal groups entries by tab (PRs, Agents, Axe) and shows contextual information for each: PR names and statuses, agent names with running indicators, and Axe lumberjack/command labels.

4.16.1 Jump Back

Both jump modals support a jump-back feature for toggling between two entries:

- **Backtick jump-back**: Pressing `` ` `` inside the Jump All Modal returns to the previous position, enabling quick toggling between two entries across tabs.
- **Apostrophe jump-back**: Pressing `'` twice (`''`) in the single-tab entry jump mode jumps back to the previously jumped-from entry. The footer shows a "JUMP" mode indicator with `' back` when a target exists.
- **Fast jump**: `Ctrl+O` runs the same current-tab jump-back path without painting hints first; when no jump-back target exists, it selects the first current-tab hint.

The single-tab variant (`'` apostrophe) shows entries only from the current tab with the same hint-character navigation.

4.17 Mentor Comment Stats in PR List

When a ChangeSpec has completed mentor reviews with comments, the PRs tab list entry shows inline stats:

- **checkmark + count** (e.g., `✓3`) — number of accepted comments
- **dot + count** (e.g., `●2`) — number of unread comments

These stats are computed from the latest commit entry's finished mentors. They update as you accept or read comments in the Mentor Review modal.

4.18 Tab Bar Display

The tab bar renders plain tab labels (`PRs` , `Agents` , `AXE`). Per-bucket counts live inside each tab's body — for example the per-panel count summaries on the Agents tab — rather than as suffixes on the tab title itself.

4.18.1 Background Task Indicator

A gear icon (⚙) with a count appears in the top bar when background tasks are running (e.g., sync, mail, accept operations). The indicator automatically hides when all background tasks complete.

4.18.2 Runners Modal

Press `,R` (leader + `R`) to open the runners modal. It shows concurrency information including hook runners, agent runners, and a **Background Tasks** section listing active and recently completed background tasks (sync, rebase, accept, mail, add-tag). Each task entry shows its type, CL name, status, and timestamps.

4.19 File Panel Trimming

When viewing agent files on the Agents tab, large files are automatically trimmed to fit the visible viewport. A blue indicator shows "N more lines below" when content is trimmed. Trim controls (`-` , `=`) are listed in the Agent Actions keybindings above. Trim state is preserved when switching between agents or refreshing data.

Pressing `G` on a trimmed file auto-expands it first, then scrolls to the bottom — so jumping to the end of a large file never leaves you staring at a trimmed page.

4.20 Image Preview Foundation

ACE renders PNG, JPEG, WebP, and GIF attachments with a Pillow-backed Rich cell preview. The renderer decodes the first image frame, preserves aspect ratio within the visible panel bounds, composites transparency, and paints colored half-block cells using truecolor when the terminal advertises it and 256-color approximations otherwise.

Generated images are already attached to successful agent completion notifications and recorded in `done.json` as `image_paths`. The Agents tab file panel and notification modal route supported raster image attachments through this preview layer before attempting text decoding. See `agent_images.md` (https://sase.sh/agent_images/) for supported image extensions, guardrails, and current preview behavior.

4.21 Agent Auto-Naming

Prompts with no `%name` directive, or with a bare `%name`, use the plain auto-name template `@`. SASE reserves the lowest available token from the sequence `0`, `1`, ..., `9`, `a`, ..., `z`, `00`, `01`, ...; with no reserved names, plain auto-naming yields concrete names such as `0`, then `1`.

An explicit `%name` value containing exactly one `@` marker is an agent-name template. SASE substitutes the same token sequence into the marker, so the first allocation for `%name:@.cld` becomes `0.cld`, `%name:build-@` becomes `build-0`, and `%name:research.@.final` becomes `research.0.final`. Later `%wait`, `#fork`, and `#resume` references can use the same template text; within a multi-agent launch, SASE rewrites those references to the concrete name already planned for that template.

Names are permanent IDs: a name used by any existing agent state remains reserved until that agent is explicitly wiped or deleted. This enables the fork-by-name workflow: press `f` on a running named agent to queue a follow-up that waits for it to finish and then loads its conversation history.

4.21.1 Provider/Model Suffixes

When the same base name is shared by multiple co-launched agents (e.g. multi-model fan-out via the `%model:` directive), the rendered display name carries a short `.<provider>` or `.<provider>(<model>)` suffix so each row is distinguishable. Provider suffixes are supplied by the LLM provider plugins via the `llm_provider_short_name` hook (built-in defaults: `cld` for Claude, `cdx` for Codex, `agy` for Antigravity). Additional provider plugins can contribute their own short names. Model-name shorthands come from the `llm_model_short_aliases` hook (e.g. `fable` for `claude-fable-5`, `gpt55` for `gpt-5.5`; see Model Short Aliases (https://sase.sh/llms/#model-short-aliases)) and are resolved against the configured model so the suffix stays compact regardless of how the model was spelled in the prompt or config. Single-runtime spawns omit the suffix.

An explicit `%name:<name>` launch fails before spawning if `<name>` is already reserved. The prompt is saved as a cancelled history entry and the error suggests the lowest free numeric suffix, such as `<name>1`. To deliberately reuse a reserved name from the TUI, launch with `%name:!<name>`; the `!` form confirms that SASE should wipe the previous owner and then claim the name for the new agent. Reviving and dismissing agents preserve their stored names.

The durable registry lives at `~/.sase/agent_name_registry.json` and is rebuilt from visible artifacts plus dismissed bundles when missing or stale. Use `sase agent names migrate-auto` to run the historical auto-name migration that moves older generated names into the permanent namespace; pass `--force` to rerun after the migration marker is present or `--json` for machine-readable output.

4.21.2 Per-Step Naming for Multi-Agent Workflows

Plan-family workflows have a stable family name plus phase suffixes. The root row keeps the family name and acts as the workflow container; generated follow-up rows and phase metadata use canonical double-dash suffixes. For example, if the initial agent family is named `a`:

1. The root workflow row keeps `a`.
2. The planner phase uses suffix `--plan` and is displayed as `a--plan` when ACE renders it as a phase child.
3. Feedback and question-continuation rounds become `a--2`, `a--3`, etc.
4. Terminal follow-ups use the phase suffix, such as `a--code`, `a--epic`, `a--legend`, or `a--commit`.

The base name (`a`) is reserved for the workflow as a whole, so `%wait:a` or `@a` references resolve to the family root. New plan-family metadata stores double-dash `role_suffix` values (`--plan`, `--2`, `--code`, ...). ACE still canonicalizes older dotted suffixes (`.plan`, `.2`, `.code`, etc.) and legacy single-dash suffixes (`-plan`, `-2`, `-code`, etc.) when reading legacy artifacts.

4.22 Agent Statuses

Each agent in the Agents tab displays a status label indicating its current state. Statuses fall into two categories: active (the agent is still running or awaiting input) and completed (the agent has finished).

4.22.1 Active Statuses

Status	Color	Description
RUNNING	Gold	Agent subprocess is executing
WAITING	Light blue	Agent is queued, waiting for another agent to succeed (`%wait`)
WAITING INPUT	Amber/orange	Workflow is paused at a human-in-the-loop (HITL) step
PLAN	Pink/magenta	Agent has produced a plan and is waiting for user approval
PLAN APPROVED	Cyan	Plan was approved; follow-up agent has been spawned
EPIC APPROVED	Cyan	Plan was approved as an epic; `--epic` follow-up is running
PLAN COMMITTED	Cyan	Plan was approved with auto-commit; `--commit` follow-up is running
QUESTION	Amber	Agent is asking the user a question (via `/sase_questions`)
RETRYING	Orange	Agent hit a retryable error and is in a countdown before retrying

`QUESTION` status survives notification dismissal. While an agent is waiting for an answer it writes a `pending_question.json` marker into its run directory and removes the marker once it resumes (whether the user replied, the agent was killed, or it crashed). Any `RUNNING` row whose run directory contains the marker is shown as `QUESTION`, so the "waiting on you" status keeps appearing even after you dismiss the matching question notification from the inbox. The `,n` shortcut (jump to the open question) reads the marker directly when no unread notification is left, so it can still reopen the question modal.

`QUESTION` also propagates up agent families. When a completed row recorded a question (`questions_times` is non-empty) but has neither a persisted `question_response_path` nor a later follow-up child, the parent workflow row inherits `QUESTION` so the family still shows as waiting on you. Once the user response is persisted, the continued work usually appears as the next numeric phase (`--2`, `--3`, ...); `--q` identifies the question phase in metadata and phase labels. On the next status pass, the parent is re-evaluated without the stale question override. If the parent has several active children, the most recently started one wins, so a newer `RUNNING` child can overtake the `QUESTION` override on the parent.

The keybinding footer renders available conditional actions as non-breaking key/label chips. When the chips do not fit on one line, the footer switches to a deterministic grid so narrow terminals and leader-mode action sets do not wrap in the middle of a binding. Mode labels such as `LEADER` are pinned on the left, and the axe/status indicator remains pinned on the right.

The footer also shows axe daemon status indicators:

Status	Color	Description
RUNNING	Green	Axe daemon is running normally
STOPPED	Red	Axe daemon is not running
STARTING	Yellow	Axe daemon is starting up
STOPPING	Yellow	Axe daemon is shutting down
RESTARTING	Deep sky blue	Axe daemon is restarting (triggered by `--restart-axe` flag)

During TUI startup the footer slot shows a live **starting** stopwatch with a rotating glyph in place of the daemon status, ticking at ~10 Hz until the TUI finishes mounting and the real axe status resolves. The background color turns from its normal tone to a slow-startup tone once the elapsed time crosses the slow threshold, giving immediate visual feedback on cold-start latency. A safety timeout forcibly retires the stopwatch if the mount signal never fires.

4.22.2 Completed Statuses

Status	Color	Description
DONE	Green	Agent completed successfully
PLAN DONE	Green	Plan workflow fully completed (all steps)
TALE DONE	Green	Tale plan workflow fully completed (all follow-ups)
EPIC CREATED	Green	Plan workflow completed and its latest `-epic` follow-up finished successfully
FAILED	Red	Agent exited with an error

Completed agents can be dismissed with `x` on a single row, or through the `X` cleanup panel for focused-panel, global, tag, marked, group, and custom selections. `DONE`, `PLAN DONE`, and `TALE DONE` rows with a saved response path are resumable from the Agents tab.

When a terminal agent becomes unread, ACE marks it with the completed-agent indicator and includes it in the Agents header unread count. Selecting that row, jumping to it with `,j`, or toggling it back to read with `U` acknowledges the row and dismisses the matching user-agent completion notification. Manually marking a row unread with `U` arms it for normal acknowledgement after you move away and return, so the marker can be used as a short-lived reminder without leaving stale inbox entries.

If the currently focused row finishes while you are already on the Agents tab, ACE still marks it unread and keeps the completion notification active until a real navigation or selection event acknowledges it. A refresh that merely preserves focus does not silently consume the unread marker.

The `unread` count in the Agents header is drawn as black text on a gold pill so the "you still have unseen completed work" signal stands out from the rest of the colored metrics. It uses the same gold tone as the top-bar notification indicator, giving you a single color to scan for.

Switching to the Agents tab does not bulk-dismiss completion notifications. ACE projects active completion notifications onto unread rows, then acknowledges rows one at a time when you select or navigate into a terminal unread row. Bulk acknowledgement is explicit through `,u`, which marks loaded unread completed agents read. Plan approvals and user questions are never auto-dismissed by this flow; they always require explicit `y` / `n` confirmation from their respective modals.

4.22.3 Agent Revival

Press `R` on the Agents tab to revive previously dismissed work. ACE opens the saved-group revival modal first, showing newest saved groups with a right-hand preview of included agents, projects, PRs, statuses, provider/model labels, and revival count. Select a group and press Enter to revive it, choose **Load more saved groups...** to page older groups, or choose **Custom revival search...** to open the older dismissed-agent search where you choose all, home, project, or PR scope manually.

Use `m` to mark related Agents-tab rows and then `s` to save and dismiss them as a group. The save modal accepts an optional human name. Leaving it blank keeps the generated display title, such as "3 agents from @review" or "2 agents in auth_retry". Saving a marked group hides the selected rows from the normal Agents tab without killing running processes. When a marked top-level workflow row has child rows, ACE also includes the children in the saved group so revival can restore the original tree.

Dismissed agents are saved as individual bundle files under month shards in `~/.sase/dismissed_bundles/YYYYMM/` and can be restored later. Saved group metadata lives under `~/.sase/dismissed_agent_groups/` and stores stable references to those bundle files plus the optional group name, status counts, projects, PRs, model/provider metadata, and agent tags. There is no limit on the number of dismissed agents or saved groups that can be stored.

Dismiss operations are O(1) per agent: each agent is saved to its own JSON file rather than a monolithic store. Parent workflow rows use `<raw_suffix>.json`; workflow children use `<raw_suffix>__c<step_index>.json`. ACE keeps a SQLite summary index in the dismissed-bundle directory so the revive modal and internal lookups can list dismissed agents without opening every bundle. Use `sase agent archive verify` to check that maintenance index, or `sase agent archive rebuild-index` to rebuild it from bundle files. The index stores metadata such as status, name, project, model, provider, workflow, and ChangeSpec metadata; it is not a full-text copy of agent chat contents.

Revival removes the agent identity from the dismissed set, restores enough artifact files for ACE to rediscover the agent, and preserves the dismissed bundle as historical recovery data. Saved-group revival skips missing bundle references with a warning and restores the remaining agents. Group metadata is not deleted after revival; ACE marks the group with `revived_at` and increments `times_revived` so the modal can show previous use. The reload path forces a full-history scan and can hydrate the just-revived row directly from the bundle, so agents still appear after revive even if the persistent artifact index was empty or stale.

Every revival also writes structured events to `~/.sase/logs/events.jsonl` (start, per-agent success, per-agent failure). Read them back with `sase revive-log` — see Agent revival audit log (https://sase.sh/troubleshooting/agent-revival/) for the record schema and CLI flags.

Legacy Dismissed-Name Prefix

Current dismiss and revive operations preserve stored agent names, per-agent tags, and top-level/workflow-child identity. Older dismissed bundles may still contain `YYmmdd.<base>` names from the previous dismissal model, and ACE keeps compatibility helpers for reading those bundles. Bare `%wait` (no target) intentionally skips legacy dismissal-prefixed candidates so it anchors on a live, visible agent.

4.23 Agents Tab Metadata Panel

The Agents tab metadata panel (cycled to via `]` / `[`) shows structured information about the selected agent:

- **Agent details**: Name, status, model, provider, CL association, and chronologically sorted timestamps:
- `Bead` — shown for agents launched by `sase bead work`, inferred from phase, exact epic, or legacy `.land` names
- `WAIT` — when the agent was spawned (waiting for a slot)
- `BEGIN` — when execution started
- `PLAN` — each plan proposal round (multiple entries when re-planning occurs)
- `FBACK` — each time the agent requested feedback from the user
- `QUEST` — each time the agent asked the user a question
- `RETRY` — each time the agent entered retry state (retryable error)
- `CODE` — when the agent began writing code
- `EPIC` — when an epic follow-up agent was launched after plan approval
- `DONE` — when execution completed
- **OUTPUT VARIABLES**: Small string values written by the agent with `sase var set KEY=VALUE`. Keys are sorted for stable display, multi-line values are indented, and the section is omitted when the agent has not published variables. These values are stored in `agent_meta.json`, so they are visible metadata rather than secret storage.
- **AGENT REPLY**: The agent's live or completed reply content, streamed from `live_reply.md` during execution and read from the artifacts directory after completion. When per-turn reply timestamps are available (recorded in `live_reply_timestamps.jsonl`), the reply is displayed with timestamp dividers between each agent turn. For agents with follow-up phases (planner, feedback rounds, coder), the AGENT REPLY section consolidates replies from all phases into a single view with purple phase dividers showing each phase's label and start time. Phase

labels are derived from canonical plan-family `role_suffix` values: `--plan` renders as `PLANNER`, `--code` as `CODER`, `--q` as `QUESTIONS`, `--epic` as `EPIC`, `--legend` as `LEGEND`, `--commit` as `COMMIT`, and numeric feedback suffixes such as `--2` as `PLANNER (round 2)`. Legacy dotted and single-dash suffixes render the same way.

- **WORKFLOW VARIABLES**: xprompt workflow output variables from step outputs with additional `meta_*` keys are grouped under a dedicated header. The special routing keys `meta_project`, `meta_changespec`, and `meta_workspace` are still promoted into the normal header fields; other metadata keys are title-cased and shown in this section.

When the file or tools panel is empty, the `g` / `G` keys automatically fall back to scrolling the metadata panel.

4.24 Agents Tab Tools Panel

The tools panel sits between the file panel and the metadata panel in the Agents-tab cycle (`]` advances forward, `[` goes back). It shows a chronological timeline of the LLM tool calls the selected agent has made — file reads, edits, bash invocations, web fetches, sub-agent launches, and so on.

Entries are read from the `tool_calls.jsonl` artifact in the agent's run directory. Each call renders as one timeline row:

- A status label colored by outcome — `ok` (success), `fail` (error), `stop` (interrupted), `agent` (sub-agent launch), or `wait` (the post-call record has not arrived yet).
- The tool name, optionally followed by a compact target (such as the file path the tool acted on) and the call's duration.
- A short, length-bounded preview of the call result on the next line, when the collector captured one.

The panel header shows the total call count, the failure count, the interrupted count, and a timestamp for the most recent reload. While a background reload is in flight (because the artifact changed on disk), `(refreshing...)` appears next to that timestamp. The body shows `No tools artifact available` when the file does not yet exist for this agent and `No tool calls recorded` when the file exists but contains zero records.

For retry chains and planner-to-coder follow-up families, the panel aggregates `tool_calls.jsonl` from related artifact directories so the selected logical agent shows one ordered tool timeline. Discovery uses the persistent artifact index when it is available; if the index is missing or stale, ACE falls back to direct lineage pointers plus a bounded scan of nearby legacy sibling artifacts.

Records are produced by writers that share one normalized on-disk format. Claude uses the SASE tool-call hook collector as the preferred source and keeps its stream-derived parser as a fallback when hooks are unavailable. Codex writes equivalent rows from its `codex exec --json` stream with `runtime: "codex"` and `source: "stream"`; current Codex start/completion events can show pending rows, result previews, failures, interruptions, and durations, while older completed-only `function_call` rows remain readable with more limited detail. Qwen writes stream-derived rows from its `--output-format stream-json` output with `runtime: "qwen"` and `source: "stream"`; start/completion (and Qwen's `tool_use` / `tool_result`) pairs collapse into single rows the same way Codex pairs do. Antigravity (`agy`) runs in plain-stdout mode and exposes no machine-readable tool-call contract, so they contribute no tool rows and the panel simply shows nothing for `agy` runs. See LLM Providers — Claude tool-

call hooks (https://sase.sh/llms/#claude-tool-call-hooks), LLM Providers — Codex tool-call capture (https://sase.sh/llms/#codex-tool-call-capture), and LLM Providers — Qwen tool-call capture (https://sase.sh/llms/#qwen-tool-call-capture) for provider integration details.

4.25 Plan Workflows

When an agent submits a plan via `/sase_plan` (or `sase plan propose`, including the `%epic` path), it enters a planning phase before executing:

- **PLAN** — The agent has produced a plan and is waiting for user approval. Shown in pink/magenta in the prompt panel.
- **PLAN APPROVED** — The plan has been approved and the follow-up agent has been spawned. Shown in cyan/turquoise.
- **PLAN REJECTED** — The user rejected the plan via the `r` keybinding. Treated as a terminal decision (not a failed agent run): the rejected agent remains on the Agents tab as a completed row with this status, and the redundant completion notification is suppressed.

Plan files generated by the agent are displayed in the file panel alongside other agent artifacts. Plan approval notifications include the LLM provider and model name, so users can see which model proposed the plan (visible in both the TUI notification modal and Telegram delivery).

When `sase plan propose` writes the plan, it also touches `~/.sase/.ace_refresh_pulse` to wake any running TUI immediately — PLAN status appears without waiting for the next auto-refresh tick. The pulse file is consumed by the inotify artifact watcher (see Auto-Refresh) and is harmless if no TUI is open.

Root plan workflows also surface PLAN when a re-proposed plan is still awaiting review. Plan and feedback timestamps from feedback-round children (`--2`, `--3`, ...; legacy `-2`, `.2`, etc.) propagate onto the root entry, and whenever the root's latest plan timestamp is newer than its latest feedback timestamp the override engine restores `PLAN` over a `RUNNING` or `DONE` label. This applies only to root plan workflows that have not yet spawned a terminal follow-up (`--code`, `--epic`, ...); once a terminal follow-up is launched, the parent moves on to `PLAN APPROVED` (or the matching follow-up status) instead.

The Plan Review modal title shows a provider-themed `PROVIDER(model)` badge between the "Plan Review" label and the plan filename — orange for Claude, lime for Codex, Antigravity indigo (`#6E5DE7`) for agy, neutral muted for other providers. The badge is omitted when provider/model metadata is absent, leaving the legacy title shape unchanged.

The same pending approvals are available from the CLI. Run `sase plan` to see pending proposals, recent approvals, and inferred rejected archived plans; run `sase plan approve <id-prefix> --kind approve|commit|epic|legend|tale` to write the same response protocol used by the TUI modal. If the selector is omitted, the CLI approves only when exactly one proposal is pending. `approve` starts the coder without committing an SDD plan, `tale` commits the plan as an SDD tale and starts the coder, `epic` and `legend` commit the matching SDD tier and launch the bead follow-up, and `commit` records the approved plan in SDD without launching a coder. `-m/--model` picks the follow-up agent's model, while `-p/--prompt` adds extra coder instructions for the `approve` and `tale` paths.

For active Agents-tab rows, `a` cycles through normal auto-approve, epic auto-approve, and disabled. Epic auto-approve is the same plan-specific path as the `%epic` directive: the next submitted plan is accepted as an epic, SDD epic artifacts are written, beads are initialized, and the epic follow-up agent is launched. It does not answer unrelated HITL prompts.

4.25.1 Plan Approval Keybindings

Key	Action
`a`	Approve and run coder without committing an SDD tale
`t`	Save as tale and run coder
`c`	Open Custom Approval
`r`	Reject the plan
`f`	Request feedback (send follow-up questions to the agent)
`e`	Edit the plan file in `$EDITOR`
`E`	Mark the plan as an epic (creates bead)
`L`	Mark the plan as a legend (creates legend-tier bead)
`y`	Copy plan content to clipboard
`Y`	Copy plan file path to clipboard
`Ctrl+D` / `U`	Scroll plan content down / up
`g` / `G`	Scroll to top / bottom
`q` / `Esc`	Cancel

The question modal also supports `y` to copy questions and selected answers.

4.25.2 Custom Approval

Pressing `c` in the plan approval modal opens a custom approval dialog. Choose the approval outcome directly: Approve, Tale, Epic, or Legend. These choices map to the same response protocol used by external approval transports: Approve runs the coder without asking the runner to commit an SDD plan, Tale commits under `sdd/tales`, Epic commits under `sdd/epics`, and Legend commits under `sdd/legends`.

Key	Action
`Enter`	Choose the highlighted action
`a`	Highlight Approve
`t`	Highlight Tale
`e`	Highlight Epic
`l`	Highlight Legend
`m`	Select coder model
`p`	Edit additional coder prompt

`Ctrl+N` / `P`	Next / previous action
`q` / `Esc`	Cancel

The dialog keeps the custom coder prompt and follow-up model controls:

- **Additional prompt** — Optional extra instructions for the coder follow-up. It is used by Approve and Tale; Epic and Legend generate their follow-up prompts from the bead xprompts.
- **Coder model** — Select an LLM model for the next follow-up agent instead of using the worker-lane default. For Approve and Tale that agent is the coder; for Epic and Legend it is the bead follow-up. Shows all registered models grouped by provider (Claude, Codex, Antigravity, Qwen, OpenCode) with a "Custom..." option for freeform input. Type to filter by provider, model id, label, or short alias; use `j` / `k` or arrows to navigate, `Enter` to select, `Esc` to clear the filter or cancel, and `'` for jump hints over the visible selectable rows. The displayed default is the worker lane resolved from the **planner's** concrete provider/model — an active worker override, a matching `llm_provider.worker_models` entry for the planner's primary lane, then the planner's own provider/model. Selecting a specific model and then re-opening the picker and choosing "Worker model (default)" resets the follow-up back to that worker default (distinct from pressing `Esc`, which keeps the current selection).

The custom approval dialog no longer exposes separate commit/run switches because the selected outcome determines the commit location and follow-up behavior.

4.26 Linked Chats in Multi-Step Workflows

When a workflow spawns multiple agents (e.g., a planner step followed by a coder step), the chat history files for each step are cross-linked via a `## Linked Chats` markdown section. This section is inserted near the top of each chat file and lists all related agents with their roles and file paths, making it easy to trace the full workflow from any individual agent's chat history.

For example, a plan-then-code workflow produces chat files with:

```
## Linked Chats

- **1. planner** — `/path/to/planner_chat.md`
- 2. coder — `/path/to/coder_chat.md`
```

The current agent's entry is bolded for quick identification.

4.27 Retry/Fallback Display

When an agent encounters a retryable error (configured via `llm_provider.retry`), the Agents tab shows retry state:

- **RETRYING** — Shown in bold orange when waiting before the next retry attempt. Includes a countdown timer: `RETRYING (45s)`.
- **↻N** — Shown after the status for running agents that have retried. The number indicates how many retries have occurred (e.g., `↻2` means two retries so far).
- **▸Model** — Appended to the retry annotation when the agent has fallen back to an alternate model (e.g., `↻3▸flash`).

4.27.1 Prior Agent Attempts

Every time the axe retry loop retries an agent — context-limit retry, provider/API-error retry, user-configured retry, or fallback-model switch — the failed attempt's partial reply, error text, timestamps, and model are snapshotted under `<artifacts_dir>/attempts/<N>/`. The AGENT REPLY area in the Agents tab renders these prior attempts inline with styled dividers before the current/final attempt, so the full arc of the agent's work stays visible in one scroll.

ACE hydrates prior-attempt history lazily. Normal Agents-tab refreshes do not enumerate every `attempts/<N>/` directory; the selected detail panel, `D` attempt-view toggle, and content search hydrate the needed attempt records on demand.

Press `D` to collapse the view to the current attempt only; press `D` again to re-expand. The binding only appears in the keybinding footer when the selected agent has one or more prior attempts.

4.28 Custom Keymaps

All TUI keybindings are configurable via the `ace.keymaps` section in `sase.yml`. You can remap any built-in key and define entirely new prefix-key modes.

4.28.1 Remapping Built-in Keys

Override any app-level keybinding under `ace.keymaps.app`:

```yaml
ace:
  keymaps:
    app:
      next_changespec: "n" # Remap j → n
      prev_changespec: "p" # Remap k → p
      show_notifications: "N" # Remap i → N
```

4.28.2 Custom Modes

Define user-defined prefix-key modes under `ace.keymaps.modes`. Each custom mode has a `prefix` key and a `keys` dict where each sub-key specifies either a `shell` command or a built-in `action`:

```yaml
ace:
  keymaps:
    modes:
      my_mode:
        prefix: ";"
        keys:
          run_tests:
            key: "t"
            shell: "just test"
          show_log:
            key: "l"
            shell: "git log --oneline -20"
          refresh:
            key: "r"
            action: "refresh"
```

Pressing `;` activates the mode, then pressing `t` runs `just test`, `l` shows the git log, etc.

4.28.3 Validation

The keymap loader validates all configuration:

- **Invalid keys** are reverted to their defaults with a warning
- **Duplicate keys** (two actions bound to the same key) are detected and the conflicting override is reverted
- **Prefix conflicts** between custom mode prefixes and existing app bindings are warned

See `docs/configuration.md` (https://sase.sh/configuration/) for the full `ace.keymaps` configuration reference.

4.29 Prompt Input Widget

The prompt input is a multiline TextArea widget that supports two editing modes: INSERT and NORMAL. The widget provides markdown syntax highlighting for prompt content (headings, bold, italic, code blocks, lists, etc.).

When loaded prompt text contains literal top-level `---` multi-agent separators, ACE renders the text as a prompt stack: one pane per agent segment. YAML frontmatter at the start stays prompt-level metadata, and `---` lines inside fenced code blocks are left alone. A `#name` multi-agent xprompt invocation stays a single pane and expands only when it is launched. During live editing, typed `---` lines stay literal text; add prompt panes with `g-` in prompt NORMAL mode. The detailed multi-agent parsing rules live in the XPrompt reference (https://sase.sh/xprompt/#multi-agent-prompts).

4.29.1 INSERT Mode (Default)

Key	Action
`Enter`	Submit; in a prompt stack, open the submit chooser
`Ctrl+S`	Submit the whole stack top-to-bottom
`Ctrl+G Enter`	Submit only the selected pane
`Ctrl+C`	Cancel the prompt; in a prompt stack, cancel only the selected pane
`Ctrl+J`	Insert a newline
`Ctrl+A`	Move to start of line (jumps to previous line start if already at col 0)
`Ctrl+E`	Move to end of line (jumps to next line end if already at end)
`Ctrl+G`	Start the prompt-local prefix; press `g` or `Ctrl+G` again to open `$EDITOR`
`Ctrl+G Enter`	Submit only the selected pane
`Ctrl+G j/k`	Focus the next / previous pane and leave the target pane in INSERT mode
`Ctrl+G J/K`	Move the active pane down / up and leave it in INSERT mode
`Ctrl+G -`	Add an empty bottom pane
`Ctrl+G =`	Show/focus the xprompt frontmatter panel
`Ctrl+G s/S`	Stash the selected pane / every non-empty pane
`Ctrl+G p/P`	Load / restore stashed prompt drafts

`Ctrl+Y`	Open the workflow YAML editor
`Ctrl+K`	Open prompt history, filtered by the current single-line prompt
`Ctrl+T`	Completion (directives, xprompts, slash skills, or file paths; see Completion)
`Ctrl+R`	Recursive fuzzy file finder using the same prompt-aware path root as file completion
`Tab`	Snippet expansion (see below)
`#@`	Open XPrompt snippet picker (type `#` then `@`)
`Escape`	Switch to vim NORMAL mode

Text automatically wraps at the terminal width, breaking at spaces (never mid-word). Line numbers appear in cyan when the text exceeds one line.

4.29.2 Prompt Stacks

Prompt stacks are the ACE editing surface for literal `---` multi-agent prompts. Loading multi-agent prompt text from history, a whole-bar editor session, or an editor buffer that returned through `%edit` splits top-level `---` segment separators into panes labeled `agent 1`, `agent 2`, and so on; the border title shows `Prompt · N agents`. Restoring stashed prompts and using marked-agent `,x` can also open a stack, but those paths load one pane per selected draft or agent instead of re-parsing each pane's text. Panes are ordered top-to-bottom for whole-stack submission. The bottom pane is active by default so you can keep drafting the newest segment; it is not a priority marker, and pressing `Enter` immediately opens the submit chooser.

Inactive panes stay compact, and the active pane takes the available height. A `---` line typed while INSERT mode is active stays literal prompt text; use `Ctrl+G -` while drafting, or `g-` from prompt NORMAL mode, to add a new bottom pane. `Ctrl+G g` and `Ctrl+G Ctrl+G` open the whole stack in `$EDITOR` when the bar already has multiple panes (a single-pane bar opens just the current prompt). Returning from a whole-bar editor session or from `%edit` reloads xprompt-style Markdown and parses `---` separators into fresh panes. History loads parse only real multi-agent prompts; a single history item with leading YAML frontmatter stays one verbatim pane instead of auto-opening the Frontmatter Panel.

In prompt INSERT mode, pressing `Ctrl+G` opens the same context-aware hint row as prompt NORMAL mode's `g` prefix, plus the editor continuation. Press `Esc` while the prefix is pending to cancel it and stay in INSERT mode.

In prompt NORMAL mode, pressing `g` opens a small hint row for the prompt-local `g` prefix actions currently available.

Key	Action
`Enter`	Open the submit chooser; when one pane remains, `Enter` submits it normally
`Ctrl+S`	Join non-empty panes with `---` separators and launch the whole stack as one multi-agent prompt
`g<enter>`	Launch the selected pane and remove it from the stack
`Ctrl+C`	Record the selected pane as cancelled history and remove it; the final remaining pane cancels normally
`Escape`	Enter NORMAL mode for stack navigation
`gj` / `gk`	Focus the next / previous pane in NORMAL mode; focus cycles at the stack edges
`gJ` / `gK`	Move the active pane down / up in NORMAL mode; reorder cycles at the stack edges

`g-`	Add an empty bottom pane in NORMAL mode and switch it to INSERT mode
`g=`	Show/focus the xprompt frontmatter panel; in the focused panel, run its deactivate/apply path
`gs`	Stash the selected pane and remove it from the stack
`gS`	Stash every non-empty pane and dismiss the prompt bar
`gp`	Load selected stashed prompt drafts into the current bar without deleting them from the stash
`gP`	Restore selected stashed prompt drafts into the current bar and delete them from the stash

Submitting one pane at a time re-attaches prompt-level frontmatter to the launched pane so local xprompts and metadata continue to resolve. Empty selected panes are dropped without launching. Whole-stack submission joins panes in top-to-bottom order and then uses the usual multi-agent launch path, including `%wait`, `%name`, `%model`, and other segment-local directives. Segment order alone does not make later agents wait; add `%wait` to the later pane when it must start after an earlier agent succeeds.

The `Enter` submit chooser accepts `a` or `Ctrl+S` for all panes, `c` for the current pane, and `Esc` / `q` to cancel without changing the stack.

Prompt stashes are a per-user draft pile stored outside prompt history. `gs` captures the selected non-empty pane plus the shared prompt frontmatter; when other panes remain the bar stays open, and when the last pane is stashed the bar closes without also recording the draft as cancelled history. `gS` captures all non-empty panes in their current order. `gP` opens a destructive restore picker: selected entries load oldest-first with ties broken by original pane index and are removed from the stash, while unselected entries stay available. `gp` opens the same picker in non-destructive load mode, so selected entries are copied into the current bar and remain available in the stash. Restore no longer opens from a global `,P` when no prompt bar is active. A small top-bar badge shows how many restorable drafts are currently stashed.

4.29.3 Completion

Press `Ctrl+T` to activate token completion. The completion kind is determined by the token under the cursor:

- **XPrompt completion**: When the cursor is on a `#`-prefixed token (e.g., `#my_pro`), completion shows matching xprompt names from all discovery sources. Built-in workspace references such as `#cd` are included; use `#cd:<path>` to run from a specific directory without VCS workspace management. Completion rows include the xprompt kind and visible typed inputs, with required arguments shown as `name: type` and optional arguments shown as `name?: type` plus a default when the default is a simple scalar. Standalone workflow references use the `#!name` insertion form; typing `#!` filters completion to entries whose canonical insertion starts with `#!`.
- **Project/ChangeSpec completion**: When the cursor is on a `#+` token, or on a `+` token that is the first character in the prompt, completion opens a project/ChangeSpec picker. The picker contains active launchable projects plus active PR-sized ChangeSpecs in `WIP`, `Draft`, `Ready`, or `Mailed` status; system-managed `home`, inactive projects, sibling records, and non-launchable projects are excluded. Typing after the trigger filters by project name, project alias, or ChangeSpec name prefix. Accepting a row inserts the canonical workspace tag such as `#gh:sase` or `#gh:my_change`, replacing existing line-start VCS tags when present or placing the tag after leading frontmatter/directives when no tag exists.

- **Slash-skill completion**: When the cursor is on a slash-skill token such as `/` or `/sase_`, completion filters the same catalog to xprompts marked as `skill: true` and inserts `/skill_name`. Packaged built-in skills are included, so `/sase_plan`, `/sase_questions`, and other bundled SASE skills are available without a project-local xprompt file.
- **XPrompt argument completion**: When the cursor is inside a known xprompt argument position, `Ctrl+T` completes the active argument instead of the xprompt name. For `path` inputs it delegates to file path completion, for `bool` inputs it offers `true` and `false`, and inside parenthesized syntax it completes missing `name=` arguments without repeating names already present in the argument list. Numeric inputs keep the type hint visible but do not invent values.
- **Directive completion**: When the cursor is on a `%`-prefixed directive token (e.g., `%m`), completion lists user-facing prompt directives and accepts aliases into their canonical forms. For example, `%m` completes to `%model` and `%w` completes to `%wait`. The panel shows each directive's aliases and whether it takes an argument or is a flag.
- **File path completion**: When the cursor is on a path-like token (starting with `/`, `./`, `../`, `~/`, or containing `/`), completion shows matching filesystem entries. Tokens starting with `@` are also recognized — the `@` prefix is preserved in the completed path (useful for file-reference arguments). Relative paths use the prompt-selected base directory: a resolvable `#cd` reference takes precedence; without `#cd`, registered workspace-provider refs and known-project refs such as `#git:<project>` or `#gh:<owner>/<repo>` can root completion in that project checkout. If no prompt workspace ref resolves, ACE uses the TUI process directory.
- **File-history completion**: When the cursor is in whitespace (or at an empty prompt prefix), `Ctrl+T` opens a list of recently referenced files drawn from prompt history, ranked by recency. Project-local `.sase/` paths are filtered out so internal bead/plan artifacts don't pollute the suggestions. Press `Ctrl+D` in the completion panel to delete the highlighted entry from the on-disk history.

Key	Action
`Ctrl+T`	Start completion or insert shared prefix
`Ctrl+N` / `Down`	Next candidate
`Ctrl+P` / `Up`	Previous candidate
`Enter` / `Ctrl+L`	Accept highlighted candidate
`Escape`	Cancel completion

Press `Ctrl+R` to open the recursive fuzzy file finder. With a token such as `src/alp`, `src/` becomes the search root and `alp` pre-seeds the fuzzy query; with no token, the finder starts at the prompt-selected base directory described above. If a `Ctrl+T` file, recent-file, or path-argument candidate is highlighted, that highlighted path seeds the recursive root instead. The finder uses `git ls-files --cached --others --exclude-standard` from the search root when possible, falls back to a bounded filesystem walk, and inserts the selected path into the prompt position captured when the finder opened. Inside the finder, type to filter, use `Ctrl+N` / `Ctrl+P` or arrows to move, `Ctrl+U` to clear the query, `Enter` to insert, and `Esc` to cancel.

ACE also computes a non-disruptive live suggestion after a short debounce while the prompt input is in INSERT mode. The suggestion appears in the prompt bar subtitle as `[^L] accept ...`; press `Ctrl+L` to accept it. `Enter` still submits the prompt as typed, so live suggestions cannot accidentally replace text on send.

Live soft completion covers directives, xprompt names, xprompt argument names, and bool argument values. File-path soft completion is disabled by default because it can scan the filesystem while typing; enable it with `ace.prompt_completion.auto_file_paths: true`. The xprompt/skill menu also opens automatically while typing matching `#name`, `#!name`, or `/skill` tokens; disable that xprompt auto-open behavior with `ace.prompt_completion.auto_xprompt_menu: false`. The directive menu likewise opens automatically while typing matching `%name` tokens; disable it with `ace.prompt_completion.auto_directive_menu: false`. Both auto-menus open only once at least one identifier character follows the marker (bare `#`, `/`, and `%` stay quiet) and never auto-accept a single match. The `#+` / offset-zero `+` project/ChangeSpec picker opens when `+` completes a valid trigger and is also available through manual `Ctrl+T`. Manual `Ctrl+T` completion still supports file paths, xprompt names, directives, skills, and project/ChangeSpec tags regardless of those settings. Live suggestions pause while the manual completion panel is open, while snippet tabstops are active, in NORMAL mode, and during feedback prompts.

For file completion, directories appear before files in the candidate list. Dotfiles are hidden unless the partial prefix starts with `.`. Accepting a directory automatically re-opens completion for the next level (drill-down). The completion panel shows up to 10 candidates at a time and scrolls to keep the highlight visible. When exactly one xprompt or file candidate matches, accepting completion inserts the canonical reference immediately.

Accepting an xprompt completion, or selecting an xprompt from the `#@` picker, opens an `xprompt args` hint panel when the xprompt has required user-facing inputs. The panel shows the supported arguments and highlights the active one. Press `:` while the accepted reference is still current to switch to colon syntax, or press `(` to insert a required-argument named snippet and use `Tab` to advance through the snippet fields.

The same smart insertion rules apply to `#@` selections and `Ctrl+T` completions. A selected xprompt with no required inputs inserts a trailing space, a single required non-text input inserts colon syntax, a single required text input inserts double-colon shorthand, and multiple required inputs insert a parenthesized named-argument snippet.

The same hint panel appears while typing narrow, known argument forms such as `#name:`, `#!name:`, `#ns/name:`, `#ns__name:`, `#name!!:`, `#name??:`, `#name(`, and `#name(arg=`. The hint is advisory; the backend xprompt parser still owns expansion semantics when the prompt is submitted. Detection intentionally stays conservative, so prose shorthand, URLs, unknown xprompt names, `#name+`, and completed colon text such as `#name: value` do not keep the prompt-bar hint open.

4.29.4 Alt Brace Syntax (`%{...}`)

The prompt input has dedicated highlighting and editing help for the `%{A | B}` alt fan-out shorthand (see the Alt Directive reference (https://sase.sh/xprompt/#alt-directive)). It distinguishes the alt delimiters from the branch separators so a fan-out is easy to read at a glance:

- The `%{` opener and `}` closer are styled as **delimiters** (bold accent).
- Top-level `|` branch **separators** use a dimmed accent so they read differently from the delimiters.
- A branch name before a top-level `=` (e.g. `sec=` in `%{sec=... | perf=...}`) is highlighted as a **branch name**.
- An unmatched `%{` (or stray closer) is flagged as an **error** span.

The alt overlay layers on top of the existing Jinja and search highlighting rather than replacing it, and it uses the same size guards, so highlighting stays responsive on large prompts.

Editing help in the ACE prompt input mirrors the Jinja auto-pair behavior and only fires for the `%{...}` shorthand:

- **Auto-pair** — typing `{` immediately after a directive-valid `%` inserts the matching `}` and leaves the cursor between the braces (`%{|}`).
- **Paired delete** — backspacing the `{` in `%{|}` also removes the auto-inserted `}`; a forward delete on `%|{}` removes both braces.
- `|` **separator normalization** — typing `|` inside a live `%{...}` span inserts a padded `|` separator, keeps the cursor after the trailing space and before the closing `}`, and normalizes comma spacing in the current branch. For example, typing `|` at the end of `%{foo ,bar, and baz` yields `%{foo, bar, and baz | }` with the cursor before `}`.

These edits are suppressed when there is an active selection or when the cursor is not inside a directive-valid `%{...}` context, so ordinary `{` and `|` typing elsewhere is unaffected. External editor integrations do not own `%{}` auto-pairing or paired delete; editor-local brace-pair plugins own that lifecycle there. The Neovim plugin (https://github.com/sase-org/sase-nvim) still provides the same separator-normalization behavior for prompt buffers.

4.29.5 NORMAL Mode

Press `Escape` in INSERT mode to enter vim-style NORMAL mode. The border title shows `[NORMAL]` and line numbers switch to relative numbering (current line shows absolute, others show offset).

Motions

Key	Action
`h` / `l`	Move left / right
`j` / `k`	Move down / up (actual lines)
`w` / `W`	Next word / WORD start
`e` / `E`	Next word / WORD end
`b` / `B`	Previous word / WORD start
`ge` / `gE`	Previous word / WORD end
`f{c}` / `F{c}`	Find char forward / backward
`t{c}` / `T{c}`	Till char forward / backward
`;` / `,`	Repeat / reverse last f/F/t/T
`%`	Matching bracket
`0` / `$`	Line start / end
`^`	First non-blank character
`{` / `}`	Previous / next paragraph boundary
`gg` / `G`	Top / bottom of document
`Ctrl+D` / `Ctrl+U`	Half-page down / up

All motions accept a numeric count prefix (e.g., `3j` moves down 3 lines).

Operators

Key	Action
`d`	Delete (takes a motion, e.g. `dw`); copies to clipboard
`c`	Change (takes a motion, e.g. `cw`); `cw` / `cW` stop at the word/WORD end; copies to clipboard
`y`	Yank (takes a motion, e.g. `yw`); copies to clipboard
`>`	Indent lines covered by a motion by two spaces
`<`	Dedent lines covered by a motion by up to two spaces
`gu`	Lowercase text covered by a motion or text object
`gU`	Uppercase text covered by a motion or text object
`g~`	Toggle case for text covered by a motion or text object
`D`	Delete to end of line
`C`	Change to end of line
`S`	Change entire line
`Y`	Yank entire line
`dd`	Delete entire line
`cc`	Change entire line
`yy`	Yank entire line
`>>`	Indent current line; count indents multiple lines
`<<`	Dedent current line; count dedents multiple lines
`guu`	Lowercase current line; count lowercases multiple lines
`gUU`	Uppercase current line; count uppercases multiple lines
`g~~`	Toggle case on current line; count toggles multiple lines
`dae`	Delete entire buffer (copies to clipboard)
`cae`	Change entire buffer (copies to clipboard)
`yae`	Yank entire buffer (copies to clipboard)

Text Objects

Text objects compose with `d`, `c`, and `y`.

Key	Action
`iw` / `aw`	Inner / a word
`iW` / `aW`	Inner / a WORD
`i"` / `a"`	Inner / a double-quoted string
`i'` / `a'`	Inner / a single-quoted string
`` i` `` / `` a` ``	Inner / a backtick-quoted string
`i(` / `a(`, `ib` / `ab`	Inner / a parenthesized block
`i[` / `a[`	Inner / a square-bracket block

`i{` / `a{`, `iB` / `aB`	Inner / a brace block
`i<` / `a<`	Inner / an angle-bracket block
`ip` / `ap`	Inner / a paragraph; `ap` includes adjacent blank lines
`ae`	Entire buffer

Other Commands

Key	Action
`i`	Enter INSERT mode
`v`	Enter charwise VISUAL mode
`V`	Enter linewise V-LINE mode
`a`	Append after cursor
`A`	Append at end of line
`I`	Insert at line start
`o`	Open line below
`O`	Open line above
`u`	Undo
`Ctrl+R`	Redo
`x`	Delete character
`X`	Delete character before cursor
`r{c}`	Replace character(s) at cursor (supports count: `3rx`)
`p`	Paste after cursor / below line from the internal register
`P`	Paste before cursor / above line from the internal register
`~`	Toggle case of character(s) at cursor (supports count: `5~`)
`.`	Repeat last mutation (supports count: `3.`)
`J`	Join current line with next (supports count: `5J`)

The border subtitle shows pending operators and counts (e.g., `2d` when a delete with count 2 is pending).

4.29.6 Visual Mode

Press `v` in NORMAL mode for charwise VISUAL mode, or `V` for linewise V-LINE mode. The border title shows `[VISUAL]` or `[V-LINE]`. `Escape` returns to NORMAL mode, and `o` swaps the active selection end.

Visual mode supports the NORMAL-mode motions and counts listed above, including word motions, paragraph motions, line motions, `f` / `F` / `t` / `T` with `;` / `,` repeats, `%`, `gg` / `G`, `Ctrl+D` / `Ctrl+U`, and the NORMAL-mode text objects. `v` exits charwise VISUAL mode; `V` exits V-LINE mode; pressing the other visual key switches selection kind.

`c` / `s`	Change selection and enter INSERT mode
`y`	Yank selection to the internal register and system clipboard
`p`	Replace selection with the internal register
`>` / `<`	Indent / dedent selected lines by two spaces
`u` / `U`	Lowercase / uppercase the selection
`~`	Toggle case in the selection

V-LINE operators always apply to whole selected lines regardless of the cursor column.

4.30 Prompt History Modal

Press `Ctrl+K` from the prompt input to open the prompt history modal. When the current prompt is a single logical line, that line pre-fills the modal filter. Press `,.` (leader + `.`) to open the same modal from the main ACE UI. The modal displays prompts previously run in ACE, ordered by most recent use. Prompts shorter than two words are skipped when writing to history, so trivial one-word inputs (e.g. `y`, `ok`) don't clutter the list.

Bare prompts are stored after launch normalization, so a prompt without an explicit workspace reference appears with the default `#git:home` prefix. Use `#cd:~` for direct home-directory runs with no VCS workspace management. Explicit workspace prefixes, including `#cd:<path>`, also feed the `Ctrl+N` / `Ctrl+P` MRU cycle. In the prompt input, `Ctrl+P` / `Ctrl+N` replaces the first workspace prefix in the text; when no prefix is present, it prepends the selected MRU prefix before the prompt body.

4.30.1 Keybindings

Key	Action
`Enter`	Submit the highlighted prompt directly
`Ctrl+G`	Edit first - load prompt into editor
`Tab` / `Ctrl+I`	Load prompt into the input widget for editing
`Ctrl+X`	Toggle visibility of cancelled prompts
`Ctrl+Y`	Copy prompt to clipboard
`Esc`	Close modal

4.30.2 Filtering

Type in the search box to filter prompts by text. Press `Ctrl+X` to toggle cancelled prompts on or off — when enabled, cancelled prompts appear in the results with an `x` marker.

Prompt-history rows are compact single-line entries: cancelled marker, last-used timestamp (`MM-DD HH:MM` when parseable), and a first-line prompt preview. The preview panel still shows the full prompt and timestamp metadata. History writes use a sidecar lock plus atomic tempfile replacement so concurrent agent launches do not truncate the shared `~/.sase/prompt_history.json` file.

4.31 Task Queue Modal

Press `,t` (leader + `t`) to open the task queue modal. It shows background tasks (hook runs, mentor executions, etc.) with live output for running tasks and completed output for finished ones.

4.31.1 Layout

The modal uses a two-panel layout: a task list on the left and an output pane on the right. Running tasks refresh their output every second.

4.31.2 Task Status Icons

Icon	Color	Meaning
`●`	Green	Running
`✓`	Cyan	Success
`✗`	Red	Error
`?`	Dim	Unknown

4.31.3 Keybindings

Key	Action
`j` / `k`	Navigate task list
`K`	Kill selected running task
`d`	Dismiss selected completed task
`D`	Dismiss all completed tasks
`e`	Open task output in `$EDITOR`
`y`	Copy task output to clipboard
`Ctrl+D` / `U`	Scroll output pane down / up
`q` / `Esc`	Close modal

4.32 Snippets

The prompt input supports expandable text snippets triggered by pressing `Tab`. Snippets are configured in the `ace.snippets` section of `sase.yml` as a mapping of trigger words to template strings:

```
ace:
  snippets:
    fix: "Please fix the following issue:\n$0"
    review: "Review this code for correctness, performance, and style."
    bug: "Bug in $1:\n\nExpected: $2\nActual: $3\n\nPlease fix.$0"
```

4.32.1 Usage

1. Type a trigger word (e.g., `fix`) in the prompt input.
2. Press `Tab`. If the word before the cursor matches a snippet, it is replaced with the template text.
3. If the template contains tabstop markers (`$1`, `$2`, ...), the cursor jumps to `$1` first. Press `Tab` again to advance to `$2`, then `$3`, and so on. `$0` marks the final cursor position after all tabstops are visited. If there are no tabstop markers, the cursor moves to the end of the expanded text.

Tab priority: Snippet expansion always takes priority over tabstop advancement. If you type a trigger word at an active tabstop and press `Tab`, the snippet expands rather than jumping to the next tabstop.

Multi-line indentation: When a multi-line snippet is expanded on an indented line, continuation lines automatically inherit the leading whitespace of the trigger line. Tabstop positions are adjusted accordingly.

Trigger words are matched against the alphanumeric/underscore word immediately before the cursor. If no snippet matches, `Tab` advances to the next tabstop (if any are remaining from a previous expansion), or behaves normally.

XPrompt-derived snippets compose normal xprompt references before they enter the snippet registry. Entries configured directly in `ace.snippets` remain literal snippet templates.

Editors using `sase lsp` can receive the same registry as LSP snippet completions after bare trigger words when the client advertises `completionItem.snippetSupport`. The server uses the editor helper operation `sase editor helper-bridge snippet-catalog` as the authoritative source and falls back to native Rust loading only for simple snippets if the helper is unavailable. Clients without snippet support do not receive these entries, because raw `$1` / `$0` markers would not behave like ACE tabstops.

4.32.2 XPrompt Picker (`#@`)

Typing `#@` (the `#` character followed by `@`) opens the XPrompt snippet picker modal. This lists all available xprompts (including project-local xprompts from `sase.yml` files) and inserts the selected reference at the cursor position. Inline-capable xprompts and workflows insert as `#name`; standalone workflows insert as `#!name`. The picker uses the same argument-aware skeletons as xprompt completion, so typed inputs can be filled immediately after selection. Markdown multi-agent xprompts are inline-capable and insert as `#name`. This is separate from the `ace.snippets` mechanism — it provides quick access to xprompt references rather than expanding static templates.

4.33 Auto-Refresh

ACE auto-refreshes data at a configurable interval (default: 10 seconds). The remaining time until the next refresh is shown in the info panel. Set `--refresh-interval 0` to disable.

Tab switches are instant: cached data is shown immediately while a background refresh runs asynchronously, so moving between tabs never blocks on disk I/O.

When the inotify-based artifact watcher is active, the periodic tick is **event-driven**: it consults per-surface dirty flags (`_dirty_changespecs` , `_dirty_agents` , `_dirty_axe`) and short-circuits the whole tick when nothing has changed. A 60-second `FULL_SANITY_REFRESH_SECONDS` floor still triggers a full reconcile to recover from missed inotify events, so a quiet TUI does ~zero work between real changes without going stale.

4.33.1 Performance Tracing

For diagnosing TUI latency, set `SASE_TUI_TRACE=1` before launching `sase ace` . Tracing is near-zero-cost when the env var is unset; with it enabled, each instrumented hot path emits one JSONL line per span to `~/.sase/perf/tui_trace.jsonl` (override via `SASE_TUI_TRACE_PATH=…`). See `docs/perf_runbook.md` (https://sase.sh/perf_runbook/) for the full span catalog, benchmark harness, and per-phase performance targets.

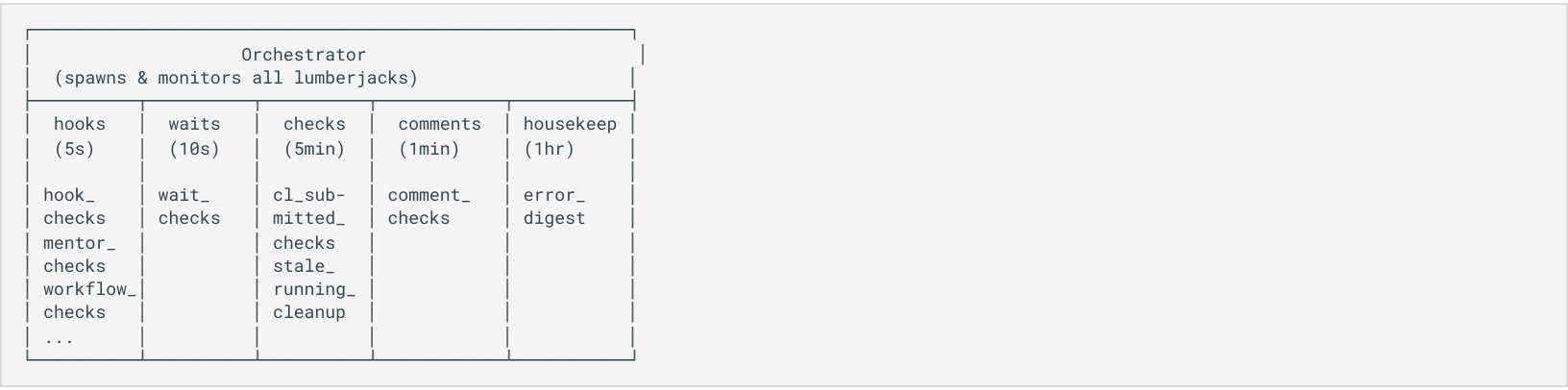
5 Axe — Background Automation Daemon

5.1 Overview

Axe is the background automation subsystem of sase. It watches ChangeSpecs (the per-CL/PR records that sase uses to track work) and periodically runs lifecycle jobs such as hook completion, mentor launch, workflow cleanup, comment polling, `%wait` dependency checks, and error digests.

Axe uses a multi-process architecture: an **Orchestrator** spawns multiple **Lumberjacks**, and each lumberjack runs a subset of jobs on its own schedule. The ACE TUI starts axe automatically unless launched with `sase ace --no-axe`; operators can also manage it directly with `sase axe start` and `sase axe stop`.

5.2 Architecture



5.2.1 Key Concepts

- **Orchestrator**: Parent process that spawns and monitors all lumberjack processes. Detects crashes and restarts failed lumberjacks automatically. Holds the axe lifecycle lock while running and forwards SIGTERM to all children on shutdown.
- **Lumberjack**: Individual scheduler loop that runs a subset of jobs on a fixed interval. Each lumberjack has a name (e.g., "hooks", "checks"), runs one or more chops per cycle, and maintains independent state and metrics.
- **Chop**: A single job unit executed by a lumberjack. Can be a script (external executable that reads context JSON) or an agent (background process launched via the agent launcher). Chops can be configured with custom environment variables and run frequency.

5.3 CLI Commands

`sase axe chop` and `sase axe lumberjack` default to their `list` views when invoked without a nested subcommand.

Command	Description
`sase axe start`	Start the orchestrator (spawns all lumberjacks)
`sase axe stop`	Stop the orchestrator gracefully
`sase axe chop list`	List configured chops
`sase axe chop run <name>`	Run a single chop in the foreground
`sase axe chop run <name> -L <lumberjack>`	Run a single chop attributed to a specific lumberjack
`sase axe lumberjack list`	List configured lumberjacks and their chops
`sase axe lumberjack run <name>`	Run a single lumberjack in the foreground

`sase axe maintenance enter`	Pause lumberjack ticks until maintenance exits
`sase axe maintenance exit`	Clear the maintenance marker
`sase axe maintenance status`	Show whether maintenance mode is active

5.3.1 Examples

```
# Start/stop the daemon
sase axe start
sase axe stop

# Run axe against only matching ChangeSpecs
sase axe start --query '!!! OR @@@'

# Inspect lumberjacks
sase axe lumberjack list
sase axe lumberjack status

# Run a single lumberjack for debugging
sase axe lumberjack run hooks

# Run a single chop once
sase axe chop run hook_checks

# Disambiguate when the same chop name appears in multiple lumberjacks
sase axe chop run hook_checks --lumberjack hooks   # -L is the short form

# Pause/resume scheduled lumberjack work
sase axe maintenance enter --reason "install plugin update"
sase axe maintenance status
sase axe maintenance exit
```

5.4 Default Lumberjacks

Axe ships with five default lumberjacks:

5.4.1 hooks (5-second interval)

High-frequency hook lifecycle management:

Chop	Description
`hook_checks`	Complete finished hooks, start stale ones
`mentor_checks`	Start mentors once hook prerequisites are met
`workflow_checks`	Complete/start CRS and fix-hook workflows
`pending_checks_poll`	Poll background check results
`comment_zombie_checks`	Mark old comment threads as ZOMBIE
`suffix_transforms`	Strip stale suffixes, update mail-readiness
`orphan_cleanup`	Release workspace claims for dead processes

5.4.2 waits (10-second interval)

Fast-polling agent dependency resolution:

Chop	Description
`wait_checks`	Resolve successful agent wait dependencies and write `ready.json`

`wait_checks` only unblocks a named dependency when the newest matching agent, or the newest matching workflow root and all of its children, has a `done.json` outcome of `"completed"`. Failed, killed, crashed, still-running, malformed, or missing `done.json` artifacts do not satisfy `%wait`; the dependent agent remains parked until a later successful run of the same dependency name appears.

5.4.3 checks (5-minute interval)

Lower-frequency status checks:

Chop	Description
`cl_submitted_checks`	Start CL submission status checks
`stale_running_cleanup`	Release workspace claims from dead processes

5.4.4 comments (1-minute interval)

Comment polling:

Chop	Description
`comment_checks`	Start critique comment checks

5.4.5 housekeeping (1-hour interval)

Periodic maintenance:

Chop	Description
`error_digest`	Send error notification digests (creates `ViewErrorReport` notification action)

The `error_digest` chop summarizes recent errors into a digest file stored at `~/.sase/axe/error_digests/digest_<timestamp>.txt`. The notification includes a `ViewErrorReport` action that opens the digest in `$EDITOR` when selected in the ACE notification modal.

5.5 Configuration

Axe is configured in `sase.yml` under the `axe:` section. See `docs/configuration.md` (https://sase.sh/configuration/) for the full configuration reference.

5.5.1 Global Settings

Setting	Default	Description

`max_hook_runners`	3	Concurrent hook runners allowed globally
`max_agent_runners`	3	Concurrent agent runners allowed globally
`zombie_timeout_seconds`	7200	Timeout for marking jobs as zombie
`query`	`""`	Optional query filter for all changespecs
`chop_script_dirs`	`[]`	Directories to search for chop scripts
`lumberjack_log_max_bytes`	52428800	Maximum bytes retained for each bounded lumberjack log
`verbose_lumberjack_diagnostics`	false	Include verbose diagnostics in chop script context JSON

The `query` setting uses the same ChangeSpec query language as ACE. CLI flags on `sase axe start` and `sase axe lumberjack run` override the configured query, runner limits, and zombie timeout for that process.

5.5.2 Lumberjack Configuration

```yaml
axe:
  lumberjacks:
    my_lumberjack:
      interval: 60 # Seconds between cycles
      chop_timeout: "60s" # Default timeout for all chops in this lumberjack
      chops:
        - name: my_chop
          description: "What this chop does"
          agent: my_agent # Optional — runs as background agent process
          run_every: "5m" # Time-based duration: run at most once per 5 minutes
          timeout: "30s" # Per-chop timeout (overrides chop_timeout)
          env:
            MY_VAR: "value" # Custom environment variables
```

Chop Fields

Field	Type	Description
`name`	`str`	Chop identifier (required)
`description`	`str`	Human-readable description (required)
`agent`	`str | null`	XPrompt/agent name — runs as a background agent process
`run_every`	`str | null`	Duration string (e.g., `"5m"`, `"2h"`) — run at most once per interval
`timeout`	`str | null`	Per-chop timeout duration (overrides the lumberjack's `chop_timeout`)
`env`	`dict[str, str]`	Custom environment variables passed to the chop

5.5.3 Script Chops

When a chop does not have an `agent` field, axe treats it as an external executable. The executable is resolved in this order:

1. An executable named exactly like the chop in one of `axe.chop_script_dirs`.
2. An executable named `sase_chop_<name>` beside the running Python interpreter.
3. An executable named `sase_chop_<name>` on `$PATH`.

Axe runs script chops as:

```
<script> --context <context.json>
```

The context file contains the effective runner limits, zombie timeout, query, lumberjack name, lumberjack state directory, and paths to serialized `all_changespecs.json` and `filtered_changespecs.json` files. Scheduled script chops within one lumberjack tick run concurrently; use `timeout` or `chop_timeout` to keep a slow script from blocking later ticks indefinitely.

Script chop stdout and stderr are streamed to the chop's per-run log file while the subprocess is still alive (see Chop Run History below). The Axe-tab dashboard tails that file so a long-running chop's output becomes visible immediately rather than only after process exit.

Chop output is part of the operator contract. Every actual chop run should write a compact, human-readable summary for both no-op and action paths. At minimum, include the chop identity or run scope, counts of inspected/skipped/updated or launched items, an explicit no-op reason, and bounded identifiers for any affected items. Agent chops should record the launched PID, prompt hash, prompt or workflow label, and enough workspace metadata to find the visible agent run. Avoid tokens, full notification bodies, full prompts, and unbounded command output in ordinary AXE logs.

5.5.4 Manual Chop Runs

Scheduled lumberjack ticks are not the only way a chop runs. Operators can launch any configured chop on demand from both the CLI and the ACE TUI; manual runs share the same execution path, run history, and live-output streaming as scheduled runs.

From the CLI:

```
sase axe chop run <chop>                      # name must be unique across lumberjacks
sase axe chop run <chop> --lumberjack <lj>    # explicit lumberjack (short form: -L <lj>)
```

When the same chop name appears under multiple lumberjacks, `sase axe chop run <chop>` fails with an unambiguous error listing the candidate lumberjacks. Pass `-L/--lumberjack` to pick one. The manual run is recorded under `~/.sase/axe/lumberjacks/<lumberjack>/chops/<chop>/` exactly like a scheduled run, except its metadata is tagged with `source = "manual"` (vs `"scheduled"`).

From the ACE TUI:

On the Axe tab, press `r` while a chop row is selected to launch that exact `(lumberjack, chop)` manually. The run uses the chop's configured environment, timeout, and (for agent chops) prompt — but bypasses any `run_every` cadence because the user explicitly asked for it. The TUI does not block while the launch happens; once the subprocess (or agent) has started, the new run becomes the newest entry in the chop's run history and the detail panel switches to it.

If the selected chop already has a live script run in flight for the same `(lumberjack, chop)`, `r` notifies and skips the launch rather than starting an overlapping duplicate. Agent chops keep the existing live-agent dedupe semantics (prompt-hash based). On non-chop rows — lumberjack rows and running bgcmd rows — `r` is a no-op; on a completed bgcmd row, `r` continues to re-run the bgcmd.

Manual runs participate in `Ctrl+N` / `Ctrl+P` history navigation just like scheduled runs. The chop-detail header marks them with a `Source: manual` chip so it is easy to tell at a glance why a run started.

5.5.5 Agent Chops and Visibility

When a chop has an `agent` field, axe launches a real background agent (via the same launcher as `sase run`) instead of shelling out to a chop script. Configured agent chops are **visible by default** in the Agents tab — recurring infra agents (e.g. orchestration housekeepers) show up alongside user-launched agents so their state, retries, and last output are discoverable. Use the prompt-side `%hide` directive if a particular chop should remain hidden.

Specialized review agents launched by axe runners (mentor, CRS, fix-hook, and summarize-hook review agents) write Agents-tab metadata as well. They persist the same `review` tag that a `%group:review` prompt would produce, so ACE groups them in the `@review` side panel instead of hiding them among untagged automation rows.

During a scheduled lumberjack tick, script chops are still dispatched concurrently, but eligible agent chops are launched sequentially in their configured order. That keeps multiple same-tick `run_every` agent chops from racing each other for workspace allocation while preserving parallel script execution.

`sase axe chop run <agent-chop>` follows the same path as the scheduled lumberjack tick, so a one-shot run records the same chop registry metadata as the periodic invocation.

5.5.6 Chop Run History

Every chop execution — whether kicked off by a scheduled lumberjack tick or by `sase axe chop run …` — is recorded as a separate run under `~/.sase/axe/lumberjacks/<lumberjack>/chops/<chop>/`. Each run is assigned a sortable, microsecond- precision `run_id` and persisted as a pair of files in a shared `runs/` directory. `index.json` (kept next to `runs/`) lists the chop's run IDs newest-first:

```
~/.sase/axe/lumberjacks/<lumberjack>/chops/<chop>/
├── index.json              # Ordered run IDs (newest first)
└── runs/
    ├── <run_id>.json       # Run metadata (see below)
    └── <run_id>.log        # Streamed stdout+stderr from the chop process
```

Each `<run_id>.json` is a serialized `ChopRunEntry` (see `src/sase/axe/state.py`). The most relevant fields are `status`, `started_at`, `finished_at`, `duration_ms`, `exit_code`, `pid` (script chops) / `agent_pid` (agent chops), `source` (`scheduled`, `manual`, or `oneshot`), `started_by`, and `output_bytes`.

A run is created in `running` state before the subprocess is launched. On exit it is updated in place with a terminal status — `success`, `failure`, `timeout`, `missing_script`, or `agent_launched` (the last is used when a chop only launches a background agent rather than running a script). `finished_at` is `null` while the run is still active.

History is pruned after every run write, retaining the newest `MAX_CHOP_RUN_HISTORY` (10) terminal runs per chop. Runs still in `running` state are always kept regardless of position, so a slow chop is never deleted out from under its own process.

5.5.7 AXE Tab Views

The Axe tab sidebar renders each lumberjack as a top-level row with its configured chops as indented children, followed by any background commands (`!!`). Each chop row shows a status marker derived from its newest cached run: `[●]` while a run is active, `[✓]` for the most recent `success`, `[!]` for `failure` or `timeout`, `[?]` for `missing_script`, `[*]` for `agent_launched` (agent chops that only spawn a background agent), and `[·]` for chops that have never run. Selection drives three distinct dashboard views:

- **Lumberjack overview** — selecting a lumberjack row shows its status, interval, cycle count, error count, and a per-chop table with each chop's last-run status, relative timestamp, and duration. For a chop whose newest run is still active, the duration column shows live elapsed runtime rather than the stale `0ms` you would otherwise see before the run finalizes.
- **Chop detail** — selecting a chop row renders the latest run's metadata (`● running` status with live elapsed runtime, PID, and a `Source:` chip for non-scheduled runs — i.e. `manual` or `oneshot`) and tails the run's `.log` file. Until the log has accumulated any bytes, the panel shows a `Waiting for output…` placeholder; the exit code is suppressed until the run finalizes.
- **Background command output** — the existing live output stream for the focused `!!` row.

`Ctrl+N` / `Ctrl+P` on the Axe tab page through the focused chop's run history (newer / older). The viewer pins to the run you selected so that a fresh tick prepending a new run does not bump you forward; the pin is cleared automatically if the pinned run is pruned or itself becomes the newest run.

5.5.8 Chop-Agent Registry

Each lumberjack maintains a durable JSON registry of the agents it has launched at `~/.sase/axe/lumberjacks/<name>/agent_chops.json`. Records carry the lumberjack/chop names, a normalized `prompt_hash`, the launched PID, the agent's artifacts timestamp, and a per-launch UUID. The registry is what lets a recurring chop dedup against an in-flight run with the same prompt body, survive lumberjack restarts (sase-12 perf overhaul), and be reattributed correctly in the Agents tab. The metadata is also propagated into the agent's `agent_meta.json` via the env vars `SASE_CHOP_LUMBERJACK`, `SASE_CHOP_NAME`, `SASE_CHOP_RUN_ID`, and `SASE_CHOP_PROMPT_HASH` (see `build_chop_launch_env()` in `src/sase/axe/chop_agents.py`).

5.6 Concurrency Management

Axe uses a cross-process runner pool to enforce global concurrency limits. The `SharedRunnerPool` uses `fcntl.flock` on a shared file (`~/.sase/axe/shared/runner_count`) to coordinate runner slots across all lumberjack processes atomically.

Hook runners and agent runners have separate limits (`max_hook_runners` and `max_agent_runners`), allowing fine-grained control over background resource usage.

5.7 Agent Completion Artifacts

When an agent run finalizes, axe writes the normal completion metadata and sends the workflow-complete notification. Successful runs also scan the agent workspace for generated image files (`.png`, `.jpg`, `.jpeg`, `.webp`, `.gif`) and Markdown files (`.md`, `.markdown`). When 10 or fewer Markdown sources are discovered after filtering, they are rendered to PDFs under the agent artifact directory, then the generated PDF paths are appended after the standard chat/diff notification attachments and before image attachments. The PDF list is persisted as `done.json.markdown_pdf_paths`; the image list is persisted as `done.json.image_paths`. Explicit artifacts created during the run with `sase artifact create -p <path> [-n <label>] [-k <kind>]` are appended after image attachments when their stored files still exist.

The scan uses git name-status output, untracked files, saved diff metadata, and the latest commit when the agent committed or opened a PR. Deleted, missing, unsupported, and duplicate paths are ignored. If more than 10 Markdown sources remain, Axe skips Markdown PDF rendering for that completion and adds a note to the notification. PDF rendering is otherwise best-effort: missing conversion tools or render failures omit that source without failing the agent run. Generated Markdown PDFs are optimized for narrow viewers with a small portrait page, small margins, and larger type. As PDFs are prepared, axe updates `workflow_state.json.pdf_status` and a compact `activity` label so ACE can show live finalization progress such as `PDF 2/4 <path>` or `PDFs done 3/4 (1 skipped)`. Successful runs also copy discovered image artifacts, including prompt-referenced images, into persistent SASE artifact storage for ACE. Prompt-referenced images are not appended to completion notifications unless they were also generated/modified files or explicit artifacts. See `agent_images.md` (https://sase.sh/agent_images/) for the full contract.

The Agents tab exposes completion artifacts through the `A` action. When artifacts exist, ACE opens the artifact panel for selection. Chat transcripts, plan files, generated PDFs/images, prompt-referenced images from saved prompt artifacts, and explicit artifacts created with `sase artifact create -p <path> [-n <label>] [-k <kind>]` all participate in the same list. Explicit artifacts are stored under `~/.sase/artifacts/` with a persistent association so they remain available after dismissing and later reviving the agent. ACE shows the picker even for a single artifact; `m` marks rows, `Enter` opens the marked set or highlighted row, and `A` opens the full list. Only one plan artifact is listed for an agent, preferring the committed SDD plan path when one exists. Inside tmux, artifact viewing opens in a right-side tmux pane, collapses the Agents list while live, uses `l` to focus the pane, and uses `A` to close it; outside tmux, ACE suspends and uses the current pane. The viewer supports images, Markdown, and PDFs, wraps `j`/`k` page navigation at the ends, uses `n`/`p` for artifact-sequence navigation, and warns when required terminal/rendering tools are missing. The direct agent run-log binding is `V`.

5.8 Maintenance Mode

Maintenance mode is a lightweight pause switch for scheduled axe work. `sase axe maintenance enter --reason <text>` writes `~/.sase/axe/maintenance.json` with the reason, caller PID, and start timestamp. Each lumberjack checks that marker at the start of every tick; while it is active, the lumberjack records a cycle and skips the chop execution for that tick.

Use maintenance mode before operations that temporarily make scheduled work unsafe or noisy, such as installing plugin updates, moving workspace directories, or running one-off cleanup. `sase axe maintenance exit` removes the marker. `sase axe maintenance status` exits 0 when active and 1 when inactive, so scripts can use it as a

guard. The next lumberjack tick clears stale markers automatically when they are older than 24 hours, malformed, or owned by a PID that is no longer running.

5.9 State Directory

```
~/.sase/axe/
├── orchestrator.pid               # Orchestrator PID
├── orchestrator.lock              # Exclusive lifecycle lock held by the live orchestrator
├── maintenance.json               # Optional maintenance marker that pauses lumberjack ticks
├── logs/
│   ├── axe.log                    # Orchestrator startup log
│   └── lumberjack-{name}.log      # Per-lumberjack logs
├── lumberjacks/
│   └── {name}/                    # Per-lumberjack state
│       ├── pid                    # Lumberjack PID
│       ├── status.json            # Current status (updated every 5s)
│       ├── metrics.json           # Cumulative metrics (updated every 30s)
│       ├── chop_timestamps.json   # Last successful run_every timestamp per chop
│       ├── agent_chops.json       # Durable registry of agents launched by this lumberjack's chops
│       ├── chops/                 # Per-chop run history (newest 10 terminal runs per chop)
│       │   └── {chop}/
│       │       ├── index.json     # Ordered run IDs (newest first)
│       │       └── runs/
│       │           ├── {run_id}.json   # ChopRunEntry metadata
│       │           └── {run_id}.log    # Streamed stdout+stderr
│       ├── tick/
│       │   ├── context.json       # Context passed to script chops
│       │   ├── all_changespecs.json
│       │   └── filtered_changespecs.json
│       └── logs/
│           └── output.log         # Lumberjack output log
├── shared/
│   └── runner_count               # Cross-process runner counter
├── error_digests/                  # Error digest files for ViewErrorReport
│   └── digest_<timestamp>.txt     # Summarized error reports
└── recent_errors.json             # Last 100 errors encountered
```

5.10 Process Lifecycle

1. `sase axe start` first checks for a live orchestrator PID. If one exists, start is a no-op and returns the existing PID.
2. If no live PID exists, startup acquires `~/.sase/axe/orchestrator.lock` and hands that lock to the detached orchestrator process. Concurrent starts wait briefly and then return the live PID or decline to start.
3. The orchestrator removes stale PID files, adopts/holds the lifecycle lock, writes `orchestrator.pid`, and spawns all configured lumberjacks as child processes.
4. Each lumberjack runs its chops on its configured interval, unless maintenance mode is active.
5. The orchestrator monitors children and restarts any that exit unexpectedly.
6. `sase axe stop` sends SIGTERM to the orchestrator, which forwards it to all children. If the orchestrator does not exit within the stop timeout, the stopper escalates to SIGKILL and cleans up stale PID files.

5.11 ACE Integration

The Axe tab in the ACE TUI provides live monitoring of the daemon:

- A lumberjack tree sidebar (lumberjack rows + their chops as children + background-command rows)

- A lumberjack overview, per-chop detail view, and run-history pager (see AXE Tab Views)
- Start/stop the orchestrator (`x` key or `!x`) and runner counts
- Footer shows daemon status: RUNNING, STOPPED, STARTING, STOPPING, or RESTARTING

The RESTARTING indicator appears when `sase ace --restart-axe` (`-R`) is used — the daemon restarts in the background while the TUI starts up normally.

See `docs/ace.md` (https://sase.sh/ace/) for the full Axe tab keybinding reference.

6 Spec-Driven Development (SDD)

SDD is sase's system for persisting the intent behind agent work. When an agent submits a plan for approval, SDD can capture both the expanded prompt snapshot and the approved planning artifact, creating a traceable chain from intent to execution. In this guide, "plan-like artifact" means a tale, epic, or legend.

6.1 Why SDD Exists

Agent plans are ephemeral by default -- they live in a single session's context window and vanish when the session ends. SDD fixes this by writing prompt snapshots and plans to disk as first-class artifacts:

- **Prompts** record the full expanded prompt the agent received, so the "why" behind the work is preserved.
- **Tales** record ordinary approved implementation plans, so decomposition decisions are queryable after the fact.
- **Epics** record executable multi-phase plans that can be handed to `sase bead work`.
- **Legends** record higher-level coordination plans that can own linked epics.
- **Myths** record long-horizon narrative, strategy, and context that is broader than active roadmap plans.
- **Research** records exploratory findings, prior art, options, critiques, and recommendations that inform later work.
- **Beads** provide structured issue tracking that links SDD artifacts to execution via plan-like bead tiers and phase IDs in commit messages.

Together, these create an audit trail from prompt snapshots to planning artifacts and supporting context. Tales, epics, and legends can link into the bead hierarchy and phase commits; myths and research notes preserve the longer-lived context those plans depend on.

6.2 Storage Modes

SDD supports two storage modes. For non-bare-git projects the mode is controlled by the `sdd.version_controlled` config option. Projects resolved as the built-in `bare_git` VCS provider always use version-controlled SDD under `sdd/`, even when the merged config leaves `sdd.version_controlled` false.

6.2.1 Local Mode (default for non-bare-git projects: `sdd.version_controlled: false`)

Files are stored in a standalone git repo inside the primary workspace:

```
{primary_workspace}/.sase/sdd/
  .git/                     # Standalone git repo for SDD tracking
  .gitignore                # Ignores beads.db
  prompts/
    {YYYYMM}/
      {plan_name}.md        # Expanded prompt (xprompts resolved, directives stripped)
  tales/
    {YYYYMM}/
      {plan_name}.md        # Normal non-epic implementation plans
  epics/
    {YYYYMM}/
      {plan_name}.md        # Executable multi-phase epic plans
  legends/
    {YYYYMM}/
      {plan_name}.md        # Higher-level coordination plans
  myths/
    README.md               # Generated directory guide
  research/
    README.md
    {YYYYMM}/
      {note_name}.md        # Research notes and critiques
  beads/                    # Bead store (canonical events + compatibility mirrors)
    config.json
    events/
      manifest.json
      streams/
        <root-id>.jsonl
    issues.jsonl
    beads.db
```

SDD auto-commits prompt and planning-artifact files to this local repo after each planning phase. The standalone repo keeps SDD history separate from the project's own git history.

6.2.2 Version-Controlled Mode (`sdd.version_controlled: true`, or any bare-git project)

Files are stored at the project root and tracked in the project's own git repo:

```
{project_root}/
  sdd/
    prompts/
      {YYYYMM}/
        {plan_name}.md
    tales/
      {YYYYMM}/
        {plan_name}.md
    epics/
      {YYYYMM}/
        {plan_name}.md
    legends/
      {YYYYMM}/
        {plan_name}.md
    myths/
      README.md
    research/
      README.md
      {YYYYMM}/
        {note_name}.md
  sdd/beads/              # Bead store (canonical events + compatibility mirrors)
    config.json
    events/
      manifest.json
      streams/
        <root-id>.jsonl
    issues.jsonl
    beads.db
```

In this mode, SDD artifacts are committed alongside code changes via `sase commit`.

For built-in bare-git projects, SASE also creates or refreshes the generated SDD guide files automatically. First-use `#git:<project>` initialization includes them in the initial commit; existing bare-repo registration, `#git` materialization, and `sase workspace open` commit and push an `Initialize SDD` init commit when the generated files are missing or stale. First SDD writes, plan archiving, and `sase bead init` also refresh the generated files before writing project-local SDD content.

Research notes live under `sdd/research/{YYYYMM}/` alongside the rest of the repository-local SDD corpus. The built-in `#research` xprompt tells the agent to create a new markdown file in the current month directory; `sase sdd` does not write research files automatically.

The directory examples above show the storage roots. Most frontmatter links include the root prefix when the root is well-known: `sdd/...` in version-controlled mode and `.sase/sdd/...` in local mode.

6.3 How SDD Works

6.3.1 Prompt Generation

When a submitted plan is accepted, SDD generates a prompt snapshot by:

1. Expanding all `#xprompt` references in the original prompt
2. Stripping `%directives` (`%model`, `%name`, `%wait`, etc.)
3. Dry-expanding embedded workflow `prompt_part` content (renders templates without executing pre/post steps)

The result is a clean, self-contained document showing exactly what the agent was asked to do.

6.3.2 Artifact Persistence

The plan file produced by the agent is:

1. Annotated with a `create_time` frontmatter field
2. Written to the action-specific SDD directory, where `{YYYYMM}` is derived from the current date. Version-controlled paths look like:
3. normal approval: `sdd/tales/{YYYYMM}/{plan_name}.md`
4. epic approval: `sdd/epics/{YYYYMM}/{plan_name}.md`
5. legend approval: `sdd/legends/{YYYYMM}/{plan_name}.md`

Prompt snapshots, plans, and research notes are organized into `YYYYMM` subdirectories (for example, `202603/`) based on the creation date. This keeps the directories manageable as the number of prompts, plans, and research artifacts grows over time. Both flat and `YYYYMM` layouts are supported for backwards compatibility — SDD also searches legacy `specs` and `plans` paths when resolving files.

Planning artifacts may also carry a `status` field (set to `done` when work completes) and a `bead_id` field linking to the bead issue tracker. Legend artifacts use `legend_bead_id` for the legend container bead and `epic_count` for the number of proposed epics; epics linked under a legend also preserve that `legend_bead_id`.

When `sase plan propose` submits a plan for approval, it touches `~/.sase/.ace_refresh_pulse` so any running ACE TUI flips the agent into the `PLAN` status immediately rather than waiting for the next auto-refresh tick. The pulse file is consumed by the inotify-based artifact watcher and is harmless when no TUI is open.

Humans can approve the pending proposal from ACE or from the CLI. `sase plan` lists pending PlanApproval notifications, recent approvals, and inferred rejected archived proposals. `sase plan approve <id-prefix> --kind tale|epic|legend` writes the same approval response as the TUI and tells the runner to commit the promoted plan under the matching SDD tier before launching the follow-up. `--kind approve` runs the coder without committing an SDD plan, while `--kind commit` records the approved plan in SDD without launching a coder.

To recall prior plans, `sase plan search [QUERY]` searches the committed `sdd/` plans (surfaced first) and the machine-local `~/.sase/plans/` archive by content. The query is optional — omit it to browse and filter with `--kind`, `--status`, `--source`, and `--since` / `--until` date bounds. Results are ranked (relevance with a query, recency without) and render as colored `compact` / `full` output or as agent-friendly `json` / `markdown` via `--format`.

6.3.3 Q&A Sections

If the agent asks clarifying questions during planning (via the `/sase_questions` skill), the Q&A exchange is appended to the prompt snapshot so the full context of planning decisions is preserved.

Multi-round Q&A is rendered as a single merged `### Questions and Answers` section with monotonic `Q1..QN` numbering across all rounds (a second round of questions continues at the next free number rather than restarting at `Q1`). The section is wrapped in exactly one `%xprompts_enabled` pair regardless of round count, and follow-up writes strip any prior Q&A block (including legacy duplicate blocks from older runs) before re-emitting the merged section. When a round carries a global note the "last non-empty wins" rule applies — a later round's note replaces the earlier one, but an empty later note preserves the earlier value.

6.3.4 Frontmatter Links

Prompt snapshots and plan-like artifacts link to each other through YAML frontmatter:

```
# sdd/prompts/202605/example.md
plan: sdd/tales/202605/example.md

# sdd/tales/202605/example.md
prompt: sdd/prompts/202605/example.md
```

`sase sdd validate` checks these bidirectional links for prompts, tales, epics, and legends. It treats unpaired historical files as warnings by default and as errors with `--strict`. Myths and research notes are durable SDD context, but they are not part of the prompt-plan link validator.

6.3.5 Model Field

Plan files may carry an optional top-level `model:` field in YAML frontmatter to record the model the work should run under. The value uses the same syntax `%model` accepts: a bare known model name (e.g. `opus`), a provider-qualified id (e.g. `codex/gpt-5.5`), or a configured local alias (e.g. `#pro`).

```
# sdd/tales/202605/example.md
prompt: sdd/prompts/202605/example.md
model: opus
```

Epic plan files can additionally annotate individual phases with their own `model:` lines so different phases can be worked by different models. The `bd/new_epic` xprompt forwards the top-level `model:` field to `sase bead create`'s `-m/--model` flag on the epic plan bead (so the land agent inherits it) and forwards each phase's `model:` annotation to that phase bead's `-m/--model` flag. The `bd/new_legend` xprompt forwards only the top-level `model:` to the legend plan bead (the legend's land-legend agent inherits it); legend plans propose epics rather than phases, so per-phase model assignment happens later when each epic is split via `bd/new_epic`. When the field is absent, `--model` is omitted and the bead falls back to the launcher default.

6.4 CLI

The `sase sdd` command group manages generated SDD documentation and frontmatter links:

With no subcommand, `sase sdd` defaults to `sase sdd list` with default options. Use the explicit `sase sdd list` form when passing list flags such as `--kind` or `--json`.

Command	Purpose
`sase sdd init`	Enable `sdd.version_controlled`, then refresh `sdd/README.md`, tier READMEs, and the directory map
`sase init sdd`	Alias for `sase sdd init`; accepts the same `-p/--path` option
`sase sdd list`	List SDD markdown files; `-k/--kind` filters to `prompts`, `tales`, `epics`, `legends`, or `all`
`sase sdd links`	Print each prompt/artifact frontmatter link and whether its reverse link is intact
`sase sdd validate`	Validate frontmatter links; `-j/--json`, `-q/--quiet`, `--strict`, and `-W/--show-warnings` tune output
`sase sdd repair-links`	Infer unambiguous prompt/artifact pairs; add `-w/--write` to update files

Each subcommand accepts `-p/--path`, which may point at an SDD root or a project root. Validation treats unpaired or ambiguous historical files as warnings by default and promotes them to errors with `--strict`; parse errors, missing targets, wrong link kinds, and broken reverse links are errors unless explicitly allowlisted for legacy migration.

`sase sdd validate` hides warning-severity issues from its text output by default — the summary line still reports the warning count and appends `(use --show-warnings to display)` so they remain discoverable without scrolling through noise on the happy path. Pass `-W/--show-warnings` to print each warning, or `--strict` to promote warnings to errors before filtering. JSON mode (`-j/--json`) and exit codes are unaffected by `-W`.

The `sase sdd init` command enables version-controlled SDD in the project-local `sase.yml`, then refreshes `sdd/README.md`, the directory map asset, and generated `README.md` files in `tales/`, `epics/`, `legends/`, `myths/`, and `research/`. Keep conceptual details here in `docs/sdd.md`; use `sase sdd init` to opt into project-local SDD and refresh generated project guides. The generated guides are safe to overwrite, so do not put hand-maintained conceptual prose in those README files.

Bare-git projects normally do not need a manual `sase sdd init`: SASE runs the same generated-file refresh during repository setup, workspace materialization, and the first version-controlled SDD write. The explicit command remains useful for manual refreshes and `--check` drift audits.

6.5 Bead Integration

SDD initializes the bead issue tracker (https://sase.sh/beads/) automatically when an epic agent spawns:

- **Local mode**: Beads are stored in `.sase/sdd/beads/`; `.sase/sdd/` is a standalone git repo and bead storage is initialized through SASE's built-in bead project bootstrap
- **VC mode**: Beads are stored in `sdd/beads/` at the project root

Plan-like beads carry a `tier` value:

- `plan` for ordinary non-epic implementation plans.
- `epic` for executable multi-phase plans.
- `legend` for higher-level coordination plans.

For larger efforts, epic files carry `bead_id` and `tier: epic` in their frontmatter. Each phase of the epic gets its own bead whose ID appears in commit messages, creating a traceable chain from epic to phase to commit. Legend files carry `legend_bead_id`, `tier: legend`, and `epic_count`; linked epics also include `legend_bead_id`. For smaller plans, commit messages include a `PLAN=<path>` tag pointing back to the plan file.

Linked epics are created as ordinary plan beads with a legend parent:

```
sase bead create --title "<title>" --type plan(sdd/epics/202605/example.md,<legend_bead_id>) --tier epic
```

Legend beads are executable kickoff points for their proposed epics. `sase bead work <legend_bead_id>` launches one epic-planning agent per stored `epic_count`; it does not create phase beads directly.

When the plan approval flow launches an epic agent, SASE passes the epic-creation xprompt a plan reference that all workspaces can resolve. In version-controlled mode this is the project-relative `sdd/epics/{YYYYMM}/{name}.md` path. In local mode it is the primary-workspace-relative `.sase/sdd/epics/{YYYYMM}/{name}.md` path. If an older flat plan layout is encountered, the resolver still checks both canonical and legacy flat/ `YYYYMM` locations for backwards compatibility.

6.6 Configuration

```yaml
sdd:
  version_controlled: false # default
```

Option	Type	Default	Description
`sdd.version_controlled`	bool	`false`	For non-bare-git projects, store SDD artifacts and beads under `sdd/` in the project repo instead of `.sase/sdd/` in the primary workspace

See `configuration.md` (https://sase.sh/configuration/) for the full configuration reference.

6.7 Multi-Workspace Behavior

SDD artifact placement follows the configured SDD mode and project workflow. In version-controlled mode, bead commands read and write the current checkout's `sdd/beads/` store; they do not merge bead records from numbered sibling workspaces. Coordinate bead state between checkouts through the normal VCS sync path.

7 XPrompt Template Reference

XPrompts are reusable prompt templates with optional typed inputs and Jinja2 support. They let you define a prompt fragment once and reference it by name anywhere a prompt is composed, keeping prompts DRY and consistent across projects. Inline prompt fragments use `#name`; standalone workflows use `#!name` when they are launched as workflows.

Use xprompts when you want to:

- Share common instructions across multiple prompts (e.g., output format rules, role definitions).
- Parameterize prompts with typed, validated arguments.
- Compose prompts from smaller building blocks using `#name(args)` syntax.

There are two related paths to keep separate:

```
launch setup:
  multi-agent xprompt fan-out check
  -> default workspace ref insertion when needed (#git:home)
  -> project alias canonicalization (#gh:bob -> #gh:bob-cli)
  -> workspace ref resolution (#cd/#git/#gh/#hg and known-project fallbacks)
  -> prompt/workflow execution

xprompt expansion inside a prompt or prompt_part:
  alias substitution
  -> fenced-block and disabled-region protection
  -> iterative reference expansion (parse -> lookup -> args -> render -> substitute)
  -> directive extraction at the launch or workflow-step boundary
```

The checked-in infographic prompt in `docs/images/xprompt-resolution-infographic.prompt.md` tracks the intended visual version of this model; the text model above is the authoritative current reference for resolver order.

7.1 Table of Contents

- CLI Subcommands
- Editor LSP
- Discovery Order
- File Format
- Reference Syntax
- Arguments
- Shorthand Syntax
- Typed Inputs
- Output Specification
- Jinja2 Integration
- Legacy Placeholders
- Tags
- Snippet Field
- Skill Field
- Bundled Skills
- Built-in XPrompts
- Config-Based XPrompts
- Local Configuration Files

- Directives
- Command Substitution
- Protected Content
- XPrompt Aliases
- Recursive Expansion
- Multi-Agent Prompts
- Multi-Agent XPrompts (Library-Defined Fan-Out)
- Relationship to Workflows

7.2 CLI Subcommands

The `sase xprompt` command provides five subcommands for working with xprompts. With no subcommand, it defaults to `sase xprompt list`. Flags belong to the explicit subcommand, so use forms like `sase xprompt expand --trace '#plan'` rather than putting `--trace` on bare `sase xprompt`.

