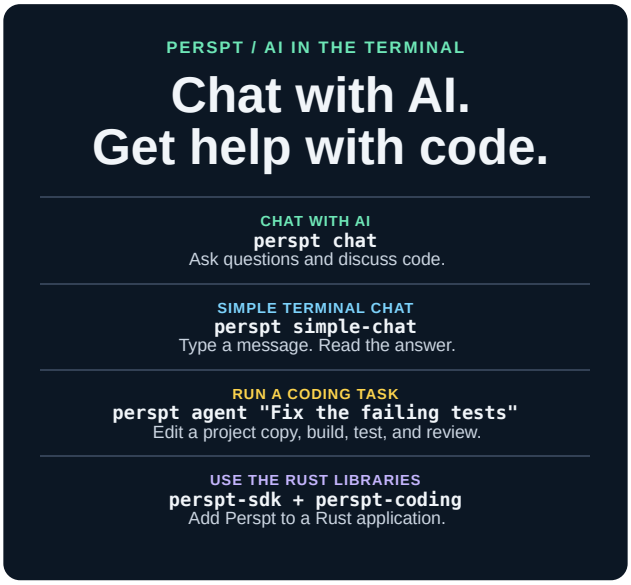
PERSPT :: DEVELOPER HANDBOOK

v0.6.7

AI chat and coding in the terminal.

A practical guide to Perspt's plain terminal chat, full-screen UI, coding agent, and Rust libraries.



01 / TERMINAL CHAT
Line-oriented `simple-chat` and full-screen `perspt chat` with cloud or local models.

02 / CODING AGENT & RUST SDK
Ask `perspt agent` to edit code and run builds and tests. Use the libraries from Rust.

Vikrant Rathore

EDITION 0.6.7 · OMNI

Contents

1 Introduction to Perspt 6
1.1 Choose a way to work 6
1.2 Give the coding agent a task 6
1.3 Watch, review, and continue the work 7
1.4 Use the Rust libraries 7
2 Quick Start 8
2.1 1. Install the CLI 8
2.2 2. Connect a model 8
2.3 3. Ask a question 8
2.4 4. Try a code change 9
2.5 5. Watch or automate a run 9
2.6 Next steps 9
3 Installation 10
3.1 Build and install from source 10
3.2 Connect a model 10
3.3 Prepare a coding project 11
3.4 Process isolation 11
3.5 Build the documentation 12
4 Getting Started 13
4.1 Choose a project task 13
4.2 Review the result 13
4.3 Watch a run 13
4.4 Automate after a successful trial 13
4.5 Inspect or continue a session 14
4.6 Read further 14
5 Tutorials 15
5.1 Learning Path 15
5.2 First Chat 15
5.3 Agent Mode Tutorial 16
5.4 Headless Mode 18
5.5 Local Models with Ollama 21
5.6 Custom Workflows 22
5.7 Example: Python ETL Pipeline 23
5.8 Example: Rust CLI Tool 24
5.9 Example: Scientific Computing 26
5.10 Monitoring Agent Execution with the Dashboard 27
6 User Guide 30
6.1 Basic Usage 30
6.2 Agent Mode 32
6.3 Advanced Features 36

6.4 Providers 39
6.5 Model Context Protocol (MCP) 40
6.6 Troubleshooting 45
6.7 Dashboard 47
7 Concepts 49
7.1 SRBN Architecture 49
7.2 Stability and Measured Progress 57
7.3 Workspace Crates 59
7.4 PSP Process 65
8 How-To Guides 68
8.1 Agent Options Reference 68
8.2 Configuration 70
8.3 Set Up Providers 73
8.4 Security and Policy Rules 74
8.5 Dashboard Setup 76
9 Configuration Guide 79
9.1 Automatic Provider Detection 79
9.2 Vertex AI authentication 79
9.3 Supported Models & Naming Conventions 80
9.4 Configuration File 80
9.5 Agent Configuration Blocks 81
9.6 Command-Line Flags 83
9.7 Dashboard Configuration 84
10 Reference 85
10.1 CLI Reference 85
10.2 Key Bindings 89
10.3 Advanced Troubleshooting 90
11 API & SDK Reference 93
11.1 Integrating the Perspt Omni SDK 93
11.2 `perspt-core` 94
11.3 `perspt-sdk` 96
11.4 `perspt-coding` 98
11.5 `perspt-research` 99
11.6 `perspt-agent` 99
11.7 `perspt-tui` 101
11.8 `perspt-cli` 102
11.9 `perspt-store` 103
11.10 `perspt-policy` 104
11.11 `perspt-sandbox` 104
11.12 `perspt-dashboard` 105
11.13 `perspt-prompt-macros` 107
11.14 `perspt-benchmark` 108
12 Developer Guide 110
12.1 Architecture 110
12.2 Contributing 119

12.3 Extending Perspt 121
12.4 Testing 124
12.5 Documentation 127
13 Changelog 129
13.1 Version 0.6.7 - “Omni” 129
13.2 Version 0.6.6 - “ (Shen DaiJun)” 129
13.3 Version 0.6.2 - “Hózhó” 135
13.4 Version 0.6.1 - “AKU” 135
13.5 Version 0.6.0 - “kukuza” 136
13.6 Version 0.5.9 - “xinli guanghua” 137
13.7 Version 0.5.8 - “Qualitaetsveredelung” 137
13.8 Version 0.5.7 - “navikaran” 139
13.9 Version 0.5.6 - “ikigai” 139
13.10 Version 0.5.5 141
13.11 Version 0.5.4 141
13.12 Version 0.5.3 142
13.13 Version 0.5.2 142
13.14 Version 0.5.1 142
13.15 Version 0.5.0 143
14 License 144
14.1 LGPL v3 License 144
14.2 What This Means 144
14.3 Third-Party Licenses 145
14.4 License Compatibility 145
14.5 Commercial Use 146
14.6 Trademark Policy 146
14.7 Contributing and License 146
14.8 License FAQ 147
14.9 Getting Legal Advice 147
14.10 Reporting License Issues 147
15 Acknowledgments 149
15.1 Theoretical Foundation 149
15.2 Core Dependencies 149
15.3 Documentation Tools 150
15.4 Development Tools 151
15.5 Community and Inspiration 151
15.6 Testing and Quality Assurance 152
15.7 Special Thanks 152
15.8 Contributing Back 153
15.9 License Information 153
15.10 Get Involved 153

Learn to use Perspt for AI chat and coding in the terminal. We start with installation and a conversation, then use the coding agent to work on a project and review the result.

Where to start

Task	Guide
Try Perspt	Quick Start: Install Perspt, connect a provider, and ask a question.
Use a local model	Local Models with Ollama: Connect Ollama and select an installed model.
Run a coding task	Agent Mode Tutorial: Ask the agent to edit code, run tests, and show the result for review.
Run without prompts	Headless Mode: Automate a task and save its result as JSON.
Use Perspt from Rust	API & SDK Reference: Choose the libraries needed for a Rust application.

What Perspt does

- **AI chat** with `perspt chat`: a full-screen conversation with formatted answers, code, and multiline input.
- **Simple terminal chat** with `perspt simple-chat`: type a message, read the answer, and optionally save the conversation.
- **Coding tasks** with `perspt agent`: ask the agent to fix a bug, add a feature, or write tests. It edits a separate copy and runs the project's tools before asking for review.
- **Rust libraries** for adding AI workflows, build and test checks, and session history to another application.

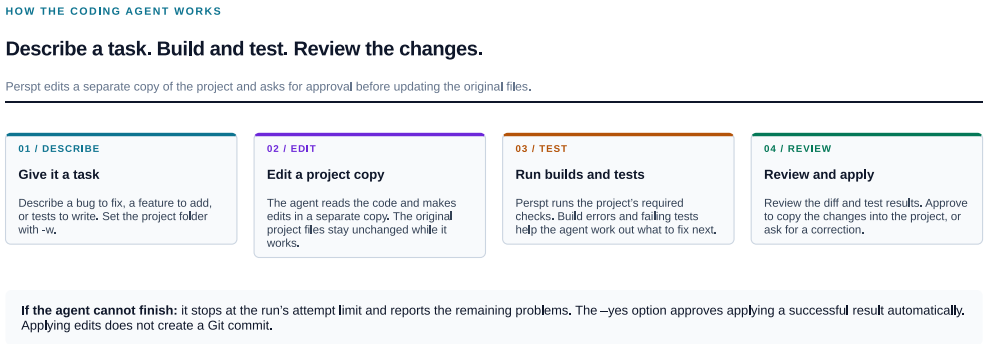


Fig. 1 The coding agent asks for approval before copying a successful result into the project. If it cannot finish within the run's limits, it reports the remaining problems.

Agent mode is experimental. We review the code and tests before committing, because passing checks can miss bugs. Introduction to Perspt explains the main commands and SRBN Architecture covers the implementation in detail.

Introduction to Perspt

Perspt is an AI chat and coding assistant that runs in the terminal. We can ask questions in chat, get help understanding code, or give the coding agent a task to carry out in a project. It connects to cloud providers and local models through Ollama.

Choose a way to work

Task	Command or library	What it does
Chat with AI	`perspt chat` or `perspt`	Opens a full-screen conversation with formatted answers, code, and multiline input.
Use simple terminal chat	`perspt simple-chat`	Reads a message one line at a time and prints the answer. It can also save a conversation to a file.
Work on a codebase	`perspt agent -w ./app "Fix the parser tests"`	Reads the project, edits a separate copy, runs the build and tests, and asks for review before applying changes.
Add Perspt to a Rust application	`perspt-sdk` and related crates	Provides libraries for building AI workflows, running checks, and recording what happened.