7.2.1 `sase xprompt expand`

Expands xprompt references in a prompt. Reads from a positional argument or stdin.

```
sase xprompt expand '#greet(Alice)'         # Expand from argument
echo '#greet(Alice)' | sase xprompt expand  # Expand from stdin
sase xprompt expand --trace '#plan'         # Show expansion trace on stderr
```

The `--trace` flag prints a detailed expansion trace to stderr showing each resolved reference, its source file, arguments, and expanded content. This is useful for debugging reference resolution order and understanding how a complex prompt is assembled.

7.2.2 `sase xprompt explain`

Shows a dry-run visualization of a workflow's execution plan without actually running it. Displays workflow metadata, input requirements, resolved arguments, and the full step-by-step execution plan with types, control flow annotations, rendered step bodies, and output schemas.

```
sase xprompt explain my_workflow                  # Explain with no args
sase xprompt explain my_workflow arg1 arg2        # With positional args
sase xprompt explain my_workflow --arg key=value  # With named args
```

7.2.3 `sase xprompt list`

Lists all available xprompts and workflows as a JSON array. Each entry includes the name, type (`"xprompt"` or `"workflow"`), kind, reference prefix, insertion text, `is_skill`, source file path, user-facing input definitions, tags, and a content preview. Clients should treat `insertion` as the authoritative reference text. Most `xprompt` and `embeddable_workflow` entries insert as `#name`, including markdown multi-agent xprompts; standalone workflows

insert as `#!name`. `is_skill` is `true` only for xprompt catalog entries marked as skills; workflows report `false`. Step inputs are omitted from the JSON `inputs` array because they are supplied by workflow execution rather than typed by a user.

```
sase xprompt list                  # JSON array to stdout
sase xprompt list | jq '.[].name'  # Extract just names
```

7.2.4 `sase xprompt graph`

Generates a directed acyclic graph (DAG) visualization of a workflow. Without a workflow name, lists all available multi-step workflows with their step counts and source paths.

```
sase xprompt graph                       # List all workflows
sase xprompt graph my_workflow           # Mermaid DAG (default)
sase xprompt graph my_workflow --format text  # Plain-text summary
```

The Mermaid output can be pasted into any Mermaid-compatible renderer. Parallel sub-steps are shown as subgraphs, and nodes include type indicators and control flow annotations.

7.2.5 `sase xprompt catalog`

Renders every visible xprompt to a formatted PDF catalog for browsing and sharing.

```
sase xprompt catalog               # Write the PDF to a tempdir and print its path
sase xprompt catalog --out /tmp/out # Write the PDF to the specified directory
```

The command collects all visible xprompt templates, renders each into an HTML section, and produces a single PDF using the bundled `catalog_template.html.j2` and `catalog_style.css`. The mobile/helper structured catalog uses the same collection and classification code, but returns JSON metadata instead of requiring a PDF renderer.

7.3 Editor LSP

`sase lsp` starts the SASE xprompt language server over stdio for editor integrations. It resolves the server command in this order:

1. `SASE_XPROMPT_LSP_CMD`, parsed as a shell-style command for development.
2. `sase-xprompt-lsp` on `PATH`.
3. A debug or release `sase-xprompt-lsp` binary under a sibling `../sase-core` checkout.
4. `cargo run --manifest-path ../sase-core/Cargo.toml -p sase_xprompt_lsp --` when `cargo` is available and the sibling checkout has a `Cargo.toml`.

Examples:

```
sase lsp
sase lsp --version
SASE_XPROMPT_LSP_CMD='cargo run --manifest-path ../sase-core/Cargo.toml -p sase_xprompt_lsp --' sase lsp
```

Use `SASE_XPROMPT_LSP_CMD` for any non-default LSP command. `SASE_CORE_DIR` is a `Justfile` build/install override, not part of `sase lsp` command resolution.

The LSP loads the supported xprompt catalog sources directly in Rust for completion, hover, diagnostics, and definition requests. `sase lsp` exports the installed package xprompt paths to the server so built-in Markdown prompts, YAML workflows, default config prompts, project-local prompts, user config prompts, and memory prompts do not require a Python helper subprocess on the completion path. The Python helper bridge remains stable for mobile clients and as a compatibility fallback for sources the Rust loader cannot discover.

When the editor advertises LSP `completionItem.snippetSupport`, the server also returns SASE snippets as ordinary `CompletionItemKind.Snippet` entries after bare trigger words such as `fix` or `review`. Snippet entries are loaded from the same registry as ACE: xprompts with `snippet` front matter plus user-defined `ace.snippets`, with `ace.snippets` winning on trigger collisions. The editor does not need to shell out or parse SASE config to discover snippets.

The Python helper operation `sase editor helper-bridge snippet-catalog` is the authoritative snippet registry because it matches ACE's xprompt composition behavior. The Rust server also has a native fallback for simple xprompt snippets and `ace.snippets` so completion can degrade gracefully if the helper is unavailable. That fallback intentionally skips xprompts that require complex Jinja or composition it cannot mirror exactly; when the helper is available, its response is preferred.

See the editor integration guide (https://sase.sh/editor/) for setup, feature coverage, helper bridge usage, and troubleshooting.

7.4 Discovery Order

Markdown xprompts are loaded from multiple locations. When two locations define an xprompt with the same name, the higher-priority source wins (first-wins).

Priority	Location	Notes
1	`.xprompts/` (CWD, hidden dir)	Highest priority; project-local overrides
2	`xprompts/` (CWD)	Non-hidden variant
3	`~/.xprompts/` (home, hidden dir)	User-wide overrides
4	`~/xprompts/` (home)	Non-hidden variant
5	`~/.config/sase/xprompts/{project}/`	Project-specific (when project is set)
6	`sase.yml` `xprompts:` section	Config-based definitions (local + global)
7	Plugin packages (`sase_xprompts` EPs)	Installed plugin xprompts
8	`<sase_package>/default_xprompts/*.md`	Built-in default markdown xprompts
9	`<sase_package>/xprompts/*.md`	Built-in package xprompts shipped with core SASE

Each directory-based source can contain individual `.md` files. YAML workflows use the same CWD, home, project, plugin, and package locations, with `.yml` or `.yaml` files loaded as workflow definitions. Within priority 6, the config merge chain applies: built-in defaults, plugin configs, `~/.config/sase/sase.yml`, overlay files (`sase_*.yml`), and finally a local `./sase.yml` in the current working directory (highest config priority).

For file-based xprompts (priorities 1-5 and 7-9), the xprompt name defaults to the filename stem (e.g., `summarize.md` defines the xprompt `summarize`). The name can be overridden via the `name` field in the YAML front matter.

Project-specific xprompts (priority 5) are namespaced: a file `bar.md` in the `foo` project directory becomes `foo/bar`. Inline-capable project xprompts are referenced as `#foo/bar`; standalone project workflows are referenced as `#!foo/bar`.

When a project is detected (via the workspace provider), CWD xprompts (priorities 1-2) and local config xprompts are also auto-namespaced with the `{project}/` prefix. For example, if the project is `myapp` and `xprompts/deploy.md` exists in the CWD, it becomes `myapp/deploy` and is referenced as `#myapp/deploy`. A project workflow with no `prompt_part` would instead be launched as `#!myapp/deploy`. This prevents name collisions between project-local xprompts and global or built-in ones.

7.5 File Format

An xprompt file is a Markdown file with optional YAML front matter delimited by `---` lines. Everything after the closing `---` is the template body.

```
---
name: greet
description: Greet a named user.
input:
  user_name:
    type: word
    description: User name to include in the greeting.
---

Hello, {{ user_name }}! Welcome aboard.
```

7.5.1 Front Matter Fields

Field	Required	Description
`name`	No	XPrompt name (defaults to filename stem)
`input`	No	Input parameter definitions (see Typed Inputs)
`snippet`	No	Opt-in to ACE snippet expansion (see Snippet Field below)
`description`	No	Human-readable one-line description of what the xprompt does
`skill`	No	Marks this xprompt as an agent skill source for `sase skill init` (see below)
`xprompts`	No	File-local helper xprompts whose names must start with `_`

If no front matter is present, the entire file content is the template body and the filename stem is the name.

Markdown xprompt files can carry file-local helper xprompts under `xprompts:`. These helpers use the same structured format as config-based xprompts, including typed inputs and descriptions, and they can reference each other transitively. During expansion they inherit the containing xprompt's arguments and template scope, so a helper can use values such as `{{ topic }}` from the outer xprompt. They are visible only while expanding the containing xprompt and must use `_`-prefixed names such as `_review_rules`; they do not leak into the global catalog,

completion catalog, or other xprompt files. This underscore rule also applies to local xprompts in ad hoc prompt front matter, while YAML workflow-local xprompts follow the workflow rules described in workflow_spec.md (https://sase.sh/workflow_spec/).

7.6 Reference Syntax

Reference inline-capable xprompts inside any prompt with the `#` prefix, including markdown-defined multi-agent xprompts whose body contains top-level `---` segment separators. Use `#!` only for standalone YAML workflows that do not have a `prompt_part` step. The marker must appear at the start of the string, after whitespace, or after one of `([{"'`. For compatibility, `#!name` is still accepted for multi-agent xprompts, but new prompts should use `#name`.

Syntax	Description
`#name`	Inline/template reference, no arguments
`#name(args)`	Inline parenthesis syntax with comma-separated arguments
`#name:arg`	Inline colon syntax, passes `arg` as a single positional arg
`#name:a,b,c`	Inline colon syntax with comma-separated multiple args
`#name:\`arg with spaces\``	Colon+backtick syntax for args containing spaces (single only)
`#name+`	Plus syntax, equivalent to `#name:true`
`#ns/name`	Namespaced reference (e.g., project-specific)
`#!name`	Standalone workflow reference, no args
`#!name(args)`	Standalone workflow reference with parenthesized arguments
`#!name:arg`	Standalone workflow reference with one colon-style arg
`#!name!!` / `#!name??`	Standalone workflow with an explicit HITL approval override

Examples:

```
sase run '#!sync'
sase run '#gh:sase #!sase/pylimit_split %approve'
```

During the compatibility window, top-level legacy invocations such as `sase run '#sync'` still run but emit a warning that points to `#!sync`. Inline expansion contexts reject standalone workflows instead of passing literal `#sync` text to the model. Shell examples should use single quotes around `#!...` so `!` is not interpreted by interactive shells.

For workspace references, underscores can be used as an alternative to colons: `#gh_sase` is equivalent to `#gh:sase`. The underscore is normalized to a colon before pattern matching, so both forms work identically. This is useful in contexts where colons are inconvenient. The `#cd` directory workflow is still a workspace reference even though it skips VCS checkout/release work.

Provider-backed references also support `@name` agent references in the ref portion. The `@name` is resolved at runtime to the named agent's ChangeSpec (branch name), allowing one agent's prompt to target another agent's workspace:

```
#gh:@planner     resolves to e.g. #gh:planner_add_config_parser
#gh_@reviewer    same, underscore form
```

This is useful when chaining agents — for example, a review agent can target the branch created by a prior agent using `@name` instead of hardcoding the branch name.

7.6.1 VCS Workspace References

Workspace-managing workflows use the same `#name:ref` reference syntax as xprompts, but they control where the agent runs before the rest of the prompt is executed. Some workspace references are VCS-backed (`#git` , `#gh` , `#hg`); `#cd` is directory-backed and does not reserve a numbered workspace.

Reference	Behavior
`#cd:<path>`	Run in a local directory without reserving a numbered SASE workspace or doing VCS checkout/release work
`#git:<ref>`	Run in a bare-git workspace
`#gh:<ref>`	Run in a GitHub workspace, when the GitHub plugin is installed
`#hg:<ref>`	Run in a Mercurial workspace, when an hg workspace plugin is installed

Prompts that do not contain a workspace reference are normalized to `#git:home` , so a bare prompt runs from the managed bare-git `home` project by default and gets normal numbered workspace, checkout, diff, and release behavior. Use `#cd:~` , `#cd:/abs/path` , `#cd:relative/path` , `#cd:../sibling` , or `#cd(.)` to choose a directory explicitly and skip VCS workspace management.

By default, a missing or uninitialized `home` ProjectSpec is bootstrapped as a managed empty bare-git project at the default `home` paths. To make bare prompts use an existing home/dotfiles bare repository, register a bare repository whose basename resolves to `home` , for example `#git:/path/to/home.git` . Use `#cd:~` when you want a direct home-directory run without VCS workspace management.

Provider-prefixed refs that point at a known project name are preserved as workspace launches even if the matching workspace plugin is not loaded in the current process. Known projects come from `~/.sase/projects/*/*.sase` (with legacy `~/.sase/projects/*/*.gp` accepted as a fallback). A launch such as `#gh:sase #!fix_just` therefore targets the registered `sase` project, allocates a numbered workspace for non-wait runs, and lets dispatch surfaces strip the wrapper ref when identifying an embedded workflow body.

Known projects may also declare `PROJECT_ALIASES` in their ProjectSpec. Alias refs in VCS workspace tags are canonicalized before workspace resolution and xprompt expansion, so `#gh:bob #p` is processed as `#gh:bob-cli #p` when the `bob-cli` project declares alias `bob` . The rewrite is exact and applies to colon, underscore, and parenthesized workspace-ref forms; it does not rewrite owner/repo paths such as `#gh:bbugyi200/bob` , partial project names, prose, or fenced code examples. See Project Aliases (https://sase.sh/project_spec/#project-aliases) for validation and management commands.

GitHub `owner/repo` refs use aliases after first use. Resolving `#gh:foo-org/foo` creates or reuses the canonical project whose `WORKSPACE_DIR` is `~/projects/github/foo-org/foo/` ; for a new repo that canonical name is typically `gh_foo-org__foo` , with generated alias `foo` . A second repo with the same basename, such as `#gh:bar-`

`org/foo`, gets a different canonical project such as `gh_bar-org__foo` and the next available alias, for example `foo-2`. Future launches can use `#gh:foo` and `#gh:foo-2`, and those refs canonicalize before prompt history, metadata, and artifacts are written.

For compatibility, existing basename ProjectSpecs are reused when their `WORKSPACE_DIR` already matches the GitHub repo. Owner/repo fallback avoids basename routing when duplicate GitHub basenames would make that ambiguous; direct `owner/repo` refs match the GitHub workspace path first, then only use a basename fallback when it is unambiguous.

ACE and the xprompt LSP provide a project/ChangeSpec completion helper for these references. Type `#+` at a token boundary, or type `+` as the first character in the prompt, to open a picker of active launchable projects and active PR-sized ChangeSpecs in `WIP`, `Draft`, `Ready`, or `Mailed` status. Accepting a project row inserts a tag such as `#gh:sase`; accepting a ChangeSpec row inserts a tag such as `#gh:my_change`. The helper filters by project name, project alias, or ChangeSpec name prefix, and it ignores system-managed `home`, inactive projects, sibling records, and non-launchable projects.

Known-project lookup defaults to active ProjectSpecs. Inactive and sibling projects are omitted from broad project-local xprompt catalogs and normal VCS workspace resolution; an explicit reference to an inactive known project fails with a hint to run `sase project activate <project>` before launching new work. Management and history code paths that need hidden projects opt into an all-state scan explicitly.

The raw colon form stops at whitespace, so paths with spaces should use the parenthesized form when possible: `#cd(/tmp/my project)`. Backtick quoting is supported for ordinary xprompt arguments, but workspace-reference path matching is intentionally conservative; prefer paths without spaces for embedded workspace tags.

Double underscores (`__`) in xprompt names are treated as forward slashes (`/`), enabling flat references to namespaced xprompts. For example, `#foo__bar` resolves to the xprompt registered as `foo/bar`, and `#a__b__c` resolves to `a/b/c`. Single underscores are not affected. This is useful when `/` is inconvenient in certain input contexts (e.g., shell completion or certain prompt editors).

Markdown headings like `# Heading` are not matched because a space after `#` prevents the pattern from firing.

7.7 Arguments

7.7.1 Positional Arguments

Positional arguments are comma-separated values inside parentheses:

```
#greet(Alice)
#format(json, 4)
```

Positional arguments are mapped to input definitions by position (0-indexed).

7.7.2 Named Arguments

Named arguments use `key=value` syntax:

```
#greet(user_name=Alice)
#format(style=json, indent=4)
```

Positional and named arguments can be mixed; positional arguments must come first:

```
#template(Alice, style=formal)
```

7.7.3 Quoted Strings

Values containing commas or special characters can be double- or single-quoted:

```
#note("Hello, world!", priority=high)
#tag('key=value')
```

7.7.4 Text Blocks

For multi-line argument values, use `[[...]]` delimiters:

```
#review([[
  This is a multi-line
  text block argument.
  Blank lines are preserved.
]])
```

Text blocks automatically strip leading whitespace from the first line and dedent continuation lines by their minimum common indentation.

7.8 Shorthand Syntax

Shorthand syntax converts line-oriented prompt text into `#name([[text]])` calls, avoiding the need for explicit text block delimiters.

7.8.1 Single-Colon Shorthand

`#name: text` at the start of a line captures text until a blank line (`\n\n`) or end of string:

```
#review: Please check this code for correctness
and performance issues.
```

This is equivalent to `#review([[Please check this code for correctness\nand performance issues.]])` .

7.8.2 Double-Colon Shorthand

`#name:: text` captures text until the next xprompt directive at a line boundary or end of string (blank lines do not terminate it):

```
#instructions:: Follow these rules:

1. Be concise
2. Be accurate

#review: Now review the code.
```

7.8.3 Paren + Shorthand

Combine parenthesized args with shorthand text:

```
#template(style=formal): Please review the following code.
#template(style=formal):: Please review the following code.

Even across blank lines (double-colon only).
```

The text is appended as a final positional text-block argument.

7.9 Typed Inputs

XPrompts can declare typed input parameters in the YAML front matter.

7.9.1 Longform Syntax

```yaml
input:
  - name: diff_path
    type: path
    description: Diff file to review.
  - name: max_retries
    type: int
    default: 3
    description: Maximum retry attempts.
```

7.9.2 Shortform Syntax

```yaml
input:
  diff_path: path
  max_retries:
    type: int
    default: 3
    description: Maximum retry attempts.
```

Both forms accept optional one-line `description` fields. Input descriptions do not change argument parsing or compact input signatures; rich surfaces such as catalogs, explain output, argument help, and editor documentation can use them as human-facing help text.

7.9.3 Supported Types

Type	Aliases	Validation
`word`	--	No whitespace allowed
`line`	--	No newlines allowed (default type)

`text`	--	Any content, no restrictions
`path`	--	No whitespace
`int`	`integer`	Must parse as an integer
`bool`	`boolean`	Accepts `true` / `false`, `yes` / `no`, `1` / `0`, `on` / `off`
`float`	--	Must parse as a float

7.9.4 Defaults

- An input with no `default` is required. Omitting it causes a template error if the caller does not supply a value.
- `default: null` means the YAML value was explicitly null. When `null` is passed as a positional or named argument value, it acts as a pass-through (the callee's own default applies).
- `default: ""` or any other value makes the input optional with that default.

7.10 Output Specification

XPrompts used as agent steps in workflows can declare an output schema for structured output validation. See the Output Specification (https://sase.sh/workflow_spec/#output-specification) section in the workflow spec for full details on the format.

7.10.1 Shortform Object

```
output: { name: word, description: text }
```

7.10.2 Shortform Array

```
output: [{ name: word, description: text, parent: { type: word, default: "" } }]
```

7.10.3 Longform

```
output:
  type: json_schema
  schema:
    properties:
      name: { type: word }
      description: { type: text }
```

When an output spec is present, the agent's response is validated against the schema. Semantic types (`word`, `line`, `text`, `path`, `bool`, `int`, `float`) are converted to JSON Schema types for validation and then checked for additional constraints (e.g., `word` rejects whitespace).

7.11 Jinja2 Integration

When the template body contains Jinja2 markers (`{{ }}`, `{% %}`, or `{# #}`), it is rendered as a Jinja2 template. Arguments (both positional and named) are available in the template context.

```
---
input: { user: word, verbose: { type: bool, default: false } }
---

Hello, {{ user }}.

{% if verbose %} Here is the detailed explanation... {% endif %}
```

7.11.1 Template Context

Variable	Description
`{{ name }}`	Named argument or input mapped by name
`{{ _1 }}`	First positional argument (1-indexed)
`{{ _2 }}`	Second positional argument, etc.
`{{ _args }}`	List of all positional arguments
`{{ root }}`	Absolute path to the primary workspace directory (omitted if unresolvable)
`{{ wait_chats }}`	List of chat-transcript paths for agents named in `%wait:<name>` directives, in the order they appear
`{{ agents["build"].path }}`	Output variables loaded from `%wait:build` when that agent used `sase var set path=...`

Named arguments and positional-to-name mappings take priority; if an xprompt is called within a workflow step, the workflow's execution scope is also available (xprompt args override scope values on conflict).

7.12 Legacy Placeholders

For templates that do not use Jinja2 syntax, a legacy placeholder mode is available. Placeholders use `{N}` syntax (1-indexed):

```
Review the {1} module and check for {2:correctness}.
```

- `{1}` -- required first positional argument.
- `{2:correctness}` -- second positional argument with default `correctness`.

Legacy mode is auto-detected: if the body contains no Jinja2 markers, legacy substitution is used.

7.13 Tags

XPrompts and workflows can be annotated with semantic role tags. Tags enable lookup-by-role instead of lookup-by-name, making the system extensible — a plugin or user can override the CRS workflow simply by defining a new xprompt with `tags: crs`.

7.13.1 Available Tags

Tag	Description

`vcs`	Workspace workflow xprompt (`#cd` , `#git` , `#gh` , `#hg`) — wraps other embedded workflows, running setup/teardown around them
`crs`	Code Review Summary workflow (singleton — `get_by_tag(crs)` returns the first match)
`fix_hook`	Fix hook workflow (singleton — used by axe to find the hook-fix agent)
`rollover`	Marks workflows whose embedded references carry forward to follow-up agent steps
`mentor`	Mentor review prompt workflow
`commit`	Commit workflow (appended by mentor review `A` key for direct commit)
`propose`	Propose workflow (appended by mentor review `a` key for propose-style amend)
`make_mentor_changes`	Apply accepted mentor comments workflow (launched by mentor review `Enter`)
`diff_file`	Injects the CL diff into the mentor prompt
`append_to_pr`	VCS-specific post-commit prompt appended when the active commit method creates a pull request
`append_to_commit_and_propose`	VCS-specific post-commit prompt appended when the active commit method creates a commit or proposal
`create_epic_bead`	Plan-approval Epic flow — creates the plan file, beads, and the epic agent prompt
`create_legend_bead`	Plan-approval Legend flow — creates a legend-tier bead with `epic_count` , then runs legend work
`work_phase_bead`	Per-phase agent prompt used by `sase bead work` (input: `bead_id`)
`land_epic`	Final land-the-epic agent prompt used by `sase bead work` after all phases complete
`land_legend`	Final land-the-legend agent prompt used by `sase bead work` after legend epics complete

7.13.2 Defining Tags

Tags can be defined in three places:

YAML workflow files (`.yml`):

```yaml
tags: vcs, rollover       # comma-separated string
# or
tags: [vcs, rollover]     # list format
```

Markdown front matter (`.md`):

```
---
name: fix_hook
tags: fix_hook
---

Fix the failing hook...
```

Config-based xprompts (`sase.yml`):

```yaml
xprompts:
  my_crs:
    content: "Review the code..."
    tags: [crs]
```

7.13.3 Tag-Based Lookup

The `get_by_tag()` function returns the first xprompt/workflow matching a tag, respecting the standard discovery order. This means higher-priority sources (e.g., project-local) can override built-in tagged xprompts.

```
from sase.xprompt.tags import XPromptTag, get_by_tag

crs_wf = get_by_tag(XPromptTag.crs)
fh_wf = get_by_tag(XPromptTag.fix_hook)
```

7.13.4 Backward Compatibility

The legacy `wraps_all: true` field on workflows is still supported — it automatically adds the `vcs` tag. New workflows should use `tags: vcs` instead.

Source: `src/sase/xprompt/tags.py`, `src/sase/xprompt/models.py`

7.14 Snippet Field

XPrompts can opt-in to ACE TUI snippet expansion by setting the `snippet` field in their front matter. When set, the xprompt's content is converted into a snippet template and merged into the ACE snippet registry at startup, so users can expand it by typing the trigger word and pressing `Tab`.

```
---
name: review
snippet: true
input:
  language: word
---

Review this {{ language }} code for correctness and style.
```

Values:

Value	Behavior
`true`	Use the xprompt's base name (part after last `/`) as trigger
`"custom_name"`	Use the custom string as the trigger word

Conversion rules:

- Normal xprompt references in the content are expanded before conversion, so snippets can compose reusable xprompts
- `{{ input_name }}` placeholders for required inputs become snippet tabstops (`$1`, `$2`, etc.)
- `{{ input_name }}` placeholders for inputs with defaults are pre-filled with the default value
- Legacy `{N}` placeholders are also converted
- XPrompts with complex Jinja2 control flow (`{% %}` or `{# #}`) are skipped
- User-defined snippets in `ace.snippets` take precedence over xprompt-derived snippets on name collision

Editor clients receive the same templates through `sase lsp` when they support LSP snippets. To troubleshoot the raw registry, run:

```
printf '{"schema_version":1}\n' | sase editor helper-bridge snippet-catalog
```

See docs/ace.md — Snippets (https://sase.sh/ace/#snippets) for snippet usage in the prompt input widget and editor completion.

Source: `src/sase/xprompt/snippet_bridge.py`, `src/sase/xprompt/models.py`

7.15 Skill Field

XPrompts can be marked as agent skill sources by setting the `skill` field in their front matter. `sase skill list` shows the loaded skill catalog without writing files. `sase skill init` reads that catalog, including bundled skill sources and runtime config overlays, to determine which xprompts should be rendered into per-provider `SKILL.md` files and deployed to agent skill directories. By default, generated skill files begin with a `sase skill use <name> --reason ...` directive so SASE can audit which skills an agent used; set `log_skill_use: false` in a skill source to omit that directive (see below). Recorded skill uses can be summarized and inspected with `sase skill log`. The compatibility alias `sase init skills` runs the same initializer.

```
---
name: sase_git_commit
skill: true
description: Commit changes using sase commit for git-based VCS
---

Commit instructions here...
```

Values:

Value	Behavior
`true`	Deploy to all registered providers
`["claude", "agy"]`	Deploy only to the listed providers

The `description` field provides a human-readable summary shown in `sase xprompt list` and `sase skill list` output. The structured catalog also marks these entries with `is_skill: true`; ACE and editor clients use that flag to offer slash-skill completions such as `/sase_plan` while keeping ordinary xprompts out of slash completion results.

The optional `log_skill_use` boolean field controls the generated audit directive. It defaults to `true`, so generated skills instruct the agent to run `sase skill use <name> --reason ...` as their first step. Set `log_skill_use: false` to suppress that directive for skills that should not record their own use (the bundled `/sase_plan` and `/sase_memory_read` skills set this). The field only affects sources that are also marked as skills.

Workflow: Edit packaged skill sources in `src/sase/xprompts/skills/`, or define user/runtime skill xprompts through the normal xprompt catalog sources. Do not include the `sase skill use` directive yourself; the generator injects it unless `log_skill_use: false` is set. Then run `sase skill list`, `sase skill init --dry-run`, and

finally `sase skill init --force` when the preview is correct. When `use_chezmoi` is enabled, `sase skill init` commits, pushes, and applies the generated files unless passed `--no-commit`, `--no-push`, or `--no-apply`. Do not edit deployed `SKILL.md` files directly. `sase init skills` is a compatibility alias for `sase skill init`.

Provider plugins declare where generated skills should be written. A source can target multiple providers, and a provider can have multiple filesystem targets. Built-in targets are:

Provider	Skill target(s)
Claude	`~/.claude/skills/<skill>/SKILL.md`
Codex	`~/.codex/skills/<skill>/SKILL.md`
Antigravity (agy)	`~/.gemini/antigravity-cli/skills/<skill>/SKILL.md`
Qwen	`~/.qwen/skills/<skill>/SKILL.md`
OpenCode	`~/.config/opencode/skills/<skill>/SKILL.md`

7.15.1 Bundled Skills

The following skills ship in `src/sase/xprompts/skills/` and are deployed by `sase skill init`. They are packaged with sase, included in `sase xprompt list`, and available to prompt completion clients even when a checkout does not have local skill files. Coding agents invoke them as `/sase_<name>`. Runtime config overlays can add more skill sources, so `sase skill list` may show entries that are not bundled here:

Skill	Purpose
`sase_agents_status`	Report on currently-running SASE agents (list, kill, show)
`sase_artifact`	Create explicit SASE artifacts from files produced during an agent run
`sase_beads`	Reference for `sase bead` commands (create, update, list, ready, show, dep)
`sase_chats`	Inspect prior sase agent chat transcripts via `sase chat list` and `sase chat show`
`sase_changespecs`	Inspect and reason about ChangeSpecs via `sase changespec search ...`, exact-name lookup, and safe edit rules
`sase_git_commit`	Commit changes for git-based VCS via `sase commit` (the only sanctioned commit path on git repos)
`sase_hg_commit`	Gemini-only commit skill for the hg/fig provider path
`sase_memory_read`	Guide audited long-term memory reads through `sase memory read`
`sase_notify`	Inspect SASE notification inbox entries via `sase notify list` and `sase notify show`
`sase_plan`	Submit a plan file for approval (used in lieu of disabled `EnterPlanMode`)
`sase_questions`	Ask the user structured questions (used in lieu of disabled `AskUserQuestion`)
`sase_var`	Attach named output variables to the current SASE agent run

7.16 Built-in XPrompts

Core xprompts ship in `src/sase/default_config.yml`, `src/sase/default_xprompts/*.md`, and `src/sase/xprompts/`. They are always available without needing a project- or user-level definition. They're at the built-in end of the discovery order, so any project, user, or config xprompt with the same name overrides the

packaged defaults. Common entries include:

Reference	Body summary
`#cd`	Switch the agent into a resolved SASE workspace directory
`#git`	Check out a git ref in an isolated workspace and show resulting changes
`#commit`	Create a normal commit from completed agent changes
`#propose`	Create a proposal from completed agent changes
`#file`	Require the agent to write its response to a named markdown artifact
`#fork`	Resume context from a prior agent conversation by name
`#fork_by_chat`	Resume context from a specific chat transcript path
`#mentor`	Run a structured mentor review against a CL
`#split_file`	Ask an agent to split one large Python file into import-safe smaller files
`#summarize`	Summarize a file in a short phrase for a specified use
`#json`	Require the agent response to satisfy a JSON schema
`#!sync`	Sync the current workspace and launch conflict-resolution help if needed
`#plan`	Asks the agent to think the work through and use its `/sase_plan` skill before any file changes
`#epic`	Marks the request as a multi-phase epic and chains `#plan`
`#legend`	Marks the request as a larger legend-level planning effort that should later split into epics
`#review`	Asks the agent to fix bugs and apply only clear-win improvements
`#prompt/approve`	Boilerplate "I've edited the previous reply with my decisions; implement this" preamble + `#plan`
`#prompt/review`	Wraps a `prompt` input and asks for a gap/ambiguity review before implementation
`#research`	Asks the agent to write a new `sdd/research/YYYYMM/` markdown file
`#research/image`	Asks the agent to generate an infographic for an existing research markdown file
`#research/more`	Asks the agent to improve an existing research markdown file by filling missed gaps
`#research/prompt`	Wraps a `prompt` input and asks for prior art, alternatives, and a recommended solution
`#research_swarm`	Fans out two independent research agents, consolidates their outputs, then generates an infographic
`#old_research_swarm`	Legacy initial, follow-up, and image research workflow (all tagged `%g:research`)
`#x:name,cmd`	Saves a freeform `sase_xcmd` command to the prompt (`@$(sase_xcmd <name> <cmd>)`)
`#bd/new_epic`	Multi-phase epic kickoff used by `sase bead work` (resolved via `XPromptTag`)
`#bd/new_legend`	Legend kickoff that records `epic_count`, commits metadata, then runs `sase bead work`
`#bd/work_phase_bead`	Per-phase agent prompt used by `sase bead work`
`#bd/land_epic`	Final land-agent prompt used by `sase bead work`
`#bd/next`	"What should I work on next?" helper that consults the bead tracker
`#bd/review/plan`	Plan-review helper for an epic plan
`#bd/review/prompt`	Prompt-review helper for an epic plan

When `#fork` / `#fork_by_chat` injects a `# Previous Conversation` block, the prior **user prompts** in that block are sanitized first: sase directives (`%name`, `%wait`, `%group`, ...), `#` / `#!` xprompt and workspace references, and any unrendered Jinja2 markers (`{{ }}`, `{% %}`, `{# #}`) are stripped so the forked agent sees clean natural-language text. Fenced code blocks and real markdown headings are preserved, and assistant responses are left untouched. Raw transcripts on disk are unchanged — the cleanup happens only when building resume history (so `sase chat show` still shows the original prompts).

`#bd/new_epic` accepts optional ChangeSpec metadata for the plan bead it creates:

```
#git:sase #bd/new_epic(plan_file_path=sdd/epics/202604/example.md, changespec=sase_feature, bug_id=12345)
```

When `bug_id` is supplied, `changespec` must also be supplied; the generated plan bead is created with the corresponding `sase bead create -c/--changespec` and `-b/--bug-id` metadata.

`#bd/new_epic` also accepts an optional `legend_bead_id`. When supplied, or when the epic plan file frontmatter contains `legend_bead_id`, the epic is linked under that legend with `--type plan(<plan_file>,<legend_bead_id>) --tier epic`. It creates the epic plan bead first, then creates phase beads sequentially in the order they appear in the plan file. Phase bead creation is intentionally not parallelized because child bead suffixes are allocated by creation order. `#bd/new_legend` creates a `--tier legend --epic-count <count>` plan bead for `sdd/legends/{YYYYMM}/...`, writes `legend_bead_id`, `tier: legend`, and `epic_count` frontmatter to the legend plan, commits that metadata, then runs `sase bead work <legend_bead_id> --yes`.

To see the exact body of any built-in inline xprompt, run `sase xprompt expand --trace '#<name>'` or browse the catalog with `sase xprompt catalog`. Use `sase xprompt explain <name>` for workflows; the explain command takes the workflow name without a `#` or `#!` marker.

7.16.1 Bundled Standalone Workflows

The repo-level `xprompts/` directory also ships standalone YAML workflows that are normally invoked with `#!`:

Reference	Inputs	Purpose
`#!sase/fix_just`	none	Repair or validate `just` workflow issues
`#!sase/pylimit_split`	`limits`	Assist with Python file-size splitting
`#!sase/refresh_docs`	`project`, `gh_ref`, `threshold`	Scheduled documentation refresh
`#!sase/audit_recent_bugs`	`project`, `gh_ref`, `threshold`	Scheduled audit of recently filed bugs
`#!sase/audit_recent_improvements`	`project`, `gh_ref`, `threshold`	Scheduled audit of recently filed improvements

The `#!sase/fix_just` workflow first bootstraps the workspace with `just install`, then records the results of `just fmt-check`, `just lint`, and `just test`. Those three checks run before any repair step, so lint/test failures are based on the original checkout state for that workflow run. If formatting failed, the workflow runs `just fmt`, stages the resulting tree with `git add -A`, creates a commit when the staged diff is non-empty, then rebases on upstream and pushes. If lint or test failed, it launches separate draft-PR repair agents through `#gh:sase #pr(...)` for `fix_just_linters` and `fix_just_tests`; workflow expansion applies the normal PR context before the agents run.

The scheduled documentation refresh workflow lives in this repo as `xprompts/refresh_docs.yml` and is invoked as `#!sase/refresh_docs`. It accepts `project`, `gh_ref`, and `threshold`, defaulting to the main `sase` repo behavior (`project=sase`, `gh_ref=sase`, `threshold=50`). For scheduled checks in linked repos, pass repo-specific values such as `#!sase/refresh_docs(project=sase-core, gh_ref=sase-org/sase-core, threshold=25)`. The `project` input selects the marker path under `~/.sase/projects/<project>/refresh_docs_marker`; `gh_ref` is forwarded to the nested docs agent as `#gh:<gh_ref>`. Scheduled lumberjack agents in `sase_athena.yml` reach these workflows by embedding `#gh:sase-org/sase` in their prompts so the `sase` project workspace is selected before resolution.

7.17 Config-Based XPrompts

XPrompts can be defined inline in `sase.yml` under the `xprompts:` key.

7.17.1 Simple Format

```yaml
xprompts:
  propose: "Please propose your changes before applying them."
```

7.17.2 Structured Format

```yaml
xprompts:
  greet:
    description: Greet a user a configurable number of times.
    input:
      name:
        type: word
        description: Name to greet.
      count:
        type: int
        default: 1
        description: Number of greetings to render.
    content: "Hello {{ name }}, count is {{ count }}"
```

Config-based xprompts have priority 6 (below file-backed project and user xprompts; above plugin and built-in definitions).

Standalone workflows must be defined as YAML files in an `xprompts/` directory (repo-level, project plugin, or built-in package). Top-level `workflows:` blocks in `sase.yml` or `sase_*.yml` overlays are no longer supported and will be ignored by the runtime; move any such definitions into `xprompts/<name>.yml` files.

7.18 Local Configuration Files

You can define project-specific xprompts in a `sase.yml` file in the current working directory. This is a full sase config file that can override any configuration, including xprompts. It is the highest-priority config source in the deep-merge system (https://sase.sh/configuration/#deep-merge-system), overriding global `sase.yml`, overlay files, plugin configs, and built-in defaults. Individual `.md` files in xprompts directories still take precedence over config-defined xprompts.

```yaml
xprompts:
  # Simple format — value is the template body
  propose: "Please propose your changes before applying them."

  # Structured format — with typed inputs and/or output
  greet:
    description: Greet a user a configurable number of times.
    input:
      name:
        type: word
        description: Name to greet.
      count:
        type: int
        default: 1
        description: Number of greetings to render.
    content: "Hello {{ name }}, count is {{ count }}"
```

7.19 Directives

Directives are in-prompt tags with a `%` prefix that modify agent runner behavior. They are extracted and stripped from the prompt before further processing.

7.19.1 Supported Directives

Directive	Alias	Description
`%model`	`%m`	Override the LLM model for this prompt
`%name`	`%n`	Assign, auto-generate, or force-reuse an agent name
`%wait`	`%w`	Wait for another agent or workflow to succeed
`%time`	`%t`	Defer launch by a duration or until an absolute wall-clock time
`%hide`	`%h`	Hide the agent from the default Agents tab display
`%approve`	`%a`	Run the agent fully autonomously (skip approval)
`%epic`		Enable plan mode and auto-approve the plan as an epic
`%edit`	`%e`	Return editor text to the prompt bar for review
`%repeat`	`%r`	Run the prompt multiple times (e.g., `%repeat:3`)
`%group`	`%g`	Assign the agent's user-managed tag (e.g., `%group:review`)
`%alt`	`%{}`	Split prompt into variants with different text (brace shorthand)

7.19.2 Syntax

Directives use the same argument syntax as xprompt references:

```
%model:claude-sonnet         # Colon syntax
%model(claude-sonnet)        # Parenthesis syntax
%model:`claude-sonnet-4`     # Backtick syntax (for values with special chars)
%model:codex/o3              # Provider/model syntax — switches both provider and model
%model:agy/flash35h          # Provider/model syntax for Antigravity (agy)
%model:opencode/anthropic/claude-sonnet-4-5 # Nested provider/model syntax
%name:reviewer               # Short-form
%n:reviewer                  # Same, using alias
%name                        # Bare — auto-generates a unique name
%name:!reviewer              # Force reuse by wiping the previous owner
%wait:agent1                 # Wait for agent1
%w:agent2                    # Wait for agent2 (alias)
%wait                        # Bare — waits for the most recently named agent
%wait:agent1,agent2          # Multi-value: equivalent to two separate %wait: lines
%wait(agent1, agent2)        # Same, paren form
%time:5m                     # Wait for 5 minutes before starting
%t:1h30m                     # Wait for 1 hour 30 minutes (alias)
%time:90s                    # Wait for 90 seconds
%time:1430                   # Wait until 14:30 today (wraps to tomorrow if past)
%time:260415/0900            # Wait until 2026-04-15 at 09:00
%wait:agent1 %time:5m        # Wait for agent1, then a 5-minute floor
%repeat:3                    # Run the prompt 3 times
%r:5                         # Same, using alias
%{#review | #test}           # Brace shorthand: branches split on top-level `|`
%alt(#review,#test)          # Long form: same two variants, comma-separated
%(#review,#test)             # Legacy shorthand, still accepted (prefer `%{...}`)
%{sec=#review | perf=#test}  # Named branches become child name suffixes
%{extra instructions}        # Single branch: split into with/without variants
%approve                     # Run fully autonomously
%a                           # Same, using alias
%edit                        # Return editor text to prompt bar
%e                           # Same, using alias
%epic                        # Enable plan mode and auto-approve the plan as an epic
%group:review                # Assign the tag "review" to this agent
%g:review                    # Same, using alias
```

The `%model` directive also supports automatic provider resolution: known model names (e.g., `opus`, `o3`, `qwen3.6-plus`) are automatically mapped to their provider. See Per-Prompt Provider Switching (https://sase.sh/llms/#per-prompt-provider-switching) for the full model-to-provider mapping.

The `%name` and `%wait` directives can be used without arguments. Bare `%name` auto-generates a permanent unique name for the agent. Bare `%wait` resolves to the most recently named agent (raises an error if no previous agent exists).

Agent-name templates contain exactly one `@` marker. The marker is not a wildcard; SASE replaces it with the next token from the shared auto-name sequence (`0`, `1`, ..., `9`, `a`, ..., `z`, `00`, ...). For example, with no reserved names, `%name:@.cld` renders as `0.cld`, `%name:build-@` renders as `build-0`, and `%name:research.@.final` renders as `research.0.final`. The older terminal `-@` form still works, but new allocations now start at token `0` and use the alphanumeric sequence instead of positive integers. Later `%wait`, `#fork`, and `#resume` references can use the same template text; in one multi-agent launch, SASE rewrites those references to the concrete name already planned for that template before spawning dependent agents.

Agent names are permanent IDs. A name that belongs to any existing agent state cannot be reused by a normal `%name:<name>` launch; SASE cancels the launch before spawning an agent, records the prompt as cancelled, and suggests the lowest free numeric suffix such as `<name>1`. To deliberately reuse a name, use `%name:!<name>` from the TUI; the `!` form is the explicit confirmation to wipe the previous owner and its persisted system state before launching the new agent with that name. Non-TUI launch surfaces reject `%name:!<name>` unless they provide an explicit confirmation path.

Named `%wait` dependencies unblock only after the newest matching agent run has a `done.json` outcome of `"completed"`. For a multi-agent workflow name, the workflow root and every child agent for that root must complete successfully. Failed, killed, crashed, still-running, malformed, or missing `done.json` artifacts do not satisfy the wait; the dependent agent stays parked until a later successful run of the same dependency name appears.

When a launch has exactly one explicit `%wait:<name>` dependency and no explicit `%name`, SASE can allocate a derived name before spawning the waiting agent: `<name>.w1`, `<name>.w2`, and so on, using the first free slot. Multi-value waits, bare `%wait`, and prompts whose name depends on unresolved xprompt expansion do not get a parent-side derived name. Repeat launches reuse this rule, then chain later repeat slots with `%wait:<previous-slot-name>`.

If a prompt includes both `#fork` / `#resume` and `%wait`, the fork-derived `.f<N>` name takes precedence over the wait-derived `.w<N>` name. The wait still controls launch ordering, but the planned agent name follows the resume/fork lineage.

The `%time` directive (alias `%t`) defers launch by a duration or until an absolute wall-clock time:

- **Durations** in `XhYmZs` format (e.g., `%time:5m`, `%time:1h30m`, `%time:90s`). When multiple `%time` durations are specified, the maximum is used.
- `HHMM` — wait until that time today (e.g., `%time:1430` for 14:30). If the time has already passed, it wraps to tomorrow.
- `yymmdd/HHMM` — wait until a specific date and time (e.g., `%time:260415/0900` for 2026-04-15 at 09:00). Raises an error if the target is in the past.

`%time` and `%wait` combine freely: dependencies wait first, then the time floor applies.

Absolute time waits cannot be combined with duration waits or with each other.

Bare `%time` is invalid — `%time` requires a duration or absolute time argument. Time-shaped values passed to `%wait` (e.g. `%wait:5m`) raise an error with a migration hint pointing to `%time:5m`.

Multi-value directives (`%wait`, `%time`, `%model`, `%alt`) accept comma-separated arguments to collapse what would otherwise be several lines: `%wait:agent_a,agent_b` is equivalent to two separate `%wait:` directives. Backtick-quoted values (e.g. `%wait:`a,b``) are treated as a single literal and not split on commas.

The `%approve`, `%edit`, and `%epic` directives are boolean flags — they take no arguments and are simply present or absent.

7.19.3 Example

```
%model:`claude-sonnet-4-20250514`
%name:code-reviewer
%wait:planner
Review the code changes and provide feedback.
```

The directives are stripped from the prompt text. The agent will use the specified model, be named "code-reviewer", and will wait for the "planner" agent to complete successfully before running.

7.19.4 Hide Directive

The `%hide` directive marks an agent as hidden. Hidden agents are not shown in the Agents tab by default — press `.` to toggle their visibility. This is useful for background agents spawned by axe or workflows that don't need active monitoring:

```
%hide
%name:background-checker
Run periodic health checks.
```

7.19.5 Approve Directive

The `%approve` directive marks an agent as fully autonomous. The agent runs without requiring human approval for plan steps or other checkpoints that would normally pause for user input:

```
%approve
%name:auto-fixer
Fix the lint errors in the codebase.
```

7.19.6 Edit Directive

The `%edit` directive causes the editor text to be loaded into the ACE prompt input bar instead of being submitted directly. This lets you compose a prompt in `$EDITOR` (via `Ctrl+G`) and then review or tweak it in the prompt bar before launching an agent:

```
%edit
Refactor the parser module to use dataclasses.
```

When the editor closes, the `%edit` directive is stripped and the remaining text appears in the prompt input bar for further editing. The agent is not launched until you press Enter in the prompt bar.

The returned text is loaded with editor-file semantics: if it contains real multi-agent `---` segment separators (outside fenced blocks and outside leading YAML frontmatter), it is returned to the ACE prompt stack as one editable prompt pane per agent segment, and any leading xprompt frontmatter is lifted into the prompt properties panel above the top pane. A buffer with no separators still returns to the bar for review; if it has leading frontmatter, that frontmatter is lifted into the properties panel rather than left as literal pane text.

7.19.7 Plan Approval and Coder Follow-up

SASE's planning workflow is driven by the `/sase_plan` skill together with the `sase plan` approval pipeline. An agent drafts a plan and submits it with `/sase_plan` (or `sase plan propose`); the plan then pauses for user approval before any execution. In the TUI, the agent shows a PLAN status after submitting the plan for review, then PLAN APPROVED once the user approves it. The `%epic` directive opts a planning agent into this same pipeline with automatic epic approval (see Epic Directive).

Once the plan is approved, sase launches a follow-up **coder** agent using the same handoff body as the `#coder` built-in xprompt (see sase/xprompts/coder.md (https://github.com/sase-org/sase/blob/main/src/sase/xprompts/coder.md)). `#coder` takes the approved plan file as its `plan_file` input, injects it with `@`, and instructs the agent to implement the plan. By default the coder does *not* inherit the planner's chat transcript — the plan file is the hand-off artifact. Set `SASE_CODER_INHERIT_PLANNER_CHAT=1` to restore the old behavior, in which case a `#fork:<planner_name>` reference is prepended to the coder prompt so it resumes the planner's session. The coder prompt also carries a `%model:` directive. A model chosen at approval time wins. When no model is chosen, the follow-up default is resolved **at handoff time from the planner agent's concrete provider/model**: an active worker override, a matching `llm_provider.worker_models` entry for the planner's primary lane, then the planner's own provider/model as the fallback. The generated prefix is a concrete `%model:<provider>/<model>` so the planner's primary context is preserved even if the global worker lane changes before launch (when the planner is missing provider/model metadata the follow-up falls back to a bare `%model:worker`). Ordinary `%model:worker` directives written elsewhere still resolve from the current effective worker lane.

Outside the TUI, `sase plan` shows the same pending PlanApproval notifications plus recent approved and inferred rejected plans. Use `sase plan approve <id-prefix> --kind approve|commit|epic|legend|tale` to approve from a shell. The `approve` kind runs the coder without committing an SDD plan; `tale` commits an SDD tale and runs the coder; `epic` and `legend` commit the matching SDD tier and launch the bead follow-up; `commit` records the approved plan in SDD without launching a coder. `-m/--model` picks the follow-up agent's model, while `-p/--prompt` adds extra coder instructions for the `approve` and `tale` paths.

7.19.8 Epic Directive

The `%epic` directive enables plan mode and marks the submitted plan for epic approval. When the agent later submits a plan with `/sase_plan` or `sase plan propose`, sase follows the same epic path as the TUI Epic action: it writes the SDD epic files, commits them as needed, initializes beads, and launches the epic follow-up agent. Unlike `%approve`, `%epic` is plan-specific and does not automatically answer unrelated questions.

```
%epic
%name:billing-epic
Plan the billing dashboard epic.
```

7.19.9 Repeat Directive

The `%repeat` directive runs the same prompt multiple times. The argument is a positive integer specifying the repeat count:

```
%repeat:3
%name:linter
Run lint checks on the codebase.
```

This launches 3 independent agents — each spawned with its own process, workspace, and `agent_meta.json`, appearing as its own top-level entry in the Agents tab. Fan-out happens at launch time: the directive is consumed when the agents are spawned, so there is no outer loop or TUI affordance ticking through iterations. The slot numbers are appended to the `%name` base (`linter.1`, `linter.2`, `linter.3`); when `%name` is omitted the auto-assigned base is used (e.g. `a.1`, `a.2`, `a.3`).

Iterations run **sequentially**: all N agents are spawned up front and register immediately in the TUI, but iteration `k+1` is automatically wait-chained behind iteration `k` via an injected `%wait:<prev_name>` directive. This turns `%repeat` into a serial iteration primitive — each iteration can observe its predecessor's work — without blocking the launcher on any single agent.

Each iteration exposes two iteration-scoped named arguments in the agent's workflow:

Variable	Meaning	Example with `%repeat:5`
`n`	Current iteration (1-based)	1, 2, 3, 4, 5
`N`	Total iterations (the `%repeat` argument)	5

These are threaded through via the `SASE_REPEAT_ITERATION` and `SASE_REPEAT_TOTAL` environment variables — the agent runner reads them, converts to ints, and passes them as named args into the workflow so they appear as Jinja2 variables in the prompt body:

```
%repeat:5
Run test suite batch {{ n }} of {{ N }}.
```

Stopping a repeat chain early with `STOP`

A repeat iteration can stop every later slot by setting the reserved `STOP` output variable before it completes:

```
%repeat:5
Process the next batch; if there is no more work, run: sase var set STOP=1
```

Because the slots are already spawned and wait-chained, "stopping" works on wake: when a later slot's `%wait` on its repeat predecessor resolves, the slot checks that predecessor's `STOP` output variable. If it is truthy, the slot propagates `STOP`, finalizes as a successful **completed** (skipped) slot — recording `repeat_stopped: true` and `stopped_by` in its `done.json` — and exits without claiming a workspace or running its prompt. Keeping the outcome `completed` lets the stop cascade down the chain through the ordinary `%wait` resolution, so each remaining slot winds down one wait-check cycle after the previous one. `STOP` is conservative: `""`, `0`, `false`, `no`, and `off` (case-insensitive) are not-stop; any other value stops the chain. See Cross-Agent Output Variables for how `STOP` behaves outside repeat chains.

7.19.10 Alt Directive

The `%alt` directive splits a single prompt into multiple variant prompts, each launched as a separate agent. Each branch replaces the directive span in the output prompt.

The preferred shorthand is `%{A | B | ...}`, which uses braces and splits branches on top-level `|` separators:

```
%{#review | #test | #docs}
Analyze the codebase.
```

This produces three agents, each with "Analyze the codebase." but with `#review`, `#test`, or `#docs` substituted in place of the directive. Branches can be arbitrary text — xprompt references, directives, plain instructions, or `[[text blocks]]`. Because branches split only on a top-level `|`, a comma is ordinary branch text: `%{foo, bar | baz}` is

two branches (`foo, bar` and `baz`), not three. Nested `()`, `[]`, `{}`, and backtick-quoted spans are not split, and any `|` inside them is treated literally.

The long form `%alt(...)` and the legacy `%(...)` shorthand remain accepted; both use parentheses with comma-separated branches:

```
%alt(#review, #test, #docs)
%(#review, #test, #docs)
```

New prompts, completions, snippets, and docs should prefer `%{...}`. `%(...)` stays parse-compatible during the migration; it may be removed in a future release.

Named Branches

Branches can be named with `id=value`. The `value` is inserted into the spawned prompt and the `id` becomes the child agent suffix. For example, `%name:review %{sec=[[security]] | perf=[[performance]]}` launches `review.sec` and `review.perf`. Unnamed branches use numeric suffixes while skipping any numeric ids already provided by named branches, so `%{2=[[named]] | [[first]] | [[second]]}` launches suffixes `2`, `1`, and `3`.

When the same named branch id appears in more than one alt directive, those directives are correlated: values with the same id render into the same child prompt, and a missing id in one correlated directive renders as empty text. Empty renders also collapse adjacent horizontal whitespace, so they do not leave doubled spaces or spaces before punctuation; only spaces and tabs are collapsed, newlines and indentation are preserved, and non-empty branches are untouched. For example:

```
%name:repo %{a=Describe | b=Explain} how this repo works %{a=in detail}.
```

This launches `repo.a` with "Describe how this repo works in detail." and `repo.b` with "Explain how this repo works.".

Single Branch (With/Without Split)

A single-branch alt is treated as a with/without split — it produces two prompts: one with the branch text and one with the directive removed entirely:

```
%{Also check for security issues.}
Review this module.
```

This launches two agents: one with "Also check for security issues. Review this module." and one with just "Review this module."

Cartesian Product

Multiple alt directives can appear in the same prompt. Branch lists with no repeated named ids form a **Cartesian product**: one agent is launched per combination. Brace and paren forms mix freely:

```
%{Focus on security | Focus on perf} %m(opus, sonnet)
Review this code.
```

This produces 2 × 2 = 4 agents (every focus area paired with every model). The multi-model directive (`%m(opus,sonnet)`) is internally rewritten as `%alt(%model:opus,%model:sonnet)`, so it participates in the Cartesian product naturally. A `%model` directive used as a branch inside `%{...}` composes the same way: `%{#review | %model:opus}` fans out a default-model review branch and an opus branch.

Repeated named ids are the exception to the Cartesian rule. Disjoint named ids and unnamed branches remain Cartesian; only the same explicit id repeated across directives is zipped together.

7.19.11 Multi-Model Directive

The `%model` directive supports launching multiple agents in parallel — one per model — when given comma-separated model names in parentheses:

```
%m(opus,sonnet)
Review this code for edge cases.
```

This launches two agents with identical prompts, each using a different model. Each agent appears as a separate entry in the Agents tab. Only the parenthesized syntax triggers multi-model behavior; colon syntax (`%m:opus`) and single-model parentheses (`%m(opus)`) always launch a single agent.

When a prompt fans out to multiple models, the spawned agents share a single base name and carry a runtime suffix so they can be told apart at a glance. Given `%m(opus,gpt-5.5) %n:foo`, the two agents are named `foo.cld` and `foo.cdx`. The runtime suffix is a short alias declared by the provider plugin (via the `llm_provider_short_name` hook) — `cld`, `cdx`, `agy`, `qwn`, `opc` for the built-in providers — falling back to the full provider name for plugins that don't declare one. If `%name` is omitted, a single auto-generated base is allocated and shared (e.g. `a.cld` / `a.cdx`) rather than each agent picking its own letter independently. Single-model prompts retain their plain `%name` value unchanged. When two models share a runtime (e.g. `%m(opus,sonnet)` — both `claude`), the model name disambiguates the suffix: `foo.cld-opus` and `foo.cld-sonnet`. Long model names (e.g. `Gemini 3.5 Flash (High)`) are replaced with a short alias (`flash35h`) declared by the provider plugin, so a same-runtime agy fan-out reads as `foo.agy-flash35h` / `foo.agy-flash35l` rather than echoing the full model string. Model arguments used for naming are first resolved through xprompt shorthand expansion, while the launched prompt keeps the original `%model` value. For example, `%n:ag %m(#flash,#pro)` can launch agents named `ag.agy-flash35h` and `ag.agy-pro31h`.

7.19.12 Multi-Value Directives

The `%wait` directive supports multiple occurrences — each adds to the wait list:

```
%wait:agent1
%wait:agent2
%wait:agent3
Do work after all three agents finish.
```

Agent dependencies and time floors can be mixed freely:

```
%wait:agent1
%time:5m
Wait for agent1 to finish, then wait at least 5 minutes from launch.
```

7.20 Command Substitution

XPrompt arguments support shell command substitution using `$(cmd)` syntax. The command is executed via the shell and its output replaces the `$(cmd)` expression.

```
#bug:$(branch_bug)          # Use output of branch_bug command as the argument
#review:$(git diff HEAD~1)  # Pass git diff output as argument
```

Nested parentheses are supported: `$(echo $(date))`. To include a literal `$(`, escape it as `\$(`.

Failed commands or commands producing empty output result in an empty string replacement. Command outputs are cached within a single expansion pass to avoid redundant execution.

7.21 Protected Content

7.21.1 Fenced Code Blocks

Content inside triple-backtick fenced code blocks is automatically protected from xprompt expansion:

````
Here's an example:

```
#foo will NOT be expanded inside this code block
```

But #foo HERE will be expanded normally.
````

This prevents accidental expansion of `#name` patterns in code examples, documentation, and similar content.

7.21.2 Disabled Regions

You can explicitly disable xprompt expansion for a region of text using the `%xprompts_enabled` directive:

```
%xprompts_enabled:false
This content is passed through verbatim.
#foo will NOT be expanded here.
%xprompts_enabled:true
Normal expansion resumes here.
#foo WILL be expanded.
```

The markers are stripped from the final output. This is useful for embedding raw xprompt syntax in documentation or for passing literal `#name` patterns to downstream consumers.

The closing `%xprompts_enabled:true` marker may appear either on its own line or **inline** at the end of a content line. In both forms the marker (and any whitespace immediately preceding an inline marker) is stripped from the final output, so prompts authored as natural prose can re-enable expansion mid-line:

```
%xprompts_enabled:false
... raw content where #foo and @bar are passed through verbatim. %xprompts_enabled:true
And expansion resumes here.
```

7.22 XPrompt Aliases

XPrompt aliases provide raw text-level substitution that runs *before* any other xprompt processing. They are defined in the `xprompt_aliases` config field in `sase.yml`.

These are global shorthand aliases for xprompt names and raw refs. They are separate from ProjectSpec `PROJECT_ALIASES`, which map alternate project names such as `bob` to canonical known projects such as `bob-cli` at the launch boundary. Project aliases are canonicalized before xprompt expansion so launch artifacts and history store the canonical project name.

The built-in defaults provide two shorthand aliases:

Alias	Target	Usage
`c`	`commit`	`#c` → `#commit`
`p`	`propose`	`#p` → `#propose`

Additional aliases can be added in user config files:

```
xprompt_aliases:
  gh_sase: "gh:sase" # #gh_sase → #gh:sase
  gh_foo: "gh:foo/bar" # #gh_foo  → #gh:foo/bar
```

When the processor encounters `#alias_name` in a prompt, it replaces the alias name portion with the target string before any xprompt resolution occurs. This is particularly useful when the target contains characters (like `:`) that must be present in the raw text for other processing logic — such as VCS directory-switching — to work correctly.

See Configuration Reference: xprompt_aliases (https://sase.sh/configuration/#xprompt_aliases) for the full field specification.

7.23 Recursive Expansion

XPrompt bodies can reference other xprompts. Expansion is iterative: after each round of substitution, the result is scanned again for new `#name` references. This continues until no known references remain, up to a maximum of 100 iterations (to guard against circular references).

7.24 Multi-Agent Prompts

A single prompt can launch multiple agents by using YAML frontmatter and `---` segment separators. SASE plans the segments in document order, but agents do not wait for earlier segments unless you add a dependency such as `%wait:<name>` or bare `%wait`. The same `---`-separator convention also applies inside an xprompt body — see Multi-Agent XPrompts (Library-Defined Fan-Out) below.

7.24.1 Frontmatter Panel (ACE TUI)

In the `sase ace` prompt input, ad hoc prompt frontmatter has a structured **Frontmatter Panel** above the prompt stack, with the same field set an xprompt `.md` file supports (`name`, `description`, `tags`, `input`, `xprompts`, `skill`, `snippet`). Open or focus it with the prompt NORMAL-mode `g=` keymap; while the panel owns focus, `g=` runs the

panel's deactivate/apply path. The panel also auto-shows when ACE has lifted leading frontmatter into the stack, such as a multi-agent prompt load or an editor-file return from `%edit` / whole-stack `Ctrl+G`. A single prompt recalled from history with leading frontmatter but no segment separator stays one verbatim pane instead of auto-opening the panel. Typing `---` in the prompt body is passive during live editing: at the very start it stays literal text, and after content it does not split the active pane. Add a top-level property with `a` (a picker sourced from the same core schema that backs the editor LSP), edit scalar/list fields inline, delete a field with `d`, and use `R` for a live-validated raw-YAML escape hatch. The panel owns the canonical YAML it serializes back onto the prompt, so the multi-agent launch path is unchanged.

The structured `input` and `xprompts` fields render as foldable sub-trees (`h` / `l`): navigate into them with `j` / `k`, then `A` / `e` / `d` (or `enter`) add, edit, and delete individual items through small typed sub-form modals. The input editor offers a `name`, a core-validated `type` (with its one-line rule shown inline), an optional `default` (blank means required), and a `description`; the xprompts editor offers a `_`-prefixed `name` (validated by the same underscore rule the launch path enforces), `content`, a compact `name:type[=default]` inputs field, and a `description`. A `#_helper` declared here lights up `<ctrl+t>` / `<ctrl+l>` completion and argument hints in every prompt pane exactly like a global xprompt — define a helper in the panel and it is instantly usable below.

7.24.2 Frontmatter-Defined Local XPrompts

YAML frontmatter at the start of a prompt can define local xprompts under the `xprompts:` key. These are defined once in the frontmatter and each segment receives only the local xprompts it actually references (including transitive dependencies). Local xprompt names **must** start with `_` to distinguish them from global xprompts.

```
---
xprompts:
  _review_rules: "Always check for error handling and edge cases."
---
#_review_rules
Review the authentication module.
```

Local xprompts support the same structured format as config-based xprompts (typed inputs, Jinja2 content):

```
---
xprompts:
  _template:
    input: { target: word }
    content: "Review the {{ target }} module."
---
#_template(auth)
```

7.24.3 Frontmatter-Declared Inputs

Prompt frontmatter can also declare `input:` arguments using the same typed shorthand as xprompt files (see Typed Inputs). The declared values are substituted into every segment's `{{ name }}` placeholders before the agents fan out:

```
---
input:
  service: word
  retries: { type: int, description: how many times to retry }
  dry_run: { type: bool, default: false }
---
Refactor the {{ service }} module ({{ retries }} retries, dry_run={{ dry_run }}).
```

When a prompt with required (default-less) inputs is submitted in `sase ace`, an **Input Collection Modal** opens after the whole-stack submit: each required input gets a typed, validated field (optional inputs stay collapsed behind a reveal toggle, showing their defaults), and the agents launch only once every value is valid. Non-interactive CLI launches (`sase run`) cannot prompt, so a required input without a default fails fast with a clear message instead of a cryptic template error — give such inputs a default or launch from the TUI.

7.24.4 Segment Separators

After the frontmatter block is consumed, subsequent `---` lines on their own act as segment separators. Each segment launches as a separate agent:

```
---
xprompts:
  _common: "Follow the project coding conventions."
---
%name:step1
#_common
Implement the new feature.
---
%name:step2
%wait:step1
#_common
Write tests for the new feature.
```

This launches two agents. `step2` starts after `step1` succeeds because the second segment includes `%wait:step1`; if that line were omitted, both agents would be eligible to run independently. Both agents share the `_common` local xprompt.

7.24.5 Cross-Agent Output Variables

Agents can publish small string values for later waited agents or segments with `sase var set KEY=VALUE`. Give the producer a stable name and make the consumer wait before referencing the producer's variables. Every producer's variables live under a single reserved `agents` dictionary keyed by agent name:

```
%name:build-@
Build the report, then run:
sase var set report_path=dist/report.md status=ok
---
%name:review
%wait:build-@
Review {{ agents["build"].report_path }} after the build status is {{ agents["build"].status }}.
```

The review prompt is rendered after the `build-@` dependency completes, so `{{ agents["build"].report_path }}` and `{{ agents["build"].status }}` come from the producer's stored `agent_meta.json` values. A consumer that has already started will not see later writes.

The `agents` key is a stable Jinja namespace for the producer, not always the producer's concrete runtime name. Agent-name templates use the template base, so a producer that launches as `build-0` from `%name:build-@` is read as `{{ agents["build"].report_path }}`, not `agents["build-0"]`. The key is otherwise the raw agent name with no identifier munging, so dotted, hyphenated, and digit-leading names all work via bracket access: `%name:research.@.final` → `{{ agents["research.final"].report_path }}`, and `%name:0n.cld` → `{{ agents["0n.cld"].report_path }}`. Identifier-safe keys also support attribute access such as `{{ agents.build.report_path }}`. `agents` is a reserved agent-run Jinja name; a workflow input named `agents` collides and fails clearly. Output variables are persisted in the producer's `agent_meta.json` and also appear in ACE's Agents-tab `OUTPUT VARIABLES` metadata section. They are visible metadata, not secret storage.

`STOP` is a reserved output-variable name, but only for `%repeat` / `%r` chain continuation: setting it stops later repeat slots (see Stopping a repeat chain early with `STOP`). It has no special meaning for ordinary `%wait` consumers, `---` segments, or `%alt` fan-outs — those read it like any other variable, e.g. `{{ agents["name"].STOP }}`.

ACE renders loaded literal `---` multi-agent prompts as a prompt stack: each top-level segment becomes an editable pane, while prompt-level frontmatter and fenced-code separators keep the same parsing rules described below. A `#name` multi-agent xprompt invocation remains a single pane until launch. During live editing, typed `---` lines are ordinary prompt text; add panes explicitly from the prompt-stack controls. Stash restore and marked-agent kill-and-edit can also seed multiple panes, but those paths preserve each selected draft or agent prompt as one pane. Use `Enter` to choose how to submit stacked panes, `g<enter>` to launch the selected pane directly, or `Ctrl+S` to submit the panes together in top-to-bottom order. See the ACE prompt-stack guide (https://sase.sh/ace/#prompt-stacks) for the editing keybindings and the default active-pane behavior.

7.24.6 Rules

- The first `---` pair at the start of the document is treated as YAML frontmatter.
- After frontmatter is consumed, all subsequent `---` lines are segment separators.
- If there is no frontmatter, ALL `---` lines are segment separators.
- A prompt with frontmatter but only one segment is a single-agent prompt with local xprompts (not multi-agent).
- `---` inside fenced code blocks is not treated as a separator.
- When a multi-agent prompt is saved to prompt history, each individual segment is also saved as a separate entry. This allows segments to appear independently in the prompt history picker for reuse.

7.24.7 Multi-Agent XPrompts (Library-Defined Fan-Out)

An xprompt itself can be a "multi-agent xprompt": its body contains `---` separators (outside fenced blocks), and referencing it as the sole content of a user-prompt segment fans the call out into one agent per body segment. The spawned agents share the same input arguments — each segment is rendered with the same `(args)` substituted in. The catalog, TUI picker, and completion UI display markdown-defined fan-out xprompts with the inline marker (`#name`). The older `#!name` form is still recognized for multi-agent xprompts for compatibility, but new prompts should use `#name`.

```
# xprompts/three_phase.md
---
input:
  target: word
---
%name:plan
Draft a plan for {{ target }}.
---
%name:code
%wait:plan
Implement {{ target }} following the plan.
---
%name:review
%wait:code
Review the {{ target }} implementation and propose follow-ups.
```

Invoking it:

```
sase run '#three_phase(login)'
```

…dispatches three agents (`plan` , `code` , `review`), each receiving `target=login` . The `%wait` directives chain them sequentially; without `%wait` they would run in parallel.

Detection happens at dispatch time (after standard `parse_multi_prompt`), in `src/sase/agent/multi_agent_xprompt.py` , and applies at every dispatch site (`sase run` , the TUI agent launcher, the query handler).

Multi-agent xprompts can also be embedded inside a larger prompt. In that case, the first rendered body segment is embedded at the reference location and the remaining rendered body segments become follow-up agent prompts:

```
sase run '#gh:sase Review this first: #three_phase(login)'
```

When the call site starts with a VCS workspace reference such as `#gh:sase` , `#git:feature` , `#hg:branch` , or a known-project underscore form such as `#gh_sase` , that workspace reference is inherited by every generated follow-up segment unless the generated segment already declares its own VCS reference. Leading launch directives stay before the inherited workspace reference, so a prompt like `%name:abq #gh:sase #three_phase(login)` keeps `%name:abq` attached to the first generated segment and prefixes `#gh:sase` onto follow-ups.

Rules and Limitations

- A sole multi-agent reference replaces the whole segment with its generated segments. An embedded multi-agent reference replaces only that reference with the first generated segment, then appends the remaining generated segments as follow-ups.
- A user-prompt segment can contain multiple multi-agent xprompt references. They expand fully in document order. Text before the first reference attaches to the first generated segment only; text between references and after the last reference is discarded.
- Ordinary inline xprompt references inside a multi-agent xprompt body remain inline xprompt references; the agent runner expands them later as normal prompt text.
- `---` inside fenced code blocks in the xprompt body is not treated as a separator.
- Recursive fan-out (a multi-agent xprompt body whose own segments reference more multi-agent xprompts) is bounded by a depth cap and will raise if exceeded.

7.25 Relationship to Workflows

XPrompts and workflows (https://sase.sh/workflow_spec/) share the same argument grammar, but the marker communicates how the reference is allowed to participate in a prompt:

- `#name(args)` expands inline-capable xprompts and workflows with a `prompt_part` step, including markdown-defined multi-agent xprompts that fan out into multiple prompt segments.
- `#!name(args)` launches standalone YAML workflows that have no `prompt_part` step.

Simple markdown xprompts are converted internally to single-step workflows with a `prompt_part` step, so they remain inline-capable and continue to use `#name`, even when their body contains top-level `---` segment separators.

YAML workflow files can set a top-level `description` and use the same input-description forms as markdown or config-defined xprompts:

```yaml
description: Refresh generated docs and report drift.
input:
  docs_dir:
    type: path
    description: Documentation root to refresh.
steps:
  - name: refresh
    bash: just docs
```

Workflow agent steps can embed xprompt references inline:

```yaml
steps:
  - name: review
    agent: |
      #mentor(prompt=[[Review error handling]])
```

See the Workflow Specification (https://sase.sh/workflow_spec/) for full details on multi-step workflows, control flow, parallel execution, and human-in-the-loop approval.

Prompt History

Every prompt you submit to `sase run` (and every launch started from ACE (https://sase.sh/ace/)) is recorded in the prompt-history store at `~/.sase/prompt_history.json`. The `sase prompt` command group is the first-class way to inspect, search, reuse, curate, and clean up that history. It is built to feel like a well-made shell-history tool: fast to scan, exact when printing text, safe around destructive actions, and scriptable through stable JSON.

`sase prompt` reads and writes the existing JSON store directly — there is no separate database to manage. Replay commands (`run`, `edit`, `select`) route through the same launch machinery as `sase run`, so foreground, `--daemon`, multi-prompt, multi-model, and xprompt behavior stay identical.

Selectors

Prompts are addressed by a **stable content ID** derived from the exact prompt text, not by a recency index that would change every time you launch something new:

- `ph_<sha256[:12]>` — the canonical ID printed by `sase prompt list`.
- A bare hash prefix (for example `8f3a9c0d`) — any unambiguous leading slice of the SHA-256 digest.
- `sha256:<full_hash>` — the fully qualified digest when you need to be unambiguous.

If a short prefix matches more than one prompt, the command prints the colliding IDs and asks for a longer selector. Adding newer prompts never changes an existing prompt's ID.

Command Inventory

Command	Purpose
`sase prompt list`	List recent prompts as a Rich table (default) or stable JSON (`-j`).
`sase prompt show`	Print one prompt as raw text, Markdown, or JSON.
`sase prompt search`	Find prompts matching a query across repo SDD snapshots and local history.
`sase prompt stats`	Summarize the store: counts, size, length percentiles, largest prompts, top chips.
`sase prompt copy`	Copy a prompt's exact text to the system clipboard.
`sase prompt run`	Replay a stored prompt through the normal launch path.
`sase prompt edit`	Open a stored prompt in the editor, then launch the edited text.
`sase prompt select`	Pick a prompt with an `fzf` picker, then launch it.
`sase prompt doctor`	Read-only health report for the store (parseability, duplicates, oversized, …).
`sase prompt delete`	Remove exactly one prompt by selector.
`sase prompt prune`	Remove prompts by objective criteria (`--keep`, `--before`, `--cancelled`).
`sase prompt save`	Save a prompt as a reusable xprompt (https://sase.sh/xprompt/) markdown file.
`sase prompt export`	Export a prompt to stdout, a file, or an SDD (https://sase.sh/sdd/) snapshot.

`list`, `show`, `search`, `stats`, and `doctor` are read-only. `list`, `search` (default `compact` format), `stats`, and `doctor` never print full prompt text — they show previews only — so they stay bounded even on a history with thousands of entries. `show`, `export`, `copy`, and `search -f json|full` are the intentional full-text escape hatches.

Run `sase prompt <command> --help` for the full flag list of any subcommand.

Search

`sase prompt list -q` filters the **local history only**. `sase prompt search` is the broader tool: it searches **two stores at once** and ranks repo-relevant snapshots first, so it answers "I remember a prompt about X — find it, whether I snapshotted it into this repo or just ran it once last month."

- **Repo SDD snapshots** — the committed `sdd/prompts/**/*.md` files written by `sase prompt export --sdd` (plus the legacy root `prompts/` and local `.sase/sdd/prompts/` layouts). These are curated and repo-specific, so they always **rank first**.
- **Local prompt history** — the machine-wide `~/.sase/prompt_history.json` store: every prompt ever submitted on this machine, across all repos.

Matching is a **case-insensitive substring** test of the literal query (no regex or globbing) against every human-readable field — title, body, locator/ID, snapshot path, `plan:` link, and tags — so each hit can report *why* it matched. Results are ranked deterministically: SDD before local, a title/locator/path hit before a body-only hit, newer before older, with a stable tiebreak so output is byte-identical across runs. `search` is **read-only**: it never writes or locks either store, so it is safe to run against a corrupt or unreadable history.

```
sase prompt search auth                 # both stores, compact, most-relevant first
sase prompt search auth -s sdd          # only repo SDD snapshots
sase prompt search auth -s local -x     # only cancelled local prompts
sase prompt search auth -t review       # only prompts tagged "review" (repeatable; ORs)
```

Output is bounded by default

The local history can hold tens of thousands of entries, so `search` shows the **20 best matches by default** and never silently truncates — every format reports the full match count, and `-n 0` returns an unlimited result set.

`-f compact` (default) groups hits by source under dim `── SDD prompts (N) ──` / `── Local history (N) ──` headers, prints one entry per hit with the matched term highlighted plus a one-line snippet (or a `plan: "…"` / `tag: "…"` line when the match was off-body), and ends with a footer such as `27 matches (3 SDD · 24 local) · showing 20`.

`-f json` emits a stable, never-colored envelope — `query`, `count`, `total`, per-source `counts`, and a `results` array carrying the full prompt `text` — so editors and scripts can build on it. `count` versus `total` makes truncation explicit:

```
sase prompt search auth -f json | jq '.total, (.results | length)'
```

`-f full` prints each hit completely, divider-separated: a **local** hit reuses the exact `sase prompt show -f markdown` rendering, and an **SDD** hit shows a compact metadata header (path, `plan`, tags) plus its body with the match highlighted.

Filtering by date, tag, source, and status

- `-a|--after` / `-b|--before` keep prompts within an inclusive date window. Both accept `2026-06-01`, `202606` (`YYYYMM`), `260601` (`YYmmdd`), a full `260601_143000` SASE timestamp, or a relative offset `30d` / `2w` / `6m` / `1y`; an unparseable date is a usage error. SDD snapshots are dated by their frontmatter timestamp, falling back to the `YYYYMM` path segment, then the file mtime; local prompts use their last-used time.
- `-t|--tag` keeps prompts carrying a matching tag — SDD `prompt_tags` frontmatter plus the embedded `#xprompt` chips parsed from the prompt body. Low-signal runner-control `%` directives (`%model`, `%name`, `%group`, …) are execution mechanics, not content tags, so they are deliberately excluded. Repeats OR together (`-t review -t auth` matches either).
- `-s|--source` scopes to `sdd`, `local`, or `all` (default).
- `-x|--cancelled` restricts **local** results to cancelled prompts; it has no effect on SDD snapshots, which have no cancelled state.
- `-c|--color` is `auto` (colorize on a TTY unless `NO_COLOR` is set), `always`, or `never`; JSON is never colored.

An empty or whitespace-only query is a usage error (exit `2`). A query that matches nothing is **not** an error (exit `0`): `compact` / `full` print `No prompts match "<query>".` and `json` returns an envelope with `count: 0`. When `-s all` finds the same prompt in both stores (identical text), the local copy collapses into the SDD hit, annotated `also in local history`.

Common Workflows

Find a recent prompt

`list` defaults to the 20 most recent non-cancelled prompts, newest first. Narrow it with a case-insensitive substring filter:

```
sase prompt list
sase prompt list -q auth          # only prompts whose text contains "auth"
sase prompt list -l 50            # widen the window to 50 entries
sase prompt list -j               # stable JSON for scripts and editor integrations
```

The table shows the prompt ID, last-used time, status, character count, project/xprompt/directive hint chips, and a one-line preview — never the full text.

Print raw prompt text

`show -f raw` writes the exact bytes of the prompt with no added or stripped newline, so it is safe in command substitution or a pipe:

```
sase prompt show ph_8f3a9c0d12ab -f raw
sase prompt show ph_8f3a9c0d12ab -f raw | wc -c
sase prompt show ph_8f3a9c0d12ab -f markdown   # metadata header + body
sase prompt show ph_8f3a9c0d12ab -f json       # metadata + full text
```

Rerun a prompt, optionally editing first

```
sase prompt run ph_8f3a9c0d12ab           # launch the exact prompt
sase prompt run ph_8f3a9c0d12ab -d        # launch as a detached background agent
sase prompt run ph_8f3a9c0d12ab -e        # open in the editor, then launch
sase prompt edit ph_8f3a9c0d12ab          # memorable alias for edit-before-launch
```

`edit` (and `run -e`) abort cleanly without launching if you save an empty buffer.

Replay under another VCS prefix

`--prefix` reuses the existing VCS-tag replacement logic, so a prompt captured under one workspace can be replayed under another without retyping it. Prefix replacement happens before the editor opens, so what you see is what will run:

```
sase prompt run ph_8f3a9c0d12ab -P "#gh:bob-cli"     # reuse a "#gh:sase" prompt elsewhere
sase prompt edit ph_8f3a9c0d12ab -P "#gh:bob-cli"    # adjust details after re-prefixing
```

This is the shared, drift-free implementation behind the `sase run "#vcs:ref ."` compatibility path.

Pick interactively with fzf

```
sase prompt select                # fzf over launched prompts, newest first
sase prompt select -q auth -e -d  # filter to "auth", edit the choice, launch in daemon mode
sase prompt select -a             # include cancelled prompts as candidates
```

If `fzf` is not installed, `select` points you at `sase prompt list` and `sase prompt run` instead of failing silently.

Recover a cancelled prompt

Prompts you typed but did not launch (or whose launch failed) are stored as **cancelled** so you can recover them. They are hidden by default:

```
sase prompt list -c               # only cancelled prompts
sase prompt list -a               # launched and cancelled together
sase prompt run ph_8f3a9c0d12ab   # relaunch a recovered prompt by ID
```

Delete a secret

If a prompt captured something sensitive, remove exactly that one prompt. `delete` confirms on a TTY unless you pass `--yes`; in a non-interactive shell it refuses to act without `--yes`:

```
sase prompt delete ph_8f3a9c0d12ab        # confirms before removing
sase prompt delete ph_8f3a9c0d12ab -y     # skip the prompt (e.g. in scripts)
```

Because IDs are content-addressed, deleting a prompt removes every stored copy of that exact text.

Prune cancelled or old prompts

`prune` cleans up by objective criteria. Predicates intersect, and `--keep` is a hard floor that always preserves the newest N entries. Always preview with `--dry-run` first:

```
sase prompt prune -c --dry-run           # show which cancelled prompts would go
sase prompt prune -c -y                  # remove all cancelled prompts
sase prompt prune -k 500 -y              # keep only the 500 newest prompts
sase prompt prune -b 2026-01-01 --dry-run # preview removing entries older than a date
```

`--before` accepts `YYYY-MM-DD`, `YYmmdd`, or a full `YYmmdd_HHMMSS` SASE timestamp. A dry run never mutates the store, and neither `delete` nor `prune` will rewrite a corrupt or unreadable store.

Save a prompt as a reusable xprompt

Bridge a useful one-off prompt into a durable xprompt (https://sase.sh/xprompt/) so you can trigger it with `#name`:

```
sase prompt save ph_8f3a9c0d12ab -n fix-auth-review -t review
sase run "#fix-auth-review"              # the existing loader resolves it

sase prompt save ph_8f3a9c0d12ab -g      # write to ~/.xprompts/ instead of ./.xprompts/
sase prompt save ph_8f3a9c0d12ab -p bob  # namespace under ~/.config/sase/xprompts/bob/
```

With no `--name`, `save` derives a deterministic slug from the prompt preview. It never overwrites an existing xprompt file unless you pass `--force`.

Export a prompt to SDD

`export` snapshots a prompt as a committed artifact. `--sdd` writes under `sdd/prompts/YYYYMM/` with provenance frontmatter (ID, hash, timestamps, status, and source) and a filename built from a clean preview slug plus the prompt ID:

```
sase prompt export ph_8f3a9c0d12ab -s             # SDD snapshot under sdd/prompts/YYYYMM/
sase prompt export ph_8f3a9c0d12ab -o prompt.md   # write to an arbitrary path
sase prompt export ph_8f3a9c0d12ab -m             # stdout, wrapped in frontmatter
sase prompt export ph_8f3a9c0d12ab                # stdout, byte-exact (like show -f raw)
```

Existing destination files are never silently overwritten — pass `--force` to replace one.

Health and Scripting

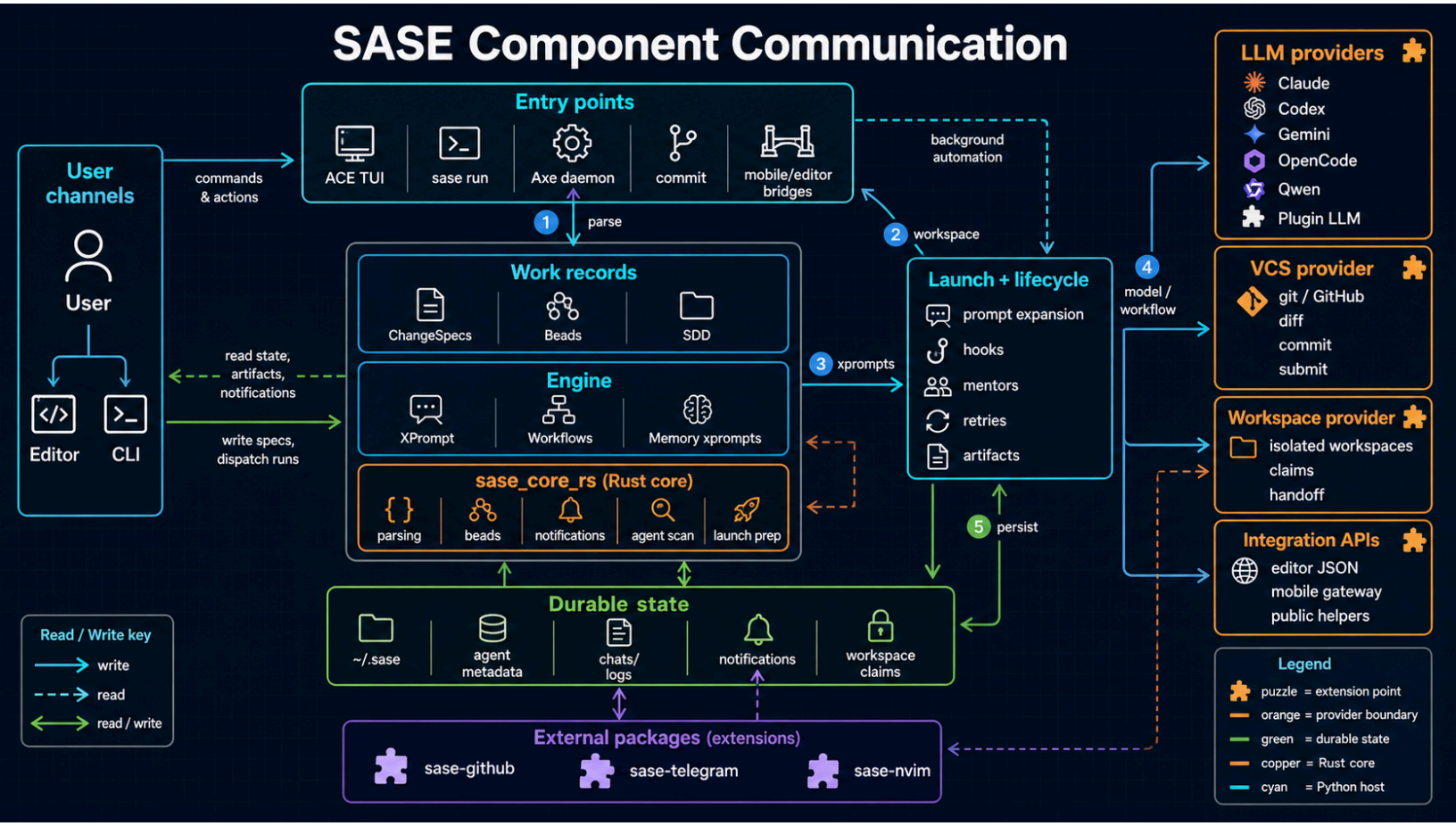
`doctor` is a read-only diagnostic that never echoes full prompt text. It reports the store path, size, and parseability; entry and cancelled counts; invalid or duplicate entries; oversized prompts; very short prompts saved through recovery paths; and whether `fzf` and a clipboard command are available:

```
sase prompt doctor          # pretty report
sase prompt doctor -j       # stable JSON for editor integrations
sase prompt stats           # length percentiles, largest prompts, top chips
sase prompt stats -j
```

The `-j` JSON output of `list`, `stats`, and `doctor` uses a stable schema and degrades cleanly under `NO_COLOR` or when piped to a non-TTY, making it safe to build tooling on top of.

Architecture

SASE is a Python orchestration layer for agentic software engineering, backed by a required Rust core for selected deterministic data operations. The system keeps work state outside any one chat transcript so agents can be launched, tracked, resumed, reviewed, retried, and handed off through stable project artifacts.



System Boundary

Area	Responsibility	Main References
CLI	Top-level `sase` commands, argument parsing, dispatch, and JSON helper bridges.	CLI reference (https://sase.sh/cli/)
ACE	Interactive TUI for ChangeSpecs, agents, notifications, artifacts, and axe status.	ACE TUI (https://sase.sh/ace/)
Axe	Background orchestrator for scheduled hooks, mentors, workflow checks, comments, cleanup, and digests.	Axe (https://sase.sh/axe/)
XPrompt	Prompt templates, reference expansion, directives, typed inputs, and reusable workflows.	XPrompts (https://sase.sh/xprompt/)
Workflows	YAML multi-step execution with agent, bash, python, parallel, loop, and human checkpoint steps.	Workflow spec (https://sase.sh/workflow_spec/)
ChangeSpecs	CL/PR-sized review records with lifecycle state, commits, hooks, comments, mentors, and timestamps.	ChangeSpecs (https://sase.sh/change_spec/)
Memory	Instruction memory, audited long-term reads, and reviewed write proposals.	Memory (https://sase.sh/memory/)
SDD	Durable prompt, tale, epic, legend, myth, and research artifacts.	SDD (https://sase.sh/sdd/)
Beads	Git-portable issue/dependency tracking and executable epic/legend launch plans.	Beads (https://sase.sh/beads/)
Providers	Pluggable LLM, VCS, workspace, config, and xprompt boundaries.	Plugins (https://sase.sh/plugins/)
Rust core	Required `sase_core_rs` extension for ported parsing, query, notification, agent scan, launch prep, and bead data operations.	Rust backend (https://sase.sh/rust_backend/)

Integrations	Public helpers and fixed bridge APIs for editors, mobile gateway, and external packages.	Integrations (https://sase.sh/integrations/)

The Python host owns user-facing orchestration, plugin calls, subprocess handling, filesystem context, TUI rendering, and workflow side effects. Rust owns reusable deterministic backend operations that need speed, stable wire contracts, or cross-frontend consistency.

Agent Launch Flow

Most agent work enters through `sase run`, ACE, axe agent chops, bead epic execution, or mobile/editor helper bridges. The launch path follows the same shape across those entry points:

1. Parse prompt text, directives, and optional multi-prompt separators.
2. Canonicalize ProjectSpec aliases in launch-bound VCS refs, so aliases such as `#gh:bob` become canonical refs such as `#gh:bob-cli` before history or artifact snapshots are written.
3. Resolve workspace references such as `#cd:<path>`, `#git:<project>`, or plugin-provided forms, rejecting inactive known projects before new work is claimed.
4. Allocate or prepare the target workspace through the workspace provider layer.
5. Expand xprompt references and standalone workflow references.
6. Invoke the selected LLM provider or workflow executor.
7. Stream subprocess output, write chat history, and persist launch metadata.
8. Record agent artifacts such as prompts, diffs, generated Markdown PDFs, images, plans, and explicit files.
9. Emit notifications and update ACE-visible status.
10. Hand review, revert, restore, or commit work to the VCS and workspace provider layers when requested.

Detached launches appear in the agent registry and ACE Agents tab. Multi-prompt launches create a sequence of detached agents. Workflow launches persist step state so ACE and axe can inspect progress and recover meaningful output.

State Model

SASE avoids making a live chat session the source of truth. The durable state lives in files and stores that can be inspected by users, agents, and automation:

State	Location / Owner	Use
ChangeSpecs	Project `.sase` files under `~/.sase/projects/`	Project lifecycle state plus review lifecycle, commits, hooks, comments, mentors, dependencies, and timestamps.
Agent metadata	Agent artifact directories under `~/.sase/`	Running/completed status, prompt snapshots, output, diffs, workflow state, and attachments.
Agent archives	`~/.sase/dismissed_bundles/` and `~/.sase/dismissed_agent_groups/`	Dismissed-agent recovery bundles and named groups for later ACE revival.
SDD artifacts	`sdd/` or `.sase/sdd/`	Prompt snapshots, plans, executable epics, legends, myths, and research notes.

Beads	`sdd/beads/` or `.sase/sdd/beads/`	Issue graph, JSONL export, SQLite query cache, epic/legend execution metadata.
Memory context	`memory/`, `~/.sase/projects/<project>/`	Agent instructions, audited reads, and write proposals.
Configuration	`~/.config/sase/sase.yml`, overlays, optional project-local config	Provider selection, axe jobs, mentors, xprompts, telemetry, mobile gateway, and defaults.
Notifications	Notification store facade backed by Rust operations	User-visible actions, unread state, agent completion, errors, and mobile events.
Workspace claims	Running-field state and provider metadata	Reservation and release of numbered workspaces for parallel agents.

`~/.sase` is the default SASE state root. Set `SASE_HOME` to move that root for isolated tests, alternate profiles, or containerized runs.

This model lets ACE, CLI commands, axe, and future frontends read the same engineering state without depending on one terminal session.

Provider Boundaries

Provider abstractions keep SASE above any single agent runtime, version-control host, or workspace strategy:

Provider Layer	What It Owns	Details
LLM provider	Agent CLI selection, concrete model mapping, subprocess invocation, retry defaults, usage metadata.	LLM providers (https://sase.sh/llms/)
VCS provider	Diff, checkout, commit, amend, proposal/PR dispatch, reword, submit, sync, revert, restore, and review metadata.	VCS providers (https://sase.sh/vcs/)
Workspace provider	Workspace reference resolution, workspace directory allocation, submit/mail preparation, workflow metadata.	Workspaces (https://sase.sh/workspace/)
Resource plugins	Extra xprompt/workflow files and default configuration.	Plugins (https://sase.sh/plugins/)
Integration APIs	Public Python helpers and fixed JSON bridge contracts for companion tools.	Integrations (https://sase.sh/integrations/)

Core SASE ships built-in providers for common local use: bundled LLM provider entry points, plain-git VCS support, bare-git workspaces, and `#cd` directory runs. Optional packages can add hosted VCS workflows, notification delivery, editor integrations, or extra prompt resources.

Rust Core Boundary

The required `sase_core_rs` extension is the shared backend boundary for deterministic logic that benefits from a stable wire contract or from being reused by non-Python frontends. Current Rust-backed areas include:

- ChangeSpec parsing and batch query operations.
- Project lifecycle parsing, update planning, and lifecycle-filtered project listing.
- Status transition planning.

- Git command output parsing.
- Notification JSONL reads and mutations.
- Agent artifact scanning and persistent indexing.
- Agent launch preparation, timestamp allocation, fan-out planning, low-level detached spawn, and workspace-claim planning.
- Bead read, mutation, JSONL, SQLite, single-store ID allocation, and deterministic work-plan operations.

The Python host still owns side effects that require app context: plugin dispatch, VCS/workspace calls, process signalling, file locks, TUI rendering, user confirmation, xprompt lookup, and workflow orchestration. See Rust backend (https://sase.sh/rust_backend/) for the complete operation list and facade map.

Read Next

Need	Page
Command discovery	CLI reference (https://sase.sh/cli/)
Contributor setup and source orientation	Development (https://sase.sh/development/)
Runtime operations	ACE (https://sase.sh/ace/), Axe (https://sase.sh/axe/), notifications (https://sase.sh/notifications/)
Durable work records	ChangeSpecs (https://sase.sh/change_spec/), memory (https://sase.sh/memory/), SDD (https://sase.sh/sdd/), beads (https://sase.sh/beads/)
Prompt and workflow execution	XPrompts (https://sase.sh/xprompt/), workflow spec (https://sase.sh/workflow_spec/)
Extension boundaries	Plugins (https://sase.sh/plugins/), LLM providers (https://sase.sh/llms/), VCS providers (https://sase.sh/vcs/), workspaces (https://sase.sh/workspace/)
Backend boundary	Rust backend (https://sase.sh/rust_backend/)

8 ChangeSpec Format Documentation

A **ChangeSpec** is a structured record for one change list (CL) or pull request (PR). It lives inside a project `.sase` file and records the change's description, dependency metadata, review URL, lifecycle status, commits, hooks, comments, mentor runs, timestamps, and computed file deltas.

8.1 Format Overview

Each ChangeSpec is a block of top-level fields and optional sections. `NAME`, `DESCRIPTION`, and `STATUS` are the normal minimum for a hand-written entry; `sase commit` creates and updates most other sections automatically.

The canonical order is:

```
NAME: <NAME>
DESCRIPTION:
  <TITLE>

  <BODY>
PARENT: <PARENT>
BUG: <BUG>
CL: <CL>
STATUS: <STATUS>
COMMITS:
  <COMMIT_ENTRIES>
DELTAS:
  <DELTA_ENTRIES>
HOOKS:
  <HOOK_ENTRIES>
COMMENTS:
  <COMMENT_ENTRIES>
MENTORS:
  <MENTOR_ENTRIES>
TIMESTAMPS:
  <TIMESTAMP_ENTRIES>
```

The parser is tolerant of some older ordering and timestamp variants, but new docs, scripts, and examples should use the order above. See individual field specifications below for optionality and automatic behavior.

IMPORTANT: When outputting multiple ChangeSpecs, separate each one with **two blank lines**.

8.2 Field Specifications

8.2.1 NAME

The unique identifier for the ChangeSpec.

Recommended format: `<project_or_area>_<descriptive_suffix>`

- Prefer a project- or area-specific prefix followed by an underscore
- Suffix should use underscores to separate words
- Suffix should be descriptive but concise
- `sase commit` appends a numeric suffix such as `_1` when it needs to make a new name unique

Examples:

- `my_project_add_config_parser`
- `feature_x_implement_validation`
- `refactor_database_layer`

8.2.2 DESCRIPTION

A comprehensive description of what the CL does and why.

Structure:

1. **TITLE** (first line): A brief one-line summary
2. **Blank line**: Always include one blank line after the title
3. **BODY** (remaining lines): Detailed multi-line description

Formatting:

- **All lines must be 2-space indented** (including the blank line)
- TITLE should be concise (one line)
- BODY should include:
- What changes are being made
- Why the changes are needed
- High-level approach or implementation details
- What will be tested (if applicable)

PR tag stripping: When a ChangeSpec is created from a PR workflow or its description is synced after a reword, any trailing `KEY=VALUE` metadata lines (matching `^[A-Z][A-Z0-9_]*=`) are automatically stripped. This prevents provider-specific tags like `AUTOSUBMIT_BEHAVIOR=SYNC_SUBMIT` or `MARKDOWN=true` from polluting the description. See commit_workflows.md (https://sase.sh/commit_workflows/#pull-request-pr) for details.

Example:

```
DESCRIPTION:
  Add configuration file parser for user settings

  This CL implements a YAML-based configuration parser that reads
  user settings from ~/.myapp/config.yaml. The parser includes a
  ConfigParser class with load() and validate() methods, along with
  type definitions for the configuration schema. Tests will cover
  valid YAML parsing, invalid config validation, and missing file
  handling.
```

8.2.3 PARENT

Specifies the dependency relationship between ChangeSpecs.

Values:

- Omit this field entirely - This CL has no dependencies (default, preferred for parallelization)
- `<parent_changespec_name>` - The `NAME` of a parent ChangeSpec that must be completed first

The PARENT field is a ChangeSpec **name** — never a VCS ref. Values like `origin/main`, `origin/master`, or the Mercurial sentinel `p4head` are not valid here; they describe checkout targets for the VCS, not dependency relationships between CLs. "No parent ChangeSpec" is represented by omitting the field entirely. `sase commit` drops the PARENT field and warns when the value passed via `-p` does not resolve to an existing ChangeSpec.

Auto-detection: When creating a new ChangeSpec via `sase commit`, the PARENT field is automatically set if the current branch corresponds to an existing ChangeSpec. This can be overridden with the `-p` / `--parent` flag (see commit_workflows.md (https://sase.sh/commit_workflows/) for details).

CRITICAL Dependency Guidelines:

- **Default to omitting PARENT** to maximize parallel development
- **Only set a PARENT when there's a real content dependency**:
- CL B calls a function/class that CL A creates
- CL B modifies a file that CL A creates
- CL B extends functionality that CL A introduces
- **DO NOT set a PARENT for**:
- Independent features that don't interact
- Changes to different files/modules
- Tests for independent features
- Documentation that doesn't reference new code

Examples:

```
# No PARENT field = no dependencies (preferred)
PARENT: my_project_add_config_parser   # Depends on another CL
```

8.2.4 CL / PR

The CL or PR identifier (e.g., CL number or PR URL). Both `CL:` and `PR:` are accepted and treated identically — use whichever matches your project's terminology.

Values:

- Omit this field entirely - CL/PR not yet created (initial state)
- `http://cl/<CL_ID>` - URL to the created CL
- `https://github.com/<owner>/<repo>/pull/<N>` - URL to the PR

Example:

```
# No CL field = CL not yet created
CL: http://cl/12345        # After CL creation
PR: https://github.com/org/repo/pull/42   # PR variant
```

8.2.5 BUG

An optional bug reference linking the CL to an issue tracker. SASE stores this as plain text. PR workflows that receive `SASE_BUG_ID` or `sase commit --bug-id` write it as `http://b/<id>` in the ChangeSpec and add `BUG=<id>` to provider tag metadata.

Example:

```
BUG: http://b/12345
```

8.2.6 STATUS

The current state of the CL in its lifecycle.

Valid Values:

Status	Description
`WIP`	Work in progress — initial development
`Draft`	CL created as a draft, not yet ready for review
`Ready`	Ready for review
`Mailed`	Sent out for review
`Submitted`	Merged / submitted to the codebase (terminal)
`Reverted`	CL was reverted after submission (terminal)
`Archived`	CL was abandoned without submission (terminal)

Valid Transitions:

```
WIP → Draft, Ready
Draft → Ready
Ready → Mailed, Draft
Mailed → Submitted
Submitted → (terminal)
Reverted → (terminal)
Archived → (terminal)
```

These transitions are enforced by the status state machine. Terminal statuses are moved to the archive project file.

Status Selection Rules:

- New CLs typically start as `WIP`
- PR workflows default new ChangeSpecs to `Draft` unless `sase commit --status` or `SASE_PR_STATUS` says otherwise
- Move to `Ready` when the CL is ready for review
- Move to `Mailed` when sent out for review
- Update status as work progresses through the lifecycle

8.2.7 COMMITS

Tracks the commit history associated with this CL. This section is managed automatically by `sase commit`.

Entry format:

```
COMMITS:
  (1) First commit note
      | CHAT: ~/.sase/chats/mybranch-commit-260328_143052.md (2m15s)
      | DIFF: ~/.sase/diffs/mybranch-260328_143052.diff
      | PLAN: sdd/tales/202603/my_plan.md
  (2) Second commit note
      Multi-line body continues here with 6-space indent.
      Blank body lines use a dot (.) placeholder.
      .
      Another paragraph after the blank line.
      | CHAT: ~/.sase/chats/mybranch-commit-260328_153012.md (1m42s)
      | DIFF: ~/.sase/diffs/mybranch-260328_153012.diff
  (2a) Proposed alternative - (!: NEW PROPOSAL)
      | DIFF: ~/.sase/diffs/mybranch-260328_160000.diff
```

Entry numbering:

- Regular entries use sequential integers: `(1)`, `(2)`, `(3)`, ...
- Proposal entries use the last regular number plus a letter suffix: `(2a)`, `(2b)`, ...
- Proposals are marked with `(!: NEW PROPOSAL)` to flag them for review.

Multi-line body: The first line of the commit message becomes the note. Subsequent paragraphs (separated by a blank line in the original message) become 6-space-indented body lines below the note. Empty body lines are stored as a dot (`.`) placeholder to preserve structure.

Drawers: Each entry can have zero or more drawer lines (6-space indent, `|` prefix):

Drawer	Format	Description
`CHAT`	`| CHAT: <path> (<duration>)`	Agent chat log file with optional run duration
`DIFF`	`| DIFF: <path>`	Saved diff file
`PLAN`	`| PLAN: <path>`	Plan file associated with this commit (via SDD)

The CHAT drawer's duration (e.g., `2m15s`) is calculated from the chat filename timestamp to the commit time. The PLAN drawer is emitted when the `SASE_PLAN` environment variable is set during the commit workflow.

8.2.8 TIMESTAMPS

Records a chronological audit trail of lifecycle events. Each entry includes a timestamp, event type, and detail string.

Entry format:

```
TIMESTAMPS:
  [260328_143052] COMMIT  (1)
  [260328_151203] STATUS  WIP -> Draft
  [260328_151510] SYNC    Synced with remote
  [260328_160044] REWORD  Updated description title
  [260328_163012] REWIND  (2)
  [260328_170100] RENAME  old_name -> new_name
  [260328_171500] REBASE  old_parent -> new_parent
```

Event types:

Type	Description
`COMMIT`	A commit was added to the ChangeSpec; detail is usually `(N)`
`STATUS`	A status transition occurred (e.g., `WIP -> Draft`)
`SYNC`	A sync operation was performed
`REWORD`	The description or PR-derived metadata was edited
`REWIND`	A rewind to a previous commit entry occurred; detail shows `(N)`
`RENAME`	The ChangeSpec name changed; detail records `old -> new`
`REBASE`	The parent relationship changed; detail records `old -> new`

New entries use the format `[YYMMDD_HHMMSS]` in the configured SASE timezone. The parser also accepts older bare `YYMMDD_HHMMSS` and `[YYYY-MM-DD HH:MM:SS]` forms for compatibility. TIMESTAMPS are recorded atomically by SASE and are not normally edited by hand. Multiple events of the same type may appear.

8.2.9 DELTAS

A computed summary of files added, modified, or deleted by this CL relative to its parent. The section is maintained automatically by sase from VCS state — it is not edited by hand.

Entry format:

```
DELTAS:
  + path/to/added_file.py
      | LINES: +128
  ~ path/to/modified_file.py
      | LINES: +12 ~7 -3
  - path/to/deleted_file.py
      | LINES: -44
```

The optional `LINES` drawer records semantic line counts. Git-style raw additions/deletions are converted so paired add/delete lines are shown as modified lines (`~N`); binary files use `LINES: binary`. Older ChangeSpecs without `LINES` drawers remain valid.

Glyph	Change type	Notes
`+`	Added	File introduced by this CL (`A` from VCS); copies are represented as added target files.
`~`	Modified	File edited (`M`); typechange, unmerged, or future statuses are coerced to modified.
`-`	Deleted	File removed (`D`).

Renames (VCS status `R`) are split into a `-` for the source path and a `+` for the target path. Line counts attach to the target path when the VCS reports them; a pure rename can therefore show `0 lines`. Entries are sorted alphabetically by path. The section is omitted entirely when there are no deltas.

When DELTAS is recomputed: refresh hooks run after commit creation, rewind, sync, proposal accept, and proposal rebase. The refresh is best-effort — if the required VCS query fails, the existing DELTAS section is left untouched and the parent workflow proceeds. Providers without line stats still refresh file-level DELTAS.

Manual refresh: run `sase changespec sync-deltas -c <CL_NAME>` to recompute DELTAS for a single ChangeSpec from the current VCS state. Optional `-p/--project-file` and `-w/--workspace-dir` flags override the inferred defaults.

In ACE, DELTAS renders with colored glyphs (green `+`, gold `~`, red `-`). The section has two semantic fold states: folded and unfolded. The folded state shows the `DELTAS:` header plus a one-line file and line-count summary; the unfolded state shows the full alphabetical entry list with inline line tokens. The shared fold model still has an internal intermediate value for other sections, but DELTAS normalizes any non-folded value to the unfolded full list.

8.2.10 HOOKS

Defines lifecycle hooks attached to this CL — shell commands that run automatically at specific points (e.g., after commit, before mail). Hooks are managed via the `h` keybinding in ACE.

Entry format:

```
HOOKS:
  just test
      | (1) [260328_143200] PASSED (12s)
      | (2) [260328_153300] FAILED (8s) - (!: Hook Command Failed)
```

Hook commands are 2-space indented. Status drawer lines are 6-space indented and start with `|`. A leading `!` on a hook command means failed runs should skip fix-hook hints; a leading `$` means the hook is not run for proposal entries and is not subject to the normal runner limit. Prefixes can be combined, for example `!$just presubmit`.

8.2.11 COMMENTS

Stores review comments and discussion threads. Comments are added via the ACE TUI or through the review workflow.

Entry format:

```
COMMENTS:
  [critique] ~/.sase/comments/auth_system_fix-critique-260328_143500.json
  [critique] ~/.sase/comments/auth_system_fix-critique-260328_150000.json - (!: Unresolved Critique Comments)
```

8.2.12 MENTORS

Configures mentor workflows for the CL — automated agents that monitor and provide guidance during development.

Entry format:

```
MENTORS:
  (1) security[1/2] reliability[1/1]
      | [260328_143700] security:auth-review - PASSED - (1m05s)
      | [260328_143705] reliability:tests-review - COMMENTED - (2m10s)
```

The entry id matches a `COMMITS` entry such as `(1)` or `(2a)`. Profile names may include progress counts; legacy entries without counts still parse.

8.3 Complete Examples

8.3.1 Example 1: Independent CL with Tests

```
NAME: auth_system_add_jwt_validator
DESCRIPTION:
  Add JWT token validation for authentication

  This CL implements JWT token validation using the PyJWT library.
  It includes a JWTValidator class that handles token parsing,
  signature verification, and expiration checking. The implementation
  supports both RS256 and HS256 algorithms. Tests cover valid tokens,
  expired tokens, invalid signatures, and malformed tokens.
STATUS: WIP
```

8.3.2 Example 2: Dependent CL

```
NAME: auth_system_integrate_validator
DESCRIPTION:
  Integrate JWT validator into authentication middleware

  This CL integrates the JWT validator from the previous CL into
  the main authentication middleware. The middleware will validate
  tokens on protected routes and handle validation errors gracefully.
  Tests verify both successful authentication and various failure
  scenarios including missing tokens, expired tokens, and invalid
  signatures.
PARENT: auth_system_add_jwt_validator
STATUS: WIP
```

8.3.3 Example 3: Config-Only CL (No Tests)

```
NAME: auth_system_update_config
DESCRIPTION:
  Update JWT configuration with new secret key

  This CL updates the production configuration file to use a new
  secret key for JWT signing. This is a config-only change that
  rotates the signing key for security purposes.
STATUS: WIP
```

8.3.4 Example 4: CL with Bug Reference

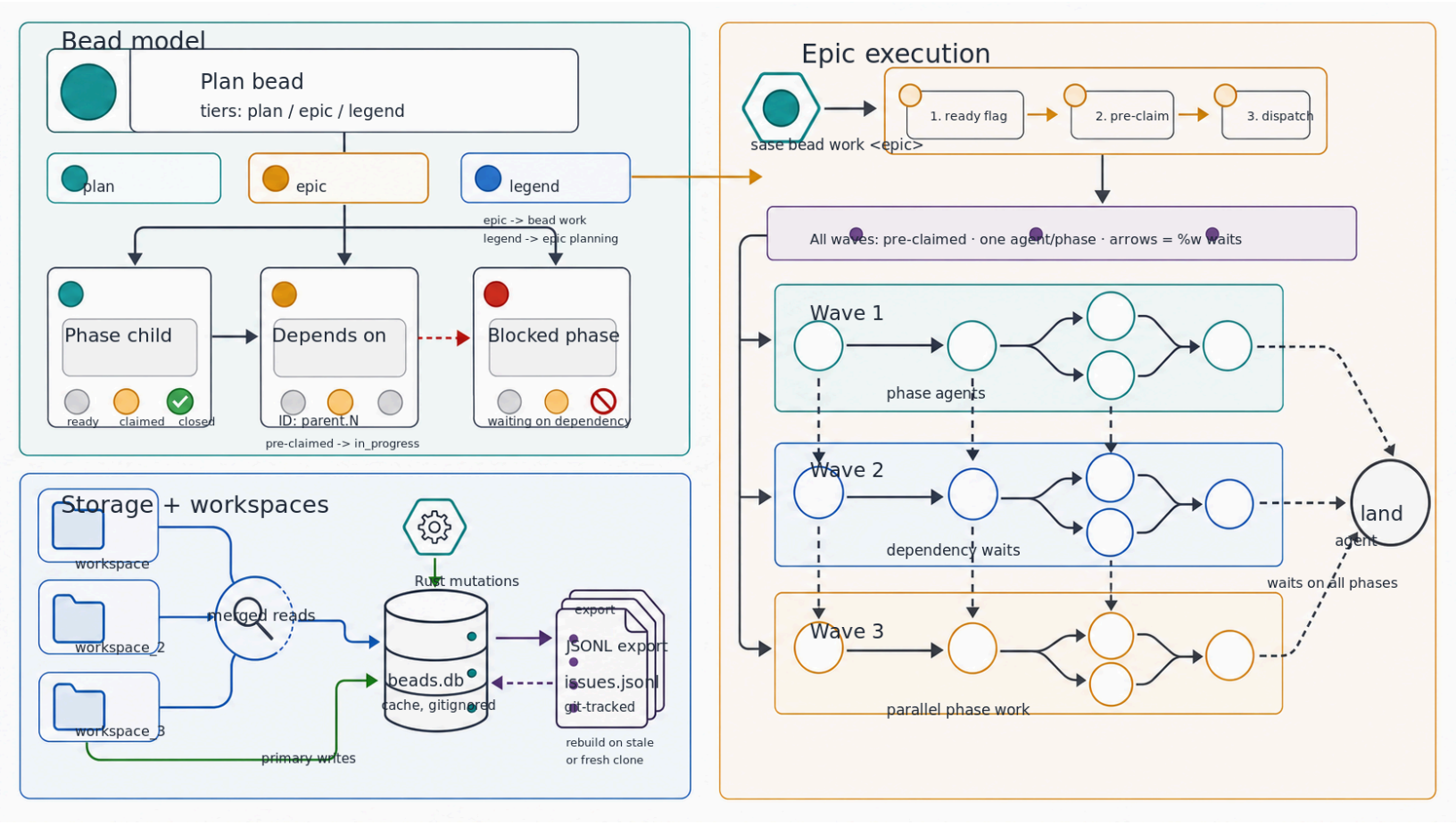
```
NAME: auth_system_fix_token_expiry
DESCRIPTION:
  Fix incorrect token expiry calculation

  The token expiry was being computed from the issue time rather
  than the current time, causing tokens to expire prematurely
  under clock skew conditions.
BUG: http://b/98765
STATUS: Draft
```

8.4 Best Practices

1. **Keep CLs Small and Focused**: Each CL should address a single, well-defined change
2. **Maximize Parallelization**: Omit `PARENT` whenever possible
3. **Include Tests**: Attach relevant test commands in `HOOKS`
4. **Write Clear Descriptions**: Explain what, why, and how
5. **Use Descriptive Names**: NAME should clearly indicate what the CL does
6. **Think About Dependencies**: Only create dependencies when truly necessary
7. **Update Status Appropriately**: Keep STATUS field current as work progresses

9 Bead Issue Tracking

Bead is a lightweight, git-native issue tracking system built into sase. It uses Rust-backed event storage, query/reduction, and mutation logic through the required `sase_core_rs` extension, with generated JSONL compatibility projections for older tooling (inspired by Fossil (https://fossil-scm.org/)). Issues are organized into plan-like containers and executable child phases. Plan beads can represent ordinary plans, executable epics, or legend-level roadmaps through their `tier` metadata.



9.1 Table of Contents

- Quick Start
- Data Model
- Issue Types
- Status Lifecycle
- Dependencies
- Storage
- Directory Structure
- Event Log + Compatibility Projections
- Sync Mechanism
- CLI Commands
- Rust Backend
- Current Checkout Source Of Truth
- ACE TUI Integration

9.2 Quick Start

```
sase bead init                                          # Initialize beads in current project
sase bead create -t "New feature" --type "plan(sdd/tales/202605/feature.md)" --tier plan
sase bead create -t "Epic" --type "plan(sdd/epics/202605/epic.md)" --tier epic
sase bead create -t "Roadmap" --type "plan(sdd/legends/202605/roadmap.md)" --tier legend --epic-count 3
sase bead create -t "Linked epic" --type "plan(sdd/epics/202605/epic.md,<legend-id>)" --tier epic
sase bead create -t "Sub-task" --type "phase(beads-001)"   # Create a phase under a plan
sase bead list                                          # List open and in-progress issues
sase bead list --status=open                            # List open issues
sase bead list --status=closed                          # List closed issues
sase bead search auth                                   # Search open, in-progress, and closed issues
sase bead ready                                         # Show issues ready to work on
sase bead show beads-001                                # View issue details
sase bead update beads-001.1 --status=in_progress       # Claim an issue
sase bead open beads-001.1                              # Reopen an issue
sase bead close beads-001.1                             # Close an issue
sase bead dep add beads-001.2 beads-001.1               # Add dependency
sase bead blocked                                       # Show blocked issues
sase bead sync                                          # Export and stage JSONL in git
sase bead stats                                         # Project statistics
sase bead doctor                                        # Health check
sase bead work beads-001                                # Launch agents for an epic or legend plan bead
```

9.3 Data Model

9.3.1 Issue Types

Type	Description	ID Format
Plan	Plan-like container with a tier	`{prefix}-{counter}`
Phase	Executable task within an epic/plan bead	`{parent_id}.{N}`

Plans are groupings that can optionally link to an SDD file via the `design` field. Phases always belong to a parent plan and use hierarchical IDs (e.g., `beads-001.1`, `beads-001.2`).

Plan beads carry a tier. The SDD paths below are the conventional version-controlled locations; in local SDD mode the same artifact classes live under `.sase/sdd/`.

Tier	SDD Path	Behavior
`plan`	`sdd/tales/{YYYYMM}/*.md`	Normal non-epic implementation plan
`epic`	`sdd/epics/{YYYYMM}/*.md`	Executable multi-phase plan accepted by `sase bead work`
`legend`	`sdd/legends/{YYYYMM}/*.md`	Higher-level coordination plan; launches epic-planning agents by `epic_count`

Linked epics use the existing parented plan syntax:

```
sase bead create --title "Epic" --type "plan(sdd/epics/202605/epic.md,<legend_bead_id>)" --tier epic
```

9.3.2 Status Lifecycle

Status	Icon	Description
`open`	○	Not started
`in_progress`	◐	Currently being worked on
`closed`	✓	Completed or abandoned

Status can transition freely between any values via `sase bead update --status=<status>`. `sase bead open <id>` is a shortcut for `sase bead update <id> --status=open`.

9.3.3 Dependencies

Dependencies are one-way relationships: issue A **depends on** issue B. An issue is:

- **Ready** if it is `open` and all its dependencies are `closed`.
- **Blocked** if it has at least one dependency with status `open` or `in_progress`.

9.4 Storage

9.4.1 Directory Structure

When version-controlled mode is effective (`sdd.version_controlled: true`, or any project resolved as the built-in `bare_git` VCS provider):

```
sdd/beads/
  config.json           # Configuration (issue prefix, counter, owner)
  events/
    manifest.json       # Event-store schema and migration metadata
    streams/
      <root-id>.jsonl   # Canonical append-only event stream
  issues.jsonl          # Generated compatibility projection
  beads.db              # SQLite compatibility cache (gitignored)
```

In non-version-controlled mode for other providers, the directory is `.sase/sdd/beads/` with the same structure.

Normal bead commands read and write one store for the active checkout. In version-controlled mode, canonical bead state lives in the current checkout's `sdd/beads/events/**` event store plus `sdd/beads/config.json`. If the event store is absent, reads fall back to legacy `issues.jsonl`. Numbered sibling workspaces and legacy stores are not merged into normal `sase bead` reads.

9.4.2 Event Log + Compatibility Projections

Rust owns the bead storage/query/mutation path. The append-only event streams are the canonical git-portable state. `issues.jsonl` remains a generated compatibility projection, and `beads.db` remains a local compatibility cache. They are kept in sync:

- **Writes** append canonical Rust events first, then regenerate `issues.jsonl` and refresh `beads.db`.
- **Reads** prefer `events/manifest.json` plus `events/streams/*.jsonl`, falling back to legacy `issues.jsonl` only when no event store is present.
- **Fresh clones** read directly from the tracked event streams and can rebuild the compatibility mirrors on demand.

The `.gitignore` excludes `beads.db*` files. The event store, `issues.jsonl`, and `config.json` are tracked in git.

9.4.3 Sync Mechanism

`sase bead sync` regenerates the compatibility projection from the canonical event store and stages the bead state in git, including `sdd/beads/events/**`, `issues.jsonl`, and `config.json`. The projection contains one JSON object per line, sorted by issue ID for clean diffs.

When both stores exist, the event store wins. Manual edits to `issues.jsonl` do not change command output unless the event store is absent.

9.5 CLI Commands

With no subcommand, `sase bead` defaults to `sase bead list` with default options. Use the explicit `sase bead list` form when passing list filters.

9.5.1 `sase bead init`

Initialize the bead store for the current project. In effective version-controlled SDD mode this is `sdd/beads/`; in local SDD mode this is `.sase/sdd/beads/`.

9.5.2 `sase bead create`

Create a new issue.

Flag	Required	Description
`-t, --title`	yes	Issue title
`-T, --type`	yes	Bead type: `plan(<file>)`, `plan(<file>,<parent>)`, or `phase(<parent_id>)`
`-d, --description`	no	Issue description
`-a, --assignee`	no	Assignee name
`--tier`	no	Plan-bead tier: `plan`, `epic`, or `legend`
`-c, --changespec`	no	Attach a ChangeSpec name to a plan bead
`-b, --bug-id`	no	Bug ID for the attached ChangeSpec; requires `--changespec`
`-E, --epic-count`	no	Positive number of epics proposed by a legend plan bead

`-m, --model`	no	Model used when this bead is launched. Provider-qualified (e.g. `codex/gpt-5.5`) or a configured local alias (e.g. `#pro`). On epic and legend plan beads this becomes the land-agent model; on phase beads it is the per-phase work model.

ChangeSpec metadata is valid only on plan beads. It is used by the epic-approval and `sase bead work` flows to keep plan beads linked to the ChangeSpec they are intended to produce.

9.5.3 `sase bead list`

List issues with optional filtering. Without `--status`, the command lists `open` and `in_progress` issues; pass `--status=closed` when you need closed history. `--status`, `--type`, and `--tier` are repeatable.

Flag	Values	Description
`-s, --status`	`open`, `in_progress`, `closed`	Filter by status (repeatable)
`-t, --type`	`plan`, `phase`	Filter by type (repeatable)
`--tier`	`plan`, `epic`, `legend`	Filter by plan-bead tier

9.5.4 `sase bead search <query>`

Find beads whose indexed text fields contain a case-insensitive literal substring. This is substring search, not regex or glob matching. Current indexed fields include ID, title, description, notes, design/plan path, owner, assignee, model, ChangeSpec name/bug ID, status, type, and tier; timestamps are not searched. Unlike `sase bead list`, search includes `open`, `in_progress`, and `closed` beads by default, so it is the quickest way to recover older context.

Compact output prints each matching bead with a short snippet. For multi-line fields such as descriptions or notes, the snippet uses the line that matched the query when possible instead of always showing the first line. JSON output exposes the exact `matched_fields` list for each result.

```
sase bead search auth
sase bead search auth --format json
sase bead search auth --format full --limit 3
sase bead search auth --status open --type phase
sase bead search auth --type plan --tier epic
```

Flag	Values	Description
`-c, --color`	`auto`, `always`, `never`	Color mode for compact output
`-f, --format`	`compact`, `json`, `full`	Output format; defaults to `compact`
`-n, --limit`	non-negative integer	Maximum results; omitted or `0` means unlimited
`-s, --status`	`open`, `in_progress`, `closed`	Filter by status (repeatable)
`--tier`	`plan`, `epic`, `legend`	Filter by plan-bead tier (repeatable)
`-t, --type`	`plan`, `phase`	Filter by type (repeatable)

9.5.5 `sase bead show <id>`

Display complete details for an issue including status, type, tier, epic count, parent/children, dependencies, blockers, description, notes, ChangeSpec metadata, model, and linked plan path.

9.5.6 `sase bead ready`

Show issues that are ready to work on: `open` status with all dependencies `closed`.

9.5.7 `sase bead open <id>`

Reopen an issue by setting its status to `open`. This is equivalent to `sase bead update <id> --status=open`.

9.5.8 `sase bead update <id>`

Update one or more fields on an issue.

Flag	Description
`-s, --status`	Change status
`-t, --title`	Change title
`-d, --description`	Change description
`-n, --notes`	Change notes
`-D, --design`	Change plan path
`-a, --assignee`	Change assignee
`--tier`	Change plan tier
`-E, --epic-count`	Change legend epic count
`-m, --model`	Change the launch model. Pass an empty string to clear.

9.5.9 `sase bead close <id> [<id2> ...]`

Close one or more issues.

Closing a plan bead also closes its phase children. Use this intentionally: phase agents should close only their assigned phase bead, not the parent epic.

Flag	Description
`-r, --reason`	Optional close reason text

9.5.10 `sase bead rm <id>`

Remove an issue and cascade-delete all its children. This is irreversible.

9.5.11 `sase bead dep add <issue> <depends_on>`

Add a dependency: `<issue>` depends on `<depends_on>`. The issue becomes blocked if the dependency is not yet closed.

9.5.12 `sase bead blocked`

Show all issues that have at least one active (non-closed) blocker.

9.5.13 `sase bead sync`

Regenerate the compatibility projection from the canonical event store and stage bead state in git. It does not create a commit; the staged event/projection files are included in the next normal project or SDD commit.

Flag	Description
`-s, --status`	Check whether bead state has unstaged changes

9.5.14 `sase bead stats`

Show project statistics: total, open, in-progress, and closed counts, plus plan and phase counts.

9.5.15 `sase bead doctor`

Run health checks on the beads database. Checks for:

- Missing `config.json`, event store, legacy projection, or compatibility cache
- Projection drift between canonical events and `issues.jsonl`
- Invalid events or unreduced orphan phase records
- Uncommitted bead-state changes
- Orphan children (phases whose parent plan is missing)

If bead commands fail before opening a store, run `sase core health` first. It verifies that the required `sase_core_rs` extension is importable and exposes the representative bead CLI binding used by the fast path.

9.5.16 `sase bead onboard`

Display a quick-start guide with common command examples.

9.5.17 `sase bead work <id>`

Run an entire epic-tier plan end-to-end by launching one agent per phase plus a final land agent, or run a legend-tier plan by launching one epic-planning agent per stored `epic_count`.

For epic-tier plans, the command:

1. Validates that `<epic_id>` resolves to an issue of type `plan` with `tier=epic`. If the plan is already marked `is_ready_to_work`, the command treats the run as a retry and schedules any remaining non-closed phases.
2. On a confirmed launch, force-reuses the deterministic bead-work names — `<epic_id>.<N>` (for each open phase), `<epic_id>` (for the land agent), and the legacy `<epic_id>.land` land-agent name — by wiping any prior owner of those names, whether that owner is a completed, dismissed, or planned reservation or a still-live agent (live owners are terminated). This also covers owners that hold the name only as a `workflow_name`. If the forced-reuse cleanup cannot complete (a wipe fails or a name is still reserved afterward), the command aborts before mutating any bead state. `--dry-run` performs no cleanup; it only warns which live agents a real launch would force-reuse.
3. Flips the epic plan bead's `is_ready_to_work` flag to `True` when it was not already ready.
4. Builds a Kahn-wave schedule from the epic's open phase children, respecting dependencies.
5. Pre-claims each phase bead — sets `status=in_progress` and `assignee=<phase_bead_id>` (i.e. `<epic_id>.<N>`).
6. Hands a single `---`-separated multi-prompt to the agent launcher. Each per-phase agent is spawned with name `<epic_id>.<N>` and references the `work_phase_bead` (https://sase.sh/xprompt/#available-tags) xprompt; a final land agent named `<epic_id>` references the `land_epic` (https://sase.sh/xprompt/#available-tags) xprompt. Phase dependencies become `%w` waits on blocker phase-agent names, and the land agent waits on every launched phase agent. Because `%w` requires a successful `done.json` outcome, a failed or killed phase keeps dependent phases and the land agent parked until the phase name is retried successfully. Phase beads with a stored `model` emit `%model:<value>`; phase beads without one emit `%model:worker`, which resolves through the worker-lane order: active worker override, matching `llm_provider.worker_models` entry, then the primary model lane. A stored model on the epic plan bead still applies only to the land agent. Each segment uses the force-reuse `%name:!<agent_name>` form so re-running `sase bead work` after a killed or failed run wipes the stale name owners before the relaunch — the command is safe to retry.

For legend-tier plans, the command:

1. Validates that `<id>` resolves to a plan bead with `tier=legend`, a positive `epic_count`, and a linked legend plan path.
2. On a confirmed launch, force-reuses the generated epic-planning and land agent names (like `<legend_id>.1.0` and `<legend_id>`) by wiping any prior owner of those names before relaunch, aborting before mutating bead state if that cleanup cannot complete. `--dry-run` performs no cleanup; it only warns which live agents a real launch would force-reuse.
3. Flips the legend plan bead's `is_ready_to_work` flag to `True` when launching.
4. Hands a single `---`-separated multi-prompt to the agent launcher. Each epic-planning segment is named `%name:!<legend_id>.<N>.0` and includes `%epic`; epic-planning segments do **not** carry `%approve`, because their generated plans go through the normal plan-approval flow. Epic `N > 1` also waits on `%w:<legend_id>.<N-1>`, so epic planning proceeds in order. After the epic-planning segments, a final land-legend segment named `<legend_id>` references the `land_legend` (https://sase.sh/xprompt/#available-tags) xprompt, waits on the last epic-

planning agent, and carries `%approve` so it can self-approve. When the legend bead has a stored `model`, that segment also emits `%model:<value>`. As with the epic-tier rendering, every segment uses the force-reuse `%name:!<agent_name>` form so the command is safe to retry after a killed or failed run.

Legend work does not create phase beads directly. The spawned epic-planning agents create epic plans, and the existing `bd/new_epic` automation handles the linked epic and phase beads after those plans are approved.

Flag	Description
`-n, --dry-run`	Print the wave plan and rendered multi-prompt without mutating state or launching
`-P, --no-push`	Commit launched bead state locally but skip the post-commit `git push`
`-y, --yes`	Skip the launch confirmation prompt

The work xprompts are resolved by `XPromptTag` (tag-based lookup), so a project-local or user-defined xprompt with the matching tag overrides the built-in. For epic-tier work, every phase and land segment carries a `%approve` directive so spawned agents can self-approve their own plans without a human-in-the-loop checkpoint between waves. For legend-tier work, only the trailing land-legend segment carries `%approve`; each epic-planning segment uses `%epic` and pauses for the normal plan-approval flow before its child phases run.

When the epic plan bead is attached to ChangeSpec metadata (`--changespec` / `--bug-id`), `sase bead work` preserves the current project's VCS context in the generated prompt. The first phase segment targets the project reference and adds a `#pr` reference for the ChangeSpec, while later phase and land segments target the ChangeSpec ref directly. For non-ChangeSpec epics launched from a known SASE workspace, each segment is still prefixed with the detected VCS workflow and project name (for example `#git:sase` or `#gh:sase-org/sase`). If the current directory is not associated with a SASE project, the prompts are left unprefixed and run in the caller's normal launch context.

If launching the multi-prompt fails partway through, the launcher SIGTERMs any already-spawned children before rolling back the pre-claims and the `is_ready_to_work` flag when this run set it (best-effort), so the epic can be retried without leaving zombie agents behind.

After the agents launch successfully, `sase bead work` commits the resulting bead-state mutation when the beads directory belongs to a git repository and canonical/projection files changed. This commit records the bead launch state only; it does not commit any code produced later by the spawned agents. Epic launches use the subject `chore: mark bead work launched for <id>`; legend launches use `chore: mark legend work launched for <id>`. If the git commit fails, the command reports that agents were already launched and exits non-zero so the operator can commit or repair the bead state explicitly. Dry runs and stores outside git do not create a commit.

When that commit succeeds and `bead.push_after_commit` is `true` (the default), `sase bead work` follows it with `git push` so the launched-work record reaches the remote without a manual follow-up step. The push inherits the caller's stdin/stdout/stderr, so interactive credential prompts still work. If the repository has no remote configured, the push is skipped silently — the local commit stands on its own. If `git push` fails (for example because the remote rejected the update), the failure is reported as a warning only: the bead-launch commit is preserved on the local branch and the warning text includes the manual `git push` invocation to retry.

Set `bead.push_after_commit: false` in `~/.config/sase/sase.yml` to disable the auto-push — useful for local-only checkouts, or when you would rather batch the bead-launch commit with later commits before pushing. Set it to `async` to keep auto-pushing but move the push off the critical path: `sase bead work` launches a detached

background `git push` and returns immediately, printing the log file where the background push records its result. The background push is non-interactive (its stdin is closed), so it cannot prompt for credentials; failures are written to that log instead of warning inline. Pass `--no-push` (`-P`) to `sase bead work` to skip the push for a single invocation regardless of the configured mode.

9.6 Rust Backend

The bead data model, event reducer, JSONL/config codecs, compatibility-cache refresh, mutation transactions, ID allocation, deterministic work-plan DAG, and common CLI output planning are implemented in `sase-core` and exposed through `sase_core_rs`. Python keeps the host logic that belongs in the application layer: locating the active bead store, relativizing plan paths, resolving VCS context and xprompts for `sase bead work`, prompting the user, launching agents, rolling back failed launches, and incrementing telemetry counters.

Common `sase bead` commands dispatch through an early CLI fast path before the full top-level parser is built. Help text and host-coupled commands still fall through to the normal Python parser/handlers where needed.

Use these checks when changing bead internals:

```
sase core health -j
pytest tests/test_bead tests/test_core_facade/test_bead_read.py tests/test_core_facade/test_bead_mutation.py
just rust-check
just bead-perf-smoke
```

9.7 Current Checkout Source Of Truth

In version-controlled mode, every `sase bead` read and mutation command uses the current checkout's `sdd/beads/events/**` event store and `sdd/beads/config.json`, with `issues.jsonl` used only as a fallback when events are absent. Running the command in `myproject/` reads that checkout's bead state; running it in `myproject_2/` reads `myproject_2/sdd/beads/`. The CLI does not merge sibling workspace stores, and duplicate IDs in another checkout do not override the active checkout's records.

ID allocation also uses only the active store's `config.json` and canonical event state. If a sibling checkout has not pulled or merged the latest bead state, it may allocate IDs based on its local state; sync bead changes through the normal VCS workflow when several agents are coordinating on the same project.

Cross-project helper surfaces, such as mobile/editor bead pickers, may inspect one canonical store per known project, but they still do not merge numbered sibling workspaces or legacy bead stores for the same project.

9.8 ACE TUI Integration

9.8.1 Plan File Linking

When creating a plan bead with `--type plan(PATH)`, the file path is stored in the `design` field. The ACE TUI can navigate from a bead to its linked SDD file.

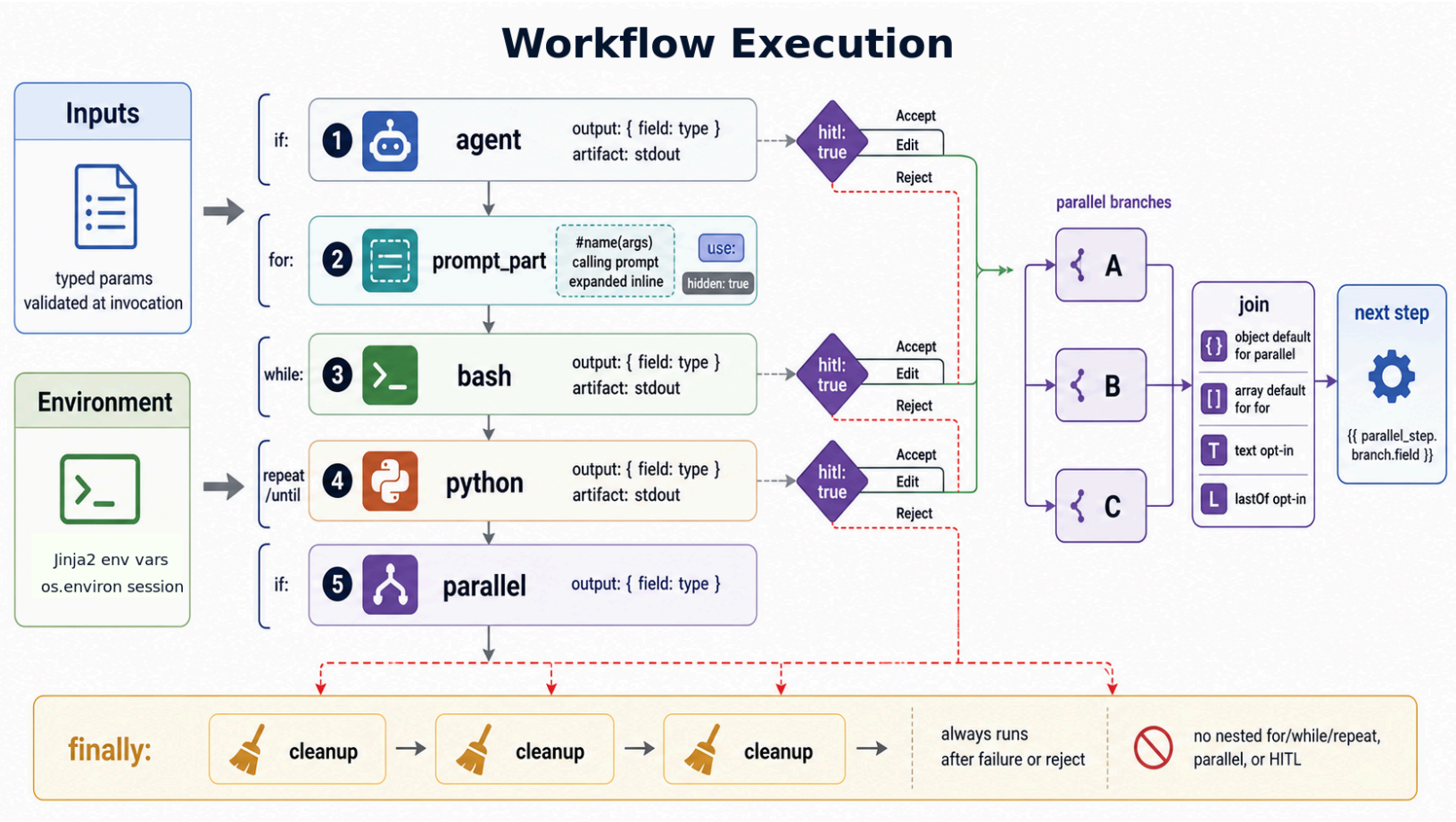
For SDD-generated epics, `PATH` should be the shared plan reference emitted by the plan approval flow: `sdd/epics/YYYYMM/*.md` in effective version-controlled mode, or `.sase/sdd/epics/YYYYMM/*.md` in local SDD mode. These relative references stay portable across checkouts while each checkout reads its own bead store.

9.8.2 Plan Approval Flow

The plan approval popup in ACE includes normal approval, **E** (Epic), and **L** (Legend) actions. Normal approval saves to `sdd/tales/`, Epic saves to `sdd/epics/` and launches the epic follow-up that creates an `epic`-tier plan bead plus phase beads, and Legend saves to `sdd/legends/` and launches a legend follow-up that records a `legend`-tier plan bead with `epic_count` before starting the legend work chain.

10 Workflow Specification

This document describes the YAML workflow format for sase xprompt workflows. Workflows enable multi-step agent pipelines with control flow, parallel execution, and human-in-the-loop approval.



10.1 Table of Contents

- Top-Level Structure
- Input Parameters
- Environment
- Step Types
- Step Imports
- Output Specification
- Artifact Passing
- Control Flow
- Parallel Execution
- Join Modes
- Template Syntax
- Agent Output-Variable Namespaces
- Cross-Step Field Type Checking
- Human-in-the-Loop
- Cleanup Steps

- Completion Markers
- Examples

10.2 Top-Level Structure

A workflow YAML file must define `steps` and can include optional metadata, inputs, environment variables, and local xprompt helpers:

```yaml
name: my_workflow # Workflow identifier (optional, defaults to filename)
tags: vcs, rollover # Semantic role tags (optional)
hidden: false # Hide the workflow run row from ACE's default Agents-tab view (optional)
input: # Input parameter definitions (optional)
  ...
environment: # Environment variables (optional)
  MY_VAR: "value"
xprompts: # Workflow-local xprompt definitions (optional)
  helper:
    content: "Use {{ setup.path }}"
steps: # Ordered list of steps (required)
  - ...
```

10.2.1 Fields

Field	Required	Description
`name`	No	Workflow identifier used in xprompt references. Defaults to filename without extension.
`tags`	No	Semantic role tags. See XPrompt Tags (https://sase.sh/xprompt/#tags) for available tags.
`hidden`	No	Hide the workflow run row from ACE's default Agents-tab view.
`wraps_all`	No	Legacy wrapper flag; new workflows should prefer `tags: vcs`.
`input`	No	Input parameter definitions. See Input Parameters.
`environment`	No	Environment variables set before any steps run. See Environment.
`xprompts`	No	Workflow-local xprompt definitions available to this workflow's steps.
`steps`	Yes	Ordered list of workflow steps to execute.

10.3 Input Parameters

Workflows can declare typed input parameters that users provide when invoking the workflow.

10.3.1 Longform Syntax

```yaml
input:
  - name: diff_path
    type: path
  - name: max_retries
    type: int
    default: 3
  - name: description
    type: text
    default: ""
```

10.3.2 Shortform Syntax

```
input:
  diff_path: path
  max_retries: { type: int, default: 3 }
  description: { type: text, default: "" }
```

Or even more concise for simple types:

```
input: { diff_path: path, split_desc: { type: line, default: "multiple CLs" } }
```

10.3.3 Supported Types

Type	Description
`word`	Single word, no whitespace
`line`	Single line, no newlines
`text`	Multi-line text (any content)
`path`	File path (no whitespace)
`int`	Integer value
`bool`	Boolean value (`true` / `false` , `yes` / `no` , `1` / `0`)
`float`	Floating point value

10.3.4 Default Values

In shortform input definitions, parameters without a `default` are required. Parameters with `default: null` or `default: ""` are optional. In current workflow YAML loading, longform entries that omit `default` are treated like `default: null` ; prefer shortform for required workflow inputs.

Default values preserve their native Python types from YAML parsing — `3` stays an `int` , `true` stays a `bool` , and `"text"` stays a `str` . This means downstream steps receive properly typed defaults without needing explicit conversion.

10.4 Environment

Workflows can declare environment variables that are set once before any steps run and persist for the entire agent session. Values support Jinja2 templates rendered against the workflow's input arguments.

```
name: deploy_workflow
input: { target: word }
environment:
  DEPLOY_TARGET: "{{ target }}"
  VERBOSE: "1"
steps:
  - name: deploy
    bash: echo "Deploying to $DEPLOY_TARGET"
```

Environment variables are injected into `os.environ` at workflow start, making them available to all bash, python, and agent steps without explicit passing.

10.5 Step Types

Each step must have exactly one of these execution types:

10.5.1 Agent Steps

Execute an LLM prompt, optionally referencing inline-capable xprompts:

```
- name: generate_plan
  agent: |
    #plan_generator(
      context="{{ previous_step.output }}",
      requirements="{{ requirements }}"
    )
  output: { plan: text, files: text }
```

The `agent` field contains a prompt template that can:

- Reference xprompts using `#xprompt_name(args)` syntax
- Use Jinja2 template variables: `{{ variable }}`
- Include multi-line content

Standalone workflows, which have no `prompt_part` step, cannot be embedded inside an `agent` prompt. Launch them with `#!workflow_name(args)` at the top level or inside an anonymous wrapper prompt such as `sase run '#gh:sase #!sase/pylimit_split %approve'`.

> **Note:** The keyword `prompt` is still accepted for backward compatibility, but `agent` is the canonical name.

10.5.2 Prompt Part Steps

Inject text into the containing agent prompt without triggering a separate LLM call. When a workflow is referenced via `#name(args)`, the `prompt_part` content is expanded inline into the calling prompt. Steps before and after the `prompt_part` still execute as pre/post-processing.

```
- name: inject
  prompt_part: |
    ---
    IMPORTANT: You should make the necessary file changes, but should NOT create a git commit.
    ---
```

This is the step type that simple `.md` xprompts are internally converted to — a single `prompt_part` step. Workflows with a `prompt_part` step can mix it with other step types (bash, python) for pre/post-processing around an inline prompt fragment:

```
steps:
  - name: setup
    bash: git status --porcelain
    output: { status: text }
  - name: inject
    prompt_part: |
      Current git status:
      {{ setup.status }}
  - name: cleanup
    bash: echo "done"
```

> **Note:** A workflow may have at most one `prompt_part` step. It is mutually exclusive with `agent`, `bash`, `python`, and `parallel` within a single step.

10.5.3 Bash Steps

Execute a shell command:

```
- name: get_status
  bash: git status --porcelain
  output: { files: text }
```

```
- name: complex_script
  bash: |
    counter_file="/tmp/counter"
    if [ ! -f "$counter_file" ]; then echo "0" > "$counter_file"; fi
    count=$(cat "$counter_file")
    echo "count=$count"
  output: { count: int }
```

10.5.4 Python Steps

Execute Python code:

```
- name: process_data
  python: |
    import json
    data = json.loads('{{ input_json }}')
    result = [item.upper() for item in data]
    print("result=" + json.dumps(result))
  output: { result: text }
```

Python steps run in a subprocess with access to installed packages.

10.5.5 Hidden Steps

Any step can be marked `hidden: true` to omit it from the normal ACE Agents-tab workflow expansion. Hidden steps execute normally and still write their outputs, but they are shown only when the workflow row is fully expanded. This is useful for internal bookkeeping steps (e.g., report steps that emit metadata outputs) that would clutter the agent list:

```
- name: report
  hidden: true
  python: |
    import json
    print(json.dumps({"meta_commit_message": "..."}))
  output: { meta_commit_message: text }
```

A workflow can also set top-level `hidden: true` to omit the workflow run row from ACE's default Agents-tab view. The workflow still executes and still writes artifacts. `appears_as_agent` is not a YAML field to set directly; it is computed from the workflow shape. When the only non-hidden step is an `agent` step, the recorded workflow state gets `appears_as_agent: true`, so ACE displays the run as an agent row rather than a generic workflow row. Anonymous `tmp_*` workflows with that computed state are included in the normal Agents-tab visible inbox unless the workflow row is explicitly hidden.

10.5.6 Parallel Steps

Execute multiple nested steps concurrently:

```
- name: fetch_all
  parallel:
    - name: fetch_users
      bash: curl -s https://api.example.com/users
      output: { users: text }
    - name: fetch_posts
      bash: curl -s https://api.example.com/posts
      output: { posts: text }
```

See Parallel Execution for details.

10.6 Step Imports

The `use` field allows reusing step definitions from shared `.yml` files in `steps/` directories. This enables extracting common step patterns into reusable libraries.

10.6.1 Syntax

```
- name: my_step
  use: shared/check_changes
  output: { status: text } # Local fields override base definition
```

The `use` value is a slash-separated path relative to a `steps/` directory, without the file extension.

10.6.2 Search Paths

Step definitions are resolved from the following `steps/` directories (in priority order):

1. `.xprompts/steps/` (CWD, hidden)
2. `xprompts/steps/` (CWD, non-hidden)
3. `~/.xprompts/steps/` (home, hidden)
4. `~/xprompts/steps/` (home, non-hidden)

5. `<sase-package>/xprompts/steps/` (built-in)

Both `.yml` and `.yaml` extensions are checked.

10.6.3 Override Behavior

Local step fields override any fields from the base definition. The `use` field itself is stripped from the merged result:

```
# In steps/shared/lint.yml:
bash: ruff check .
output: { errors: text }

# In your workflow:
- name: lint_project
  use: shared/lint
  output: { errors: text, warnings: text }  # Overrides base output
```

10.6.4 Security

Paths containing `..` or starting with `/` are rejected.

10.7 Output Specification

Steps can declare an output schema for structured output parsing.

10.7.1 Simple Object Format

```
output: { field_name: type, another_field: type }
```

Example:

```
- name: parse_config
  bash: echo "name=myapp\nversion=1.0"
  output: { name: word, version: word }
```

10.7.2 Array Format

For steps that produce a list of objects:

```
output: [{ name: word, description: text }]
```

Example:

```
- name: generate_items
  agent: Generate a list of items
  output: [{ name: word, description: text, priority: { type: int, default: 0 } }]
```

10.7.3 Nested Defaults

Fields can have defaults using nested dict syntax:

```
output:
  name: word
  parent: { type: word, default: "" }
  priority: { type: int, default: 0 }
```

10.7.4 Output Parsing

Step output is parsed in this order:

1. **JSON**: If output is valid JSON, parse and validate against schema
2. **Key=Value**: Parse lines like `key=value` into a dict
3. **Text fallback**: Store raw output as `_raw` or `_output`

For bash/python steps, output should be printed as `key=value` lines:

```
echo "success=true"
echo "count=42"
echo "message=Operation completed"
```

10.8 Artifact Passing

The `artifact` field captures a step's stdout to a file, making it available to downstream steps as a file path.

10.8.1 Syntax

```
- name: generate_report
  bash: |
    echo "Full report content here..."
  artifact: stdout
  output: { summary: text }
```

10.8.2 Behavior

When `artifact: stdout` is set:

1. The step's stdout is saved to `{artifacts_dir}/{step_name}.stdout`
2. The file path is injected as the `_artifact` field in the step's output context
3. Downstream steps can reference it via `{{ step_name._artifact }}`

10.8.3 Example

```
steps:
  - name: build
    bash: |
      make all 2>&1
      echo "status=success"
    artifact: stdout
    output: { status: word }

  - name: review
    agent: |
      Review the build output at {{ build._artifact }}
      Build status: {{ build.status }}
```

10.8.4 Restrictions

Only `bash` and `python` steps can use `artifact`. It is **not** allowed on `agent`, `prompt_part`, `parallel`, or nested (parallel substep) steps. The only valid value is `"stdout"`.

10.9 Control Flow

10.9.1 Conditional Execution (`if`)

Skip a step if a condition is false:

```
- name: optional_step
  bash: echo "This runs conditionally"
  if: "{{ run_optional }}"
```

```
- name: skip_on_failure
  bash: echo "Only runs if previous succeeded"
  if: "{{ previous_step.success }}"
```

10.9.2 For Loops (`for`)

Iterate over a list:

```
# Single variable
- name: process_items
  bash: echo "Processing {{ item }}"
  for: { item: "{{ items }}" }
  output: { result: text }

# Multiple parallel variables (must have equal length)
- name: process_pairs
  bash: echo "{{ name }} has id {{ id }}"
  for:
    name: "{{ names }}"
    id: "{{ ids }}"
  output: { name: word, id: int }
```

The `for` field maps variable names to Jinja2 expressions that evaluate to lists. All lists must have equal length.

10.9.3 For-Loop Error Handling (`on_error`)

For-loop iterations use different defaults by step type:

- `agent` steps default to `on_error: continue`, so one failed iteration is recorded and the remaining iterations still run.
- `bash` and `python` steps default to `on_error: stop`, so the first failed iteration aborts the loop.

You can set `on_error: stop` or `on_error: continue` explicitly on any `for:` step:

```
- name: review_files
  agent: Review {{ file }}
  for: { file: "{{ files }}" }
  on_error: continue
  output: { summary: text }
```

10.9.4 While Loops (`while`)

Execute step while a condition is true (checked after each iteration):

```
# Short form
- name: poll_status
  bash: |
    status=$(check_status)
    echo "pending=$status"
  while: "{{ poll_status.pending }}"
  output: { pending: bool }

# Long form with max iterations
- name: poll_with_limit
  bash: echo "active={{ check_active }}"
  while:
    condition: "{{ poll_with_limit.active }}"
    max: 10
  output: { active: bool }
```

The step runs at least once, then continues while the condition is true. Default max iterations: 100.

10.9.5 Repeat/Until Loops (`repeat`)

Execute step until a condition becomes true (do-while semantics):

```
- name: retry_operation
  bash: |
    result=$(attempt_operation)
    echo "success=$result"
  repeat:
    until: "{{ retry_operation.success }}"
    max: 5
  output: { success: bool }
```

The step runs at least once, then repeats until the `until` condition is true. Default max iterations: 100.

10.9.6 Combined Control Flow

`if` can be combined with `for`:

```
- name: conditional_loop
  bash: echo "Processing {{ item }}"
  if: "{{ should_process }}"
  for: { item: "{{ items }}" }
```

`for` can be combined with `parallel`:

```
- name: parallel_per_item
  for: { item: "[1, 2, 3]" }
  parallel:
    - name: task_a
      bash: echo "A processing {{ item }}"
    - name: task_b
      bash: echo "B processing {{ item }}"
```

10.10 Parallel Execution

The `parallel` field runs nested steps concurrently.

10.10.1 Basic Usage

```
- name: parallel_tasks
  parallel:
    - name: task_a
      bash: echo "result=done_a"
      output: { result: word }
    - name: task_b
      bash: echo "result=done_b"
      output: { result: word }
```

10.10.2 Accessing Results

Default join mode is `object`, nesting results under step names:

```
# After parallel_tasks completes:
{{ parallel_tasks.task_a.result }}  # "done_a"
{{ parallel_tasks.task_b.result }}  # "done_b"
```

10.10.3 Nested Step Restrictions

Steps within `parallel` cannot have:

- `for`, `repeat`, or `while` loops
- Nested `parallel` blocks
- `hitl: true`

10.10.4 Fail-Fast Behavior

Parallel steps are fail-fast: if any nested step fails, remaining nested steps are cancelled best-effort.

10.11 Join Modes

The `join` field controls how iteration/parallel results are combined.

Mode	Default For	Description
`array`	`for` loops	Collect results as a list
`object`	`parallel`	Merge results into a single object
`text`	-	Concatenate results as newline-separated text
`lastOf`	-	Keep only the last result

10.11.1 Examples

```
# Array join (default for for:)
- name: collect_results
  bash: echo "value={{ item }}"
  for: { item: "[1, 2, 3]" }
  output: { value: int }
# Result: [{"value": 1}, {"value": 2}, {"value": 3}]

# Text join
- name: concatenate
  bash: echo "line={{ item }}"
  for: { item: "['a', 'b', 'c']" }
  join: text
  output: { line: word }
# Result: "line=a\nline=b\nline=c"

# lastOf join
- name: keep_last
  bash: echo "final={{ item }}"
  for: { item: "['first', 'middle', 'last']" }
  join: lastOf
  output: { final: word }
# Result: {"final": "last"}

# Object join (default for parallel:)
- name: merge_parallel
  parallel:
    - name: a
      bash: echo "key_a=value_a"
    - name: b
      bash: echo "key_b=value_b"
# Result: {"a": {"key_a": "value_a"}, "b": {"key_b": "value_b"}}
```

10.12 Template Syntax

Workflows use Jinja2 for template rendering.

10.12.1 Variable Access

```
# Input parameters
{{ parameter_name }}

# Step outputs
{{ step_name.field }}
{{ step_name.nested.field }}

# Loop variables (within for: loops)
{{ item }}
{{ name }}

# Output variables from waited named agents, loaded when this run starts
{{ agents["build"].report_path }}
{{ agents["research.final"].summary }}
```

10.12.2 Filters

```
# Convert to JSON
{{ data | tojson }}

# Get length
{{ items | length }}
```

10.12.3 Boolean Logic

```
# Conditions
{{ value and other_value }}
{{ value or fallback }}
{{ not value }}

# Defined check
{{ variable is defined }}
```

10.12.4 Conditionals in Templates

```
agent: |
  {% if condition %}
  Include this text
  {% endif %}

  {{ "yes" if flag else "no" }}
```

10.13 Agent Output-Variable Namespaces

Workflow step outputs are the normal way to pass data between steps in one YAML workflow. Cross-agent output variables are a separate handoff for SASE agents. After a consumer's `%wait` dependencies complete, SASE reads small string values that the producers wrote with `sase var set KEY=VALUE` and adds them to the consumer's Jinja context.

Those values are loaded when the consumer starts; they are not live-updated after rendering begins. Have the producer call `sase var set` before it finishes. In workflows launched as agents, the `agents` dictionary is available to workflow template rendering, including `agent`, `bash`, `python`, `environment`, and `prompt_part` templates.

Every producer's variables live under a single `agents` dictionary keyed by the producer's stable name. Agent-name templates use the template base instead of the concrete allocated name. The key is the raw agent name with no identifier munging, so dotted, hyphenated, and digit-leading names work via bracket access:

Producer name or template	Referenced as
`%name:build-agent`	`{{ agents["build-agent"].report_path }}`
`%name:build-@`	`{{ agents["build"].report_path }}`
`%name:research.@.final`	`{{ agents["research.final"].report_path }}`
`%name:0n.cld`	`{{ agents["0n.cld"].report_path }}`

Identifier-safe keys also support attribute access such as `{{ agents.build.report_path }}`.

The `agents` dictionary is an agent-level handoff, not a replacement for workflow `output:` schemas or the ACE `WORKFLOW VARIABLES` section. Use `output:` for values produced and consumed inside one YAML workflow; use `sase var set` when a later named agent, segment, or agent-launched workflow needs a small value from a completed producer. `agents` is a reserved agent-run Jinja name; a workflow input named `agents` collides and fails clearly.

10.14 Cross-Step Field Type Checking

The workflow validator checks `{{ step_name.field }}` template references against each step's declared output schema at load time. This catches field name typos before the workflow runs.

10.14.1 What It Checks

- `{{ build.atrifact_path }}` produces an error if `build` only defines `artifact_path` in its output
- Works with parallel step nesting: `{{ parallel.nested.field }}`
- Recognizes the special `_artifact` field on steps with `artifact: stdout`
- Recognizes the special `approved` field on `bash` and `python` HITL steps
- Skips for-loop iteration variables (e.g., `item`)

10.14.2 Error Format

```
Step 'deploy': references 'build.atrifact_path' but 'build' output has no field 'atrifact_path'. Available:
['artifact_path', 'status']
```

10.15 Human-in-the-Loop

The `hitl: true` directive pauses execution for user approval.

10.15.1 Basic Usage

```
- name: generate_plan
  agent: Generate a migration plan
  output: { plan: text }
  hitl: true
```

10.15.2 Approval Flow

When a HITL step completes:

1. Step output is displayed to the user
2. User can:
3. **Accept**: Continue to next step
4. **Edit**: Modify the output before continuing
5. **Reject**: Abort the workflow

Some interfaces may show feedback or rerun controls, but the current workflow executor only handles accept, edit, and reject as workflow-control actions.

10.15.3 Accessing Approval Status

After an accepted `bash` or `python` HITL step, `step.approved` is set to `true` for downstream conditions:

```
- name: prompt_user
  bash: echo "message=Continue with operation?"
  output: { message: text }
  hitl: true

- name: execute_if_approved
  if: "{{ prompt_user.approved }}"
  bash: perform_operation
```

10.15.4 Restrictions

- HITL steps cannot be nested within `parallel` blocks
- HITL works with `agent`, `bash`, and `python` step types
- `agent` HITL steps do not automatically add an `approved` field to their output

10.16 Cleanup Steps

Steps marked with `finally: true` run even when prior steps fail or are HITL-rejected. This is useful for cleanup and teardown operations.

10.16.1 Syntax

```yaml
steps:
  - name: setup
    bash: mkdir -p /tmp/workdir
    output: { dir: text }

  - name: main_work
    agent: Do the main work in {{ setup.dir }}
    output: { result: text }

  - name: cleanup
    finally: true
    bash: rm -rf /tmp/workdir
```

10.16.2 Rules

- Put `finally: true` steps at the **end** of the workflow, after all non-finally steps. After a failure, later non-finally steps are skipped while `finally` steps still run.
- Cannot be used on `prompt_part` or nested (parallel substep) steps
- Can be combined with `if:` for conditional cleanup:

```yaml
- name: conditional_cleanup
  finally: true
  if: "{{ setup.dir }}"
  bash: rm -rf {{ setup.dir }}
```

10.17 Completion Markers

When a multi-step workflow finishes, a `done.json` marker is written to track completion status. For multi-agent workflows (workflows that spawn follow-up agents), `done.json` is written to both the current step's artifacts directory and the root artifacts directory. The root copy ensures that:

- `%wait` dependencies resolve correctly (the workflow name is recognized as successful)
- Agent name allocation preserves the workflow's reserved name
- `find_named_agent()` can discover the completed workflow from the root

Structure:

```json
{
  "cl_name": "branch-name",
  "project_file": "/path/to/project.spec",
  "timestamp": "260327_143000",
  "artifacts_timestamp": "260327_143000",
  "outcome": "completed",
  "workspace_num": 1,
  "name": "agent-name",
  "model": "opus",
  "llm_provider": "claude",
  "vcs_provider": "git"
}
```

The `outcome` field is either `"completed"` or `"failed"`. Failed markers additionally include `"error"` and `"traceback"` fields.

10.17.1 Workflow-Aware Wait Success

The `%wait` directive treats a workflow dependency as satisfied only when the newest matching workflow run completed successfully:

1. Find all agents belonging to the workflow (matching `workflow_name`)
2. Select the newest root agent (no `parent_timestamp`) when one exists
3. Require the root `done.json` outcome to be `"completed"`
4. Require every child agent for that root to have a `done.json` outcome of `"completed"`

If older artifacts only retained `workflow_name` on children and no root can be identified, the newest matching artifact is used as a compatibility fallback, but it still must have outcome `"completed"`. Failed, killed, crashed, still-running, malformed, or missing `done.json` artifacts do not satisfy `%wait`; the dependent agent remains waiting until a later successful run of the same workflow name appears.

If a workflow name is not recognized (no agents with that `workflow_name`), the system falls back to single-agent success checking with the same `"completed"` outcome requirement.

10.18 Examples

10.18.1 Complete Workflow Files

The following workflow files demonstrate these features:

- `eval_ifs_loops.yml` - Conditional execution, for loops, while/repeat loops
- `eval_parallel.yml` - Parallel execution with different join modes
- `split.yml` - Real workflow with HITL, agent steps, and xprompt references

10.18.2 Minimal Example

```yaml
name: simple_workflow
input: { name: word }
steps:
  - name: greet
    bash: echo "message=Hello, {{ name }}!"
    output: { message: line }
```

10.18.3 Multi-Step with Conditionals

```yaml
name: conditional_workflow
input:
  run_optional: { type: bool, default: true }
  items: { type: text, default: '["a", "b", "c"]' }

steps:
  - name: setup
    bash: echo "ready=true"
    output: { ready: bool }

  - name: process_items
    bash: echo "processed={{ item }}"
    if: "{{ setup.ready }}"
    for: { item: "{{ items }}" }
    output: { processed: word }

  - name: optional_step
    bash: echo "ran=true"
    if: "{{ run_optional }}"
    output: { ran: bool }

  - name: summary
    bash: echo "items_count={{ process_items | length }}"
    output: { items_count: int }
```

10.18.4 Parallel with Dependencies

```yaml
name: parallel_workflow
steps:
  - name: fetch_data
    parallel:
      - name: users
        bash: echo "count=100"
        output: { count: int }
      - name: orders
        bash: echo "count=500"
        output: { count: int }

  - name: combine
    bash: |
      total={{ fetch_data.users.count + fetch_data.orders.count }}
      echo "total=$total"
    output: { total: int }
```

10.18.5 Retry with HITL

```yaml
name: retry_workflow
steps:
  - name: attempt_operation
    bash: |
      # Simulated operation that may fail
      success=$((RANDOM % 2))
      echo "success=$success"
    repeat:
      until: "{{ attempt_operation.success }}"
      max: 3
    output: { success: bool }
    hitl: true

  - name: finalize
    if: "{{ attempt_operation.approved }}"
    bash: echo "finalized=true"
    output: { finalized: bool }
```

11 Workspace Provider Reference

The **workspace provider layer** is an abstraction that handles workspace-level operations that vary across VCS hosting environments. While the VCS provider (https://sase.sh/vcs/) handles low-level version control commands (commit, diff, checkout), the workspace provider handles higher-level concerns: workflow type detection, reference resolution (e.g., `#git:repo` or `#gh:org/repo`), change submission, mail preparation, and workspace directory management.

A **workspace reference** is a prompt prefix such as `#cd:/tmp/project`, `#git:sase`, `#gh:sase`, or `#hg:mychange`. It tells SASE which project and workspace should be used before the rest of the prompt or workflow runs.

11.1 Plugin Architecture

Workspace providers are implemented as pluggy (https://pluggy.readthedocs.io/) plugins, following the same pattern as VCS plugins. The core `sase` package bundles the **BareGitWorkspacePlugin** for local bare-remote git repositories. Additional backends must be installed in the same Python environment as `sase`:

Package	Plugin	Description
`sase` (core)	`CdWorkspacePlugin`	Local directory runs via `#cd:<path>` without VCS
`sase` (core)	`BareGitWorkspacePlugin`	Bare-git repos (local filesystem remote)
`sase-github`	`GitHubWorkspacePlugin`	GitHub-hosted repos (PR workflows via `gh` CLI)

Plugins register themselves via the `sase_workspace` entry point group. The plugin manager loads all registered plugins and dispatches operations through pluggy hooks. Most hooks use `firstresult=True` — the first plugin that returns a non-`None` result wins.

11.1.1 Hook Specification

All workspace operations are defined in `WorkspaceHookSpec` (`src/sase/workspace_provider/_hookspec.py`). Each method is prefixed with `ws_` to namespace them within the pluggy project.

11.2 Key Data Types

11.2.1 WorkflowMetadata

Each workspace plugin declares metadata about the workflow type it supports:

Field	Type	Description
`workflow_type`	string	Short name used in `#type:ref` prompts (e.g., `"cd"`, `"git"`, `"gh"`)
`ref_pattern`	string	Regex matching `#type:ref` or `#type(ref)` syntax
`display_name`	string	Human-readable name (e.g., `"Git (bare)"`, `"GitHub"`)
`pre_allocated_env_prefix`	string	Env-var prefix for pre-allocated workspace variables
`vcs_family`	string	VCS family (e.g., `"git"`, `"hg"`)
`vcs_provider_name`	string	Specific VCS provider name (e.g., `"bare_git"`, `"github"`)

Built-in metadata includes `SASE_CD` for `#cd` and `SASE_GIT` for `#git`. Plugin packages can add prefixes such as `SASE_GH` and `SASE_HG`.

11.2.2 ResolvedRef

Result of resolving a workspace reference:

Field	Type	Description
`project_file`	string	Path to the project spec file
`project_name`	string	Name of the project
`primary_workspace_dir`	string	Path to the primary workspace directory
`checkout_target`	string	Branch or revision to check out
`extra`	dict[str, str]	Additional plugin-specific data

For clone-based git workflows, `primary_workspace_dir` is the primary checkout path and `get_workspace_directory()` derives numbered sibling workspaces from it. Some provider plugins can leave `primary_workspace_dir` empty and resolve numbered workspaces through their own helper command.

11.3 Hook Reference

11.3.1 Metadata and Detection

Hook	Returns	Description
`ws_get_workflow_metadata`	`WorkflowMetadata | None`	Declare this plugin's workflow type metadata
`ws_detect_workflow_type`	`str | None`	Detect workflow type from a project file
`ws_get_change_label`	`str | None`	Get the change label (e.g., `"PR"`, `"CL"`)
`ws_get_workspace_name`	`str | None`	Get the workspace/project name for a CWD

`ws_get_workflow_metadata` is the only hook that collects results from **all** plugins (not `firstresult`). This allows the registry to build a complete map of all available workflow types.

11.3.2 Reference Resolution and Workflow Setup

Hook	Returns	Description
`ws_resolve_ref`	`ResolvedRef | None`	Resolve a `#type:ref` reference to workspace info
`ws_setup_workflow`	`dict[str, str] | None`	Set up environment variables for a workflow run
`ws_get_workspace_directory`	`str | None`	Get or create a workspace directory for a clone

11.3.3 Change Submission and Review

Hook	Returns	Description
`ws_submit`	`tuple[bool, str | None] | None`	Submit a ChangeSpec (merge, push, etc.)
`ws_prepare_mail`	`object | None`	Prepare a change for mailing/review
`ws_extract_change_identifier`	`tuple[str, str] | None`	Extract identifier from a CL/PR URL
`ws_supports_reviewer_comments`	`bool | None`	Check if a CL URL supports reviewer comments
`ws_generate_reviewer_comments_script`	`str | None`	Generate a script to fetch reviewer comments

`ws_generate_submitted_check_script`	`str | None`	Generate a script to check if a CL is submitted

11.3.4 Commit Formatting

Hook	Returns	Description
`ws_format_commit_description`	`bool | None`	Format a commit description file (add tags, prefix, etc.)

11.4 Registry Functions

The workspace provider package (`sase.workspace_provider`) exports convenience functions that call through the plugin manager. These are the primary API for consumers:

Function	Description
`detect_workflow_type()`	Detect workflow type for a project file
`get_change_label()`	Get the change label for a project
`resolve_ref()`	Resolve a workspace reference
`submit_changespec()`	Submit a ChangeSpec
`get_workspace_directory()`	Get the directory for a workspace number
`prepare_mail()`	Prepare a change for review
`format_commit_description()`	Format a commit description
`get_all_workflow_metadata()`	Get metadata from all registered plugins
`get_workflow_names()`	Get all registered workflow type names
`get_display_name()`	Get display name for a workflow type
`get_display_name_by_vcs()`	Get display name by VCS provider name
`get_display_name_by_vcs_family()`	Get display name by VCS family
`get_workspace_name()`	Get workspace/project name for a directory
`get_ref_patterns()`	Get all registered ref patterns
`get_pre_allocated_env_prefix()`	Get env-var prefix for a workflow type

11.5 Directory Workflow (`#cd`)

The core package also registers `#cd:<path>` as a workspace workflow. It resolves a local directory, makes that directory the agent/workflow CWD, and deliberately skips numbered workspace allocation, checkout, diff, submit, and release behavior. `#cd` is the right choice for one-off work in an existing directory when SASE should not prepare or release a VCS workspace.

Prompts without any workspace reference are normalized to `#git:home` ; use `#cd:~` when you want a direct home-directory run with no VCS workspace management.

Supported examples include `#cd:~` , `#cd:/tmp/project` , `#cd:../sibling` , and `#cd(.)` . The target must already exist and must be a directory. `~` and environment variables are expanded, and relative paths are resolved from the launching process's current directory. Prefer the parenthesized form for paths with spaces, for example

`#cd(/tmp/my project)`.

11.6 Bare-Git Reference Auto-Initialization

The bundled bare-git provider resolves `#git:<ref>` in four modes:

1. A registered project shorthand, using `~/.sase/projects/<name>/<name>.sase` when it contains `BARE_REPO_DIR` and `WORKSPACE_DIR`.
2. A ChangeSpec name found across registered projects.
3. A missing project shorthand with no slash, which initializes a new bare-git project using `~/.sase/repos/<name>.git` as the bare repository and `~/projects/git/<name>/` as the primary checkout.
4. A bare repository path, deriving the project name from the path basename and creating the matching ProjectSpec with that bare path and the default `~/projects/git/<name>/` primary checkout path.

The missing-project shorthand is intended for first use from an xprompt or prompt bar: `#git:new_tool #!workflow` creates the bare-git project on demand.

`#git:home` is special because it is the default for bare prompts. If the `home` ProjectSpec is missing or has not yet recorded `BARE_REPO_DIR`, SASE bootstraps a managed empty bare-git project at the default `home` paths. To point a project at an existing bare repository, use `#git:<bare-repo-path>`; the path basename becomes the SASE project name, so `#git:/path/to/home.git` registers the `home` project. Add `#cd:~` to a prompt for a one-off direct home-directory run without VCS.

Bare-git projects use version-controlled SDD under `sdd/`. SASE creates or refreshes generated SDD guide files during new project initialization, existing bare-repo registration, and first `#git` or `sase workspace open` materialization. When materialization owns the checkout setup, SASE commits and pushes only those generated guide paths with an `Initialize SDD` init commit.

11.7 Known-Project VCS Fallback

SASE also recognizes provider-prefixed VCS refs that target registered project names even when the corresponding workspace plugin is not available in the current process. Known projects are discovered from `~/.sase/projects/*/*.sase` (with legacy `~/.sase/projects/*/*.gp` accepted as a fallback) by reading each `WORKSPACE_DIR:` entry. For example, if the `sase` project is registered, `#gh:sase #!some/workflow` and the underscore shorthand `#gh_sase #!some/workflow` are treated as VCS workspace launches rather than ordinary xprompt references.

Known-project fallback is lifecycle-aware. Normal launch and xprompt/catalog discovery paths only include active projects; a registered project with `PROJECT_STATE: inactive` or `PROJECT_STATE: sibling` is hidden from launch pickers and broad known-project lookup. Legacy `archived` and `closed` values are read as inactive. If a prompt explicitly names an inactive known project, launch resolution fails with an activation hint instead of silently allocating work. Use `sase project list --state all` to inspect hidden projects and `sase project activate <project>` before launching normal work there. Configured linked repositories use hidden `PROJECT_STATE: sibling` records instead of normal launch discovery. To prepare one, pass its linked-repo name as the workspace CLI's project override: `sase workspace open -p <linked_repo> -r "<reason>" <workspace_num>`. In a SASE-

launched agent session, that command records the linked-repo name in the run artifacts; ACE uses that record for opened-workspace context, and the commit finalizer uses it to enforce only configured numbered linked-repo workspaces the agent explicitly opened.

Non-wait launches allocate the next available numbered workspace for the project and set the VCS update target to the provider default revision. When registered workspace metadata provides an env prefix, SASE passes the matching `<PREFIX>_PRE_ALLOCATED`, `<PREFIX>_WORKSPACE_NUM`, and `<PREFIX>_WORKSPACE_DIR` values into the child process. Launches that start with a wait directive keep workspace number `0` until the dependency is ready, then resolve a real workspace during normal runner setup. Directory runs such as `#cd` also use workspace number `0` because they do not reserve a numbered workspace. Before applying the current launch context, SASE removes inherited `SASE_*_PRE_ALLOCATED`, `SASE_*_WORKSPACE_NUM`, and `SASE_*_WORKSPACE_DIR` variables so nested or follow-up launches do not accidentally reuse a stale parent workspace.

11.8 Relationship to VCS Provider

The workspace provider and VCS provider are complementary plugin systems:

Concern	VCS Provider	Workspace Provider
Scope	Low-level VCS commands	High-level workspace operations
Examples	`git commit`, `git diff`, `hg amend`	Ref resolution, submit, mail prep, workspace dirs
Plugin granularity	One active plugin per detected VCS type	All plugins registered, firstresult dispatch
Entry point	`sase_vcs`	`sase_workspace`
Hook prefix	`vcs_`	`ws_`

A single plugin package (e.g., `sase-github`) typically provides both a VCS plugin and a workspace plugin.

11.9 Workspace Directory Layout

SASE resolves every workspace through a per-project store rather than by string-appending `_<num>` to the primary checkout path. The store assigns workspaces stable numeric identities and chooses a physical path according to the configured root policy.

11.9.1 Numeric Identity

Range	Meaning
`#0`	Primary checkout from ProjectSpec `WORKSPACE_DIR`. Also used as the placeholder for deferred launches.
`#1` – `#9`	Reserved. The allocator never hands these out, but legacy tests and call sites that pass them by hand still work via the compatibility wrapper.
`#10`+	Claim-allocated numbered workspaces. New agents allocate from this unified pool starting at `#10`.

Older releases allocated agent workspaces starting at `#1` and special-cased axe at `#100`. The current allocator uses one shared pool for every claim source; `claim_next_axe_workspace()`, `get_first_available_axe_workspace()`, launch executor pre-claims, and `axe` deferred claims all start at `#10` unless a caller passes explicit `min_workspace` / `max_workspace` bounds.

User-facing checkout suffixes are `<project>_<num>` regardless of root policy. The primary checkout retains its `WORKSPACE_DIR` path with no suffix.

11.9.2 Root Policy

The physical location of managed checkouts is controlled by `workspace.root` (see `docs/configuration.md` (https://sase.sh/configuration/#workspace)) and the `SASE_WORKSPACE_ROOT` environment override:

Value	Layout
`xdg-state`	Default. Platform state root plus namespace: `$XDG_STATE_HOME/sase/workspaces/<project_key>/<project>_<num>/` on Linux, `~/Library/Application Support/sase/workspaces/...` on macOS, `%LOCALAPPDATA%\sase\workspaces\...` on Windows.
`adjacent`	Legacy `<primary>_<num>/` siblings of the primary checkout. Explicit opt-in; byte-for-byte compatible with previous releases.
absolute path	Treat the configured path as the managed-root base and create `<project_key>/<project>_<num>/` checkouts under it.

`SASE_WORKSPACE_ROOT` overrides `workspace.root` for the process and is interpreted as an explicit managed root directory, with the project namespace appended underneath it. Use an absolute path for predictable behavior. It is the recommended override for ephemeral test runs and CI sandboxes.

The `project_key` namespace under managed roots is derived from a single Git remote slug when available, otherwise from the primary-path basename plus a short hash so two projects with the same basename do not collide. An explicit `workspace.project_key` in config wins over the heuristic.

Each managed checkout writes a `.sase/checkout.json` marker recording the project name, project key, workspace number, primary workspace path, and the registry path. CWD-based project inference (used by `sase bead`, the file panel, and similar callers) reads the nearest marker first and only falls back to sibling-pattern scanning for adjacent legacy layouts.

11.9.3 Registry

For non-adjacent roots SASE maintains a per-project registry alongside the checkouts. The registry tracks every workspace the store owns — including primary `#0` — and records `checkout_dir`, `materialization`, `role`, `pinned`, `created_at`, and `last_used_at`. Registry writes are atomic. `sase workspace repair` is the canonical way to reconcile the registry against the filesystem after a manual delete or a partially completed migration. Adjacent checkouts keep their legacy sibling behavior and normally do not write a persistent registry; `cleanup` and `repair` treat a missing registry as "nothing managed here" rather than an error.

11.9.4 Adjacent Compatibility And Migration

The default `workspace.root` is `xdg-state` for unconfigured installations and projects. Existing adjacent `<primary>_<num>/` directories are not moved during ordinary resolution; SASE only creates new managed checkouts under the configured root. To carry old adjacent checkouts into the managed root, run:

```
sase workspace migrate --to xdg-state [--symlink-transition] [--dry-run]
sase workspace migrate --finalize [--dry-run]
```

The first form moves every existing `<primary>_<num>` checkout under the managed root and records it in the registry. With `--symlink-transition`, the original `<primary>_<num>` path becomes a symlink to the canonical managed checkout so legacy tooling that walks `..` for siblings keeps working. Migration refuses to overwrite a real directory at the managed destination; pre-existing managed content is reported and skipped. `--dry-run` reports the planned actions without touching the filesystem or registry.

Once workflows have adapted to the managed paths, `sase workspace migrate --finalize` removes the leftover transition symlinks without touching the canonical checkouts.

11.9.5 Backup, Container, And Network-Storage Caveats

- **Backups.** With `adjacent`, every numbered checkout sits inside the user's normal source tree and gets captured by Borg/Restic/Syncthing/BTRFS snapshots. Managed roots move execution state out of the primary backup surface; if you want crashed-agent working trees included, add `~/.local/state/sase/workspaces/` (or the platform equivalent) to your backup profile explicitly.
- **BTRFS / ZFS snapshots.** A snapshot of `~/projects` no longer freezes every workspace atomically once the canonical checkouts live elsewhere. Snapshot the state root alongside the source tree if atomicity matters.
- **NFS / network home directories.** If `$HOME` is on NFS and your source tree is on local SSD, switching to `xdg-state` can move workspaces to slow storage. Set `workspace.root` to an absolute path on the fast volume, or keep `adjacent`.
- **Containers / devcontainers / Toolbx / Distrobox.** Tools that bind-mount `~/projects` into a container will not see managed checkouts under `~/.local/state`. Either add a second mount for the managed root or keep `workspace.root: adjacent` for the containerized project.
- **Recursive search performance.** `rg`, `fd`, IDE workspace-wide search and similar tools fan out N times across adjacent siblings. Managed roots avoid this by default.

11.9.6 Post-Default Migration Guidance

Users and environments that still rely on sibling checkouts should set `workspace.root: adjacent` explicitly, either in the project-local `sase.yml` or globally in `~/.config/sase/sase.yml`. The managed-root default is intentionally non-migrating: it prevents silent moves, but a project with old adjacent clones and no explicit config will create new non-primary checkouts under the state root after the default change.

Before switching shared CI images, containers, or network-mounted homes to the default, confirm the state root is mounted, backed up, and on storage fast enough for agent work. Use an absolute `workspace.root` when the platform state directory is not the right operational location.

11.10 `sase workspace` CLI

The `sase workspace` command surface inspects numbered workspace paths and maintains the per-project registry used by non-adjacent roots. All subcommands accept `-p/--project NAME` to override the project; without it, the project is inferred from the current directory via the nearest managed-checkout marker, the workspace provider

hook, and finally a scan of `~/.sase/projects/`. With no subcommand, `sase workspace` defaults to `sase workspace list` with default options. Use `sase workspace list -p NAME` or `sase workspace list --json` when passing list flags.

Command	Description
`sase workspace list [-j/--json]`	List the registry view for the project, root policy, project key, root path, and primary `#0`.
`sase workspace path NUM`	Print the configured checkout path for `NUM` without cloning or preparing it.
`sase workspace open NUM -r/--reason REASON [-p/--project PROJECT] [-c/--clean]`	Materialize the checkout if needed, stash or otherwise back up local changes through the VCS provider, clean it, sync it to the provider default parent revision, then print the path. Requires a non-empty `-r/--reason`.
`sase workspace cleanup -s/--stale`	Remove unclaimed managed checkouts older than `workspace.cleanup_ttl_days`. `-n/--dry-run` previews.
`sase workspace repair [-n]`	Drop registry entries whose checkout is gone; re-materialize missing registered checkouts that still have live RUNNING claims.
`sase workspace migrate --to xdg-state [-s/--symlink-transition] [-n]`	Move existing `<primary>_<num>` adjacent checkouts under the managed `xdg-state` root and register them. Exits non-zero on skipped refusals.
`sase workspace migrate --finalize`	Remove `<primary>_<num>` transition symlinks once workflows have adapted to the managed paths.

`path` always resolves `#0` to the primary checkout. For other numbers, it prints the configured path without cloning. Use this command when you only need to inspect the path.

`open` is intentionally more forceful. It materializes the requested checkout, backs up uncommitted local changes through the normal workspace-preparation path, cleans it, checks out the active VCS provider's default parent revision, runs the provider's workspace sync hook when available, and then prints the path. For built-in bare-git projects, it first makes sure the primary checkout has generated SDD guide files. `list` and `path` remain read-only and do not run SDD initialization. `--clean` is accepted as a compatibility flag for this default behavior. Use a claim-range number such as `10` when handing a numbered checkout to an external shell, editor, or debugging tool. `#0` is the primary checkout, and `#1` through `#9` are reserved compatibility numbers rather than good choices for new manual checkouts. For linked-repo work, `-p/--project` is still the project selector: pass the configured linked repo name there, then the workspace number as the positional argument.

`cleanup` and `repair` skip workspace `#0` and any workspace number with an active claim. `cleanup --include-shares` opts workflow-share checkouts into the same cleanup pass.

`migrate --to xdg-state` is opt-in. Existing adjacent checkouts are left in place until the command is invoked. With `--symlink-transition` it leaves a `<primary>_<num>` symlink at the original adjacent path so legacy tooling that still walks `..` for siblings keeps working; the canonical checkout lives under the managed root. Migration refuses to overwrite a real directory at the managed destination — pre-existing managed content is reported and skipped instead of clobbered. `cleanup --stale` removes both the canonical managed checkout and its transition symlink. Once workflows are adapted, `migrate --finalize` removes the leftover transition symlinks without touching the canonical checkouts.

11.11 Disabling Plugins

The workspace provider registry loads provider entry points directly. It does not currently consult the resource-plugin disable switches described in docs/configuration.md (https://sase.sh/configuration/#plugin-system).

12 Mentors

12.1 Overview

Mentors are automated AI code review agents for ChangeSpecs. A ChangeSpec is SASE's local record for one proposed code change; mentors watch its commits, decide whether configured review profiles match, and then run focused review agents in the background.

The Axe daemon drives mentor checks. When a mentor finds issues, it writes structured JSON comments with severities (`error`, `warning`, or `suggestion`). You review those comments in the ACE TUI and can launch an apply agent for the accepted comments.

The lifecycle is:

1. Match mentor profiles against ChangeSpec commits.
2. Register matching profiles in the ChangeSpec `MENTORS` field.
3. Wait for non-skipped hooks on the latest regular commit to become ready.
4. Start mentor agents when runner slots are available.
5. Save structured JSON output and file snapshots.
6. Review, accept, and apply comments from ACE.

12.2 Configuration

Mentors are configured through `mentor_profiles` in `sase.yml`. SASE loads merged config, so user-level profiles from `~/.config/sase/sase.yml` and project-local profiles from `./sase.yml` can both contribute profiles. See `docs/configuration.md` (https://sase.sh/configuration/#mentor_profiles) for the field reference.

A profile defines when it should run. Each profile contains one or more mentors, and each mentor has a role plus focus areas that constrain what it should review.

```yaml
mentor_profiles:
  - profile_name: python_review
    file_globs:
      - "*.py"
    mentors:
      - mentor_name: style_checker
        role: "Python style expert"
        focus_areas:
          - focus_name: style
            description: "PEP 8 compliance and code style"
          - focus_name: naming
            description: "Variable and function naming conventions"

  - profile_name: security_review
    diff_regexes:
      - "password|secret|token|api_key"
    mentors:
      - mentor_name: security
        role: "Security reviewer"
        focus_areas:
          - focus_name: credentials
            description: "Hardcoded credentials and secret exposure"
          - focus_name: injection
            description: "SQL injection, XSS, and command injection"

  - profile_name: first_look
    first_commit: true
    mentors:
      - mentor_name: architecture
        role: "Software architect"
        focus_areas:
          - focus_name: design
            description: "Overall design and architectural patterns"
```

12.2.1 Profile Matching Criteria

Each profile must have at least one matching criterion. If a profile has more than one criterion, SASE uses OR logic: any criterion can match.

Criterion	Matches against
`file_globs`	Changed file paths extracted from the diff
`diff_regexes`	Diff content using Python regular expressions
`amend_note_regexes`	Commit or amend notes using Python regular expressions
`first_commit`	The first regular commit entry of a ChangeSpec

Profile matching ignores proposal entries such as `(2a)` and uses regular commit entries such as `(1)` and `(2)`. Matching profiles are added to the latest regular commit's `MENTORS` entry before the mentors actually start.

12.2.2 Project Scoping

Mentor profiles from a project-local `sase.yml` are automatically scoped to that project. This means a profile defined in `/home/user/myproject/sase.yml` will only match ChangeSpecs belonging to `myproject`, preventing local profiles from firing on unrelated projects.