Follow Quick Start to install Perspt and start a conversation. Basic Usage covers chat controls and Providers explains provider setup.

Give the coding agent a task

A useful task describes the feature or bug, the project folder, and the expected behavior:

```
perspt agent -w ./app "Add CSV export with tests for empty input"
```

The agent reads the repository and edits a separate copy of it. Perspt runs the tools provided by the detected language plugin, such as `cargo check` and `cargo test` for Rust. The built-in plugins support Rust, Python, and JavaScript/TypeScript projects. If a build or test fails, the agent uses the reported errors to try a correction. Limits on attempts and tool calls prevent a run from continuing indefinitely.

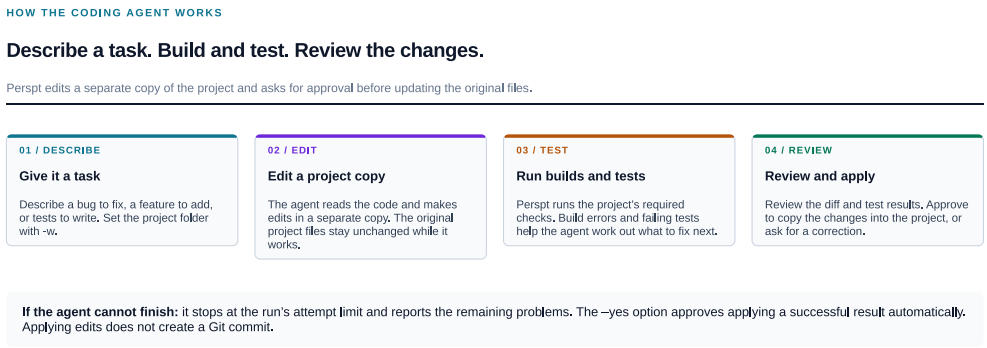


Fig. 2 The project files are updated after the required checks pass and the changes are approved. Perspt does not create a Git commit.

Review the code and the tests. A passing suite can miss bugs or leave requirements untested. Tests can change with the implementation by default. For tasks that must preserve existing behavior, the `backward-compatible` policy also runs the historical tests. An `external-oracle` policy adds acceptance tests maintained separately from the agent's work. See Configuration Guide for these settings.

Watch, review, and continue the work

- **Review edits in the terminal.** The agent shows the diff and the check results before applying changes. `--yes` approves this step automatically for unattended runs.
- **Watch in the browser.** Add `--dashboard` and open `http://127.0.0.1:3000` while the agent runs.
- **Read the session history.** `perspt replay <session-id>` reads recorded actions without calling a model or rerunning the tools.
- **Continue an interrupted run.** `perspt resume --last` restores the saved progress and checks that the run is still authorized.
- **Undo applied edits.** `perspt ledger --rollback <session-prefix>` undoes that session's latest set of applied changes. It refuses to overwrite files changed afterward.

Linux uses Bubblewrap and macOS uses `sandbox-exec` to isolate the agent's child processes. Native Windows requires an explicit option to run without this isolation, or WSL. See Installation for setup and Agent Mode for the full agent guide.

Use the Rust libraries

Perspt's libraries can be used from another Rust application. `perspt-agent` runs coding sessions, `perspt-core` connects to model providers, and `perspt-store` saves session history. `perspt-sdk` and `perspt-coding` provide the types and checks used by the agent. The research package is an early example of another domain, not a complete research assistant.

API & SDK Reference contains a crate guide and a Rust example. The architecture and mathematical details are in SRBN Architecture.

Quick Start

Start with a conversation, then try a small code change. For chat, we need access to an AI model. For coding tasks, we also need the project's compiler and test tools.

1. Install the CLI

From the repository root, Rust 1.100 or later can build and install the CLI:

```
git clone https://github.com/eonseed/perspt.git
cd perspt
cargo install --locked --path crates/perspt-cli --features bundled
perspt --version
```

The `bundled` feature builds DuckDB into the executable, so we do not need a separate DuckDB library. Compilation can take several minutes. See Installation for platform and agent prerequisites.

2. Connect a model

Set one provider key in the terminal where Perspt will run:

```
export GEMINI_API_KEY="<api-key>"
perspt chat
```

OpenAI and Anthropic use `OPENAI_API_KEY` and `ANTHROPIC_API_KEY` respectively. With several credentials set, an explicit `provider::model` route selects the intended provider. Model availability depends on the provider account.

For a local conversation, start Ollama in another terminal, then pull and select a model:

```
ollama pull llama3.2
perspt chat --model ollama::llama3.2
```

See Local Models with Ollama for local setup and Providers for other providers.

3. Ask a question

In the full-screen chat, type a question such as `Explain the difference between a Rust String and &str` and press **Enter**. Use **Ctrl+J** for a newline, **Page Up** / **Page Down** to scroll, and **Esc** to exit. Configured read-only MCP operations are available through `/mcp`.

For a line-oriented interface:

```
perspt simple-chat --log-file chat.txt
```

Type a message and press **Enter**; **Ctrl+D** exits. This command takes no positional prompt argument. It reads piped input one non-empty line at a time, and its output includes the banner, input echo, and session messages as well as model responses. For a single piped question:

```
printf '%s\n' 'Explain Rust ownership in three sentences.' | perspt simple-chat
```

4. Try a code change

Start in an existing project with working local checks. For a Rust project, run `cargo test` first, then give Perspt a specific task:

```
perspt agent -w ./my-project \
  "Add tests for empty input in the parser. Preserve the public API."
```

The agent edits a separate copy of the project and runs its build and tests. It uses error messages to try corrections and asks for review before copying the result back. Applying edits does not create a Git commit. Agent mode is experimental; required tools and platform isolation are described in Installation.

Review the diff and the tests before accepting the result. Passing tests can miss bugs or leave requirements untested. If the agent runs out of attempts, it stops and reports the remaining errors.

5. Watch or automate a run

```
# Live local browser view at http://127.0.0.1:3000
perspt agent --dashboard -w ./my-project "Fix the failing parser test"

# Non-interactive run; verification still applies
perspt agent --yes --output-summary summary.json \
  -w ./my-project "Add tests for empty parser input"
```

`--yes` automatically approves copying a successful result into the project. It is required for runs without an interactive terminal. An incomplete or failed run exits with a nonzero status. The embedded dashboard stops when the agent exits.

Next steps

- Agent Mode Tutorial: Follow a complete coding task and inspect the result.
- Configuration Guide: Choose providers, test policies, and verification settings.
- Agent Mode: Review edits, continue interrupted work, and read session history.
- API & SDK Reference: Reuse the Rust SDK.

Installation

Chat needs the Perspt binary and a model provider. Coding tasks also need the project's own build and test tools. The documentation publisher has no Python runtime dependency.

Build and install from source

We need Rust 1.100 or later and a native C/C++ build toolchain for the bundled DuckDB build. From the repository root:

```
git clone https://github.com/eonseed/perspt.git
cd perspt
cargo install --locked --path crates/perspt-cli --features bundled
perspt --version
perspt --help
```

Cargo installs the executable on its binary path, usually `~/.cargo/bin`. The `bundled` feature compiles DuckDB into the executable. Compilation can take several minutes.

To build without installing:

```
cargo build --locked -p perspt-cli --release --features bundled
./target/release/perspt --version
```

The workspace root is a virtual manifest, so local installation targets `crates/perspt-cli`. The `perspt` crate is a library facade; the executable belongs to `perspt-cli`.

For development builds without `bundled`, we need the exact DuckDB shared library matching `Cargo.toml`. The CI setup in `.github/actions/setup-duckdb/action.yml` pins the library and its checksums. See Contributing for development checks.

Connect a model

Set one credential and start chat:

```
export GEMINI_API_KEY="<api-key>"
perspt chat
```

Other supported providers and explicit `provider::model` routes are described in Providers. A modern terminal with UTF-8 and color support gives the best full-screen experience. `perspt simple-chat` provides line-oriented terminal chat.

For local execution, start `ollama serve` in another terminal, pull a model, and select it explicitly:

```
ollama pull llama3.2
perspt chat --model ollama::llama3.2
```

Prepare a coding project

Run the project's checks locally before starting an agent task. Verification processes run without network access under the normal sandbox policy. We install or sync dependencies first, so checks can run offline. `--allow-dependency-mutation` grants agent dependency operations when needed; it does not bypass other policies.

Project	Tools and preparation
Rust	Install Cargo and rustc. Add `rust-analyzer` with `rustup component add rust-analyzer`. Fetch dependencies and confirm `cargo check` and `cargo test` work.
Python	Install Python and uv. Sync the project's environment, including pytest. The syntax stage uses host `ty` when available, otherwise offline `compileall` through uv; Pyright is a separately detected fallback. The LSP prefers `ty` with `pyright-langserver` as fallback.
JavaScript / TypeScript	Install Node.js, npm, TypeScript, and `typescript-language-server`. Install project dependencies and configure the build and test scripts. The plugin runs `npx tsc --noEmit`, `npm run build`, and `npm test`.

The built-in plugins cover Rust, Python, and JavaScript/TypeScript. There is no built-in Go plugin. Python project tools are optional for working on Python projects and benchmark fixtures; they are not needed to publish these docs.

Process isolation

Linux agent runs use Bubblewrap, which must be installed and usable on the host. macOS uses `sandbox-exec`. A missing required isolation backend blocks process execution by default.