You can also explicitly set the `projects` field on any profile to restrict it to specific projects:

```yaml
mentor_profiles:
  - profile_name: backend_review
    projects: ["myproject", "other_project"]
    file_globs:
      - "*.py"
    mentors:
      - mentor_name: reviewer
        role: "Python expert"
        focus_areas:
          - focus_name: correctness
            description: "Logic and correctness"
```

Profiles defined in user-level config (`~/.config/sase/sase.yml`) remain unscoped by default and apply to all projects unless `projects` is explicitly set.

12.2.3 Mentor Definition

Each mentor in a profile requires these fields:

- **`mentor_name`**: Unique identifier within the profile.
- **`role`**: Persona for the review (e.g., "Python expert", "Security reviewer"). This shapes the LLM's review perspective.
- **`focus_areas`**: List of review dimensions, each with a `focus_name` and `description`. These define what the mentor looks for and structure its output.

12.2.4 Debugging Profile Matching

Use `sase config mentor-match <changespec_name>` to trace which profiles match a given ChangeSpec and why. The command loads the current config, inspects the ChangeSpec commits, and reports per-criterion results.

```
sase config mentor-match my_feature_cl
```

12.3 Execution Lifecycle

12.3.1 1. Profile Registration

When a ChangeSpec is in a reviewable status (`Ready` or `Mailed`), the scheduler checks all mentor profiles against the regular commits since the last mentor entry. Matching profiles are registered in the `MENTORS` field with `[0/N]` counts before execution begins.

12.3.2 2. Hook Readiness

Mentors wait for all non-skipped hooks on the latest regular commit to become ready. A hook is ready when it passed, or when a failed hook has been handled by a fix-hook, proposal, summarizer, or metahook. Hooks prefixed with `!` are skipped and do not block mentor eligibility.

12.3.3 3. Execution

Each mentor runs as a background process:

1. The mentor is marked STARTING (prevents race conditions).
2. A background subprocess is spawned with its own session.
3. The `#mentor` xprompt workflow renders the prompt with the mentor role and focus areas.
4. The project VCS workflow (`#git` , `#gh` , or `#hg`) provides the workspace context.
5. The LLM response is parsed as structured JSON and saved to `~/.sase/mentors/` .

12.3.4 4. Completion

Mentors use these statuses:

Status	Meaning
STARTING	Registered and about to spawn a runner
RUNNING	Background mentor runner is active
PASSED	Completed successfully with no review comments
COMMENTED	Completed successfully with one or more comments
FAILED	Execution error or invalid JSON response

KILLED	Manually killed or auto-killed because a newer commit exists
DEAD	Runner process disappeared or its PID was reused

12.3.5 5. Stale Mentor Cleanup

When a newer commit is added to a ChangeSpec, mentors running against older commits are automatically killed. This prevents stale reviews from continuing after new code is committed.

12.4 Output Format

Mentor output is saved as JSON to `~/.sase/mentors/`. Each output file records the mentor metadata and a `comments` array. Each comment includes:

Field	Type	Description
`focus_name`	string	Focus area this comment addresses
`file_path`	string	Path to the file being commented on
`line_number`	integer	Line number in the file
`description`	string	Actionable description of the issue
`severity`	string	One of `error`, `warning`, or `suggestion`

12.5 MENTORS Field in ChangeSpec Files

Mentor status is tracked in the MENTORS section of ChangeSpec files:

```
MENTORS:
  (1) python_review[2/2] security_review[1/1]
      | [260320_150530] python_review:style_checker - COMMENTED - (0h2m15s)
      | [260320_150530] python_review:naming - PASSED - (0h1m45s)
      | [260320_151500] security_review:security - COMMENTED - (0h3m10s)
  (2) python_review[0/2]
      | python_review:style_checker - STARTING
```

The entry id matches a regular `COMMITS` entry. The header line shows profile names with `[started/total]` counts. Status lines show timestamp, `profile:mentor`, status, and duration for completed mentors.

12.6 ACE TUI Integration

12.6.1 Review Mentors (`,m`)

Press `,m` on the PRs tab to open the Mentor Review modal for the current ChangeSpec. The modal shows all mentor comments and lets you accept or reject individual suggestions.

Key	Action
`j` / `k`	Navigate between mentors
`n` / `p`	Navigate between comments within a mentor

`N` / `P`	Navigate between accepted comments only
`Ctrl+D` / `Ctrl+U`	Scroll comment details down / up
`Space`	Toggle acceptance of the current comment
`a`	Apply accepted comments and propose (amend with propose)
`A`	Apply accepted comments and commit
`r`	Run a mentor profile (opens profile picker)
`y`	Copy the current comment to the clipboard
`K`	Kill the selected running mentor
`Esc` / `q`	Close modal

Apply Modes

There are two ways to apply accepted mentor comments:

- **`a`** — Launches the `make_mentor_changes` workflow with the `propose` xprompt appended, so the agent proposes its changes as an amend.
- **`A`** — Launches the `make_mentor_changes` workflow with the `commit` xprompt appended, so the agent commits directly.

Both modes save the accepted comments as an artifact under `~/.sase/mentors/` before launching the apply agent.

Running Mentor Profiles

Press `r` to open a profile picker showing all configured mentor profiles. Selecting a profile starts (or restarts) all mentors in that profile for the current ChangeSpec. This is useful for re-running a specific review after making changes.

Running and completed mentor agents are visible in the Agents tab under the `@review` tag. The same tag is used for CRS, fix-hook, and summarize-hook review agents, so review automation can be inspected, killed, dismissed, or resumed from one side panel.

12.6.2 Code Snippets in Review

Each mentor comment in the review modal is displayed alongside a syntax-highlighted code snippet centered on the referenced line number. The snippet uses Rich's Monokai theme with line numbers, word wrapping, and the target line highlighted. Syntax highlighting is determined by the file extension (supports Python, JavaScript, TypeScript, Go, Rust, Markdown, YAML, JSON, shell, SQL, and other common languages).

Code snippets are loaded from file snapshots when available. Older mentor outputs without snapshots fall back to the VCS provider when a revision is available.

12.6.3 File Snapshots

When a mentor completes, the contents of all files referenced in its comments are snapshotted and saved alongside the mentor output at `~/.sase/mentors/<cl>-<profile>-<mentor>-<ts>-files.json`. This ensures that the Mentor Review modal can display code snippets instantly without fetching files from VCS, even if the working tree has changed since the mentor ran.

12.6.4 Read Tracking

The Mentor Review modal tracks which comments you have viewed. As you navigate between comments, each one is automatically marked as read. The modal header displays the current global comment position (`N / total`) and an acceptance count (`✓ N accepted`). The side panel shows a mini progress bar for each mentor (`■` = read, `□` = unread) along with an accepted/total count. The side panel also shows a status indicator per mentor: `▸` (selected), `●` (running), `✗` (failed/killed), or `✓` (all comments accepted). Read state persists across modal opens and is stored per ChangeSpec and commit entry.

Unread comment counts also appear inline in the PRs tab list (see ACE docs (https://sase.sh/ace/#mentor-comment-stats-in-pr-list)).

12.6.5 Kill Mentors (`,M`)

Press `,M` on the PRs tab to kill all running mentors for the current ChangeSpec.

12.6.6 Fold Mentors (`z m`)

Press `z m` to toggle the MENTORS section visibility in the ChangeSpec detail view.

12.7 Trigger Conditions

Mentors run on ChangeSpecs that meet **all** of the following:

- Status is **Ready** or **Mailed** (not WIP, Draft, Submitted, Reverted, or Archived).
- The ChangeSpec has at least one commit.
- At least one mentor profile's matching criteria are satisfied.
- All non-skipped hooks for the latest regular commit are ready.

13 Commit Workflows

Sase provides three unified workflows for landing code changes: **commit**, **propose**, and **pull request**. All three share the same CLI command (`sase commit`), the same `CommitWorkflow` orchestrator, and the same VCS provider abstraction, but differ in what they produce and how they track the result.

13.1 Overview

Workflow	XPrompt	Method	What it produces	Tracking
Commit	`#commit`	`create_commit`	Git commit on current branch	COMMITS entry
Propose	`#propose`	`create_proposal`	Saved diff file	COMMITS entry
PR	`#pr`	`create_pull_request`	New branch + PR	ChangeSpec

```
#commit / #propose / #pr
        |
        v
Agent edits files
        |
        v
Provider-neutral commit finalizer
        |
        v
Commit skill wrapper (/sase_git_commit, /sase_hg_commit, ...)
        |
        v
sase commit -> CommitWorkflow -> VCS provider -> tracked output
```

13.2 How It Works

13.2.1 1. Agent makes code changes

The agent receives an xprompt (`#commit`, `#propose`, or `#pr`) which sets the `SASE_COMMIT_METHOD` environment variable and injects an instruction telling the agent **not** to create commits directly.

13.2.2 2. Commit finalizer checks for uncommitted work

When a provider invocation succeeds inside a SASE-launched agent session, the provider-neutral **commit finalizer** runs in the shared LLM invocation layer before normal success postprocessing. In practice this means the process has `SASE_AGENT_TIMESTAMP` set. The finalizer checks the main workspace for uncommitted changes through the active VCS provider. It enforces configured numbered linked repositories only after the agent opens that linked workspace with `sase workspace open -p <linked_repo> -r "<reason>" <workspace_num>`, which records the linked-repo name in the run's artifacts. Static linked repos are still checked only as advisory work, as described below. It does not scan arbitrary same-remote numbered workspaces just because their paths appear in run artifacts. If everything is clean, the agent response is postprocessed normally.

There are two special cases before the normal enforced-work follow-up path:

- If the only enforced dirty file is a tracked markdown file under `sdd/tales/`, `sdd/epics/`, `sdd/legends/`, or `sdd/myths/`, and the only file diff is one leading-front-matter line changing from `status: wip` to `status: done`, SASE creates a direct closeout commit with the message `chore: Mark SDD plan done` and a `TYPE=sdd` runtime tag.
- Linked repos configured with `workspace.strategy: none` are static singletons. Dirty static linked repos are included in the follow-up prompt as advisory work: the agent is told to commit them only if it made those changes in this session, and leaving them dirty does not fail the finalizer. A run with only advisory static linked-repo changes can still get a follow-up prompt, but it finalizes successfully after that pass even if the advisory repo stays dirty.

If enforced changes or advisory static linked-repo changes remain, the finalizer starts bounded follow-up passes with the same provider. Each pass sends one follow-up prompt that lists dirty files and instructs the agent to use a commit skill such as `/sase_git_commit` or `/sase_hg_commit`. For the main workspace, the skill name is selected from the detected VCS provider; provider-specific generated skills can be scoped to the runtimes that support that provider. For configured linked repos, the current finalizer checks `git status` only in the resolved linked-repo `workspace_dir` assigned to the same workspace number after that linked-repo name appears in `opened_linked_workspaces.json`, and emits Git commit-skill instructions that first `cd` into that linked workspace. Non-static dirty linked repos are enforced after they are opened; static linked repos are advisory as described above.

Generated skills normally run an observable wrapper such as `sase_git_commit`, which records skill invocation evidence and then delegates to `sase commit`. A typical Git skill invocation omits `--type` because the xprompt already set `SASE_COMMIT_METHOD`:

```
sase_git_commit -M commit_message.md -f src/example.py
```

The low-level equivalent is `sase commit -M commit_message.md -f src/example.py -t <method>`. The method defaults to `$SASE_COMMIT_METHOD` if the `-t` flag is omitted. If both the environment and `-t/--type` are set, they must resolve to the same method unless `SASE_COMMIT_METHOD_ALLOW_OVERRIDE=1` is set.

If `SASE_BEAD_ID` is set, the finalizer first asks the agent to decide whether the uncommitted changes were made in the current session. For changes the agent did make, it instructs the agent to close and verify the bead before invoking the commit skill. This keeps bead lifecycle state ahead of the commit/proposal/PR dispatch while avoiding accidental closure for unrelated dirty work.

The finalizer uses the shared instruction helpers in `sase.commit_instructions`, so the bead and method wording stays consistent between main-workspace and linked-repository commit guidance. `commit.finalizer.max_passes` controls how many follow-up invocations may run before SASE fails the invocation with a clear error and, when an artifacts directory is available, a `commit_finalizer_result.json` artifact.

13.2.3 CLI Arguments

Short	Long	Description
`-m`	`--message`	Commit message string (mutually exclusive with `-M`)
`-M`	`--message-file`	Path to file containing the commit message / PR description (mutually exclusive with `-m`)
`-f`	`--file`	File to stage (repeatable; omit to stage all)
`-n`	`--name`	Branch/CL name (required for `create_pull_request`)
`-B`	`--bug-id`	Bug ID to associate with the commit (overrides `$SASE_BUG_ID`)
`-c`	`--checkout-target`	Branch point for PR (default: `HEAD~1`)
`-p`	`--parent`	Parent ChangeSpec **name** (overrides auto-detection from current branch). Must be an existing ChangeSpec in the active project file or archive — if it does not resolve, the PARENT field is omitted with a warning. Never pass a VCS ref (e.g., `origin/main`, `p4head`).

`-r`	`--resume`	Resume a previously-checkpointed commit after manual conflict resolution. When set, `-m` / `-M` / `-f` and other commit args are ignored (the payload is loaded from the checkpoint). See Resume after Conflict below.
`-s`	`--status`	ChangeSpec status for PRs (`wip`, `draft`, `ready`). Overrides `$SASE_PR_STATUS`; default is `draft`.
`-t`	`--type`	Commit method — accepts full names or short aliases (see table below)

Type Aliases

The `-t/--type` flag accepts both full method names and short aliases:

Alias	Full Method
`commit`	`create_commit`
`propose`	`create_proposal`
`pr`	`create_pull_request`

The COMMITS entry note is always derived from the first line of the commit message — there is no separate `--note` flag.

13.2.4 3. CommitWorkflow orchestrates

`CommitWorkflow` (`src/sase/workflows/commit/workflow.py`) is the central dispatcher. It runs through these stages:

```
Bead association   (inject SASE_BEAD_ID into message when set)
    |
Bead lifecycle     (close bead, sync beads)                              [skip for proposals]
    |
Plan handling      (append PLAN= to message, mark plan done)             [skip for proposals]
    |
Precommit command  (e.g. `just fix`)
    |
PR name suffixing  (compute _<N> suffix for unique branch names)         [PR only]
    |
Detect parent CL   (auto-set PARENT from current branch's ChangeSpec)    [PR only]
    |
PR metadata        (append PR tags and project prefix)                   [PR only]
    |
Runtime tags       (append/update AGENT= and MACHINE= provenance)        [commit/PR only]
    |
PR body            (build body with final tags and agent footer)         [PR only]
    |
Diff capture       (save the pre-dispatch diff for tracking)
    |
Checkpoint         (save resolved payload and tracking state for resume)
    |
VCS dispatch       (call provider.create_commit / create_proposal / create_pull_request)
    |
ChangeSpec         (create ChangeSpec entry in project file)             [PR only]
    |
Result marker      (write commit_result.json for xprompt post-steps)
    |
COMMITS entry      (append entry to project file)                        [commit/propose only]
```

13.2.5 4. XPrompt reads the result

The xprompt post-steps read `commit_result.json` from `$SASE_ARTIFACTS_DIR` and emit metadata outputs (`meta_new_commit`, `meta_commit_message`, `meta_changespec`, etc.) for downstream consumption.

13.3 CLI Inputs and Internal Payload

The `sase commit` CLI builds an internal `CommitWorkflow` payload from flags. It does **not** accept a positional JSON payload.

Typical commit or proposal:

```
sase commit -M commit_message.md -f src/auth.py -f src/login.py -t commit
```

Typical PR:

```
sase commit -M pr_description.md -n feature_branch -B 12345 -s ready -t pr
```

The internal payload has this shape:

```
{
  "message": "Commit message (required for commit/propose)",
  "name": "Branch or PR name (required for PR)",
  "files": ["optional", "list", "of", "specific", "files"]
}
```

The CLI maps `-m` / `-M` to `message`, repeated `-f` flags to `files`, `-n` to `name`, `-B` to `bug_id`, `-c` to `checkout_target`, `-p` to `parent`, and `-s` to `status`. Omitted `-f` means "stage all changes" and is represented as an empty `files` list.

Bead association is not a user-supplied CLI flag. For new commit attempts, `sase commit` reads `SASE_BEAD_ID`; when it is set, the CLI adds that bead to the workflow payload, and `CommitWorkflow` enforces that the bead ID appears in the first line of the dispatched commit or PR message. Conflict resumes reuse the bead value captured in the original checkpoint.

Runtime provenance tags are also not user-supplied CLI flags. For `create_commit` and `create_pull_request`, `CommitWorkflow` appends or updates trailing `AGENT=<sase agent name>` and `MACHINE=<host name>` lines in the commit message. `AGENT` comes from `SASE_AGENT_NAME`, falling back to `SASE_ARTIFACTS_DIR/agent_meta.json` when available; it is omitted for manual non-agent commits with no agent name. `MACHINE` comes from the local hostname. Runtime-owned `AGENT` and `MACHINE` values replace inherited or configured PR tags with the same keys. `create_proposal` does not get runtime commit tags because it saves a diff instead of creating a VCS commit.

Internal fields added by `CommitWorkflow`:

Field	Set by	Purpose
`_cl_name`	Environment	Fallback CL name for proposals
`_plan_path`	`_handle_sase_plan`	Plan file path for VCS staging

`_pr_body`	`_build_pr_body`	Enriched PR description with agent info
`_skip_bead_amend`	Internal	Skip post-commit bead amend
`bead_id`	Environment	Bead ID resolved from `SASE_BEAD_ID`

13.4 Result Format

After a successful dispatch, `commit_result.json` contains:

```
{
  "method": "create_commit",
  "result": "<commit_hash | diff_path | null>",
  "message": "The commit message",
  "name": "Branch/CL name",
  "bead_id": "Bead ID if SASE_BEAD_ID was set",
  "changespec_name": "ChangeSpec name (PR only)",
  "entry_id": "COMMITS entry ID (commit/propose only)",
  "diff_path": "Saved pre-dispatch diff path, when available"
}
```

For compatibility with older xprompt post-steps, the marker also includes alias keys: `commit_result`, `commit_changespec_name`, `commit_entry_id`, and `commit_diff_path`.

13.5 Workflow Details

13.5.1 Commit (`#commit`)

Creates an actual git commit on the current branch and pushes it.

Git operations:

1. Stage files (`git add -A` or specific files)
2. Stage `sdd/beads/` directory and plan file
3. Validate staged changes exist
4. Merge with `origin/<default_branch>` to keep the branch current
5. `git commit -m <message>`
6. Post-commit bead amend (append bead note)
7. Push to remote with retry on failure

Returns: `(True, commit_hash)`

Tracking: Appends a COMMITS entry to the project file with the commit note, diff path, chat path, and plan path (when `SASE_PLAN` is set). Multi-line commit messages are supported: the first paragraph becomes the note, and subsequent paragraphs (separated by a blank line) become an indented body below the note. Empty body lines are stored as a dot (`.`) placeholder to preserve structure. See change_spec.md (https://sase.sh/change_spec/#commits) for the full entry format including drawers.

13.5.2 Propose (`#propose`)

Saves the current diff without committing and cleans the workspace. This is useful for parking work-in-progress changes that aren't ready to land.

Git operations:

1. Save diff to `~/.sase/diffs/<cl_name>-<timestamp>.diff`
2. Clean workspace (`git reset --hard HEAD` + `git clean -fd`)

Returns: `(True, diff_path)`

Tracking: Appends a proposal COMMITS entry to the project file. Bead lifecycle and plan handling are skipped because proposals don't represent landed changes. Runtime `AGENT` and `MACHINE` commit tags are also skipped because no VCS commit is created.

13.5.3 Pull Request (`#pr`)

Creates a new branch, commits changes, pushes, and creates a PR (via the GitHub plugin or equivalent).

Input parameters:

```
input:
  - name: name # Branch/PR name (required)
    type: word
  - name: bug_id # Bug ID (optional, default: 0)
    type: int
  - name: status # Initial ChangeSpec status: draft, wip, or ready (optional, default: draft)
    type: word
```

Git operations:

1. `git checkout -b <name>` (create new branch)
2. Stage files and bead/plan paths
3. `git commit -m <message>`
4. `git push -u origin <name>`
5. (GitHub plugin creates the actual PR via `gh`)

Returns: `(True, pr_url)` after GitHub plugin processing

Parent detection: If the current branch corresponds to an existing ChangeSpec, that ChangeSpec is automatically set as the PARENT of the new PR ChangeSpec. This creates a chain of related changes without manual bookkeeping.

BUG propagation: When `SASE_BUG_ID` is set in the environment and non-zero, the value is propagated to two places: the BUG field of the created ChangeSpec (as `http://b/<bug_id>`), and a `BUG=<bug_id>` line prepended to the PR tag block (taking precedence over any static `BUG` key in `vcs_provider.pr_tags` config).

Project prefix: When `vcs_provider.use_project_pr_prefix` is `true`, a `[<project>]` prefix is prepended to the PR title (GitHub) or CL description (Mercurial). This prefix is only applied to the external representation — it does not appear in the ChangeSpec DESCRIPTION or git commit message, and is automatically stripped when reading descriptions back.

PR tag inheritance: When creating a child PR (one whose PARENT is an existing ChangeSpec), PR tags from the parent PR's body are automatically inherited. The merge order is: parent PR tags (lowest priority) -> config `pr_tags` -> `BUG` tag (highest priority), followed by runtime-owned `AGENT` and `MACHINE` tags. Inherited or configured `AGENT` and `MACHINE` values are ignored so child PRs do not retain stale parent runtime provenance.

PR tags: Any key-value pairs configured in `vcs_provider.pr_tags` are appended as `TAG=VALUE` lines to the commit message before building the PR body. This supports provider-specific metadata (e.g., Google CL tags) without manual entry. `AGENT` and `MACHINE` are reserved for runtime provenance and are owned by the commit workflow rather than static config. See configuration.md (https://sase.sh/configuration/#vcs_provider) for the config format.

PR tag stripping: When PR tags are present in the commit description (trailing lines matching `^[A-Z][A-Z0-9_]*=`), they are automatically stripped before writing the DESCRIPTION field of the created ChangeSpec. This prevents provider-specific metadata (e.g., `AUTOSUBMIT_BEHAVIOR=SYNC_SUBMIT`, `MARKDOWN=true`) from polluting the human-readable description. The same stripping is applied when syncing descriptions after a reword operation.

Tracking: Creates a ChangeSpec in the project file (not a COMMITS entry). The PR name is automatically suffixed with `_<N>` if a ChangeSpec with the same base name already exists.

13.6 VCS Provider Abstraction

The three dispatch methods are defined in `VCSHookSpec` and implemented by each VCS plugin:

Plugin	`create_commit`	`create_proposal`	`create_pull_request`
`BareGitPlugin`	Commit + push	Save diff + clean	Branch + commit + push
`GitHubPlugin`	Inherits from git	Inherits from git	+ creates PR via `gh` CLI
`HgPlugin`	`hg commit` + mail	`sase_hg_clean`	Not supported natively

All methods return `tuple[bool, str | None]` (success flag and optional result string).

Plugins that support resume also implement `vcs_finalize_commit(payload, cwd)`, which re-runs the idempotent portion of a commit (bead amend, push with retry) after a previously-checkpointed workflow has had its merge conflicts resolved by hand. See Resume after Conflict below for how this fits into the overall flow. Providers that cannot safely replay finalization (e.g., Mercurial today) can leave it unimplemented — `CommitWorkflow.resume` catches the `NotImplementedError` and only replays the tracking steps.

13.7 Run Result

`CommitWorkflow.run()` and `CommitWorkflow.resume()` return a `RunResult` with three states:

State	Exit code	Meaning
`OK`	`0`	Commit succeeded end-to-end (or resume replayed tracking).

`FAILED`	`1`	Unrecoverable failure — bail out, no checkpoint is left behind on fatal errors.
`CONFLICT`	`2`	VCS dispatch hit a merge conflict; a checkpoint is left on disk for resume.

The `sase commit` CLI propagates these states to its process exit code, so wrapper skills (`/sase_git_commit`) can branch on `$?` to distinguish a real failure from a conflict that the user needs to resolve.

13.8 Resume after Conflict

`CommitWorkflow` persists its progress to a checkpoint file so that a dispatch interrupted by a merge conflict can be finished by hand without re-running the whole flow:

```
SASE_ARTIFACTS_DIR/commit_state.json             # preferred, when running under a workflow
~/.sase/commit_state/<session>.json              # fallback when no artifacts dir is set
```

Normal flow:

1. `CommitWorkflow.run()` snapshots its resolved state (payload, CL name, project file, diff path, reserved name, parent CL) to the checkpoint **before** calling the VCS dispatch method.
2. If dispatch succeeds, the checkpoint is updated with the dispatch result, tracking steps run, and the file is deleted on success.
3. If dispatch fails because of a merge conflict (`RunResult.CONFLICT`), the checkpoint is retained and the CLI prints:

> `create_commit` hit a merge conflict: ... Resolve the conflict, then run `sase commit --resume` to finish.

Resume flow (`sase commit --resume`):

1. Load the checkpoint from disk (if missing, the command errors out).
2. Re-check the working tree for conflict markers — if they're still present, refuse to continue with `CONFLICT`.
3. Verify the commit at `HEAD` matches the subject line from the checkpointed message. If it doesn't, abort with `FAILED`; the user is expected to re-run `sase commit` from scratch rather than resume into a foreign commit.
4. Call the provider's `vcs_finalize_commit` hook to replay idempotent post-commit work (bead amend, push with retry).
5. Re-run the tracking steps (COMMITS entry append, ChangeSpec creation) using the snapshotted payload.
6. Delete the checkpoint on success.

Resume is VCS-agnostic: the same `--resume` flag works for commits, proposals, and PRs. Skills emit the on-conflict instructions automatically, so agents know to hand control back to the user rather than retry blindly.

13.9 Environment Variables

Variable	Purpose
`SASE_COMMIT_METHOD`	Dispatch method (set by xprompt `environment:` section)
`SASE_COMMIT_METHOD_ALLOW_OVERRIDE`	Allow `-t/--type` to override a conflicting `SASE_COMMIT_METHOD`

`SASE_ARTIFACTS_DIR`	Directory for `commit_result.json` and other artifacts
`SASE_AGENT_NAME`	Agent name used for `AGENT=` runtime commit provenance
`SASE_BEAD_ID`	Bead ID to automatically associate with the commit
`SASE_PLAN`	Plan file path for staging and status update
`SASE_AGENT_PROJECT_FILE`	Project file for COMMITS/ChangeSpec tracking
`SASE_AGENT_CL_NAME`	CL name used for proposal diff naming
`SASE_PR_NAME`	PR name (set by `#pr` xprompt input)
`SASE_PR_STATUS`	Initial PR ChangeSpec status (`draft` , `wip` , `ready`)
`SASE_BUG_ID`	Bug ID for PR metadata
`SASE_VCS_PROVIDER`	Override VCS provider detection (see vcs.md (https://sase.sh/vcs/))
`SASE_LINKED_REPOS_JSON`	JSON metadata for configured linked repos passed to agents
`SASE_LINKED_REPO_<ENV_NAME>_DIR`	Workspace-matched path for one configured linked repo
`SASE_DISABLE_COMMIT_STOP_HOOK`	Skip the commit finalizer when set

13.10 Commit Finalizer

For SASE-launched agent sessions, the normal path is the provider-neutral finalizer in `src/sase/llm_provider/commit_finalizer.py` . It runs after a successful provider invocation and before success postprocessing. The finalizer is deliberately outside any one runtime's native hook system, so Claude, Codex, Antigravity (`agy`), Qwen, OpenCode, and provider plugins share the same behavior.

Flow:

1. Skip when `commit.finalizer.enabled` is false, `SASE_DISABLE_COMMIT_STOP_HOOK=1` is set, or the process is outside a SASE agent session (`SASE_AGENT_TIMESTAMP` is unset).
2. Resolve the project directory from provider/workspace environment variables.
3. Check the main workspace through the VCS provider's diff helpers.
4. Check configured linked repos from `SASE_LINKED_REPOS_JSON` , or from project config when available, with `git status --porcelain` : numbered linked repos are limited to names in `opened_linked_workspaces.json` , while static `workspace.strategy: none` linked repos are advisory.
5. Auto-commit an exact tracked SDD markdown `status: wip` to `status: done` closeout when that is the only enforced change and the file is under `sdd/tales/` , `sdd/epics/` , `sdd/legends/` , or `sdd/myths/` .
6. If dirty enforced repos or advisory static linked repos exist, run follow-up provider invocations up to `commit.finalizer.max_passes` . When an artifacts directory is available, also write `commit_finalizer_pass_<N>_prompt.md` and `commit_finalizer_pass_<N>_response.md` .
7. Re-check every dirty target. If all enforced repos are clean, write `commit_finalizer_result.json` with status `finalized` when artifacts are enabled, and append the follow-up response to the agent's final response.
8. If enforced changes remain after `commit.finalizer.max_passes` , write status `failed` when artifacts are enabled and fail the invocation instead of silently accepting dirty work.

Configured linked repos are resolved to workspace-matched directories before agent launch. For example, an agent in `sase_10` sees a `../sase-core` linked repo as `sase-core_10` when that checkout is available or can be materialized. Repos configured with `workspace.strategy: none` are exposed to the agent and reported as advisory dirty targets when they are Git repos, but they are not enforced because their singleton ownership is ambiguous. The current linked-repo dirty-check path is Git-specific and opened-workspace gated: non-Git linked-repo paths can still be exposed to the agent through environment variables and metadata, and unopened numbered linked repos are ignored, but the finalizer does not enforce them as dirty targets.

When the only enforced dirty state is the exact SDD status closeout described above, the finalizer creates the commit itself instead of running a follow-up provider invocation. The result artifact records `reason: "auto_committed_done_plan_status"`.

The obsolete provider-native commit hook scripts are no longer shipped. Active SASE-launched runs rely on the provider-neutral finalizer instead of runtime-specific commit hook configuration.

13.11 Diff Storage

Diffs saved by proposals (and other operations) are stored in:

```
~/.sase/diffs/<name>-<timestamp>.diff     # Active diffs
~/.sase/reverted/<name>.diff              # Reverted CLs
~/.sase/archived/<name>.diff              # Archived CLs
```

Diffs can be re-applied to a workspace with `apply_diff_to_workspace()` from `sase.workflows.commit_utils.workspace`.

13.12 Design Principles

- **Fail-fast:** If `commit_result.json` is missing when the xprompt post-steps run, the workflow fails explicitly rather than silently retrying. The finalizer and commit skills are the sanctioned path to commit creation.
- **Single responsibility:** `CommitWorkflow` owns all orchestration (precommit, beads, plans, VCS dispatch, tracking). XPrompt steps only read and report results.
- **Proper proposal semantics:** Proposals save diffs and clean the workspace without creating commits. Bead lifecycle and plan handling are skipped because proposals don't represent landed changes.
- **VCS agnostic:** The same `CommitWorkflow` and xprompt definitions work across Git, GitHub, and Mercurial backends. Only the VCS plugin implementation differs.

14 Notifications

14.1 Overview

Sase includes a notification system that surfaces important events from background processes (axe, workflows, mentors) to the user through the ACE TUI. Notifications are stored as JSONL and persisted to `~/.sase/notifications/notifications.jsonl`.

14.2 Viewing Notifications

Press `i` on any tab in ACE to open the notifications modal. Notifications display relative timestamps (e.g., "2m ago", "1h ago") and can be marked as read or dismissed.

14.2.1 Modal Keybindings

Key	Action
`j` / `k`	Navigate between notifications
`Enter`	Select notification (jump to CL, approve plan, etc)
`x`	Dismiss notification (or bulk-dismiss every marked row when marks are present)
`m`	Toggle the per-row mark on the highlighted notification
`M`	Toggle mute on the highlighted notification
`s`	Snooze the highlighted notification (opens duration picker)
`e`	Open attached file in `$EDITOR`
`V`	Open the current image attachment in the image viewer
`Ctrl+N` / `Ctrl+P`	Cycle through attached files
`Ctrl+D` / `Ctrl+U`	Scroll file content down / up
`[` / `]`	Switch notification tag tabs
`R`	Mark all notifications as read
`Esc` / `q`	Close modal

Plan and question notifications require confirmation (`y` / `n`) before dismissal to prevent accidental loss of pending approvals. The same `y` / `n` confirmation is used for bulk dismissal when at least one marked plan or question notification is included in the batch.

14.2.2 Sectioned Layout

The modal renders notifications in four fixed-order sections, each with a colored header row and per-section count:

Section	Color	Contents
PRIORITY	Red	Plan approvals, user questions, mentor reviews, non-error axe notifications, and CRS workflow results
ERRORS	Orange	Axe error digests and agent error reports (sender `axe` or `user-agent` paired with the `ViewErrorReport` action)
INBOX	Gold	Everything else
MUTED	Cyan	Notifications the user has muted (or that are still snoozed). Mute dominates priority — a muted plan appears under MUTED, not PRIORITY.

Empty sections are not rendered. Section header rows are non-selectable; `j` / `k` skip over them automatically.

When unread notifications include tags, a compact tab strip appears above the list. `All` is always present, `done` is pinned immediately after `All` when successful agent-completion rows exist, and any other tags are sorted alphabetically. A tagged notification appears in `All` and in each matching tag tab; untagged notifications appear only in `All`. The fixed section order and newest-first sorting are recomputed within the active tab. Switching tabs with `[` / `]` or a mouse click clears modal-local marks so a hidden row is never bulk-dismissed by accident.

Within each section, rows are ordered newest-first by their `timestamp` field. The fixed section order (PRIORITY → ERRORS → INBOX → MUTED) is never reshuffled — only the rows inside each section are sorted. Rows with equal timestamps keep their original arrival order, and rows whose timestamp can't be parsed fall to the bottom of their section rather than breaking the modal. The sort runs on every modal rebuild, so live actions like mark-read, dismiss, mute, and snooze update the visible order immediately.

14.2.3 Marks and Bulk Dismiss

Press `m` on a notification to toggle a per-row mark. Marks are scoped to the open modal — closing the modal clears them. While at least one row is marked, `x` switches from "dismiss the highlighted row" to "dismiss every marked row"; plan and question rows in the batch use the same `y` / `n` confirmation prompt as a single dismissal.

14.2.4 Mute and Snooze

Press `M` on a notification to toggle its muted state. Muted notifications are dimmed in the list, prefixed with `~`, and moved to the **MUTED** section. They are still delivered to the JSONL store and remain visible in the modal — only the bell indicator and toast pipeline ignore them.

Press `s` to snooze a notification for `15m`, `1h`, `4h`, or until tomorrow morning. Snoozed notifications are implicitly muted (so they fall into the MUTED section) and display a `⏰ <remaining>` badge counting down to the snooze expiry. Toggling mute off cancels any pending snooze. The snooze expiry is persisted, so the notification re-emerges from MUTED on its own once the timer runs out.

14.2.5 Top-Bar Indicator

The notification indicator in the TUI top bar takes its color from the highest-priority unread bucket present:

- **Orange** — at least one unread PRIORITY or ERRORS notification (plan approval, user question, mentor review, axe error digest, agent error report, …)
- **Gold** — only regular INBOX notifications are unread
- **Cyan** — only MUTED (or snoozed) notifications are unread
- **Dim zero** — no unread notifications at all

Silent notifications never contribute to the indicator (see Silent Notifications below).

14.3 Notification Types

The following events generate notifications:

Sender	Event
`plan`	A plan file is ready for user review and approval
`question`	An agent is asking the user a question (via `/sase_questions`)
`hitl`	A workflow HITL step is waiting for user input
`memory.proposed`	A long-term memory proposal is ready for human review
`sync`	A sync operation completed for a ChangeSpec
`axe`	Hourly error digest summarizing recent axe errors
`mentors`	All mentors finished for a ChangeSpec entry (or none matched)
Workflow-specific sender label	Workflow completion (success or failure)

14.3.1 Agent Completion Attachments

Agent completion notifications attach the standard chat transcript and diff first. On failures they also include the error report and output log when those files exist. When a successful agent added or modified 10 or fewer Markdown files, SASE renders best-effort PDF artifacts and appends those PDFs after the standard artifacts. When the run added or modified image files, SASE appends those generated images after any Markdown PDFs. Explicit artifacts created during the run with `sase artifact create -p <path> [-n <label>] [-k <kind>]` are read from the persistent artifact index and appended last when their stored files still exist. Supported Markdown extensions are `.md` and `.markdown`; supported image extensions are `.png`, `.jpg`, `.jpeg`, `.webp`, and `.gif`.

Attachment paths are discovered from local git changes, untracked files, saved proposal/commit diffs, and the latest commit when the agent committed or opened a PR. Missing, deleted, unsupported, and duplicate paths are ignored. If more than 10 Markdown sources remain after filtering, SASE skips Markdown PDF rendering for that completion and includes a note explaining the limit. The final PDF and image lists are also written to `done.json` as `markdown_pdf_paths` and `image_paths` for agent metadata consumers. Explicit artifact paths are read from the explicit-artifact association index at notification time, deduplicated against the standard attachments, and ignored if the index is unavailable.

In ACE, completion artifacts are opened from the Agents tab with `A`. The artifact panel supports marking multiple files and opening the full artifact sequence, so notification attachments, generated PDFs/images, plan files, and explicit artifacts use one selection workflow. ACE may also include image files referenced by saved prompt artifacts in that picker. Those prompt-referenced images are persisted or synthesized as ACE artifact-list entries, but they are not appended to notification delivery payloads unless they also appear in `done.json.image_paths` or were saved explicitly with `sase artifact create`.

The Agents tab also treats user-agent completions as unread work items. When a terminal agent is selected after it has been marked unread, or when the user jumps to it with the unread-agent shortcut, ACE clears the row's unread marker and dismisses the matching completion notification. Plan approvals and user questions remain explicit response workflows and are not auto-read merely by selection.

Unread state on the Agents tab is projected from the active user-agent completion notifications in the store rather than written as separate per-row state — when the underlying notification is dismissed (per-row selection, response modal, or any other path) the row's unread marker clears on the next refresh. Manually toggling a row unread with `U` overrides this projection locally so a deliberately re-flagged row is not immediately re-cleared. Plan approvals and user questions still require an explicit `y` / `n` response and are never auto-dismissed by row navigation.

See `agent_images.md` (https://sase.sh/agent_images/) for the full attachment contract and ACE image preview notes.

For user-agent completion and failure notifications, `action_data` also includes `bead_display` when the agent name maps to a bead created by `sase bead work`. The value includes the bead ID plus the issue description or title when the bead can be resolved, and falls back to the ID alone otherwise. Cross-project lookups prefer the agent's owning project, then the caller's current bead view, then all known SASE projects.

14.3.2 Mentors-Complete Notification

A mentors-complete notification uses sender `mentors` and fires once per `(ChangeSpec, COMMITS entry)` under either of two conditions:

- **All mentors terminal** — every mentor that was started for the entry has reached a terminal status (`PASSED`, `COMMENTED`, `FAILED`, `DEAD`, or `KILLED`).
- **No matching profile** — every hook is ready and no mentor profile matched the ChangeSpec, so no mentors will run.

Selecting the notification jumps to the PRs tab, focuses the target ChangeSpec, and pushes the Mentor Review modal when at least one mentor produced reviewable output.

Idempotency is enforced via `~/.sase/notifications/mentors_complete.json`, keyed on `(project_file, changespec_name, entry_id)` — so the notification survives process restarts and project-spec archival without re-firing. The sender suppresses the notification on the same axe cycle that just wrote the `MENTORS` field for the latest entry, preventing premature firing on `Draft → Ready` transitions.

14.3.3 Memory Proposal Notification

`sase memory write --notify` first saves the proposal, then best-effort creates a `memory.proposed` notification. The notification includes the `memory` tag, evidence entries that resolved to local file paths, `action: memory_review`, and `action_data.proposal_id`. Selecting it in ACE suspends the main TUI and opens the same interactive review app as `sase memory review`, preselected on that proposal. Review decisions still happen in that app; the notification is only the entry point. Proposal creation still succeeds if notification delivery fails, and the CLI reports the notification id when delivery succeeds.

14.4 Notification Fields

Each notification contains:

Field	Type	Description
`id`	string	UUID4 unique identifier
`timestamp`	string	ISO-8601 creation timestamp
`sender`	string	Source identifier (e.g., "plan", "sync", "axe")
`notes`	list[string]	Human-readable message lines
`files`	list[string]	Associated file paths (e.g., plan files, error digest files, generated agent images)

`tags`	list[string]	Optional normalized labels for filtering and modal tag tabs
`action`	string	Action type: `HITL`, `JumpToChangeSpec`, `PlanApproval`, etc.
`action_data`	dict	Action-specific data (e.g., response directory, CL name)
`read`	bool	Whether the notification has been read
`dismissed`	bool	Whether the notification has been dismissed
`silent`	bool	Silent notifications are stored but hidden from the TUI
`muted`	bool	Muted notifications appear under the MUTED section and are excluded from the bell indicator and toasts
`snooze_until`	string|null	ISO-8601 timestamp at which a snoozed notification automatically un-mutes

14.5 Silent Notifications

Notifications from hidden background agents (summarize-hook, fix-hook, mentor) are created with `silent=True`. Silent notifications are written to the JSONL file (preserving the audit trail) but excluded from the TUI unread count, bell indicator, toast, notification modal, and Telegram delivery. They remain visible to local inspection commands such as `sase notify list`.

Agent completion and failure events from hidden background agents still write a notification row, but with the silent flag set. This keeps the JSONL audit trail complete while keeping the inbox focused on user-facing agent work.

14.6 Tags

Senders may attach `tags` to a notification. Tags are normalized when notifications are created: whitespace is trimmed, empty values are dropped, values are lowercased, and duplicates are removed while preserving sender order. Tags do not change priority, error classification, unread counts, mute, snooze, or auto-dismiss matching.

Successful visible and hidden user-agent completion notifications that jump back to the agent row carry the `done` tag. Failed user-agent notifications do not carry `done`; failures remain error reports.

Memory proposal notifications created by `sase memory write --notify` carry the `memory` tag. Use the `memory` tab in ACE or `sase notify list --tag memory` to find proposal review notification rows.

In ACE, tags create modal tabs above the notification list. The `done` tab is intended as the quick path for successful agent completions; reading or jumping to a done Agents-tab row dismisses its matching completion notification, so it disappears from both `All` and `done` after the next refresh. Failed agent notifications stay untagged by `done` and continue to render under **ERRORS**.

14.7 CLI

The `sase notify` command can create notifications and inspect the local notification inbox.

Bare `sase notify` is a read-only shortcut for `sase notify list`. Use `sase notify create` when writing a notification from JSON input:

```
echo '{"sender": "test", "notes": ["Hello"], "tags": ["review"]}' | sase notify create
sase notify create -s my_sender < notification.json
sase notify create -s my_sender --tag review --tag handoff < notification.json
```

For read-only inspection, list recent notifications as either a compact table or stable JSON:

```
sase notify
sase notify list
sase notify list -j -l 20
sase notify list -j --sender axe
sase notify list -j --unread
sase notify list -j --tag done
sase notify list -j --tag memory
sase notify list -j -q digest
sase notify list -j --all
```

Use the explicit `list` subcommand when passing list flags; for example, use `sase notify list -j`, not `sase notify -j`.

`sase notify list -j` prints notifications newest first with `id`, `timestamp`, `age`, `sender`, `priority`, `notes`, `files`, `tags`, `action`, `action_data`, `read`, `dismissed`, `silent`, `muted`, and `snooze_until`. The `-q/--query` filter matches tags as well as ids, senders, notes, files, actions, and action data. Dismissed notifications are hidden unless `--all` is provided.

Inspect one notification by id:

```
sase notify show --id <notification_id>
sase notify show --id <notification_id> -f json
sase notify show --id <notification_id> -f markdown
```

The default `show` format is markdown. It includes the notification tags, notes, attached file paths, action data, and state flags. Axe error digest notifications usually point to the actionable report through `files` or `action_data.error_report_path`; read that attached file for the detailed errors.

To create a local test notification with a persistent PNG attachment for ACE modal image-preview checks, run `tools/test_image_notification` from the repository root.

See `docs/configuration.md` (https://sase.sh/configuration/#sase-notify) for the full CLI reference.

14.8 Storage

Notifications are stored in JSONL format at `~/.sase/notifications/notifications.jsonl`. The production store backend is `sase_core_rs`: appends and state mutations take a shared sidecar lock, and rewrites use a tempfile plus rename so multiple axe processes and the TUI can access the file without truncate-before-lock exposure. Common state-only updates such as mark-read, mark-all-read, mute, snooze, and dismiss use a count-only Rust mutation path unless the caller needs rehydrated notification rows; this keeps inbox counters cheap when ACE or a bridge process only needs mutation metadata.

Source: `src/sase/notifications/`

15 Mobile Gateway

The SASE mobile gateway is a workstation-hosted HTTP gateway for paired mobile clients. The phone is only a client: it pairs with the host, stores a bearer token, calls product-shaped SASE APIs, and subscribes to server-sent events. The gateway never exposes a generic file, shell, or RPC surface.

The implementation is split across repos:

- `sase` owns user configuration, CLI startup, and lifecycle glue through `sase mobile gateway start`.
- `sase.integrations.mobile_notifications` is the stable Python facade used by the gateway host bridge to project local notifications, build attachment manifests, and execute plan/HITL/question actions. The sibling `_mobile_notification_*` modules are internal implementation details.
- `sase.integrations.mobile_agents` and `sase.integrations.mobile_helpers` are the fixed-operation bridge facades used by the Rust gateway to list/launch/kill/retry agents and to expose ChangeSpec, xprompt, bead, and update helpers. The sibling `_mobile_agent_*` and `_mobile_helper_*` modules are internal implementation details.
- `../sase-core/crates/sase_gateway` owns the Rust HTTP server, wire records, pairing/token storage, audit log, SSE event stream, and committed API contract snapshot.
- `../sase-android/README.md` is the Android client handoff for build/test commands, fake-gateway coverage, foreground UX smoke checks, and Epic 7 limitations.
- `docs/mobile_mvp_runbook.md` (https://sase.sh/mobile_mvp_runbook/) is the install/operate/security runbook for private APK distribution, Tailscale Serve remote access, push payload boundaries, troubleshooting, and rollback.

15.1 Start Locally

Install the local checkout first so the Python CLI and sibling Rust binaries are available:

```
just install
cargo build -p sase_gateway --manifest-path ../sase-core/Cargo.toml
```

`sase mobile gateway start` resolves the gateway binary from `PATH`, then from the sibling `../sase-core` debug or release target. Use `-c /path/to/sase_gateway` only when you need to override that lookup.

Start the gateway from SASE:

```
sase mobile gateway start
```

By default this runs the Rust gateway in the foreground on `127.0.0.1:7629`, waits for `GET /api/v1/health`, creates a one-time pairing challenge, and prints:

```
Starting SASE mobile gateway at http://127.0.0.1:7629
Pairing code: 123456
Pairing ID: pair_abc123
Expires at: 2026-05-06T15:00:00Z
Keep this process running while mobile clients connect.
```

Keep that process running while clients connect. Stop it with `Ctrl-C`.

Useful startup overrides:

```
sase mobile gateway start -p 7630
sase mobile gateway start -H /tmp/sase-mobile-state
sase mobile gateway start -c "../sase-core/target/debug/sase_gateway"
```

`-H` sets the SASE home root passed to the Rust gateway as `--sase-home`; gateway files are then stored below that root in `mobile_gateway/`. The matching configuration keys live under `mobile_gateway`:

```yaml
mobile_gateway:
  bind_address: "127.0.0.1"
  port: 7629
  state_dir: ""
  allow_non_loopback: false
  command: ""
  agent_bridge_command: ""
  helper_bridge_command: ""
  push_provider: "disabled"
  fcm_project_id: ""
  fcm_service_account_json: ""
  fcm_credential_env: ""
  fcm_dry_run: false
  push_timeout_seconds: 5
  push_retry_limit: 1
  startup_timeout_seconds: 10
```

15.2 Host Bridge CLI Commands

The gateway shells out only to fixed JSON-over-stdin bridge commands. These commands are intentionally narrow integration surfaces for the Rust gateway and mobile clients; they are not general user workflows and they do not accept mobile-supplied shell commands, cwd values, environment variables, or host paths.

Agent bridge operations:

Command	Purpose
`sase mobile agent-bridge list-agents`	Return running agents, with optional recent completions
`sase mobile agent-bridge resume-options`	Return copy/share/direct-launch resume prompt options
`sase mobile agent-bridge launch-text`	Launch a text prompt in home or known-project context
`sase mobile agent-bridge launch-image`	Store an uploaded image and launch an image prompt
`sase mobile agent-bridge kill-agent`	Kill an agent by exact name
`sase mobile agent-bridge retry-agent`	Retry an agent by name, timestamp, or mobile context

Helper bridge operations:

Command	Purpose
`sase mobile helper-bridge changespec-tags`	List active ChangeSpec prompt tags for a known project
`sase mobile helper-bridge xprompt-catalog`	Return the mobile-safe structured xprompt catalog
`sase mobile helper-bridge beads-list`	List open or in-progress beads
`sase mobile helper-bridge beads-show`	Inspect one bead by ID
`sase mobile helper-bridge update-start`	Start the configured SASE update worker
`sase mobile helper-bridge update-status`	Poll structured update worker status

15.3 Push Hints

Push delivery is disabled by default. When enabled, the gateway sends hint-only records derived from the same event stream used by SSE. Push payloads contain safe IDs, categories, reasons, and short display text; they do not contain bearer tokens, pairing codes, prompt bodies, response text, attachment contents, attachment tokens, or host paths. The mobile client must always fetch authenticated gateway state after a push wake or notification tap.

Local development can use the test provider, which records attempted delivery without leaving the process:

```
sase mobile gateway start -P test
```

FCM HTTP v1 is configured with non-secret pointers. Use either a service-account JSON path or an environment variable containing a short-lived bearer token or service-account JSON:

```
export SASE_FCM_CREDENTIAL='...'
sase mobile gateway start \
  -P fcm \
  -F my-firebase-project \
  -E SASE_FCM_CREDENTIAL \
  -D
```

Equivalent config:

```
mobile_gateway:
  push_provider: "fcm"
  fcm_project_id: "my-firebase-project"
  fcm_service_account_json: "~/.config/sase/firebase-service-account.json"
  fcm_credential_env: ""
  fcm_dry_run: false
  push_timeout_seconds: 5
  push_retry_limit: 1
```

Only credential paths or environment-variable names are passed on the gateway command line. Do not commit service-account JSON, Firebase project secrets, `google-services.json`, signing keys, or local tailnet hostnames.

After pairing, a mobile client registers its provider token through the authenticated session routes. `provider_token` is the opaque device/app token from the push provider; treat it as device credential material and avoid committing or logging real values.

```
TOKEN="sase_mobile_example"

curl -sS -X POST http://127.0.0.1:7629/api/v1/session/push-subscriptions \
  -H "Authorization: Bearer $TOKEN" \
  -H 'Content-Type: application/json' \
  -d '{
    "schema_version": 1,
    "provider": "fcm",
    "provider_token": "opaque-fcm-token",
    "app_instance_id": "pixel-9-app-instance",
    "device_display_name": "Pixel 9",
    "platform": "android",
    "app_version": "0.1.0",
    "hint_categories": ["notifications", "agents", "update"]
  }'
```

Duplicate registrations with the same provider token update the existing subscription. Clients can list active subscriptions and revoke one by ID:

```
curl -sS http://127.0.0.1:7629/api/v1/session/push-subscriptions \
  -H "Authorization: Bearer $TOKEN"

SUBSCRIPTION_ID="sub_example"

curl -sS -X DELETE "http://127.0.0.1:7629/api/v1/session/push-subscriptions/$SUBSCRIPTION_ID" \
  -H "Authorization: Bearer $TOKEN"
```

15.4 Private Remote Access And Packaging

The MVP runbook covers debug APK installation, signed release APKs, Firebase-configured internal builds, versioning, Tailscale Serve setup, threat model, and rollback steps:

```
docs/mobile_mvp_runbook.md
```

Keep the gateway on `127.0.0.1` when using Tailscale Serve. Tailscale proxies the loopback gateway inside the tailnet, so the SASE process does not need a LAN or public bind. Use `--allow-non-loopback` / `-L` only for explicit LAN or tailnet address binds during short smoke windows.

15.5 Hardening Test Commands

CI-friendly smoke slices for the three-repo mobile gateway surface:

```
# Android fake gateway, push hint, and background wake regressions.
(cd ../sase-android && ./gradlew testDebugUnitTest --tests org.sase.mobile.testing.BackgroundHardeningSmokeTest)
(cd ../sase-android && ./gradlew testDebugUnitTest --tests org.sase.mobile.testing.FakeGatewayTest)

# Optional local emulator/device coverage.
(cd ../sase-android && ./gradlew connectedDebugAndroidTest)

# Rust push subscription, test provider, and temporary listener smoke coverage.
(cd ../sase-core && cargo test -p sase_gateway push_subscription)
(cd ../sase-core && cargo test -p sase_gateway test_push_provider_records_hint_attempts)
(cd ../sase-core && cargo test -p sase_gateway listener_smoke_exercises_pairing_auth_and_session)

# Python gateway config/argv bridge coverage.
.venv/bin/pytest tests/test_mobile_gateway.py
```

15.6 Pairing Flow

The local host starts pairing with `POST /api/v1/session/pair/start`. The response contains a short-lived one-time code and a `pairing_id`; it does not contain a long-lived credential. A mobile client finishes pairing by sending the code, the `pairing_id`, and device metadata to `POST /api/v1/session/pair/finish`.

Example:

```
PAIRING_ID="pair_abc123"
PAIRING_CODE="123456"

curl -sS http://127.0.0.1:7629/api/v1/session/pair/finish \
  -H 'Content-Type: application/json' \
  -d "{
    \"schema_version\": 1,
    \"pairing_id\": \"$PAIRING_ID\",
    \"code\": \"$PAIRING_CODE\",
    \"device\": {
      \"display_name\": \"Pixel 9\",
      \"platform\": \"android\",
      \"app_version\": \"0.1.0\"
    }
  }"
```

The finish response returns the bearer token exactly once:

```
{
  "schema_version": 1,
  "device": {
    "schema_version": 1,
    "device_id": "dev_example",
    "display_name": "Pixel 9",
    "platform": "android",
    "app_version": "0.1.0",
    "paired_at": "2026-05-06T15:00:00Z",
    "last_seen_at": null,
    "revoked_at": null
  },
  "token_type": "bearer",
  "token": "sase_mobile_example"
}
```

Store the token on the client as a secret. Future authenticated requests use:

```
TOKEN="sase_mobile_example"

curl -sS http://127.0.0.1:7629/api/v1/session \
  -H "Authorization: Bearer $TOKEN"
```

Only `GET /api/v1/health`, `POST /api/v1/session/pair/start`, and `POST /api/v1/session/pair/finish` are unauthenticated. All other routes require this bearer token.

15.7 Events

Authenticated clients subscribe to `GET /api/v1/events` with `Accept: text/event-stream`:

```
curl -N http://127.0.0.1:7629/api/v1/events \
  -H "Authorization: Bearer $TOKEN" \
  -H 'Accept: text/event-stream'
```

Events are JSON `EventRecordWire` records carried in SSE `data:` lines. Each event has a stable monotonic string ID such as `0000000000000001`. Reconnect with `Last-Event-ID` to replay buffered events newer than the last processed ID:

```
curl -N http://127.0.0.1:7629/api/v1/events \
  -H "Authorization: Bearer $TOKEN" \
  -H 'Accept: text/event-stream' \
  -H 'Last-Event-ID: 0000000000000001'
```

The first implementation keeps the event buffer in memory. After a gateway restart or buffer overflow, clients must handle a `resync_required` event by fetching full state again.

15.8 Notifications, Actions, And Attachments

All notification, action, and attachment routes require the paired device bearer token:

```
BASE_URL="http://127.0.0.1:7629"
TOKEN="sase_mobile_example"
AUTH_HEADER="Authorization: Bearer $TOKEN"
```

List unread, non-silent notifications newest first:

```
curl -sS "$BASE_URL/api/v1/notifications?unread=true&limit=25" \
  -H "$AUTH_HEADER"
```

Include dismissed or silent rows only when a client is rebuilding local state:

```
curl -sS "$BASE_URL/api/v1/notifications?include_dismissed=true&include_silent=true" \
  -H "$AUTH_HEADER"
```

Inspect a plan approval notification before acting on it:

```
NOTIFICATION_ID="abcdef12-plan"

curl -sS "$BASE_URL/api/v1/notifications/$NOTIFICATION_ID" \
  -H "$AUTH_HEADER"
```

Detail responses include full notes, action state, and attachment manifests. Download tokens are minted only in detail responses, are bound to the authenticated device, expire after a short TTL, and must still pass path and size checks at download time.

Mark a notification read or dismiss it without taking its pending action:

```
curl -sS -X POST "$BASE_URL/api/v1/notifications/$NOTIFICATION_ID/mark-read" \
  -H "$AUTH_HEADER"

curl -sS -X POST "$BASE_URL/api/v1/notifications/$NOTIFICATION_ID/dismiss" \
  -H "$AUTH_HEADER"
```

Both routes mutate only notification state and return `notification_id`, `read`, `dismissed`, and `changed`. Repeating a route is idempotent: `changed` is `false` when the requested state was already set. Successful state mutations audit the device and publish `notifications_changed` so clients can refresh list/detail state.

Plan approval actions use the notification ID or any unique pending-action prefix:

```
PREFIX="abcdef12"

curl -sS -X POST "$BASE_URL/api/v1/actions/plan/$PREFIX/approve" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"commit_plan":true,"run_coder":false}'

curl -sS -X POST "$BASE_URL/api/v1/actions/plan/$PREFIX/run" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"coder_prompt":"Focus on tests"}'

curl -sS -X POST "$BASE_URL/api/v1/actions/plan/$PREFIX/reject" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"feedback":"Please narrow the scope"}'

curl -sS -X POST "$BASE_URL/api/v1/actions/plan/$PREFIX/feedback" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"feedback":"Revise the rollout section"}'
```

Epic and legend approvals use the same route shape:

```
curl -sS -X POST "$BASE_URL/api/v1/actions/plan/$PREFIX/epic" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1}'

curl -sS -X POST "$BASE_URL/api/v1/actions/plan/$PREFIX/legend" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1}'
```

HITL prompts can be accepted, rejected, or returned with feedback:

```
HITL_PREFIX="hitl0001"

curl -sS -X POST "$BASE_URL/api/v1/actions/hitl/$HITL_PREFIX/accept" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1}'

curl -sS -X POST "$BASE_URL/api/v1/actions/hitl/$HITL_PREFIX/reject" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1}'

curl -sS -X POST "$BASE_URL/api/v1/actions/hitl/$HITL_PREFIX/feedback" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"feedback":"Use a smaller change"}'
```

Question prompts support stable option IDs, option indices, labels, and custom free text:

```
QUESTION_PREFIX="quest001"

curl -sS -X POST "$BASE_URL/api/v1/actions/question/$QUESTION_PREFIX/answer" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"selected_option_id":"safe","global_note":"Use the durable path"}'

curl -sS -X POST "$BASE_URL/api/v1/actions/question/$QUESTION_PREFIX/answer" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"selected_option_index":1}'

curl -sS -X POST "$BASE_URL/api/v1/actions/question/$QUESTION_PREFIX/custom" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"custom_answer":"Use SQLite","global_note":"Small local DB"}'
```

Download an attachment by using a token from a notification detail response:

```
ATTACHMENT_TOKEN="att_example"

curl -sS "$BASE_URL/api/v1/attachments/$ATTACHMENT_TOKEN" \
  -H "$AUTH_HEADER" \
  -o attachment.bin
```

Mutating notification state and action routes audit device ID, endpoint, target notification/prefix, and outcome. Duplicate, stale, ambiguous, already-handled, unsupported, and missing-target cases return typed `ApiErrorWire` records and do not overwrite existing response files.

15.9 Agents

Agent lifecycle routes also require the paired device bearer token. They call fixed host bridge operations, not mobile-supplied commands, cwd values, or environment variables.

List running agents:

```
curl -sS "$BASE_URL/api/v1/agents" \
  -H "$AUTH_HEADER"
```

Include recent completed/failed agents, filter by status or known project, and cap the response:

```
curl -sS "$BASE_URL/api/v1/agents?include_recent=true&project=sase&status=running&limit=25" \
  -H "$AUTH_HEADER"
```

Fetch copy/share/direct-launch resume and wait prompt options:

```
curl -sS "$BASE_URL/api/v1/agents/resume-options" \
  -H "$AUTH_HEADER"
```

Launch a text agent:

```
curl -sS -X POST "$BASE_URL/api/v1/agents/launch" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"prompt":"Summarize the current failures","name":"mobile.summary"}'
```

Launch in a known SASE project context by passing the project name, not a path. The bridge resolves only `<sase_home>/projects/<project>/<project>.sase` (falling back to legacy `.gp`) and uses that file's `WORKSPACE_DIR` when it needs a project cwd:

```
curl -sS -X POST "$BASE_URL/api/v1/agents/launch" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"project":"sase","prompt":"Run the focused mobile gateway tests","name":"mobile.sase"}'
```

Project lifecycle still applies. Inactive projects are hidden from broad mobile helper catalogs, and normal launch resolution refuses new workspace claims for inactive projects with the same activation hint shown by CLI/TUI launches. Run `sase project activate <project>` on the host before launching new mobile-initiated work in an inactive project.

For home-mode launches, omit `project` or pass `"home"`. For VCS-ref launches, include the normal SASE prompt syntax such as `#gh:12345` or legacy `#gh@12345`; the bridge normalizes legacy `@` refs before launch and persists a product-shaped context ID such as `project:sase:gh:12345`. Android must never send host paths as project context.

Launch an image agent with base64 image bytes. The host stores the image under SASE-owned gateway state and injects the absolute saved path into the agent prompt:

```
IMAGE_B64="$(base64 -w0 screenshot.png)"
IMAGE_BYTES="$(wc -c < screenshot.png)"

curl -sS -X POST "$BASE_URL/api/v1/agents/launch-image" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d "{
    \"schema_version\": 1,
    \"prompt\": \"Review this screenshot\",
    \"name\": \"mobile.image\",
    \"original_filename\": \"screenshot.png\",
    \"content_type\": \"image/png\",
    \"byte_length\": $IMAGE_BYTES,
    \"base64_image\": \"$IMAGE_B64\"
  }"
```

Kill an agent by exact name:

```
AGENT_NAME="mobile.summary"

curl -sS -X POST "$BASE_URL/api/v1/agents/$AGENT_NAME/kill" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"reason":"mobile"}'
```

Retry an agent by exact name, artifact timestamp, or durable mobile context:

```
curl -sS -X POST "$BASE_URL/api/v1/agents/$AGENT_NAME/retry" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1}'
```

Successful launch, image launch, kill, and retry routes audit the paired device and publish `agents_changed` SSE events with a reason and agent name when available. Mobile launch context is appended to `<sase_home>/mobile_gateway/agent_launch_contexts.jsonl`; mobile kill context is stored under `<sase_home>/mobile_gateway/agent_kill_contexts/`; and the last known product-shaped project context per device is stored under `<sase_home>/mobile_gateway/device_project_contexts/`.

15.10 Workflow Helpers

Workflow helper routes require the paired device bearer token. They return direct JSON records with a common `result` object containing `status`, `message`, `warnings`, `skipped`, and `partial_failure_count`. Android should render `partial_success` by showing the primary payload and surfacing the structured skipped rows; it should not parse the human message text for control flow.

List active ChangeSpec tags for a known project:

```
curl -sS "$BASE_URL/api/v1/changespec-tags?project=sase&limit=25" \
  -H "$AUTH_HEADER"
```

Fetch the structured xprompt catalog for a native picker. Optional PDF generation is best-effort and requested explicitly:

```
curl -sS "$BASE_URL/api/v1/xprompts/catalog?project=sase&tag=changespec&limit=50" \
  -H "$AUTH_HEADER"

curl -sS "$BASE_URL/api/v1/xprompts/catalog?project=sase&include_pdf=true" \
  -H "$AUTH_HEADER"
```

Each xprompt catalog entry includes the display-only `input_signature` plus mobile editor metadata: `insertion`, `reference_prefix`, `kind`, `definition_path` when a real source file can be resolved, and an `inputs` array of `{name, type, required, default_display, position}` records. Android should insert `insertion` when present, fall back to `#<name>` for older gateways, and use `inputs` only for prompt-adjacent argument hints. The raw launch prompt remains authoritative and is sent unchanged.

List open/in-progress beads in a project and inspect one bead:

```
curl -sS "$BASE_URL/api/v1/beads?project=sase&status=in_progress&limit=25" \
  -H "$AUTH_HEADER"

BEAD_ID="sase-26.4.6"

curl -sS "$BASE_URL/api/v1/beads/$BEAD_ID?project=sase" \
  -H "$AUTH_HEADER"
```

Omit `project` to use the paired device's remembered project context when present. Pass `all_projects=true` to force a cross-project bead lookup across known SASE projects.

Start the fixed SASE update worker and poll structured job status:

```
curl -sS -X POST "$BASE_URL/api/v1/update/start" \
  -H "$AUTH_HEADER" \
  -H 'Content-Type: application/json' \
  -d '{"schema_version":1,"request_id":"mobile-update-1"}'

JOB_ID="job_123"

curl -sS "$BASE_URL/api/v1/update/$JOB_ID" \
  -H "$AUTH_HEADER"
```

The helper endpoints are read-only except `POST /api/v1/update/start`. Mobile cannot provide a shell command, cwd, environment, workspace path, project file path, or arbitrary bridge argv. The gateway invokes fixed `sase mobile helper-bridge <operation>` commands; the update worker itself runs only the configured `chat_install.command`. Successful update start/status checks publish `helpers_changed` events, but polling `GET /api/v1/update/{job_id}` is the authoritative completion path for the MVP.

15.11 Storage And Revocation

Gateway state is stored under `<sase_home>/mobile_gateway/`. With the default SASE home, that is `~/.sase/mobile_gateway/`.

- Paired devices live in `devices.json`.
- Mobile agent launch, kill, upload, and per-device project context state also lives under this directory.
- Raw bearer tokens are not written to disk; only SHA-256 token hashes are stored.
- Audit records append to `audit.jsonl` with device ID, endpoint, target ID when available, and outcome. Audit records avoid pairing codes and bearer tokens.
- The Rust store includes a revocation primitive so future UI/API work can mark a device revoked. Revoked tokens fail authentication.

15.12 Network Exposure

The gateway binds to `127.0.0.1` by default. This is intentional: a local-only bind is the safe desktop development and same-host test path.

For private remote access, prefer Tailscale Serve or an equivalent private tailnet path that terminates access control outside the gateway. Keep the gateway bound to loopback when possible, then serve the loopback endpoint through the private tailnet configuration.

LAN and public-interface binds are explicit opt-in only:

```
sase mobile gateway start -b 100.64.0.10 -p 7629 -L
```

Only use `--allow-non-loopback` / `-L` on trusted private networks. Do not expose the gateway directly to the public internet. The MVP pairing flow uses auditable one-time pairing codes and bearer tokens, not a hardened mTLS or SPAKE2 protocol.

15.13 Contract Snapshot

The committed API contract snapshot for Android/client work lives in the sibling Rust repo:

```
../sase-core/crates/sase_gateway/contracts/api_v1/mobile_api_v1.json
```

Regenerate it from the Rust workspace with:

```
cd ../sase-core
cargo run -p sase_gateway -- \
  --contract-out crates/sase_gateway/contracts/api_v1/mobile_api_v1.json
```

Keep the JSON snapshot, Rust wire tests, and this document aligned whenever the gateway route or record shape changes.

15.14 Known MVP Limitations

- Notification reads are authoritative REST reads from the host JSONL store. Successful gateway state/action mutations publish `notifications_changed` SSE events, but passive file watching is intentionally out of the MVP.
- Attachment downloads are capped by the gateway's configured max attachment bytes. Oversized, missing, directory, traversal, symlinked, or unknown-risk files appear in manifests without download tokens.
- Telegram remains supported through the shared pending-action compatibility path. If an older Telegram install has only legacy pending-action JSON, mobile can read it, but Telegram is not yet required to use the shared store as its only source of truth.
- Agent project context is intentionally an MVP metadata and known-project cwd selector. It does not expose arbitrary host directory selection, and clients must use SASE prompt syntax for VCS refs rather than sending raw repo paths.
- Workflow helper routes are native helper APIs, not generic command execution. ChangeSpec, xprompt, and bead helpers are read-only; xprompt PDF generation is optional; and update completion events are opportunistic while status polling remains authoritative.

16 Mobile MVP Runbook

This runbook covers the private Android MVP that connects a phone to a workstation-hosted SASE mobile gateway. The phone is a client only: it stores a paired bearer token, renders notifications, opens fixed SASE workflows, and fetches authoritative state from the gateway. It does not run agents locally and the gateway does not expose arbitrary shell, file-browse, or RPC access.

Use this with:

- Android app: `../sase-android`
- Rust gateway: `../sase-core/crates/sase_gateway`
- SASE host CLI: this repo, `sase mobile gateway start`

Prerequisites:

- JDK 21, Android SDK command-line tools, Android platform `android-35`, and Android build tools `35.0.0`.
- `ANDROID_HOME` or `ANDROID_SDK_ROOT` set when the Android SDK is not discoverable automatically.
- `adb` on `PATH` for physical-device or emulator installs.
- The sibling `sase-android` and `sase-core` checkouts beside this repo.

16.1 Build And Install

16.1.1 Debug APK

Use a debug APK for local development and manual smoke tests:

```
cd ../sase-android
./gradlew testDebugUnitTest lintDebug assembleDebug
adb install -r app/build/outputs/apk/debug/app-debug.apk
```

The debug build does not require Firebase project files. If `app/google-services.json` is absent, the app still builds, but push registration reports an unconfigured provider in Settings.

16.1.2 Internal APK With FCM

For an internal/direct APK that should receive FCM push hints:

1. Create a Firebase Android app with package `org.sase.mobile`.
2. Place the downloaded Android config at `../sase-android/app/google-services.json`.
3. Keep `google-services.json` local; it is ignored by git.
4. Build the APK:

```
cd ../sase-android
./gradlew testDebugUnitTest lintDebug assembleDebug
```

FCM payloads are hints only. They may contain event IDs, categories, routing IDs, short safe titles, and short safe bodies. They must not contain bearer tokens, pairing codes, prompt bodies, response text, attachment contents, attachment tokens, host paths, signing material, or Firebase credentials.

16.1.3 Signed Release APK

Release signing is configured from local-only Gradle properties, `local.properties`, or environment variables. Do not commit keystores or signing values.

Required keys:

```
SASE_ANDROID_RELEASE_STORE_FILE=/absolute/path/to/sase-mobile-upload.jks
SASE_ANDROID_RELEASE_STORE_PASSWORD=...
SASE_ANDROID_RELEASE_KEY_ALIAS=sase-mobile
SASE_ANDROID_RELEASE_KEY_PASSWORD=...
```

Build:

```
cd ../sase-android
./gradlew testDebugUnitTest lintDebug assembleRelease
```

The release build currently leaves minification off for private distribution. Revisit minification and keep rules before any broad public release.

16.1.4 Versioning And Upgrades

The Android MVP keeps paired-host metadata, bearer tokens, cached inbox state, action drafts, foreground-mode preference, push registration status, and remembered update jobs in app-private storage. Preserve the application ID `org.sase.mobile` and increase `versionCode` for upgrades. Users who install a differently signed APK may need to uninstall first, which deletes local session/cache state and requires re-pairing.

16.2 Host Setup

Install the SASE checkout and build the Rust gateway:

```
cd /path/to/sase_100
just install
cargo build -p sase_gateway --manifest-path ../sase-core/Cargo.toml
```

The host CLI starts the first gateway binary it can resolve from `PATH`, `../sase-core/target/debug/sase_gateway`, or `../sase-core/target/release/sase_gateway`. Pass `-c /path/to/sase_gateway` only when you need to override that resolution.

Start the gateway on loopback:

```
sase mobile gateway start
```

The command prints a pairing code, pairing ID, and expiration. Keep the process running while mobile clients connect.

For an emulator pointed at host loopback, use `http://10.0.2.2:7629` as the base URL in the Android app. For a physical device on the same trusted LAN, bind explicitly to a LAN address only when needed:

```
sase mobile gateway start -b 192.0.2.10 -L
```

Do not bind to `0.0.0.0` or a public interface unless you have a separate network control that restricts access to your own device. Pairing and bearer auth protect the product API, but the workstation remains the trust boundary.

16.3 Push Provider

Push delivery is disabled by default:

```
mobile_gateway:
  push_provider: "disabled"
```

Use the test provider for local gateway coverage without sending traffic off the workstation:

```
sase mobile gateway start -P test
```

Use FCM only for internal APKs that include Firebase Android configuration:

```
export SASE_FCM_CREDENTIAL='...'
sase mobile gateway start \
  -P fcm \
  -F my-firebase-project \
  -E SASE_FCM_CREDENTIAL \
  -D
```

Equivalent config:

```
mobile_gateway:
  push_provider: "fcm"
  fcm_project_id: "my-firebase-project"
  fcm_service_account_json: "~/.config/sase/firebase-service-account.json"
  fcm_credential_env: ""
  fcm_dry_run: false
  push_timeout_seconds: 5
  push_retry_limit: 1
```

The host passes credential paths or environment-variable names to the Rust gateway, not credential contents on the process command line. Store service-account files under user-private config directories.

16.4 Private Remote Access

Tailscale Serve is the recommended remote-access path for the MVP because it keeps the gateway on host loopback and proxies it only inside the tailnet. Tailscale documents Serve as a way to route traffic from tailnet devices to a local service and notes that Funnel is the public-internet option; do not use Funnel for the mobile MVP. See the official Serve docs: https://tailscale.com/docs/features/tailscale-serve (https://tailscale.com/docs/features/tailscale-serve) and https://tailscale.com/kb/1242/tailscale-serve (https://tailscale.com/kb/1242/tailscale-serve).

Use a recent Tailscale client. The Serve CLI changed in Tailscale 1.52, and the current command accepts a local port, partial URL, or full local URL as its target. If the command prompts to enable HTTPS for the tailnet, follow the Tailscale consent flow. Tailnet ACLs still apply to Serve traffic.

Start SASE on loopback:

```
sase mobile gateway start
```

In another terminal, expose that loopback service to your tailnet:

```
tailscale serve --bg http://127.0.0.1:7629
tailscale serve status
```

Use the reported tailnet HTTPS URL as the Android base URL. Keep Tailscale ACLs limited to the users/devices that should operate the workstation. Stop serving when finished:

```
tailscale serve reset
```

Fallback options:

- Emulator: `http://10.0.2.2:7629` while the gateway binds host loopback.
- Trusted LAN: bind a specific host LAN address with `-L` and pair only on that network.
- USB reverse/tunnel tooling: acceptable for local development, but document the exact command in your local notes.

Avoid:

- Public tunnel services and Tailscale Funnel.
- Committing tailnet hostnames, Firebase service accounts, Android signing keys, or local gateway URLs.
- Exposing the gateway on a shared network without explicit need and a short operating window.

16.5 Manual Smoke

Run the automated gates first:

```
(cd ../sase-android && ./gradlew testDebugUnitTest lintDebug assembleDebug)
(cd ../sase-core && cargo test -p sase_gateway push_subscription)
(cd ../sase-core && cargo test -p sase_gateway test_push_provider_records_hint_attempts)
.venv/bin/pytest tests/test_mobile_gateway.py
```

Then smoke the end-to-end path:

1. Start `sase mobile gateway start`.
2. Install the APK and pair from Settings.
3. Verify session check, inbox refresh, and notification detail load.
4. Approve/reject a plan notification, launch a text agent, and kill or retry an agent.
5. Turn on foreground connected mode, background the app, trigger a gateway event, reopen the app, and verify state refreshes.
6. For FCM builds, verify push registration in Settings, send a test hint, tap the local notification, and confirm the app fetches host state before showing detail-sensitive UI.

7. Forget the host and verify the app returns to the unpaired state.

16.6 Troubleshooting

- Gateway refuses to bind: non-loopback addresses require `-L`; prefer loopback plus Tailscale Serve.
- Emulator cannot connect: use `10.0.2.2`, not `127.0.0.1`, from inside the emulator.
- Physical device cannot connect: verify phone and host are on the same tailnet/LAN and that the displayed base URL matches the exposed address.
- Push says unconfigured: check `app/google-services.json` for Android and `mobile_gateway.push_provider`/FCM credential config for the host.
- Push arrives but detail is stale: push is only a wake hint; verify the app can reach the authenticated gateway and refresh after receipt or tap.
- Auth failures after reinstall or host reset: forget the host in Android Settings and pair again.
- Foreground notification will not appear: verify Android notification permission and that connected mode is enabled.

16.7 Rollback

To roll back mobile access:

1. Stop the gateway process.
2. Stop Tailscale Serve with `tailscale serve reset`.
3. Forget the host in the Android app to revoke local session state and push subscription state.
4. Remove or rotate local FCM service-account credentials if they were exposed.
5. Use Telegram fallback for pending notifications/actions while mobile is disabled.

16.8 Threat Model

16.8.1 Assets

- Workstation SASE state, notifications, pending action files, agent launch context, and attachment files.
- Mobile bearer token and cached display state.
- Pairing code and pairing ID during their short validity window.
- FCM provider token/service account and Android app `google-services.json`.
- Android signing key.

16.8.2 Lost Phone

Risk: a paired phone can use its bearer token until the host forgets/revokes that device or the token becomes unusable.

Controls:

- Android stores the token in app-private secure storage.
- Users can forget the host from the app, and host-side session/token storage can be removed.
- Pairing codes are one-time and short-lived.

Operator response:

- Stop the gateway.
- Remove the device/session entry from gateway state or reset the mobile gateway state directory.
- Rotate any FCM device subscription if the host has not already marked it inactive.

16.8.3 LAN Attacker And Non-Loopback Bind

Risk: binding to LAN or `0.0.0.0` exposes the gateway transport to anyone who can route to the host, increasing online attack surface even though API routes require auth.

Controls:

- The Python launcher refuses non-loopback binds unless `--allow-non-loopback` / `-L` is passed.
- Product APIs require bearer auth after pairing.
- Helper routes expose fixed operations, not arbitrary commands.

Guidance:

- Prefer loopback plus Tailscale Serve.
- If LAN binding is necessary, bind a specific LAN IP for a short smoke window and stop it afterward.

16.8.4 Public Tunnels

Risk: public tunnels expose pairing and authenticated routes to the internet and make brute-force, scanner, and logging risks materially worse.

Controls and guidance:

- Do not use public tunnels or Tailscale Funnel for the MVP.
- Tailscale Serve keeps access inside the tailnet and respects tailnet ACLs.
- Pair only while physically operating both devices.

16.8.5 Attachment URL And Token Leakage

Risk: attachment download tokens grant temporary access to specific declared files for a paired device.

Controls:

- Tokens are minted only from authenticated detail responses.
- Tokens are bound to the authenticated device, expire after a short TTL, and are not stored in audit logs.
- Download checks reject traversal, symlink crossing, missing or changed files, non-regular files, and oversized files.

Guidance:

- Do not put attachment tokens in push payloads, logs, screenshots, or external issue trackers.
- Treat notification detail screens as potentially sensitive.

16.8.6 Arbitrary-Command Expansion

Risk: mobile helper or launch surfaces could become a general host command channel.

Controls:

- Gateway routes are product-shaped: notification actions, agent launch/kill/retry, fixed helpers, and update start.
- Helper bridge commands are configured by the host and invoked with fixed operations.
- Mobile clients cannot send cwd values, environment variables, host file paths, shell fragments, or arbitrary bridge argv for helper operations.

Guidance:

- Keep new mobile features behind explicit product commands.
- Reject proposals that turn mobile into a generic shell, file browser, or RPC client.

16.8.7 Push Provider Compromise Or Logs

Risk: FCM or intermediary logs may retain push metadata.

Controls:

- Push records are hints only.
- Provider tokens and service-account credentials are local-only and not audited.
- Delivery failures are best-effort diagnostics; user actions do not depend on push success.

Allowed in FCM payloads:

- Event ID or notification ID.
- Category/reason.
- Safe route hint.
- Short generic title/body.

Never allowed in FCM payloads:

- Bearer tokens, pairing codes, service-account contents, prompt bodies, response text, attachment contents, attachment tokens, raw host paths, local tailnet names, or signing material.

16.8.8 Notification Content Sensitivity

Risk: lock-screen notifications can be read by people near the phone.

Controls:

- Local notification rendering uses sanitized hint models.
- Sensitive content remains behind an authenticated gateway fetch after app open.

Guidance:

- Keep notification text generic for private builds.
- If a future build needs richer text, add an explicit user setting and revisit lock-screen visibility.

16.9 Known Limitations

- The MVP is for private/internal distribution, not Play Store production.
- FCM is the first push provider; UnifiedPush/ntfy remain future provider options behind the transport-agnostic subscription model.
- Push diagnostics are mostly in tests/logs and Android Settings state.
- Foreground connected mode improves durability but still follows Android background execution limits.
- Telegram remains the fallback for users who need a mature remote notification path.
- No follow-up beads were created during this close-out; material gaps above should be triaged under the parent epic or a future planning pass.

17 `sase ace` Performance Runbook

This runbook explains how to capture and compare performance data for ACE, the `sase ace` terminal user interface. It started as the Phase 1 deliverable for the TUI performance overhaul (bead `sase-w.1`, `sdd/epics/202604/tui_perf_overhaul_1.md`), and later performance phases still rely on the tracing and benchmark harness described here.

17.1 Trace recorder

`SASE_TUI_TRACE=1` enables `tui_trace(...)` context managers spread across the ChangeSpec, agents, and AXE hot paths. Each entered span emits one JSONL line to:

```
~/.sase/perf/tui_trace.jsonl
```

Override the destination with `SASE_TUI_TRACE_PATH=/tmp/foo.jsonl`. When the env flag is unset the context managers are near-zero-cost no-ops.

Each record contains at least:

```
ts            unix epoch seconds
span          dotted span name (e.g. "agents.refresh_panel_widgets")
duration_ms   wall time inside the span
current_tab   "changespecs" | "agents" | "axe" | null
```

…plus any per-call counters (`count`, `agents`, `panels`, `output_bytes`, …) and any global context fields seeded via `sase.ace.tui.util.trace.set_trace_context(...)` (the app pushes `current_tab` and `current_idx` automatically).

Point-in-time records emitted by `trace_event(...)` contain `event` instead of `span` / `duration_ms`. They are used for selection and highlight watcher transitions where there is no timed block to measure.

Timed spans currently wired (by file):

- `actions/changespec/_display.py` — `changespec.refresh_display`, `changespec.refresh_debounced`, `changespec.refresh_detail_only`
- `actions/changespec/_loading.py` — `changespec.filter`
- `actions/agents/_display.py` — `agents.refresh_display`, `agents.refresh_debounced`
- `actions/agents/_display_panels.py` — `agents.refresh_panel_widgets`, `agents.refresh_panel_highlights`
- `actions/agents/_loading_helpers.py` — `agents.load_from_disk`
- `actions/agents/_loading_live_hints.py` — `agents.live_hint_refresh`
- `widgets/changespec_list.py` — `widget.changespec_list.update_list`, `widget.changespec_list.update_highlight`, `widget.changespec_list.patch_changespec_row`
- `widgets/changespec_detail.py` — `widget.changespec_detail.update_display`
- `widgets/agent_list.py` — `widget.agent_list.update_list`, `widget.agent_list.update_highlight`, `widget.agent_list.patch_agent_row`, `widget.agent_list.try_remove_rows`
- `widgets/agent_detail.py` — `widget.agent_detail.update_display`, `widget.agent_detail.update_display_immediate`
- `widgets/ancestors_children_panel.py` — `widget.ancestors_children.update_relationships`, `widget.ancestors_children.update_relationships_from_index`
- `widgets/prompt_panel/_agent_display.py` — `widget.prompt_panel.update_display`, `widget.prompt_panel.update_header_only`
- `widgets/file_panel/__init__.py` — `widget.file_panel.update_display`
- `widgets/thinking_panel.py` — `widget.thinking_panel.update_display`

- `widgets/axe_dashboard.py` — `widget.axe_dashboard.update_display`

Spans nest cleanly: a single keypress that fires `agents.refresh_debounced` will record one outer span plus inner `widget.agent_list.update_highlight` and `agents.refresh_panel_highlights` spans.

ACE deliberately keeps live-workspace pencil hints off the startup-critical agents loader. The first load classifies only cheap persisted `diff_path` badges. After that agents list has applied, `agents.live_hint_refresh` runs VCS probes for active, non-terminal rows that do not yet have a persisted diff and patches changed rows in place. During startup investigations, treat `agents.load_from_disk` and `agents.live_hint_refresh` as separate costs: the former controls time to first interactive Agents-tab paint, while the latter explains deferred pencil-badge updates.

Trace events currently wired include:

- `selection.current_idx.set`
- `widget.changespec_list.watch_highlighted` and `.suppressed`
- `widget.agent_list.watch_highlighted` and `.suppressed`
- `widget.bgcmd_list.watch_highlighted` and `.suppressed`

17.2 Quick capture

```
SASE_TUI_TRACE=1 sase ace
# … exercise the path you care about (cold start, query change, j/k burst,
#   auto-refresh, large reply select) …
# Quit with q.

# Inspect:
jq -c 'select(.span | startswith("widget.agent_list."))' \
   ~/.sase/perf/tui_trace.jsonl | head -20
```

To inspect point events instead of timed spans:

```
jq -c 'select(.event)' ~/.sase/perf/tui_trace.jsonl | head -20
```

For key-to-paint timing during j/k navigation, also enable the separate perf recorder:

```
SASE_TUI_TRACE=1 SASE_TUI_PERF=1 sase ace
jq -c . ~/.sase/perf/tui_jk.jsonl | head -20
```

Override the key-to-paint path with `SASE_TUI_PERF_PATH=/tmp/tui_jk.jsonl`.

Agents that launch the TUI via `sase ace --tmux` get `SASE_TUI_TRACE=1` and `SASE_TUI_PERF=1` injected automatically; export the variable to `0` before invoking to opt out.

17.3 Synthetic-data benchmark harness

The harness lives at `tests/perf/bench_tui_trace.py`. It generates in-memory ChangeSpec and agent fixtures, then drives the TUI through Textual Pilot without touching real `~/.sase` data. It is marked `pytest.mark.slow`, so it does not run as part of `just test`.

Run via pytest:

```
pytest -s -m slow tests/perf/bench_tui_trace.py
```

Or as a script (writes a baseline numbers file the next phase can diff):

```
python -m tests.perf.bench_tui_trace --output ~/.sase/perf/tui_perf_baseline.json
```

The script also accepts explicit trace and key-to-paint output paths:

```
python -m tests.perf.bench_tui_trace \
  --output ~/.sase/perf/tui_perf_baseline.json \
  --trace-path ~/.sase/perf/tui_trace.jsonl \
  --perf-path ~/.sase/perf/tui_jk.jsonl
```

Fixture sizes:

```
ChangeSpecs: 100,  500, 2000   (tests/perf/fixtures.py: CHANGESPEC_SIZES)
Agents:       50,  200, 1000   (tests/perf/fixtures.py: AGENT_SIZES)
Large reply:   1,    5,   20 MB (LARGE_REPLY_SIZES_MB)
```

Scenarios per fixture size:

- cold start
- query change
- repeated query edits
- 50-key j/k burst
- auto-refresh with no changes
- large-reply select

The per-scenario summary records wall-clock times, then aggregates p50 / p95 / max for every trace span and key-to-paint action observed during that scenario.

17.4 Targets per phase gate

The targets below come from `sdd/research/202604/sase_perf_research.md` and are restated here so each phase agent has a single page to check against. A phase is green when the relevant targets are met **without regressing** any other span.

```
j/k highlight p95             < 16 ms
key-to-paint p95              < 33 ms
debounced detail paint        < 150–250 ms
warm ChangeSpec reload, 1k    < 100 ms
no-change auto-refresh stall  ~0 ms (event-driven path; Phase 7)
large reply first paint       immediate plain render, syntax later/optional
```

Per-phase responsibilities:

- **Phase 2** (ChangeSpec j/k hot path): `widget.changespec_list.update_list` call count drops to zero for j/k navigation; `update_highlight` p95 < 16 ms at 500 specs.

- **Phase 3** (data layer): warm ChangeSpec reload < 100 ms at 1k specs; `changespec.filter` p95 should drop materially after the snapshot cache and query context land.
- **Phase 4** (agent panel + list): `agents.refresh_panel_highlights` and `widget.agent_list.update_highlight` p95 < 16 ms at 1k agents.
- **Phase 5** (incremental loader): `agents.load_from_disk` near zero on a no-change auto-refresh.
- **Phase 6** (artifact + render caching): `widget.prompt_panel.update_display` / `widget.file_panel.update_display` immediate first paint on the largest reply fixture.
- **Phase 7** (event-driven auto-refresh): no-change auto-refresh shows no agents/changespec spans firing at all.

17.5 Adding a new span

```
from sase.ace.tui.util.trace import tui_trace

with tui_trace("module.name", count=len(items)):
    ...
```

Names use dotted lowercase. Counters should be ints / strs only — the emitter falls back to `str(...)` for unknown types via `default=str`, but keeping payloads JSON-friendly speeds downstream `jq` slicing.

When a span boundary forces a refactor (most existing hot paths split into `foo()` → `_foo_impl()` so the wrapping context manager doesn't fight indentation rules), keep both methods next to each other and let the public name stay the trace span name.

18 Telemetry

This document describes the Prometheus-based telemetry system in sase. Telemetry tracks metrics across all major subsystems — agent lifecycle, LLM providers, axe daemon, hooks, beads, VCS/workspace, and notifications — and provides CLI tools for monitoring, health checks, and a bundled Docker Compose monitoring stack.

18.1 Table of Contents

- Overview
- Configuration
- CLI Commands
- status
- list
- snapshot
- dashboard
- health
- export-config
- Architecture
- Metric Catalog
- Monitoring Stack
- Integration Points

18.2 Overview

The telemetry system uses Prometheus (https://prometheus.io/) metrics to instrument sase internals. Key design principles:

- **Lightweight when disabled**: All metrics are lightweight no-op stubs when `telemetry.enabled` is `false` (the default). Instrumentation calls still run, but they do not create Prometheus clients, open network connections, or start an exposition server.
- **Dual data collection**: Short-lived processes (agents) push metrics to a Push Gateway. Long-lived processes (axe daemon) expose metrics via an HTTP endpoint for Prometheus to scrape.
- **37 metrics across 8 subsystems**: Comprehensive coverage of the full sase lifecycle.

Telemetry data appears only after telemetry is enabled and an instrumented process has run. For local dashboards, export the monitoring stack first, start it with Docker Compose, then run or restart the relevant sase processes so they initialize telemetry with the enabled config.

18.3 Configuration

Telemetry is configured under the `telemetry` key in `sase.yml`:

```yaml
telemetry:
  enabled: false # Toggle telemetry on/off globally
  prometheus:
    exposition_port: 9464 # HTTP server port for the axe exposition server
    pushgateway_url: "localhost:9091" # Push Gateway address
    url: "localhost:9090" # Prometheus API address for charts mode
  health_thresholds:
    error_rate_warn: 10.0 # % threshold for warn status
    error_rate_critical: 25.0 # % threshold for critical status
    retry_rate_warn: 10.0
    retry_rate_critical: 25.0
    p95_latency_warn: 300.0 # seconds
    p95_latency_critical: 600.0
```

Field	Type	Default	Description
`telemetry.enabled`	bool	`false`	Enable or disable telemetry globally.
`telemetry.prometheus.exposition_port`	int	`9464`	HTTP server port for the axe metric exposition endpoint.
`telemetry.prometheus.pushgateway_url`	str	`localhost:9091`	Push Gateway address used by short-lived runner processes.
`telemetry.prometheus.url`	str	`localhost:9090`	Prometheus API address used by dashboard charts mode.
`telemetry.health_thresholds.*_warn`	float	varies	Percentage threshold for WARN health status.
`telemetry.health_thresholds.*_critical`	float	varies	Percentage threshold for CRITICAL status.
`telemetry.health_thresholds.p95_latency_*`	float	300/600	P95 latency thresholds in seconds.

Source: `src/sase/default_config.yml`, `src/sase/telemetry/_config.py`

18.4 CLI Commands

With no subcommand, `sase telemetry` defaults to `sase telemetry list` with default options. Use the explicit `sase telemetry list` form when passing metric-catalog filters.

18.4.1 `sase telemetry status`

Quick health check and configuration display. Shows telemetry enabled/disabled status, metric counts by type, and reachability of the Push Gateway and HTTP exposition server.

```
sase telemetry status
```

When telemetry is disabled, this command prints the config snippet needed to enable it and does not probe metric endpoints.

18.4.2 `sase telemetry list`

Display the metric catalog from internal definitions, grouped by subsystem.

```
sase telemetry list                     # All metrics
sase telemetry list -s "Agent Lifecycle" # Filter by subsystem
sase telemetry list -t counter          # Filter by type (counter, gauge, histogram)
```

Flag	Values	Default	Description
`-s, --subsystem`	subsystem name	-	Filter to a subsystem
`-t, --type`	`counter`, `gauge`, `histogram`	-	Filter by metric type

18.4.3 `sase telemetry snapshot`

Fetch and display current metric values from the Push Gateway or exposition endpoint.

```
sase telemetry snapshot                         # Rich table (default)
sase telemetry snapshot -f json                 # JSON output
sase telemetry snapshot -f prometheus           # Raw Prometheus text format
sase telemetry snapshot -S pushgateway          # Force pushgateway source
```

Flag	Values	Default	Description
`-S, --source`	`auto`, `pushgateway`, `exposition`	`auto`	Data source
`-f, --format`	`rich`, `json`, `prometheus`	`rich`	Output format
`-s, --subsystem`	subsystem name	-	Filter by subsystem

18.4.4 `sase telemetry dashboard`

Live auto-refreshing TUI dashboard with two modes: summary (default) and charts.

```
sase telemetry dashboard                # Summary mode, 5s refresh
sase telemetry dashboard -c             # Charts mode with historical data
sase telemetry dashboard -c -r 24h      # Charts over last 24 hours
sase telemetry dashboard -c -s "LLM Provider"  # Focus on one subsystem
sase telemetry dashboard -i 10          # 10-second refresh interval
```

Flag	Values	Default	Description
`-i, --interval`	int (seconds)	`5`	Refresh interval
`-S, --source`	`auto`, `pushgateway`, `exposition`	`auto`	Data source
`-c, --charts`	flag	-	Enable charts mode with historical data
`-r, --range`	`1h`, `6h`, `24h`, `7d`	`1h`	Time range for charts
`-s, --subsystem`	subsystem name	-	Focus on a single subsystem (larger charts)

Summary mode shows styled stat panels with color-coded gauges (Active Agents, Active Workspaces, Active Beads) and compact subsystem metric tables.

Charts mode (`-c`) renders historical line and bar charts via Prometheus range queries, showing: Agent Run Duration, Active Agents, Agent Throughput, LLM Token Usage, LLM Latency, and Error Rate. Falls back to summary mode when Prometheus is unreachable.

The `--source` flag selects the Push Gateway or exposition endpoint for summary mode. Charts mode uses the Prometheus HTTP API from `telemetry.prometheus.url` because it needs historical range queries.

18.4.5 `sase telemetry health`

Traffic-light health assessment with OK/WARN/CRITICAL status for subsystems that have scraped data. Configured thresholds apply to agent and LLM error rates plus hook retry rates; other entries are informational or use fixed heuristics.

```
sase telemetry health                    # Rich output
sase telemetry health -j                 # Machine-readable JSON
```

Flag	Values	Default	Description
`-j, --json`	flag	-	JSON output
`-S, --source`	`auto`, `pushgateway`, `exposition`	`auto`	Data source

Exit codes: `0` (healthy), `1` (degraded/warn), `2` (critical).

18.4.6 `sase telemetry export-config`

Export the bundled monitoring stack (Docker Compose, Prometheus config, Grafana dashboard) to a local directory.

```
sase telemetry export-config                    # Default: ./sase-monitoring/
sase telemetry export-config -o /tmp/monitoring  # Custom output directory
sase telemetry export-config -f                 # Overwrite existing
```

Flag	Values	Default	Description
`-o, --output-dir`	path	`./sase-monitoring/`	Target directory
`-f, --force`	flag	-	Overwrite target if it exists

After exporting, start the stack with:

```
cd sase-monitoring && docker compose up -d
```

18.5 Architecture

```
┌──────────────────────────────────────────────────────────┐
│                    sase telemetry CLI                    │
│  status · list · snapshot · dashboard · health · export  │
├──────────────────────────────────────────────────────────┤
│                      Scrape Client                       │
│          (fetches from Push Gateway or exposition)       │
├──────────────────────┬───────────────────────────────────┤
│   Push Gateway       │   HTTP Exposition Server          │
│ (short-lived procs)  │   (long-lived: axe daemon)        │
├──────────────────────┴───────────────────────────────────┤
│                 Prometheus Metrics Layer                 │
│       metrics.py (37 metric singletons, stub/real switch)│
├──────────────────────────────────────────────────────────┤
│                 Instrumentation Points                   │
│       Agent · LLM · Axe · Hooks · Beads · VCS · Notify · Memory │
└──────────────────────────────────────────────────────────┘
```

18.5.1 Source Layout

File / Directory	Purpose
`src/sase/telemetry/__init__.py`	Public API exports
`src/sase/telemetry/metrics.py`	Module-level metric singletons (37 attrs)
`src/sase/telemetry/_registry.py`	Init, Push Gateway integration, atexit
`src/sase/telemetry/_stubs.py`	No-op stub classes used when telemetry is off
`src/sase/telemetry/_config.py`	Configuration loading from sase.yml
`src/sase/telemetry/catalog.py`	Structured metric catalog and grouping
`src/sase/telemetry/scrape.py`	HTTP client and Prometheus text parser
`src/sase/telemetry/prom_query.py`	Prometheus HTTP API client (range queries)
`src/sase/telemetry/charts.py`	Terminal chart rendering via plotext
`src/sase/telemetry/cli_*.py`	CLI subcommand handlers
`src/sase/telemetry/monitoring/`	Bundled Docker Compose + Prometheus + Grafana

18.6 Metric Catalog

37 metrics organized into 8 subsystems:

18.6.1 Agent Lifecycle

Prometheus Name	Type	Labels	Description
`sase_agent_runs_total`	counter	llm_provider, status, workflow	Total agent runs
`sase_agent_run_duration_seconds`	histogram	llm_provider, workflow	Agent run duration
`sase_agent_active`	gauge	llm_provider, project	Currently active agents
`sase_agent_spawns_total`	counter	llm_provider, project	Total agent spawns
`sase_agent_kills_total`	counter	reason	Total agent kills

18.6.2 LLM Provider

Prometheus Name	Type	Labels	Description
`sase_llm_invocations_total`	counter	provider, status	Total LLM invocations
`sase_llm_invocation_duration_seconds`	histogram	provider	Invocation duration
`sase_llm_errors_total`	counter	provider, error_type	LLM errors
`sase_llm_retries_total`	counter	provider	LLM retries
`sase_llm_retry_spawns_total`	counter	outcome	Cross-process retries
`sase_llm_input_tokens_total`	counter	provider	Input tokens consumed
`sase_llm_output_tokens_total`	counter	provider	Output tokens generated
`sase_llm_cache_read_tokens_total`	counter	provider	Cache-read tokens

18.6.3 Axe Orchestrator

Prometheus Name	Type	Labels	Description
`sase_axe_cycles_total`	counter	cycle_type	Total axe cycles
`sase_axe_cycle_duration_seconds`	histogram	cycle_type	Cycle duration
`sase_axe_lumberjacks_active`	gauge	-	Active lumberjacks
`sase_axe_lumberjack_restarts_total`	counter	-	Lumberjack restarts
`sase_axe_errors_total`	counter	error_type	Axe errors

18.6.4 Hooks / Mentors / Workflows

Prometheus Name	Type	Labels	Description
`sase_hook_executions_total`	counter	hook_type, status	Hook executions
`sase_hook_duration_seconds`	histogram	hook_type	Hook duration
`sase_hook_retries_total`	counter	hook_type	Hook retries
`sase_mentor_executions_total`	counter	status	Mentor executions
`sase_workflow_executions_total`	counter	workflow, status	Workflow executions
`sase_workflow_duration_seconds`	histogram	workflow	Workflow duration
`sase_zombie_detections_total`	counter	-	Zombie process detections

18.6.5 Beads

Prometheus Name	Type	Labels	Description
`sase_bead_operations_total`	counter	operation	Bead CRUD operations
`sase_bead_status_transitions_total`	counter	from_status, to_status	Bead status transitions

`sase_bead_active`	gauge	project, status	Active beads

18.6.6 VCS / Workspace

Prometheus Name	Type	Labels	Description
`sase_vcs_commits_total`	counter	provider, type	VCS commits
`sase_vcs_operations_total`	counter	provider, operation, status	VCS operations
`sase_workspace_acquisitions_total`	counter	project	Workspace acquisitions
`sase_workspace_releases_total`	counter	project	Workspace releases
`sase_workspace_active`	gauge	project	Active workspaces

18.6.7 Notifications

Prometheus Name	Type	Labels	Description
`sase_notifications_sent_total`	counter	type, status	Notifications sent

18.6.8 Memory

Prometheus Name	Type	Labels	Description
`sase_memory_proposals_proposed_total`	counter	-	Memory proposals created
`sase_memory_proposals_approved_total`	counter	edited	Memory proposals approved
`sase_memory_proposals_rejected_total`	counter	-	Memory proposals rejected

18.7 Monitoring Stack

The `sase telemetry export-config` command exports a ready-to-use Docker Compose stack:

18.7.1 Services

Service	Port	Description
Push Gateway	9091	Receives metrics from short-lived agent processes
Prometheus	9090	Scrapes Push Gateway and axe exposition server; stores metrics
Grafana	3000	Pre-configured dashboard with panels for all subsystems

18.7.2 Prometheus Scrape Configuration

Prometheus is configured with two scrape jobs:

- **`sase_axe`**: Scrapes the axe daemon's HTTP exposition server (port 9464)
- **`pushgateway`**: Scrapes the Push Gateway (port 9091)

Global scrape interval: 15 seconds.

18.7.3 Alerting Rules

14 alert rules covering:

- Agent error rate thresholds (10%, 25%)
- Agent p95 latency thresholds (300s, 600s)
- LLM error rate and retry rate
- Axe cycle errors
- Hook retry rates
- Zombie process detection
- Push Gateway reachability

Alert labels use `severity: warning` and `severity: critical`.

18.7.4 Grafana Dashboard

A pre-built Grafana dashboard is automatically provisioned with panels for all telemetry subsystems, accessible at `http://localhost:3000` after starting the stack.

18.8 Integration Points

Telemetry is instrumented across the codebase:

Subsystem	Location	What is tracked
Agent	Agent runner setup/finalize modules under `src/sase/axe/`	Runs, duration, spawns, kills
LLM Provider	`src/sase/llm_provider/_invoke.py`	Invocations, tokens, errors, retries
Axe	`src/sase/axe/orchestrator.py`, `src/sase/axe/lumberjack.py`	Cycles, errors, lumberjack activity
Hooks/Mentors	`src/sase/ace/hooks/execution.py`, runner modules under `src/sase/axe/`	Execution counts, durations, retries
Beads	`src/sase/bead/project.py`, `src/sase/main/bead_fast_path.py`	CRUD operations, status transitions
VCS	VCS provider plugins under `src/sase/vcs_provider/plugins/`	Commits, operations
Notifications	`src/sase/notifications/senders.py`	Notifications sent
Memory	`src/sase/memory/proposals.py`	Proposal review lifecycle

The axe orchestrator calls `init_telemetry(start_http_server=True)` on startup to begin exposing metrics. Agent processes use `push_metrics()` to send data to the Push Gateway on exit via an `atexit` handler.

Agent Revival Audit Log

The TUI's revive flow (`R` on the Agents tab) writes a JSONL audit log to `~/.sase/logs/events.jsonl` so post-mortem questions like "what was the last agent I tried to revive?" can be answered without grepping through dismissed-bundle directories. Paths shown here use the default SASE state root; when `SASE_HOME` is set, the log lives under that root instead.

Three event kinds are emitted:

event	meaning
`agent_revive_started`	User confirmed a revival in the modal. One record per invocation (single or batch); captured before any state mutation so a mid-flight crash still leaves a trace.
`agent_revived`	A single agent was revived successfully. Batch revivals produce one record per agent.
`agent_revive_failed`	Either the "no dismissed agents" early-out fired, or an exception was raised mid-flow. Batch revivals produce one failure record per agent that failed.

Record schema

Common fields on every record:

- `timestamp` — local time as `YYmmdd_HHMMSS`.
- `event` — one of the three names above.
- `batch_size` — `1` for a single revival; the count of selected agents for a batch.
- `agents` — only on `agent_revive_started`; a list of `[agent_type, cl_name, raw_suffix]` identities that were selected before mutation began.
- `outcome` — `"success"` or `"failure"` (omitted for `agent_revive_started`).
- `selection_scope` — `"all"`, `"home"`, `"project"`, or `"cl"`, plus `selection_project` / `selection_cl` when applicable. These fields are omitted when the revive was not launched from a scoped selection.
- `saved_group_id` / `saved_group_title` — present when the revival was launched from the saved-group modal rather than the custom scoped search.

Per-agent fields on `agent_revived` / `agent_revive_failed`:

- `agent_identity` — `[agent_type, cl_name, raw_suffix]`.
- `agent_type`, `cl_name`, `raw_suffix`, `project_file`.
- `bundle_path` — the path under `~/.sase/dismissed_bundles/YYYYMM/` that the revive flow loaded. Revive preserves the bundle as historical recovery data.
- `child_suffixes` — only on `agent_revived`; workflow steps / follow-up agents revived alongside the parent (sorted).

Failure-only fields:

- `stage` — one of `dismissed_set_update`, `artifact_restore`, `bundle_marking`, `reload`, `refresh_display`, `saved_group_load`, `saved_group_bundle_load`, `saved_group_mark_revived`, or `no_dismissed_agents`.
- `error_type` / `error_message` — exception class and `str(exc)` truncated to 500 characters.
- `reason` — set for non-exceptional failures (currently only `no_dismissed_agents`).

CLI

```
sase revive-log              # last 20 records, rich table
sase revive-log --all        # every record
sase revive-log --since -7d  # records on/after the daterange start
sase revive-log --outcome failure
sase revive-log --json       # one JSON object per line for agent consumption
```

`--since` accepts the same daterange grammar as `sase logs`.

Backfill

There is no recoverable audit history before this feature shipped. Existing dismissed bundles remain usable, but the first revival after the feature lands is the earliest structured event.

Editor Integration

SASE exposes two editor-facing surfaces for prompt and xprompt editing:

- `sase lsp` is the interactive editor path. Configure your editor to launch it as a stdio language server when you want completions, snippets, hover, diagnostics, code actions, and jump-to-definition while editing a prompt.
- `sase editor helper-bridge ...` is the integration/debugging path. It reads one JSON request from stdin and writes one JSON response to stdout, so clients can fetch the same catalogs without implementing an LSP client.

Language Server

`sase lsp` starts the SASE xprompt language server over stdio. Run `--version` or `--help` in a terminal to verify the wrapper, then point your editor's LSP configuration at `sase lsp`:

```
sase lsp --version
sase lsp --help
```

In editor configuration terms, the command is `sase` and the argument list is `["lsp"]`.

The wrapper resolves the server command in this order:

1. `SASE_XPROMPT_LSP_CMD`, parsed as a shell-style command for development.
2. `sase-xprompt-lsp` on `PATH`.
3. A debug or release `sase-xprompt-lsp` binary under a sibling `../sase-core` checkout.
4. `cargo run --manifest-path ../sase-core/Cargo.toml -p sase_xprompt_lsp --` when `cargo` is available and the sibling checkout has a `Cargo.toml`.

Use `SASE_XPROMPT_LSP_CMD` when you need to point the editor wrapper at a different source checkout or command. The `Justfile` uses `SASE_CORE_DIR` and `SASE_LINKED_REPO_SASE_CORE_DIR` (with the legacy `SASE_SIBLING_REPO_*` variables as fallbacks) for local `sase-core` build/install targets, but `sase lsp` itself does not read those variables.

The wrapper also exports installed package xprompt locations, bundled default config, plugin xprompt directories, and plugin config paths to the Rust server. The server refreshes its catalog when the LSP session starts, keeps a short cache for completion requests, and exposes a `sase.xpromptLsp.refreshCatalog` command for clients that surface LSP commands.

LSP Features

The xprompt language server is focused on prompt and xprompt editing:

Feature	Behavior
XPrompt completion	Completes `#name`, `#!workflow`, namespaced references, and slash-skill references from the structured catalog.
Project/ChangeSpec tags	Completes `#+query` and first-character `+query` from the active project/ChangeSpec catalog, inserting canonical VCS workspace tags.
Argument assistance	Completes named arguments, path inputs, and bool values for typed xprompt inputs where the catalog exposes input metadata.
Directive completion	Completes SASE prompt directives such as `%model`, `%wait`, and other known directive names.
File completion	Completes path-like tokens and recent file-history entries for prompt references.
Snippets	Offers SASE snippets after bare trigger words when the client advertises LSP snippet support.
Hover	Shows xprompt metadata, descriptions, previews, source display paths, tags, and active input hints.
Diagnostics	Reports unknown xprompts, unknown slash skills, unknown directives, malformed xprompt arguments, and argument type/arity issues.

Definition	Jumps from xprompt and slash-skill references to real source files when the catalog provides a resolvable path.

Snippet completions come from the same registry ACE uses: xprompts with `snippet` front matter plus user-defined `ace.snippets`, with `ace.snippets` winning on trigger collisions. The server asks the host helper bridge for that authoritative registry and falls back to native Rust loading only for simple xprompt snippets and configured `ace.snippets` when the helper is unavailable.

Helper Bridge

Editor integrations that do not need live LSP behavior can call fixed helper operations directly:

```
printf '{"schema_version":1,"project":"sase"}\n' | sase editor helper-bridge xprompt-catalog
printf '{"schema_version":1,"project":"sase"}\n' | sase editor helper-bridge snippet-catalog
```

`xprompt-catalog` returns the structured xprompt catalog used by mobile/editor clients, including insertion text, reference prefix, kind, tags, typed inputs, display/source fields, and `definition_path` when SASE can resolve a real file.

`snippet-catalog` returns the composed snippet registry:

- XPrompt-derived snippets from markdown files with `snippet` front matter.
- User snippets from `ace.snippets` in merged SASE config.
- Valid trigger words only; user snippets override xprompt snippets on collision.

Both helper operations read one JSON object from stdin and write one compact JSON object to stdout. They are fixed catalog operations, not a general shell or filesystem bridge.

Authoring Snippets

Use `ace.snippets` for local trigger-word templates:

```
ace:
  snippets:
    fix: "Please fix the following issue:\n$0"
    review: "Review this code for correctness, performance, and style.\n$0"
```

Use xprompt front matter when a reusable prompt should also appear as a snippet:

```
---
name: review
snippet: true
input:
  path: path
---

Review {{ path }} for correctness, tests, and maintainability.
```

Required xprompt inputs become snippet tabstops. Optional inputs are pre-filled from defaults. XPrompts with complex Jinja control flow are skipped by snippet conversion so the generated editor template stays predictable.

Troubleshooting

Symptom	Check
`sase lsp` cannot start	Run `sase lsp --version`; install `sase-xprompt-lsp`, build `../sase-core`, or set `SASE_XPROMPT_LSP_CMD`.
Snippets do not appear	Confirm the editor advertises LSP `completionItem.snippetSupport`; inspect `sase editor helper-bridge snippet-catalog`.
Completion catalog looks stale	Restart the LSP session after changing installed plugin resources or package xprompts.
Jump-to-definition is missing	Check whether the catalog entry has a real `definition_path`; plugin or built-in virtual entries may only have display paths.
A user snippet is ignored	Trigger names must contain only ASCII letters, digits, or `_`.

Related Pages

- XPrompt reference (https://sase.sh/xprompt/) for xprompt syntax, discovery order, typed inputs, snippets, and workflows.
- Integration APIs (https://sase.sh/integrations/#editor-helper-bridge) for the Python helper facade.
- Configuration (https://sase.sh/configuration/#sase-editor) for CLI flag and environment-variable reference.
- ACE snippets (https://sase.sh/ace/#snippets) for the in-TUI prompt widget behavior.

19 Integration APIs

SASE exposes a small set of Python helpers for external plugins, chat clients, mobile clients, and editor integrations. These APIs live under `sase.integrations` when they are meant for integration-facing use, or under the subsystem package when they are already part of an existing provider contract. Externally consumed public symbols use `# pyvision: <repo-uri>` pragmas so unused-code tooling validates them against the tracked files of the consuming repository.

For editor setup and user-facing behavior, start with the editor integration guide (https://sase.sh/editor/). This page focuses on the integration-facing Python and bridge contracts.

19.1 ChangeSpec XPrompt Tags

`sase.integrations.changespec_tags.list_changespec_xprompt_tags()` returns copyable VCS xprompt references for active ChangeSpecs. A ChangeSpec is SASE's stored record for a change list or pull request, and an xprompt tag is the `#workflow:target` reference that launches an agent in that workspace. This helper is intended for plugins and editors that need to show a picker of targets such as `#gh:my_change` or `#git:local_branch`.

```python
from sase.integrations.changespec_tags import list_changespec_xprompt_tags

listing = list_changespec_xprompt_tags(project="sase")
for entry in listing.entries:
    print(entry.project, entry.name, entry.status, entry.workflow_type, entry.tag)

if listing.skipped:
    print("Some ChangeSpecs could not be tagged:", listing.skipped)
```

The optional `project` argument is an exact project-name filter. Terminal ChangeSpecs are excluded after normalizing workspace/status suffixes, so `Submitted`, `Archived`, and `Reverted` entries are not returned. Results are sorted deterministically by project, ChangeSpec name, and normalized status.

The helper uses the same active-project default as normal ChangeSpec discovery. Inactive projects are omitted from the broad list. The mobile helper bridge wraps explicit inactive-project tag requests with a partial-success warning telling the caller to reactivate the project before launching new work.

Each returned `ChangeSpecTagEntry` has:

Field	Description
`project`	Parsed project basename
`name`	ChangeSpec `NAME`
`status`	Normalized non-terminal status
`workflow_type`	Detected workspace workflow type, such as `git`, `gh`, or `hg`
`tag`	Copyable xprompt target in `#{workflow_type}:{name}` form

If workspace workflow detection fails for an entry, that ChangeSpec is omitted and a human-readable message is appended to `ChangeSpecTagListing.skipped`. This lets callers still show the rest of the list while surfacing degraded entries.

Source: `src/sase/integrations/changespec_tags.py`

19.2 Agent Status Groups

`sase.integrations.agent_status_groups` exposes the same status-bucketing semantics used by the ACE Agents tab for external chat or editor surfaces that want a compact running-agent summary.

```python
from sase.agent.running import list_all_agents
from sase.integrations.agent_status_groups import group_agent_statuses, status_bucket_header

for group in group_agent_statuses(list_all_agents()):
    print(status_bucket_header(group.bucket, len(group.agents)))
    for agent in group.agents:
        print("  ", agent.name, agent.status)
```

Buckets are emitted in ACE display order and empty buckets are omitted:

Each returned `AgentStatusGroup` contains the bucket label and the running-agent records assigned to that bucket.

Bucket	Meaning
`Stopped`	User-facing blockers such as `PLAN` and `QUESTION`.
`Failed`	Terminal failure statuses (`FAILED...`).
`Running`	Active execution, including `PLAN APPROVED` and unrecognized actives.
`Waiting`	`WAITING` agents with timer/dependency progress.
`Done`	Terminal success/plan handoff states.

Source: `src/sase/integrations/agent_status_groups.py`, `src/sase/agent/status_buckets.py`

19.3 Mobile Notification Bridge

`sase.integrations.mobile_notifications` is the stable host-side facade used by the Rust mobile gateway to expose the local notification inbox to mobile clients. External callers should import from this facade only; the `sase.integrations._mobile_notification_*` modules hold the split implementation and are internal.

```python
from sase.integrations.mobile_notifications import (
    build_mobile_attachment_manifests,
    execute_mobile_hitl_action,
    execute_mobile_plan_action,
    execute_mobile_question_action,
    read_mobile_notification_snapshot,
    resolve_mobile_notification_detail,
)

snapshot = read_mobile_notification_snapshot(unread_only=True, limit=25)
if snapshot.rows:
    detail = resolve_mobile_notification_detail(snapshot.rows[0].id)
else:
    detail = None

if detail:
    attachments = build_mobile_attachment_manifests(detail)
    print(detail.action, detail.action_state, [item.display_name for item in attachments])

result = execute_mobile_plan_action("abcdef12", "approve", commit_plan=True, run_coder=False)
print(result.notification_id, result.response_file)

hitl_result = execute_mobile_hitl_action("12345678", "continue")
question_result = execute_mobile_question_action("87654321", "answer", custom_answer="Use the default.")
print(hitl_result.message, question_result.message)
```

Snapshot reads project notifications into mobile-safe rows with display paths, host paths, action state, read/dismissed state, mute/snooze state, and priority counts. Detail reads include dismissed and silent rows so clients can rebuild local state after an event-stream resync. Action helpers resolve exact IDs or unique prefixes, write

the corresponding response JSON once, and run best-effort host side effects. The facade supports plan approvals, workflow human-in-the-loop actions, and user-question answers. Action failures raise `MobilePlanActionError` with deterministic `code` and `target` fields for duplicate, stale, ambiguous, unsupported, missing, and invalid requests.

Source: `src/sase/integrations/mobile_notifications.py`

19.4 Mobile Agent And Helper Bridges

`sase.integrations.mobile_agents` and `sase.integrations.mobile_helpers` are stable facades for the workstation-hosted mobile gateway bridge commands. The Rust gateway invokes them through fixed JSON-over-stdin operations rather than exposing a generic shell, cwd, environment, or filesystem API to mobile clients.

Agent bridge operations cover `list-agents`, `resume-options`, `launch-text`, `launch-image`, `kill-agent`, and `retry-agent`. Launch requests may name a known SASE project or use normal SASE prompt refs for VCS context; Android and other mobile clients must not send host paths. Image launches store uploads under SASE-owned gateway state, then inject the saved path into the agent prompt. Launch, kill, retry, upload, and per-device project context metadata lives under `<sase_home>/mobile_gateway/`.

Helper bridge operations cover `changespec-tags`, `xprompt-catalog`, `beads-list`, `beads-show`, `update-start`, and `update-status`. ChangeSpec, xprompt, and bead helpers are read-only. The only mutating helper operation is `update-start`, which starts the configured `chat_install.command` worker and reports status through structured polling. Bead helper reads use one canonical bead store per project, typically the project's current checkout at `sdd/beads/`, where `events/**` is canonical and `issues.jsonl` is a compatibility projection; they do not merge numbered sibling workspaces or legacy bead stores. The structured xprompt catalog includes `definition_path` when the source can be resolved to a real file, so mobile and editor clients can offer jump-to-definition without reverse-engineering display paths.

All-known helper reads are lifecycle-aware and enumerate active projects by default. Inactive projects are left out of broad ChangeSpec tag, xprompt catalog, and bead lists. Explicit ChangeSpec tag and xprompt catalog requests for an inactive project report warnings in the structured `result.warnings` / `result.skipped` fields where the bridge can still return a partial result. Explicit bead requests resolve the requested project's canonical bead store directly; the lifecycle filter only applies to the all-known bead list.

Bridge commands read a JSON object from stdin and write a compact JSON object to stdout:

```
printf '{"schema_version":1,"project":"sase","limit":20}\n' | sase mobile helper-bridge changespec-tags
printf '{"schema_version":1,"project":"sase","limit":20}\n' | sase mobile agent-bridge list-agents
```

External callers should import from these facade modules only. The `_mobile_agent_*` and `_mobile_helper_*` modules are private split implementations kept small for testability and should not be imported by plugins or clients. The public HTTP route contract is documented in `docs/mobile_gateway.md` (https://sase.sh/mobile_gateway/).

Source: `src/sase/integrations/mobile_agents.py`, `src/sase/integrations/mobile_helpers.py`

19.5 Editor Helper Bridge

`sase.integrations.editor_helpers` exposes an editor-branded helper bridge over the same fixed JSON operations used by the mobile helper facade. The current CLI surface is:

```
printf '{"schema_version":1,"project":"sase"}\n' | sase editor helper-bridge xprompt-catalog
printf '{"schema_version":1,"project":"sase"}\n' | sase editor helper-bridge snippet-catalog
```

The `xprompt-catalog` operation returns the structured xprompt catalog, including insertion metadata, typed inputs, source display fields, and `definition_path` for entries backed by a resolvable file. The `snippet-catalog` operation returns the composed ACE snippet registry from xprompt snippets plus user snippets configured under `ace.snippets`. Editor integrations should use this bridge or `sase lsp` instead of importing private catalog modules directly.

Source: `src/sase/integrations/editor_helpers.py`, `src/sase/integrations/xprompt_lsp.py`

19.6 Chat Update Worker

Chat integrations that need to update a SASE install can call `sase.integrations.chat_install.start_chat_install_worker()`. The helper starts a detached worker process and returns a chat-safe result object instead of blocking the chat request on the full update. Poll with `read_chat_install_status()` when `start_chat_install_worker()` returns a `job_id`.

```python
from sase.integrations.chat_install import read_chat_install_status, start_chat_install_worker

result = start_chat_install_worker()
print(result.status, result.message)
if result.log_path:
    print(result.log_path)

if result.job_id:
    status = read_chat_install_status(result.job_id)
    print(status.status, status.message)
```

The worker sequence is:

1. Acquire `~/.sase/chat_install/install.lock`; if another worker owns it, return `already_running`.
2. Resolve the registered primary workspace for the `sase` project.
3. Stop axe.
4. Optionally sync the workspace through the selected VCS provider.
5. Run `chat_install.command` from that workspace with `chat_install.timeout_seconds`.
6. Restart axe, retrying up to `chat_install.restart_attempts`.

`start_chat_install_worker()` returns `ChatInstallLaunchResult` with one of these launch statuses: `config_missing_command`, `workspace_resolution_failed`, `already_running`, `launched`, or `launch_failed`. `read_chat_install_status()` returns `running`, `succeeded`, `failed`, or `not_found`. Worker logs live under `~/.sase/chat_install/logs/`. Configuration fields are documented in `docs/configuration.md` (https://sase.sh/configuration/#chat_install). The API, config key, and state paths keep the `chat_install` name for compatibility, but chat integrations should present this workflow to users as an update.

Source: `src/sase/integrations/chat_install.py`

20 Plugin System

Sase uses Python entry points (https://packaging.python.org/en/latest/specifications/entry-points/) to discover optional functionality installed in the same Python environment as `sase`. Runtime providers use pluggy (https://pluggy.readthedocs.io/) hooks; resource plugins expose package data such as xprompt files and `default_config.yml`.

The core `sase` package provides the plugin infrastructure, the built-in LLM providers, and local git/directory workspace support. Extra packages add hosted VCS workflows, internal workflows, or integrations without changing the core package.

20.1 Plugin Groups

Sase defines six entry point groups:

Entry Point Group	Entry Point Value	Purpose	Example Plugin
`sase_vcs`	Provider class	VCS provider plugins (git, hg, etc.)	`sase-github`
`sase_workspace`	Provider class	Workspace provider plugins (ref resolution, submit)	`sase-github`
`sase_llm`	Provider class	LLM provider plugins	built-in or third-party
`sase_xprompts`	Package module	XPrompt templates and workflows	`my_sase_plugin`
`sase_config`	Package module	Default configuration (`default_config.yml`)	`sase-github`, `my_sase_plugin`
`sase_plugin_manifest`	Package module	Plugin metadata resource used by diagnostics	third-party plugin packages

Provider-class entry points resolve to a class that is instantiated and registered with pluggy. Package-module entry points resolve to a module whose package resources are read by Sase.

20.2 Available Plugin Packages

Package	Description	Entry Points
`sase` (core)	Bare-git VCS, bare-git and `#cd` workspaces, and built-in LLM providers	`sase_vcs: bare_git`, `sase_workspace: bare_git, cd`, `sase_llm: agy, claude, codex, opencode, qwen`
`sase-github`	GitHub VCS and workspace support, including GitHub CLI (`gh`) PR operations	`sase_vcs: github`, `sase_workspace: github`, `sase_config: sase_github`, `sase_xprompts: sase_github`
`sase-telegram`	Telegram integration via chop scripts (`sase_chop_tg_outbound`, `sase_chop_tg_inbound`)	CLI scripts (not pluggy entry points)
`sase-nvim`	Neovim integration, including project spec syntax and prompt helpers	standalone Neovim plugin files (not Python entry points)

20.3 Installation

```
# Core sase (includes BareGitPlugin for plain git repos)
pip install sase

# Add GitHub PR support
pip install sase-github
```

20.4 CLI Diagnostics

`sase plugin` defaults to `sase plugin list`.

```
sase plugin
sase plugin list --verbose
sase plugin doctor
sase plugin doctor --json
```

`sase plugin list` inventories installed SASE entry points, plugin distributions, configured axe chop scripts, and available unconfigured chop scripts. `sase plugin doctor` runs the same inventory plus health checks for resource entry point load failures, missing configured chops, unconfigured scripts, GitHub CLI/auth prerequisites when GitHub plugins are installed, and Telegram `pass`/environment prerequisites when Telegram chop scripts are present. The doctor status is `ERROR` for resource entry point load failures or missing configured script chops. Unconfigured available scripts and optional integration prerequisites report `WARN`. Use the explicit `list` or `doctor` subcommand when passing flags; `sase plugin --verbose` and `sase plugin --json` are not valid forms.

20.5 How Plugins Are Discovered

Plugin discovery uses `importlib.metadata.entry_points()` to find installed packages that declare one of Sase's entry point groups.

There are two discovery paths:

1. **Provider classes**: `sase_vcs`, `sase_workspace`, and `sase_llm` entry points resolve to classes. The relevant registry loads the class, instantiates it, and registers the instance with a pluggy `PluginManager`.
2. **Package resources**: `sase_xprompts`, `sase_config`, and `sase_plugin_manifest` entry points resolve to modules. The shared helper in `src/sase/main/plugin_discovery.py` sorts config and xprompt entry points by name, loads the modules, and skips module load failures after logging them at debug level. `sase plugin doctor` loads resource entry points directly so packaging problems are visible as diagnostics instead of only debug logs.

20.5.1 VCS Plugins (pluggy)

VCS plugins use pluggy's hook system. The hook specification is defined in `VCSHookSpec` (`src/sase/vcs_provider/_hookspec.py`). Each hook method uses `firstresult=True`, meaning the first plugin to return a non-`None` result wins.

The VCS registry (`src/sase/vcs_provider/_registry.py`) uses `sase_vcs` entry points in two ways:

1. Detection/classification builds a pluggy manager containing all registered VCS plugins.
2. Runtime operations create a `VCSPluginManager` for the selected provider name, such as `bare_git`, `github`, or `hg`.

20.5.2 Workspace Plugins (pluggy)

Workspace plugins use pluggy's hook system, similar to VCS plugins. The hook specification is defined in `WorkspaceHookSpec` (`src/sase/workspace_provider/_hookspec.py`). Most hooks use `firstresult=True`; the exception is `ws_get_workflow_metadata` which collects results from all plugins. All hook method names are prefixed with `ws_`.

The workspace registry (`src/sase/workspace_provider/_registry.py`) creates a singleton `WorkspacePluginManager` , registers `WorkspaceHookSpec` , and loads all `sase_workspace` provider classes from entry points. This is why all workspace metadata can be listed at once while hook dispatch still lets a single plugin handle each operation.

See docs/workspace.md (https://sase.sh/workspace/) for the full workspace provider reference.

20.5.3 LLM Plugins (pluggy)

LLM provider plugins use pluggy's hook system. The hook specification is defined in `LLMHookSpec` (`src/sase/llm_provider/_hookspec.py`). Core dispatch hooks (`llm_invoke` , `llm_resolve_model_name`) use `firstresult=True` so the first matching plugin handles a call; metadata hooks (`llm_provider_name` , `llm_known_model_names` , `llm_skill_template_context` , `llm_skill_deploy_subpath` , `llm_cli_status_color` , `llm_autodetect_priority` , `llm_autodetect_cli_name` , `llm_default_retry_config`) are invoked per-plugin by the registry so each provider contributes its own metadata. All hook method names are prefixed with `llm_` .

Core Sase ships Claude, Codex, Antigravity (`agy`), Qwen, and OpenCode providers as built-in entry points. Additional providers belong in external plugin packages that declare `sase_llm` entry points and provide their own metadata hooks.

See docs/llms.md (https://sase.sh/llms/) for the full LLM provider reference, including authoring new providers with `@hookimpl` .

20.5.4 XPrompt Plugins

Plugin packages can contribute xprompt templates by declaring a `sase_xprompts` entry point that points to a module. The module's package directory is searched for `xprompts/*.md` files and `xprompts/*.yml` / `xprompts/*.yaml` workflow files. Plugin xprompts are priority 8 in the discovery order (https://sase.sh/xprompt/#discovery-order) (above built-in files and below config-based xprompts).

20.5.5 Config Plugins

Plugin packages can provide default configuration by declaring a `sase_config` entry point. The referenced module's package must contain a `default_config.yml` file. Plugin configs are merged between the bundled package defaults and the user's `sase.yml` . See the Deep-Merge System (https://sase.sh/configuration/#deep-merge-system) for details on the merge chain.

20.6 Disabling Plugins

Resource plugins can be disabled via environment variables:

Variable	Effect
`SASE_DISABLE_PLUGINS`	Disable resource plugin loading for config and xprompts
`SASE_DISABLE_PLUGIN_XPROMPTS`	Disable xprompt/workflow resource plugins only

`SASE_DISABLE_PLUGIN_CONFIG`	Disable plugin `default_config.yml` resource loading only

Any non-empty value enables the disable. The VCS, workspace, and LLM provider registries currently load their provider entry points directly and do not consult these resource-plugin disable switches.

20.7 Writing a Plugin

A sase plugin is a standard Python package that declares entry points in `pyproject.toml`.

20.7.1 Example: VCS Plugin

```
# pyproject.toml
[project.entry-points."sase_vcs"]
my_vcs = "my_sase_plugin.vcs:MyVCSPlugin"

[project.entry-points."sase_config"]
my_vcs = "my_sase_plugin"
```

The VCS plugin class implements hooks from `VCSHookSpec` using the `@hookimpl` decorator:

```
from sase.vcs_provider._hookspec import hookimpl

class MyVCSPlugin:
    @hookimpl
    def vcs_checkout(self, revision: str, cwd: str) -> tuple[bool, str | None] | None:
        # Implementation here
        ...

    @hookimpl
    def vcs_diff(self, cwd: str) -> tuple[bool, str | None] | None:
        # Implementation here
        ...
```

Methods should return `None` (implicitly or explicitly) for operations they don't support, allowing other plugins to handle them.

20.7.2 Example: Workspace Plugin

```
# pyproject.toml
[project.entry-points."sase_workspace"]
my_workspace = "my_sase_plugin.workspace:MyWorkspacePlugin"
```

The workspace plugin class implements hooks from `WorkspaceHookSpec` using the `@hookimpl` decorator:

```python
from sase.workspace_provider._hookspec import WorkflowMetadata, hookimpl

class MyWorkspacePlugin:
    @hookimpl
    def ws_get_workflow_metadata(self) -> WorkflowMetadata | None:
        return WorkflowMetadata(
            workflow_type="my_vcs",
            ref_pattern=r"#my_vcs:(\w+)",
            display_name="My VCS",
            pre_allocated_env_prefix="SASE_MYVCS",
            vcs_family="git",
            vcs_provider_name="my_vcs",
        )

    @hookimpl
    def ws_detect_workflow_type(self, project_file: str) -> str | None:
        # Return workflow type if this plugin handles the project
        ...
```

20.7.3 Example: XPrompt Plugin

Place xprompt files in your package's `xprompts/` directory and register the module:

```toml
[project.entry-points."sase_xprompts"]
my_plugin = "my_sase_plugin"
```

```
my_sase_plugin/
├── __init__.py
└── xprompts/
    ├── my_template.md
    └── my_workflow.yml
```

Use `.md` for prompt templates and `.yml` / `.yaml` for workflow definitions.

20.7.4 Example: Config Plugin

Place a `default_config.yml` alongside your module and register it:

```toml
[project.entry-points."sase_config"]
my_plugin = "my_sase_plugin"
```

```
my_sase_plugin/
├── __init__.py
└── default_config.yml
```

Plugin configs are merged using the deep-merge system (https://sase.sh/configuration/#deep-merge-system). User config in `sase.yml` takes precedence over plugin defaults.

20.7.5 Example: LLM Provider Plugin

LLM providers declare a `sase_llm` provider class:

```toml
[project.entry-points."sase_llm"]
my_llm = "my_sase_plugin.llm:MyLLMProvider"
```

The provider implements hooks from `LLMHookSpec` using `@hookimpl`, including `llm_invoke()` for execution and metadata hooks such as `llm_provider_name()`, `llm_known_model_names()`, and `llm_autodetect_priority()`. See docs/llms.md (https://sase.sh/llms/#external-provider-plugins) for the full provider contract.

21 LLM Provider Integration

This document describes the LLM provider abstraction layer in sase. The system supports pluggable LLM backends (Claude Code, Codex, Antigravity CLI (`agy`), Qwen Code, and OpenCode are bundled; additional providers can ship as external plugins) behind a shared orchestration layer that handles preprocessing, invocation, and postprocessing.

21.1 Table of Contents

- Overview
- Provider Architecture
- Commit Finalization
- Claude Code Integration
- Antigravity (`agy`) Integration
- Codex CLI Integration
- Qwen Code Integration
- OpenCode Integration
- External Provider Plugins
- Configuration
- Per-Prompt Provider Switching
- Model Tier System
- Worker Model
- Temporary Default Override
- Environment Variables
- CLI Flags
- Retry and Fallback
- Token Usage Tracking
- Prompt Preprocessing Pipeline
- Subprocess Streaming
- Postprocessing
- Chat History
- Invocation Lifecycle

21.2 Overview

The LLM provider layer decouples prompt handling from the underlying LLM backend. All providers share a common preprocessing pipeline, subprocess streaming mechanism, and postprocessing workflow. The actual LLM invocation is delegated to a pluggable provider selected at runtime.

Key design principles:

- **Providers are thin**: They only construct CLI commands and run subprocesses. All preprocessing and postprocessing lives in the shared orchestration layer.
- **Registry-based selection**: Providers register themselves by name and are resolved via config or explicit override.
- **Tier-based model selection**: Callers request a "large" or "small" tier; the provider maps it to a concrete model.

- **Runtime-uniform commit enforcement**: SASE agent sessions use a shared commit finalizer instead of provider-specific native stop hooks.

21.2.1 Source Layout

File	Purpose
`src/sase/llm_provider/__init__.py`	Public API exports
`src/sase/llm_provider/base.py`	`LLMProvider` abstract base class
`src/sase/llm_provider/_hookspec.py`	Pluggy hook specifications (`LLMHookSpec`)
`src/sase/llm_provider/_plugin_manager.py`	Plugin manager wrapping pluggy (`LLMPluginManager`)
`src/sase/llm_provider/claude.py`	Claude Code provider implementation
`src/sase/llm_provider/codex.py`	Codex CLI provider implementation
`src/sase/llm_provider/agy.py`	Antigravity CLI (`agy`) provider implementation
`src/sase/llm_provider/qwen.py`	Qwen Code provider implementation
`src/sase/llm_provider/opencode.py`	OpenCode provider implementation
`src/sase/llm_provider/registry.py`	Provider registration and lookup
`src/sase/llm_provider/config.py`	Config file reader (`sase.yml`)
`src/sase/llm_provider/temporary_override.py`	Primary/worker temporary override state and resolution
`src/sase/llm_provider/commit_finalizer.py`	Provider-neutral dirty-workspace finalizer
`src/sase/llm_provider/types.py`	`ModelTier`, `InvokeResult`, `LoggingContext` types
`src/sase/llm_provider/_invoke.py`	`invoke_agent()` orchestrator
`src/sase/llm_provider/_subprocess.py`	Provider stream-parser compatibility exports
`src/sase/llm_provider/_plan_utils.py`	Shared plan utilities
`src/sase/llm_provider/preprocessing.py`	Shared prompt preprocessing pipeline
`src/sase/llm_provider/postprocessing.py`	Logging, chat history, audio
`src/sase/llm_provider/retry_config.py`	`ProviderRetryConfig` (per-provider retry defaults)

21.3 Provider Architecture

21.3.1 Base Class

All providers implement the `LLMProvider` abstract base class:

```
class LLMProvider(ABC):
    @abstractmethod
    def invoke(
        self,
        prompt: str,
        *,
        model_tier: ModelTier,
        suppress_output: bool = False,
        model_override: str | None = None,
    ) -> InvokeResult: ...
```

Parameter	Type	Description
`prompt`	`str`	Already-preprocessed prompt text
`model_tier`	`ModelTier`	`"large"` or `"small"`
`suppress_output`	`bool`	If `True`, suppress real-time console output
`model_override`	`str | None`	Concrete model name from `%model`, a temporary override, or retry

Returns `InvokeResult(content=..., usage=...)`. Providers raise `subprocess.CalledProcessError` for failed CLI exits or a provider-specific exception for launch/configuration failures.

21.3.2 Registry

Providers are discovered via `importlib.metadata.entry_points(group="sase_llm")`. The built-in providers are packaged the same way as external provider plugins; their entry points live in `pyproject.toml`:

```
[project.entry-points."sase_llm"]
claude = "sase.llm_provider.claude:ClaudeCodeProvider"
codex  = "sase.llm_provider.codex:CodexProvider"
agy = "sase.llm_provider.agy:AgyProvider"
opencode = "sase.llm_provider.opencode:OpenCodeProvider"
qwen   = "sase.llm_provider.qwen:QwenProvider"
```

External plugin packages declare additional entries under the same group.

To get a provider instance:

```
provider = get_provider()          # Uses default from config
provider = get_provider("claude")  # Explicit provider name
```

21.3.3 Selection Logic

1. If `provider_name` is passed to `invoke_agent()`, use that.
2. If the prompt has a `%model` directive, resolve explicit `provider/model` syntax first, then known model names from installed plugin metadata.
3. If no explicit provider/model was supplied, use an active temporary override from `~/.sase/llm_override.json`.
4. Otherwise, read the `llm_provider.provider` field from `~/.config/sase/sase.yml`.
5. If no config exists (or provider is empty), auto-detect by walking registered plugins in ascending `llm_autodetect_priority()` order and picking the first whose `llm_autodetect_cli_name()` is on `PATH`. Built-in priorities: `claude=0`, `codex=10`, `qwen=15`, `opencode=18`, `agy=30`. External plugins slot in by declaring their own priority. `agy` autodetects via the `agy` CLI name in the late-fallback slot.

21.4 Commit Finalization

After a provider returns successfully, `invoke_agent()` runs the provider-neutral commit finalizer before success postprocessing when the process is a SASE agent session (`SASE_AGENT_TIMESTAMP` is set). The finalizer checks the active project workspace through the active VCS provider and checks configured linked repositories as Git worktrees at their resolved `workspace_dir`. If it finds dirty enforced work, it sends the same provider a bounded follow-up prompt that lists the dirty files and instructs the agent to use the appropriate commit skill, such as `/sase_git_commit`. Dirty static linked repos (`workspace.strategy: none`) are included in that prompt only as advisory work and do not fail the run if they remain dirty. A narrow generated SDD plan closeout, where the only enforced change is one markdown file's frontmatter `status: wip` becoming `status: done`, is committed directly with a `TYPE=sdd` commit instead of consuming a provider follow-up pass.

The finalizer skips when the call is outside a SASE agent session, when `commit.finalizer.enabled` is false, or when `SASE_DISABLE_COMMIT_STOP_HOOK=1` is set. When an artifacts directory is available, each follow-up pass writes `commit_finalizer_pass_<N>_prompt.md` and `commit_finalizer_pass_<N>_response.md`; the final outcome is written to `commit_finalizer_result.json`. If the workspace remains dirty after `commit.finalizer.max_passes`, the invocation is converted into an `LLMInvocationError` rather than being logged as a successful clean run.

The older provider-native commit hook scripts are no longer shipped; SASE-launched agent sessions rely on the shared finalizer path.

21.5 Claude Code Integration

The `ClaudeCodeProvider` invokes the `claude` CLI tool.

21.5.1 Command Construction

```
claude -p --verbose --model <alias> --output-format stream-json --dangerously-skip-permissions --session-id <uuid>
[extra_args...]
```

The prompt is written to stdin. Output is streamed as JSON events; SASE extracts assistant text and token usage from the stream.

21.5.2 Model Mapping

Tier	Claude CLI Alias
`large`	`opus`
`small`	`sonnet`

21.5.3 Environment Variables

Variable	Description
`SASE_LLM_LARGE_ARGS`	Extra CLI args for `large` tier (generic, preferred)

`SASE_LLM_SMALL_ARGS`	Extra CLI args for `small` tier (generic, preferred)
`SASE_CLAUDE_LARGE_ARGS`	Extra CLI args for `large` tier (Claude-specific fallback)
`SASE_CLAUDE_SMALL_ARGS`	Extra CLI args for `small` tier (Claude-specific fallback)

The generic `SASE_LLM_*_ARGS` variables take precedence. Values are split on whitespace and appended to the command.

21.5.4 Timer Display

While waiting for a response, a `provider_timer("Waiting for Claude")` spinner is shown (unless `suppress_output` is `True`).

21.5.5 Claude Tool-Call Hooks

To record what tools an agent actually invoked (file reads, edits, bash commands, etc.), `ClaudeCodeProvider.invoke()` asks Claude Code to call back into SASE every time a tool runs. It does this by writing a pair of `PreToolUse` and `PostToolUse` hook entries into the workspace's `.claude/settings.local.json` for the duration of the agent run. Each entry matches all tools (`"matcher": "*"`) and invokes the `sase_claude_tool_hook` console script, which reads the Claude-supplied JSON payload from stdin and appends one normalized record (schema version 3) to `$SASE_ARTIFACTS_DIR/tool_calls.jsonl`:

- The `PreToolUse` hook writes a pending entry capturing the tool name and a redacted version of its input.
- The `PostToolUse` hook writes the matching result entry: success/failure/interrupted status, the call's duration, and a length-bounded preview of the response.

The ACE Tools panel reads this same `tool_calls.jsonl` to render the per-agent timeline — see Agents Tab Tools Panel (https://sase.sh/ace/#agents-tab-tools-panel).

Installation and cleanup are wrapped in a `claude_hooks_session()` context manager that is careful not to corrupt user-managed Claude settings:

- Writes to `.claude/settings.local.json` go through `tmp + os.replace` so a killed agent cannot leave a half-written file behind.
- Each SASE-installed hook command carries a `_sase_managed` sentinel value. On exit, cleanup removes only entries carrying that sentinel; any pre-existing user or project hooks (including hooks for unrelated events such as `Notification`) are left untouched.
- "Home-mode" launches — agents started outside a tracked workspace, identified by the absence of all three of `SASE_GIT_WORKSPACE_DIR`, `SASE_CD_WORKSPACE_DIR`, and `SASE_ACTIVE_PROJECT_DIR` — skip the settings mutation entirely. They emit a `claude_hooks_skipped` diagnostic to `tool_calls_writer_errors.jsonl` so the operator can see why the hook records are missing, and rely on the stream-derived fallback writer (below) to populate the timeline.
- If `.claude/settings.local.json` exists but is malformed JSON, it is left alone, the run logs a diagnostic, and the fallback writer takes over.

- If SASE created the file (it did not pre-exist) and only SASE entries remain at exit, both the file and an empty `.claude` directory are removed so the workspace is left clean.

The collector script itself is intentionally non-blocking: malformed JSON, non-object payloads, exceptions inside the collector, a missing `SASE_ARTIFACTS_DIR`, and unrecognized hook event names all produce a best-effort diagnostic (or a silent no-op when stdin is empty) and exit 0. This guarantees that a SASE-side bug can never make Claude surface the hook as a tool-call failure to the agent.

The hook-based writer coexists with a stream-derived fallback writer in the LLM provider layer, which parses tool calls out of the Claude streaming response. Both writers append to the same artifact, and the Tools-panel reader accepts schema versions 1, 2, and 3. When hook and stream records describe the same `tool_use_id`, the reader keeps the hook-derived record and suppresses the duplicate stream-derived row; otherwise, older stream-only artifacts remain readable.

The normalized tool-call artifact is still Python/TUI-owned glue rather than a shared `sase-core` contract. Move it into `../sase-core` only if another frontend or integration needs to produce or consume exactly the same schema through the Rust boundary.

Source: `src/sase/llm_provider/claude.py`, `src/sase/llm_provider/_claude_hooks.py`, `src/sase/llm_provider/_tool_calls.py`, `src/sase/scripts/sase_claude_tool_hook.py`, `src/sase/ace/tui/tools/reader.py`

21.6 Antigravity (`agy`) Integration

The `AgyProvider` invokes Google's Antigravity CLI (`agy`), the replacement for the retired consumer Gemini CLI. It is a plain-stdout provider: Antigravity CLI 1.0.10 does not document a machine-readable JSON/stream output mode, so SASE streams plain stdout instead of parsing a structured event stream.

21.6.1 Command Construction

```
agy --print-timeout <duration> --model <model> --dangerously-skip-permissions --print <prompt>
```

The prompt is passed as the value of `--print` (not on stdin) as a single argv element, so prompts containing quotes, newlines, or shell metacharacters are never shell-interpolated. `--print-timeout` defaults to `24h` (Antigravity's own `5m` default is too short for long agentic runs) and is a Go duration string.

Because the current Antigravity CLI does not document a stable stdin or prompt-file contract for print mode, SASE cannot fall back to streaming the prompt when that single argv element becomes too large for the OS. `AgyProvider` therefore rejects prompts above a conservative 120 KiB UTF-8 guard before spawning `agy`, with an error that names the upstream argv transport limitation and asks the user to reduce the prompt or use a stdin-capable provider.

21.6.2 Model Mapping

`agy` model display names are used verbatim — they contain spaces and parentheses (e.g. `Gemini 3.5 Flash (High)`). The tier defaults are:

Tier	Model	Short alias
`large`	`Gemini 3.5 Flash (High)`	`flash35h`
`small`	`Gemini 3.5 Flash (Low)`	`flash35l`

All other `agy models` names remain reachable through `%model:agy/<exact name>`, the model picker, and configured aliases.

21.6.3 Environment Variables

Variable	Description
`SASE_AGY_PATH`	Path to the Antigravity CLI binary (default: `"agy"`).
`SASE_AGY_PRINT_TIMEOUT`	Override the `agy --print-timeout` Go duration (default: `"24h"`).
`SASE_AGY_LARGE_ARGS`	Extra args for the `large` tier (after `SASE_LLM_LARGE_ARGS`).
`SASE_AGY_SMALL_ARGS`	Extra args for the `small` tier (after `SASE_LLM_SMALL_ARGS`).

21.6.4 Skill Deployment

`sase skill init -p agy` writes generated SASE skills to `~/.gemini/antigravity-cli/skills/`, the documented Antigravity global skill path. The leading `.gemini` here is an Antigravity-owned path, not a Gemini CLI path.

21.6.5 Structured Artifacts Parity Gap

Antigravity CLI 1.0.10 exposes no stable machine-readable contract: there is no documented `--output-format stream-json`, JSON event mode, or stable log/conversation schema. Because SASE will not scrape Antigravity's human TUI rendering to synthesize artifacts, the `agy` provider intentionally does **not** support the following until a stable upstream contract exists:

- **Tool-call timeline** — no `tool_calls.jsonl` rows are written, so the ACE Agents Tab Tools Panel (https://sase.sh/ace/#agents-tab-tools-panel) simply shows nothing for `agy` runs rather than inventing rows from display glyphs or prose.
- **Usage accounting** — `InvokeResult.usage` is `None` and no `usage.json` is written; `agy` print mode exposes no stable token counters.
- **Thinking extraction** — no thinking artifact is produced.

The plain-stdout path still writes `live_reply.md` (and `live_reply_timestamps.jsonl`) like every other provider, so the final reply, chat history, and resume support work normally. These structured features are fast-follow work gated on a future Antigravity machine-readable output/log/conversation contract.

21.6.6 Timer Display

While waiting for a response, a `Waiting for Antigravity` spinner is shown (unless `suppress_output` is `True`).

21.7 Codex CLI Integration

The `CodexProvider` invokes the OpenAI `codex` CLI tool.

21.7.1 Command Construction

Normal mode:

```
codex exec --model <model> --dangerously-bypass-approvals-and-sandbox --json --color never --skip-git-repo-check -
[extra_args...]
```

The prompt is written to stdin. Output is streamed as NDJSON events, with assistant text extracted from `item.completed` events.

21.7.2 Model Mapping

Tier	Codex Model
`large`	`gpt-5.5`
`small`	`codex-mini-latest`

21.7.3 Plan Handling

The Codex provider does not enable Codex CLI's native plan mode. SASE planning flows are implemented at the orchestration layer through workflows, xprompts, and the `sase_plan` skill, so provider behavior stays consistent across runtimes.

21.7.4 Environment Variables

Variable	Description
`SASE_LLM_LARGE_ARGS`	Extra CLI args for `large` tier (generic, preferred)
`SASE_LLM_SMALL_ARGS`	Extra CLI args for `small` tier (generic, preferred)
`SASE_CODEX_PATH`	Path to the Codex CLI binary (default: PATH, then NVM_BIN)
`SASE_CODEX_LARGE_ARGS`	Extra CLI args for `large` tier (Codex-specific fallback)
`SASE_CODEX_SMALL_ARGS`	Extra CLI args for `small` tier (Codex-specific fallback)
`SASE_CODEX_DISABLE_SHADOW_HOME`	Set to `1` to disable the disposable Codex home

The generic `SASE_LLM_*_ARGS` variables take precedence over `SASE_CODEX_*_ARGS`.

By default, SASE launches Codex with a per-invocation shadow `CODEX_HOME` under `~/.cache/sase/codex_home/`. The shadow home copies `config.toml` and symlinks other Codex home entries back to the real Codex home so Codex can read auth, hooks, skills, logs, and caches while any config rewrites stay disposable. The shadow directory

is removed after each Codex subprocess exits. Set `SASE_CODEX_DISABLE_SHADOW_HOME=1` to pass through the inherited environment directly for debugging or emergency compatibility.

21.7.5 Codex Tool-Call Capture

SASE captures Codex tool calls from the `codex exec --json` NDJSON stream; it does not install Codex hooks or mutate user Codex configuration for telemetry. When `SASE_ARTIFACTS_DIR` is present, the stream parser appends normalized Codex records to `$SASE_ARTIFACTS_DIR/tool_calls.jsonl` for the ACE Agents Tab Tools Panel (https://sase.sh/ace/#agents-tab-tools-panel).

Current fixture coverage is based on Codex CLI `0.130.0`. For stream items that expose both start and completion events (`command_execution`, `file_change`, and named tool items), SASE writes `ToolUse` and `ToolResult` rows with `runtime: "codex"` and `source: "stream"`. The Tools-panel reader collapses those pairs into one row, preserving pending rows while a command is still running and showing result previews, failure/interruption status, and duration when the stream exposes enough data to compute it.

Older Codex stream shapes that only expose a completed `function_call` item remain readable as legacy `FunctionCall` rows. Those records can show the tool name and compact input target, but they do not invent response summaries, durations, or failure details that Codex did not emit.

Codex tool-call summaries use the same bounded and redacted artifact helpers as the other providers. Set `SASE_TOOL_LOG_FULL=1` only for explicit debugging sessions when raw tool input or output is needed in the local artifact.

21.7.6 Timer Display

While waiting for a response, a `provider_timer("Waiting for Codex")` spinner is shown (unless `suppress_output` is `True`).

21.8 Qwen Code Integration

The `QwenProvider` invokes the `qwen` CLI tool.

21.8.1 Command Construction

```
qwen --input-format text --output-format stream-json --yolo --model <model> [extra_args...]
```

The prompt is written to stdin using Qwen's text input mode. Output is streamed as JSON events; SASE extracts assistant text from `assistant` events and falls back to the final `result` text when no assistant text is emitted.

21.8.2 Model Mapping

Tier	Qwen Model
`large`	`qwen3.6-plus`
`small`	`qwen3-coder-flash`

21.8.3 Authentication

Configure Qwen Code through its supported auth and settings flow before using it from SASE. Qwen OAuth free tier access ended on 2026-04-15; use API keys, Alibaba Cloud Coding Plan, OpenRouter, Fireworks, or another Qwen-supported provider instead of relying on the discontinued OAuth free tier.

21.8.4 Environment Variables

Variable	Description
`SASE_LLM_LARGE_ARGS`	Extra CLI args for `large` tier (generic, preferred)
`SASE_LLM_SMALL_ARGS`	Extra CLI args for `small` tier (generic, preferred)
`SASE_QWEN_PATH`	Path to the Qwen Code CLI binary (default: `qwen`)
`SASE_QWEN_LARGE_ARGS`	Extra CLI args for `large` tier (Qwen-specific fallback)
`SASE_QWEN_SMALL_ARGS`	Extra CLI args for `small` tier (Qwen-specific fallback)

The generic `SASE_LLM_*_ARGS` variables take precedence over `SASE_QWEN_*_ARGS`.

Qwen Code config is left in Qwen's normal locations (`~/.qwen/settings.json` and project `.qwen/settings.json`). SASE does not create a shadow Qwen home in the first implementation because local Qwen was unavailable during this phase, so no normal headless-run config mutation could be verified.

21.8.5 Qwen Tool-Call Capture

SASE captures Qwen tool calls from the `qwen --output-format stream-json` event stream; it does not install Qwen hooks. When `SASE_ARTIFACTS_DIR` is present, the stream parser normalizes Qwen's nested `tool_use` and `tool_result` blocks into records appended to `$SASE_ARTIFACTS_DIR/tool_calls.jsonl` for the ACE Agents Tab Tools Panel (https://sase.sh/ace/#agents-tab-tools-panel) with `runtime: "qwen"` and `source: "stream"`. Malformed or unsupported tool-shaped events emit a diagnostic instead of producing a malformed record. The Tools-panel reader collapses each start/result pair into a single row.

21.8.6 Commit Finalization

SASE-launched Qwen runs use the shared provider-neutral commit finalizer described above; active SASE settings do not need repo-local or global Qwen commit-hook configuration.

21.8.7 Timer Display

While waiting for a response, a `provider_timer("Waiting for Qwen")` spinner is shown (unless `suppress_output` is `True`).

21.9 OpenCode Integration

The `OpenCodeProvider` invokes the `opencode` CLI tool.

21.9.1 Command Construction

```
opencode run --format json --dangerously-skip-permissions --model <provider/model> --dir <cwd> [extra_args...] <prompt>
```

The prompt is passed as OpenCode's `run [message..]` argument without shell interpolation. Output is streamed as JSONL events; SASE extracts assistant text from `text` events, captures errors from `error` events, and accumulates token counters from `step_finish` events when OpenCode reports them.

21.9.2 Model Mapping

OpenCode model IDs normally include an upstream provider prefix. Use `%model:opencode/<provider/model>` to route a single SASE prompt to a concrete OpenCode model.

Tier	OpenCode Model
`large`	`anthropic/claude-sonnet-4-5`
`small`	`openai/gpt-5-mini`

21.9.3 Authentication and Config

Configure OpenCode through its normal auth and settings flow before using it from SASE. OpenCode stores auth under its XDG data directory and reads config from its XDG config directory plus project `.opencode` config. Use `opencode models` to inspect the models available in your configured OpenCode environment.

SASE deploys OpenCode skills under `~/.config/opencode/skills/`, which OpenCode scans as part of its config directory. SASE does not create a shadow OpenCode data/config home in this first implementation because OpenCode's normal headless run writes session/database state under its XDG data directory while reading auth/config from the standard locations.

21.9.4 Environment Variables

Variable	Description
`SASE_LLM_LARGE_ARGS`	Extra CLI args for `large` tier (generic, preferred)
`SASE_LLM_SMALL_ARGS`	Extra CLI args for `small` tier (generic, preferred)
`SASE_OPENCODE_PATH`	Path to the OpenCode CLI binary (default: `opencode`)
`SASE_OPENCODE_LARGE_ARGS`	Extra CLI args for `large` tier (OpenCode-specific fallback)
`SASE_OPENCODE_SMALL_ARGS`	Extra CLI args for `small` tier (OpenCode-specific fallback)

The generic `SASE_LLM_*_ARGS` variables take precedence over `SASE_OPENCODE_*_ARGS`.

21.9.5 Timer Display

While waiting for a response, a `provider_timer("Waiting for OpenCode")` spinner is shown (unless `suppress_output` is `True`).

21.10 External Provider Plugins

Additional LLM providers are shipped as external packages that declare `[project.entry-points."sase_llm"]` in their own `pyproject.toml`. Plugins carry all their own metadata (model names, skill deploy path, CLI status color, auto-detect priority, retry defaults) via pluggy `@hookimpl` methods — sase core has no plugin-specific branching.

External provider packages own their CLI invocation details, model metadata, skill deployment path, auto-detect priority, and retry defaults. Install the provider package in the same environment as sase to make its `sase_llm` entry point available.

21.11 Configuration

The LLM provider reads its configuration from `~/.config/sase/sase.yml` under the `llm_provider` key.

21.11.1 Config File

```yaml
llm_provider:
  provider: claude # or "qwen", "opencode", "agy" (default: auto-detect)
  worker_models:
    claude: codex/gpt-5.5 # worker default when primary is on Claude
    codex: claude/opus # worker default when primary is on Codex
  model_tier_map:
    large: opus
    small: sonnet
  model_aliases:
    other: claude/opus
```

21.11.2 Config Fields

Field	Type	Default	Description
`llm_provider.provider`	string	auto-detect	Which registered provider to use. Auto-detects by plugin-declared priority; built-ins default to claude → codex → qwen → opencode → agy.
`llm_provider.worker_models`	dict	unset	Optional worker-lane targets for plan follow-ups and epic phase agents, keyed by the effective primary lane. Values accept aliases, bare models, or explicit `provider/model`.
`llm_provider.model_tier_map.large`	string	-	Model identifier for the `large` tier
`llm_provider.model_tier_map.small`	string	-	Model identifier for the `small` tier
`llm_provider.model_aliases`	dict	-	Model aliases for `%model:<alias>` / `%m:<alias>`. Values can be bare known models, explicit `provider/model`, or nested provider-local model paths.

21.12 Per-Prompt Provider Switching

The `%model` directive (see xprompt directives (https://sase.sh/xprompt/#directives)) can switch both the model and the LLM provider for a single prompt. Provider resolution uses configured aliases first, then concrete provider/model syntax and known model metadata.

21.12.1 Configured Model Aliases

Use `llm_provider.model_aliases` to define launch-time aliases for reusable prompts:

```
llm_provider:
  model_aliases:
    other: claude/opus
```

Then prompts can use:

```
%model:other
%m(other,gpt-5.5)
```

Alias values may point at another alias, a bare known model such as `opus`, an explicit provider/model string such as `claude/opus`, or a nested provider-local path such as `opencode/anthropic/claude-sonnet-4-5`. Cycles are ignored and fall back to the raw input.

Reserved alias: `other`

The literal alias name `other` is reserved as a context-aware key. When a temporary default override is active, `%model:other` (and `%m:other`) resolves to the `(provider, model)` that was the effective default *immediately before* the override was set — captured in the override's `pre_override_*` snapshot. When no override is active, `other` falls back to whatever the user configured under `llm_provider.model_aliases.other` (or the literal model name `other` if no alias is configured).

This makes `%m(other, …)` always pair "the alternate model" with the current default, even when the user has temporarily switched their default via the ACE `,o` chord. Without the snapshot, `%m(other, …)` on an override-displaced default could otherwise launch the override's model side-by-side with itself.

Reserved alias: `worker`

The literal alias name `worker` is reserved for the worker lane. `%model:worker` and `%m(worker)` resolve to the current effective worker provider/model and shadow any `llm_provider.model_aliases.worker` entry.

This alias is how delegated launch sites opt into worker-lane selection without hardcoding a concrete model. For example, `sase bead work` emits `%model:worker` for phase agents that do not have an explicit per-bead model.

21.12.2 Explicit Provider/Model Syntax

Use `provider/model` to specify both explicitly:

```
%model:codex/o3
%model:claude/opus
%model:agy/flash35h
%model:qwen/qwen3.6-plus
%model:opencode/anthropic/claude-sonnet-4-5
```

21.12.3 Automatic Provider Resolution

Known model names are automatically mapped to their provider:

Model Name	Provider
`opus`, `sonnet`, `haiku`, `claude-fable-5`	claude
`gpt-5.5`, `gpt-5.3-codex`, `codex-mini-latest`, `o3`, `o4-mini`, `gpt-5.4`, `gpt-4.1`, `gpt-4.1-mini`, `gpt-4o`, `gpt-4o-mini`	codex
`Gemini 3.5 Flash (High)`, `Gemini 3.5 Flash (Medium)`, `Gemini 3.5 Flash (Low)`, `Gemini 3.1 Pro (High)`, `Gemini 3.1 Pro (Low)`, `Claude Sonnet 4.6 (Thinking)`, `Claude Opus 4.6 (Thinking)`, `GPT-OSS 120B (Medium)`	agy
`qwen3.6-plus`, `qwen3-coder-plus`, `qwen3-coder-flash`, `qwen3-max`, `qwen-plus`, `qwen-max`	qwen
`anthropic/claude-sonnet-4-5`, `anthropic/claude-opus-4-5`, `openai/gpt-5`, `openai/gpt-5-mini`, `google/gemini-3-flash-preview`, `qwen/qwen3-coder-plus`	opencode

Each installed plugin contributes its own model names via the `llm_known_model_names()` hook.

For unrecognized model names, the prompt falls back to the default provider and a warning is logged at invocation time.

Source: `src/sase/llm_provider/registry.py`, `src/sase/llm_provider/_invoke.py`

21.12.4 Model Short Aliases

Providers also declare compact display shorthands for long model ids via the `llm_model_short_aliases()` hook. These shorthands appear in provider/model agent-name suffixes (https://sase.sh/ace/#providermodel-suffixes) on the Agents tab and act as filter terms in the coder model picker. They are display-only: `%model` resolution uses known model names and configured model aliases, not these shorthands. For example, `%model:fable` does *not* select `claude-fable-5` — it falls back to the default provider (with a warning) unless you define `fable` as a configured model alias yourself.

Provider	Shorthands
claude	`claude-fable-5` → `fable`
codex	`codex-mini-latest` → `mini`, `gpt-5.5` → `gpt55`, `gpt-5.4` → `gpt54`, `gpt-5.3-codex` → `gpt53`, `gpt-4.1` → `gpt41`, `gpt-4.1-mini` → `gpt41m`, `gpt-4o-mini` → `gpt4om`
agy	`Gemini 3.5 Flash (High)` → `flash35h`, `Gemini 3.5 Flash (Medium)` → `flash35m`, `Gemini 3.5 Flash (Low)` → `flash35l`, `Gemini 3.1 Pro (High)` → `pro31h`, `Gemini 3.1 Pro (Low)` → `pro31l`, `Claude Sonnet 4.6 (Thinking)` → `sonnet46t`, `Claude Opus 4.6 (Thinking)` → `opus46t`, `GPT-OSS 120B (Medium)` → `gptoss120m`
qwen	`qwen3.6-plus` → `qwen36p`, `qwen3-coder-plus` → `qwen3cp`, `qwen3-coder-flash` → `qwen3cf`
opencode	`anthropic/claude-sonnet-4-5` → `sonnet45`, `anthropic/claude-opus-4-5` → `opus45`, `openai/gpt-5` → `gpt5`, `openai/gpt-5-mini` → `gpt5m`, `google/gemini-3-flash-preview` → `flash3`, `qwen/qwen3-coder-plus` → `qwen3cp`

Source: `llm_model_short_aliases()` in each provider module under `src/sase/llm_provider/`

21.13 Model Tier System

The model tier system abstracts away specific model names. Callers request either `"large"` (most capable) or `"small"` (faster/cheaper), and the provider maps the tier to a concrete model.

21.13.1 Type Definition

```
ModelTier = Literal["large", "small"]
```

21.13.2 Legacy Mapping

The old `"big"` / `"little"` terminology is still supported for backward compatibility:

Old Value	New Tier	Display Label
`"big"`	`"large"`	`BIG`
`"little"`	`"small"`	`LITTLE`

The `model_size` parameter on `invoke_agent()` is deprecated. Use `model_tier` instead.

21.13.3 Global Override

The model tier can be overridden globally via environment variable or CLI flag. The override forces ALL invocations to use the specified tier regardless of what the caller requests.

Resolution order:

1. `SASE_MODEL_TIER_OVERRIDE` env var (accepts `"large"`, `"small"`, `"big"`, `"little"`)
2. `SASE_MODEL_SIZE_OVERRIDE` env var (legacy, same values)
3. `--model-tier` / `--model-size` CLI flag (sets the env var)
4. Caller's `model_tier` parameter (default: `"large"`)

21.14 Worker Model

The worker model is an optional secondary default for delegated execution work. It is used by plan follow-up agents when the approval does not pick a specific follow-up model, and by `sase bead work` phase agents that do not have an explicit per-bead model. Planning and landing agents stay on the primary default unless their prompt or bead explicitly asks for a different model.

Configure it under `llm_provider.worker_models`:

```yaml
llm_provider:
  provider: claude
  worker_models:
    claude: codex/gpt-5.5
    codex: claude/opus
```

Each key selects which worker target to use for the current effective primary lane. Keys are matched in this order: exact `provider/model` first, bare model next, and provider last. Provider keys are defaults only, so `claude/opus` or `opus` beats `claude` when both are present. Values accept the same syntax as `%model`: a bare known model (`gpt-5.5`), a configured alias, an explicit `provider/model` pair (`codex/gpt-5.5`), or a nested provider-local model path.

For example:

```yaml
llm_provider:
  worker_models:
    claude/opus: codex/gpt-5.5
    sonnet: codex/o3
    claude: agy/flash35h
```

With that config, primary `claude/opus` uses `codex/gpt-5.5`, primary `claude/sonnet` uses `codex/o3`, and other Claude primary models use `agy/flash35h`.

21.14.1 Lane Precedence

Primary launches and worker launches resolve through separate lanes. The worker lane falls through to the primary lane only when no worker-specific setting exists:

```
Primary lane:
1. explicit %model directive
2. active primary temporary override (~/.sase/llm_override.json)
3. llm_provider.provider + requested model tier
4. provider auto-detection

Worker lane:
1. explicit %model directive or per-bead model
2. active worker temporary override (~/.sase/llm_worker_override.json)
3. matching llm_provider.worker_models entry
4. primary lane steps 2-4
```

Because of that fallthrough, leaving `worker_models` unset, empty, or unmatched preserves the old behavior: worker launches use the same effective default that a normal launch would have used. Active primary temporary overrides affect which mapping key is selected, so a primary override to `codex/o3` can match `codex/o3`, `o3`, or `codex`.

21.14.2 TUI Controls

Press `,o` in ACE to open the **Model Overrides** panel. The panel shows both lanes, their current effective model, and the source of that model (`override`, `config`, `follows primary`, or `default`). Use `s` / `c` / `x` for primary override set/change/clear and `w` / `W` for worker override set/change/clear. Active temporary worker overrides also appear as a compact `W ...` chip in the top bar; permanent `worker_models` config is visible in the modal instead.

The worker override state file is `~/.sase/llm_worker_override.json`. It uses the same JSON format, expiry behavior, and atomic writes as the primary override file.

21.15 Temporary Default Override

In addition to the tier-based global override, sase supports a **concrete** provider/model override that acts as a temporary session-level default. The ACE `,o` chord opens the dual-lane Model Overrides panel for primary and worker overrides (see docs/ace.md (https://sase.sh/ace/#model-overrides) for the TUI flow).

The temporary override only changes the *default* provider/model selection for new agent launches. It does **not** override:

- Already-running agents — they keep whatever provider/model they were launched with.
- Explicit `%model` prompt directives — they still take precedence.
- An explicit `provider_name=` argument to `invoke_agent()` — it still wins.

`SASE_MODEL_TIER_OVERRIDE` / `SASE_MODEL_SIZE_OVERRIDE` still force the tier for tier-based launches. A concrete temporary override supplies a provider and model directly, so it is used only when no explicit model/provider was requested.

21.15.1 Resolution Order (default provider/model)

When no `%model` directive and no explicit `provider_name` are present, the default is resolved as:

1. **Active primary temporary override** at `~/.sase/llm_override.json` (if not expired).
2. `llm_provider.provider` from the merged `sase.yml` config.
3. Auto-detection by plugin-declared priority (built-ins: claude, codex, qwen, opencode, then agy).

A concrete temporary override sets both the default provider and a concrete `model_override` for the next launch — so the agent metadata (running marker, plan review badge, agent rows) reflects the actual model that will run, not just the configured default.

21.15.2 State File

```
{
  "provider": "opencode",
  "model": "anthropic/claude-sonnet-4-5",
  "raw_model": "opencode/anthropic/claude-sonnet-4-5",
  "created_at": 1777470000.0,
  "expires_at": 1777473600.0,
  "source": "ace",
  "pre_override_provider": "claude",
  "pre_override_model": "opus",
  "pre_override_raw_model": "opus"
}
```

Field	Type	Description
`provider`	`str`	Resolved provider name (e.g. `"claude"`, `"codex"`, `"opencode"`).
`model`	`str`	Concrete model passed to the provider (e.g. `"o3"`, `"opus"`).
`raw_model`	`str`	Original user input (e.g. `"codex/o3"`, `"opencode/anthropic/..."`).

`created_at`	`float`	Unix timestamp when the override was set.
`expires_at`	`float | None`	Unix timestamp when the override expires; `null` means "until cleared".
`source`	`str`	Free-form tag indicating who set the override (e.g. `"ace"`).
`pre_override_provider`	`str | None`	Snapshot of the effective provider *before* the override was set. Used to resolve the reserved `"other"` alias dynamically.
`pre_override_model`	`str | None`	Snapshot of the effective model *before* the override. Pairs with `pre_override_provider` to form the `"other"` target.
`pre_override_raw_model`	`str | None`	Cosmetic copy of the displaced model's raw user-input form. May be `None` on legacy state files written before this field.

Writes are atomic (temp file + `os.replace`). Reads are best-effort self-cleaning: an expired or unparseable file is deleted on next access, so a forgotten override never lingers past its `expires_at`, even with no TUI running.

21.15.3 Model Resolution

The user-supplied `raw_model` is normalized through the same rules as `%model`:

- `provider/model` selects the provider explicitly (e.g. `codex/o3` or `opencode/anthropic/claude-sonnet-4-5`).
- A bare known model name infers its provider from plugin metadata (e.g. `sonnet` → claude).
- An unknown bare model is accepted and runs on the current default provider, matching `%model` behavior.

21.15.4 Duration Parsing

Durations accept compact unit suffixes: `15m`, `1h`, `1h30m`, `90m`, `2h15m30s`. Bare integers are interpreted as minutes (`45` → 45 minutes). The case-insensitive sentinel `until cleared` (or `until_cleared`) means "no expiry — persists until the user clears it from the TUI or another sase process clears the state file."

21.15.5 Public API

The override primitives live in `src/sase/llm_provider/temporary_override.py`:

Function	Purpose
`get_active_temporary_override(now=None, role=...)`	Read the active primary or worker override (auto-deletes expired/malformed files).
`set_temporary_override(raw, dur, source=, role=...)`	Write a new primary or worker override, replacing any existing one for that lane.
`clear_temporary_override(role=...)`	Remove the lane's override file. Safe to call when nothing is active.
`parse_override_duration(value)`	Parse a user-facing duration string into seconds (or `None`).
`resolve_effective_default_provider_model()`	Centralized helper used by metadata pre-resolution paths.
`resolve_effective_worker_provider_model()`	Resolve the worker lane: worker override, matching `worker_models`, then fallback.

21.15.6 Examples

- ACE chord `,o`, pick `codex/o3`, duration `1h` → `~/.sase/llm_override.json` is written; new launches default to CODEX(o3) for the next hour.
- ACE chord `,o`, pick `opencode/anthropic/claude-sonnet-4-5`, duration `1h` → new launches default to OPENCODE(anthropic/claude-sonnet-4-5).
- ACE chord `,o`, pick `sonnet`, duration `30m` → known bare model; provider resolves to claude via plugin metadata.
- ACE chord `,o`, choose **Clear override** → `~/.sase/llm_override.json` is removed; defaults revert to permanent config / autodetect.
- ACE chord `,o`, set worker override to `codex/gpt-5.5` for `1h` → `~/.sase/llm_worker_override.json` is written; new `%model:worker` launches use CODEX(gpt-5.5) until the override expires or is cleared.

21.16 Environment Variables

Complete reference of environment variables used by the LLM provider layer.

21.16.1 Generic (Provider-Agnostic)

Variable	Description
`SASE_LLM_LARGE_ARGS`	Extra CLI args for `large` tier invocations
`SASE_LLM_SMALL_ARGS`	Extra CLI args for `small` tier invocations
`SASE_MODEL_TIER_OVERRIDE`	Force all invocations to a specific model tier
`SASE_MODEL_SIZE_OVERRIDE`	Legacy alias for `SASE_MODEL_TIER_OVERRIDE`

21.16.2 Claude-Specific

Variable	Description
`SASE_CLAUDE_LARGE_ARGS`	Claude-specific extra args for `large` tier
`SASE_CLAUDE_SMALL_ARGS`	Claude-specific extra args for `small` tier

21.16.3 Codex-Specific

Variable	Description
`SASE_CODEX_PATH`	Path to the Codex CLI binary
`SASE_CODEX_LARGE_ARGS`	Codex-specific extra args for `large` tier
`SASE_CODEX_SMALL_ARGS`	Codex-specific extra args for `small` tier
`SASE_CODEX_DISABLE_SHADOW_HOME`	Set to `1` to disable the disposable Codex home

21.16.4 Qwen-Specific

Variable	Description
`SASE_QWEN_PATH`	Path to the Qwen Code CLI binary
`SASE_QWEN_LARGE_ARGS`	Qwen-specific extra args for `large` tier
`SASE_QWEN_SMALL_ARGS`	Qwen-specific extra args for `small` tier

21.16.5 Antigravity (`agy`)-Specific

Variable	Description
`SASE_AGY_PATH`	Path to the Antigravity CLI binary (default: `"agy"`).
`SASE_AGY_PRINT_TIMEOUT`	Override the `agy --print-timeout` Go duration (default: `"24h"`).
`SASE_AGY_LARGE_ARGS`	Antigravity-specific extra args for `large` tier
`SASE_AGY_SMALL_ARGS`	Antigravity-specific extra args for `small` tier

21.16.6 OpenCode-Specific

Variable	Description
`SASE_OPENCODE_PATH`	Path to the OpenCode CLI binary
`SASE_OPENCODE_LARGE_ARGS`	OpenCode-specific extra args for `large` tier
`SASE_OPENCODE_SMALL_ARGS`	OpenCode-specific extra args for `small` tier

External provider plugins document their own environment variables in their respective repos.

21.16.7 VCS Provider

Variable	Description
`SASE_VCS_PROVIDER`	Override VCS provider (`"git"` , `"hg"` , or `"auto"`)

21.17 CLI Flags

21.17.1 ace

Flag	Values	Description
`-m, --model-tier`	`large` , `small`	Override model tier for all LLM invocations
`--model-size`	`big` , `little`	Deprecated alias for `--model-tier`
`--vcs-provider`	`git` , `hg` , `auto`	Override VCS provider

21.17.2 axe

Flag	Values	Description
`--vcs-provider`	`git`, `hg`, `auto`	Override VCS provider

The `ace` command wires `--model-tier` / `--model-size` into the `model_tier_override` parameter of `AceApp`. The `--vcs-provider` flag is wired to the `SASE_VCS_PROVIDER` environment variable for downstream resolution.

21.18 Retry and Fallback

The LLM provider layer supports per-provider retry and fallback configuration. When an agent encounters a retryable error, it can automatically wait and retry, then optionally fall back to an alternate model.

21.18.1 Configuration

Retry behavior is configured per provider under `llm_provider.retry` in `sase.yml`:

```yaml
llm_provider:
  retry:
    claude:
      max_retries: 3
      error_patterns:
        - "API Error: 500"
      wait_times: [60, 300, 1800]
      fallback_model: "sonnet"
```

21.18.2 Config Fields

Field	Type	Default	Description
`max_retries`	int	`0`	Maximum retry attempts. `0` disables retrying.
`error_patterns`	list[str]	`[]`	Case-insensitive substring patterns matched against error output.
`wait_times`	list[int]	`[30]`	Per-retry wait times in seconds. Last value reused if list is too short.
`fallback_model`	`str | null`	`null`	Alternate model to use after exhausting all retries.
`continuation_prompt`	`str | null`	`null`	Text prepended to `state.current_prompt` on every retry (used to nudge the agent).
`preserve_workspace`	bool	`false`	Preserve on-disk edits across legacy in-process retry attempts.
`spawn_new_agent`	bool	`false`	Opt in to spawn-on-retry: a retryable error spawns a fresh detached child agent (as if `sase run -d` had been invoked) instead of in-process retry. See Spawn-on-Retry below.

21.18.3 Default Configuration

Retry defaults can come from two places: configured policy under `llm_provider.retry` and provider-supplied defaults from the `llm_default_retry_config()` hook. The bundled `default_config.yml` already provides configured policy for Claude and Codex; user config can replace or extend it through the normal config merge.

Claude:

- **max_retries**: 3
- **error_patterns**: `["API Error: 500", "API Error: 529", "Internal server error", "overloaded_error"]`
- **wait_times**: `[60, 300, 1800]` (1 min, 5 min, 30 min)
- **fallback_model**: `"sonnet"`

Codex:

- **max_retries**: 3
- **error_patterns**: `["exceeded retry limit", "429 Too Many Requests", "Too Many Requests", "rate limit", "failed to connect to websocket"]` — the Codex CLI's own give-up message, the terminal rate-limit status, and the transient websocket transport error. A bare `403 Forbidden` is deliberately excluded so a persistent auth failure is not retried forever.
- **wait_times**: `[60, 300, 1800]` (1 min, 5 min, 30 min) — rate limits need a real cool-down

21.18.4 Provider-Supplied Retry Defaults

Providers can also declare retry defaults through the `llm_default_retry_config()` hook. Both Claude and Codex declare a recovery entry that is merged with their configured policy.

Claude:

- **error patterns**: `"Prompt is too long"`, `"socket connection was closed unexpectedly"`, and `"API Error"`
- **max_retries**: 3
- **wait_times**: `[0]` — used only when no config layer supplies `wait_times`; the bundled Claude policy supplies `[60, 300, 1800]`, so that is the out-of-the-box backoff
- **continuation_prompt**: A short nudge that tells the coder to inspect `git status` / `git diff` before resuming, since prior edits are preserved on disk after a context-limit, socket-close, or API-error retry
- **preserve_workspace**: `true`

Codex:

- **error patterns**: `"exceeded retry limit"`, `"429 Too Many Requests"`, `"Too Many Requests"`, `"rate limit"`, and `"failed to connect to websocket"` — the transient transport / rate-limit failure mode where the Codex CLI exhausts its own internal reconnects and exits non-zero
- **max_retries**: 3

- **wait_times**: `[60, 300, 1800]` — the bundled Codex policy supplies the same backoff
- **continuation_prompt**: The same `git status` / `git diff` resume nudge as Claude
- **preserve_workspace**: `true`

Configured `llm_provider.retry.<provider>` values are merged on top of provider-supplied defaults: explicit falsy values (`max_retries: 0` to opt out entirely, `continuation_prompt: ""` to disable the nudge) override the built-in via key-presence checks. `error_patterns` is a de-duplicated union of built-in and configured lists.

On every retry attempt the `continuation_prompt` (if non-empty) is idempotently prepended to `state.current_prompt` before the next invocation — the prepend is gated on a `startswith` check so repeated retries don't stack duplicate nudges. Workspaces are preserved across Claude's built-in context-limit, socket-close, and API-error retries (no workspace wipe), so on-disk edits remain available to the restarted session.

21.18.5 Retry Flow

```
Error detected
│
├── Does error match error_patterns? (case-insensitive substring)
│   ├── No  → fail immediately
│   └── Yes → retry_count < max_retries?
│       ├── Yes → wait (wait_times[retry_count]) → retry
│       └── No  → fallback_model configured and not already using fallback?
│           ├── Yes → set fallback model override → retry once
│           └── No  → fail
```

Wait periods are interruptible — if the agent is killed during a wait, it stops immediately.

21.18.6 TUI Display

The ACE Agents tab reflects retry state (see Retry/Fallback Display (https://sase.sh/ace/#retryfallback-display)):

- **RETRYING (Ns)** — Waiting before the next attempt (bold orange, with countdown)
- **↻N** — Retry count annotation on running agents
- **▸Model** — Fallback model annotation (e.g., `↻3▸flash`)

21.18.7 Metadata Tracking

If any retries occurred or a fallback model was used, retry metadata is written to `done.json` in the agent's artifacts directory after execution completes (runs that succeed on the first attempt omit these fields):

```
{
  "retry_count": 2,
  "retry_errors": ["An unexpected critical error occurred: ..."],
  "used_fallback": false
}
```

When `used_fallback` is `true`, the metadata also includes the `fallback_model` that served the final attempt.

Source: `src/sase/llm_provider/retry_config.py`, `src/sase/axe/run_agent_exec_finalize.py`

21.18.8 Spawn-on-Retry

When `ProviderRetryConfig.spawn_new_agent=True`, a retryable error spawns a fresh detached child agent (as if `sase run -d` had been invoked) instead of running the next attempt in-process. The failing parent transfers its workspace claim to the child via `transfer_workspace_claim()` and exits with status `FAILED (RETRIED)`. This trades the small cost of a fresh process for two benefits:

- The workspace is preserved by design — the child skips `prepare_workspace()` and inherits the parent's in-progress edits via the transferred workspace claim. (Legacy in-process retry runs `prepare_workspace()` between attempts and wipes uncommitted file edits unless `preserve_workspace=True`.)
- A retry boundary becomes a real process boundary, which is more robust against memory leaks, lingering child processes, and stale interpreter state.

Linkage fields (written to both `agent_meta.json` and `done.json` so retry chains are queryable from either side):

Field	Meaning
`retry_of_timestamp`	Backward link: the parent agent's run timestamp.
`retried_as_timestamp`	Forward link: the child agent's run timestamp (written on the parent at handoff).
`retry_chain_root_timestamp`	The root agent's timestamp — stable across the entire chain.
`retry_attempt`	Depth in the chain (1-based).

State is carried across the boundary by a `retry_handoff.json` file written to the parent's artifacts directory; the child reads it before launch.

Fallback behavior: spawn-on-retry is opt-in (default `false`). If spawning fails (e.g. workspace transfer fails), the legacy in-process retry runs as a fallback so the user is never worse off.

Source: `src/sase/axe/run_agent_retry_spawn.py`, `src/sase/llm_provider/retry_config.py`

21.19 Legacy Thinking Metadata

Older parser helpers can still read provider thinking/reasoning artifacts when a caller uses them directly. For Claude extended-thinking events whose `thinking` text is empty but whose payload contains an opaque `signature`, those helpers produce an encrypted-thinking placeholder instead of hiding the block. When Claude also reports `message.usage.output_tokens`, the placeholder includes an approximate output-token count so the caller can tell that reasoning occurred even though the raw thought text is not available. The Agents tab now uses the Tools panel for provider tool activity instead of exposing these thinking helpers as a panel.

21.20 Token Usage Tracking

The LLM provider layer tracks token usage for providers that emit parseable usage events. Claude and Qwen usage is read from their stream-json result events. OpenCode usage is accumulated from `step_finish` token counters. Codex currently captures assistant text and reasoning summaries but does not emit `usage.json`.

When usage is available, input tokens, output tokens, cache-creation tokens, and cache-read tokens are persisted as a `usage.json` artifact in the agent run directory.

21.20.1 Artifact Format

```
{
  "input_tokens": 12345,
  "output_tokens": 6789,
  "cache_creation_input_tokens": 0,
  "cache_read_input_tokens": 3456
}
```

When telemetry is enabled, token counts are also recorded as Prometheus counters (`sase_llm_input_tokens_total`, `sase_llm_output_tokens_total`, `sase_llm_cache_read_tokens_total`) for monitoring and dashboards. See docs/telemetry.md (https://sase.sh/telemetry/) for the full telemetry reference.

Source: `src/sase/llm_provider/_subprocess.py`, `src/sase/llm_provider/types.py`

21.21 Prompt Preprocessing Pipeline

Before any prompt reaches a provider, it passes through the shared preprocessing pipeline defined in `preprocessing.py`. The pipeline has an early phase used for xprompt expansion and directive extraction, then a late phase used for command, file, template, and formatting work.

21.21.1 Steps

Phase	Step	Syntax	Description
Early	Optional workflow Jinja2	`{{ var }}`	Render workflow-supplied template context before xprompt
Early	xprompt references	`#name`	Expand reusable prompt snippets or workflows
Early	Prompt directives	`%model`, `%m`, other `%...` directives	Extract directives after xprompt expansion
Late	Disabled/fenced protection	`%xprompts_enabled:false`, fenced code	Protect regions that should not be rewritten
Late	Command substitution	`$(cmd)`	Execute shell commands and inline their output
Late	File references	`@path`	Process, validate, or skip file references
Late	Top-level Jinja2	`{{ var }}`	Render remaining top-level Jinja2 templates
Late	Prettier formatting	-	Format with prettier for consistent markdown
Late	Comment stripping	`<!-- ... -->`	Remove HTML/markdown comments
Late	Restore protected regions	fenced code / disabled-region placeholders	Restore protected content after rewrites

21.21.2 Order Matters

The pipeline runs in strict order. Prompt directives are extracted after xprompt expansion, so directives embedded in xprompts are honored. Late-phase command substitution and file-reference processing run with fenced blocks protected, so examples inside code fences are not executed or rewritten.