Chat and plain terminal chat work natively on Windows. Native Windows has no Perspt OS process-sandbox backend. WSL provides the Linux isolation path. To explicitly permit host-user execution on native Windows:

```
perspt agent --allow-unisolated -w . "Fix the parser tests"
```

For resumed runs, the durable setting is:

```
[verification]
allow_unisolated = true
```

The agent still works on a project copy, limits permitted edits, runs required checks, and asks for approval. Child processes in reduced isolation have the host user's filesystem and network access.

Build the documentation

Install Guidedog and confirm `guidedog --version` succeeds. Guidedog includes its PDF and Graphviz engines. The standalone Typst CLI is needed when changing the diagram and cover sources; builds can otherwise use the checked-in assets.

```
# Complete website, book PDF, and proposal PDFs
cargo run -p xtask -- docs

# HTML only
cargo run -p xtask -- docs --no-pdf

# Regenerate figures and cover artwork after source edits
cargo run -p xtask -- assets

# Live preview
cargo run -p xtask -- serve
```

The site is written to `docs/_build/site/index.html`. The book PDF is written to `docs/_build/site/books/perspt-book-en.pdf` and copied to `docs/perspt.pdf`. See Documentation for the publication checks.

Getting Started

Start with a conversation, then try the coding agent on a project. Follow Quick Start for installation and a first conversation. This page gives a short path through the next tasks.

Choose a project task

Start in a project whose tests already run locally. A small task with an observable result is easiest to review:

```
perspt agent -w ./my-project \
  "Add parser tests for empty input and malformed headers. Preserve the public API."
```

Perspt reads the code, edits a separate project copy, and runs the required checks. Build errors and failing tests guide further corrections. The agent asks for review before applying a successful result. If it runs out of attempts, it stops and reports what remains to be fixed.

Review the result

Inspect the diff and the build and test results in the review window. Approve the edits to copy them into the project, or ask the agent for a correction. Perspt does not create a Git commit.

We still assess the tests and the intended behavior. Passing tests can miss bugs or leave requirements untested. The default test policy lets tests evolve with implementation. Configuration Guide explains how to require existing tests or separately maintained acceptance tests.

Watch a run

```
perspt agent --dashboard -w ./my-project "Fix the failing parser test"
```

Open `http://127.0.0.1:3000` while the agent runs. The embedded dashboard shares the agent's session store and stops when the agent exits. For completed sessions, use `perspt dashboard`. See Dashboard for details.

Automate after a successful trial

```
perspt agent --yes --max-turns 12 --rejection-budget 4 \
  --output-summary summary.json \
  -w ./my-project "Add parser tests for empty input"
```

`--yes` automatically approves applying a successful result. It is required without a terminal. An incomplete or failed run exits with a nonzero status. See Headless Mode for automation examples.

Inspect or continue a session

```
perspt status
perspt ledger --recent
perspt replay <session-id>
perspt resume --last
```

Replay reads the saved session history without calling the model or rerunning tests. Resume restores saved progress and checks that the run is still authorized. `perspt ledger --rollback <session-prefix>` undoes the latest set of edits applied by that session. It refuses to overwrite files changed afterward.

Read further

- Agent Mode Tutorial: A complete example with review and verification.
- Basic Usage: Chat commands and keyboard controls.
- Installation: Project tools and platform isolation requirements.
- API & SDK Reference: Use Perspt's libraries from a Rust application.

Tutorials

Step-by-step guides to master Perspt.

Learning Path

Order	Tutorial	Outcome
1	First Chat	Understand basic TUI interaction
2	Local Models with Ollama	Set up Ollama for privacy
3	Agent Mode Tutorial	Master multi-file autonomous coding
4	Headless Mode	Run agent without interactive prompts
5	Custom Workflows	Automate with scripts and CI/CD
6	Example: Python ETL Pipeline	Build a real Python project from scratch
7	Example: Rust CLI Tool	Build a real Rust project from scratch
8	Example: Scientific Computing	Apply agent mode to scientific computing
9	Monitoring Agent Execution with the Dashboard	Monitor agent sessions via the web dashboard

First Chat

Start your first conversation with an LLM using Perspt.

Prerequisites

- Perspt installed (see Installation)
- An API key from any provider, or Ollama running locally

Step 1: Set Your API Key

```
export GEMINI_API_KEY="your-key"
```

Step 2: Launch the TUI

```
perspt
```

Perspt auto-detects the provider from the environment variable and launches the chat TUI. You will see a status bar showing the provider and model.

Step 3: Chat

Type a message and press **Enter**. Perspt streams the response in real time with markdown formatting (code blocks, headers, lists, bold, italic).

```
You: Explain Rust's ownership model in 3 sentences.

Assistant: Rust's ownership model ensures each value has exactly one owner at a
time. When the owner goes out of scope, the value is automatically dropped.
Borrowing rules allow temporary references without transferring ownership, and
the compiler enforces these rules at compile time to prevent data races and
dangling pointers.
```

Step 4: Navigate

Key	Action
Enter	Send message
Ctrl+J	Insert newline in input
Page Up / Down	Scroll chat history
Ctrl+Up / Down	Scroll one line
Esc or **Ctrl+Q**	Exit
/save	Save conversation to file

Step 5: Try Simple CLI Mode

For a minimal interface without the TUI:

```
perspt simple-chat
# Or with logging:
perspt simple-chat --log-file session.txt
```

Type your question, get a streamed text answer. Type `exit` or `Ctrl+D` to quit.

Next Steps

- Agent Mode Tutorial - Try autonomous multi-file coding
- Local Models with Ollama - Use Ollama for offline conversations

Agent Mode Tutorial

Ask Perspt to add a small feature, run the tests, and show the changes for review. In this tutorial we add a function to a Rust library. The task uses only the standard library, so no new dependencies are needed.

Prerequisites

- Perspt installed and a model provider configured; see Quick Start.
- Cargo and rustc installed, with the project's required tools available.
- Process isolation set up for agent mode; see Installation.

Agent mode is experimental. A run may stop before finishing, and passing tests can still miss bugs. We review both the code and the tests before accepting the result.

1. Create a small project

In a directory outside the Perspt source checkout:

```
cargo new --lib parser-demo
cd parser-demo
cargo test
```

The generated library has a passing test. This gives the agent a working starting point.

2. Describe the function and its tests

```
perspt agent -w . \
  "Add a public parse_count function to src/lib.rs.
   It takes a &str and returns Result<u32, std::num::ParseIntError>.
   Trim whitespace before parsing. Reject empty input, negative numbers,
   and non-numeric text. Add tests for valid input, whitespace,
   empty input, negative numbers, and invalid text."
```

The prompt gives a name, a signature, and examples of expected behavior. These details make the result easier to test and review.

3. Follow the build and test results

The agent reads the project and edits a separate copy. Perspt runs the Rust checks and sends any build or test errors back to the agent. The agent can try corrections within the run's limits.

Keep improvements. Reject regressions. While the required checks still fail, Perspt can save an attempt that reduces its error score enough. An attempt that fails this progress check is discarded in favor of the best saved version. The original project files are updated only after the required checks pass and the result is approved.

To watch in a browser, add `--dashboard` to the command and open `http://127.0.0.1:3000` while the agent runs. See Monitoring Agent Execution with the Dashboard for the dashboard pages.

4. Review the diff

When a result is ready, the review window shows the edited files, diff, and check results.

- **y** approves applying the changes.
- **n** rejects the changes.
- **c** sends feedback so the agent can try a correction.
- **d** shows the full diff.

Check the implementation of `parse_count` and the assertions in its tests. For example, a test for negative input should assert that parsing returns an error, rather than silently converting the value.

Approval copies the edited files into the project. Perspt does not create a Git commit.

5. Inspect the result locally

```
cargo test
git diff -- src/lib.rs
```

A successful result defines `parse_count` and tests the cases requested in the prompt. The exact test names and organization depend on the implementation. We can then commit the reviewed changes with Git.

If the agent cannot finish

A run that reaches its attempt limits stops and reports the remaining problems. Inspect the compiler output, failing tests, and session history before trying again:

```
perspt status
perspt ledger --recent
perspt replay <session-id>
```

Replay reads the recorded history without calling the model or rerunning the tools. For missing tool errors, confirm that the project's compiler and tests run locally. For interrupted work, `perspt resume --last` continues from saved progress after checking that the run is still authorized.

Run without the review prompt

Once the workflow is familiar, `--yes` automatically approves applying a successful result:

```
perspt agent --yes --output-summary result.json -w . \
  "Add a test for a number larger than u32::MAX to parse_count."
```

The required checks still run. An incomplete or failed run exits with a nonzero status. The JSON file records the outcome, session ID, and changed paths. See Headless Mode for scripts and CI examples.

Other languages and settings

The same command can work on Python and JavaScript/TypeScript projects when their required tools and dependencies are installed. Installation lists the prerequisites. For tasks that add dependencies, an explicit `--allow-dependency-mutation` grant is needed.

- Agent Mode: Review controls, test policies, and session management.
- Advanced Features: Model selection, attempt limits, and parallel tasks.
- Configuration Guide: Require historical tests or a separately maintained acceptance suite.
- SRBN Architecture: How the error score and acceptance rules work.

Headless Mode

Run Perspt's experimental SRBN agent without interactive prompts. This is designed for CI/CD pipelines, batch code generation, and automated workflows.

Overview

In headless mode (`--yes` flag), the agent auto-approves final promotion, skipping the interactive review modal. Combined with `--workdir`, it enables bounded autonomous project generation. It does not bypass capability grants, verification gates, budgets, the ledger, or workspace confinement.

```
perspt agent --yes -w /tmp/output "Create a Python ETL pipeline"
```

When to Use Headless Mode

- **CI/CD pipelines** - Generate boilerplate or scaffold projects in automation
- **Batch processing** - Run multiple agent tasks in sequence from a script
- **Rapid prototyping** - Skip review when iterating quickly

- **Testing the agent** - Validate agent behavior without manual intervention

When NOT to use headless mode:

- **Production codebases** - Always review changes before committing
- **Security-sensitive projects** - Manual review catches policy violations
- **Learning** - Interactive mode teaches you how SRBN works

Basic Headless Run

```
export GEMINI_API_KEY="your-key"

# Create a project autonomously
perspt agent --yes -w /tmp/my-project \
  "Create a Python data validation library using Pydantic.
   Include src layout, pyproject.toml, and pytest tests."
```

The agent will:

1. Detect language plugins
2. Plan the work graph — a single node by default; with `--max-parallel-nodes` above 1, one governed architect turn may decompose the task
3. Execute all nodes within their capability grants
4. Run verification (LSP + tests) on each node
5. Commit stable nodes to the ledger
6. Print a summary

Key Flags

Flag	Description
`--yes` / `-y`	Auto-approve final promotion (headless mode)
`-w, --workdir <DIR>`	Working directory for the project
`--max-turns <N>`	Maximum model turns per node (default 12)
`--max-calls-per-turn <N>`	Maximum model-issued and nested tool calls per turn (default 8)
`--rejection-budget <N>`	Shared non-descending and recovery budget (default 4)
`--output-summary <FILE>`	Write the terminal session summary as JSON

> **Headless requirements**
> A run without a terminal (CI, piped output) requires `--yes`: promotion approval cannot be prompted without a terminal, so the run fails fast instead of silently escalating. `--max-parallel-nodes` above 1 also requires `--yes`. A run that ends in escalation or incomplete work exits with a nonzero status, so CI never reads a stopped node as success.

Reading Structured Progress

In headless mode, Perspt prints its startup banner (task, workspace, detected domain), then streams ledger narration lines and per-node status changes as the run progresses. When the run finishes it prints the outcome, session id, turns used, ledger head hash, and any promoted paths.

For a machine-readable record, pass `--output-summary`:

```
persp agent --yes -w /tmp/out --output-summary summary.json \
  "Build a CLI tool in Rust that converts CSV to JSON"
```

The JSON summary contains the session id, node id, outcome, turns used, ledger head, and promoted paths.

Checking Session Status

After a headless run, inspect the results:

```
# Session status
persp status

# Recent ledger entries
persp ledger --recent

# Ledger statistics
persp ledger --stats

# Deterministic, credential-free replay of a session
persp replay <SESSION_ID>

# Resume a failed session
persp resume --last
```

Scripting Multiple Tasks

Run multiple agent tasks from a shell script:

```
#!/bin/bash
set -e
export GEMINI_API_KEY="your-key"

tasks=(
  "Create a Python CSV parser library"
  "Create a Rust JSON validator CLI"
  "Create a Python REST API with FastAPI"
)

for i in "${!tasks[@]}"; do
  dir="/tmp/project-$i"
  mkdir -p "$dir"
  persp agent --yes --max-turns 8 -w "$dir" "${tasks[$i]}"
  echo "=== Project $i complete ==="
done
```

Safety Recommendations

1. **Always set loop bounds** - `--max-turns` and `--rejection-budget` bound runaway sessions
2. **Use disposable directories** - Point `-w` to a fresh directory
3. **Review after generation** - Inspect the output before using it in production
4. **Audit with the ledger** - `persp replay <SESSION_ID>` reconstructs what the agent did, no logging flag required
5. **Check the exit status** - A nonzero exit means escalation or incomplete work, not verified success

See Also

- Agent Mode Tutorial - Interactive agent mode tutorial

- SRBN Architecture - SRBN technical details
- Agent Options Reference - Full agent CLI reference

Local Models with Ollama

Run Perspt with local models for privacy and offline usage.

Prerequisites

- Ollama installed
- Enough memory for the installed model and its context. Check the model's own requirements.

Setup

```
# Install Ollama (macOS)
brew install ollama

# Start the Ollama service
ollama serve

# Pull a model
ollama pull llama3.2
```

Using Ollama with Perspt

Perspt auto-detects Ollama if no cloud API keys are set:

```
# Unset any cloud keys
unset OPENAI_API_KEY ANTHROPIC_API_KEY GEMINI_API_KEY

# Launch Perspt - auto-detects Ollama
perspt

# Or specify a model explicitly
perspt chat --model ollama::llama3.2
```

Agent Mode with Local Models

```
perspt agent --model ollama::qwen2.5-coder -w ./my-project "Create a Python utility"
```

Agent mode needs tool-calling support. A model that works in chat may not support the agent's structured tool calls. Pull the selected model first, check that its route supports the required operations, and start with a small task. An explicit `ollama::` prefix keeps the local route selected even when cloud credentials are present.

```
ollama pull qwen2.5-coder
perspt providers --probe
```

Provider probes make live requests to configured routes. The example model IDs illustrate route syntax; installed models are listed by `ollama list`.

Custom Workflows

Integrate Perspt into scripts, CI/CD pipelines, and automation.

Simple CLI for Scripting

The `simple-chat` command provides line-oriented chat and optional logging. Its stdout includes session messages; use the agent JSON summary for machine-readable coding outcomes.

```
# Interactive
perspt simple-chat

# With session logging
perspt simple-chat --log-file session.txt
```

Batch Agent Runs

Run multiple agent tasks from a script:

```
#!/bin/bash
set -e
export GEMINI_API_KEY="your-key"

perspt agent --yes --max-turns 8 -w /tmp/proj1 "Create a Python CSV parser"
perspt agent --yes --max-turns 8 -w /tmp/proj2 "Create a Rust CLI calculator"
```

CI/CD Integration

Use headless mode in CI/CD pipelines:

```
# GitHub Actions example
name: Generate Boilerplate
on:
  workflow_dispatch:
    inputs:
      task:
        description: 'Task description'
        required: true

jobs:
  generate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Install Perspt
        run: cargo install --locked --path crates/perspt-cli --features bundled
      - name: Run Agent
        env:
          GEMINI_API_KEY: ${{ secrets.GEMINI_API_KEY }}
          PERSPT_TASK: ${{ inputs.task }}
        run: |
          perspt agent --yes --max-turns 12 \
            -w ./generated "$PERSPT_TASK"
      - name: Commit Results
        run: |
          git add generated/
          git commit -m "Add generated project"
```

Post-Run Analysis

After an agent run, use the management commands:

```
# Session status
perspt status

# Audit a session from the ledger
perspt replay <SESSION_ID>
perspt prompts explain-session --db-path <PATH> <SESSION_ID>
perspt context explain-turn --db-path <PATH> <SESSION_ID>

# Ledger history
perspt ledger --recent
perspt ledger --stats

# Resume incomplete sessions
perspt resume --last
```

Example: Python ETL Pipeline

This tutorial demonstrates building a complete Python ETL (Extract, Transform, Load) pipeline using Perspt's agent mode.

Task Description

We will ask the agent to create a data pipeline that:

- Reads CSV files with validation
- Transforms data using Pydantic models
- Handles missing values and edge cases
- Includes comprehensive pytest tests

Running the Agent

Initialize a `uv` project first so `pyproject.toml` exists before the session starts:

```
export GEMINI_API_KEY="your-key"
uv init --lib /tmp/etl-demo

perspt agent --yes --allow-dependency-mutation -w /tmp/etl-demo \
  --actuator-model gemini::gemini-3.1-pro \
  --explorer-model gemini::gemini-3.5-flash \
  "Create a Python ETL pipeline package. It should:
   1. Read CSV files into Pydantic models for validation
   2. Transform records (clean nulls, normalize strings, compute derived fields)
   3. Write validated records to a new CSV
   4. Include a CLI entry point in src/main.py
   5. Use uv for dependency management with pandas and pydantic
   6. Include comprehensive pytest tests for each module"
```

Example output layout

A successful run may produce a layout like this. The exact files and test count depend on the generated implementation:

```
/tmp/etl-demo/
+-- pyproject.toml          # [build-system] with uv_build backend
+-- uv.lock
+-- src/
|   +-- etl_pipeline/
|   |   +-- __init__.py
|   |   +-- core.py         # Main pipeline logic
|   |   +-- validator.py    # Pydantic models
|   |   +-- transformer.py  # Data transformations
|   +-- main.py             # CLI entry point
+-- tests/
    +-- test_core.py
    +-- test_validator.py
    +-- test_transformer.py
```

Verification

The engine verifies every checkpoint with real sensors: `ty check` (or `python -m compileall` / `pyright` fallback) and offline `pytest` measure the energy of each candidate, and a checkpoint is accepted on a hard pass or sufficient measured descent. Only a hard-passing result can be delivered. When the run finishes, the terminal summary reports the outcome, the session id, turns used, the ledger head hash, and the promoted paths.