21.21.3 Home Mode

When `is_home_mode=True`, file-reference processing skips copy side effects. This is used when the invocation doesn't need workspace-local copies from `@path` references.

21.21.4 Source Functions

The preprocessing steps delegate to functions from two libraries:

- **`xprompt`**: `process_xprompt_references()`, `extract_prompt_directives()`, `is_jinja2_template()`, `render_toplevel_jinja2()`
- **`file_references`**: `process_command_substitution()`, `process_file_references()`, `validate_file_references()`, `format_with_prettier()`, `strip_html_comments()`

21.22 Subprocess Streaming

Providers use shared helpers in `_subprocess.py` and the `_subprocess_*` modules to stream LLM output in real time. Plain text, JSON-line, and provider-specific parsers share the same artifact hooks for live replies and usage files.

21.22.1 Mechanism

1. The provider spawns the CLI tool via `subprocess.Popen`. Providers that consume prompts from stdin set `stdin=PIPE`; OpenCode passes the prompt as the final `opencode run` argument.
2. The prompt is supplied using the provider's documented transport, either stdin or an argv message argument.
3. Stdout and stderr are set to **non-blocking** mode via `os.set_blocking()`.
4. A `select.select()` loop with a 0.1s timeout polls for readable data on both streams.
5. Lines are read, parsed when needed, and optionally printed to the console in real time.
6. After the process exits (`process.poll() is not None`), any remaining buffered output is drained.
7. Helpers return stdout/assistant text, stderr diagnostics, return code, and usage data when the provider reports it.

21.22.2 Live Reply File

When `SASE_ARTIFACTS_DIR` is set, the streaming output is also written in real-time to `<SASE_ARTIFACTS_DIR>/live_reply.md`. This file is used by the ACE TUI Agents tab to display the agent's reply as it streams in, and remains available after execution completes for the metadata panel's AGENT REPLY section.

Providers that support richer streams may write companion artifacts. Codex writes reasoning summaries to `<SASE_ARTIFACTS_DIR>/codex_thinking.jsonl`; providers with token counters write `<SASE_ARTIFACTS_DIR>/usage.json`.

21.22.3 Output Suppression

When `suppress_output=True`, lines are still captured but not printed to the console. This is used for background invocations where the caller only needs the final result.

21.23 Postprocessing

After a provider returns (or raises an error), the orchestration layer runs postprocessing steps.

21.23.1 On Success (`postprocess_success`)

1. **Audio notification**: Plays a sound via `run_bam_command("Agent reply received")` (skipped if `suppress_output`).
2. **Log to sase.md**: Appends a timestamped entry with the prompt and response to `<artifacts_dir>/sase.md` (if `artifacts_dir` is set).
3. **Save chat history**: Writes to `~/.sase/chats/` if `workflow` is set. See Chat History.

21.23.2 On Error (`postprocess_error`)

1. **Rich error display**: Prints the prompt and error via `print_prompt_and_response()` with an `_ERROR` suffix on the agent type label (skipped if `suppress_output`).
2. **Log to sase.md**: Same as success, but the response is the error message and the agent type gets an `_ERROR` suffix.
3. **Save error chat history**: Writes to `~/.sase/chats/` with an `_ERROR` agent suffix.

21.23.3 sase.md Log Format

Each entry in the log file follows this format:

```
## <timestamp> - <agent_type> - iteration <N> - tag <workflow_tag>

### PROMPT:

\`\`\` <prompt text> \`\`\`

### RESPONSE:

\`\`\` <response text> \`\`\`

---
```

21.23.4 Prompt File Saving

Before invocation, the preprocessed prompt is saved to `<artifacts_dir>/<agent_type>_prompt.md` (or `<agent_type>_iter_<N>_prompt.md` if an iteration number is set). This allows reviewing the exact prompt that was sent.

21.24 Chat History

Chat histories are stored as markdown files in `~/.sase/chats/`.

21.24.1 File Naming

```
<branch_or_workspace>-<workflow>-[<agent>-]<timestamp>.md
```

Part	Source	Example
`branch_or_workspace`	Output of `branch_or_workspace_name`	`my_feature`
`workflow`	Workflow name, normalized	`crs`, `run`
`agent`	Agent type (omitted if same as workflow)	`editor`, `planner`
`timestamp`	`YYmmdd_HHMMSS` format	`260214_153042`

Dashes and slashes in workflow names are normalized to underscores.

21.24.2 File Format

```
# Chat History - <workflow> (<agent>)

**Timestamp** <display_timestamp>

**MODEL** <provider>/<model>

**AGENT** <sase_agent_name>

## Previous Conversation

<previous history if resuming>

---

## Prompt

<prompt text>

## Response

<response text>
```

The `MODEL` and `AGENT` blocks are omitted when the invocation did not provide that metadata. `MODEL` can contain just a model name, just a provider name, or both. When both provider and model are known, it is rendered as `<provider>/<model>` unless the model already includes that prefix.

21.24.3 Resume Support

The `sase run --resume` flag resumes a previous conversation by agent name. The `#fork` workflow resolves the agent name to its artifacts directory, extracts the response path from `done.json`, and delegates to `#fork_by_chat` which loads the chat history and prepends it to the new conversation. The `--resume` flag also accepts a history file basename or full path for direct chat-file-based resumption via the `#fork_by_chat` workflow.

Fork expansion is recursive: if the loaded chat history itself contains `#fork` or `#fork_by_chat` references, those are expanded inline as well. Legacy `#resume` and `#resume_by_chat` references in old transcripts are still recognized. Cycle detection prevents infinite loops when chat histories reference each other.

21.25 Invocation Lifecycle

The `invoke_agent()` function in `_invoke.py` orchestrates the complete lifecycle of an LLM invocation. Here is the end-to-end flow:

```
invoke_agent(prompt, agent_type, model_tier, ...)
│
├── 1. Handle deprecated model_size → model_tier mapping
├── 2. Check SASE_MODEL_TIER_OVERRIDE / SASE_MODEL_SIZE_OVERRIDE env vars
├── 3. Build LoggingContext from parameters
│
├── 4. Preprocess prompt unless skip_preprocessing=True
│   ├── early phase: optional workflow Jinja2, xprompt expansion, directive extraction
│   └── late phase: command substitution, file refs, top-level Jinja2, formatting, comment stripping
│
├── 5. Resolve %model / temporary provider-model override
├── 6. Display decision counts (if not suppressed)
├── 7. Print prompt via Rich (if not suppressed)
├── 8. Generate or use provided timestamp
├── 9. Save prompt to artifacts directory
│
├── 10. Get provider from registry and invoke
│   ├── Build CLI command with flags
│   ├── Spawn subprocess (Popen)
│   ├── Supply prompt via provider transport
│   └── Stream stdout/stderr in real-time
│
├── 11. Run commit finalizer for SASE agent sessions
│   ├── Skip when disabled or outside an agent session
│   ├── Check main workspace and configured Git linked repos
│   ├── Treat static linked repos as advisory dirty targets
│   ├── Auto-commit exact tracked SDD done-status closeouts
│   └── Run bounded follow-up provider invocations until enforced repos are clean or failed
│
├── 12. Postprocess
│   ├── Success path:
│   │   ├── Audio notification
│   │   ├── Log to sase.md
│   │   └── Save chat history
│   └── Error path:
│       ├── Rich error display
│       ├── Log error to sase.md
│       └── Save error chat history
│
└── 12. Return AIMessage(content=response), or raise LLMInvocationError on failure
```

21.25.1 Parameters

Parameter	Type	Default	Description
`prompt`	`str`	(required)	Raw prompt to send
`agent_type`	`str`	(required)	Agent type label (e.g., `"editor"`)
`model_tier`	`ModelTier`	`"large"`	Model tier to use
`model_size`	`"big" | "little" | None`	`None`	Deprecated, use `model_tier`
`iteration`	`int | None`	`None`	Iteration number for logging
`workflow_tag`	`str | None`	`None`	Workflow tag for logging
`artifacts_dir`	`str | None`	`None`	Directory for sase.md, prompt, and stream files
`workflow`	`str | None`	`None`	Workflow name for chat history
`suppress_output`	`bool`	`False`	Suppress console output
`timestamp`	`str | None`	`None`	Shared timestamp (`YYmmdd_HHMMSS`)
`is_home_mode`	`bool`	`False`	Skip file copying for `@` references
`branch_or_workspace`	`str | None`	`None`	Override the chat-history filename prefix

`decision_counts`	`dict[str, Any] | None`	`None`	Planning agent decision counts
`provider_name`	`str | None`	`None`	Override provider (default from config)
`skip_preprocessing`	`bool`	`False`	Use `prompt` as already-preprocessed input
`directives`	`PromptDirectives | None`	`None`	Pre-extracted directives for `skip_preprocessing`

21.25.2 Return Value

On success, returns an `AIMessage` (from `langchain_core.messages`) whose `content` is the provider response. On provider failure, `invoke_agent()` logs the error and raises `LLMInvocationError` with the formatted error text.

22 VCS Provider Reference

The **VCS provider layer** is an abstraction that lets sase commands work with both **Git** and **Mercurial** repositories. Commands and workflows that touch version control, including `sase commit`, `sase ace`, `sase axe`, `sase revert`, and `sase restore`, delegate to a provider interface rather than calling VCS commands directly.

Git and Mercurial share the same SASE concepts: ChangeSpecs, workspace checkout, diff capture, commit/proposal dispatch, review submission, revert, and restore. Provider-specific capabilities and prerequisites still matter. For example, GitHub pull-request operations require the optional `sase-github` plugin and the GitHub CLI, while Mercurial support requires a maintained provider plugin that supplies `sase_hg_*` helper commands.

22.1 Plugin Architecture

VCS providers are implemented as pluggy (https://pluggy.readthedocs.io/) plugins. The core `sase` package only bundles the **BareGitPlugin** (for plain git repositories). Additional VCS backends are installed as separate packages:

Package	Plugin	Description
`sase` (core)	`BareGitPlugin`	Standard git operations (bundled)
`sase-github`	`GitHubPlugin`	Git + GitHub CLI (`gh`) for PR operations

Install optional providers via pip:

```
pip install sase-github   # GitHub PR support
```

Plugins register themselves via the `sase_vcs` entry point group. The plugin manager loads all registered plugins and dispatches VCS operations through pluggy's `firstresult=True` hook system — the first plugin that returns a non-`None` result wins.

22.1.1 Hook Specification

All VCS operations are defined in `VCSHookSpec` (`src/sase/vcs_provider/_hookspec.py`). Each method is prefixed with `vcs_` and returns `tuple[bool, str | None]` (success flag and optional output). Plugins implement only the hooks they support; unsupported operations return `None` and are skipped.

The hooks are organized into several groups:

- **Core operations** — `vcs_checkout`, `vcs_diff`, `vcs_diff_revision`, `vcs_apply_patch`, `vcs_apply_patches`, `vcs_add_remove`, `vcs_clean_workspace`, `vcs_commit`, `vcs_amend`, `vcs_rename_branch`, `vcs_rebase`, `vcs_archive`, `vcs_prune`, `vcs_stash_and_clean`
- **Optional core** — `vcs_resolve_revision`, `vcs_resolve_current_changespec_head_ref`, `vcs_show_revision`, `vcs_diff_with_untracked`, `vcs_committed_diff`, `vcs_get_default_parent_revision`, `vcs_diff_name_status`, `vcs_diff_line_stats`, `vcs_file_at_revision`
- **Sync operations** — `vcs_sync_workspace`, `vcs_is_sync_in_progress`, `vcs_get_conflicted_files`, `vcs_continue_sync`, `vcs_abort_sync`
- **Commit dispatch** — `vcs_create_commit`, `vcs_create_proposal`, `vcs_create_pull_request` (the three commit workflow methods dispatched by `CommitWorkflow`), plus `vcs_finalize_commit` (replays idempotent post-commit work — bead amend, push-with-retry — when `sase commit --resume` finishes a workflow whose dispatch was interrupted by a merge conflict; plugins that cannot safely replay finalization can leave this unimplemented, and the workflow will only replay its tracking steps). See commit_workflows.md (https://sase.sh/commit_workflows/#resume-after-conflict).
- **VCS-agnostic operations** — `vcs_abandon_change`, `vcs_prepare_description_for_reword`, `vcs_normalize_bug_value`, `vcs_get_change_url`, `vcs_get_change_body`

- **Info and review hooks** — `vcs_reword`, `vcs_reword_add_tag`, `vcs_get_description`, `vcs_get_branch_name`, `vcs_get_cl_number`, `vcs_get_workspace_name`, `vcs_has_local_changes`, `vcs_get_bug_number`, `vcs_mail`, `vcs_fix`, `vcs_upload`, `vcs_find_reviewers`, `vcs_rewind`
- **Branch naming hooks** — `vcs_derive_branch_name`, `vcs_derive_branch_name_with_suffix` (compute branch names from ChangeSpec names), `vcs_can_rename_branch` (check if branch renaming is supported)
- **Classification hooks** — `vcs_detect_repo_type` (detect VCS markers like `.hg/` or `.git/`) and `vcs_classify_repo` (classify git repos by remote URL, e.g. GitHub vs bare)

22.1.2 Disabling Plugins

The VCS provider registry loads provider entry points directly. It does not currently consult the resource-plugin disable switches described in docs/configuration.md (https://sase.sh/configuration/#plugin-system). Use `SASE_VCS_PROVIDER` or `vcs_provider.provider` to force a provider selection.

22.2 Provider Selection

Sase uses a 3-tier resolution strategy to decide which VCS provider to use. The first tier that returns a concrete provider wins.

22.2.1 Tier 1: Environment Variable

The `SASE_VCS_PROVIDER` environment variable takes highest priority.

```
# Force the Git provider family; GitHub remotes are reclassified when the plugin is installed.
SASE_VCS_PROVIDER=git sase commit my_feature

# Force hg provider
SASE_VCS_PROVIDER=hg sase ace

# Defer to next tier
SASE_VCS_PROVIDER=auto sase commit my_feature
```

The `--vcs-provider` CLI flag on `sase ace` and `sase axe` sets this variable internally:

```
# Equivalent to SASE_VCS_PROVIDER=git sase ace
sase ace --vcs-provider git

# Same for axe
sase axe --vcs-provider hg start
```

Valid values: `git`, `hg`, `auto`.

22.2.2 Tier 2: Configuration File

If the environment variable is not set (or is unset entirely — not `"auto"`), sase checks `~/.config/sase/sase.yml`:

```
vcs_provider:
  provider: git # or "hg" or "auto"
```

Setting `provider: auto` defers to auto-detection (Tier 3).

Note: If the environment variable is set to `"auto"`, the config file is skipped entirely and auto-detection runs directly. Only an unset environment variable consults the config.

22.2.3 Tier 3: Auto-Detection

If neither the environment variable nor config file specifies a provider, sase walks up the directory tree from the current working directory looking for `.hg/` or `.git/` directories. The first one found determines the provider.

- `.hg/` found first → Mercurial provider (`"hg"`)
- `.git/` found first → Git provider. If a plugin claims the Git remote, such as `sase-github` claiming GitHub URLs, that provider name wins.
- `.git/` found with a hosted remote (e.g., GitHub) but no VCS plugin claims the repo → falls back to `"bare_git"`. This preserves baseline commit capability even without provider-specific plugins like `sase-github`.
- `.git/` found without a readable `origin` URL and no plugin claim → **Error**: `VCSProviderNotFoundError`
- Neither found → **Error**: `VCSProviderNotFoundError`

With the bundled and documented optional providers, `detect_vcs()` commonly returns `"github"`, `"bare_git"`, or `"hg"`. Additional plugins may return their own provider names. `detect_vcs_family()` collapses `"github"` and `"bare_git"` into `"git"` for contexts that only care about the VCS family.

22.3 Per-Command VCS Usage

22.3.1 `sase commit`

Dispatches to one of three VCS methods (`create_commit`, `create_proposal`, `create_pull_request`) via the `CommitWorkflow` orchestrator. See `docs/commit_workflows.md` (https://sase.sh/commit_workflows/) for the full workflow reference.

Key VCS operations used:

Operation	Git	Mercurial
Bug number	Returns empty string (not applicable)	`sase_hg_branch_bug` command
Workspace name	`git config --get remote.origin.url` (extracts repo name)	`workspace_name` command
Create commit	`git add` + `git commit` + `git push`	`hg commit --name "<name>" --logfile "<logfile>"`
Create proposal	Save diff + provider workspace clean	`sase_hg_clean <diff_name>`
Create PR	Branch + commit + push; GitHub plugin creates the PR	Not supported natively
Change URL	GitHub plugin reads `gh pr view --json url -q .url`	`http://cl/<branch_number>`

Common CLI forms:

```
sase commit -m "Update parser"                        # create_commit
sase commit -t propose -m "Try parser cleanup"        # create_proposal
sase commit -t pr -n parser_cleanup -m "Update parser" # create_pull_request
```

22.3.2 `sase ace` TUI Actions

The ace TUI provides interactive actions that use VCS operations:

Sync (`s` key)

Syncs the workspace with the remote repository.

Step	Git	Mercurial
Checkout	`git checkout <name>`	`sase_hg_update <name>`
Sync	`git fetch origin` + `git rebase origin/<default_branch>`	`sase_hg_sync`

The git sync auto-detects the default branch via `git symbolic-ref refs/remotes/origin/HEAD`, then probes `origin/master` and `origin/main`, and finally falls back to `main`.

Mail (`m` key)

Pushes changes for review. The flow differs significantly between providers.

Git flow:

1. Display branch name and commit description
2. Prompt user to confirm push
3. `git push -u origin <branch>`
4. With the GitHub provider, check or create a PR through `gh`
5. Update ChangeSpec with the PR URL when the provider can return one

Mercurial flow:

1. Prompt for reviewers (1 or 2, or `@` to run `p4 findreviewers -c <cl_number>`)
2. Modify CL description with reviewer tags and startblock configuration
3. Reword CL description via `sase_hg_reword`
4. Prompt user to confirm mail
5. `hg mail -r <revision>`

Show Diff (`d` key)

Displays the diff for a ChangeSpec. Uses `diff()` for uncommitted changes or `diff_revision()` for committed revisions.

Type	Git	Mercurial
Uncommitted	`git diff HEAD`	`hg diff`
Revision	`git diff origin/<default>...<rev>` (merge-base)	`hg diff -c <rev>`

Revert (`X` key / status change to "Reverted")

Reverts a ChangeSpec by saving its diff and pruning the revision.

1. Save diff to `~/.sase/reverted/<name>.diff` via `diff_revision()`
2. Prune revision via `prune()`
3. Update status to "Reverted"

Operation	Git	Mercurial
Prune	`git branch -D <revision>`	`sase_hg_prune <revision>`

Restore (status change from "Reverted" to "WIP"/"Drafted")

Restores a previously reverted ChangeSpec.

1. Checkout parent or default branch via `checkout()`
2. Apply stashed diff via `apply_patch()`
3. Run `sase commit` to re-create the commit

Operation	Git	Mercurial
Checkout	`git checkout <target>`	`sase_hg_update <target>`
Apply patch	`git apply <path>`	`hg import --no-commit <path>`

Archive (status change to "Archived")

Archives a ChangeSpec by saving the diff, archiving the revision, and updating status.

1. Checkout the CL via `checkout()`
2. Save diff to `~/.sase/archived/<name>.diff`
3. Archive revision via `archive()`

Operation	Git	Mercurial
Archive	`git tag archive/<name> <name>` + `git branch -D <name>`	`sase_hg_archive <name>`

Reword (`w` key)

Amends the commit message without changing code.

Operation	Git	Mercurial
Reword	`git commit --amend -m <description>`	`sase_hg_reword <description>`

The Mercurial provider applies ANSI-C escape quoting to the description (escaping backslashes, single quotes, newlines, tabs, carriage returns) because `sase_hg_reword` uses `$'...'` shell quoting internally.

22.3.3 `sase axe`

Background daemon that periodically checks ChangeSpecs and runs hooks. Uses VCS operations for:

- **Hook running** — Workspace checkout and sync before running hooks
- **Mentor checks** — Checking for changes via `has_local_changes()`
- **Workspace sync** — Periodic sync via `sync_workspace()`

The `--vcs-provider` flag works identically to `sase ace`.

22.3.4 `sase revert`

Standalone command to revert a ChangeSpec. Performs the same operations as the ace TUI revert action:

1. Save diff via `diff_revision()` to `~/.sase/reverted/<name>.diff`
2. Prune revision via `prune()`
3. Update status to "Reverted"

22.3.5 `sase restore`

Standalone command to restore a reverted ChangeSpec:

1. Checkout parent (or default branch) via `checkout()`
2. Apply saved diff via `apply_patch()` from `~/.sase/reverted/` or `~/.sase/archived/`
3. Run `sase commit` to re-create the commit

22.4 Git Provider Details

Git support is split across providers. **BareGitPlugin** (bundled with core sase) handles standard `git` commands and bare-repo-backed workflows. **GitHubPlugin** (from the optional `sase-github` package) adds GitHub CLI (`gh`) support for PR operations and GitHub workspace references.

22.4.1 Branch Naming

Git branch names match ChangeSpec names exactly — no prefix stripping or underscore-to-hyphen conversion. Two VCS hooks control branch name derivation:

- `vcs_derive_branch_name()` — returns the base branch name (ChangeSpec name without `__<N>` suffix)
- `vcs_derive_branch_name_with_suffix()` — returns the full branch name including suffix

Immutable branch aliases: When a provider cannot rename branches (e.g., GitHub with open PRs), sase persists branch aliases in `~/.sase/projects/<project>/branch_map.json`. This maps the current ChangeSpec name to the actual git branch name. The `vcs_can_rename_branch()` hook tells the system whether renaming is possible — GitHub returns `False` for branches with open PRs, so alias mappings are used instead of `git branch -m`.

22.4.2 Branch Management

- Creates feature branches with `git checkout -b <name>` during commit
- Renames branches with `git branch -m <new_name>` (when `vcs_can_rename_branch()` returns `True`)
- Falls back to branch alias mapping when renaming is not possible
- Current branch detected via `git rev-parse --abbrev-ref HEAD`

22.4.3 PR Integration

GitHub PR operations use the `gh` CLI:

- **Create PR**: `gh pr create --fill` (auto-fills title/body from commit)
- **View PR**: `gh pr view --json url -q .url`
- **Get PR number**: `gh pr view --json number -q .number`

The bundled bare-git provider does not create PRs. Its mail action pushes the resolved branch to `origin`.

22.4.4 GitHub Plugin Scope

The GitHub plugin covers the core git/PR lifecycle by combining GitHub-specific hooks with the shared git provider mixins. It can classify GitHub remotes, create and inspect PRs, preserve immutable branch aliases for open PRs, resolve workspace references such as `#gh:<ref>`, and submit merged PRs through `gh pr merge`.

It does not currently provide the richer Mercurial-specific automation surface. In particular, GitHub PRs do not get plugin-supplied default ChangeSpec hooks, metahooks, mentor profiles, PR tags, or a provider-specific precommit/fix command unless users configure those in `sase.yml`. Reviewer-comment polling and comment-response automation are not enabled for GitHub PR URLs, reviewer discovery during mail preparation is not implemented, `vcs_rewind` has no GitHub backend, BUG values are left as provided, and Mercurial-only refresh/split workflows do not have GitHub equivalents.

These are plugin capability gaps, not core VCS limitations: ordinary git operations, diffing, branch management, commit/proposal/PR dispatch, conflict resume, and workspace setup are still provided by the shared git implementation.

22.4.5 Sync

```
git fetch origin
git rebase origin/<default_branch>
```

The default branch is auto-detected from `git symbolic-ref refs/remotes/origin/HEAD`, then `origin/master`, then `origin/main`, and finally `main`.

22.4.6 Archive

Preserves commits via a tag before deleting the branch:

```
git tag archive/<name> <name>
git branch -D <name>
```

22.4.7 Diff

- **Uncommitted changes**: `git diff HEAD` (falls back to `git diff` for empty repos)
- **Specific revision**: `git diff origin/<default>...<rev>` (three-dot merge-base syntax, showing the full PR diff). Falls back to `git diff <rev>~1 <rev>` for edge cases (detached HEAD, orphan branches), then to `git show` for root commits.

22.4.8 Workspace Info

- **Repository name**: Extracted from `git config --get remote.origin.url` (strips `.git` suffix), falls back to `git rev-parse --show-toplevel` basename
- **Local changes**: `git status --porcelain`
- **Commit description**: `git log --format=%B -n1 <revision>` (full) or `git log --format=%s -n1 <revision>` (short)

22.4.9 Tag Operations

Adding tags to commit descriptions:

```
git log --format=%B -n1 HEAD    # Read current message
git commit --amend -m "<msg>\n<tag>=<value>"    # Append tag
```

22.5 Mercurial Provider Details

Mercurial support is provided by external provider plugins. A Mercurial provider uses a combination of standard `hg` commands and `sase_hg_*` wrapper commands.

22.5.1 Core Commands

Operation	Command
Commit	`hg commit --name <name> --logfile <logfile>`
Amend	`sase_hg_amend [--no-upload] <note>`

Checkout	`sase_hg_update <revision>`
Sync	`sase_hg_sync`
Archive	`sase_hg_archive <revision>`
Prune	`sase_hg_prune <revision>`
Rename	`sase_hg_rename <new_name>`
Rebase	`sase_hg_rebase <branch> <new_parent>`
Reword	`sase_hg_reword <description>`
Add tag	`sase_hg_reword --add-tag <name> <value>`
Clean	`sase_hg_clean <diff_name>` (saves diff and cleans)

22.5.2 Branch and Workspace Info

Info	Command
Branch name	`branch_name`
CL number	`branch_number`
Bug number	`sase_hg_branch_bug`
Workspace name	`workspace_name`
Local changes	`branch_local_changes`

22.5.3 Description Management

Operation	Command
Full description	`cl_desc -r <revision>`
Short description	`cl_desc -s`

22.5.4 Review Operations

Operation	Command
Mail for review	`hg mail -r <revision>`
Find reviewers	`p4 findreviewers -c <cl_number>`
Upload	`hg upload tree`
Fix	`hg fix`

22.5.5 Diff and Patch

Operation	Command
Uncommitted diff	`hg diff`
Revision diff	`hg diff -c <revision>`

Apply patch	`hg import --no-commit <path>`
Rewind	`sase_hg_rewind <diff_paths>`

22.5.6 Change URL

CL URLs follow the pattern `http://cl/<number>`, where the number comes from `branch_number`.

22.5.7 Description Escaping

The `prepare_description_for_reword()` method escapes descriptions for `sase_hg_reword`'s `$'...'` shell quoting:

- `\` → `\\` (backslashes first)
- `'` → `\'`
- newline → `\n`
- tab → `\t`
- carriage return → `\r`

22.6 Diff Management

Sase maintains diff files in `~/.sase/` for tracking changes across operations.

22.6.1 Diff Storage Locations

Directory	Purpose	When Used
`~/.sase/diffs/YYYYMM/<cl_name>-<timestamp>.diff`	Pre-commit/amend diff snapshots	Every commit and amend
`~/.sase/reverted/<name>.diff`	Stashed diff for reverted CLs	`sase revert` / ace revert action
`~/.sase/archived/<name>.diff`	Stashed diff for archived CLs	ace archive action

22.6.2 Patch Application

Provider	Apply Command
Git	`git apply <path>`
Mercurial	`hg import --no-commit <path>`

Multiple patches can be applied at once — both providers accept multiple paths in a single command.

22.6.3 Stash and Clean

The `stash_and_clean()` operation preserves local work before switching or cleaning a workspace:

Provider	Steps
Git	`git status --porcelain` → `git stash push --include-untracked -m …`
Mercurial	`sase_hg_clean <diff_name>`

22.7 Configuration Reference

22.7.1 Full `sase.yml` Example

```
# ~/.config/sase/sase.yml

vcs_provider:
  provider: auto # "git", "hg", or "auto" (default: "auto")
  pr_tags: {} # optional TAG=value lines appended to PR commit messages
  use_project_pr_prefix: false # prepend [<project>] to PR titles / CL descriptions
```

22.7.2 Environment Variable

```
# Override VCS provider for a single command
SASE_VCS_PROVIDER=git sase commit my_feature

# Set for the entire shell session
export SASE_VCS_PROVIDER=hg
```

22.7.3 CLI Flags

Available on `sase ace` and `sase axe` only:

```
sase ace --vcs-provider git
sase ace --vcs-provider hg
sase ace --vcs-provider auto

sase axe start --vcs-provider git
```

Valid values for all three methods: `git`, `hg`, `auto`.

22.7.4 Schema

The `vcs_provider` section in `sase.yml` is validated against the schema at `~/.config/sase/sase.schema.json`:

```
{
  "vcs_provider": {
    "type": "object",
    "additionalProperties": false,
    "properties": {
      "provider": {
        "type": "string",
        "enum": ["git", "hg", "auto"],
        "default": "auto"
      },
      "workspace_root": {
        "type": "string"
      },
      "default_hooks": {
        "type": "array",
        "items": { "type": "string" }
      },
      "pr_tags": {
        "type": "object",
        "additionalProperties": { "type": "string" },
        "default": {}
      },
      "use_project_pr_prefix": {
        "type": "boolean",
        "default": false
      }
    }
  }
}
```

22.8 Classification Hooks

VCS provider detection is pluggable via two classification hooks:

22.8.1 `vcs_detect_repo_type`

Checks for VCS markers in a directory (e.g., `.hg/`, `.git/`). Each plugin checks for its own marker and returns the VCS type name (e.g., `"hg"`) or `None`. Used during auto-detection when walking up the directory tree.

22.8.2 `vcs_classify_repo`

For git repositories, further classifies by examining the remote URL. For example, the `sase-github` plugin claims repos with `github.com` URLs (returning `"github"`), while unclaimed repos fall through to the `"bare_git"` provider. This allows hosting-specific plugins to provide enhanced functionality (e.g., PR operations via `gh` CLI) without modifying the core.

22.9 Troubleshooting

22.9.1 "No VCS provider found" Error

Cause: Auto-detection could not find `.hg/` or `.git/` in the current directory or any parent, no explicit provider was configured, or a Git repo could not be classified because no plugin claimed it and `origin` was missing or unreadable.

Fix: Either run sase from within a VCS-managed directory, or set the provider explicitly:

```
SASE_VCS_PROVIDER=git sase commit my_feature
```

22.9.2 GitHub: `gh` CLI Not Installed

GitHub PR operations (`get_change_url`, `mail`, `get_cl_number`) require the GitHub CLI (https://cli.github.com/). Without it, these operations will fail with a "command not found" error.

Symptoms:

- `sase commit` completes but reports "Failed to retrieve change URL"
- `sase ace` mail action fails with "gh pr create failed"
- No PR URL shown after commit

Fix: Install the GitHub CLI and authenticate:

```
# macOS
brew install gh

# Then authenticate
gh auth login
```

22.9.3 Mercurial: Plugin Not Installed

Mercurial support requires an installed provider plugin. Without one, hg repositories will not be detected.

Symptoms:

- Auto-detection does not recognize `.hg/` directories
- "No VCS provider found" error in hg repositories

Fix: Install the maintained Mercurial provider plugin for your environment.

22.9.4 Mercurial: `sase_hg_*` Commands Not Found

The Mercurial provider depends on `sase_hg_*` wrapper commands. If these are not in your PATH, operations will fail.

Symptoms:

- "sase_hg_amend command not found"
- "sase_hg_sync command not found"
- Any core hg operation failing with "command not found"

Fix: Ensure your PATH includes the directory containing the `sase_hg_*` scripts.

22.9.5 Auto-Detection Picks Wrong Provider in Nested Repos

If you have nested repositories (e.g., a git repo inside an hg workspace), auto-detection walks up from the current directory and picks the **first** VCS directory it finds.

Example: If you're in `/workspace/git-repo/subdir/` and both `/workspace/.hg/` and `/workspace/git-repo/.git/` exist, auto-detection will find `.git/` first and use the Git provider.

Fix: Override the provider explicitly:

```
# Force Mercurial for this session
export SASE_VCS_PROVIDER=hg

# Or use config file
# ~/.config/sase/sase.yml
vcs_provider:
  provider: hg
```

23 Rust Backend (`sase_core_rs`)

A subset of sase's core APIs is served by a Rust extension distributed as `sase-core-rs` (https://pypi.org/project/sase-core-rs/) on PyPI and built from the sibling `sase-core` (https://github.com/sase-org/sase-core) repo. `sase` declares `sase-core-rs>=0.1.1,<0.2.0` as a hard runtime dependency, so a normal `pip install sase` (or `uv tool install sase`) on a supported platform pulls a prebuilt wheel automatically — no Rust toolchain required, no env-var selection, no Python fallback for ported operations.

The shipped Rust-backed operations are grouped by the Python facade that calls them:

- Project parsing: `parse_project_bytes`
- Project lifecycle helpers: read effective `PROJECT_STATE`, apply a lifecycle update to ProjectSpec text, and list lifecycle-filtered project records for CLI/TUI/launch discovery
- Query parsing and evaluation: `tokenize_query`, `parse_query`, `canonicalize_query`, legacy one-shot `evaluate_query_many`, and the product persistent-corpus path (`compile_corpus`, `compile_query`, `evaluate_many`) used by `sase.core.query_corpus_facade`
- Agent artifact scan/index operations: `scan_agent_artifacts`, `rebuild_agent_artifact_index`, `upsert_agent_artifact_index_row`, `delete_agent_artifact_index_row`, `query_agent_artifact_index`, and dismissed projection replacement for hiding dismissed identities in indexed visible-inbox queries
- Status and status-transition helpers: `read_status_from_lines`, `apply_status_update`, and `plan_status_transition`
- Git query parsers: `parse_git_name_status_z`, `parse_git_branch_name`, `derive_git_workspace_name`, `parse_git_conflicted_files`, and `parse_git_local_changes`
- Notification JSONL store operations: `read_notifications_snapshot`, `append_notification`, `apply_notification_state_update`, and `rewrite_notifications`
- Agent cleanup planning plus deterministic cleanup mutations: dismissed-identity index writes, artifact-marker deletion, workspace-release text mutation, and hook/mentor/comment kill marking. In the current ACE host path, dismissed-bundle JSON persistence and its summary SQLite index are Python-owned.
- Agent launch preparation, low-level detached spawn, timestamp allocation, fan-out planning, and RUNNING-field workspace-claim planning/mutation helpers
- Bead data operations: read queries (`show`, `list`, `ready`, `blocked`, `stats`, `doctor`, epic-child lookups), merged multi-workspace reads, mutations (`init`, `create`, `update`, `open`, `close`, `rm`, `dep add`, ready-to-work flags, sync-clean checks, compatibility projection export), deterministic epic/legend work planning, and the early `sase bead` CLI fast path for common read/write commands

The intentionally Python-owned facade surfaces (host logic, not backend fallbacks) are:

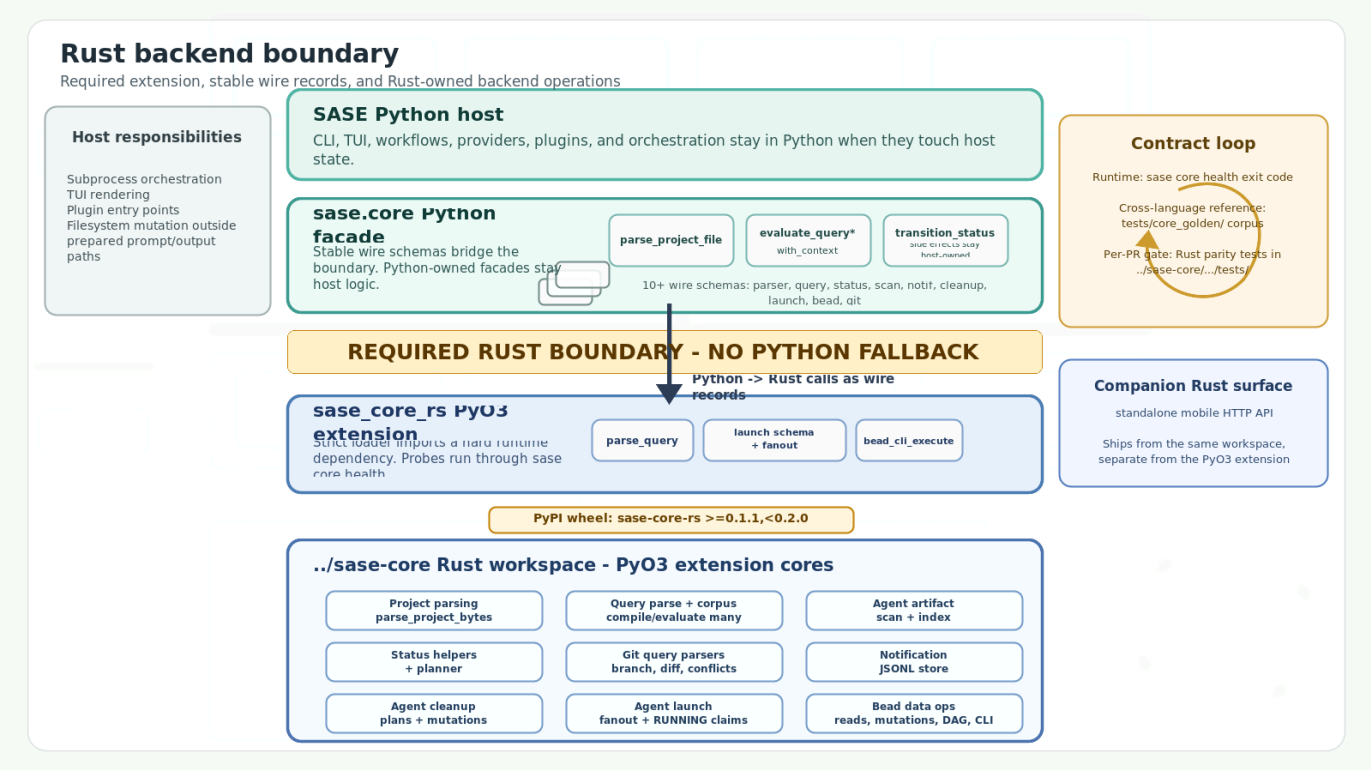
- `parse_project_file` — Python file-path API. The Rust binding consumes bytes; routing the file-path API through it would either re-read the file or duplicate the Python parser's tokenization for no measurable win.
- `build_query_context`, `evaluate_query`, `evaluate_query_with_context` — per-row query host logic. Batch product filtering uses a cached Rust query corpus; the public `evaluate_query_many(query, changespecs)` API remains as a compatibility wrapper that compiles a temporary Rust corpus for one call.
- `build_changespec_graph_index` — ChangeSpec graph index construction.
- `transition_changespec_status` — the side-effecting status transition (acquires a file lock, rewrites the project file, performs archive moves and suffix renames). The pure decision step inside it routes through Rust via `plan_status_transition`.
- Project lifecycle mutations stay on the Python host path: `sase project` resolves the mutable ProjectSpec file, holds the ProjectSpec lock, checks live `RUNNING` claims and artifact markers, and delegates only the pure `PROJECT_STATE` parse/update/list operations to Rust.
- High-level subprocess orchestration, process liveness checks outside launch, filesystem mutation outside the prepared prompt/output path, TUI rendering, and plugin entry points stay on the host by design.

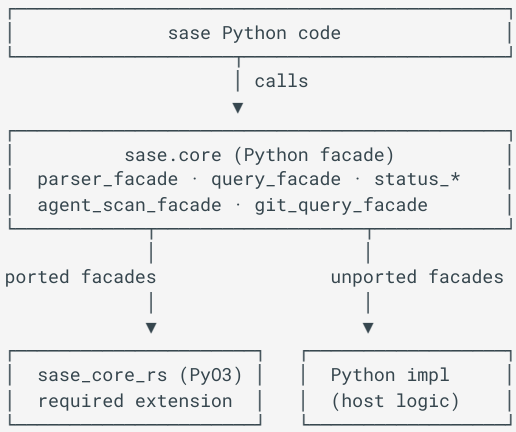
- Agent launch host responsibilities stay in Python: provider/workspace plugin calls, VCS preallocation env mapping, project-file locking, workspace-directory cleanup, TUI notifications, xprompt catalog expansion, history writes, chop registry recording, and user-facing launch callbacks. Rust owns deterministic launch planning/preparation and the low-level detached spawn binding.
- Agent cleanup process signalling, dismissed-bundle persistence, dismissed-bundle summary indexing, and TUI orchestration stay on the Python host path. The Rust boundary owns reusable cleanup planning, compact dismissed-identity writes, artifact deletion, workspace-release content rewrites, and ChangeSpec-entry kill marking exposed through Python helpers in `sase.core.agent_cleanup_*`.
- Bead host responsibilities stay in Python where they touch the surrounding application: storage-location discovery, SASE workspace/project lookup, VCS prompt context for `sase bead work`, xprompt resolution, user confirmation, agent launch, rollback of already-spawned children, and telemetry increments. Rust owns the bead data model, storage/query engine, JSONL codecs, mutation transactions, single-store ID allocation, deterministic work-plan DAG, and CLI output planning.
- Explicit agent artifact storage remains Python-owned because it copies or moves user files into `~/.sase/artifacts/` and updates the local JSONL association index under a file lock. Rust owns the separate agent-run artifact scanner and its persistent query index. Python owns best-effort lifecycle orchestration around that index: syncing dismissed-agent projection inputs before ACE loads, refreshing rows after marker mutations, and dispatching `sase agent index gc`.

23.1 Why a Rust Backend?

The `sase.core` package is a stable Python facade carved out specifically so individual operations can be re-served by faster Rust implementations one at a time. Parsing project `.sase` files dominates many cold-path workloads (TUI startup, large search results, axe lumberjack scans), so it was the first operation routed through this seam.

23.2 Architecture





The facade lives at `src/sase/core/`:

Module	Purpose
`rust.py`	Strict `sase_core_rs` loader (`require_rust_extension`, `require_rust_binding`)
`health.py`	`sase core health` Rust-extension probe + report
`parser_facade.py`	`parse_project_file` Python API + Rust-backed `parse_project_bytes`
`project_lifecycle_facade.py`	Rust-backed ProjectSpec lifecycle parse/update/list helpers
`project_lifecycle_wire.py`	Project lifecycle and project-record wire dataclasses

`wire.py`	Stable wire record types that cross the Python ↔ Rust boundary
`wire_conversion.py`	Python `ChangeSpec` ↔ wire record serialization
`query_facade.py`	`parse_query` (Rust); per-row query context/eval (Python host logic); batch compatibility wrapper over Rust corpus
`query_corpus_facade.py`	Persistent Rust query corpus wrapper for cached batch evaluation
`notification_store_facade.py`	Notification JSONL snapshot, append, rewrite, and state mutation facade (Rust)
`notification_store_wire.py`	Stable notification snapshot/update wire records across the Rust boundary
`status_facade.py`	Status line helpers + planner (Rust); side-effecting transition (Python host logic)
`graph_index_facade.py`	`build_changespec_graph_index()` facade (Python host logic)
`agent_scan_facade.py`	Agent artifact scan plus persistent index query/rebuild/update/delete facade (Rust)
`agent_scan_wire.py`	Stable wire records for agent-artifact scans and index maintenance
`agent_artifact_facade.py`	Compatibility import surface for explicit/default agent-artifact helpers (Python host logic)
`agent_cleanup_wire.py`	Stable cleanup planning and side-effect intent wires
`agent_cleanup_facade.py`	Agent cleanup target conversion and `plan_agent_cleanup()` facade
`agent_cleanup_execution.py`	Host-safe wrappers for Rust-backed deterministic cleanup mutations
`agent_launch_wire.py`	Stable launch, workspace-claim, and fan-out wire records
`agent_launch_facade.py`	Rust-backed launch preparation, spawn, timestamp allocation, and fan-out planning
`agent_launch_claims.py`	Rust-backed RUNNING-field claim planning/mutation helpers
`bead_read_facade.py`	Rust-backed bead read facade for one active bead store
`bead_mutation_facade.py`	Rust-backed bead mutation facade
`bead_wire.py`	Stable bead issue/dependency conversion helpers across the Rust boundary
`status_wire.py`	Stable wire records for the status state machine
`status_wire_conversion.py`	Python plan reference + project-file → request-wire converter
`git_query_facade.py`	Pure Git query parsers facade (Rust)
`git_query_wire.py`	Stable wire records for the Git query parsers

The Rust extension is a sibling repo at `../sase-core/`, organized as a Cargo workspace with a PyO3 crate at `crates/sase_core_py/`.

23.3 Mobile Gateway

The mobile gateway is also built from the sibling `../sase-core/` workspace, but it is a standalone Rust HTTP server rather than a PyO3 binding. The `crates/sase_gateway` crate owns the host gateway's wire records, pairing/token store, bind policy, authenticated session route, SSE event stream, audit log, and committed mobile API contract snapshot.

The Python repo owns user-facing startup through `sase mobile gateway start`, configuration defaults, and lifecycle glue. See `docs/mobile_gateway.md` (https://sase.sh/mobile_gateway/) for local setup, pairing, Tailscale Serve guidance, security notes, and the contract snapshot path used by future Android work.

23.4 Bead Backend

The `sase bead` path is Rust-owned for data operations. Canonical bead state lives in append-only event streams under `sdd/beads/events/**` when present; `issues.jsonl` is regenerated as a compatibility projection, and `beads.db` is a compatibility cache. Event reduction, JSONL/config parsing, cache refresh, mutations, single-store ID allocation, deterministic epic work planning, and common CLI output planning all live in `sase-core` and are exposed through `sase_core_rs`. Python remains the host layer for path discovery, VCS context, xprompt lookup, confirmation prompts, launch/rollback, and telemetry side effects.

Golden contract fixtures live under `tests/test_bead/golden/`:

- `cli/` pins stdout/stderr for `init`, `create`, `list`, `show`, `ready`, `blocked`, `stats`, `dep add`, `update`, `open`, `close`, `rm`, `sync --status`, and representative error paths.
- `jsonl/` pins current and legacy JSONL shapes, corrupt-line tolerance, empty/missing import behavior, hierarchy, dependencies, cross-epic blockers, and ChangeSpec metadata.
- `stores/current/` is a complete deterministic bead store used by the CLI golden tests.
- Rust parity fixtures in `../sase-core/crates/sase_core/tests/fixtures/bead/` pin legacy JSONL import, deterministic event migration, event-backed reads, and regenerated projection behavior.

Run the focused contract tests with:

```
pytest tests/test_bead/test_cli_golden.py tests/test_bead/test_jsonl_golden_fixtures.py
cargo test --workspace bead
```

The reproducible bead benchmark harness is:

```
python tests/perf/bench_bead.py --runs 5 --output /tmp/bead-bench.json
python tests/perf/bench_bead.py --runs 5 --issues 10000 --dependencies 20000 --output /tmp/bead-bench-large.json
```

By default the shell measurements run `python -m sase.main.entry`; pass `--sase-bin "$(command -v sase)"` to measure an installed console script. The harness reports JSON summaries for:

- shell command latency for `sase bead list`, `ready`, and `show`;
- direct Python `BeadProject` reads;
- synthetic stores sized by `--issues` and `--dependencies`.

The Phase A local baseline on the then-active 399-line store was approximately:

Command / action	Baseline
`sase bead list`	0.32s
`sase bead ready`	0.34s
`sase bead show <id>`	0.36s

Plain Python startup	0.08s
Importing `sase.main.entry`	0.23s
Importing `sase_core_rs`	0.02s

Post-migration targets for bead checks and future regression floors:

- `sase bead list`, `ready`, and `show` on the current-size store: p50 under 120ms from shell command start.
- Direct Rust read bindings after Python startup: p50 under 10ms.
- Large synthetic store, 10k issues / 20k dependencies: direct Rust read queries under 50ms p50, and write plus JSONL export under 150ms p50.
- No drift in the Phase A golden CLI output unless the migration plan records an intentional compatibility change.

23.5 Installing the Rust Backend

23.5.1 Released `sase` (recommended for users)

`sase-core-rs` is a regular runtime dependency of `sase`. A standard install pulls a prebuilt wheel for the host platform from PyPI; no Rust toolchain is needed:

```
pip install sase
# or
uv tool install sase
```

The release matrix ships wheels for CPython 3.12+ on Linux x86_64, Linux aarch64, macOS universal2, and Windows x86_64. After install, `python -c "import sase_core_rs"` succeeds inside the same venv that runs `sase`.

23.5.2 Source / development workflow

`just install` automatically builds and installs `sase_core_rs` from a sibling `../sase-core` checkout when one exists and a Rust toolchain (`cargo`) is on `PATH`. This satisfies the `sase-core-rs` runtime dependency from local source so the editable `sase` install does not have to round-trip through PyPI:

```
git clone https://github.com/sase-org/sase-core.git ../sase-core
just install     # builds sase_core_rs from ../sase-core, then installs sase in editable mode
```

Docs-only commands do not need the application package or the Rust extension. `just docs-check` and `just docs-pdf-check` install only MkDocs tooling into `.venv`, which is why documentation CI can run without checking out `../sase-core`.

`just rust-install` remains the explicit way to (re)build only the extension, and `just rust-install-uv-tool` targets the uv-tool venv at `$(uv tool dir)/sase` for users who installed `sase` via `uv tool install` and want the latest local Rust code instead of the published wheel:

```
just rust-install                # repo .venv (used by `just test`, benchmarks)
just rust-install-uv-tool        # $(uv tool dir)/sase
just rust-install /path/to/venv  # any other venv (pipx, system Python, custom location)
```

Both targets install `maturin` into the target venv on demand and run `maturin develop --release` inside `../sase-core/crates/sase_core_py/`, so re-running them after a `../sase-core` update is the supported way to refresh an existing source install.

23.5.3 No pure-Python fallback

A working `sase` install requires a loadable `sase_core_rs` extension. There is no `SASE_CORE_BACKEND` env var, no Python escape hatch for ported operations, and no silent fallback when the wheel is missing. A misbuilt or absent extension fails fast: ported facades raise `ImportError` from `sase.core.rust.require_rust_extension`, or `AttributeError` from `require_rust_binding` when the wheel is too old to expose the requested binding. `sase core health` exits non-zero in either case.

If a contributor's local checkout does not have a working `sase_core_rs`, the fix is to run `just install` (or `just rust-install` against a sibling `../sase-core/`) — not to disable Rust.

23.6 Backend Health Check

`sase core health` is the scriptable answer to "is the Rust extension loadable and working?". It imports `sase_core_rs`, calls cheap parser, launch, and bead probes (`parse_query("status:Ready")`, `agent_launch_wire_schema_version()`, `plan_agent_launch_fanout(...)`, and a temporary-store `bead_cli_execute(["show", ...])`), and reports module path / version / Python version / platform tag in one block. Two output modes:

```
sase core health        # human-readable, line-oriented
sase core health -j     # machine-readable JSON (alias: --json)
```

Exit codes:

Extension state	`status`	Exit
importable, parser + launch + bead probes work	`ok`	0
missing or misbuilt	`error`	1
importable but a parser/launch/bead probe fails	`error`	1
importable but missing a representative binding	`error`	1

A misbuilt wheel that fails to import with a non-`ImportError` is surfaced verbatim in the `error` / `error_kind` fields rather than silently masked.

Release jobs and CI install-smokes call `sase core health` instead of probing `import sase_core_rs` and a binding by hand: it is the same check, but its exit code is the contract.

23.7 Runtime Version Inventory

`sase version` complements `sase core health` by answering "which local SASE packages is this process actually using?" It reports the host `sase` distribution, the required `sase-core-rs` distribution, and installed SASE plugin packages, including entry-point plugins and script-only plugin packages.

```
sase version            # human-readable runtime/package inventory
sase version -v         # add install, source, git, and plugin-signal audit fields
sase version -j         # stable JSON payload for support/debug tooling
```

The command is local-only: it does not check latest available releases. Editable development installs prefer source metadata and git state over stale installed distribution metadata, so a checkout after tag `v0.1.2` may display a PEP 440 local version such as `0.1.2+4.g26c39e004`. Verbose and JSON output keep the installed distribution version and the source version side by side so stale editable metadata is visible.

23.8 Justfile Targets

Each target prints a friendly skip message when `../sase-core` is absent and exits 0, so contributors without the sibling checkout are never blocked.

Target	Description
`just rust-install`	Build + install `sase_core_rs` via `maturin develop --release` (installs maturin if missing)
`just rust-install-uv-tool`	Same as `rust-install` but targets `$(uv tool dir)/sase` for users who installed sase via `uv tool install`
`just rust-test`	`cargo test --workspace` in `../sase-core`
`just rust-fmt`	Auto-format Rust sources with `cargo fmt --all`
`just rust-fmt-check`	CI-mode formatting verification (`cargo fmt --all -- --check`)
`just rust-clippy`	`cargo clippy --workspace --all-targets -- -D warnings`
`just rust-check`	Combined Rust check: `rust-fmt-check` + `rust-clippy` + `rust-test`
`just rust-bench`	Run the direct-parser Rust benchmark (`cargo run --release --example bench_parse`)
`just bench-core`	Python `parse_project_bytes` benchmark (Rust-direct + facade rows)
`just bead-perf-smoke`	Tiny `sase bead` shell/facade/work-plan benchmark used as the CI smoke artifact
`just bench-agent-scan`	Python agent-artifact scan benchmark vs current direct loaders
`just bench-agent-launch`	Fake-spawn launch benchmark through the Rust preparation binding
`just launch-perf-check`	CI-friendly launch regression check against the Phase 1 fan-out baseline
`just phase7-perf-check`	Run the Phase 7 regression-floor checker against the recorded Rust ceilings

23.9 Performance

Phase 7 captured a deliberate measurement pass after the Rust default flip; Phase 8 then deleted the Python halves of the ported operations, so the historical Python comparisons are frozen evidence rather than live measurements. The raw JSON artifacts live under `sdd/tales/202604/perf_artifacts/`; the tables below summarize the medians a reader should expect when running the same harnesses against the same Rust extension.

23.9.1 Workstation profile

All numbers below come from a single capture machine (Phase 7B + 7C, 2026-04-29):

- Linux x86_64, CPython 3.14.3.

- `sase-core-rs` editable install built from a sibling `../sase-core/` checkout via `just rust-install`; metadata in every artifact's `metadata.rust_module_path` / `metadata.rust_module_version` records the exact extension probed.
- Sample sizes per scenario are recorded inline below and pinned in each artifact's `metadata.runs` / `metadata.warmup`. End-to-end TUI/CLI runs are 10–12 samples; microbenchmarks are 20–200 samples per scenario.

The historical `python_median` columns are preserved as Phase 7B baselines — they are no longer reproducible from a post-Phase 8 install (the Python halves are gone) but remain useful for understanding why each operation was kept on Rust. `speedup` reads `python_median / rust_median` against those frozen Python numbers.

23.9.2 Core operations (Phase 7B microbenchmarks)

Driver: `tests/perf/phase7/run_phase7b.py`. One `*_summary.json` artifact per shipped operation under `sdd/tales/202604/perf_artifacts/rust_backend_phase7_<op>_summary.json`; each artifact embeds the Phase 7A `Phase7Metadata` envelope, the relevant scenario summaries, and pre-computed `(workload, scenario)` comparison rows.

Operation	Workload	Scenario	py median (Phase 7B)	rust median	speedup
`parse_project_bytes`	golden_myproj	facade	296 µs	124 µs	**2.4×**
`parse_project_bytes`	synthetic_200_specs	facade	26.7 ms	19.1 ms	**1.4×**
`parse_query`	parse_only	direct	12.3 µs	5.8 µs	**2.1×**
`scan_agent_artifacts`	synthetic_6p_200pp	scan_facade	145 ms	120 ms	**1.21×**
`read_status_from_lines`	synthetic_200_specs_pure	read_status_from_lines	182 µs	349 µs	0.52×
`apply_status_update`	synthetic_200_specs_pure	apply_status_update	256 µs	377 µs	0.68×
`plan_status_transition`	synthetic_200_specs_pure	plan_status_transition	8.66 µs	18.78 µs	0.46×
`parse_git_name_status_z`	synthetic_medium (1k)	parse_git_name_status_z	626 µs	880 µs	0.71×
`parse_git_branch_name`	normalizers_x4	parse_git_branch_name	8.74 µs	10.40 µs	0.84×
`derive_git_workspace_name`	normalizers_x5	derive_git_workspace_name	11.7 µs	13.5 µs	0.87×
`parse_git_conflicted_files`	normalizers_50_lines	parse_git_conflicted_files	5.35 µs	6.83 µs	0.78×
`parse_git_local_changes`	normalizers_150_entries	parse_git_local_changes	4.51 µs	5.68 µs	0.79×

The historical one-shot Rust `evaluate_query_many(query, dicts)` binding remains a non-product diagnostic row only. The shipped product route uses a persistent Rust query corpus: compile `ChangeSpec` wire records once per stable list object, then compile/evaluate each query string against that cached corpus. Query-corpus Phase 6 measured the product query-keystroke path at 37-74x faster than the Python batch reference on the synthetic workloads and added the `synthetic_1000_specs` persistent query-keystroke row to the regression floor.

The full per-percentile data (min / median / p95 / max) is in each artifact's `workloads[].baseline` / `workloads[].candidate`; the `comparisons[]` rows pre-compute `ratio`, `speedup`, and `percent_delta` for every `(workload, scenario)` pair.

23.9.3 End-to-end TUI / CLI surfaces (Phase 7C)

Driver: `tests/perf/bench_phase7_e2e.py`. One artifact per `(surface, backend)` invocation under `sdd/tales/202604/perf_artifacts/rust_backend_phase7_<surface>_<backend>.json`; the home-tree `sase agent list` rows sit in the gitignored `sdd/tales/202604/perf_artifacts/local_only/` dir because they reflect a workstation-specific tree.

Surface	Workload	runs	rust	python (Phase 7B)	speedup
`sase_run_startup`	`import_run_query_cold`	12	249.3 ms	252.6 ms	1.01×
`sase_agents_status_listing`	`synthetic_8_projects_25_agents`	12	298.5 ms	774.5 ms	**2.59×**
`sase_agents_status_listing`	`home_tree` (local-only artifact)	5	885.8 ms	1,799.7 ms	**2.03×**
`sase_ace_cold_open`	`synthetic_100_cs_50_agents`	10	1,472 ms	1,237 ms	0.84×

`sase_run_startup` measures cold subprocess `python -c "from sase.main.query_handler._query import run_query"`; it deliberately stops at the dispatcher's provider boundary, never resolves a provider, never touches the network, and never claims a workspace. The `metadata.extra.boundary` field in the artifact records this scope so a future agent can push the boundary further toward provider resolution without invalidating the comparison.

23.9.4 Agent launch migration (Phase 9)

Driver: `tests/perf/bench_agent_launch.py`; regression check: `tests/perf/check_agent_launch_regression.py`. The harness uses temp ProjectSpec files and fake subprocess writes so it never starts an LLM CLI, but it now runs launch preparation through the production Rust binding. The committed Phase 1 baseline is `sdd/tales/202605/perf_artifacts/agent_launch_phase1_baseline.json`.

The Phase 1 baseline intentionally includes parent-side fan-out sleeps: three-way `%model` and `%r` launches each spent about 2,001 ms in the parent before the migration. `just launch-perf-check` runs the current harness without those sleeps and fails if `model_fanout` or `repeat_fanout` exceeds 25% of the Phase 1 median. Single-prompt, VCS, and deferred-workspace fake launches also have a generous 100 ms median ceiling so the gate catches accidental blocking work without depending on a specific workstation's sub-millisecond numbers.

23.9.5 Where Rust helps, where it does not

Wins:

- `sase agent list -j` cold listing is the headline Rust win — ~2.6× on the synthetic 8×25 tree and ~2.0× on this workstation's home tree. The cold subprocess wall-time is dominated by `scan_agent_artifacts`, which Rust ports.
- `parse_project_bytes` is a clean ~2.4× win on small files and ~1.4× on a 200-spec synthetic file.

- `parse_query` direct parsing is a ~2.1× win on the parse-only workload.

Honest negatives (kept on Rust for shared-core hygiene rather than user-perceived latency):

- The status-line helpers (`read_status_from_lines` , `apply_status_update` , `plan_status_transition`) and the small Git normalizers (`parse_git_branch_name` , `derive_git_workspace_name` , `parse_git_conflicted_files` , `parse_git_local_changes`) are 13–55% slower under Rust than the historical Python implementations on the inputs they actually see in production. These are dispatch-overhead-dominated cores at sub-10-µs absolute cost; the gap is single-digit microseconds and is invisible against the surrounding subprocess / atomic-write cost.
- `parse_git_name_status_z` is consistently ~25–30% slower than the historical Python on synthetic streams, but the end-to-end `git diff --name-status -z` workloads in `bench_git_query_ops` show parse is single-digit microseconds next to multi-millisecond subprocess cost.
- `sase ace` cold open is ~19% slower under Rust on the synthetic Pilot harness. The harness mocks `find_all_changespecs` , so the Rust scan/parse hot paths are not exercised; what remains is AceApp / Pilot constructor cost plus per-call PyO3 dispatch overhead at small inputs. Treat it as a known small-input dispatch tax, not a routed-op regression.
- `sase run` startup is dispatch-neutral at the cold-import scope: the cost users pay before the dispatcher can call any LLM is dominated by Python interpreter startup + sase package import.

23.9.6 Performance regression floor

`tests/perf/baselines/phase7_regression_floor.json` pins absolute Rust ceilings for the anchors that matter (`golden_myproj` and `synthetic_200_specs` for `parse_project_bytes` , `parse_only` for `parse_query` , `synthetic_6p_200pp` for `scan_agent_artifacts` , `golden_myproj_pure` for `apply_status_update` , the `synthetic_1000_specs` persistent query-corpus product route, and the synthetic 5k notification-store snapshot/mutation routes). The relative `must_beat_python` check is disabled for anchors whose Python halves were deleted in Phase 8D — only the absolute Rust ceiling stays in force. `parse_query.parse_only.direct` and the persistent query-corpus product route keep `must_beat_python: true` because both comparable rows are still produced by the current harnesses. The CI `phase7-perf-floor` GitHub Actions job runs the checker (`tests/perf/phase7_check_regression.py`) on every PR and uploads `rust_backend_phase7_floor_check.json` as the build artifact.

`sdd/tales/202605/perf_artifacts/agent_launch_phase1_baseline.json` pins the launch migration baseline. The `launch-perf-floor` GitHub Actions job runs `just launch-perf-check` on every PR and uploads `agent_launch_regression_check.json` so a fan-out latency regression has a comparable report.

23.9.7 Triage support note

When investigating a Rust-extension issue:

- **Confirm the extension is loaded.** `sase core health` (or `sase core health -j` for scripts) prints the `sase_core_rs` module path / version and the result of cheap parser, launch, and bead binding probes. Exit code 0 means the extension loaded and worked; non-zero means the wheel is missing, stale, or misbuilt.

- **Recognise a wheel-load failure.** A missing or stale extension surfaces as `ImportError` / `AttributeError` from a shipped operation, or as `sase core health` exit code 1 with `error_kind` / `error` fields naming the underlying import error. The publish-workflow `install-smoke` runs `sase core health` on every release and dumps `pip list` plus `sase_core_rs.__file__` / `__version__` on failure.
- **There is no env-var escape hatch.** If `sase_core_rs` is broken, the user-facing fix is reinstalling sase or pinning to a known-good `sase-core-rs` version, not setting an env var. See the Rollback section below.

23.10 Verifying The Backend

A handful of commands cover "is my install healthy?" end-to-end:

```
sase core health             # Rust health: status + module path + version + platform
sase core health -j          # same, JSON for scripting
sase version                 # local host/core/plugin package inventory
sase version -j              # same, JSON for support/debug tooling

just check                   # formatting, lint, SDD validation, and tests
just rust-check              # cargo fmt --check + clippy + cargo test (requires sibling ../sase-core checkout)
just bead-perf-smoke         # tiny Rust-backed bead shell/facade/work-plan benchmark
just launch-perf-check       # launch fan-out regression floor against the Phase 1 baseline
just phase7-perf-check       # Phase 7 regression-floor check against the recorded Rust ceilings
```

CI runs the full test suite under CPython 3.12 / 3.13 / 3.14 (`.github/workflows/ci.yml`). The dedicated `bead-backend` job checks the sibling Rust core (`just rust-check`), focused Python bead tests, cross-repo bead facade parity tests, and `just bead-perf-smoke` . The publish workflow's `install-smoke` job installs the built `sase` wheel into a fresh venv and runs `sase core health` ; on failure it dumps `pip list` , Python/platform info, and `sase_core_rs.__file__` / `__version__` so missing-wheel or ABI-mismatch failures are diagnosable from the build log without a manual repro.

23.11 Golden Contract

After Phase 8 the Python halves of the ported operations are gone, so the compatibility seam is the *golden corpus* rather than a live Python/Rust dual-run comparison. The corpus pins the Rust extension's expected output byte-for-byte across parser, query, agent scan, status, and Git query helpers:

Surface	Tests
ChangeSpec parser	`tests/test_core_golden.py` , `tests/test_core_wire.py` , `tests/test_core_facade/test_parser.py`
Query parse / canonical form	`tests/test_core_query_golden_*` (errors / eval / tokens / wire), `tests/test_core_facade/test_query.py`
Agent artifact scan	`tests/test_core_agent_scan.py` + `tests/agent_scan_golden/` fixture builder
Notification store	`tests/test_core_notification_store.py` , `tests/test_core_facade/test_notification_store.py`
Status helpers + planner	`tests/test_core_facade/test_status.py` , `tests/test_core_status_lines.py` , `tests/test_core_status_wire.py`
Git query parsers	`tests/test_core_git_query.py`
Agent launch	`tests/test_core_agent_launch_wire.py` , `tests/test_agent_launch_executor.py` , `tests/perf/test_agent_launch_regression.py`
Beads	`tests/test_bead/` , `tests/test_core_facade/test_bead_*.py` , `../sase-core/crates/sase_core/tests/bead_*`

Strict-loader contract	`tests/test_core_rust.py`, `tests/test_core_health.py`

The `tests/core_golden/` corpus (`myproj.sase`, `myproj-archive.sase`) plus the `inline_snapshot` JSON expectations in `test_core_golden.py` are the cross-language reference: any change to the Rust output that breaks a snapshot must be matched by an equivalent change in the corresponding `sase-core` Rust parity test (`../sase-core/.../tests/`) before either side ships.

23.12 Rollback

After Phase 8 the rollback model is **wheel/package fix, not env-var workaround**. There is no `SASE_CORE_BACKEND` escape hatch, no Python implementation to fall back to for ported operations, and no per-user mitigation that bypasses Rust.

- A Rust-side regression is fixed and re-released as a `sase-core-rs` patch version that `sase` depends on. The pinned range is updated in `pyproject.toml` and a `sase` patch release pulls the corrected wheel.
- For a regression that drifted before Phase 8 closed, the only safe path is to revert the Phase 8 PR(s) that removed the Python halves, ship a patch release that restores the Python implementations, then redo verification before re-attempting the deletion.

If a user reports a `sase core health` failure post-release, the support workflow is:

1. Verify the installed `sase-core-rs` version (`pip show sase-core-rs` or the JSON output of `sase core health -j`).
2. Reinstall: `pip install --force-reinstall sase` (or `uv tool install --force sase`) to repull the wheel.
3. If the wheel itself is broken on the user's platform, pin to the previous `sase-core-rs` version and file a bug in `../sase-core` with the `sase core health -j` output, the platform tag, and the failing binding.

The Phase 6/7 release-cycle artefacts (`SASE_CORE_BACKEND=python` escape hatch, dual-run JSONL, `parity-gate` job) are deleted; do not reach for them when triaging post-Phase-8 issues.

23.13 Migration History

The migration ran across nine phases. Phases 0–7 added the Rust backend behind a default-Python escape hatch and the parity gate; Phase 8 deleted the dispatcher, the dual-run plumbing, and the Python halves of every ported operation that did not need them as host logic. The full per-phase narrative lives in `sdd/research/202604/rust_backend_migration.md` and `sdd/epics/202604/rust_backend_phase{0..8}*.md`. The handoffs that record each subphase's changes are alongside their plan files (`sdd/epics/202604/rust_backend_phase8_phase8{a..g}_handoff.md`).

CLI Reference

This page is a command index for the top-level `sase` CLI. It is meant for discovery and routing: use it to find the surface that owns a workflow, then follow the links to the detailed command, flag, or subsystem reference.

For exhaustive flag tables, see the configuration reference (https://sase.sh/configuration/#cli-flags).

Daily Operation

Command	Purpose	Details
`sase ace`	Open ACE, the interactive control surface for ChangeSpecs, live agents, notifications, and axe state.	ACE TUI (https://sase.sh/ace/)
`sase run [PROMPT]`	Launch an agent or workflow from a prompt, an xprompt reference, a workflow reference, history, or an editor buffer.	XPrompts (https://sase.sh/xprompt/), workflows (https://sase.sh/workflow_spec/)
`sase agent list`	List active and recent agents across projects.	ACE Agents tab (https://sase.sh/ace/#tab-system)
`sase agent show`	Render one agent's detail panel by name.	Agent attachments (https://sase.sh/agent_images/)
`sase agent kill`	Terminate a running agent.	ACE TUI (https://sase.sh/ace/)
`sase agent tag`	Set, clear, or list user-defined agent tags used for grouping.	ACE TUI (https://sase.sh/ace/)
`sase agent archive`	Maintain dismissed-agent bundle summary indexes (`rebuild-index`, `verify`).	ACE TUI (https://sase.sh/ace/#agent-revival)
`sase agent artifacts`	Inspect and migrate physical agent artifact storage layout.	Configuration (https://sase.sh/configuration/#directory-sharding)
`sase agent index`	Manage the persistent agent artifact SQLite index (`status`, `rebuild`, `verify`, `gc`).	ACE TUI (https://sase.sh/ace/)
`sase agent names migrate-auto`	Backfill the permanent agent-name registry from legacy auto-generated names; pass `--force` to rerun.	ACE TUI (https://sase.sh/ace/)
`sase chat list`	List recent chat transcripts.	XPrompts (https://sase.sh/xprompt/)
`sase chat show`	Show one chat transcript by agent name, path, or basename.	XPrompts (https://sase.sh/xprompt/)
`sase prompt list`	List, search, and filter previously submitted prompts (pretty table or JSON).	Prompt history (https://sase.sh/prompt/)
`sase prompt show`	Print one prompt's exact text as raw, Markdown, or JSON.	Prompt history (https://sase.sh/prompt/)
`sase prompt run`	Replay a stored prompt by selector, optionally editing or re-prefixing it first.	Prompt history (https://sase.sh/prompt/)
`sase prompt save`	Save a stored prompt as a reusable xprompt, or `export` it to a file or SDD snapshot.	Prompt history (https://sase.sh/prompt/), XPrompts (https://sase.sh/xprompt/)
`sase prompt prune`	Curate the prompt-history store with `delete`, `prune`, and read-only `doctor` / `stats`.	Prompt history (https://sase.sh/prompt/)
`sase notify`	Shortcut for `sase notify list`.	Notifications (https://sase.sh/notifications/)
`sase notify create`	Create a notification from JSON input.	Notifications (https://sase.sh/notifications/)
`sase notify list`	List recent notifications, optionally filtered by sender, tag, unread state, or query.	Notifications (https://sase.sh/notifications/)
`sase notify show`	Show one notification as Markdown or JSON.	Notifications (https://sase.sh/notifications/)
`sase repro replay`	Replay an Agents-tab reproduction bundle through the headless TUI harness and emit a verdict.	ACE TUI (https://sase.sh/ace/#agents-tab-reproduction-bundles)
`sase repro capture agents-tab`	Capture a commit-safe out-of-band Agents-tab bundle from current filesystem state.	ACE TUI (https://sase.sh/ace/#agents-tab-reproduction-bundles)

`sase run` can run in the foreground, launch detached background agents with `--daemon`, resume previous conversations, or expand multi-prompt input into sequential background launches. ACE uses the same launch machinery when users start agents from the TUI.

Command groups with an exact `list` child default to that list view when invoked bare, including `sase amd`, `sase bead`, `sase chat`, `sase file`, `sase file-history`, `sase memory`, `sase notify`, `sase plugin`, `sase project`, `sase prompt`, `sase sdd`, `sase skill`, `sase telemetry`, `sase workspace`, and `sase xprompt`. Nested groups such as `sase agent tag`, `sase axe chop`, and `sase axe lumberjack` follow the same rule.

The bare form is only the default view. When you need flags that belong to the list command, keep the `list` subcommand explicit, for example `sase notify list -j`, `sase memory list -j`, or `sase workspace list --json`.

Work Tracking And Planning

Command	Purpose	Details
`sase changespec current`	Render the ChangeSpec associated with the current workspace.	ChangeSpecs (https://sase.sh/change_spec/)
`sase changespec migrate-extension`	Rename legacy `.gp` ProjectSpec files to the canonical `.sase` extension.	ProjectSpec (https://sase.sh/project_spec/)
`sase changespec search`	Search and filter ChangeSpecs with the query language.	Query language (https://sase.sh/query_language/)
`sase changespec sync-deltas`	Recompute the `DELTAS` field for a ChangeSpec from VCS state.	ChangeSpecs (https://sase.sh/change_spec/)
`sase init`	Check and initialize AMD, memory, SDD, and skills from one coordinator.	Initialization (https://sase.sh/init/)
`sase amd` / `sase amd list`	Inventory project, home, and chezmoi `AGENTS.md` files plus nearby provider shims.	Initialization (https://sase.sh/init/#agent-markdown-documents)
`sase amd init`	Create or refresh `AGENTS.md` files and provider shims for the selected AMD root or roots.	Initialization (https://sase.sh/init/#agent-markdown-documents)
`sase init amd`	Alias for `sase amd init`.	Initialization (https://sase.sh/init/#agent-markdown-documents)
`sase memory` / `sase memory list`	Show loaded, referenced, available, and missing memory files.	Memory (https://sase.sh/memory/#inspect-context)
`sase memory read`	Agent-side read of one long-term memory file with an attributable audit event.	Memory (https://sase.sh/memory/#audited-reads)
`sase memory write`	Agent-side proposal for human-reviewed long-term memory; `--notify` can add an inbox item.	Memory (https://sase.sh/memory/#propose-memory)
`sase memory review`	Human listing, inspection, approval, editing, or rejection of pending memory proposals.	Memory (https://sase.sh/memory/#review-proposals)

`sase memory log`	Summarize audited memory reads; `--include proposals` also shows proposal and review events.	Memory (https://sase.sh/memory/#audited-reads)
`sase memory init`	Create or refresh project/home memory files and AGENTS memory references.	Initialization (https://sase.sh/init/#memory-initialization)
`sase init memory`	Alias for `sase memory init`.	Initialization (https://sase.sh/init/#memory-initialization)
`sase sdd init`	Enable version-controlled SDD and refresh generated guide files.	SDD (https://sase.sh/sdd/)
`sase init sdd`	Alias for `sase sdd init`.	SDD (https://sase.sh/sdd/)
`sase sdd list`	List SDD prompt, tale, epic, legend, or all Markdown artifacts.	SDD (https://sase.sh/sdd/)
`sase sdd links`	Inspect prompt/artifact frontmatter links.	SDD (https://sase.sh/sdd/)
`sase sdd validate`	Validate SDD frontmatter links.	SDD (https://sase.sh/sdd/)
`sase sdd repair-links`	Infer and optionally write missing bidirectional SDD links.	SDD (https://sase.sh/sdd/)
`sase bead onboard`	Print the bead quick-start guide.	Beads (https://sase.sh/beads/)
`sase bead init`	Initialize bead storage for the current project.	Beads (https://sase.sh/beads/#storage)
`sase bead create`	Create plan, epic, legend, or phase issues.	Beads (https://sase.sh/beads/#cli-commands)
`sase bead list`	List bead issues by status, type, or tier.	Beads (https://sase.sh/beads/#cli-commands)
`sase bead search`	Search bead IDs, titles, notes, plan paths, metadata, and lifecycle fields.	Beads (https://sase.sh/beads/#sase-bead-search-query)
`sase bead ready`	Show open issues whose dependencies are closed.	Beads (https://sase.sh/beads/#dependencies)
`sase bead blocked`	Show issues blocked by open dependencies.	Beads (https://sase.sh/beads/#dependencies)
`sase bead show`	Show one issue.	Beads (https://sase.sh/beads/#cli-commands)
`sase bead update` / `open` / `close` / `rm`	Mutate issue metadata or lifecycle state.	Beads (https://sase.sh/beads/#cli-commands)
`sase bead dep add`	Add an issue dependency.	Beads (https://sase.sh/beads/#dependencies)
`sase bead sync`	Export the bead database to git-tracked JSONL and stage it.	Beads (https://sase.sh/beads/#sync-mechanism)
`sase bead stats` / `doctor`	Inspect project statistics or bead-store health.	Beads (https://sase.sh/beads/#rust-backend)
`sase bead work`	Launch phase agents for an epic, or epic-planning agents for a legend.	Beads (https://sase.sh/beads/#sase-bead-work-id)
`sase project list`	List active projects by default, or include hidden lifecycle states with `--state`.	Project lifecycle (https://sase.sh/project_spec/#project-lifecycle)

`sase project show`	Show lifecycle, workspace, launchability, and warning details for one project.	Project lifecycle (https://sase.sh/project_spec/#project-lifecycle)
`sase project set-state` / aliases	Update `PROJECT_STATE` under the ProjectSpec lock.	Project lifecycle (https://sase.sh/project_spec/#project-lifecycle)
`sase project alias`	List, add, remove, or clear `PROJECT_ALIASES` under the ProjectSpec lock.	Project aliases (https://sase.sh/project_spec/#project-aliases)
`sase plan` / `sase plan list`	Show the plan pipeline dashboard: pending proposals, recent approvals, and inferred rejects.	XPrompt directives (https://sase.sh/xprompt/#plan-directive)
`sase plan approve`	Approve one pending plan by ID or prefix; `--kind` chooses approve/commit/epic/legend/tale.	XPrompt directives (https://sase.sh/xprompt/#plan-directive)
`sase plan propose`	Submit a plan file for approval from the plan skill path.	XPrompt directives (https://sase.sh/xprompt/#plan-directive)
`sase questions`	Ask structured user questions from the questions skill path.	XPrompt directives (https://sase.sh/xprompt/#directives)

ChangeSpecs are CL/PR-sized review records. SDD stores durable prompt and planning artifacts. Beads add git-portable dependency tracking and executable epics on top of those artifacts.

`sase project` defaults to `sase project list`, and `sase project list` defaults to active projects. Use `sase project list --state all --json` to inspect inactive and sibling projects, `sase project deactivate <project>` to hide a dormant project from default launch views, and `sase project activate <project>` to make a hidden project launchable again. Deactivating refuses projects with live `RUNNING` claims or active artifact markers unless `--force` is passed. Deprecated `archive` and `close` aliases still set `inactive` for compatibility. ACE's `,p` Project Management panel provides the interactive counterpart, including marking multiple projects, editing a ProjectSpec in `$EDITOR`, and deleting obsolete SASE project directories after confirmation. There is no CLI delete subcommand; full project-directory deletion is only available from ACE's `,p` panel and removes state under `~/.sase/projects/`, not workspace checkouts.

`sase project alias list [PROJECT] [-j|--json]`, `add PROJECT ALIAS`, `remove PROJECT ALIAS`, and `clear PROJECT` manage ProjectSpec aliases. The ACE `,p` Project Management panel also displays aliases, includes them in filtering, and opens an alias editor with `A`. Alias refs are accepted in launch-bound VCS workspace tags, but prompt history, agent metadata, and artifacts use the canonical project name. Providers may also generate aliases automatically; for example, the GitHub provider can create `foo` and then `foo-2` aliases for distinct `owner/foo` repositories while keeping stable canonical project records. Use the same `sase project alias` commands to inspect or adjust those generated aliases.

Active-only project discovery is also the default for launch pickers, ChangeSpec searches, project-local xprompt catalogs, broad mobile helper catalogs, and all-known bead helper reads. Sibling records are hidden from those surfaces and are intended for configured linked repositories. To open one, pass its linked-repo name as the workspace CLI's project override: `sase workspace open -p <linked_repo> -r "<reason>" <workspace_num>`. Agent-history views that need older artifacts opt into all project states explicitly.

`sase plan` defaults to `sase plan list`. The dashboard has Proposed, Approved, and Rejected sections. Proposed rows include an `id_prefix`, agent, project, provider/model, plan path, and response directory; pass that prefix to `sase plan approve <prefix>`. If the selector is omitted, exactly one pending proposal must exist. The approval kind is the workflow choice: `approve` runs the coder without asking the runner to commit an SDD plan, `tale` commits the plan as an SDD tale and then runs the coder, `epic` and `legend` commit the matching SDD tier and launch the bead follow-up, and `commit` records the approved plan in SDD without launching a coder. Use `-m/--model` to pick the follow-up agent's model. Use `-p/--prompt` to add extra coder instructions for the `approve` and `tale` paths.

Automation

Command	Purpose	Details
`sase axe start`	Start the axe orchestrator and its lumberjacks.	Axe (https://sase.sh/axe/)
`sase axe stop`	Stop the running orchestrator.	Axe (https://sase.sh/axe/#cli-commands)
`sase axe chop list`	List configured chop jobs.	Axe chops (https://sase.sh/axe/#chop-fields)
`sase axe chop run <name>`	Run one chop in the foreground.	Axe chops (https://sase.sh/axe/#script-chops)
`sase axe lumberjack list`	List configured lumberjacks.	Axe lumberjacks (https://sase.sh/axe/#default-lumberjacks)
`sase axe lumberjack run <name>`	Run one lumberjack in the foreground for debugging.	Axe lumberjacks (https://sase.sh/axe/#lumberjack-configuration)
`sase axe lumberjack status`	Show lumberjack process status.	Axe (https://sase.sh/axe/)
`sase axe maintenance enter`	Pause scheduled lumberjack ticks with a recorded reason.	Maintenance mode (https://sase.sh/axe/#maintenance-mode)
`sase axe maintenance exit`	Resume scheduled lumberjack ticks.	Maintenance mode (https://sase.sh/axe/#maintenance-mode)
`sase axe maintenance status`	Inspect the maintenance marker.	Maintenance mode (https://sase.sh/axe/#maintenance-mode)

Axe runs scheduled hooks, mentors, comment polling, workflow checks, `%wait` dependency resolution, cleanup, and error digests. ACE starts axe automatically unless launched with `sase ace --no-axe`.

Prompt And Workflow Authoring

Command	Purpose	Details
`sase xprompt expand`	Expand xprompt references in prompt text, with optional trace output.	XPrompt reference syntax (https://sase.sh/xprompt/#reference-syntax)
`sase xprompt explain`	Dry-run a workflow and show the execution plan.	Workflows (https://sase.sh/workflow_spec/)
`sase xprompt list`	Emit the structured xprompt catalog as JSON.	XPrompt catalog (https://sase.sh/xprompt/#cli-subcommands)
`sase xprompt graph`	Generate a workflow DAG as Mermaid or text.	Workflow graphing (https://sase.sh/xprompt/#cli-subcommands)

`sase xprompt catalog`	Render visible xprompts to a formatted PDF catalog.	XPrompt catalog (https://sase.sh/xprompt/#cli-subcommands)
`sase lsp`	Start the xprompt language server over stdio.	Editor integration (https://sase.sh/editor/#language-server)
`sase editor helper-bridge`	JSON helper operations for editor integrations.	Editor helper bridge (https://sase.sh/editor/#helper-bridge)
`sase file list`	Emit JSON filesystem completion candidates.	Editor completion commands (https://sase.sh/configuration/#sase-file)
`sase file-history list`	Emit recently referenced files for editor completion.	Editor completion commands (https://sase.sh/configuration/#sase-file-history)
`sase file-history delete`	Remove one path from the file-reference history.	Editor completion commands (https://sase.sh/configuration/#sase-file-history)
`sase skill` / `sase skill list`	Inspect generated skill sources, provider targets, and deployed-file drift.	Initialization (https://sase.sh/init/#skill-initialization), bundled skills (https://sase.sh/xprompt/#bundled-skills)
`sase skill init`	Generate and deploy agent skill files from xprompt source templates.	Initialization (https://sase.sh/init/#skill-initialization), bundled skills (https://sase.sh/xprompt/#bundled-skills)
`sase skill log`	Summarize or inspect audited generated skill-use events.	Skill field (https://sase.sh/xprompt/#skill-field)
`sase skill use`	Agent-side audit event recording that a generated skill was used.	Skill field (https://sase.sh/xprompt/#skill-field)
`sase init skills`	Compatibility alias for `sase skill init`.	Initialization (https://sase.sh/init/#skill-initialization)

Use `#name(...)` for inline xprompt expansion and `#!workflow(...)` for standalone workflow references. Workspace references such as `#cd:<path>`, `#git:<project>`, and plugin-provided references are resolved before the prompt or workflow runs.

Review And Delivery

Command	Purpose	Details
`sase commit`	Dispatch a commit, proposal, or PR through the configured VCS provider.	Commit workflows (https://sase.sh/commit_workflows/)
`sase revert`	Revert a ChangeSpec by pruning its change and archiving its diff.	Commit workflows (https://sase.sh/commit_workflows/)
`sase restore`	Restore a reverted ChangeSpec by reapplying its archived diff.	Commit workflows (https://sase.sh/commit_workflows/)
`sase comments`	Preview mentor comments from JSON with syntax-highlighted code context.	Mentors (https://sase.sh/mentors/)

Delivery commands delegate to the VCS and workspace provider layers, so the same command surface can support plain git, GitHub pull requests, and other provider plugins.

Operations And Diagnostics

Command	Purpose	Details

`sase doctor`	Run read-only install, config, provider, project, and state diagnostics for support.	Doctor support reports
`sase config layers`	Show the configuration merge chain.	Configuration (https://sase.sh/configuration/)
`sase config show`	Dump the final merged configuration, optionally filtered by key.	Configuration (https://sase.sh/configuration/)
`sase config mentor-match`	Trace mentor profile matching for a ChangeSpec.	Mentors (https://sase.sh/mentors/)
`sase core health`	Check that the required `sase_core_rs` extension is loadable and working.	Rust backend (https://sase.sh/rust_backend/)
`sase validate`	Run SASE validation checks: initialization drift plus SDD frontmatter validation.	Initialization (https://sase.sh/init/), SDD (https://sase.sh/sdd/)
`sase telemetry status`	Show telemetry configuration and reachability.	Telemetry (https://sase.sh/telemetry/)
`sase telemetry list`	Display the metric catalog.	Telemetry (https://sase.sh/telemetry/)
`sase telemetry snapshot`	Fetch current metric values.	Telemetry (https://sase.sh/telemetry/)
`sase telemetry dashboard`	Open the live telemetry dashboard.	Telemetry (https://sase.sh/telemetry/)
`sase telemetry health`	Run subsystem health assessment.	Telemetry (https://sase.sh/telemetry/)
`sase telemetry export-config`	Export the bundled monitoring stack.	Telemetry (https://sase.sh/telemetry/)
`sase version`	Show the local `sase`, `sase-core-rs`, and installed plugin package inventory for this runtime.	Runtime inventory (https://sase.sh/configuration/#sase-version)
`sase logs`	Collect and package agent run logs for a date range.	Configuration CLI flags (https://sase.sh/configuration/#sase-logs)
`sase revive-log`	Inspect the agent-revival audit log (start / success / failure events).	Agent revival audit log (https://sase.sh/troubleshooting/agent-revival/)
`sase artifact create`	Move an explicit file into persistent agent artifact storage.	Agent attachments (https://sase.sh/agent_images/)
`sase var set`	Attach named output variables for ACE metadata and waited-agent Jinja rendering.	XPrompt variables (https://sase.sh/xprompt/#cross-agent-output-variables)
`sase plugin list`	Inventory installed SASE plugin entry points and configured/available chop scripts.	Plugins (https://sase.sh/plugins/#cli-diagnostics)
`sase plugin doctor`	Diagnose plugin resource loading, configured chops, and optional integration prerequisites.	Plugins (https://sase.sh/plugins/#cli-diagnostics)
`sase path`	Print well-known paths such as schemas and xprompt directories.	Configuration CLI flags (https://sase.sh/configuration/#sase-path)
`sase workspace list`	List managed workspace checkouts in the registry, including primary `#0`.	Workspace provider (https://sase.sh/workspace/)

`sase workspace path`	Print the checkout path for a workspace number.	Workspace provider (https://sase.sh/workspace/)
`sase workspace open`	Materialize, prepare, and print a workspace path.	Workspace provider (https://sase.sh/workspace/)
`sase workspace cleanup`	Remove stale unclaimed managed checkouts older than the configured TTL.	Workspace provider (https://sase.sh/workspace/)
`sase workspace repair`	Reconcile the workspace registry with the filesystem.	Workspace provider (https://sase.sh/workspace/)
`sase workspace migrate`	Opt-in move of adjacent checkouts to a managed root, with optional symlink transition and finalization.	Workspace provider (https://sase.sh/workspace/)
`sase mobile gateway start`	Start the workstation-hosted mobile gateway.	Mobile gateway (https://sase.sh/mobile_gateway/)
`sase mobile agent-bridge`	Fixed JSON bridge used by the mobile gateway for agent operations.	Mobile gateway (https://sase.sh/mobile_gateway/)
`sase mobile helper-bridge`	Fixed JSON bridge used by the mobile gateway for workflow helper operations.	Mobile gateway (https://sase.sh/mobile_gateway/)

Operational commands are intentionally narrow. Helper bridges expose fixed JSON operations for editor and mobile clients; they are not general shell or filesystem APIs.

Doctor Support Reports

`sase doctor` is the first command to run when SASE behaves unexpectedly. It is read-only by default: it does not launch agents, call LLM APIs, repair state, run tests, or scan full artifact history. The human output is grouped by subsystem and puts next-step commands beside warnings and errors.

Common forms:

```
sase doctor                # compact human report
sase doctor -v             # include every check plus bounded details
sase doctor -j             # stable JSON report for scripts or support bundles
sase doctor -D             # add slower read-only deep checks
sase doctor -C runtime     # run one group
sase doctor -C llm.default  # run one check
```

Exit codes are designed for support-first use. `OK`, `WARN`, and all-skipped reports exit `0`; `ERROR` exits `1`. Use `sase doctor -s` / `--strict` when automation should treat warnings as failures.

The JSON report uses `schema_version: 1` and stable top-level fields such as `status`, `counts`, `selected_checks`, and `checks`. Individual check `data` payloads stay bounded and may gain additional keys over time, so scripts should key off check ids and statuses rather than assuming every nested field is permanent.

When asking for help, attach `sase doctor -v` for a readable report or `sase doctor -j` for a machine-readable report.

Development

This page orients contributors working in the `sase` repository. It covers local setup, verification, source layout, and documentation publishing paths.

Setup

Requirements:

- Python 3.12+
- `uv` (https://docs.astral.sh/uv/)
- `just` (https://github.com/casey/just)

```
uv venv .venv
source .venv/bin/activate
just install
sase --help
```

`just install` installs the package in editable mode with development dependencies. When a sibling `../sase-core` checkout is present and `cargo` is available, it also builds and installs the local `sase_core_rs` extension before resolving Python dependencies.

Verification Commands

```
just install       # Install with dev deps
just fmt           # Auto-format code and Markdown
just lint          # Run ruff, mypy, pyvision, keep-sorted, and SDD validation
just test          # Fast parallel test run, including PNG visual snapshots
just test-slow     # Slow pytest subset only
just test-visual   # ACE PNG visual regression snapshots only
just test-terminal-smoke  # Optional real-terminal ACE smoke test
just test-cov      # Parallel test run with coverage + 50% gate, including visual snapshots
just check         # CI-style checks: formatting, lint, SDD validation, tests
just test-tox      # Test across Python 3.12, 3.13, 3.14
just clean         # Remove build artifacts
just build         # Build wheel and sdist
```

`just test`, `just test-slow`, `just test-visual`, and `just test-cov` size the pytest-xdist worker pool from local CPU count, capped at 16. Set `SASE_PYTEST_WORKERS=<N>` to override that value. Test selectors are normalized from the directory where `just` was invoked, so this works the same from the repository root or a subdirectory:

```
just test tests/main/test_parser.py::test_example
```

`just lint` and `just fix-keep-sorted` bootstrap a project-local `keep-sorted` executable into `.venv/bin/` from `PATH`, or by running `go install github.com/google/keep-sorted@v0.8.0` when Go is available. If neither `keep-sorted` nor Go is installed, those recipes fail with a setup error before linting YAML keep-sorted blocks.

Default test runs exclude `slow` and `terminal_smoke` markers but include the ACE PNG snapshot regression tests. Use `just test-visual` for focused visual-snapshot work; both recipes install the optional PNG rasterizer dependencies when they are missing. Direct `pytest` runs still inherit the repository `pyproject.toml` default marker expression, which excludes `slow`, `terminal_smoke`, and `visual` unless you pass your own `-m` selector.

Use `just test-terminal-smoke` only when you need to verify the ACE startup path through a real PTY. It installs `pexpect` and `pyte`, runs the optional `terminal_smoke` marker, and stays out of default tests and CI until that path has proved stable.