After completion, verify manually:

```
cd /tmp/etl-demo
uv run pytest -v
```

Key Observations

- Initializing with `uv init --lib` creates a `src/` layout with `[build-system]` in `pyproject.toml`, and `--allow-dependency-mutation` lets the agent run governed `uv add` commands for `pandas`, `pydantic`, and `pytest`
- The dispatcher prevents nodes with conflicting write footprints from running concurrently
- `ty check` (with `compileall` / `pyright` fallback) checks syntax and types (`V_syn`)
- `uv run --no-sync pytest` runs the test suite offline inside the sandbox (`V_log`)
- Work is promoted only after the run's final checkpoint passes every hard predicate (a `HardPass` outcome)

See Also

- Agent Mode Tutorial - Agent mode tutorial
- Example: Rust CLI Tool - Rust project example

Example: Rust CLI Tool

Build a Rust command-line tool using Perspt's agent mode.

Task Description

We will ask the agent to create a CLI tool that converts CSV files to JSON.

Running the Agent

Initialize a Cargo project first so the `rust` plugin and offline verifier sensors have a manifest to inspect:

```
export GEMINI_API_KEY="your-key"
cargo init /tmp/csv2json

perspt agent --yes --allow-dependency-mutation -w /tmp/csv2json \
  --actuator-model gemini::gemini-3.1-pro \
  --explorer-model gemini::gemini-3.5-flash \
  "Create a Rust CLI tool called csv2json that:
   1. Reads a CSV file from a path argument
   2. Converts each row to a JSON object
   3. Outputs JSON array to stdout or a file (--output flag)
   4. Handles errors gracefully with anyhow
   5. Uses clap for argument parsing
   6. Includes unit tests and integration tests"
```

Example output layout

```
/tmp/csv2json/
+-- Cargo.toml
+-- src/
|   +-- main.rs        # Entry point, clap args
|   +-- lib.rs          # Core conversion logic
|   +-- converter.rs    # CSV -> JSON conversion
+-- tests/
    +-- integration.rs  # Integration tests
```

Verification

The agent uses the `rust` plugin's verifier profile inside the sandboxed verifier:

- **Syntax & Build (V_syn)**: `cargo check --message-format=json` and `cargo build --message-format=json`
- **Tests (V_log)**: `cargo test`
- **Lint (advisory)**: `cargo clippy -- -D warnings` (steers repairs without blocking a hard pass)
- **Sensor readiness (V_boot)**: tracks whether required toolchain sensors were available and runnable

```
cd /tmp/csv2json
cargo test
cargo run -- --help
```

Key Observations

- The `rust` plugin runs `cargo check`, `cargo build`, and `cargo test` as required hard-gate stages against the candidate overlay
- `--allow-dependency-mutation` permits governed `cargo add` calls when the task needs crates such as `clap` or `anyhow`
- `cargo test` runs both unit and integration tests (`V_log`)
- The dispatcher prevents nodes with conflicting write footprints from running concurrently

See Also

- Agent Mode Tutorial - Agent mode tutorial
- Example: Python ETL Pipeline - Python project example

Example: Scientific Computing

This tutorial demonstrates using Perspt's headless agent mode for scientific computing tasks: CFD simulation setup and wind tunnel data analysis.

> **Experimental**
> These examples demonstrate advanced prompt patterns for scientific computing. Results depend on model capability and may require iteration. Treat them as starting points rather than guaranteed production workflows.

CFD Simulation Setup (Python)

Create a computational fluid dynamics simulation scaffolding:

```
export GEMINI_API_KEY="your-key"

perspt agent --yes --allow-dependency-mutation -w /tmp/cfd-sim \
  --actuator-model gemini-3.1-pro \
  --explorer-model gemini-3.5-flash \
  "Create a Python CFD simulation package that:
   1. Defines a 2D grid mesh using numpy arrays
   2. Implements a simple Navier-Stokes solver (lid-driven cavity)
   3. Uses finite difference method for spatial discretization
   4. Includes a Jacobi iterative pressure solver
   5. Outputs velocity and pressure fields as numpy arrays
   6. Includes visualization with matplotlib (contour plots)
   7. Has pytest tests validating conservation laws
   8. Uses Pydantic for simulation parameters (Re, dt, grid_size)"
```

Expected output structure:

```
/tmp/cfd-sim/
+-- pyproject.toml
+-- src/
|   +-- cfd_sim/
|   |   +-- __init__.py
|   |   +-- mesh.py          # Grid generation
|   |   +-- solver.py        # Navier-Stokes solver
|   |   +-- pressure.py      # Jacobi pressure solver
|   |   +-- visualization.py # matplotlib plotting
|   |   +-- parameters.py    # Pydantic config
|   +-- main.py
+-- tests/
    +-- test_mesh.py
    +-- test_solver.py
```

Wind Tunnel Data Analysis (Python)

Analyze experimental wind tunnel data:

```
perspt agent --yes --allow-dependency-mutation -w /tmp/wind-tunnel \
  --actuator-model gemini-3.1-pro \
  --explorer-model gemini-3.5-flash \
  "Create a Python wind tunnel data analysis package that:
   1. Reads pressure tap data from CSV files
   2. Computes lift and drag coefficients (Cl, Cd) from pressure distributions
```

```
   3. Implements trapezoidal integration for force computation
   4. Generates polar plots of pressure coefficient (Cp) distribution
   5. Supports multiple angle-of-attack sweeps
   6. Includes Pydantic models for AirfoilData and TestConditions
   7. Exports results as JSON and matplotlib figures
   8. Has pytest tests with known NACA 0012 reference data"
```

Finite Element Solver (Rust)

Build a simple FEM solver in Rust:

```
perspt agent --yes --allow-dependency-mutation -w /tmp/fem-solver \
  --actuator-model gemini-3.1-pro \
  --explorer-model gemini-3.5-flash \
  "Create a Rust finite element package that:
   1. Defines 2D triangular mesh elements
   2. Assembles global stiffness matrix for heat conduction
   3. Applies Dirichlet boundary conditions
   4. Solves using Gauss elimination (nalgebra)
   5. Outputs nodal temperatures
   6. Includes tests with known analytical solutions"
```

Tips for Scientific Tasks

1. **Be specific about algorithms** - Name the numerical method, discretization scheme, and solver type in the prompt
2. **Include validation criteria** - Reference known analytical solutions or benchmark data for tests
3. **Set numerical tolerances in the tests** - The descent margin controls checkpoint progress, not the accuracy required by a scientific assertion.
4. **Raise the loop bounds** - Scientific code often requires more iterations for convergence: `--max-turns 20 --rejection-budget 8`
5. **Review carefully** - Numerical correctness requires domain expertise beyond what the LSP can verify

See Also

- Headless Mode - Headless mode guide
- Agent Mode Tutorial - Interactive agent mode

Monitoring Agent Execution with the Dashboard

This tutorial walks through using the Perspt web dashboard to observe an agent session in real time.

Prerequisites

- Perspt installed (see Installation)
- A project directory to work in

Step 1: Start an agent with its dashboard

```
perspt agent --dashboard -w ./myproject "Fix the failing parser test"
```

Leave this terminal running. The embedded dashboard shares the agent's store connection; it stops when the agent exits.

Step 2: Open the local dashboard

Open `http://127.0.0.1:3000` in a browser. After the agent exits, `perspt dashboard` can reopen historical sessions. A standalone dashboard process can conflict with an active writer's database lock, so we use the embedded option during a run.

Step 3: Open the Overview

Navigate to `http://localhost:3000` in your browser. The Overview page shows your active session with:

- **Status badge** - "running" while the agent works
- **Node count** - stable/total nodes projected from the session's latest graph revision
- **Events** - how many ledger events the session has recorded

Step 4: Open the Session Detail

Click your session to open `/sessions/{id}`. The detail page shows the task, working directory, status, and detected toolchain, plus node totals, event and measurement counts, the last and average energy, and a per-node summary table.

Step 5: Explore the Topology

Click **Topology** (`/sessions/{id}/topology`). The page shows the work graph as the ledger recorded it:

- **Revision lineage** - every graph revision, newest marked "latest"
- **Nodes of the latest revision** - each node with its state badge (stable, running, stopped, blocked, retired)
- **Edge table** - parent -> child relationships

Step 6: Watch Energy Convergence

Click **Energy** to see the trajectory of measured candidates, in ledger order: one row per measurement with its node, generation, energy, and whether it was a hard pass. The summary bar shows the count, average, minimum, and maximum energy and how many measurements passed hard. Watch values decrease as accepted checkpoints descend.

Step 7: Check the Backlog

Click **Backlog** for the conditional-capacity diagnostics: node state counts, how many nodes sit in the backlog (and how many of those are still unmeasured), the backlog potential $\Phi(W)$, its drift, and a per-node energy table.

Step 8: Understand Decisions

The **Decisions** page is the raw PSP-9 event trace: a flat, Merkle-chained table with one row per ledger record — its sequence number, event kind, a truncated payload summary, and the short hash that chains it to its predecessor. This is the same record that `perspt replay` reconstructs.

Step 9: Review Governance

The **Governance** page shows the session's authority epoch and grant status, recent calibration epochs, adjudication verdicts, and delayed audit samples still pending a label.

Step 10: Monitor Live Updates

The dashboard receives Server-Sent Events (SSE) from the server every 2 seconds. You can keep the browser open and watch as the agent progresses through its task.

When the agent completes, the session status changes to “completed” (green badge) on the Overview page.

User Guide

In-depth coverage of every Perspt feature.

Guide	Coverage
Basic Usage	TUI navigation, chat commands, and session management.
Agent Mode	Governed tool loop, work graph, measured verification, and review.
Advanced Features	Per-role model routing, acceptance gate, budgets, ledger, and resume.
Providers	Configure OpenAI, Anthropic, Gemini, Ollama, and more.
Model Context Protocol (MCP)	Connect governed external tools to agent mode and the chat TUI.
Troubleshooting	Diagnose and fix common issues.
Dashboard	Real-time web monitoring of agent sessions.

Basic Usage

Choose full-screen chat, plain terminal chat, or a coding task. Chat modes support conversation; agent mode edits a candidate copy and runs project checks.

Launching the TUI

```
# Auto-detect provider from env keys
perspt

# Explicit model (provider comes from config or env detection)
perspt chat --model claude-fable

# Use a specific config file
perspt --config ./config.toml chat --model gpt-5.5
```

The TUI provides:

- Markdown rendering (code blocks, headers, lists, bold, italic)
- Premium ASCII table rendering with multi-line cell wrapping
- Inline LaTeX math transpilation to styled Unicode equations (bold and cyan)
- Real-time response streaming
- Scroll navigation
- Status bar (provider, model, streaming indicator, and reasoning toggles)

Keyboard Shortcuts

Key	Action
Enter	Send message
Ctrl+J	Insert newline in input

Key	Action
Page Up / Down	Scroll chat history by 10 lines
Shift+Up / Down	Scroll chat history by 1 line
Ctrl+Up / Down	Scroll chat history by 1 line
Home / End	Go to start / end of input line
Ctrl+C / **Ctrl+Q**	Quit
Ctrl+R	Toggle inner reasoning process display
Ctrl+A / **Ctrl+E**	Go to start / end of input line
Ctrl+B / **Ctrl+F**	Move cursor left / right
Ctrl+D / **Ctrl+H**	Delete / Backspace character
Ctrl+K / **Ctrl+U**	Kill to end / start of line
Ctrl+W	Delete word before cursor

Slash Commands (Directives)

Both CLI and TUI modes support slash commands typed directly in the input box:

Command	Description
`/exit` or `/quit`	Exit the application session
`/clear`	Reset the active conversation history
`/model <name>`	Switch the active LLM model on the fly
`/save <path>` (TUI only)	Export full conversation history to a markdown file
`/mcp` (TUI only)	Show MCP configuration state, discovery failures, policy rejections, and admitted operation names without contacting the model
`/mcp accept [JSON]`, `decline`, or `cancel` (TUI only)	Answer a pending MCP elicitation request
`/help`	Print the menu of available slash commands

The chat TUI can use latest MCP tools, resources, prompts, completions, sampling, elicitation, subscriptions, and tasks from any `[[external_tools]]` server entry in `config.toml` with `modes = ["chat"]` or `modes = ["agent", "chat"]` is considered for read-only admission. Every allowed remote tool also needs an exact local policy table. Tool activity is shown as a concise transient status only while a call is active. MCP protocol events, arguments, results, discovery notices, and subscription traces do not become conversation or reasoning content. The selected model chooses an admitted tool automatically when needed; `/mcp` is the explicit diagnostic surface. Tool results are labeled as untrusted content before returning to the model. **Ctrl+R** continues to control genuine model reasoning, including tool-aware turns. Unicode and multiline terminal clipboard paste is inserted at the input cursor. See Model Context Protocol (MCP) for configuration and the security boundary.

Simple CLI Mode

For line-oriented chat with optional session logging:

```
perspt simple-chat
perspt simple-chat --model gpt-4o-mini
perspt simple-chat --log-file session.txt
```

Type your message, press Enter. Responses stream to stdout. Type `exit` or press `Ctrl+D` to quit.

Piped input is line-oriented. Each non-empty line becomes a separate message. The command takes no positional prompt, and stdout includes a banner, input echoes, and session messages as well as responses. It does not combine a multiline file into a single prompt or provide a response-only output mode.

Autonomous Agent Mode

For complex software engineering and coding jobs that require self-stabilizing, multi-file edits:

```
perspt agent "Implement a parsing logic for custom CSV formats in src/csv.rs"
```

The default run uses one work node. The agent edits a candidate copy, measures project checks, and requests promotion approval for a deliverable result. See Agent Mode for review, verification policies, and optional parallel dispatch.

Provider Auto-Detection

Perspt checks environment variables in this priority order:

Env Variable	Provider	Default Model
`VERTEX_PROJECT_ID`	Vertex AI	`vertex::gemini-2.5-flash`
`GEMINI_API_KEY`	Gemini	`gemini-3.1-flash-lite-preview`
`OPENAI_API_KEY`	OpenAI	`gpt-4o-mini`
`ANTHROPIC_API_KEY`	Anthropic	`claude-3-5-sonnet-20241022`
`GROQ_API_KEY`	Groq	`llama-3.1-8b-instant`
`COHERE_API_KEY`	Cohere	`command-r-plus`
`XAI_API_KEY`	xAI	`grok-beta`
`DEEPSEEK_API_KEY`	DeepSeek	`deepseek-chat`
(none)	Ollama (local)	`llama3.2`

See Providers for full provider details.

Agent Mode

Give Perspt a coding task and a project directory. The agent edits a candidate copy, runs detected project checks, and requests approval before promotion. Agent mode is experimental. A hard pass means the required checks passed; a run can also stop with unresolved errors. The runtime contracts are described in Stability and Measured Progress.

Launching Agent Mode

```
perspt agent -w <DIR> "<task>"

# Examples
perspt agent -w ./my-project "Create a Python REST API"
perspt agent -y -w /tmp/demo "Build a Rust CLI tool"
```

A run without a terminal (CI, pipes) requires `--yes`: promotion approval cannot be prompted without one, so the CLI fails fast instead of silently escalating a fully verified run. Any outcome other than a hard pass exits with a nonzero process status, so scripts never read a stopped node as success.

How a task runs

1. **Read the project.** Perspt explores the files and starts a task. The default run works on one task at a time; optional parallel execution can split larger work into several tasks.
2. **Edit a separate copy.** The agent reads and changes files in a temporary project copy. Permission and command rules limit what it can do.
3. **Build and test.** Perspt runs the detected language tools. The required checks run before each decision to keep an attempt; smaller syntax checks also run during editing.
4. **Try corrections.** Build errors and failing tests tell the agent what went wrong. While checks still fail, an attempt must improve the recorded error score enough to be kept. Otherwise, Perspt restores its best saved version.
5. **Review and apply.** When the required checks pass, an interactive run asks for approval before copying edits into the project. A run that cannot finish within its limits stops and reports the remaining errors.

Applying edits does not create a Git commit. We review the code and tests before committing, because passing checks can miss bugs. The internal names for the temporary copy, progress decisions, and file copying are `CandidateWorkspace`, the acceptance gate, and promotion. SRBN Architecture explains these implementation details.

Test Evidence Policies

Tests are part of the project being developed, not an automatically immutable artifact. The coding gate always runs the resulting project suite. The `[verification] test_policy` setting controls whether another test-evidence view is also required:

Policy	When to use it
`evolving` (default)	Feature work, refactoring, API migrations, and corrected behavior where implementation and tests legitimately evolve together. New tests run; changed or obsolete historical expectations are not resurrected.
`backward-compatible`	Additive changes and bug fixes that promise recognized pre-existing test files will continue to pass. The resulting suite and the restored regression view must both pass.
`external-oracle`	Work with independently maintained acceptance evidence. Perspt copies the configured overlay onto a private candidate view and runs its command as an additional blocking test stage. Put withheld material outside the workspace.

The first policy says the project's configured suite passed for its resulting contract; it does not turn tests authored by the actuator into independent proof. The external policy supplies that independence when the user, CI, or a release process owns the acceptance suite. Configuration and complete examples are in Configuration Guide.

Windows Process Isolation

On native Windows, TUI chat and `simple-chat` require no special option. Coding remains strict by default and refuses to start verifier or inspection processes because Perspt does not yet implement a restricted-token/ACL sandbox. Use `--allow-unisolated` for a single acknowledged run, or configure `[verification] allow_unisolated = true` when durable resume is required. The candidate overlay, deterministic admission kernel, measurement gates, ledger, and promotion approval remain active, but child processes can access anything the host user can access. WSL provides the Linux sandbox path.

Multi-Node Dispatch

By default the work graph executes one node at a time. `--max-parallel-nodes` raises the number of concurrent nodes; values above 1 require `--yes` because promotion approval cannot be prompted per node while other nodes run. The dispatcher schedules by footprint conflict — two nodes whose declared file footprints overlap never run concurrently — and each node writes into content-addressed staging. Conflict detection is dependency-aware (downstream refinements win by edge precedence), and one global integration gate measures the merged result before anything is promoted.

Node Classes

Each work-graph node carries a class describing the kind of work it performs:

Class	Description
Explore	Read-only investigation of the workspace.
Plan	Task decomposition and graph revision.
Implement	Code changes toward the node goal.
Verify	Measurement-only verification work.
Test	Test authoring and execution.
Integrate	Merging staged work across nodes.
Repair	Recovery from a failed or rejected node.
Interface	Public API definitions, type signatures, traits.

The planner currently emits `Implement` nodes; the full taxonomy is the SDK contract that dispatch and the ledger record.

Review Modal (Interactive)

For a single-node run without `--yes`, the review modal presents the affected files, the verification gate results (syntax, build, tests, lint, plus test pass/fail counts), the scalar energy `V(x)`, and a degraded flag listing any sensors that could not run.