Visual Snapshot Workflow

ACE visual tests live under `tests/ace/tui/visual/` and compare deterministic Textual screenshots against committed PNG goldens. Run them normally first:

```
just test-visual
```

When a visual test fails, inspect the artifacts under `.pytest_cache/sase-visual/<node>/<snapshot>/`. Each failure directory contains the actual PNG capture and, when a golden exists, the expected PNG plus a diff PNG, a human-readable `summary.txt`, and a structured `failure.json` sidecar. The sidecar carries the test source location, repo-relative golden path, and pixel-diff stats so tooling can map a failure back to the test and the committed golden. Accept intentional visual changes only by rerunning the relevant test with the explicit update flag:

```
just test-visual -- --sase-update-visual-snapshots tests/ace/tui/visual/test_ace_png_snapshots.py
```

Review changed PNG files as normal test data. Do not pass `--sase-update-visual-snapshots` to `just check`, `just fmt`, or broad CI-style commands.

PNG comparison tolerance is environment-gated. Dedicated local visual runs require exact pixel equality against the committed golden; any drift there is a real regression that needs investigation, not relaxation. Broad commands that include visual tests, such as `just test` and `just test-cov`, set `SASE_VISUAL_PNG_MAX_DIFF_RATIO=0.001` so tiny font/rasterizer jitter does not break the full local suite. Runs in GitHub Actions allow a small ratio-only renderer-drift tolerance for the same reason. Treat a `just test-visual` failure as a true regression even if the same test passes in CI, and accept new goldens only with `--sase-update-visual-snapshots` after inspecting the diff PNG locally.

Visual Failure Report

`tools/render_visual_snapshot_failure_report` consumes the `failure.json` sidecars and writes `.pytest_cache/sase-visual-report/`:

- `visual-failure-report.html` - self-contained HTML with PNG/SVG embedded as data URIs, one anchored section per failure.
- `summary.md` - compact table for `$GITHUB_STEP_SUMMARY` with links into the report and to the committed golden.
- `annotations.sh` - escaped `::error file=...,line=...` workflow commands.
- `manifest.jsonl` - aggregate of every loaded `failure.json` for ad-hoc inspection.

Run it locally against a failed run with `tools/render_visual_snapshot_failure_report --repo <owner/repo> --sha <commit>` and open the HTML file directly. The script is safe to run when there are no failures; it exits 0 without writing artifacts.

In GitHub Actions the `visual-test` job invokes the renderer twice on failure: once to build the report before upload, then again after upload with `--report-url "$VISUAL_REPORT_URL"` so the summary and annotations point at the freshly uploaded artifact. The HTML is uploaded via `actions/upload-artifact@v7` with `archive: false`, which is what makes the per-failure anchors browsable directly from the Actions UI. Expected links point at the immutable

`https://github.com/<repo>/blob/<sha>/<expected_repo_path>` URL; actual/diff links point at the report artifact rather than a public PNG URL because the raw PNGs are only uploaded as a zipped `ace-visual-artifacts` bundle and have no stable per-file URL.

Add a visual test when the risk is layout, styling, focus highlighting, modal composition, or a regression that is hard to express as state. Prefer a plain state/widget test when the behavior can be asserted through model state, rendered text, selection identity, key handling, or a small widget contract.

Required Rust Core

Ported `sase.core` operations are served by the required Rust extension `sase_core_rs`, distributed as the `sase-core-rs` package and built from the sibling `../sase-core` repo during source development. Normal installs pull a prebuilt wheel; local source installs can build the extension with `just install` or `just rust-install`.

There is no pure-Python fallback for ported operations. Use the health check after install changes:

```
sase core health
```

See the Rust backend reference (https://sase.sh/rust_backend/) for the Python/Rust boundary, shipped Rust-backed operations, source build path, and benchmark expectations.

Source Map

The repository is organized around the CLI entry point, operational subsystems, provider boundaries, and docs/tests:

Path	Purpose
`src/sase/main/`	CLI parser registration and subcommand handlers.
`src/sase/ace/`	ACE TUI, ChangeSpec rendering, query integration, actions, widgets, and TUI state.
`src/sase/agent/`	Agent launch, detached spawn, prompt fan-out, running-agent metadata, artifact lookup, and naming.
`src/sase/axe/`	Axe orchestrator, lumberjacks, chop execution, scheduled jobs, maintenance mode, and automation state.
`src/sase/xprompt/`	XPrompt expansion, directives, workflow loading, execution, tracing, explaining, and graphing.
`src/sase/xprompts/`	Bundled xprompt templates, workflows, and schemas shipped with the package.
`src/sase/workflows/`	Change lifecycle workflows for commit, mentor, CRS, accept, and rewind operations.
`src/sase/memory/`	Memory inventory, audited read logs, and proposal write/review flows.
`src/sase/core/`	Python facade and stable wire records for operations served by `sase_core_rs`.
`src/sase/bead/`	Python host layer for bead storage discovery, CLI integration, and epic/legend launch flow.
`src/sase/sdd/`	Spec-driven development file and bead integration helpers.
`src/sase/llm_provider/`	Built-in LLM providers and provider registry.
`src/sase/vcs_provider/`	VCS provider hook specs, plugin registry, and built-in git provider.
`src/sase/workspace_provider/`	Workspace provider hook specs, plugin registry, `#cd`, and bare-git workspace support.

`src/sase/running_field/`	Workspace claim and slot-management helpers.
`src/sase/notifications/`	Notification delivery and storage integration.
`src/sase/telemetry/`	Prometheus metrics, health, dashboard, and monitoring export support.
`src/sase/version/`	Runtime inventory collection and rendering for the `sase version` CLI command.
`src/sase/integrations/`	Public helper APIs consumed by external plugins and editors.
`src/sase/scripts/`	Packaged utility scripts used by axe chops and support commands.
`tests/`	Python test suite, with subdirectories mirroring major `src/sase/` areas.
`docs/`	MkDocs Material site source.
`sdd/`	Project-local prompt, tale, epic, legend, research, and bead artifacts.
`xprompts/`	Repository-local xprompts and workflows for SASE maintenance agents.
`tools/`	Development scripts used by `just` targets and CI checks.
`memory/`	SASE memory files used by repository agents.

Detailed subsystem pages often include narrower source-layout tables. Use this page for initial orientation, then jump to the specific reference for the area you are changing.

Repository XPrompts

The checkout's top-level `xprompts/` directory is project-local to the `sase` repository. When SASE resolves prompts from this project checkout, those entries are namespaced as `sase/<name>` so they do not collide with user or packaged prompts. Use the catalog's `insertion` value to know whether an entry should be invoked with `#` or `#!`.

Useful visible entries include:

Reference	Purpose
`#!sase/reads`	Fan out a reading-recommendation request across Antigravity, Claude, and Codex, then consolidate the final list.
`#sase/sync`	Sync the primary SASE workspace and restart axe.

`#!sase/reads` accepts a required `topic` and an optional `reference_query`. By default, the workflow passes this Dataview query to the research agents:

```
LIST WITHOUT ID title + " (" + url + ")"
FROM "ref"
WHERE
  source_path AND url AND (
    parent = [[ai_ref]]
    OR parent.parent = [[ai_ref]]
    OR parent.parent.parent = [[ai_ref]]
    OR parent.parent.parent.parent = [[ai_ref]]
    OR parent.parent.parent.parent.parent = [[ai_ref]]
  )
SORT title
```

Each research agent is expected to use `/bob_dataview` to run that query against Bryan's Bob vault, treat every returned title and URL entry as already-known, and only then search for new reading candidates. A normal invocation can rely on the default query:

```
#!sase/reads(agent memory systems)
```

Some repository workflows are marked `hidden: true` because they are automation helpers, such as docs refresh, recent bug/improvement audits, and Python line-limit splitting. That flag hides workflow run rows in ACE; it does not mean the workflow is unavailable. Use `sase xprompt list` or the ACE xprompt browser from a source checkout when you need the exact current catalog.

Documentation Workflow

The docs site is a MkDocs Material project:

Path	Purpose
`mkdocs.yml`	Main docs site configuration, strict build, navigation, blog, RSS, and theme settings.
`mkdocs-pdf.yml`	PDF handbook build configuration, inheriting the main site config.
`docs/`	Markdown, images, stylesheets, JavaScript, redirects, headers, and PDF templates.
`site/`	Generated site output. It is rebuilt by docs commands and deployed as the static asset directory.

Run the strict site build after changing docs navigation, links, images, or Markdown pages:

```
just docs-check
```

Run SASE validation when a change can affect generated initialization files or SDD frontmatter links. This is the same validation lane used by `just lint`, so it can report user/home initialization drift as well as repository-local issues:

```
sase validate
```

Run the handbook build and validation when a change materially affects the public handbook, PDF styling, navigation, or generated-site assets:

```
just docs-pdf-check
```

`just docs-check` installs only MkDocs tooling, then runs `mkdocs build --strict`. `just docs-pdf-check` installs the PDF tooling, installs Chromium for Playwright, builds `mkdocs-pdf.yml` in an isolated temporary site directory, post-processes and validates the handbook there, and copies only `downloads/sase-handbook.pdf` back into `site/`.

Docs Deployment

Production docs are deployed by `.github/workflows/docs-deploy.yml`, not by a Cloudflare dashboard build command. The workflow:

1. Checks out the repo and installs `uv`, `just`, and Python 3.12.
2. Runs `just docs-check`.
3. Runs `just docs-pdf-check`.

4. Verifies `site/index.html`, `site/_headers`, the blog and series pages, and `site/downloads/sase-handbook.pdf`.
5. Deploys the prebuilt `site/` directory through `wrangler.jsonc`.
6. Smoke-tests the deployed handbook PDF from the deployment URL and `https://sase.sh/`.

The GitHub repository must provide a `CLOUDFLARE_API_TOKEN` Actions secret with permission to deploy the `sase` Cloudflare Worker. Keep dashboard-managed Git builds disabled or unused for production so they cannot race the checked in workflow's prebuilt artifact deploy.

24 Configuration Reference

This document is the central reference for all sase configuration: config files, YAML sections, environment variables, and CLI flags.

24.1 Table of Contents

- Config File Location
- Deep-Merge System
- Configuration Sections
- amd_h1_title
- ace
- llm_provider
- commit
- linked_repos
- vcs_provider
- axe
- mentor_profiles
- metahooks
- xprompts
- xprompt_aliases
- use_chezmoi
- precommit_command
- timezone
- chat_install
- mobile_gateway
- sdd
- bead
- workspace
- telemetry
- Environment Variables
- CLI Flags
- Directory Sharding

24.2 Config File Location

All sase configuration lives under `~/.config/sase/`. The base config file is:

```
~/.config/sase/sase.yml
```

Overlay files matching the glob `~/.config/sase/sase_*.yml` are merged on top of the base file. A project-local `./sase.yml` in the current working directory usually takes highest priority. The ACE TUI deliberately disables project-local config loading for its own process so opening `sase ace` inside a repo does not inherit that repo's agent-run settings. See Deep-Merge System below.

24.3 Deep-Merge System

Sase builds a merged configuration through five layers, each merged on top of the previous:

1. `default_config.yml` — bundled package defaults
2. **Plugin** `default_config.yml` **files** — from installed plugin packages (via `sase_config` entry points), sorted by entry-point name; lists concatenate
3. `sase.yml` — user config (`~/.config/sase/sase.yml`); lists **replace** defaults (not concatenate)
4. `sase_*.yml` **overlays** — sorted alphabetically; lists **concatenate**
5. **Local** `sase.yml` — project-level config in the current working directory; lists **concatenate** (highest priority)

This allows splitting configuration across multiple files (e.g., `sase_work.yml`, `sase_personal.yml`) without duplication, plugins can provide sensible defaults that users can override, and individual projects can customize behavior without changing global config.

Merge semantics:

Type	Behavior
Dicts	Merged recursively (overlay keys override base keys).
Lists	Concatenated in layers 2, 4, and 5; **replaced** in layer 3 (user config).
Scalars	Override (overlay value replaces base value).

For example, given a base file with two mentor profiles and an overlay or local project config that adds a third, the merged result contains all three profiles. A user `~/.config/sase/sase.yml` list replaces earlier defaults instead. If two files define the same scalar key (e.g., `axe.max_hook_runners`), the later layer wins.

Source: `src/sase/config/core.py`

24.4 Configuration Sections

24.4.1 amd_h1_title

Opts a root into a generated AMD-managed `AGENTS.md` by providing the Markdown H1 title for that file.

```yaml
amd_h1_title: "Structured Agentic Software Engineering (SASE) - Agent Instructions" # default: null
```

Field	Type	Default	Description
`amd_h1_title`	string | null	`null`	H1 title used by the AMD `AGENTS.md` generator when enabled for that scope.

For ordinary project roots, this field is intentionally local to the root being initialized. The AMD generator reads only that root's `./sase.yml` value, so a global `~/.config/sase/sase.yml` value does not opt every repository on the machine into generated `AGENTS.md` files.

Home roots are the exception. When `sase amd init` targets the live home root, user config from `~/.config/sase/sase.yml` and `~/.config/sase/sase_*.yml` can provide the home `AGENTS.md` title. When it targets the chezmoi home source root, source-side config under `dot_config/sase/` is used instead. With `use_chezmoi: true`, AMD initializes the chezmoi home source root rather than writing a live-home `AGENTS.md`.

Source: `src/sase/default_config.yml`, `config/sase.schema.json`

24.4.2 ace

Configures the ACE TUI behavior. Defaults are provided by `src/sase/default_config.yml`.

```yaml
ace:
  inactive_seconds: 600 # seconds before showing IDLE indicator (default: 600)
  keymaps:
    app:
      next_changespec: "j"
      prev_changespec: "k"
      # ... all app-level keybindings are configurable
    modes:
      # Built-in modes (fold, copy, leader, bang) are configurable
      leader_mode:
        prefix: "comma"
        keys:
          repeat_last: "comma" # press the leader prefix, then this key; defaults render as `,,`
          projects: "p"
          temporary_llm_override: "o"
          full_history_refresh: "y"
      fold_mode:
        prefix: "z"
        keys:
          cycle_commits: "c"
          cycle_hooks: "h"
      # Custom modes can be added here
      my_mode:
        prefix: ";"
        keys:
          run_tests:
            key: "t"
            shell: "just test"
```

Field	Type	Default	Description
`inactive_seconds`	int	`600`	Seconds of inactivity before the IDLE badge appears in the TUI top bar.
`keymaps`	dict	-	Configurable keybindings (see below).
`prompt_completion`	dict	see below	Live soft-completion settings for the ACE prompt input.
`repro_output_dir`	str	`""`	Base directory for Agents-tab reproduction bundles (https://sase.sh/ace/#agents-tab-reproduction-bundles). Empty means `<SASE_HOME>/repros` (default `~/.sase/repros`).
`snippets`	dict[string]	`{}`	Trigger-word → template mappings for prompt input snippet expansion.

The IDLE indicator can also be triggered manually via the leader-mode `,I` keybinding. External tools can query idle status via `sase.ace.tui_activity.is_idle()`.

`ace.keymaps`

All TUI keybindings are configurable. The `keymaps` section has two sub-sections:

`app` — App-level keybindings. Each key is an action name mapped to a key string. See `src/sase/default_config.yml` for the full list of configurable actions and their defaults.

`modes` — Prefix-key mode definitions. Built-in modes (`fold_mode`, `copy_mode`, `leader_mode`, `bang_mode`) can be reconfigured, and custom modes can be added. Each mode has:

Field	Type	Description
`prefix`	str	The activation key for the mode.
`keys`	dict	Sub-key definitions. For custom modes, each entry needs a `key` field and either `shell` or `action`.

Custom mode key fields:

Field	Type	Required	Description
`key`	str	yes	The sub-key to press after the prefix.
`shell`	str	no*	Shell command to execute.
`action`	str	no*	Built-in action name to invoke.

*Exactly one of `shell` or `action` must be provided.

The keymap loader validates configuration: invalid keys are reverted to defaults, duplicate bindings are warned, and prefix conflicts between custom modes and app bindings are detected.

Source: `src/sase/default_config.yml`, `src/sase/ace/tui/keymaps/`

`ace.snippets`

Defines expandable text snippets for the prompt input widget. Each entry maps a trigger word to a template string. Press `Tab` in the prompt input to expand the trigger word before the cursor.

```yaml
ace:
  snippets:
    fix: "Please fix the following issue:\n$0"
    review: "Review this code for correctness, performance, and style."
    plan: "#plan\n$0"
```

Templates can contain `$0` to mark where the cursor should be placed after expansion. If no `$0` is present, the cursor moves to the end of the expanded text.

See docs/ace.md — Snippets (https://sase.sh/ace/#snippets) for usage details.

Source: `src/sase/ace/tui/widgets/prompt_text_area.py`

`ace.prompt_completion`

Controls automatic non-disruptive suggestions in the ACE prompt input. Suggestions appear in the prompt-bar subtitle and are accepted with `Ctrl+L`; `Enter` still submits the prompt as typed. Manual `Ctrl+T` completion is independent of these settings, and the `Ctrl+R` recursive fuzzy file finder is always manual.

```yaml
ace:
  prompt_completion:
    auto: soft
    debounce_ms: 90
    auto_file_paths: false
    auto_xprompt_menu: true
    auto_directive_menu: true
    max_auto_rows: 1
```

Field	Type	Default	Description
`auto`	bool/string	`soft`	Automatic mode. `soft`, `true`, `on`, `yes`, or `1` enable subtitle suggestions; false/off disables them.
`debounce_ms`	int	`90`	Delay before computing a live suggestion after text or cursor changes.
`auto_file_paths`	bool	`false`	Allow live suggestions to scan file-path candidates. Manual `Ctrl+T` file completion still works when false.
`auto_xprompt_menu`	bool	`true`	Automatically open the xprompt/skill completion menu while typing matching `#name`, `#!name`, or `/skill` tokens.
`auto_directive_menu`	bool	`true`	Automatically open the directive completion menu while typing matching `%name` tokens.
`max_auto_rows`	int	`1`	Reserved row limit for automatic completion modes; current soft mode shows one suggestion.

The `#+query` and first-character `+query` project/ChangeSpec picker uses the same completion panel but is triggered directly by the `+` token; it is not disabled by `auto_xprompt_menu`. Manual `Ctrl+T` project/ChangeSpec completion works regardless of these automatic-completion settings.

File-path completion roots relative lookups in the prompt-selected workspace. A resolvable `#cd` reference takes precedence; without `#cd`, registered workspace-provider refs and known-project refs such as `#git:<project>` or `#gh:<owner>/<repo>` can root lookup in that project checkout. If no prompt workspace ref resolves, lookups fall back to the TUI process directory. These root rules are shared by live path suggestions, manual `Ctrl+T` path completion, and the manual `Ctrl+R` recursive finder.

Source: `src/sase/ace/tui/widgets/prompt_completion.py`, `src/sase/ace/tui/widgets/_prompt_soft_completion.py`, `src/sase/ace/tui/widgets/prompt_completion_root.py`, `src/sase/ace/tui/widgets/recursive_file_finder.py`

24.4.3 llm_provider

Configures which LLM backend sase uses and how model tiers map to concrete models. See docs/llms.md (https://sase.sh/llms/) for the full LLM provider architecture, preprocessing pipeline, and invocation lifecycle.

```yaml
llm_provider:
  provider: claude # or "qwen", "opencode", "agy" (default: auto-detect)
  worker_models:
    claude: codex/gpt-5.5 # worker default when primary is on Claude
    codex: claude/opus # worker default when primary is on Codex
  model_tier_map:
    large: opus
    small: sonnet
  model_aliases:
    other: claude/opus
```

Field	Type	Default	Description
`llm_provider.provider`	string	auto-detect	Which registered provider to use. Auto-detects by plugin-declared priority; built-ins default to claude → codex → qwen → opencode → agy.
`llm_provider.worker_models`	dict	unset	Optional worker-lane targets keyed by the effective primary lane. Values accept bare known models, aliases, or explicit `provider/model`.
`llm_provider.model_tier_map.large`	string	-	Model identifier for the `large` tier.
`llm_provider.model_tier_map.small`	string	-	Model identifier for the `small` tier.
`llm_provider.model_aliases`	dict	-	Model aliases usable from `%model:<alias>` / `%m:<alias>`. Values can be bare known models, explicit `provider/model`, or nested provider-local model paths.

Model aliases are resolved when an agent launches, so reusable xprompts can point at names such as `%model:other` while each user's `sase.yml` controls the concrete provider/model. Unknown aliases and unknown model values keep the existing fallback behavior and run on the default provider.

The optional `worker_models` config defines the worker lane used by delegated work, currently including `sase bead work` phase agents that do not have an explicit per-bead model. Entries are selected from the current effective primary lane in this order: exact `provider/model`, bare model, then provider. Provider entries are defaults only and do not override model-specific entries. When no entry matches, the worker lane follows the primary default lane: active primary override, configured provider/tier, then provider auto-detection. See Worker Model (https://sase.sh/llms/#worker-model) for the full precedence order and TUI behavior.

The alias name `other` is reserved: when a temporary LLM override is active (see Temporary Default Override (https://sase.sh/llms/#temporary-default-override)), `%model:other` resolves to the `(provider, model)` that was the effective default *immediately before* the override was set, rather than to the static `model_aliases.other` target. When no override is active, `other` falls back to the configured alias as usual.

The alias name `worker` is also reserved: `%model:worker` resolves through the worker lane and shadows any `model_aliases.worker` entry.

The TUI also supports **temporary** session-level provider/model overrides that do **not** edit this config. Primary overrides are persisted to `~/.sase/llm_override.json`; worker overrides are persisted to `~/.sase/llm_worker_override.json`. Expired entries are deleted on next read. See docs/llms.md (https://sase.sh/llms/#temporary-default-override) for the resolution order, state-file format, and precedence relative to `SASE_MODEL_TIER_OVERRIDE`.

`llm_provider.retry`

Per-provider retry and fallback configuration. See docs/llms.md (https://sase.sh/llms/#retry-and-fallback) for the full retry flow and TUI display.

```yaml
llm_provider:
  retry:
    claude:
      max_retries: 3
      error_patterns:
        - "API Error: 500"
      wait_times: [60, 300, 1800]
      fallback_model: "sonnet"
      continuation_prompt: "Please continue from the last preserved work."
      preserve_workspace: true
      spawn_new_agent: false
```

Field	Type	Default	Description
`llm_provider.retry.<provider>`	dict	-	Retry config for a specific provider (e.g., `agy`, `claude`, `codex`).
`llm_provider.retry.<provider>.max_retries`	int	`0`	Maximum retry attempts. `0` disables retrying.
`llm_provider.retry.<provider>.error_patterns`	list	`[]`	Case-insensitive substring patterns matched against error output.
`llm_provider.retry.<provider>.wait_times`	list	`[30]`	Per-retry wait times in seconds. Last value reused if list is shorter.
`llm_provider.retry.<provider>.fallback_model`	str	`null`	Alternate model to use after exhausting all retries.
`llm_provider.retry.<provider>.continuation_prompt`	str	`null`	Prompt text prepended when continuing after a retryable failure.
`llm_provider.retry.<provider>.preserve_workspace`	bool	`false`	Preserve on-disk edits across legacy in-process retry attempts.
`llm_provider.retry.<provider>.spawn_new_agent`	bool	`false`	Retry by launching a fresh detached agent that inherits the workspace.

Configured retry policy is merged with provider-supplied retry defaults when a provider declares them. For list fields such as `error_patterns`, built-in patterns are kept and configured patterns are appended with duplicates removed. Claude's provider hook adds workspace-preserving matching for context-limit, socket-close, and Claude CLI API-error output, plus a continuation nudge. Those hook defaults are merged with the bundled Claude policy in `default_config.yml`, so the configured wait times and fallback model still apply unless you override them.

Source: `src/sase/llm_provider/retry_config.py`, `src/sase/llm_provider/config.py`

24.4.4 commit

Configures commit enforcement around SASE-launched agents. The current commit finalizer is provider-neutral and runs in the shared LLM invocation layer after a successful provider invocation in a SASE agent session, identified by `SASE_AGENT_TIMESTAMP`.

```yaml
commit:
  finalizer:
    enabled: true
    max_passes: 2
```

Field	Type	Default	Description
`commit.finalizer.enabled`	bool	`true`	Run the post-invocation commit finalizer for SASE-launched agent sessions.
`commit.finalizer.max_passes`	int	`2`	Maximum follow-up invocations before a still-dirty enforced workspace fails the run.

When enabled, the finalizer checks the main workspace through the active VCS provider. For configured `linked_repos` Git worktrees using the numbered-workspace strategy, it checks only linked-repo names recorded in the run's `opened_linked_workspaces.json` artifact, normally written by `sase workspace open -p <linked_repo> -r "<reason>" <workspace_num>` during the agent run. In that command, `-p/--project` names the configured linked repo's hidden backing project record. Dirty enforced workspaces trigger a follow-up invocation that instructs the same provider to use the appropriate commit skill. Dirty static linked repos (`workspace.strategy: none`) are reported to that follow-up as advisory work and do not fail the finalizer if they remain dirty. Advisory-only static linked-repo changes still get one follow-up prompt so the agent can commit them when it made those changes. When the only enforced change is one tracked markdown file under `sdd/tales/`, `sdd/epics/`, `sdd/legends/`, or `sdd/myths/`, and that file's only diff is leading front matter changing exactly from `status: wip` to `status: done`, the finalizer creates a direct `chore: Mark SDD plan done` commit instead of invoking the provider again. When `$SASE_ARTIFACTS_DIR` is set, each pass writes prompt/response artifacts there, and the final outcome is recorded in `commit_finalizer_result.json`.

Set `SASE_DISABLE_COMMIT_STOP_HOOK=1` for a one-off bypass. The environment variable name is historical; it now disables the provider-neutral finalizer.

Source: `src/sase/llm_provider/commit_finalizer.py`, `src/sase/commit_instructions.py`

24.4.5 linked_repos

Declares related repositories that should be visible to launched agents. Git linked-repo worktrees using numbered workspace resolution are also eligible for commit-finalizer checks at their resolved `workspace_dir`, but only after the agent run opens that linked workspace with `sase workspace open -p <linked_repo> -r "<reason>" <workspace_num>` and records the linked repo in its artifacts. The `-p/--project` value is the linked repo's configured name; SASE materializes a hidden sibling-state ProjectSpec for that name when needed. Entries can live in user config or a project-local `sase.yml`; local entries are resolved relative to the project's primary workspace directory.

The deprecated `sibling_repos` key is still accepted as an alias during the compatibility window. Prefer `linked_repos` in new config.

```yaml
linked_repos:
  - name: core
    path: ../sase-core
    description: Shared backend/domain behavior used by SASE frontends.
  - name: github
    path: ../sase-github
    description: GitHub VCS and workspace provider plugin.
  - name: chezmoi
    path: ~/.local/share/chezmoi
    description: User dotfiles source managed by chezmoi.
    workspace:
      strategy: none
```

Field	Type	Default	Description
`linked_repos[].name`	string	required	Stable alias used in generated environment variable names and memory summaries.
`linked_repos[].path`	string	required	Primary checkout path. Relative paths resolve from the project's primary workspace.
`linked_repos[].description`	string	required	Human-readable purpose used when generating agent memory for the linked repository.
`linked_repos[].workspace.strategy`	string	`"suffix"`	`suffix` exposes a workspace-matched checkout for workspace `N`; `none` always exposes the primary checkout.

For `suffix` linked repos, workspace numbers `0` and `1` use the primary checkout. Higher workspace numbers use workspace-matched linked-repo checkouts, materializing the checkout through the same `workspace.root` policy when the workspace provider can do so. With explicit `workspace.root: adjacent` that path is a legacy adjacent checkout such as `sase-core_10`; with the default `xdg-state` it lives under the managed state root. Linked repos with `workspace.strategy: none` are exposed to agents and can appear as advisory dirty targets in commit finalizer prompts, but they are not commit-finalizer enforcement targets. SASE passes the resolved paths into environment variables and agent metadata:

Variable	Description
`SASE_LINKED_REPOS_JSON`	JSON metadata for all resolved linked repos.
`SASE_LINKED_REPO_<ENV_NAME>_DIR`	Workspace-matched directory for a linked repo.
`SASE_LINKED_REPO_<ENV_NAME>_PRIMARY_DIR`	Primary checkout directory for that linked repo.

The legacy `SASE_SIBLING_REPOS_JSON` and `SASE_SIBLING_REPO_<ENV_NAME>_*` variables are still emitted alongside the canonical ones during the compatibility window.

`<ENV_NAME>` is the uppercased, sanitized repo `name`; duplicates are uniquified with a numeric suffix.

Source: `src/sase/linked_repos.py`, `src/sase/agent/launch_spawn.py`

24.4.6 vcs_provider

Configures the version control system backend. See docs/vcs.md (https://sase.sh/vcs/) for the full VCS provider reference including per-command behavior, Git/Mercurial details, and troubleshooting.

```yaml
vcs_provider:
  provider: auto # "git", "hg", or "auto" (default: "auto")
  workspace_root: ~/workspace # optional workspace root directory
  default_hooks: # optional list overriding built-in default hooks
    - "!$my_presubmit"
    - "$my_lint"
  pr_tags: # optional key-value tags appended to PR commit messages
    Bug: "b/12345"
  use_project_pr_prefix: false # prepend [<project>] to PR titles (default: false)
```

Field	Type	Default	Description
`vcs_provider.provider`	string	`"auto"`	VCS provider: `"git"`, `"hg"`, or `"auto"` for directory detection.
`vcs_provider.workspace_root`	string	-	Legacy VCS helper workspace root. New numbered-checkout layout is configured by `workspace.root` below.
`vcs_provider.default_hooks`	list[string]	-	Hook commands added to new ChangeSpecs. Replaces built-in defaults.
`vcs_provider.pr_tags`	dict[string, str]	`{}`	Key-value tags appended as `TAG=VALUE` lines to PR commit messages.
`vcs_provider.use_project_pr_prefix`	bool	`false`	Prepend `[<project>]` to PR titles / CL descriptions (see below).

When `default_hooks` is not set, plugins may provide their own defaults via `default_config.yml` (for example, Mercurial-specific hooks from a provider plugin). The core `sase` package has no built-in default hooks.

When `use_project_pr_prefix` is `true`, a `[<project>]` prefix is prepended to PR titles (GitHub) or CL descriptions (Mercurial) without polluting the ChangeSpec DESCRIPTION or git commit message. The prefix is automatically stripped when reading descriptions back.

Source: `src/sase/vcs_provider/config.py`, `src/sase/ace/hooks/defaults.py`

24.4.7 axe

Configures the `sase axe` lumberjack-based daemon. The axe architecture uses an orchestrator that spawns multiple lumberjacks, each running a set of chops on a fixed interval. Defaults are provided by `src/sase/default_config.yml`.

```yaml
axe:
  max_hook_runners: 3 # concurrent hook runners (default: 3)
  max_agent_runners: 3 # concurrent agent runners (default: 3)
  zombie_timeout_seconds: 7200 # seconds (default: 7200 = 2 hours)
  query: "" # query filter for ChangeSpecs (default: all)
  chop_script_dirs: [] # additional directories to search for chop scripts
  lumberjacks:
    hooks:
      interval: 5
      chop_timeout: "90s"
      chops:
        - name: hook_checks
          description: Complete finished hooks and start stale ones, with zombie detection
        - name: mentor_checks
          description: Start mentor workflows once all hook prerequisites are met
        - name: workflow_checks
          description: Complete finished CRS/fix-hook workflows and start stale ones
        - name: pending_checks_poll
          description: Poll background is_cl_submitted and critique_comments checks for results
        - name: comment_zombie_checks
          description: Mark comment threads older than zombie_timeout as ZOMBIE
        - name: suffix_transforms
          description: Strip stale suffixes from older proposals and update mail-readiness markers
        - name: orphan_cleanup
          description: Release workspace claims orphaned by reverted CLs with dead PIDs
    waits:
      interval: 10
      chops:
        - name: wait_checks
          description: Resolve successful agent wait dependencies and write ready.json
    checks:
      interval: 300
      chops:
        - name: cl_submitted_checks
          description: Check if CLs have been submitted
        - name: stale_running_cleanup
          description: Clean up stale RUNNING entries
    comments:
      interval: 60
      chops:
        - name: comment_checks
          description: Check for new comments on CLs
    housekeeping:
      interval: 3600
      chops:
        - name: error_digest
          description: Summarize recent errors into a notification
```

Top-level fields:

Field	Type	Default	Description
`max_hook_runners`	int	`3`	Maximum concurrent hook runners (non-`$` hooks) across all ChangeSpecs.
`max_agent_runners`	int	`3`	Maximum concurrent agent runners (agents and mentors) across all ChangeSpecs.
`zombie_timeout_seconds`	int	`7200`	Seconds after which a running hook or workflow is flagged as a zombie.
`query`	string	`""`	Query string for filtering ChangeSpecs (empty = all).
`chop_script_dirs`	list[string]	`[]`	Additional directories to search for external chop scripts.
`lumberjack_log_max_bytes`	int	`52428800`	Maximum bytes retained for each bounded lumberjack log.
`verbose_lumberjack_diagnostics`	bool	`false`	Include verbose diagnostics in chop script context JSON.
`lumberjacks`	dict	-	Mapping of lumberjack name → config (see below).

Lumberjack fields (per entry under `lumberjacks`):

Field	Type	Default	Description
`interval`	int	`1`	Seconds between chop polling cycles.
`chop_timeout`	string	-	Default duration limit for each chop in this lumberjack, such as `"90s"`
`chops`	list[string|object]	`[]`	List of chops to run on each cycle (see below).

Chop fields (per entry under `chops`):

Field	Type	Required	Default	Description
`name`	string	yes	-	Chop name identifying the chop script to run.
`description`	string	yes	-	Human-readable description of what the chop does.
`agent`	string	no	`null`	XPrompt reference to launch as a background agent (accepts legacy `xprompt` key).
`run_every`	string	no	-	Time-based duration string (e.g., `"60m"`, `"30s"`, `"2h"`). Limits how often the chop runs.
`timeout`	string	no	-	Per-chop duration limit. Overrides the lumberjack-level `chop_timeout` when set.
`env`	dict[string]	no	`{}`	Environment variables passed to the chop script subprocess.

On a scheduled lumberjack tick, script chops remain concurrent, but configured agent chops launch sequentially in config order. This prevents same-tick `run_every` agent chops from racing while they allocate workspaces. A manual `sase axe chop run <agent-chop>` still launches only that one agent chop.

The built-in `wait_checks` chop writes `ready.json` only after named `%wait` dependencies complete successfully. Failed, killed, crashed, still-running, malformed, or missing `done.json` artifacts do not satisfy the dependency.

Each chop entry can also be a plain string (chop name only, legacy format):

```yaml
chops:
  # Object format (required for new chops)
  - name: hook_checks
    description: Check for completed or failed hooks
  - name: custom_chop
    description: Run custom analysis
    agent: "#analyze"
    run_every: "5m"
    env:
      MY_API_KEY: "secret"
  # String format (legacy, description defaults to empty)
  - hook_checks
```

CLI flags on `sase axe start` override `max_hook_runners`, `max_agent_runners`, `zombie_timeout_seconds`, and `query` for a single run (see CLI Flags).

Source: `src/sase/axe/config.py`, `src/sase/default_config.yml`

24.4.8 mentor_profiles

Defines mentor agents that run automated code reviews when a ChangeSpec's diff, changed files, or amend notes match configurable criteria. Each profile groups one or more mentors with shared matching rules. See docs/mentors.md (https://sase.sh/mentors/) for the full mentor system reference.

```yaml
mentor_profiles:
  - profile_name: python_review
    file_globs:
      - "*.py"
    mentors:
      - mentor_name: style_checker
        role: "Python style expert"
        focus_areas:
          - focus_name: style
            description: "PEP 8 compliance and code style"
          - focus_name: naming
            description: "Variable and function naming conventions"

  - profile_name: first_commit_review
    first_commit: true
    mentors:
      - mentor_name: architecture
        role: "Software architect"
        focus_areas:
          - focus_name: design
            description: "Overall design and architectural patterns"
```

Profile fields:

Field	Type	Required	Description
`profile_name`	string	yes	Unique name identifying this profile.
`mentors`	list	yes	List of mentor definitions (see below).
`file_globs`	list[string]	no*	Glob patterns matched against changed file paths.
`diff_regexes`	list[string]	no*	Regex patterns matched against the diff content.
`amend_note_regexes`	list[string]	no*	Regex patterns matched against commit/amend notes.
`first_commit`	bool	no	If true, match only on the first commit of a ChangeSpec.
`projects`	list[string]	no	Only match ChangeSpecs in these projects. Auto-set for local `sase.yml` profiles.

*At least one of `file_globs`, `diff_regexes`, `amend_note_regexes`, or `first_commit` must be provided per profile.

Mentor fields:

Field	Type	Required	Description
`mentor_name`	string	yes	Unique name identifying this mentor within its profile.
`role`	string	yes	Role or persona for the mentor (e.g., "Security reviewer").
`focus_areas`	list[object]	yes	List of review focus areas (see below).

Focus area fields:

Field	Type	Required	Description
`focus_name`	string	yes	Short name for this focus area (e.g., "correctness").
`description`	string	yes	Description of what this focus area reviews.

Mentors run automatically on ChangeSpecs with Ready or Mailed status when their matching criteria are met. Mentor comments are structured JSON with severity levels (error, warning, suggestion) that can be reviewed and applied through the ACE TUI's Mentor Review modal (`,m`).

Source: `src/sase/config/mentor.py`

24.4.9 metahooks

Metahooks intercept failing hooks before the summarize agent runs. They match based on the hook command (substring match) and the hook output (regex match). When a metahook matches, it can trigger specialized handling instead of the default summarization.

```yaml
metahooks:
  - name: scuba
    hook_command: sase_hg_presubmit
    output_regex: "SCUBA_ERROR.*timeout"

  - name: flaky_test
    hook_command: blaze test
    output_regex: "FLAKY"
```

Field	Type	Required	Description
`name`	string	yes	Unique identifier for this metahook.
`hook_command`	string	yes	Substring matched against the executed hook command.
`output_regex`	string	yes	Regex pattern matched against hook output (multiline).

Source: `src/sase/config/metahook.py`

24.4.10 xprompts

Defines reusable prompt snippets that can be referenced with `#name` syntax in any prompt. Supports both simple string content and structured definitions with typed inputs and Jinja2 templates.

```yaml
xprompts:
  # Simple string format
  greeting: "Hello, please review this code."

  # Structured format with inputs
  review:
    input:
      language: word
      strict: { type: bool, default: false }
    content: "Review this {{ language }} code.{{ ' Be strict.' if strict }}"

  # With tags for semantic role lookup
  my_crs:
    content: "Summarize the code review..."
    tags: [crs]
```

Xprompts defined in `sase.yml` are priority 6 out of 9 in the resolution order:

1. `.xprompts/*.md` (CWD, hidden directory)
2. `xprompts/*.md` (CWD)
3. `~/.xprompts/*.md` (home, hidden directory)
4. `~/xprompts/*.md` (home)
5. `~/.config/sase/xprompts/{project}/*.md` (project-specific)
6. `sase.yml` `xprompts:` section (local `./sase.yml` overrides global; see Deep-Merge System)
7. Plugin packages (via `sase_xprompts` entry points)
8. `<sase_package>/default_xprompts/*.md` (built-in default markdown xprompts)
9. `<sase_package>/xprompts/*.md` (built-in package xprompts)

Earlier sources win on name conflicts. File-based xprompts use YAML front matter for metadata and the file body for content.

Source: `src/sase/xprompt/loader.py`

24.4.11 xprompt_aliases

Defines raw text-level alias substitutions that are applied *before* any xprompt processing. This is useful for creating shorthand references where the alias must be present in the raw text for other processing logic (such as VCS directory-switching) to work correctly.

```yaml
xprompt_aliases:
  c: commit # #c → #commit
  p: propose # #p → #propose
  gh_sase: "gh:sase" # #gh_sase → #gh:sase
  gh_foo: "gh:foo/bar" # #gh_foo → #gh:foo/bar
```

Field	Type	Default	Description
`xprompt_aliases`	dict[string]	`{c: commit, p: propose}`	Mapping of alias name → target. Applied as text substitution

The built-in defaults provide `#c` as a shorthand for `#commit` and `#p` for `#propose`. Additional aliases can be added in user config files.

Each entry maps an alias name to a target string. When the processor encounters `#alias_name` in a prompt, it replaces it with `#target` before any other xprompt resolution occurs. Only `#`-prefixed references are substituted; the alias name must match `[a-zA-Z_][a-zA-Z0-9_]*`.

Source: `src/sase/xprompt/processor.py`

24.4.12 use_chezmoi

Enables chezmoi-aware home-file writes. When set to `true`, SASE writes generated home instructions, memory, skills, and home-directory xprompt paths through the chezmoi source tree under `~/.local/share/chezmoi/home/` instead of writing the live home files directly. For example, `~/.xprompts/` is remapped to `~/.local/share/chezmoi/home/dot_xprompts/`.

This affects initialization workflow as well as xprompt editing. `sase amd init` targets the chezmoi home source root when it needs to initialize home-level `AGENTS.md`; `sase memory init` writes home memory there and may run the configured chezmoi deploy path; `sase skill init` writes provider skill files there before optional commit, push, and apply steps.

Home-level AMD provider shims in the chezmoi source are managed as `*.md.tmpl` files containing `@{{ .chezmoi.homeDir }}/AGENTS.md`, so deployed provider files render to absolute home imports on each machine.

```yaml
use_chezmoi: true # default: false
```

Field	Type	Default	Description
`use_chezmoi`	bool	`false`	Write home-managed SASE files through the chezmoi source directory.

Source: `src/sase/config/core.py`

24.4.13 precommit_command

A shell command to run before commits (e.g., linting, formatting). If set, the commit workflow executes this command before creating a commit. An empty string (the default) means no precommit command is run.

```yaml
precommit_command: "just fix" # default: ""
```

Field	Type	Default	Description
`precommit_command`	string	`""`	Shell command to run before commits. Empty string means disabled.

Source: `src/sase/default_config.yml`, `src/sase/workflows/commit/workflow.py`

24.4.14 timezone

The timezone used for formatting timestamps in notifications, agent logs, and TUI displays.

```yaml
timezone: "America/New_York" # default: "America/New_York"
```

Field	Type	Default	Description
`timezone`	string	`"America/New_York"`	IANA timezone name for timestamp display.

24.4.15 chat_install

Configuration for chat-driven update workflows. External chat integrations can call `sase.integrations.chat_install.start_chat_install_worker()` to run the configured command in a detached worker while briefly stopping axe, syncing the registered primary workspace for the `sase` project, and restarting axe afterward.

```yaml
chat_install:
  command: "" # default: disabled
  sync_workspace: true
  timeout_seconds: 900
  restart_attempts: 3
```

Field	Type	Default	Description
`chat_install.command`	string	`""`	Shell command to run from the registered `sase` primary workspace. Empty string disables use.
`chat_install.sync_workspace`	bool	`true`	Sync the registered `sase` primary workspace via the selected VCS provider before updating.
`chat_install.timeout_seconds`	int	`900`	Maximum runtime for the update command before returning exit code `124`.
`chat_install.restart_attempts`	int	`3`	Number of axe restart attempts after the update command completes/fails.

Only one chat update worker may run at a time; a lock under `~/.sase/chat_install/install.lock` rejects concurrent starts. Worker output is written to timestamped logs under `~/.sase/chat_install/logs/`. The configuration key and state paths remain named `chat_install` for compatibility. See `docs/integrations.md` (https://sase.sh/integrations/#chat-update-worker) for the integration-facing Python API.

Source: `src/sase/default_config.yml`, `src/sase/integrations/chat_install.py`

24.4.16 mobile_gateway

Configuration for `sase mobile gateway start`, which launches the workstation-hosted Rust gateway for paired mobile clients.

```yaml
mobile_gateway:
  bind_address: "127.0.0.1"
  port: 7629
  state_dir: ""
  allow_non_loopback: false
  command: ""
  push_provider: "disabled"
  fcm_project_id: ""
  fcm_service_account_json: ""
  fcm_credential_env: ""
  fcm_dry_run: false
  push_timeout_seconds: 5
  push_retry_limit: 1
  startup_timeout_seconds: 10
```

Field	Type	Default	Description

`mobile_gateway.bind_address`	string	`"127.0.0.1"`	Host address to bind. Non-loopback values require explicit opt-in.
`mobile_gateway.port`	int	`7629`	Gateway HTTP port.
`mobile_gateway.state_dir`	string	`""`	SASE state root for gateway storage. Empty uses the Rust gateway default.
`mobile_gateway.allow_non_loopback`	bool	`false`	Allow LAN or tailnet binds after explicit user opt-in.
`mobile_gateway.command`	string	`""`	Gateway binary command override, parsed without a shell.
`mobile_gateway.push_provider`	string	`"disabled"`	Push provider: `disabled`, `test`, or `fcm`.
`mobile_gateway.fcm_project_id`	string	`""`	Firebase project ID for FCM HTTP v1.
`mobile_gateway.fcm_service_account_json`	string	`""`	Local service-account JSON path. Do not commit this file.
`mobile_gateway.fcm_credential_env`	string	`""`	Env var containing an FCM bearer token or service-account JSON.
`mobile_gateway.fcm_dry_run`	bool	`false`	Ask FCM to validate messages without delivering them.
`mobile_gateway.push_timeout_seconds`	float	`5`	Timeout per push provider HTTP attempt.
`mobile_gateway.push_retry_limit`	int	`1`	Retry attempts for best-effort push delivery.
`mobile_gateway.startup_timeout_seconds`	float	`10`	Seconds to wait for gateway readiness before exiting.

Push payloads are hint-only and must not contain bearer tokens, pairing codes, prompt bodies, response text, attachment contents, attachment tokens, or host paths. Only credential paths or environment-variable names are placed on the gateway command line. See `docs/mobile_gateway.md` (https://sase.sh/mobile_gateway/#push-hints) for setup examples and security notes.

Source: `src/sase/default_config.yml`, `src/sase/integrations/mobile_gateway.py`

24.4.17 sdd

Configuration for spec-driven development features, including prompt, tale, epic, legend, myth, research, and bead storage.

```yaml
sdd:
  version_controlled: false # default: false
```

Field	Type	Default	Description
`sdd.version_controlled`	bool	`false`	For non-bare-git projects, store SDD artifacts and beads under `sdd/` in the project repo instead of `.sase/sdd/` in the primary workspace.

When enabled, prompt snapshots, tales, epics, legends, myths, research notes, and the bead database directory are placed in the project root so they can be committed with the code. When disabled, SDD writes to a standalone `.sase/sdd/` git repo in the primary workspace for providers that support local SDD mode. Projects resolved as the built-in `bare_git` VCS provider always use version-controlled SDD under `sdd/`, even if this option is false. See `docs/sdd.md` (https://sase.sh/sdd/) for storage behavior and `docs/beads.md` (https://sase.sh/beads/) for the bead system reference.

Built-in bare-git projects also auto-create or refresh generated SDD guide files during first-use `#git:<project>` initialization, existing bare-repo registration, `#git` /workspace materialization, and the first version-controlled SDD write. Setup/materialization flows commit and push only those generated init paths with an `Initialize SDD` init commit when needed.

Running `sase sdd init` or its `sase init sdd` alias is an explicit non-bare-git opt-in: it creates or updates the project-local `sase.yml` so `sdd.version_controlled` is true, then refreshes generated SDD guide files and the directory map.

Source: `src/sase/default_config.yml`

24.4.18 bead

Configuration for the bead issue tracker.

```yaml
bead:
  push_after_commit: true # default: true (also accepts false or async)
```

Field	Type	Default	Description
`bead.push_after_commit`	bool or str	`true`	Controls the post-commit `git push` after `sase bead work`. `true` pushes synchronously (failures only warn); `false` skips the push; `async` launches a detached background push and returns immediately, logging the result to a file. `sase bead work --no-push` overrides this per-invocation.

Set to `false` for local-only checkouts, or when you would rather batch the bead-launch commit with later commits before pushing. Set to `async` to keep auto-pushing without blocking the command on remote network/credential latency. See `docs/beads.md` (https://sase.sh/beads/#sase-bead-work-id) for the full `sase bead work` flow.

Source: `src/sase/default_config.yml`

24.4.19 workspace

Controls how SASE chooses the physical location of managed workspace checkouts. See `docs/workspace.md` (https://sase.sh/workspace/#workspace-directory-layout) for the directory-layout reference and CLI workflows.

```yaml
workspace:
  root: xdg-state # "xdg-state", "adjacent", or an absolute path
  project_key: "" # explicit project-key override; empty = derive from git remote / primary path
  cleanup_ttl_days: 14 # age threshold for `sase workspace cleanup --stale`
```

Field	Type	Default	Description
`workspace.root`	string	`"xdg-state"`	Root policy. `"xdg-state"` uses the platform state dir; `"adjacent"` keeps the legacy `<primary>_<num>/` layout as an explicit opt-in; an absolute path is used as the managed-root base. `SASE_WORKSPACE_ROOT` overrides this base directory.
`workspace.project_key`	string	`""`	Override the per-project namespace under managed roots. Empty derives a stable key from a single git remote slug or the primary-path basename plus a short hash.

`workspace.cleanup_ttl_days`	int	`14`	Minimum age (in days) of an unclaimed managed checkout before `sase workspace cleanup --stale` will remove it.

Platform defaults for the `xdg-state` policy:

Platform	Managed root
Linux	`$XDG_STATE_HOME/sase/workspaces` (falls back to `~/.local/state/sase/workspaces`)
macOS	`~/Library/Application Support/sase/workspaces`
Windows	`%LOCALAPPDATA%\sase\workspaces`

Numeric identity is the same on every root policy: `#0` is the primary checkout, `#1` – `#9` are reserved, and managed claim workspaces start at `#10`. See `docs/workspace.md` (https://sase.sh/workspace/#numeric-identity) for the full identity model and backup/container/NFS caveats.

For non-adjacent policies, physical checkouts live under `<managed-root>/<project_key>/<project>_<num>/`. For example, `workspace.root: /mnt/sase-workspaces` with project key `github.com_org_repo` places workspace `#10` at `/mnt/sase-workspaces/github.com_org_repo/<project>_10/`. When `SASE_WORKSPACE_ROOT` is set, it supplies the same `<managed-root>` base for the process.

Existing adjacent checkouts are not moved automatically by the default. Run `sase workspace migrate --to xdg-state` to carry legacy `<primary>_<num>/` directories into the managed root, or set `workspace.root: adjacent` explicitly to keep the old sibling layout.

`sase workspace open NUM -r "<reason>"` is an explicit preparation command for a checkout you plan to use outside a normal `sase run` launch. It uses the same root policy when it materializes the checkout, backs up uncommitted local changes through the active VCS provider, cleans the checkout, checks out and syncs the provider default parent revision, and prints the resulting path. For manual scratch work, choose a claim-range number such as `10`; `#0` is the primary checkout and `#1` through `#9` are reserved compatibility numbers.

Source: `src/sase/default_config.yml`, `src/sase/workspace_provider/store.py`

24.4.20 telemetry

Configures Prometheus-based telemetry for monitoring sase internals. See docs/telemetry.md (https://sase.sh/telemetry/) for the full telemetry reference including CLI commands, metric catalog, and monitoring stack setup.

```yaml
telemetry:
  enabled: false
  prometheus:
    exposition_port: 9464
    pushgateway_url: "localhost:9091"
  health_thresholds:
    error_rate_warn: 10.0
    error_rate_critical: 25.0
    retry_rate_warn: 10.0
    retry_rate_critical: 25.0
    p95_latency_warn: 300.0
    p95_latency_critical: 600.0
```

Field	Type	Default	Description
`telemetry.enabled`	bool	`false`	Enable or disable telemetry globally.
`telemetry.prometheus.exposition_port`	int	`9464`	HTTP server port for metric exposition.
`telemetry.prometheus.pushgateway_url`	str	`localhost:9091`	Prometheus Push Gateway address.
`telemetry.health_thresholds.error_rate_warn`	float	`10.0`	Error rate % threshold for WARN health status.
`telemetry.health_thresholds.error_rate_critical`	float	`25.0`	Error rate % threshold for CRITICAL status.
`telemetry.health_thresholds.retry_rate_warn`	float	`10.0`	Retry rate % threshold for WARN health status.
`telemetry.health_thresholds.retry_rate_critical`	float	`25.0`	Retry rate % threshold for CRITICAL status.
`telemetry.health_thresholds.p95_latency_warn`	float	`300.0`	P95 latency threshold (seconds) for WARN status.
`telemetry.health_thresholds.p95_latency_critical`	float	`600.0`	P95 latency threshold (seconds) for CRITICAL.

Source: `src/sase/default_config.yml`, `src/sase/telemetry/_config.py`

24.5 Environment Variables

24.5.1 LLM Provider

Variable	Description
`SASE_MODEL_TIER_OVERRIDE`	Force all LLM invocations to a specific tier (`large` or `small`).
`SASE_MODEL_SIZE_OVERRIDE`	Legacy alias for `SASE_MODEL_TIER_OVERRIDE` (`big` or `little`).
`SASE_LLM_LARGE_ARGS`	Extra CLI args appended for `large` tier invocations (any provider).
`SASE_LLM_SMALL_ARGS`	Extra CLI args appended for `small` tier invocations (any provider).
`SASE_CLAUDE_LARGE_ARGS`	Claude-specific extra args for `large` tier (fallback if generic unset).
`SASE_CLAUDE_SMALL_ARGS`	Claude-specific extra args for `small` tier (fallback if generic unset).
`SASE_CODEX_PATH`	Path to the Codex CLI binary (default: PATH lookup, then NVM_BIN/codex).
`SASE_CODEX_LARGE_ARGS`	Codex-specific extra args for `large` tier (fallback if generic unset).
`SASE_CODEX_SMALL_ARGS`	Codex-specific extra args for `small` tier (fallback if generic unset).
`SASE_CODEX_DISABLE_SHADOW_HOME`	Set to `1` to launch Codex with the inherited `CODEX_HOME`.
`SASE_QWEN_PATH`	Path to the Qwen Code CLI binary (default: `qwen`).
`SASE_QWEN_LARGE_ARGS`	Qwen-specific extra args for `large` tier (fallback if generic unset).
`SASE_QWEN_SMALL_ARGS`	Qwen-specific extra args for `small` tier (fallback if generic unset).
`SASE_OPENCODE_PATH`	Path to the OpenCode CLI binary (default: `opencode`).
`SASE_OPENCODE_LARGE_ARGS`	OpenCode-specific extra args for `large` tier (fallback if generic unset).
`SASE_OPENCODE_SMALL_ARGS`	OpenCode-specific extra args for `small` tier (fallback if generic unset).

`SASE_AGY_PATH`	Path to the Antigravity CLI binary (default: `agy`).
`SASE_AGY_PRINT_TIMEOUT`	Override the `agy --print-timeout` Go duration (default: `24h`).
`SASE_AGY_LARGE_ARGS`	Antigravity-specific extra args for `large` tier (fallback if generic unset).
`SASE_AGY_SMALL_ARGS`	Antigravity-specific extra args for `small` tier (fallback if generic unset).

For the per-provider args, the generic `SASE_LLM_*_ARGS` variables are checked first. If unset, the provider-specific variable is used as a fallback. Values are split on whitespace and appended to the CLI command.

SASE-launched Codex subprocesses use a disposable shadow `CODEX_HOME` by default. The shadow home is created under `~/.cache/sase/codex_home/`, receives a copy of the real `config.toml`, symlinks other Codex home entries back to the real home, and is removed when the subprocess exits. If the real Codex home does not provide `AGENTS.override.md` or `AGENTS.md`, SASE also links `~/AGENTS.md` into the shadow as Codex's `$CODEX_HOME/AGENTS.md` fallback. This prevents Codex runtime config rewrites from dirtying the user-managed Codex config while preserving auth, hooks, skills, logs, and caches.

Qwen Code uses `qwen --input-format text --output-format stream-json --yolo --model <model>` and expects users to configure Qwen auth through Qwen's supported settings path. Qwen OAuth free tier access ended on 2026-04-15; use API keys, Alibaba Cloud Coding Plan, OpenRouter, Fireworks, or another Qwen-supported provider.

OpenCode uses `opencode run --format json --dangerously-skip-permissions --model <provider/model> --dir <cwd> <prompt>` and expects users to configure OpenCode auth/settings through its normal XDG paths. OpenCode model names usually include a provider prefix; use `opencode models` to list models in your configured environment.

24.5.2 VCS Provider

Variable	Description
`SASE_VCS_PROVIDER`	Override VCS provider selection (`git`, `hg`, or `auto`).
`SASE_WORKSPACE_ROOT`	Override the workspace-root base for this process. Use an absolute path; `WorkspaceStore` appends `<project_key>/<project>_<num>/` for managed checkouts.
`SASE_BUG_ID`	Bug ID for PR workflows. When set and non-zero, injects `BUG=<id>` into PR tags and ChangeSpec.
`SASE_BEAD_ID`	Bead ID for commit workflows. When set, `sase commit` automatically tags the commit message.
`SASE_DISABLE_COMMIT_STOP_HOOK`	Disable commit finalization for this process.
`SASE_LINKED_REPOS_JSON`	Resolved linked-repo metadata passed to launched agents.
`SASE_LINKED_REPO_<ENV_NAME>_DIR`	Workspace-matched directory for one configured linked repo.

24.5.3 Plugin System

These switches affect plugin-provided resource loading. The VCS, workspace, and LLM provider registries load provider entry points directly.

Variable	Description
`SASE_DISABLE_PLUGINS`	Disable plugin-provided xprompts, workflows, and config defaults.
`SASE_DISABLE_PLUGIN_XPROMPTS`	Disable plugin-provided xprompt and workflow files.
`SASE_DISABLE_PLUGIN_CONFIG`	Disable plugin-provided `default_config.yml` files and config xprompts.

24.5.4 State Root

Variable	Description
`SASE_HOME`	Override the SASE state root. Defaults to `~/.sase`; project files, chats, artifacts, notifications, dismissed bundles, saved groups, and logs move under this root.

24.5.5 General

Variable	Description
`SASE_TMPDIR`	Override the temp directory for all sase operations. Falls back to system default when unset.
`SASE_AGENT_AUTO_APPROVE_PLAN_ACTION`	Plan-specific auto-approval action for an agent; currently `approve` or `epic`.
`SASE_AGENT_AUTO_PLAN_ACTION`	Backward-compatible alias for `SASE_AGENT_AUTO_APPROVE_PLAN_ACTION`.
`SASE_AGENT_AUTO_APPROVE`	Legacy boolean auto-approve flag; maps plan submissions to normal approval.
`SASE_XPROMPT_LSP_CMD`	Override the command used by `sase lsp` to launch the xprompt language server.
`SASE_CORE_DIR`	Preferred `sase-core` source checkout for `Justfile` Rust build/install targets; overrides `../sase-core`.
`SASE_PYTEST_WORKERS`	Override the xdist worker count used by `just test`, `just test-slow`, `just test-visual`, and `just test-cov`.
`SASE_JUST_INVOCATION_DIR`	Internal value set by `just` so test selectors are normalized from the caller's directory.

24.5.6 Workspace Management (Internal)

These are set automatically by sase when launching agent subprocesses and are not intended for manual use. Workspace plugins declare an env-var prefix, then SASE passes `<PREFIX>_PRE_ALLOCATED`, `<PREFIX>_WORKSPACE_NUM`, and `<PREFIX>_WORKSPACE_DIR` into the child process. Built-in prefixes include `SASE_CD` for `#cd` and `SASE_GIT` for `#git`; plugin packages may add prefixes such as `SASE_GH` for GitHub. The launcher clears inherited `SASE_*_PRE_ALLOCATED`, `SASE_*_WORKSPACE_NUM`, and `SASE_*_WORKSPACE_DIR` variables before applying the current launch's values so follow-up agents cannot inherit stale workspace claims.

Variable	Description
`SASE_SYNC_CWD`	Working directory override for sync operations.
`<PREFIX>_PRE_ALLOCATED`	Set to `"1"` when a workspace provider has pre-allocated a launch context.
`<PREFIX>_WORKSPACE_NUM`	Pre-allocated workspace number, or `0` for non-claiming directory contexts.
`<PREFIX>_WORKSPACE_DIR`	Pre-allocated workspace directory path.

`SASE_CD_*`, `SASE_GIT_*`, ...	Concrete forms for built-in and plugin-provided workspace prefixes.

24.6 CLI Flags

Command groups that default to a nested `list` command still parse flags at the subcommand level. Use the explicit `list` form when passing list options, such as `sase notify list -j`, `sase memory list -j`, or `sase workspace list --json`.

24.6.1 `sase ace`

Flag	Values	Default	Description
`[query]`	string	last saved query or `!!!`	Query string for filtering ChangeSpecs.
`-m, --model-tier`	`large`, `small`	-	Override model tier for all LLM invocations.
`-M, --model-size`	`big`, `little`	-	Deprecated alias for `--model-tier`.
`-p, --profile`	optional path	-	Profile the TUI session with pyinstrument (default output `$SASE_TMPDIR/ace_profile_<ts>.txt`); after exit, print a shortened path and copy it to the system clipboard when possible.
`-r, --refresh-interval`	int (seconds)	`10`	Auto-refresh interval (0 to disable).
`-R, --restart-axe`	flag	-	Restart the axe daemon on startup (no-op if axe is not running).
`-t, --tab`	`changespecs`, `agents`, `axe`	`agents`	Tab to focus on startup.
`-T, --tmux`	flag	-	Launch ACE in a new tmux window named `sase_tmux_<N>` and print the session/window target for external control.
`-x, --no-axe`	flag	-	Disable auto-starting the axe daemon.
`-v, --vcs-provider`	`git`, `hg`, `auto`	-	Override VCS provider.

24.6.2 `sase axe`

Flag	Values	Default	Description
`-v, --vcs-provider`	`git`, `hg`, `auto`	-	Override VCS provider.

24.6.3 `sase axe start`

Flag	Values	Default	Description
`-q, --query`	string	`""` (all)	Query string for filtering ChangeSpecs.
`-H, --max-hook-runners`	int	config or `3`	Maximum concurrent hook runners.

`-A, --max-agent-runners`	int	config or `3`	Maximum concurrent agent runners.
`-z, --zombie-timeout`	int (seconds)	config or `7200`	Timeout before marking a hook/workflow as a zombie.

For `sase axe start`, CLI flags take precedence over values from the `axe` config section in `sase.yml`. If neither is set, the built-in defaults from `default_config.yml` are used.

24.6.4 `sase repro`

Agents-tab reproduction bundles capture and replay the loader/apply sequence used to render agent rows. The command is intended for debugging row disappearance, reappearance, and duplicate-parent regressions; see Agents Tab Reproduction Bundles (https://sase.sh/ace/#agents-tab-reproduction-bundles).

Form	Flag	Values	Default	Description
`sase repro capture agents-tab`	`--output`	path	required	Directory where `agents_tab_repro.json` and capture artifacts are written.
`sase repro capture agents-tab`	`--commit-safe`	flag	enabled	Redact local names and paths for a shareable bundle.
`sase repro capture agents-tab`	`--no-commit-safe`	flag	-	Keep unredacted local identifiers in the capture.
`sase repro capture agents-tab`	`--size`	`WxH`	`120x40`	Terminal size label stored with the bundle.
`sase repro capture agents-tab`	`--json`	flag	-	Emit a machine-readable capture result.
`sase repro replay`	`path`	path	required	Bundle JSON file or bundle directory to replay.
`sase repro replay`	`--assert-stable`	flag	-	Exit non-zero if replay invariants fail.
`sase repro replay`	`--json`	flag	-	Emit a machine-readable replay verdict.
`sase repro replay`	`--write-artifacts`	path	-	Directory for replay screen text and SVG artifacts.
`sase repro replay`	`--size`	`WxH`	`120x40`	Headless terminal size used for replay.

24.6.5 `sase axe stop`

No flags. Stops the running axe orchestrator.

24.6.6 `sase axe maintenance`

Maintenance mode pauses scheduled lumberjack ticks without stopping the orchestrator.

Command	Flags / exit code	Description
`sase axe maintenance enter`	`-r, --reason` required	Write the maintenance marker with a reason.
`sase axe maintenance exit`	exits 0	Remove the marker if present.

`sase axe maintenance status`	exits 0 when active, 1 when inactive	Print the active marker reason, PID, timestamp.

See axe.md — Maintenance Mode (https://sase.sh/axe/#maintenance-mode) for the runtime behavior.

24.6.7 `sase commit`

Dispatches a commit, proposal, or PR via the VCS provider layer. See commit_workflows.md (https://sase.sh/commit_workflows/) for the full flow, payload, checkpoint, and resume semantics.

Flag	Values	Default	Description
`-m, --message`	string	-	Commit message (mutually exclusive with `-M`).
`-M, --message-file`	path	-	File containing the commit message / PR description (mutually exclusive with `-m`).
`-f, --file`	path (repeatable)	stage all	Specific file to stage. Repeat for multiple; omit to stage everything.
`-n, --name`	string	-	Branch/CL name (required for `create_pull_request`).
`-B, --bug-id`	int	`$SASE_BUG_ID`	Bug ID to associate with the commit.
`-c, --checkout-target`	string	`HEAD~1`	Branch point for PR creation.
`-p, --parent`	ChangeSpec name	auto	Parent ChangeSpec name (overrides branch-based auto-detection). Unresolvable values are dropped.
`-r, --resume`	flag	-	Resume a previously-checkpointed commit after manual conflict resolution.
`-s, --status`	`wip` / `draft` / `ready`	`$SASE_PR_STATUS` / draft	ChangeSpec status override for PRs.
`-t, --type`	`commit` / `propose` / `pr` …	`$SASE_COMMIT_METHOD`	Commit method — full names (`create_commit`, etc.) and short aliases are both accepted.

24.6.8 `sase changespec search`

Flag	Values	Default	Description
`query`	string	(required)	Query string for filtering ChangeSpecs.
`-f, --format`	`plain`, `rich`, `markdown`	`rich`	Output format (`markdown` for agent-friendly output).

Search uses the normal active-project discovery scope. Inactive and sibling projects are omitted from this CLI path; run `sase project list --state all` or `sase project show <project>` to inspect hidden projects, then reactivate the project with `sase project activate <project>` before using normal search and launch surfaces for new work.

24.6.9 `sase changespec migrate-extension`

One-time cleanup for older installs: renames legacy ProjectSpec files under `~/.sase/projects` from `.gp` to `.sase`, including archive siblings. Current readers still accept `.gp` as a fallback, so migration is not required before using SASE; it just normalizes on-disk filenames to the canonical extension.

If a `.sase` sibling already exists with identical contents, the redundant `.gp` copy is removed. If the sibling differs, the command reports a conflict and preserves both files unless `--force` is set.

Flag	Values	Default	Description
`--force`	flag	-	Replace an existing differing `.sase` sibling with the legacy `.gp` file.
`--projects-dir`	path	`~/.sase/projects/`	Override the project root scanned for legacy `.gp` files.

24.6.10 `sase project`

With no subcommand, `sase project` defaults to `sase project list`. Project lifecycle state is stored as `PROJECT_STATE` metadata in the ProjectSpec header; missing state means `active`.

Form	Flags	Description
`sase project list`	`-s, --state active|inactive|sibling|all`	List projects in one lifecycle state; default is `active`.
`sase project list`	`-j, --json`	Emit machine-readable lifecycle records.
`sase project show <project>`	`-j, --json`	Show state, source, project/archive files, workspace, launchability, warnings.
`sase project set-state <project> <state>`	`-f, --force`	Set `active`, `inactive`, or `sibling`.
`sase project activate <project>`	`-f, --force`	Alias for `set-state <project> active`; `--force` has no effect for `active`.
`sase project deactivate <project>`	`-f, --force`	Alias for `set-state <project> inactive`.

Deactivating refuses projects with live `RUNNING` claims or live artifact markers (`running.json`, `waiting.json`, or `pending_question.json`) unless `--force` is passed. Legacy `archive` and `close` aliases still set `inactive`, and legacy `PROJECT_STATE: archived` / `PROJECT_STATE: closed` files are read as inactive. The system-managed `home` project cannot be mutated through this command. Normal launch and discovery surfaces default to active projects; use `list --state all` or `show` for historical or sibling inspection, then `activate` before launching normal work in a hidden project. The `sibling` state (a legacy backing state name) is intended for configured linked repository records used by `sase workspace open -p <linked_repo> -r "<reason>" <workspace_num>`.

ACE exposes the same lifecycle mutations through the Project Management panel at `,p`. That panel also supports marks for bulk lifecycle operations, ProjectSpec editing through `$EDITOR`, and confirmed deletion of whole SASE project directories under `~/.sase/projects/` without deleting workspace checkouts. The temporary model override uses `,o` by default.

24.6.11 `sase revert`

Flag	Values	Default	Description
`name`	string	(required)	NAME of the ChangeSpec to revert.

24.6.12 `sase restore`

Flag	Values	Default	Description
`[name]`	string	-	NAME of the reverted ChangeSpec to restore.
`-l, --list`	flag	-	List all reverted ChangeSpecs.

24.6.13 `sase run`

Flag	Values	Default	Description
`[query]`	string	-	Prompt text, inline reference (`#name`), standalone workflow reference (`#!name`), or `.` for history picker.
`-d, --daemon`	flag	-	Run as a detached background agent (appears in TUI Agents tab).
`-l, --list`	flag	-	List all available chat history files.
`-r, --resume`	string	-	Resume a previous conversation by agent name or history file basename.

When invoked with no arguments, opens `$EDITOR` for composing a prompt interactively. When invoked with `.`, opens a prompt history picker. Multi-prompt queries (containing `---` separators) are auto-detected and launched as sequential daemon agents.

24.6.14 `sase repro`

`sase repro` captures and replays debugging bundles for narrow, reproducible TUI bug classes. The current target is the Agents-tab loader/apply sequence used to diagnose row disappearance, reappearance, and duplicate workflow parents.

Form	Flags	Description
`sase repro replay <path>`	`--assert-stable`, `--json`, `--write-artifacts <dir>`, `--size`	Replay a bundle JSON file or bundle directory through the headless TUI harness.
`sase repro capture agents-tab`	`--output <dir>`, `--commit-safe`, `--no-commit-safe`, `--size`, `--json`	Capture a baseline bundle from current filesystem state. `--commit-safe` redaction is the default.

Use the in-TUI `,C` capture when a transient row-list bug has just happened in a live ACE session. The CLI capture path is out-of-band: it loads current filesystem state and cannot reconstruct refreshes that already passed through the running TUI.

24.6.15 `sase xprompt`

With no subcommand, `sase xprompt` defaults to `sase xprompt list`.

24.6.16 `sase xprompt expand`

Flag	Values	Default	Description
`[prompt]`	string	stdin	Prompt text to expand (reads from stdin if omitted).
`-t, --trace`	flag	-	Print expansion trace to stderr showing resolved references.

24.6.17 `sase xprompt explain`

Flag	Values	Default	Description
`workflow_name`	string	(required)	Workflow name to explain.
`[args]`	string	-	Positional arguments for the workflow.
`-a, --arg`	string	-	Named argument as `KEY=VALUE` (repeatable).

24.6.18 `sase xprompt list`

No flags. Outputs a JSON array of all available xprompts with name, type, source, inputs, tags, `is_skill`, and preview. Clients that insert references should prefer `kind` / `insertion` metadata when present so standalone workflows are inserted as `#!name` and inline-capable entries, including markdown multi-agent xprompts, are inserted as `#name`. Slash skill completion clients should filter to entries where `is_skill` is `true`.

24.6.19 `sase xprompt graph`

Flag	Values	Default	Description
`[workflow_name]`	string	-	Workflow name to graph. Lists all workflows if omitted.
`-f, --format`	`mermaid`, `text`	`mermaid`	Output format for the DAG visualization.

24.6.20 `sase xprompt catalog`

Flag	Values	Default	Description
`-o, --out`	path	tempdir	Directory where the rendered PDF should be saved.

24.6.21 `sase init`

Bare `sase init` is the onboarding coordinator for SASE-managed resources. It runs read-only planners for AMD, memory, SDD, and skills, prints a grouped summary, and prompts once per initializer that needs work when stdin is interactive. Non-interactive runs never prompt; they print the drift summary and ask the caller to rerun with `--yes`. The AMD planner only generates managed project `AGENTS.md` from bare `sase init` when the current project's own `./sase.yml` sets `amd_h1_title`.

Advanced deploy controls stay on explicit subcommands such as `sase init amd --check`, `sase init memory --no-commit`, and `sase skill init --no-push`.

Flag	Values	Default	Description
`-c, --check`	flag	-	Report initialization drift without writing; exits non-zero when changes are needed.
`-y, --yes`	flag	-	Run every needed initializer in AMD, memory, SDD, skills order without prompting.

24.6.22 `sase amd`

With no subcommand, `sase amd` defaults to `sase amd list`.

Form	Flags	Description
`sase amd`	-	Show the same read-only agent-markdown inventory as `sase amd list`.
`sase amd list`	-	Inspect project, home, and chezmoi `AGENTS.md` files and provider shims.
`sase amd init`	`-c, --check`	Create or refresh `AGENTS.md` files and shims for the selected AMD root or roots, or report drift.
`sase init amd`	`-c, --check`	Compatibility alias for `sase amd init`.

24.6.23 `sase memory`

With no subcommand, `sase memory` defaults to `sase memory list`.

Form	Flags	Description
`sase memory`	-	Show the same read-only memory context dashboard as `sase memory list`.
`sase memory list`	-	Show loaded, referenced, available, and missing memory files for the current launch context.
`sase memory read <path>`	`-r, --reason <reason>` required	Agent-side read of a `type: long` memory note without leading frontmatter, plus an audit event.
`sase memory write`	`--title`, `--target` or `--slug`, repeatable `--evidence`, `--from-chat`, `--keyword`, `--body`, `--file`, `--allow-large`, `--manual-author`, `--notify`, `--json`	Create an attributable long-term memory proposal without modifying canonical memory files.
`sase memory review [id]`	`--list`, `--show`, `--approve`, `--edit`, `--reject`, `--all`, `--target`, `--edited-file`, `--reason`, `--json`	Human review of pending memory proposals; a bare TTY command opens the interactive review app.
`sase memory log`	`--path`, `--agent`, `--id`, `--include`, `--json`	Summarize or inspect audited memory reads, optionally including proposal and review events.

Examples:

```
# read requires SASE agent identity; write requires agent identity unless --manual-author is used for demos
sase memory read generated_skills.md --reason "Need generated skill context"
sase memory write --title "Generated skills" --slug generated_skills --evidence chat:abc123 --body "Durable memory body"
--notify
sase memory review --list
sase memory review mem-20260523-142233-a1b2c3d4 --approve
sase memory log
sase memory log --include proposals
sase memory log --path generated_skills.md
sase memory log --id <read-id>
```

24.6.24 `sase memory init`

Creates or refreshes project and home memory roots and keeps `AGENTS.md` memory references reachable. It creates a minimal `AGENTS.md` when absent for repositories that are not opted into AMD-managed instructions, and it still repairs provider instruction shims for compatibility. Direct home shims use absolute `@/path/to/home/AGENTS.md` imports; when `use_chezmoi` is enabled, home shim sources are written as `*.md.tmpl` files. When the project-local `./sase.yml` sets `amd_h1_title`, memory init synchronizes that project's AMD-managed short/long memory blocks and inserts missing long-memory `description` frontmatter. By default it also tries to commit, rebase-pull, and push generated project-side files. `sase init memory` is a compatibility alias for this command. Generated linked-repository memory lists direct paths for `workspace.strategy: none` linked repos and only includes `sase workspace open` guidance when a configured linked repo uses numbered workspace resolution.

Flag	Values	Default	Description
`-c, --check`	flag	-	Report memory initialization drift without writing project or home files.
`-C, --no-commit`	flag	-	Write files, but skip only the project git commit/pull/push path; home deployment still follows config.

24.6.25 `sase init sdd`

`sase init sdd` is an alias for `sase sdd init`. It enables version-controlled SDD in the project-local `sase.yml`, then creates or refreshes generated SDD README files and the directory map asset. Bare-git projects run the generated-file refresh automatically during repository setup and first SDD writes, but the explicit command remains available for manual opt-in, refresh, and `--check` audits.

Flag	Values	Default	Description
`-c, --check`	flag	-	Report SDD config and generated-file drift without writing files.
`-p, --path`	path	`./sdd` or `./.sase/sdd`	SDD root or project root path.

24.6.26 `sase skill`

With no subcommand, `sase skill` defaults to the read-only `sase skill list` dashboard. It reports loaded skill sources, provider targets, and deployed-file drift without writing files. `sase skill init` generates and deploys agent skill files from xprompt sources marked with the `skill` field. Generated skill files begin with a `sase skill use` directive so agent-side skill use can be audited and later summarized with `sase skill log`, unless the source

sets `log_skill_use: false`. See xprompt.md — Skill Field (https://sase.sh/xprompt/#skill-field) for the skill-source contract and provider targets. Existing files are skipped in non-interactive runs unless `--force` is passed; interactive runs prompt before overwriting. `sase init skills` is a compatibility alias for `sase skill init`.

Form	Flags	Description
`sase skill`	-	Show the same read-only dashboard as `sase skill list`.
`sase skill list`	-	Inspect generated skill sources, provider targets, and deployed-file drift.
`sase skill init`	`-f, --force`	Overwrite existing deployed skill files without confirmation.
`sase skill init`	`-n, --dry-run`	Show what would be written without writing files.
`sase skill init`	`-p, --provider {claude,agy,codex,opencode,qwen}`	Deploy only for one provider.
`sase skill init`	`-A, --no-apply`	With `use_chezmoi`, skip `chezmoi apply` after generated files are committed and pushed.
`sase skill init`	`-C, --no-commit`	With `use_chezmoi`, skip the entire git commit, push, and apply sequence.
`sase skill init`	`-P, --no-push`	With `use_chezmoi`, commit generated files but skip pull/rebase, push, and `chezmoi apply`.
`sase skill log`	`-a, --agent`; `-R, --runtime`; `-s, --skill`; `-i, --id`; `-j, --json`	Summarize or inspect audited generated skill-use events.
`sase skill use`	`-r, --reason <reason>` required	Agent-side audit event recording that the current agent is using a generated skill.
`sase init skills`	same as `sase skill init`	Compatibility alias for `sase skill init`.

24.6.27 `sase workspace`

Workspace commands inspect and maintain the managed checkout registry for the inferred project, or for the project named by `-p/--project`. With no subcommand, `sase workspace` defaults to `sase workspace list` with default options. Use `sase workspace list -p <project>` or `sase workspace list --json` when passing list flags.

Command	Flag / argument	Values	Description
`sase workspace list`	`-p, --project`	project name	Query a project other than the one inferred from the current directory.
`sase workspace list`	`-j, --json`	flag	Emit a machine-readable JSON object.
`sase workspace path`	`workspace_num`	integer	Workspace number to resolve; `0` is the primary checkout and managed claims normally start at `10`.
`sase workspace path`	`-p, --project`	project name	Query a project other than the inferred one.

`sase workspace open`	`workspace_num`	integer	Workspace number to materialize, prepare, and print.
`sase workspace open`	`-p, --project`	project name	Query a project other than the inferred one; for configured linked repos, pass the linked repo name.
`sase workspace open`	`-r, --reason`	text	Required non-empty reason for opening and preparing the workspace.
`sase workspace open`	`-P, --print`	flag	Explicitly print the prepared path; this is also the current default behavior.
`sase workspace open`	`-c, --clean`	flag	Compatibility flag for the default prepare/clean/sync behavior.
`sase workspace cleanup`	`-p, --project`	project name	Clean a project other than the inferred one.
`sase workspace cleanup`	`-s, --stale`	flag	Remove unclaimed managed checkouts older than `workspace.cleanup_ttl_days`.
`sase workspace cleanup`	`-i, --include-shares`	flag	Also consider workflow-share managed checkouts for removal.
`sase workspace cleanup`	`-n, --dry-run`	flag	Report planned removals without touching the filesystem.
`sase workspace repair`	`-p, --project`	project name	Repair a project other than the inferred one.
`sase workspace repair`	`-n, --dry-run`	flag	Report registry/filesystem reconciliation without writing.
`sase workspace migrate`	`-p, --project`	project name	Migrate a project other than the inferred one.
`sase workspace migrate`	`-t, --to`	`xdg-state`	Target managed root policy for migration.
`sase workspace migrate`	`-s, --symlink-transition`	flag	Leave `<primary>_<num>` symlinks pointing to migrated managed checkouts.
`sase workspace migrate`	`-f, --finalize`	flag	Remove transition symlinks left behind by a prior migration.
`sase workspace migrate`	`-n, --dry-run`	flag	Report planned migration or finalization actions without touching files or the registry.

For built-in bare-git projects, `sase workspace open` may initialize generated SDD guide files in the primary checkout before materializing a numbered workspace. `sase workspace list` and `sase workspace path` remain read-only and do not run SDD initialization.

24.6.28 `sase bead`

With no subcommand, `sase bead` defaults to `sase bead list`.

Flag	Values	Default	Description
subcommand	`init`, `create`, `list`, `show`, `ready`, `open`, `update`, `close`, `rm`, `dep`, `blocked`, `sync`, `stats`, `doctor`, `onboard`, `work`	`list`	Bead subcommand

`sase bead create`

Flag	Values	Default	Description
`-t, --title`	string	(required)	Issue title
`-T, --type`	string	(required)	Bead type: `plan(<file>)`, `plan(<file>,<parent>)`, or `phase(<parent_id>)`
`-d, --description`	string	-	Issue description
`-a, --assignee`	string	-	Assignee name
`--tier`	`plan`, `epic`, `legend`	-	Plan-bead tier
`-c, --changespec`	ChangeSpec name	-	Attach ChangeSpec metadata to a plan bead
`-b, --bug-id`	string	-	Bug ID for the attached ChangeSpec; requires `--changespec`
`-E, --epic-count`	positive integer	-	Number of epics proposed by a legend plan bead

`sase bead list`

Flag	Values	Default	Description
`-s, --status`	`open`, `in_progress`, `closed`	-	Filter by status (repeatable)
`-t, --type`	`plan`, `phase`	-	Filter by type (repeatable)
`--tier`	`plan`, `epic`, `legend`	-	Filter by plan-bead tier (repeatable)

`sase bead search`

Flag	Values	Default	Description
`query`	string	(required)	Literal non-empty text to search for
`-c, --color`	`auto`, `always`, `never`	`auto`	Color mode for compact output
`-f, --format`	`compact`, `json`, `full`	`compact`	Output format
`-n, --limit`	non-negative integer	(unlimited)	Maximum results to print; `0` also means unlimited
`-s, --status`	`open`, `in_progress`, `closed`	-	Filter by status (repeatable); all statuses are searched by default
`--tier`	`plan`, `epic`, `legend`	-	Filter by plan-bead tier (repeatable)
`-t, --type`	`plan`, `phase`	-	Filter by type (repeatable)

`sase bead show`

Flag	Values	Default	Description
`id`	string	(required)	Issue ID

`sase bead open`

Flag	Values	Default	Description
`id`	string	(required)	Issue ID to reopen

`sase bead update`

Flag	Values	Default	Description
`id`	string	(required)	Issue ID to update
`-s, --status`	`open`, `in_progress`, `closed`	-	Change status
`-t, --title`	string	-	Change title
`-d, --description`	string	-	Change description
`-n, --notes`	string	-	Change notes
`-D, --design`	path	-	Change plan path
`-a, --assignee`	string	-	Change assignee
`--tier`	`plan`, `epic`, `legend`	-	Change plan-bead tier
`-E, --epic-count`	positive integer	-	Change legend epic count

`sase bead close`

Flag	Values	Default	Description
`ids`	string	(required)	One or more issue IDs
`-r, --reason`	string	-	Optional close reason text

`sase bead rm`

Flag	Values	Default	Description
`id`	string	(required)	Issue ID to remove

`sase bead dep add`

Flag	Values	Default	Description
`issue`	string	(required)	Issue that depends
`depends_on`	string	(required)	Issue being depended upon

`sase bead sync`

Flag	Values	Default	Description
`-s, --status`	flag	-	Check sync status without committing

`sase bead work`

Flag	Values	Default	Description
`id`	string	(required)	Epic or legend plan bead ID.
`-n, --dry-run`	flag	-	Print the wave plan and rendered multi-prompt without mutating state.
`-P, --no-push`	flag	-	Commit launched bead state locally but skip the post-commit `git push`.
`-y, --yes`	flag	-	Skip the launch confirmation prompt when launching phase or epic agents.

24.6.29 `sase sdd`

`sase sdd` manages SDD prompt/artifact documentation and frontmatter links. Every subcommand accepts `-p/--path`, which may point at an SDD root or at a project root containing `sdd/`. With no subcommand, `sase sdd` defaults to `sase sdd list`. `sase init sdd` is an alias for `sase sdd init`.

Subcommand	Flags	Description
`init`	`-p/--path`, `-c/--check`	Create or refresh `sdd/README.md`, tier READMEs, and the directory map asset; `--check` reports drift without writing
`list`	`-p/--path`, `-k/--kind`, `-j/--json`	List SDD markdown files; kind is `prompts`, `tales`, `epics`, `legends`, or `all`
`links`	`-p/--path`, `-j/--json`	List prompt/artifact frontmatter links and bidirectional status
`validate`	`-p/--path`, `-j/--json`, `-q/--quiet`, `--strict`, `-W/--show-warnings`	Validate SDD frontmatter links; strict mode turns unpaired historical files into errors
`repair-links`	`-p/--path`, `-w/--write`	Infer unambiguous prompt/artifact pairs and optionally write link fixes

24.6.30 `sase validate`

`sase validate` is the top-level SASE validation command. It currently runs `sase init --check` and `sase sdd validate`, prints one status line per check, and exits non-zero if any check fails. Because `sase init --check` includes home-level memory and skill deployment surfaces, this command can fail on user/home initialization drift even when repository-local SDD validation passes.

24.6.31 `sase doctor`

Runs the read-only support diagnostics bundle for the active runtime, configuration, provider setup, project/workspace state, bead store, agent index, and telemetry when configured. Default mode is bounded and safe to run before asking for help; deep mode adds slower read-only checks.

Flag	Values	Default	Description
`-j`, `--json`	flag	-	Emit the `schema_version: 1` JSON support report.
`-v`, `--verbose`	flag	-	Show every check plus bounded details in human output.
`-D`, `--deep`	flag	-	Include slower read-only deep checks.
`-s`, `--strict`	flag	-	Exit non-zero for warnings as well as errors.
`-L`, `--list-checks`	flag	-	List registered default and deep check ids without running them.
`-C`, `--check`	id/group	repeat	Run only the selected check id or group; may be passed multiple times.
`-p`, `--project`	string	infer	Inspect a named project when doctor cannot infer one from the checkout.

Default exit behavior is `0` for `OK`, `WARN`, and `SKIP`, and `1` for `ERROR`. Attach `sase doctor -v` or `sase doctor -j` when asking for help.

24.6.32 `sase version`

`sase version` reports the local runtime that the current `sase` process is using. It does not query PyPI, GitHub, or latest available releases. The inventory always includes the host `sase` package and the required `sase-core-rs` Rust core distribution, then adds installed SASE plugin packages discovered through SASE entry points, SASE console scripts, or `sase-*` distribution names.

The default human output is a compact runtime panel plus a package table with role, effective version, and code directory. Development checkouts use PEP 440 local versions such as `0.1.2+4.g26c39e004` or `0.1.2+0.g26c39e004.dirty`. Editable installs prefer source metadata over stale installed distribution metadata, while `--verbose` and `--json` expose both values for auditability.

Flag	Values	Default	Description
`-j, --json`	flag	-	Emit a stable JSON object with `schema_version: 1`, runtime, and packages.
`-v, --verbose`	flag	-	Include install type, dist/source versions, git metadata, and plugin signals.

24.6.33 `sase var`

`sase var set` attaches small named string values to the current SASE agent run by merging them into `agent_meta.json["output_variables"]`. The command is agent-scoped and requires `SASE_AGENT=1` and `SASE_ARTIFACTS_DIR`. The variables appear in ACE's Agents-tab `OUTPUT VARIABLES` metadata panel. Later agents that wait on this agent with `%wait` load the stored strings when they start and can render them through the `agents` Jinja dictionary in prompts and xprompt workflows.

Form	Flags / arguments	Description
`sase var set KEY=VALUE [...]`	positional assignments	Store one or more output variables for the current agent.

Keys must be valid Jinja attribute identifiers (`[A-Za-z_][A-Za-z0-9_]*`). Values are strings split on the first `=`, so values may contain additional equals signs. Multiple calls merge into the same variable map; later writes for the same key replace earlier values. The command does not update prompts that have already started rendering, so write variables before the producing agent completes and before dependent agents unblock. Downstream prompts read each producer's variables from the single `agents` dictionary keyed by the producer's stable agent name, e.g. `{{ agents["build"].report_path }}` (or `{{ agents.build.report_path }}` for identifier-safe names). Do not store secrets; output variables are persisted in `agent_meta.json` and shown in ACE.

`STOP` is a reserved output variable. `sase var set` stays generic and stores it like any other key, but repeat orchestration interprets it: setting `STOP` (e.g. `sase var set STOP=1`) inside a `%repeat` / `%r` iteration stops the remaining repeat slots, which finalize as successful skipped slots. Truthiness is conservative — `""`, `0`, `false`, `no`, and `off` (case-insensitive) are not-stop; any other value stops the chain. `STOP` affects only repeat-chain continuation; ordinary `%wait` consumers read it as a normal variable. See Repeat Directive (https://sase.sh/xprompt/#repeat-directive) in the xprompt reference for the full cascade semantics.

24.6.34 `sase telemetry`

With no subcommand, `sase telemetry` defaults to `sase telemetry list`.

Flag	Values	Default	Description
subcommand	`status`, `list`, `snapshot`, `dashboard`, `health`, `export-config`	`list`	Telemetry subcommand

See docs/telemetry.md (https://sase.sh/telemetry/) for the full CLI reference including per-subcommand flags.

24.6.35 `sase logs`

Flag	Values	Default	Description
`daterange`	string	(required)	Date range to collect (e.g., `-7d`, `260318`, `260315..260318`)

Supported date range formats:

- **Absolute**: `YYmmdd` or `YYmmddHHMMSS`
- **Relative**: `-Nd` (days ago), `-Nh` (hours ago), `-Nm` (minutes ago), `0d` (today)
- **Ranges**: `START..END` (e.g., `-7d..0d`); single point means "from that point to now"

24.6.36 `sase editor`

`sase editor` exposes JSON-over-stdin helper operations for editor integrations. It is intentionally a fixed-operation bridge rather than a generic shell or filesystem API.

Form	Input	Description
`sase editor helper-bridge xprompt-catalog`	JSON object on stdin	Return the structured xprompt catalog; accepts the same schema as the mobile `xprompt-catalog` helper operation.
`sase editor helper-bridge snippet-catalog`	JSON object on stdin	Return the composed ACE snippet registry used by `sase lsp` and editor completion clients.

The structured catalog includes insertion metadata (`insertion`, `reference_prefix`, `kind`), typed argument metadata, display/source fields, and `definition_path` when SASE can resolve a real file to jump to.

The snippet catalog uses the same source ordering as ACE: xprompts marked with `snippet` front matter plus user-defined `ace.snippets`, with `ace.snippets` winning on trigger collisions.

24.6.37 `sase file`

With no subcommand, `sase file` defaults to `sase file list`.

Form	Flags	Description

`sase file list`	`-p/--path`, `-t/--token`	Emit JSON filesystem completion candidates rooted at `--path` and filtered by the cursor token.

24.6.38 `sase file-history`

With no subcommand, `sase file-history` defaults to `sase file-history list`.

Form	Flags	Description
`sase file-history list`	none	Emit the recency-ordered file-reference history as a JSON array.
`sase file-history delete`	`-p/--path`	Remove one entry from the file-reference history.

24.6.39 `sase plugin`

With no subcommand, `sase plugin` defaults to `sase plugin list` with default options. Use `sase plugin list --verbose` or `sase plugin doctor --json` when passing diagnostic flags.

Form	Flags	Description
`sase plugin list`	`-j/--json`, `-v/--verbose`	Inventory installed SASE entry points and configured or available chop scripts.
`sase plugin doctor`	`-j/--json`, `-v/--verbose`	Diagnose resource loading, configured chops, and optional integration prerequisites.

24.6.40 `sase lsp`

Starts the xprompt language server over stdio for editor integrations. `SASE_XPROMPT_LSP_CMD` can override the server command during development. Without that override, `sase lsp` uses `sase-xprompt-lsp` from `PATH`, then checks a sibling `../sase-core` checkout for debug/release binaries, then falls back to `cargo run` from that sibling checkout when Cargo is available.

Flag	Values	Default	Description
`-V`, `--version`	flag	-	Print the xprompt LSP version and exit

24.6.41 `sase path`

Flag	Values	Default	Description
`name`	`xprompts-dir`, `xprompts-schema`, `xprompts-collection-schema`, `config-schema`	(required)	Which path to print

24.6.42 `sase notify`

With no subcommand, `sase notify` defaults to the compact `sase notify list` view. Use `sase notify list` for JSON, limit, query, unread, dismissed, or the clearest sender/tag filtering form. Use `sase notify create` to write a notification from stdin JSON.