- **y** - Approve and promote
- **n** - Reject and restore the best checkpoint

- **c** - Send feedback for correction
- **e** - Open files in your editor
- **d** - Toggle full diff view
- **s** - Skip this review

`Left`/`Right` move the button selection, `Enter` confirms the selected decision, and `Esc` dismisses the modal.

Session Management

```
# Check session state (ledger, measurements, energy, gate, denials)
perspt status

# Abort the current session by revoking its authority epoch
perspt abort

# Resume the most recent session
perspt resume --last
```

`perspt status` prints the ledger event count and head, measurement and denial counts, the last energy and gate decision, search forest, branch, and no-good counts, adjudicator verdicts, the capacity $\Phi(W)$, and validator independence statistics. `perspt resume` prints the session id, task, working directory, and status, then verifies the ledger chain before continuing; live authority is never serialized, so resume always mints fresh capabilities.

Domains

`--domain` selects the domain package (`coding`, `research`); by default the best-matching domain is detected from the workspace, with `coding` as the fallback.

Exploration-Only Runs

`--exploration-only` runs only the read-only exploration phase: a deterministic repository map plus an interactive explorer tool loop. Nothing is mutated or promoted, and the run never prompts, so it works without `--yes` even in non-interactive contexts.

Headless Summary

In headless mode the agent prints the terminal summary lines — outcome, session id, turns used, ledger head, and promoted paths:

```
Outcome: HardPass
Session: 019820f3-...
Turns: 6
Ledger head: 4f2c9a1b8d7e6a50
Promoted paths: src/api.py, tests/test_api.py
```

`--output-summary <FILE>` writes the same summary as JSON for CI integration.

Dashboard Monitoring

After a session finishes, `perspt dashboard` opens its database read-only for inspection. A separate dashboard process can conflict with an active writer's DuckDB lock. For live monitoring, use the embedded option below. See Dashboard for the page map and interpretation.

Live Dashboard Monitoring

Use `--dashboard` to start the web monitoring dashboard alongside the agent:

```
perspt agent --dashboard "Build a REST server"
# Open http://127.0.0.1:3000 in a browser
```

The embedded dashboard reads through the same database connection the agent writes to, providing real-time views of work-graph topology, energy convergence, and the ledgered decision trace. Use `--dashboard-port` to change the port. The server stops when the agent exits.

> **Note**
> The embedded dashboard runs unauthenticated on `127.0.0.1`. It is intended for local monitoring only; do not expose the port beyond the local machine.

See Advanced Features for per-role model routing, the acceptance gate, budgets, and search configuration, and Agent Options Reference for the full flag reference.

Advanced Features

Per-Role Model Routing

The runtime routes each role through the `[models]` table of the config file. Values are fully qualified `provider::model` routes so identity, calibration, and replay never depend on an ambient default provider:

```
[models]
architect = "vertex::gemini-3.1-pro"
actuator = "vertex::gemini-3.5-flash"
verifier = "vertex::gemini-3.1-pro"
speculator = "vertex::gemini-3.5-flash-lite"
adjudicator = "vertex::gemini-3.1-pro"
```

The `speculator` entry is consumed only as the fallback route for the explorer role. On the command line, three roles can be overridden per run:

```
perspt agent \
  --actuator-model vertex::gemini-3.5-flash \
  --explorer-model vertex::gemini-3.5-flash-lite \
  --adjudicator-model vertex::gemini-3.1-pro \
  -w ./project "Task description"
```

`--model` is an alias for `--actuator-model`. `--fallback-model` adds an ordered sticky failover route for the actuator; repeat the option to add more routes. Role-specific models are never silently reused as actuator fallbacks:

```
perspt agent \
  --actuator-model vertex::gemini-3.5-flash-lite \
  --fallback-model vertex::gemini-3.5-flash \
  --fallback-model vertex::gemini-3.1-pro \
  -w ./project "Task"
```

The adjudicator, when configured, is a no-tool validator of the realized diff. Its verdict is an uncalibrated conjunctive veto, never an energy measurement.

The Acceptance Gate

A candidate is accepted when it is a hard pass — every sensor clean — or when it achieves the required measured energy descent

$$V(y) \leq V(\text{best}) - \rho_{\text{gate}}.$$

The energy is measured on the realized filesystem by the language plugins (compilers, tests, linters, LSP diagnostics), never estimated by a model. `--rho-gate` sets the required descent per accepted checkpoint:

```
# Default: rho = 0.5
perspt agent --rho-gate 0.5 -w . "Precise task"

# Accept smaller measured improvements
perspt agent --rho-gate 0.25 -w . "Incremental refactor"
```

A rejected candidate restores the best accepted checkpoint, so the workspace never regresses past a measured state.

Budgets

The loop is finite by construction. Each budget is a hard bound, and exhausting one escalates with a residual certificate rather than looping:

Flag	Description
`--max-turns <N>`	Maximum model turns per node (default 12).
`--max-calls-per-turn <N>`	Maximum direct and nested tool calls per turn (default 8).
`--rejection-budget <N>`	Shared non-descending and recovery budget (default 4).
`--max-parallel <N>`	Maximum independent compiler, test, and lint sensors (default 4).
`--max-parallel-nodes <N>`	Concurrent work-graph nodes (default 1; above 1 requires `--yes`).

Bounded Search

On a gate failure the runtime can open a bounded search forest instead of retrying blindly. Branches are sequential and eager, with a hard cap of three identities per forest; every dead end is recorded as an exact-keyed no-good so it is never re-proposed, every branch takes pre-action reservations that settle against observed actuals, and exactly one candidate is committed through the ordinary acceptance gate. The `[exploration]` config block bounds the search:

```
[exploration]
initial_branches = 1     # branches opened before any expansion trigger
max_branches = 3         # branch identities per forest (hard cap 3)
distinct_family = true   # prefer a distinct model family on expansion
max_workspace_files = 2000
max_workspace_bytes = 50000000
```

`distinct_family` is a prior, never a certificate. The workspace caps bound the cumulative eager-copy file and byte reservations. Search activity appears in `perspt status` as forest, branch, committed, and no-good counts.

Prompt Programs

Every model call is compiled from typed prompt sections into a per-call prompt program. The program's route, dialect, section provenance, and tool-surface hash are ledgered, so a session's exact prompts can be audited after the fact:

```
# List every compiled section: id, version, stage, role, hash
perspt prompts list

# Show the programs a session actually compiled, with digests
perspt prompts explain-session --db-path <FILE> <SESSION_ID>
```

`--allow-experimental-prompts` substitutes validated `[prompts]` bundle sections live. This is Gate AE and remains experimental until a change record passes paired evaluation.

Resident Context

The conversation the model sees is paged: content is stored as content-addressed pages, and the transported conversation carries tombstones in place of folded content. The composed request is checked against the route's input allowance and dialect byte limit before any call, and the model recalls folded pages only through the governed `context_recall` tool. Compactions and refusals are recorded in the ledger:

```
# Show a session's recorded context events (compactions, refusals)
perspt context explain-turn --db-path <FILE> <SESSION_ID>
```

Merkle Ledger

Every stable node is committed to a content-addressed Merkle ledger stored in DuckDB. This provides:

- **Auditability** - Full trace of what each node produced
- **Rollback** - Undo a session's newest completed promotion, unless its files changed afterward
- **Resume** - Continue interrupted sessions with verified context

```
perspt ledger --recent
perspt ledger --stats
```

Run Summaries

`--output-summary <FILE>` writes the terminal run summary as JSON (session id, node id, outcome, turns used, ledger head, promoted paths) for CI integration. Any outcome other than a hard pass exits with a nonzero process status:

```
perspt agent -y --output-summary summary.json -w . "Task"
cat summary.json
```

Resume and Replay

```
# Resume the most recent session
perspt resume --last

# Credential-free audit replay of a session
perspt replay <SESSION_ID>
```

Resume verifies the ledger chain, rebuilds staging by folding the ledger, and re-enters dispatch. Live authority is intentionally never serialized: resume always rechecks the durable authority epoch and mints fresh capabilities. `--persistent-grants` on the original run stores signed grant intent so an unattended resume can complete bracketed promotion intents.

Exploration-Only Runs

`--exploration-only` runs only the read-only exploration phase: a deterministic repository map plus an interactive explorer tool loop. Nothing is mutated or promoted:

```
perspt agent --exploration-only -w . "Map the request handling path"
```

Governed Dependency Mutation

Dependency changes (`cargo add`, `uv add`, `npm install`) are an external effect and are denied by default. `--allow-dependency-mutation` grants them, still subject to the kernel and the ledger:

```
perspt agent --allow-dependency-mutation -w . "Add pandas and use it"
```

Providers

Use an explicit model route when several providers are configured. A route has the form `provider::model`. The provider prefix selects an adapter and the model part is passed through to it. A bare model name uses the configured or detected provider.

Supported adapters

Provider	Prefix	Credential or connection
OpenAI and compatible servers	`openai`	`OPENAI_API_KEY`; optional `OPENAI_BASE_URL`.
Anthropic	`anthropic`	`ANTHROPIC_API_KEY`.
Google Gemini	`gemini` or `google`	`GEMINI_API_KEY`.
Google Vertex AI	`vertex`	`VERTEX_PROJECT_ID` and Application Default Credentials; optional `VERTEX_LOCATION`.
Groq	`groq`	`GROQ_API_KEY`.
Cohere	`cohere`	`COHERE_API_KEY`.
xAI	`xai`	`XAI_API_KEY`.
DeepSeek	`deepseek`	`DEEPSEEK_API_KEY`.
Ollama	`ollama`	Local server; no API key. Optional `OLLAMA_BASE_URL`.