Form	Flags	Description
`sase notify`	`-s/--sender`, `-t/--tag`	Shortcut for `sase notify list` with default compact output
`sase notify create`	`-s/--sender`, `-t/--tag`	Create a notification from stdin JSON
`sase notify list`	`-j/--json`, `-l/--limit`, `-q/--query`, `-t/--tag`, `-s/--sender`, `-u/--unread`, `-a/--all`	List recent notifications; `-j` emits the stable JSON shape
`sase notify show`	`-i/--id`, `-f/--format` (`markdown` or `json`)	Show one notification by id; defaults to markdown

Create accepts a JSON `tags` field and repeatable `-t/--tag`; CLI tags are appended to JSON tags, then normalized and deduplicated. `sase notify list -q` also matches tags, and `sase notify list --tag <tag>` filters to notifications with that exact normalized tag.

24.6.43 `sase plan`

With no subcommand, `sase plan` defaults to the `sase plan list` dashboard.

Form	Flags	Description
`sase plan approve [selector]`	`-k/--kind`, `-m/--model`, `-p/--prompt`	Approve one pending proposal by notification ID or unique ID prefix.
`sase plan` / `sase plan list`	`-j/--json`	List pending, approved, and inferred rejected plan proposals.
`sase plan propose <plan_file>`	-	Submit a Markdown plan file for approval from the `/sase_plan` skill.

`sase plan list` prints a Rich dashboard by default and emits a stable JSON projection with `summary`, `proposed`, `approved`, and `rejected` keys when `-j/--json` is set. Use the Proposed row's `id_prefix` as the selector for `sase plan approve`; omitting the selector is valid only when exactly one pending proposal exists. Approval kind `approve` runs the coder without asking the runner to commit an SDD plan, `tale` commits the plan as an SDD tale and then runs the coder, `epic` and `legend` commit the matching SDD tier and launch the bead follow-up, and `commit` records the approved plan in SDD without launching a coder. The `-m/--model` flag applies to the follow-up agent; `-p/--prompt` adds extra coder instructions only for the `approve` and `tale` paths.

24.6.44 `sase artifact`

`sase artifact create` is intended for code agents running with `SASE_AGENT=1` and `SASE_ARTIFACTS_DIR` set. It moves a generated file into persistent SASE artifact storage and associates it with the current agent so the Agents tab can open it with `A`, even after the agent has been dismissed and revived.

Form	Flags	Description
`sase artifact create`	`-p/--path`, `-n/--label`, `-k/--kind`	Store one explicit artifact for the current agent

24.6.45 `sase questions`

Flag	Values	Default	Description
`questions_json`	string	(required)	JSON string containing questions to ask

24.6.46 `sase agent`

`sase agent` provides cross-project visibility into running agents. Subcommands:

Subcommand	Flags	Description
`list`	`-a/--all`, `-j/--json`, `-p/--project`	List running agents. `-a` includes DONE/FAILED agents (capped at 50 per project). `-j` emits a JSON array with a stable schema. `-p` limits output to a single project.
`show`	`-n/--name`	Render a full detail panel (prompt, reply, metadata) for a single agent by name.
`kill`	`-n/--name`	SIGTERM a running agent by name.
`tag`	`set` / `unset` / `list`	Manage the user-defined tag on an agent (used by the Agents tab tag side panels). `tag set -n <agent> -t <tag>` replaces any prior tag; `tag unset -n <agent>` clears it; `tag list [-n <agent>]` prints tags as JSON (filtered when given).
`archive`	`rebuild-index` / `verify`	Maintain the dismissed-agent bundle summary index under `~/.sase/dismissed_bundles/`. `verify` exits non-zero if rows are stale or missing.
`artifacts`	`layout status` / `migrate` / `verify` / `rollback`, `-P/--project`, `-p/--projects-root`, `-i/--index-path`, `-j/--json`	Inspect and migrate the physical `ace-run` artifact directory layout. `status` reports flat and sharded directory counts, `migrate` moves flat timestamp directories into day shards, `verify` checks current or manifest-backed state, and `rollback` reverses a manifest-backed migration.
`index`	`status` / `rebuild` / `verify` / `gc`, `-i/--index-path`, `-p/--projects-root`, `-j/--json`	Maintain the persistent agent artifact index. Defaults are `~/.sase/agent_artifact_index.sqlite` and `~/.sase/projects`; `status` performs a lightweight visible-inbox check without scanning source artifacts, `verify` exits non-zero when the index diverges from source artifacts, and `gc` rebuilds the index from source artifacts and replaces the dismissed projection.
`names`	`migrate-auto`, `-f/--force`, `-j/--json`	Maintain the permanent agent-name registry. `migrate-auto` runs the historical generated-name namespace migration; `--force` reruns it after the completion marker exists and `--json` emits a machine-readable summary.

24.6.47 `sase chat`

`sase chat` discovers and inspects saved agent chat transcripts. With no subcommand, it defaults to `sase chat list`. Subcommands:

Subcommand	Flags	Description
`list`	`-j/--json`, `-l/--limit`, `-q/--query`	List recent transcripts. `-j` emits the stable JSON shape consumed by the `/sase_chats` skill.
`show`	`-n/--agent`, `-p/--path`, `-b/--basename`, `-f/--format`	Show one transcript by agent name, path, or basename. `--format` accepts `raw`, `resume`, or `response`.

24.7 Directory Sharding

A fresh install writes agent artifacts (chat logs, notifications, prompt history, workflow state, etc.) directly under `~/.sase/<kind>/`. After a few months of heavy use those directories can accumulate tens of thousands of files, which slows down filesystem walks and makes `ls`-style inspection painful.

New files are automatically written into a `YYYYMM/` shard inside each high-volume directory (keyed by the current month). Readers transparently merge sharded and non-sharded files, so the layout is backwards-compatible — existing unsharded files at the top level are still found and the layout is fully read/write compatible across both forms.

ACE run artifacts also support a day-sharded physical layout under each project's artifact root. Use `sase agent artifacts layout status` to inspect flat versus sharded `ace-run` directories, `migrate` to move legacy flat timestamp directories into shards while writing index aliases, `verify` to check the current or manifest-backed state, and `rollback -m <manifest>` to reverse a migration when needed. Migration skips live artifact directories with `running.json`, refuses existing targets, and can be previewed with `--dry-run`.

25 Query Language Reference

The ChangeSpec query language filters ChangeSpecs using boolean expressions that combine string matching, property filters, and operational shorthands. It is used by the PRs tab in `sase ace [query]` and by other ChangeSpec filters such as `sase axe start --query`.

Normal query surfaces use active-project ChangeSpec discovery. Inactive projects are omitted from CLI search and day-to-day ACE/axe scans. Views that are specifically about agent history or old artifacts opt into all project lifecycle states explicitly.

This page documents ChangeSpec queries. The Agents tab in ACE has a separate agent query language with agent-specific property keys.

25.1 String Matching

Bare words and double-quoted strings perform case-insensitive substring matching against all searchable fields. Use quoted strings when the value contains spaces, punctuation, or other characters that are not valid in a bare word:

```
foobar          bare word, matches "FooBar", "FOOBAR", etc.
"foo bar"       quoted string, allows spaces and special characters
```

Prefix a quoted string with `c` to force case-sensitive matching:

```
c"FooBar"       matches only "FooBar", not "foobar" or "FOOBAR"
```

Inside quoted strings, the following escape sequences are recognized: `\\` (literal backslash), `\"` (literal quote), `\n` (newline), `\r` (carriage return), and `\t` (tab).

25.2 Searchable Fields

String matches search across these ChangeSpec fields as one combined text corpus:

- **name** -- the ChangeSpec name
- **description** -- the ChangeSpec description text
- **status** -- the status text as stored on the ChangeSpec
- **project** -- project directory basename (derived from file path)
- **parent** -- parent ChangeSpec name (if set)
- **cl** -- CL identifier (if set)
- **commits** -- history entry notes and suffixes
- **hooks** -- hook display commands and status line suffixes
- **comments** -- reviewer names, file paths, and suffixes
- **mentors** -- mentor status line suffixes

For normalized status matching, prefer the `status:` property filter instead of a plain string match. `status:` strips workspace suffixes and the legacy `READY TO MAIL` suffix before comparing.

25.3 Property Filters

Property filters match against a specific ChangeSpec field rather than performing a full-text substring search. The supported property filters are exact and case-insensitive:

```
status:WIP            match ChangeSpecs with base status "WIP"
project:myproject     match ChangeSpecs in the "myproject" project
ancestor:parent_cl    match if name or parent chain includes "parent_cl"
name:foo              match ChangeSpecs whose name is exactly "foo"
sibling:bar           match ChangeSpecs in the same sibling family as "bar"
```

Valid property keys: `status`, `project`, `ancestor`, `name`, and `sibling`. Values can be bare words (alphanumeric, `_`, `-`) or quoted strings (e.g. `status:"in progress"`).

25.3.1 Property Shorthand Prefixes

Shorthand	Expands To	Description
`+myproject`	`project:myproject`	Filter by project
`^parent_cl`	`ancestor:parent_cl`	Filter by ancestor (name or parent chain)
`~bar`	`sibling:bar`	Filter by sibling family
`&foo`	`name:foo`	Filter by exact name

25.3.2 Status Shorthands

Shorthand	Expands To
`%d`	`status:DRAFT`
`%m`	`status:MAILED`
`%r`	`status:REVERTED`
`%s`	`status:SUBMITTED`
`%w`	`status:WIP`
`%y`	`status:READY`

Status shorthands are case-insensitive (`%D` and `%d` are equivalent).

25.3.3 Status Matching

The `status:` filter compares the base status only. For example, a ChangeSpec whose stored status is `Ready (sase_102)` matches `status:Ready`. It also treats the legacy `Ready - (!: READY TO MAIL)` form as base status `Ready`.

25.3.4 Ancestor Matching

The `ancestor:` filter (and `^` shorthand) walks the parent chain recursively. A ChangeSpec matches if its own name equals the value, or if any parent, grandparent, etc. in the chain equals the value. Cycle detection prevents infinite loops.

25.3.5 Sibling Matching

The `sibling:` filter (and `~` shorthand) strips any `__<N>` revert suffix from both the search value and the ChangeSpec name, then compares base names. This matches all members of a "family" -- the original plus its `__1`, `__2`, etc. variants.

25.4 Boolean Operators

25.4.1 AND (Implicit and Explicit)

Adjacent terms are combined with implicit AND. The `AND` keyword can also be used explicitly:

```
feature test          implicit AND
feature AND test      explicit AND
```

25.4.2 OR

The `OR` keyword combines alternatives (lower precedence than AND):

```
feature OR bugfix
```

25.4.3 NOT

The `!` operator or `NOT` keyword negates the following expression:

```
!draft               exclude ChangeSpecs containing "draft"
NOT draft            same as above
!"work in progress"  negate a quoted string
```

Multiple `!` operators stack (double negation cancels out).

25.4.4 Precedence

From tightest to loosest binding:

1. `!` / `NOT` (unary negation)
2. `AND` (explicit or implicit juxtaposition)
3. `OR`

Parentheses override precedence:

```
(feature OR bugfix) AND !skip
feature AND (test OR lint)
```

25.5 Special Shorthands

These shorthands filter ChangeSpecs by error, agent, or process state recorded in status or suffix fields.

25.5.1 Error Suffix

Syntax	Meaning
`!!!`	Match ChangeSpecs that have an error suffix in status, commits, hooks, or comments
`!`	Same as `!!!` when standalone: followed by whitespace or at end
`!!`	Match ChangeSpecs with no error suffix (equivalent to `NOT !!!`; standalone only)

25.5.2 Running Agents

Syntax	Meaning
`@@@`	Match ChangeSpecs with a running agent marker in hooks, comments, or mentors
`@`	Same as `@@@` when standalone
`!@`	Match ChangeSpecs with no running agents (equivalent to `NOT @@@`)

25.5.3 Running Processes

Syntax	Meaning
`$$$`	Match ChangeSpecs with a running process marker in hooks or comments
`$`	Same as `$$$` when standalone
`!$`	Match ChangeSpecs with no running processes (equivalent to `NOT $$$`)

25.5.4 Any Special

Syntax	Meaning
`*`	Errors OR agents OR processes (equivalent to `!!! OR @@@ OR $$$`)

Note: `!`, `@`, `$`, `!!`, `!@`, `!$`, and `*` are only treated as special shorthands when standalone (at end of input or followed by whitespace). When `!` is followed by other characters, as in `!"foo"`, it acts as the `NOT` operator.

25.6 Practical Examples

```
%w                          all WIP ChangeSpecs
%w +myproject               WIP ChangeSpecs in "myproject"
feature %d                  drafted ChangeSpecs containing "feature"
!!! %m                     mailed ChangeSpecs with errors
!! !@ !$                   no errors, no agents, no processes
^base_cl %w                 WIP descendants of "base_cl"
~my_cl                      all siblings of "my_cl" (including reverted variants)
"fix bug" OR "refactor"     ChangeSpecs matching either phrase
(bug OR fix) AND !test      bug/fix ChangeSpecs excluding test ones
c"README" +docs             case-sensitive "README" in the docs project
*                           anything with errors, agents, or processes
```

26 ProjectSpec Format

A ProjectSpec is SASE's project-level `.sase` file. It groups the active ChangeSpecs (https://sase.sh/change_spec/) for one project and may also store project metadata used by workspace and agent coordination.

ProjectSpec files live under `~/.sase/projects/<project>/<project>.sase`. Terminal ChangeSpecs are moved to the adjacent archive file, `~/.sase/projects/<project>/<project>-archive.sase`. Legacy `.gp` files from earlier releases remain readable as a fallback; the `sase changespec migrate-extension` command renames them to the canonical `.sase` extension. That migration changes only the ProjectSpec filenames; it does not rewrite ChangeSpec blocks or alter review state.

26.1 Format

A ProjectSpec has two parts:

1. Optional project metadata before the first `NAME:` line.
2. One or more ChangeSpec blocks, separated by two blank lines.

The ChangeSpec parser finds blocks by scanning for `NAME:` lines. Project metadata is read by narrower helpers and must stay before the first ChangeSpec.

```
BARE_REPO_DIR: ~/.sase/repos/my_project.git
WORKSPACE_DIR: ~/projects/git/my_project/
PROJECT_STATE: active
PROJECT_ALIASES: docs
RUNNING:
  #10 | 12345 | run | my_project_add_config_parser_1 | 260509_121314


NAME: my_project_add_config_parser_1
DESCRIPTION:
  Add configuration file parser

  This CL implements configuration loading and validation.
BUG: http://b/12345
STATUS: WIP


NAME: my_project_add_docs_1
DESCRIPTION:
  Document configuration setup

  This CL adds user-facing documentation for the configuration file.
PARENT: my_project_add_config_parser_1
STATUS: WIP
```

26.2 BUG Field

`BUG:` is a ChangeSpec field, not required project metadata. Put it inside each ChangeSpec that should link to a bug or issue. SASE stores the value as text; common values are a plain identifier or a URL:

```
BUG: 12345
BUG: http://b/12345
BUG: https://b/12345
```

PR workflows that receive `SASE_BUG_ID` or `sase commit --bug-id` write the ChangeSpec field as `http://b/<id>`. Child ChangeSpecs may inherit the parent's `BUG:` when SASE creates them through the commit workflow.

26.3 Project Metadata Fields

Project metadata fields are optional and appear before the first `NAME:` line. SASE currently uses these fields:

- **BARE_REPO_DIR**: Path to the local bare git repository for the built-in `#git` workflow.

- **WORKSPACE_DIR**: Path to the primary checkout (workspace `#0`). Managed numbered checkouts are resolved through the per-project workspace store rather than by appending `_<num>` to this path; see `docs/workspace.md` (https://sase.sh/workspace/#workspace-directory-layout) for the directory-layout reference and `docs/configuration.md` (https://sase.sh/configuration/#workspace) for the `workspace.root` knob.
- **PROJECT_STATE**: Project lifecycle state. Valid values are `active`, `inactive`, and `sibling`. Missing `PROJECT_STATE` means `active`, so existing projects do not need a migration. Legacy `archived` and `closed` values are read as `inactive`.
- **PROJECT_ALIASES**: Comma-separated alternate project names accepted in VCS workspace references. Aliases are canonicalized to the real project name before launch state, prompt history, and agent artifacts are written.
- **RUNNING**: Active workspace claims written and released by SASE while agents or workflows are running.

`BARE_REPO_DIR` and `WORKSPACE_DIR` are created by first-use `#git:<project>` initialization or `#git:<bare-repo-path>` registration. They are parsed only before the first ChangeSpec.

`PROJECT_STATE` is managed by `sase project`. If you edit this field by hand, keep it before `RUNNING:` or the first `NAME:` line and use one of the valid lowercase values.

`PROJECT_ALIASES` is managed by `sase project alias` and ACE's Project Management panel. If you edit it by hand, keep it before `RUNNING:` or the first `NAME:` line and use the same comma-separated form SASE writes.

26.3.1 Project Aliases

Project aliases let a known project expose short names without changing the canonical project record. For example, `PROJECT_ALIASES: bob` in `~/.sase/projects/bob-cli/bob-cli.sase` makes launch-bound VCS refs such as `#gh:bob`, `#gh_bob`, and `#gh(bob)` behave like `#gh:bob-cli`, `#gh:bob-cli`, and `#gh(bob-cli)`.

Workspace providers can also create aliases automatically. The GitHub provider uses this for first-use `owner/repo` refs: `#gh:foo-org/foo` can create a canonical SASE project such as `gh_foo-org__foo` with `WORKSPACE_DIR` set to `~/projects/github/foo-org/foo/` and `PROJECT_ALIASES: foo`. If another GitHub repo has the same basename, such as `#gh:bar-org/foo`, the provider keeps a distinct canonical project such as `gh_bar-org__foo` and allocates the first available short alias, starting with `foo-2`, then `foo-3`, and so on.

Existing basename projects are compatibility anchors. If `~/.sase/projects/foo/foo.sase` already points at `~/projects/github/foo-org/foo/`, the GitHub provider reuses `foo` instead of migrating or renaming it. No automatic ProjectSpec rename is required; generated aliases can be inspected or adjusted with `sase project alias`.

Aliases are resolved at the launch/xprompt boundary before workspace resolution, xprompt expansion, prompt history writes, and agent artifact writes. The alias should not persist in `submitted_xprompt.md`, `raw_xprompt.md`, `agent_meta.json`, prompt history, display names, history sort keys, or VCS refs. Normal launch and history surfaces show the canonical project name; project-management surfaces show the configured aliases.

Validation rules:

- Missing `PROJECT_ALIASES` means the project has no aliases.
- Values are comma-separated, trimmed, deduplicated, and stored in sorted order.

- Alias names use the same syntax as SASE project names.
- Automatic alias allocation tries the requested short name first, then appends `-2`, `-3`, and higher suffixes until it finds a value that does not collide.
- An alias cannot equal its canonical project name.
- An alias cannot collide with a real project name or with another project's alias across non-system projects in any lifecycle state.
- Invalid or duplicate manually edited aliases are reported as parse warnings; CLI and TUI mutation helpers reject invalid writes.

CLI commands:

```
sase project alias list [PROJECT] [-j|--json]
sase project alias add PROJECT ALIAS
sase project alias remove PROJECT ALIAS
sase project alias clear PROJECT
```

Alias mutation uses the normal ProjectSpec lock and can target active, inactive, or sibling projects. The system-managed `home` project cannot be mutated.

ACE exposes aliases in the `,p` Project Management panel. Rows show compact alias information, the detail pane shows the full list, the text filter matches aliases, and `A` opens the alias editor for the highlighted project. Alias edits replace the selected project's alias set; marked bulk operations remain lifecycle-only.

26.3.2 Project Lifecycle

Project lifecycle state controls whether a project appears in the default lists used to start new work or browse current work. It is project-level metadata; it does not delete project files and is separate from a ChangeSpec whose `STATUS` is `Archived`.

State	Meaning
`active`	Normal work state. Missing `PROJECT_STATE` also means `active`, so existing projects need no migration.
`inactive`	Dormant, historical, or finished project. Hidden from default launch pickers and discovery lists.
`sibling`	Configured linked-repository bookkeeping (legacy backing state name). Hidden from default launch pickers and discovery lists.

Legacy `PROJECT_STATE: archived` and `PROJECT_STATE: closed` files remain readable and are normalized to `inactive`. New writes use only canonical lifecycle values.

Default project discovery is active-only. That includes ACE project selection, `sase changespec search`, known-project workspace references such as `#gh:sase`, project-local xprompt catalogs, broad mobile helper catalogs, and all-known bead helper reads. These records are intentionally hidden from those surfaces; use `sase workspace open -p <linked_repo> -r "<reason>" <workspace_num>` for configured linked repositories. Agent-history views that need old artifacts pass an explicit all-state scan.

Use `sase project list --state all` to inspect inactive and sibling projects, `sase project show <project>` to see state, workspace, launchability, and warnings, and `sase project activate <project>` before using normal launch surfaces for an inactive or sibling project. The `deactivate`, `activate`, and `set-state` forms update the ProjectSpec under the normal ProjectSpec lock. Deprecated `archive` and `close` aliases still set `inactive` for compatibility. Deactivating refuses projects with live `RUNNING` claims or active artifact markers unless `--force` is passed. The system-managed `home` project cannot be mutated through this command.

ACE exposes the same lifecycle operations through the `,p` project management panel. The panel loads non-system projects across all states, defaults to the active filter, offers text and state filters, supports marks for bulk activate/deactivate operations and bulk full-directory deletion, and uses the same blocked-operation checks before deactivating a project. It can also open the selected ProjectSpec in `$EDITOR`. Its delete action removes the whole SASE project directory under `~/.sase/projects/` after confirmation, including ProjectSpecs, project-local config, and artifacts; it does not remove workspace checkouts. This is broader than `Ctrl+D` in project launch pickers, which only removes an empty project's ProjectSpec files.

Common workflows:

- Deactivate a dormant project: `sase project deactivate old-project`
- List inactive projects: `sase project list --state inactive`
- List sibling project records: `sase project list --state sibling`
- Inspect every lifecycle state as JSON: `sase project list --state all --json`
- Reactivate from the CLI: `sase project activate old-project`
- Add a short project alias: `sase project alias add bob-cli bob`
- Inspect project aliases as JSON: `sase project alias list bob-cli --json`
- Reactivate from ACE: press `,p`, highlight the project, then press `a`
- Edit aliases from ACE: press `,p`, highlight the project, then press `A`
- Bulk-deactivate from ACE: press `,p`, mark projects with `m`, then press `d`

Maintenance and agent-history scans intentionally keep reading all project directories. This keeps live `RUNNING` claims, stale-claim cleanup, dismissed-agent recovery, agent-name collision checks, and historical Agents-tab rows visible even after a project is inactive.

The `RUNNING` section is managed by SASE. Each entry has this shape:

```
RUNNING:
  #<WORKSPACE_NUM> | <PID> | <WORKFLOW> | <CHANGESPEC_NAME> | <TIMESTAMP> | PINNED
```

The timestamp and `PINNED` marker are optional. Do not edit `RUNNING` by hand unless you are repairing a stale workspace claim and have verified the process is gone.

26.4 ChangeSpec Fields

Each ChangeSpec in a ProjectSpec follows the ChangeSpec format (https://sase.sh/change_spec/). For hand-written entries, the normal minimum fields are:

1. **NAME**: Unique ChangeSpec identifier. SASE-generated names normally start with `<project>_` and end with a numeric uniqueness suffix such as `_1`.
2. **DESCRIPTION**: A title, a blank line, and a body, all indented by two spaces.
3. **STATUS**: One of the lifecycle statuses documented in `change_spec.md` (https://sase.sh/change_spec/#status). New manual work typically starts as `WIP`.

Common optional fields include:

- **PARENT**: The `NAME` of a parent ChangeSpec that must land first. Omit it when there is no dependency.
- **CL / PR**: URL for the created review, omitted until the CL or PR exists. `CL:` and `PR:` are parsed the same way.
- **BUG**: Bug or issue reference for this ChangeSpec.
- **COMMITS**, **DELTAS**, **HOOKS**, **COMMENTS**, **MENTORS**, and **TIMESTAMPS**: See `change_spec.md` (https://sase.sh/change_spec/) for details.

26.5 Example

```
WORKSPACE_DIR: ~/projects/git/my_project/
PROJECT_STATE: active
PROJECT_ALIASES: docs


NAME: my_project_add_config_parser_1
DESCRIPTION:
  Add configuration file parser for user settings

  This CL implements a YAML-based configuration parser that reads
  user settings from ~/.myapp/config.yaml. The parser includes load
  and validation behavior, plus tests for valid YAML, invalid config,
  and missing file handling.
BUG: http://b/12345
STATUS: WIP


NAME: my_project_integrate_parser_1
DESCRIPTION:
  Integrate config parser into application startup

  This CL loads the parser during application initialization and
  surfaces validation errors clearly. Tests cover valid and invalid
  startup configuration.
PARENT: my_project_add_config_parser_1
STATUS: WIP


NAME: my_project_add_docs_1
DESCRIPTION:
  Document configuration setup

  This CL explains where the configuration file lives, shows common
  examples, and documents the supported keys.
PARENT: my_project_integrate_parser_1
STATUS: WIP
```

26.6 Important Notes

- **Project file path**: Use `~/.sase/projects/<project>/<project>.sase` for active ChangeSpecs and `~/.sase/projects/<project>/<project>-archive.sase` for terminal history.

- **Project metadata**: Keep `BARE_REPO_DIR`, `WORKSPACE_DIR`, `PROJECT_STATE`, and `RUNNING` before the first `NAME:` line. `PROJECT_ALIASES` is also project metadata and belongs in the same header area.
- **Blank lines between ChangeSpecs**: Separate ChangeSpecs with exactly two blank lines.
- **NAME field**: Prefer SASE-generated names, which use the project prefix and a numeric suffix.
- **PARENT field**: Set it only to another ChangeSpec `NAME`; omit it when there is no dependency.
- **CL / PR field**: Omit until the CL or PR exists, then set it to the review URL.
- **No file modification lists**: Keep file lists out of `DESCRIPTION`; SASE records file-level deltas separately.

27 Agent Attachments and Image Previews

27.1 Overview

SASE treats files produced by agents as first-class completion artifacts. When a successful agent adds or modifies a supported image file, the completion path records the image in `done.json` and appends it to the notification file list after the standard chat and diff artifacts, and after any generated Markdown PDFs. When a successful agent adds or modifies up to 10 Markdown files, core SASE renders PDF artifacts and attaches those PDFs to the same completion notification. Explicit artifacts saved with `sase artifact create` are appended after generated images when the agent completion notification is sent. Notification plugins can then deliver those files from `Notification.files` without re-scanning the workspace.

ACE is SASE's terminal UI. It has two image surfaces: lightweight in-panel previews for notification and file-panel attachments, and the separate `A` artifact viewer for opening completed agent artifacts.

ACE can also surface image files referenced in saved prompt artifacts (`raw_xprompt.md` and `*_prompt.md`) even when the image itself was not part of the agent's git diff. For current successful runs, those prompt-referenced images are copied into persistent SASE artifact storage with the other default image artifacts so the Agents-tab artifact picker can still open them after a workspace is cleaned up. Legacy runs without persisted default artifacts fall back to prompt-file discovery at view time. Prompt-referenced images are not notification delivery attachments unless they also appear in `done.json.image_paths` or were saved explicitly with `sase artifact create`.

Supported image extensions are:

- `.png`
- `.jpg`
- `.jpeg`
- `.webp`
- `.gif`

27.2 Image Attachment Contract

Generated-image discovery runs when an agent finalizes successfully. This contract covers images added to `done.json.image_paths` and completion notifications. The collector checks candidate paths in stable order:

1. tracked files changed relative to `HEAD`
2. untracked files in the agent workspace
3. files named by the saved proposal or commit diff
4. files touched by the latest commit when the agent committed or opened a PR

Only existing files with supported image extensions are kept. Paths are resolved to absolute paths so outbound notification processes can attach them even when they run outside the agent workspace. Duplicates are removed while preserving order, and image paths are appended after any already-attached chat, diff, or generated PDF files.

The same list is persisted as `image_paths` in the agent's `done.json`. Agent metadata consumers should read that field instead of trying to infer generated images from arbitrary notification files.

Source: `src/sase/axe/image_attachments.py`

27.2.1 Prompt-Referenced Images

Default artifact persistence also scans the saved prompt files in the agent artifacts directory:

- `raw_xprompt.md`
- every sibling `*_prompt.md` file

Any path-like token ending in a common image suffix is resolved as an absolute, home-relative, or workspace-relative path. Existing files are added as ACE image artifacts after `done.json.image_paths`, duplicates are removed, and the file does not need to appear in the agent's git diff. This is useful when a prompt asks an agent to inspect or transform an existing screenshot, mockup, or reference image and the resulting run should keep that image one keypress away in ACE.

Prompt-referenced images are ACE artifact-list entries, not notification delivery attachments. Current runs persist them to the global artifact index during finalization; legacy runs can still synthesize them from prompt artifacts when ACE loads the row. Downstream notification plugins should continue to use `done.json.image_paths` for the generated-image notification contract.

Source: `src/sase/core/agent_artifact_defaults.py`

27.3 Markdown PDF Attachment Contract

Markdown discovery runs on successful agent finalization with the same candidate ordering as image discovery. Supported source extensions are `.md` and `.markdown`. Sources are resolved to existing workspace files, generated run artifacts are excluded, and duplicates are removed before rendering. If more than 10 Markdown sources remain after filtering, SASE skips PDF rendering for that run and adds a completion-notification note instead of rendering a large attachment set.

Core SASE renders discovered Markdown sources into the current agent artifacts directory:

```
<artifacts_dir>/markdown_pdfs/<sanitized-relative-source-path>.pdf
<artifacts_dir>/markdown_pdfs/index.json
```

Rendering is best-effort. Missing Pandoc/PDF-engine tools or conversion errors do not fail the agent run; failed sources are omitted. Successful PDF paths are persisted as `markdown_pdf_paths` in `done.json`, and `index.json` records `source_path` to `pdf_path` mappings for diagnostics. When the 10-source limit is exceeded, `done.json.markdown_pdf_paths` is empty and the source count is carried through completion handling for the user-facing skip note.

While PDFs are being prepared, the runner writes `workflow_state.json.pdf_status` plus a compact `activity` label. ACE loads those fields during refresh and shows messages such as `Preparing PDFs from Markdown...`, `PDF 2/4 <path>`, or `PDFs done 3/4 (1 skipped)` on the Agents row and in the agent header. This status is transient finalization state; the durable output remains `done.json.markdown_pdf_paths` and `markdown_pdfs/index.json`.

Markdown PDFs use a built-in small-screen layout by default: a narrow portrait page, small margins, larger readable body text, and wrapping-friendly CSS for code blocks, tables, links, and other long content. The preferred `wkhtmltopdf` path receives both the default stylesheet and explicit page/margin options; LaTeX fallbacks receive the same page size, margin, font size, and line-height defaults through Pandoc variables.

Completion notifications attach generated Markdown PDFs after the saved chat and diff files, before image attachments. The Agents tab file panel also loads `markdown_pdf_paths` alongside plan and image files for completed agents.

Sources:

- `src/sase/attachments/markdown_pdf.py`
- `src/sase/axe/run_agent_exec.py`

27.4 Explicit Artifact Contract

Agents can save a generated file explicitly with:

```
sase artifact create -p <path> [-n <label>] [-k <kind>]
```

The CLI command is intended for agent processes: it requires `SASE_AGENT=1` and `SASE_ARTIFACTS_DIR` so SASE knows which run owns the artifact. It moves the source file into persistent SASE artifact storage, records an association with the current agent, and lets ACE show the artifact even after the agent is dismissed and later revived. During completion notification delivery, SASE appends existing explicit artifact files after chat, diff, generated Markdown PDFs, and generated image attachments. Duplicate paths and missing files are ignored, and explicit-artifact index failures do not fail the completion path.

Source: `src/sase/core/agent_artifact_facade.py`

27.5 Notification Delivery

Core SASE stores generated PDFs, generated images, and explicit artifact attachments in the existing `Notification.files` list. There is no separate notification schema field for typed attachments yet. This keeps the contract compatible with existing notification storage and lets downstream plugins decide how to render each file:

- Telegram integrations can send images as photos and keep markdown/diff files as documents.
- Google Chat integrations can upload image files directly into the completion thread.
- The ACE notification modal can still open attached files in `$EDITOR` with `e` and cycle them with `Ctrl+N` / `Ctrl+P`.

See `notifications.md` (https://sase.sh/notifications/) for the notification model and modal keybindings.

27.6 ACE Artifact Viewer

The Agents tab exposes completed agent artifacts through the `A` key. When artifacts exist, ACE opens the artifact panel for selection. Chat transcripts, plan files, generated Markdown PDFs, generated images, prompt-referenced images, and explicit artifacts created with `sase artifact create -p <path> [-n <label>] [-k <kind>]` all use the same list. The panel is shown even for a single artifact so users can confirm the artifact label, kind, and path before opening it.

The selected agent's prompt/detail header also includes an `ARTIFACTS` section for non-chat artifacts. Paths are shown relative to the agent workspace when possible, home-relative when appropriate, and with hint numbers when hint mode is active.

The panel supports one-key selectors, `j` / `k` navigation, `m` to mark rows, `Enter` to open the marked set or highlighted row, `y` to copy highlighted Markdown contents, `Y` to copy the highlighted artifact path, and `A` to open every artifact in list order. Copied paths are workspace-relative when possible and fall back to home-relative paths. When multiple artifacts are opened together, the terminal viewer adds `n` / `p` navigation between artifacts in addition to page navigation.

When ACE is running inside tmux, the artifact viewer launches in a right-side tmux pane and the Agents list collapses while the pane is live. Press `l` from the Agents tab to focus the tracked artifact pane, or press `A` again to close it. Row-changing navigation is guarded while the pane is open so the TUI does not drift to a different agent than the viewer. Outside tmux, ACE suspends and opens the viewer in the current terminal pane. The viewer chooses its mode from the artifact kind and file extension: supported images are displayed directly, PDFs are converted to PNG pages, and Markdown is rendered to PDF before paging. The page loop uses `j` / `k` to move between pages, wrapping at the first and last page, `n` / `p` to move between artifacts in a sequence, `r` to refresh, and `q` to close the viewer.

Only one plan artifact is listed for each agent. If run metadata contains both an archived plan path and an SDD tale path, committed plans prefer the SDD path; uncommitted plans prefer the archived path unless only the SDD path is available.

Viewer dependencies are intentionally outside the agent completion path. `kitten` is required for terminal display, `pdftoppm` is required for PDF/Markdown paging, and Markdown rendering also needs `pandoc` plus one supported PDF engine. If a dependency is missing, ACE shows a warning instead of failing the TUI or changing the stored artifact list.

Source: `src/sase/ace/tui/graphics/viewer.py`

27.7 ACE Image Preview Foundation

The notification modal and Agents tab file panel route supported image extensions through the preview layer before attempting text decoding.

The internal preview layer renders PNG, JPEG, WebP, and GIF attachments as a portable Rich cell preview. It uses Pillow to decode the first image frame, apply EXIF orientation, fit it inside the visible panel bounds, composite transparency onto a dark background, and apply a mild preview-only sharpen pass after resizing. Each terminal cell samples a 2 x 2 pixel block and chooses the closest Unicode block mask with foreground and background colors, which preserves more edges, diagonals, UI details, and text-like shapes than a fixed half-block sampler.

No Kitty, iTerm2, Sixel, or other terminal image protocol support is required. The renderer only emits colored Unicode text through Rich/Textual, so it works the same way in ordinary terminals, multiplexed sessions, SSH sessions, and environments with no image protocol support. Preview quality depends on the visible pane size and terminal color depth: larger panes provide more sampled cells, and truecolor terminals preserve colors better than 256-color terminals.

ACE checks only terminal color depth from the environment. When `COLORTERM=truecolor`, `COLORTERM=24bit`, or a truecolor marker in `TERM` is present, previews use 24-bit color; otherwise they use 256-color approximations. Missing files, unsupported extensions, decode errors, missing Pillow, and images above the renderer guardrails

show a concise text fallback with the file path, byte size when available, and the relevant editor or artifact action. Use `e` in notifications, `E` on the Agents tab, or the `A` artifact viewer whenever full-fidelity viewing is needed.

Source: `src/sase/ace/tui/graphics/`

Acknowledgements

Boris' Method

Before sase existed, Boris Cherny (https://borischerny.com/) -- the inventor of Claude Code (https://docs.anthropic.com/en/docs/claude-code) -- pioneered a practical workflow for parallel agentic development: five git checkouts, each in its own tmux tab, each running Claude Code in plan mode. This was one of the first demonstrations that a single developer could effectively supervise multiple agents working on different tasks simultaneously, and it proved that the bottleneck in agentic software engineering isn't the agent -- it's the coordination layer around it.

sase builds directly on this insight. Where Boris' method relies on the developer to manually manage the parallelism -- switching between tmux tabs, keeping track of which checkout is doing what, copy-pasting prompts, and mentally tracking the state of five concurrent workstreams -- sase replaces that manual overhead with structured infrastructure:

- **Workspaces instead of manual checkouts** -- sase's workspace provider system creates and manages isolated working copies programmatically, eliminating the need to manually set up and maintain parallel git checkouts.
- **ChangeSpecs instead of mental bookkeeping** -- Each unit of work gets a tracked lifecycle with status, metadata, and history, replacing the cognitive load of remembering what's happening in each tmux tab.
- **XPrompts instead of ad-hoc prompts** -- Reusable, composable prompt templates with YAML front matter replace the prompt fragments scattered across shell history and scratch files.
- **True SDD instead of plan mode** -- Plan mode produces ephemeral plans that vanish when the session ends. sase persists prompt snapshots plus ordinary tales, executable epics, and higher-level legends under `sdd/`. The `sdd/research/` corpus keeps exploratory notes with that same repository-local history. For larger efforts, epic files carry a bead ID in their frontmatter that links them to an epic-tier bead, and each phase of the epic gets its own bead whose ID appears in the corresponding commit messages -- creating a traceable chain from epic to phase to commit. For smaller tales, commit messages include a `PLAN=<path>` tag pointing back to the plan file. The result is spec-driven development where the full history of intent, decomposition, and execution is preserved and queryable, not trapped in a single agent session's context window.
- **ACE instead of tmux** -- A single TUI provides unified navigation, filtering, and management across all active workstreams, replacing the manual tab-switching workflow.
- **AXE instead of manual supervision** -- A background daemon handles scheduling, monitoring, and lifecycle management of agent runs, so the developer doesn't need to babysit each session.

The core idea -- that one developer can multiply their output by running several agents in parallel -- was right. sase just replaces the duct tape with a proper framework.

Beads

Steve Yegge's (https://steve-yegge.medium.com/) beads (https://github.com/steveyegge/beads) (`bd`) introduced a critical insight: AI coding agents suffer from a "50 First Dates" problem -- every new session starts from scratch with no memory of prior work. His solution was a dependency-aware graph issue tracker designed specifically for agents, backed by Dolt (https://github.com/dolthub/dolt) (a version-controlled SQL database), with hash-based IDs for collision-free multi-agent writes, hierarchical epics-to-subtasks via dotted IDs, atomic `--claim` operations for agent coordination, and semantic memory compaction to keep the working set small. The key idea was that agents don't need TODO files or markdown plans -- they need a structured, persistent memory layer they can query and update across sessions, so that multi-session and multi-agent workflows stay coherent.

The `sase bead` command is a from-scratch reimplementation that carries this idea into sase's architecture while drastically simplifying the surface area:

- **SQLite + JSONL instead of Dolt** -- sase stores issues in SQLite for fast local queries and exports to a sorted JSONL file for git portability. Fresh clones rebuild the database automatically from JSONL, giving version-controlled persistence without an external database engine.

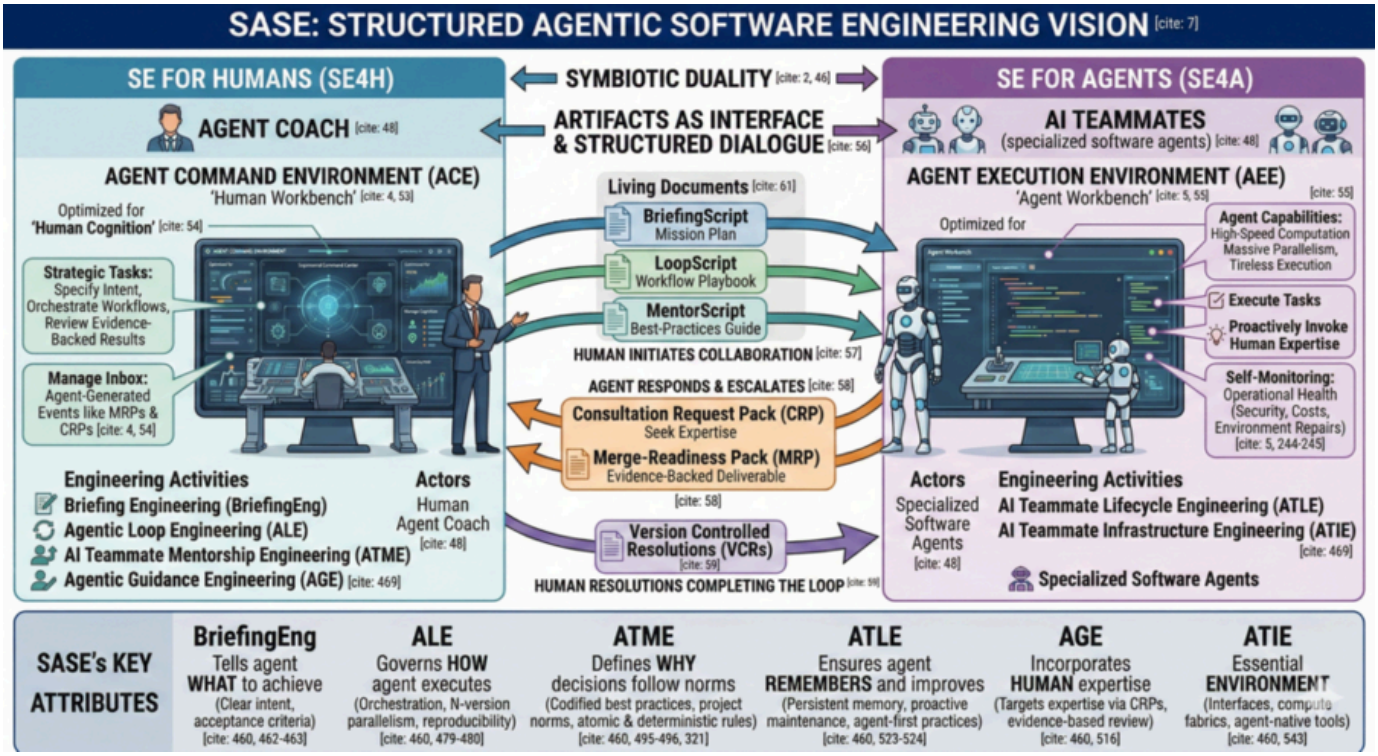
- **Plan tiers instead of arbitrary nesting** -- sase uses plan-like beads with explicit `plan`, `epic`, and `legend` tiers plus executable phase children, rather than deeply nested dotted-ID trees. Linked epics use `--type plan(<plan_file>,<legend_bead_id>) --tier epic`; legend beads store `--epic-count` and `sase bead work` launches one epic-planning agent per proposed epic.
- **Multi-workspace aggregation** -- Because sase already manages multiple parallel workspaces, `sase bead` can read issues across all workspace clones through a merged in-memory view, giving every agent visibility into the full project state without Dolt's sync machinery.
- **No external binary** -- beads ships as a ~37MB Go binary with its own daemon process; `sase bead` is installed as part of sase and uses the required `sase_core_rs` extension for local storage/query/mutation speed, with no separate daemon to run.

The philosophical debt is real: beads proved that giving agents structured issue tracking -- not just chat history -- fundamentally changes what's possible in multi-session agentic workflows. `sase bead` takes the ~5% of that system that matters for sase's orchestration model and integrates it natively.

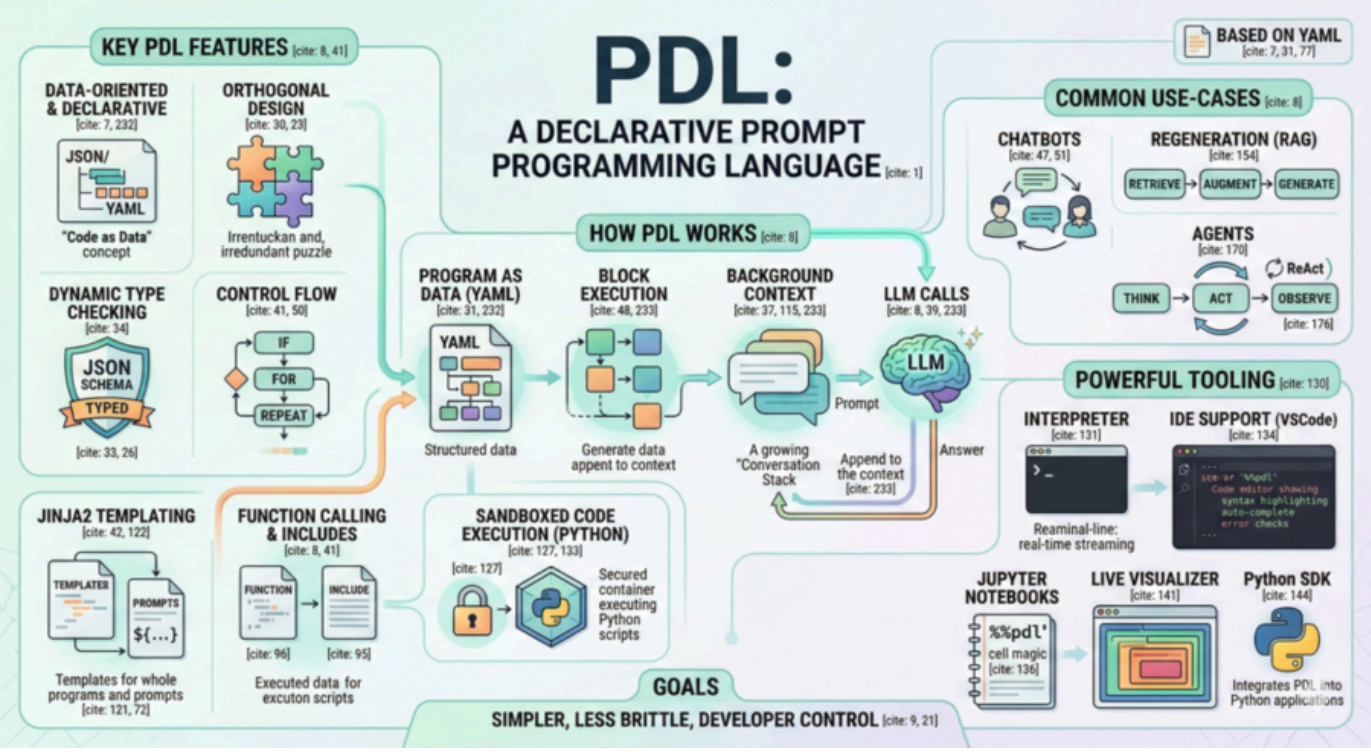
Research Papers

This project was heavily influenced by two research papers:

- **Agentic Software Engineering: Foundational Pillars and a Research Roadmap** (https://arxiv.org/abs/2509.06216) (Hassan et al., 2025) -- This paper's vision of Structured Agentic Software Engineering (SASE) inspired the project's name and overall direction. Its concepts of the Agent Command Environment (ACE) and Agent Execution Environment (AEE) directly informed the naming and design of the `ace` TUI and `axe` daemon, respectively.



- **PDL: A Declarative Prompt Programming Language** (https://arxiv.org/abs/2410.19135) (Vaziri et al., 2024) -- PDL's approach to declarative, YAML-based prompt programming influenced the design of xprompt workflows, sase's YAML-defined multi-step pipelines for orchestrating LLM calls and tool execution.



Images generated with Nano Banana (https://notebooklm.google.com/) (Google's NotebookLM) via Gemini.

2 SASE Blog

The SASE Blog is the publishing surface for essays about Structured Agentic Software Engineering (SASE): durable work units, orchestration, provider-independent workflows, review state, and the coordination layer around coding agents.

2.1 Start Here

Start with Hello, SASE: Your First 15 Minutes (https://sase.sh/blog/posts/hello-sase-your-first-15-minutes/) if you want the practical path: install, provider readiness, a safe first run, and the first visible agent record.

The SASE Blog Series (https://sase.sh/series/agentic-software-engineering/) also begins with [00] The Missing Operating Layer for Coding Agents (https://sase.sh/blog/posts/why-coding-agents-need-orchestration/), the launch essay on why coding-agent work needs durable prompts, plans, state, review, dependencies, retries, and handoff.

More posts in the series are in the repository as drafts and will be published over time. The generated archive below lists only the entries included in the public site.

2.2 From Reading To Practice

- Start with ACE (https://sase.sh/ace/), the Agentic ChangeSpec Explorer, to learn the terminal interface for daily agent work.
- Read the SDD flow (https://sase.sh/sdd/) to see how Spec-Driven Development turns plans, epics, and phase beads into executable work.
- Open the repository (https://github.com/sase-org/sase) for source, issues, and implementation details.

May 10, 2026 · 6 min read

2.3 [01] Hello, SASE — Your First 15 Minutes Orchestrating Coding Agents

(https://sase.sh/blog/posts/hello-sase-your-first-15-minutes/)

2.4 [01] Hello, SASE — Your First 15 Minutes Orchestrating Coding Agents

(https://sase.sh/blog/posts/hello-sase-your-first-15-minutes/#01-hello-sase-your-first-15-minutes-orchestrating-coding-agents_1)

SASE (pronounced "sassy" — yes, really) is a coordination layer that sits above coding-agent CLIs like Claude Code, Codex, or Antigravity CLI (`agy`). This post is the practical on-ramp: by the end you'll have installed `sase`, checked that a provider CLI is ready, launched a safe read-only agent run, found the resulting agent record, and picked up the vocabulary you'll keep bumping into in the rest of the docs. Plan on roughly fifteen minutes at a terminal, plus however long your favorite model takes to think.

Continue reading (https://sase.sh/blog/posts/hello-sase-your-first-15-minutes/)

May 8, 2026 · 18 min read

2.5 [00] The Missing Operating Layer for Coding Agents (https://sase.sh/blog/posts/why-coding-agents-need-orchestration/)

2.6 [00] The Missing Operating Layer for Coding Agents (https://sase.sh/blog/posts/why-coding-agents-need-orchestration/#00-the-missing-operating-layer-for-coding-agents_1)

SASE is not a better model. SASE is the layer I wanted after realizing that the hard part of running coding agents is not always "can the model write the patch?" Sometimes the hard part is "where did the patch go?", "what was it trying to do?", "who is waiting on it?", "why did it start six follow-up agents while I was brushing my teeth?", and "can I please see the diff before the robot commits crimes against `just check`?"

That layer needs reusable prompts, durable plans, dependency-aware work items, review records, background automation, notifications, and a control surface that lets humans steer without becoming a full-time air-traffic controller.

Borrowing the name from the research paper discussed later, SASE calls that layer **Structured Agentic Software Engineering**. This post is the map of the fundamentals: XPrompts, SDD, Beads, ACE, AXE, plugins, and why SASE wraps coding-agent CLIs instead of raw model APIs.

Continue reading (https://sase.sh/blog/posts/why-coding-agents-need-orchestration/)

3 SASE Blog Series

SASE, short for **Structured Agentic Software Engineering**, uses **agentic software engineering** to mean software work where AI agents operate inside durable engineering systems: plans, work queues, review records, tests, commits, dependencies, and handoffs. The SASE Blog Series explains why that coordination layer matters and how SASE implements it.

The canonical essays live on the SASE Blog (https://sase.sh/blog/). This page is the series hub: it lists the published posts and points readers at the current product guides for further reading.

3.1 The Series

The SASE Blog Series begins with [00], the argument for a durable operating layer around coding-agent work, then moves to [01], the 15-minute hands-on path for installing SASE and launching a safe first run. Later posts will cover reusable prompts, background automation, planning, review state, mobile control, editor integration, and the roadmap.

Post	Status
[00] The Missing Operating Layer for Coding Agents (https://sase.sh/blog/posts/why-coding-agents-need-orchestration/)	Published 2026-05-08
[01] Hello, SASE: Your First 15 Minutes (https://sase.sh/blog/posts/hello-sase-your-first-15-minutes/)	Published 2026-05-10

The remaining series entries are forthcoming. Until each one is published, the public site keeps the draft pages out of the navigation, generated archive, RSS feed, search index, and sitemap.

3.2 Reader Paths

Alongside the series, the current product guides make each concept concrete:

- 15-minute quickstart (https://sase.sh/blog/posts/hello-sase-your-first-15-minutes/) for the practical install, readiness, and first-run path.
- ACE TUI (https://sase.sh/ace/) for the interactive control surface.
- Spec-Driven Development (https://sase.sh/sdd/) for plans, epics, legends, and executable phase work.
- ChangeSpecs (https://sase.sh/change_spec/) for reviewable CL/PR-sized work records.
- Beads (https://sase.sh/beads/) for issue-like work items, dependency ordering, and multi-agent epic execution.
- XPrompts (https://sase.sh/xprompt/) for reusable prompt templates and workflow packaging.
- GitHub (https://github.com/sase-org/sase) for source, issues, and implementation details.

3.3 Publishing Notes

The blog is the canonical publishing surface for SASE essays. Unpublished drafts keep their stable slugs, frontmatter dates, and categories in the repository, but only published entries are linked from the public site, RSS feed, search index, and generated archive pages.

[00] The Missing Operating Layer for Coding Agents

SASE is not a better model. SASE is the layer I wanted after realizing that the hard part of running coding agents is not always "can the model write the patch?" Sometimes the hard part is "where did the patch go?", "what was it trying to do?", "who is waiting on it?", "why did it start six follow-up agents while I was brushing my teeth?", and "can I please see the diff before the robot commits crimes against `just check`?"

That layer needs reusable prompts, durable plans, dependency-aware work items, review records, background automation, notifications, and a control surface that lets humans steer without becoming a full-time air-traffic controller.

Borrowing the name from the research paper discussed later, SASE calls that layer **Structured Agentic Software Engineering**. This post is the map of the fundamentals: XPrompts, SDD, Beads, ACE, AXE, plugins, and why SASE wraps coding-agent CLIs instead of raw model APIs.

If you want to install first and read philosophy later, jump to [01] Hello, SASE: Your First 15 Minutes (https://sase.sh/blog/posts/hello-sase-your-first-15-minutes/). That post is the practical quickstart. This one explains what the pieces are and why they exist.

One notation note before we start:

> **Friction note:** Blocks like this call out SASE pain points, rough edges, or future improvements. SASE is useful today, but it is not a marble statue. It is a useful toolbox with several labels still written in Sharpie.

What SASE Is

SASE is a local orchestration layer above coding-agent CLIs such as Codex, Claude Code, Antigravity CLI (`agy`), Qwen Code, and OpenCode. It gives those agents a common workflow: launch in isolated workspaces, expand reusable prompts, save prompt and response artifacts, track CL/PR-sized work as ChangeSpecs, coordinate dependency graphs with Beads, and supervise background work through AXE.

The repo split is intentionally boring:

Repo	What it does
`sase` (https://github.com/sase-org/sase)	The Python host package: CLI, ACE TUI, AXE daemon, XPrompt expansion, SDD, Beads integration, config, and built-in providers.
`sase-core` (https://github.com/sase-org/sase-core)	Shared Rust core for deterministic data operations and cross-frontend APIs. It also houses the mobile gateway and XPrompt LSP crates.
`sase-github` (https://github.com/sase-org/sase-github)	GitHub VCS/workspace provider plugin. It uses `gh` for PR operations and ships GitHub-focused xprompts such as `#gh`, `#new_pr_desc`, and `#prdd`.
`sase-telegram` (https://github.com/sase-org/sase-telegram)	Telegram integration package. It runs as inbound/outbound AXE chops so you can receive notifications, answer approvals, and launch or steer agents from chat.
`sase-nvim` (https://github.com/sase-org/sase-nvim)	Neovim integration for SASE syntax, xprompt completion, hover, diagnostics, and the XPrompt LSP.

The short version: `sase` owns the cockpit, `sase-core` owns shared engine-room logic, and the plugins add providers or frontends without forcing the core workflow to become GitHub-only, Telegram-only, or Neovim-only.

Install The Smallest Useful Thing

SASE needs Python 3.12+, `uv` (https://docs.astral.sh/uv/), and at least one authenticated coding-agent CLI. Install the core package:

```
uv tool install sase --python 3.12
sase version
sase doctor
```

Add GitHub support when you want PR/workspace integration:

```
uv tool install sase --python 3.12 --with sase-github
gh auth login
sase plugin doctor
```

Add Telegram support when you want chat-driven notifications and remote control:

```
uv tool install sase --python 3.12 --with sase-telegram
sase plugin doctor
```

Install both plugins together if that is your normal setup:

```
uv tool install sase --python 3.12 --with sase-github --with sase-telegram
```

If you are replacing an existing `uv tool` install, add `--force` to the same command. The quickstart has the fuller walkthrough: Hello, SASE: Your First 15 Minutes (https://sase.sh/blog/posts/hello-sase-your-first-15-minutes/).

> **Friction note:** Plugin installation is improving, but it is still a little too easy to install `sase` correctly and forget that `sase-github` also needs an authenticated `gh` CLI, while `sase-telegram` needs Telegram bot secrets and AXE chops configured. `sase doctor` and `sase plugin doctor` are the first places to look when something feels haunted. Technically it is not haunted. Usually.

SASE Wraps Agents, Not Models

SASE deliberately wraps CLI agents rather than raw model APIs. A SASE provider plugin constructs commands for existing agent runtimes: Codex CLI, Claude Code, Antigravity CLI (`agy`), Qwen Code, OpenCode, or another provider that implements the same boundary. The LLM provider docs (https://sase.sh/llms/) describe that layer in detail.

This buys a lot:

- You inherit each CLI's auth, sandboxing, approval model, local tool behavior, and provider-specific improvements.
- You can swap providers per prompt with `%model` instead of rewriting the orchestration layer.
- SASE can focus on work state: prompts, workspaces, plans, Beads, ChangeSpecs, and UI.
- Users can keep using the agent CLI they already trust, which is a boring advantage and therefore an excellent one.

The trade-off is real: SASE has less direct control over token accounting, tool protocols, streaming details, and model semantics than it would have with raw APIs. It also inherits provider pricing and policy shifts. For example, Anthropic says (https://support.claude.com/en/articles/15036540-use-the-claude-agent-sdk-with-your-claude-plan) that starting **June 15, 2026**, Claude Agent SDK and `claude -p` usage moves to a separate monthly Agent SDK credit bucket. Past that credit, usage can flow to standard API-rate usage credits if enabled; otherwise those requests stop until the credit refreshes.

That is not a reason to avoid provider CLIs. It is a reason to make the orchestration layer provider-aware, explicit, and replaceable.

The Codex App Is The Real Competitor

I consider the Codex app (https://developers.openai.com/codex/app) SASE's closest competitor. Not because it has the same architecture, but because it targets the same daily shape of work: many agent threads, local worktrees, review surfaces, automations, Git actions, IDE/app integration, and a human trying to keep the whole circus pointed at useful software.

OpenAI's docs describe Codex app features such as automations and background worktrees (https://developers.openai.com/codex/app/features) and pricing tiers that include Codex on the web, CLI, IDE extension, and app surfaces (pricing (https://developers.openai.com/codex/pricing)). That is exactly the kind of product surface SASE has to take seriously.

The difference is emphasis. Codex app is a polished product around OpenAI's agent stack. SASE is an open, local, provider-pluggable operating layer for people who want durable work records, Git-portable state, custom prompt systems, AXE automation, and cross-provider routing. That makes SASE less shiny in places and more hackable in others. I have made peace with this, mostly.

The Fundamental Loop

Here is the SASE loop:

1. You type a prompt, usually with one or more XPrompt references.
2. SASE expands the prompt, strips directives, resolves workspace references, and launches one or more agents.
3. Each agent runs in a managed workspace and writes prompt, transcript, status, and artifacts.
4. If the work needs planning, SASE records it in `sdd/` as a prompt snapshot, tale, epic, or bead graph.
5. ACE shows the live state. AXE watches the background state. Plugins translate VCS and notification operations.

The docs that matter most at first are XPrompts (https://sase.sh/xprompt/), SDD (https://sase.sh/sdd/), Beads (https://sase.sh/beads/), ACE (https://sase.sh/ace/), AXE (https://sase.sh/axe/), VCS providers (https://sase.sh/vcs/), and plugins (https://sase.sh/plugins/).

XPrompts Are The Smallest Load-Bearing Idea

An XPrompt (https://sase.sh/xprompt/) is a reusable prompt reference. You write `#foo`, SASE expands `foo`, and the agent sees the rendered text. It sounds small. It is not small. XPrompts are how SASE keeps prompts composable enough to reuse and structured enough to orchestrate.

The hierarchy runs from tiny to large:

Level	Where it lives	Use it for
Config xprompt	`xprompts:` in `sase.yml`	Aliases and tiny reusable phrases. Example: `x: "xprompt"` or `xw: "xprompt workflow"`.
Structured config xprompt	`sase.yml` with `content`, `description`, `input`	Small templates with typed inputs.

Markdown xprompt	`.xprompts/`, `xprompts/`, `~/.xprompts/`, project config dirs, plugins, or built-ins	Normal reusable prompt bodies. This is the sweet spot.
Markdown xprompt with frontmatter	Same as above	Inputs, snippets, skill metadata, and local helper xprompts.
Multi-agent markdown xprompt	Markdown file with top-level `---` segment separators	Fan-out or sequenced multi-agent work without needing a YAML workflow. Prefer this for most multi-agent work.
YAML xprompt workflow	`.yml` workflow file launched with `#!name` or embedded through a `prompt_part`	Real control flow: `agent`, `bash`, `python`, `parallel`, approvals, step outputs, artifact passing. Use it only when the structure earns its keep.

My recommendation is simple: prefer markdown xprompts, including multi-agent markdown xprompts, until you genuinely need a YAML workflow. YAML workflows are powerful, but power is how a three-line prompt becomes a small enterprise resource-planning system wearing a fake mustache.

Cases where YAML workflows really are necessary:

- `#!sase/fix_just`, because it has to run setup/repair/check steps around the agent.
- `#!sase/pylimit_split`, because it coordinates analysis and mechanical follow-up work.
- `#!sase/refresh_docs`, because it compares docs drift and launches targeted documentation work.
- Research swarms, because they need fan-out, aggregation, and artifacts.
- Bead epic/legend creation workflows, because they write SDD plans, initialize beads, and launch follow-up work.

The key distinction: a multi-agent markdown xprompt is excellent when the structure is "run these prompt segments, maybe with `%wait` ordering." A YAML workflow is for "run code, branch, gather outputs, call agents, validate, and continue."

```
---
input:
  target:
    type: str
    description: What to improve.
---
%n:plan
Plan a safe change for {{ target }}.
---
%n:code
%w:plan
#fork:plan
Implement the approved safe change for {{ target }}.
---
%n:review
%w:code
Review the diff and call out risks.
```

That is a multi-agent markdown xprompt. It is readable. It does not need a workflow engine. It can sit happily in `xprompts/three_phase.md` until the day it needs Bash, Python, or step outputs.

> **Friction note:** XPrompt discovery is intentionally flexible: repo-local, user-local, config-defined, plugin-shipped, and built-in sources all participate. That is powerful, but the mental model can get slippery. Use `sase xprompt list`, `sase xprompt explain`, and the ACE XPrompt Browser when you are not sure which `#thing` wins.

XPrompt Directives, In One Place

Directives are `%` tags that change launch behavior. They are extracted from the prompt before the agent sees it. The full reference is in XPrompts: Directives (https://sase.sh/xprompt/#directives); this is the practical tour.

Directive	Alias	What it does	Example
`%model`	`%m`	Select a provider/model. Parenthesized multi-values fan out into one agent per model.	`%m:claude/opus audit this API` or `%m(codex/gpt-5.5,claude/sonnet) compare the design`
`%name`	`%n`	Give an agent a stable name. Bare `%name` auto-names; `@` templates allocate suffixes; `!` force-reuses from confirmed TUI launches.	`%n:reviewer`, `%n:build-@`, `%n:!reviewer`
`%wait`	`%w`	Start only after named agents or workflows complete successfully. Bare `%wait` waits for the most recently named agent.	`%w:planner`, `%wait:agent1,agent2`, `%wait`
`%time`	`%t`	Delay launch by a duration or until wall-clock time. Dependencies wait first, then the time floor applies.	`%t:5m`, `%time:1h30m`, `%time:1430`, `%time:260415/0900`
`%hide`	`%h`	Hide the agent from the default Agents tab display. ACE can toggle hidden rows back into view.	`%h %n:background-log-checker inspect logs`
`%approve`	`%a`	Run autonomously without pausing for human approval.	`%a #!sase/fix_just`
`%epic`		Plan first, then auto-approve the submitted plan as an SDD epic with beads and follow-up work.	`%epic %n:checkout-epic plan the checkout rewrite`
`%edit`	`%e`	Open the editor and return the edited text to the ACE prompt bar instead of launching immediately.	`%e draft a careful prompt for the migration`
`%repeat`	`%r`	Run the same prompt serially multiple times; later slots wait on earlier slots. A slot can set `STOP` to stop the chain.	`%r:5 %n:flaky-repro try to reproduce the flaky test once`
`%group`	`%g`	Assign the agent's user-managed tag for ACE grouping and filtering.	`%g:review review the latest ChangeSpec`
`%alt`	`%(`	Split one prompt into variants. Named variants become child suffixes; multiple `%alt` and multi-model directives form a Cartesian product.	`%alt(sec=focus on auth,perf=focus on hot paths) review this diff`

Directives compose. This launches two named model variants, groups them under `review`, and keeps them hidden unless you toggle hidden agents:

```
sase run '%n:api-review %g:review %h %m(codex/gpt-5.5,claude/sonnet) review the API boundary'
```

And this chains a planner, coder, and reviewer without inventing a YAML workflow:

```
%n:planner
Plan a safe docs refresh.
---
%n:coder
%w:planner
#fork:planner
Implement the plan.
---
%n:reviewer
%w:coder
Review the diff and list follow-ups.
```

Forks Instead Of Multi-Turn Agents

SASE does not really have a first-class "multi-turn agent" concept. It has durable agent records, transcripts, artifacts, and `#fork`.

`#fork:<agent-name>` injects sanitized previous conversation context into a new agent prompt. That gives you the useful part of "continue this agent" without making every workflow depend on one mutable, forever-growing chat session. It also plays nicely with `%wait`: wait for the prior agent to finish, then fork from it.

```
sase run '%n:design #cd:$(pwd) propose a small architecture improvement'
sase run '%n:implement %w:design #fork:design implement the approved part only'
```

The forked agent gets a new record and a new workspace. You still have lineage, but the unit of work stays inspectable.

SDD: Prompts, Tales, Epics, And Beads

SASE's Spec-Driven Development (https://sase.sh/sdd/) directory is where agent intent becomes durable project state.

The core folders are:

- `sdd/prompts/`: expanded prompt snapshots. XPrompts are resolved, directives are stripped, and the exact prompt that launched work is saved with metadata.
- `sdd/tales/`: ordinary approved implementation plans. A tale is the plan you want a human or agent to understand later.
- `sdd/epics/`: executable multi-phase plans. An epic can be turned into Beads and driven by `sase bead work`.
- `sdd/beads/`: git-portable issue/dependency state: bead data, events, JSONL compatibility output, and the SQLite query cache.

The flow is intentionally concrete:

```
sase sdd list
sase sdd validate
sase bead ready
sase bead show <bead-id>
sase bead work <epic-id>
```

An epic can produce phase beads. Phase beads can depend on each other. `sase bead ready` shows only unblocked work, and `sase bead work <epic-id>` can launch agents for ready phases and then land the result when dependencies are satisfied.

This is where Steve Yegge's Beads (https://github.com/gastownhall/beads) influence is most obvious. Beads makes agent-friendly work items git-portable and dependency-aware. SASE borrows that spirit, then integrates it with SDD plans, ChangeSpecs, ACE, AXE, and local workspace orchestration.

> **Friction note:** SDD has a lot of nouns. Prompt, tale, epic, legend, bead, ChangeSpec. The nouns are there because the lifecycle stages are different, but the docs and UI need to keep doing better at teaching "what do I touch today?" versus "what exists for the full research roadmap?"

ACE: The Cockpit

`sase ace` opens the Agentic ChangeSpec Explorer, the terminal UI for daily work. ACE has three main tabs:

- **PRs**: ChangeSpecs, statuses, commits, hooks, comments, mentor output, diffs, file deltas, mail/submit flows, rewind, revert, restore, and archive operations.
- **Agents**: live and recent agents, groups, tags, hidden rows, child workflow steps, prompt panels, transcript panels, artifact viewers, tool metadata, file panels, retry/fork/wait/kill actions, and model/provider badges.
- **Axe**: the daemon view: lumberjacks, chops, run history, live output, wait checks, hook checks, mentor checks, comment polling, and error digests.

ACE is fun because it treats agents as work records, not mystical chat bubbles. You can fork an agent, wait on one, retry a failed run, inspect its artifacts, view its changed files, jump to the workspace, or hide background noise until you care about it. You can also open the XPrompt Browser, insert snippets, complete directives, complete file paths, and compose multi-agent prompts directly in the prompt input widget.

The VCS support is the part that makes ACE feel like engineering software instead of a prettier terminal. SASE's VCS providers are pluggy-based. Bare Git support ships with `sase`; GitHub support lives in `sase-github`. ACE shows the same review objects either way: file deltas, diffs, commit lists, ChangeSpec status, and provider-backed actions.

In practice, this means you can:

- inspect a ChangeSpec's diff from the TUI;
- view added/modified/deleted file counts and line deltas;
- rewind a PR/CL to an earlier state;
- revert an agent's committed work and archive the ChangeSpec;
- restore a reverted ChangeSpec by re-applying its diff;
- launch follow-up agents against the exact work record you are looking at.

AXE, Lumberjacks, And Chops

AXE (https://sase.sh/axe/) is the background daemon. It runs **lumberjacks**, and lumberjacks run **chops**. Yes, the naming theme got away from me. No, I am not apologizing yet.

A lumberjack is a scheduled lane of background work. A chop is one unit of work in that lane. Chops can be scripts or agent prompts. Built-in lumberjacks handle things like hook checks, wait checks, mentor checks, workflow cleanup, comment polling, stale-running cleanup, and error digests.

My own machine has more opinionated chops in my chezmoi config:

- `sase_fix_just`: periodically runs `#!sase/fix_just` against `sase` when there is not already an open fixer ChangeSpec.
- `sase_pylimit_split`: runs `#!sase/pylimit_split` to keep Python files from turning into archaeology sites.
- `sase_refresh_docs`, `sase_core_refresh_docs`, `sase_github_refresh_docs`, `sase_nvim_refresh_docs`, and `sase_telegram_refresh_docs`: keep docs fresh across linked repos when drift passes a threshold.
- `gh_actions_fix`: checks configured GitHub repositories for failed Actions runs, de-dupes seen failures, fetches logs, and launches a focused fixer agent.
- `tg_inbound` and `tg_outbound`: connect AXE to `sase-telegram`, polling chat input and sending notifications.

That is the pattern: AXE is not "one more agent." It is the supervisor that notices state changes and schedules the right scripts or agents.

> **Friction note:** AXE is powerful, but it needs more friendly defaults and clearer onboarding. The daemon model is correct; the "why is a lumberjack holding my CI logs?" learning curve is still a curve.

Telegram: The Pocket Cockpit

`sase-telegram` (https://github.com/sase-org/sase-telegram) gives SASE a chat bridge. It is implemented as AXE chops: an outbound chop reads notifications and sends them to Telegram, while an inbound chop polls Telegram and turns replies or slash commands into SASE actions.

Useful things it can do:

- notify you when an agent finishes, fails, asks a question, or needs plan approval;
- let you approve, reject, or give feedback on plans from your phone;
- list, kill, fork, retry, or inspect agents with slash commands;
- show ChangeSpec and Bead summaries;
- launch agents from messages, including messages with images or PDF attachments;
- keep outbound notifications quiet when ACE sees you actively working at the terminal.

Telegram is not meant to replace ACE. It is the thing you use when an agent asks a yes/no question while you are away from the keyboard and your laptop is, unreasonably, not strapped to your face.

Neovim, The XPrompt LSP, And The Prompt Widget

`sase-nvim` (https://github.com/sase-org/sase-nvim) is the canonical editor integration. The important idea is not "Neovim gets a plugin," although it does. The important idea is that SASE exposes an XPrompt language server.

The XPrompt LSP can provide:

- completion for `#xprompt`, `#!workflow`, slash skills, directives, arguments, and file paths;
- hover text for xprompt definitions and inputs;

- diagnostics for malformed references or arguments;
- go-to-definition for xprompt files;
- snippets and skeleton insertion for typed xprompt inputs;
- YAML schema help for workflow files.

ACE's prompt input widget overlaps with that on purpose. It uses the same catalog and helper machinery for directive completion, xprompt insertion, slash-skill insertion, argument hints, snippets, file completion, and prompt history.

The division of labor is ergonomic: ACE is fastest for launching and steering work in the cockpit; Neovim is better for writing longer prompt files, editing workflow YAML, navigating xprompt definitions, and using editor-native muscle memory. The same prompt system should feel familiar in both places.

> **Friction note:** The editor story should not be Neovim-only forever. `sase-nvim` is the reference client because I live there, but the LSP exists so other editors can use the same xprompt intelligence without copying SASE internals.

Scarcity Is Coming For Our Robot Budgets

I am using **AI era of scarcity** as my shorthand for a trend I picked up from The AI Daily Brief (https://podcasts.apple.com/us/podcast/the-ai-daily-brief-artificial-intelligence-news/id1680633614) and its "token scarcity" framing: the era of infinite-feeling subsidized inference is giving way to usage limits, provider tiers, routing decisions, and pricing details that matter.

SASE is an answer to that world because it assumes agent work should be schedulable, inspectable, and routable across providers. The `worker_models` config field is a small example:

```
llm_provider:
  provider: codex
  worker_models:
    claude: codex/gpt-5.5
    codex: claude/opus
```

That means delegated worker-lane jobs can use a different provider/model than the primary lane. If the primary planner is Claude, workers can go to Codex. If the primary planner is Codex, workers can go to Claude Opus. The goal is not "always use the biggest model." The goal is "put scarce reasoning where it matters and route routine follow-up work somewhere sensible."

The same idea appears in prompts:

```
sase run '%n:api-audit %m(codex/gpt-5.5,claude/sonnet) audit the API boundary and compare findings'
```

That launches a model fan-out. Sometimes the right answer is not trusting one model harder. Sometimes it is asking two models, comparing the overlap, and letting ACE keep the results from turning into tab soup.

> **Friction note:** SASE needs better budget visibility. It can route work today, but the future version should make cost, quota, rate limits, and provider health visible in the same way ACE makes agent state visible.

Useful Commands

These are the commands I reach for most:

Command	Why you use it
`sase doctor`	Read-only install, config, provider, project, and state diagnostics.
`sase version`	Exact SASE, Rust core, and plugin package inventory.
`sase ace`	Open the TUI cockpit.
`sase run "..."`	Launch an agent, xprompt, or workflow.
`sase agent list`	See active and recent agent runs from the terminal.
`sase xprompt list`	See available xprompts and workflows.
`sase xprompt explain "#foo"`	Inspect how a prompt reference resolves.
`sase xprompt graph "#!workflow"`	Visualize workflow structure.
`sase plan`	Review, approve, and manage submitted plans.
`sase sdd list` / `sase sdd validate`	Inspect and validate SDD artifacts.
`sase bead ready` / `sase bead work`	Find unblocked bead work or execute an epic.
`sase axe lumberjack status`	Check scheduled background automation.
`sase plugin doctor`	Verify plugin entry points and chop scripts.
`sase workspace open -p <linked_repo> -r "<reason>" <n>`	Open a configured linked repo's matching numbered workspace.
`sase lsp`	Start the XPrompt language server for editor integrations.
`sase mobile gateway start`	Start the workstation-hosted mobile gateway.

The CLI reference (https://sase.sh/cli/) is the full inventory.

The Papers Behind The Name

SASE is heavily inspired by the paper "Agentic Software Engineering: Foundational Pillars and a Research Roadmap" (https://arxiv.org/abs/2509.06216). The paper presents the **Structured Agentic Software Engineering (SASE)** vision, and this project takes its name and framing directly from that vocabulary. It splits the future of software engineering into **SE for Humans** and **SE for Agents**, then proposes two workbenches: **ACE**, the **Agent Command Environment**, where humans orchestrate and mentor agent teams, and **AEE**, the **Agent Execution Environment**, where agents execute work and call humans in for ambiguity or complex trade-offs. SASE maps that lineage into local tooling: its **ACE** cockpit is the **Agentic ChangeSpec Explorer**, but it deliberately echoes the paper's Agent Command Environment; **AXE** is the background execution/supervision daemon that echoes the paper's Agent Execution Environment.

SASE is also inspired by IBM's Prompt Declaration Language (https://github.com/IBM/prompt-declaration-language) and the PDL paper (https://arxiv.org/abs/2410.19135). PDL argues for declarative, composable prompt programs that keep prompts visible rather than burying them in framework code. SASE's YAML xprompt workflows borrow that idea, then specialize it for local software-engineering work: agent steps, Bash/Python steps, workspace references, SDD files, Beads, and VCS state.

The lesson from both papers is the same: agentic coding becomes agentic software engineering only when prompts, artifacts, process, and supervision become first-class.

Gas Town, Beads, And The Interface Question

Steve Yegge's Beads (https://github.com/gastownhall/beads) and Gas Town (https://docs.gastownhall.ai/) have also influenced SASE. Gas Town's docs describe a world of towns, rigs, the Mayor, Deacon, Witness, Refinery, crew workspaces, and polecat worker worktrees. The phrase from the docs that best captures the philosophy is the "Propulsion Principle": if work lands on an agent's hook, the agent runs it.

SASE agrees with the premise that agents can run agents and perform useful autonomous work. Where it differs is focus. Gas Town appears to explore what becomes possible when a town of agents dispatches and executes work through roles and autonomous propulsion. SASE assumes that premise is true, then asks a narrower product question: what is the right local interface for one developer supervising many coding agents across real repos, real diffs, real plans, and real PRs?

One concrete difference is xprompt workflows. SASE YAML workflows can intersperse agent calls with Python and Bash steps, pass outputs, branch, parallelize, and validate. I did not find an equivalent control surface in the public Gas Town docs; Gas Town's public model is more role/rig/dispatch oriented. That does not make one approach universally better. It just means SASE leans harder into "prompt/workflow as local programmable artifact."

Future Directions: Memory, Mobile, Web

Three directions are incomplete but important:

- Memory (https://sase.sh/memory/): SASE already has short-term project memory, audited long-term memory reads, and proposal-based writes. The next frontier is better retrieval, staleness handling, trust boundaries, and UI.
- Mobile: the mobile gateway (https://sase.sh/mobile_gateway/) and Android MVP work point toward a real mobile SASE client. Telegram covers a lot today, but a purpose-built app can expose richer state than chat buttons.
- Web: the Rust core boundary exists partly so a future web interface can share the same domain behavior as ACE, Telegram, and editor integrations instead of becoming a separate almost-SASE.

These are exciting because they all point at the same principle: the agent state should be durable and shared across surfaces. The terminal should not be the only window into the work.

> **Friction note:** The future-surface story is promising but unfinished. Today, ACE is the daily driver; Telegram and Neovim are useful companions; mobile and web are still early. The architecture is moving in the right direction, but nobody should pretend the phone app is already the Death Star. Also, given my luck, the exhaust port would be YAML.

The Point

Coding agents need more than better prompts. They need an operating layer: a place where intent, workspaces, plans, dependencies, review state, automation, notifications, and provider choices become explicit.

That is what SASE is trying to be.

Not the model. Not the IDE. Not the VCS host. The layer that lets those pieces cooperate without making a human keep the entire system in short-term memory and twelve terminal tabs.

The next post is the practical on-ramp: [01] Hello, SASE: Your First 15 Minutes (https://sase.sh/blog/posts/hello-sase-your-first-15-minutes/).

Series Navigation

This is [00] in the SASE Blog Series (https://sase.sh/series/agentic-software-engineering/).

- Previous: none.
- Next: [01] Hello, SASE: Your First 15 Minutes (https://sase.sh/blog/posts/hello-sase-your-first-15-minutes/).
- Continue reading: SASE Blog Series (https://sase.sh/series/agentic-software-engineering/), blog home (https://sase.sh/blog/), or ACE guide (https://sase.sh/ace/).

[01] Hello, SASE — Your First 15 Minutes Orchestrating Coding Agents

SASE (pronounced "sassy" — yes, really) is a coordination layer that sits above coding-agent CLIs like Claude Code, Codex, or Antigravity CLI (`agy`). This post is the practical on-ramp: by the end you'll have installed `sase`, checked that a provider CLI is ready, launched a safe read-only agent run, found the resulting agent record, and picked up the vocabulary you'll keep bumping into in the rest of the docs. Plan on roughly fifteen minutes at a terminal, plus however long your favorite model takes to think.

This is [01] in the SASE Blog Series. If you'd rather read about *why* a system like this exists before touching it, [00] The Missing Operating Layer for Coding Agents (https://sase.sh/blog/posts/why-coding-agents-need-orchestration/) makes that argument. The two posts can be read in either order; this one runs first and names the parts afterward.

Step 1 — Install SASE (≈90 seconds)

SASE needs Python 3.12+, `uv` (https://docs.astral.sh/uv/), and one authenticated coding-agent CLI such as Claude Code, Codex, Antigravity CLI (`agy`), Qwen Code, or OpenCode. With Python and `uv` in place:

```
uv tool install sase --python 3.12
sase version
```

If `sase version` prints the SASE package plus the `sase-core-rs` package, the CLI is installed. The first install can stretch past 90 seconds when `uv` has to fetch wheels — that's normal, not a hang.

What you just did. Installed the public `sase` CLI and its Rust core extension as a user tool, without cloning the repository or setting up a contributor environment.

Step 2 — Check provider readiness (≈2 minutes)

SASE orchestrates an existing provider CLI; it does not replace that provider's own install or authentication flow. Run the read-only doctor before the first agent launch:

```
sase doctor
```

If the provider check reports a missing executable or an authentication gap, install and authenticate one provider CLI, then run `sase doctor` again. The LLM provider reference (https://sase.sh/llms/) keeps the setup pointers in one place so this quickstart can stay focused.

What you just did. Verified that SASE can find a usable coding-agent provider before spending time on an agent run.

Step 3 — Launch a safe first agent (≈3 minutes, plus model time)

Start with a read-only task in the directory you want the agent to inspect:

```
sase run "#cd:$(pwd) summarize what this repository does; do not change files"
sase agent list
```

The `#cd:$(pwd)` prefix makes the target workspace explicit. `sase run` allocates an isolated **workspace** — a sibling clone of the repo named `sase_<N>` — and runs the provider CLI there. That isolation is what lets you fire off several agents at once without them colliding, and what lets a failed run be retried without touching your real checkout.

The launched agent gets its own durable record on disk: prompt, reply transcript, artifacts directory, status, and workspace path. `sase agent list` gives you the first visible handle for that record while the model is thinking or after it finishes.

What you just did. Dispatched a read-only coding-agent run inside an explicit workspace (https://sase.sh/workspace/), then looked up the resulting SASE agent record.

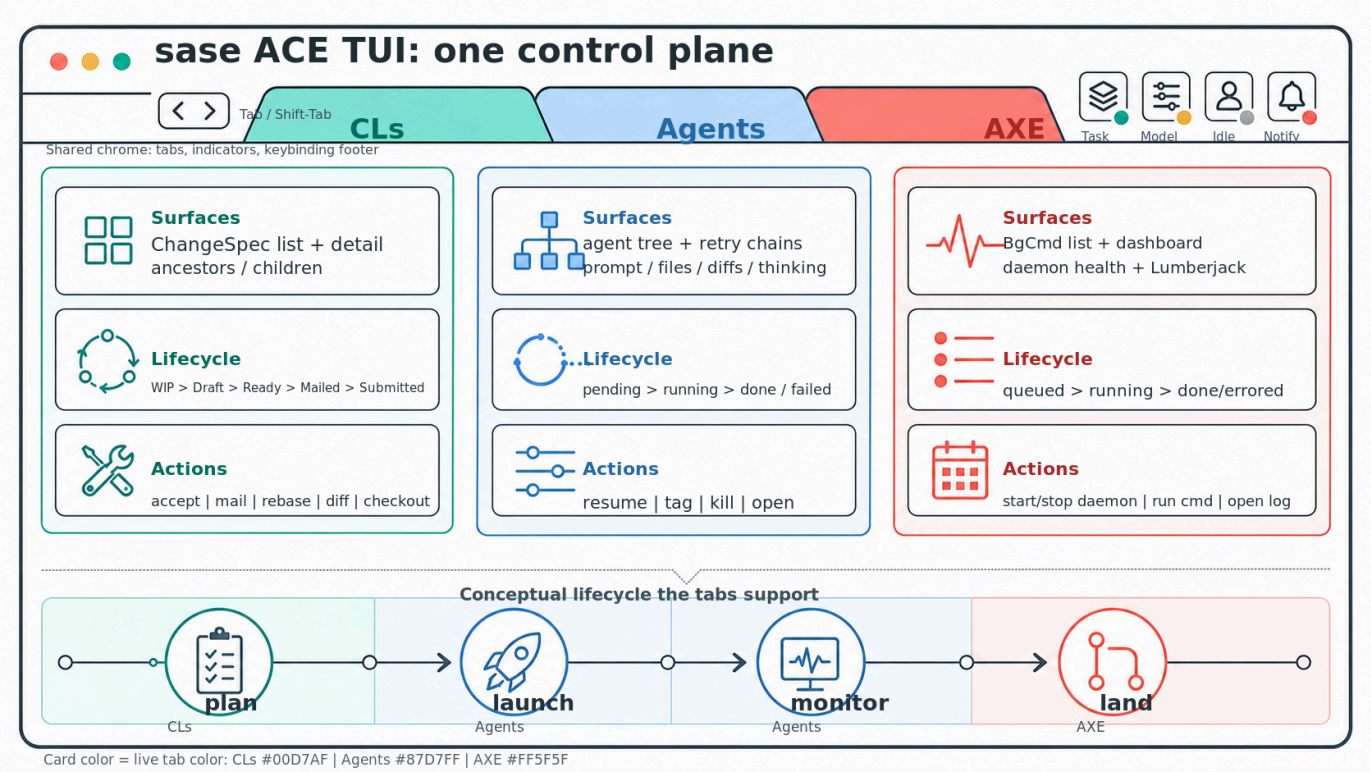
Step 4 — Open ACE and find the result (≈3 minutes)

ACE is the TUI control surface. Open it:

```
sase ace
```

ACE has three tabs:

- **Agents** — live and recent agent records. Find the run you just launched: prompt, reply transcript, workspace path, status, retry chain.
- **PRs** — every ChangeSpec on this project. A **ChangeSpec** is SASE's durable record of one CL/PR-sized unit of work; think of it as the long-lived sibling of a pull request that holds the description, parent, status (WIP → Draft → Ready → Mailed → Submitted), commits, hooks, comments, and mentor activity all in one place. The ChangeSpec guide (https://sase.sh/change_spec/) goes deeper when you're curious.
- **Axe** — the background daemon's view: scheduled jobs, hooks waiting to complete, mentor launches, error digests. ACE auto-starts AXE the first time it opens, so this tab is already ticking before you click it.



What you just did. Observed one `sase run` produce a persistent agent artifact visible in ACE (https://sase.sh/ace/), with AXE (https://sase.sh/axe/) handling lifecycle work in the background.

Step 5 — Try one tiny edit (≈3 minutes, plus model time)

After you have seen the agent record, try a low-risk change:

```
sase run "#cd:$(pwd) make a tiny documentation-only improvement and explain the diff"
sase agent list
```

Now the agent has permission to make a visible diff in its isolated workspace. Review the resulting workspace or ChangeSpec before bringing anything back to your primary checkout.

What you just did. Moved from a read-only run to a small editable task after confirming where SASE records agent state.

Step 6 — Reuse the prompt as an XPrompt (≈3 minutes)

A one-off prompt is fine once. The second time you find yourself reaching for it, wrap it as an **XPrompt** so you're not retyping the same paragraph forever. Create `xprompts/docstring.md` in your project root:

```
Add a one-line docstring to the most recently edited Python function in this repo. Keep the wording terse; do not change
behavior.
```

Now the same agent run is one tag:

```
sase run "#cd:$(pwd) #docstring"
```

That is the smallest XPrompt shape — a single Markdown file becomes a reusable prompt part. XPrompts also support YAML files with typed inputs, multi-step workflows (prompt parts, Python, bash, parallel fan-out, approvals), and `---` separators for multi-agent dispatch. The XPrompts guide (https://sase.sh/xprompt/) covers the full surface, and the workflow spec reference (https://sase.sh/workflow_spec/) documents the YAML form.

What you just did. Turned a one-off prompt into a reusable XPrompt, the smallest unit of repeatable agent work in SASE.

Step 7 — Plan bigger work with SDD and Beads (≈3 minutes)

When a task is too big to hand to a single agent and hope, SASE asks you to write a plan first. **Spec-Driven Development (SDD)** keeps those plans as first-class artifacts on disk under three (admittedly whimsical) names: ordinary plans are *tales*, executable multi-phase plans are *epics*, and longer cross-cutting plans are *legends*. Any of them can be filed as a **bead** — a git-portable, issue-like work unit with status, dependencies, and an assignee.

The smallest useful loop:

```
sase bead onboard         # walks through the issue-tracking quick start
sase bead ready           # lists work whose blockers are closed
sase bead show <bead-id>  # inspects one bead in detail
```

Once an epic plan exists and its phase beads are filed, `sase bead work <epic-id>` builds a dependency schedule from the open phases, pre-claims each phase bead, launches one agent per phase in the right order, and runs a final land agent after the phases finish. That's the on-ramp from one-shot prompts to multi-agent execution with actual ordering — no more babysitting `sase run` calls in a shell loop.

What you just did. Stepped from one-shot prompts into Spec-Driven Development (https://sase.sh/sdd/) with Beads (https://sase.sh/beads/) as dependency-aware work units.

The component map (recap)

The names you'll keep bumping into, in one place:

- **ACE** (https://sase.sh/ace/) — the TUI control surface for ChangeSpecs, agents, notifications, and automation.
- **AXE** (https://sase.sh/axe/) — the background automation daemon. Runs hooks, mentor launches, comment polling, dependency unblocking, error digests.
- `sase run` — the entry point that launches an agent or workflow. See the CLI reference (https://sase.sh/cli/).
- **Workspaces** (https://sase.sh/workspace/) — isolated `sase_<N>` clones of the repo so agents can work in parallel without touching your checkout.
- **ChangeSpecs** (https://sase.sh/change_spec/) — durable CL/PR-sized review records: status lifecycle, commits, hooks, comments, mentors.
- **Beads** (https://sase.sh/beads/) — dependency-aware, git-portable work units. Powers epic execution.
- **XPrompts** (https://sase.sh/xprompt/) — reusable prompt templates and YAML workflows with typed inputs and multi-agent fan-out. See also workflow specs (https://sase.sh/workflow_spec/).
- **SDD** (https://sase.sh/sdd/) — Spec-Driven Development. Plans, epics, and legends as first-class artifacts on disk.
- **Plugins and providers** (https://sase.sh/plugins/) — model and VCS providers behind a common boundary: Claude Code, Antigravity CLI (`agy`), Codex, Qwen Code, OpenCode for agents; bare git and GitHub for version control.

What to read next

- [00] The Missing Operating Layer for Coding Agents (https://sase.sh/blog/posts/why-coding-agents-need-orchestration/) — the conceptual half of the series, for when you want the *why* to match the *how*.
- SASE Blog Series (https://sase.sh/series/agentic-software-engineering/) — all ten posts in one place.
- CLI reference (https://sase.sh/cli/) — every `sase` subcommand on one page.
- The SASE repository (https://github.com/sase-org/sase) — source, issues, and project direction. If something on this page didn't work, an issue is the fastest way to make the next reader's first 15 minutes smoother.

Series Navigation

This is [01] in the SASE Blog Series (https://sase.sh/series/agentic-software-engineering/).

- Previous: [00] The Missing Operating Layer for Coding Agents (https://sase.sh/blog/posts/why-coding-agents-need-orchestration/).

- Next: more series posts are forthcoming.
- Continue reading: SASE Blog Series (https://sase.sh/series/agentic-software-engineering/), blog home (https://sase.sh/blog/), or ACE guide (https://sase.sh/ace/).