Model access and tool-calling support depend on the account, server, and model. Perspt's adapter list is not a catalog of available models. `perspt providers` shows configured routes; `perspt providers --probe` makes live requests to test their behavior.

Choose a route

Replace `<model-id>` with an identifier available through the selected provider:

```
export OPENAI_API_KEY="<api-key>"
perspt chat --model 'openai::<model-id>'
```

For local chat, select an installed model:

```
ollama list
perspt chat --model ollama::llama3.2
```

The local ID here is an example, not a requirement. See Local Models with Ollama for setup.

Persist the default

The configuration file can set a provider and model:

```
provider = "openai"
model = "<model-id>"
```

A CLI `--model` override takes precedence for that run. Agent roles can use different fully qualified routes through `[providers.<id>]` and `[models]`. See Set Up Providers for setup and Advanced Features for role routing.

Vertex AI

The Vertex adapter resolves ADC tokens per request unless `VERTEX_API_KEY` supplies an explicit bearer-token override. `VERTEX_LOCATION` defaults to `global`.

```
gcloud auth application-default login
export VERTEX_PROJECT_ID="<project-id>"
export VERTEX_LOCATION="global"
perspt chat --model 'vertex::<model-id>'
```

OpenAI-compatible endpoints

A server must support the OpenAI request format expected by the adapter:

```
export OPENAI_API_KEY="<server-key>"
export OPENAI_BASE_URL="http://localhost:8000/v1"
perspt chat --model 'openai::<served-model-id>'
```

A base URL override alone does not establish compatibility with a provider-specific authentication or deployment protocol.

Model Context Protocol (MCP)

Perspt is an optional MCP client for the governed coding agent and the interactive chat TUI. The implementation uses the official Rust MCP SDK and speaks **MCP 2026-07-28 only**. It starts with `server/discover` and does not fall back to `initialize` or an older protocol version. A server that does not advertise 2026-07-28 is rejected.

The client supports stdio and stateless Streamable HTTP, including:

- tools, resources, resource templates, prompts, and completions;
- explicit roots disclosure;

- opt-in model sampling and user elicitation;
- progress, logging, cancellation, catalog-change, resource-update, task, and custom notifications;
- `subscriptions/listen` with list-change and resource filters;
- multi-round tool results and the tasks extension for long-running calls.

MCP is absent when no `[[external_tools]]` table is configured. It does not replace Perspt's coding-domain tools, planner, verifier, candidate workspace, or acceptance gate.

Authority and Trust

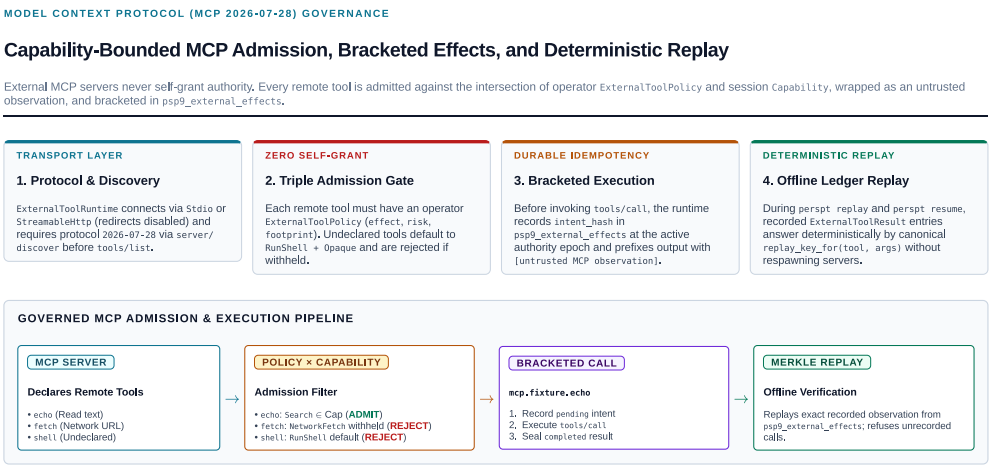


Fig. 3 MCP 2026-07-28 strict discovery handshake, local policy admission, and write-ahead bracketed tool execution.

A server description, schema, prompt, resource, notification, or result is untrusted input. A server cannot grant itself authority. Each remote tool must match a user-owned policy assigning its effect, risk, and scheduling footprint. Undeclared tools fail closed. Agent calls still pass the capability kernel and are write-ahead bracketed; replay uses the recorded result and never reconnects.

Resources and prompts are exposed through namespaced, locally defined read-only operations only when the server advertises the capability:

```
mcp.<server>._perspt_resources_list
mcp.<server>._perspt_resource_templates_list
mcp.<server>._perspt_resource_read
mcp.<server>._perspt_prompts_list
mcp.<server>._perspt_prompt_get
mcp.<server>._perspt_complete
```

Their output is returned as untrusted data and is never silently inserted into the system prompt. Remote names beginning with `_perspt_` are reserved and rejected, preventing collisions with these client operations. `_perspt_prompt_get` accepts optional, equally sized `argument_names` and `argument_values` string arrays; entries at the same index form one MCP prompt argument. Each array is capped at 64 entries.

Quick Setup

This stdio example enables independent agent and chat lifecycles, discloses one root, permits sampling and elicitation, and subscribes to one resource:

```
[[external_tools]]
id = "docs"
transport = "stdio"
command = ["company-docs-mcp", "--stdio"]
modes = ["agent", "chat"]
timeout_ms = 30000
max_result_bytes = 1048576
max_stderr_bytes = 65536
max_task_wait_ms = 300000
env_from_env = { DOCS_TOKEN = "COMPANY_DOCS_TOKEN" }

roots = [{ uri = "file:///absolute/path/to/project", name = "Project" }]
sampling = true
max_sampling_tokens = 4096
elicitation = true
subscriptions = true
resource_subscriptions = ["file:///absolute/path/to/project/guide.md"]
tasks = true

[external_tools.tools.search]
effect = "search"
risk = "low"
footprint = { selectors = [{ kind = "scoped_argument", family = "company-docs", field =
"query", access = "read" }] }
```

The policy key must exactly match the remote name. A footprint field must exist in the advertised input schema. After admission the model-facing tool is `mcp.docs.search`.

For Streamable HTTP use these transport fields:

```
[[external_tools]]
id = "records"
transport = "streamable_http"
url = "https://tools.example.test/mcp"
modes = ["agent", "chat"]
headers_from_env = { Authorization = "RECORDS_AUTHORIZATION" }

[external_tools.tools.lookup]
effect = "data_read"
risk = "low"
footprint = { selectors = [{ kind = "scoped_argument", family = "records", field = "id",
access = "read" }] }
```

Set `RECORDS_AUTHORIZATION` to the complete server value, for example `Bearer ...`. Non-loopback endpoints require HTTPS and redirects are disabled. Streamable HTTP is stateless under 2026-07-28; Perspt does not recover by negotiating a legacy session.

Chat TUI

Run `perspt --config ./config.toml chat` and type `/mcp`. It reports discovery failures, policy rejections, and admitted operations. `/help` includes MCP and elicitation commands. The model calls tools automatically; `/mcp` performs no remote call.

Normal chat starts without discovery or policy messages in the conversation. When the model selects an admitted operation, the TUI shows one transient, human-readable activity label and then the final answer. Raw MCP lifecycle events, tool arguments, results, subscription notifica-

tions, and server logs stay out of both the conversation and the reasoning panel. Use `/mcp` when you explicitly want discovery and admission diagnostics. **Ctrl+R** remains the reasoning control and displays only reasoning emitted by the selected model; tool-aware Qwen turns use the same reasoning stream.

Chat has a fixed read-only authority ceiling. Mutating, shell, network-fetch, dependency, graph, and policy effects are rejected even when the same tool is admissible in the governed agent. `perspt simple-chat` intentionally creates no MCP lifecycle.

For form or URL elicitation the input remains active and the TUI shows the request as JSON. Respond explicitly with:

```
/mcp accept {"field":"value"}
/mcp accept
/mcp decline
/mcp cancel
```

Perspt does not open an elicitation URL automatically. Form responses are JSON objects. Perspt advertises client-side schema validation as disabled, leaving the server responsible for validating the requested schema.

Terminal bracketed paste is enabled in both chat input paths. Normal terminal paste shortcuts (Command-V, Ctrl-Shift-V, or Shift-Insert, depending on the terminal) insert Unicode and multiline clipboard text at the cursor. The mode is disabled again on TUI exit.

Agent Behavior

Agent servers are discovered when a session assembles its node catalog. External schemas use the ordinary deferred tool-search path. Proposals, arguments, effects, grants, execution, results, and uncertain completion are governed like the rest of the coding tool plane.

Sampling uses the already selected local route. MCP model preferences cannot redirect credentials or choose an unconfigured provider. Perspt forwards `maxTokens`, temperature, stop sequences, tools, and tool choice. It advertises tool-aware sampling but not `includeContext`; requests for implicit MCP-server context are rejected.

The command-line agent has no interactive MCP form surface, so an enabled elicitation request receives `decline` immediately. Chat is interactive. SDK hosts may install their own `McpElicitationProvider`.

Subscriptions and Dynamic Catalogs

With `subscriptions = true` (the default), Perspt intersects its filter with server capabilities before opening `subscriptions/listen`. Tool-list changes invalidate the SDK cache; the shared runtime removes old bindings and repeats local admission before a chat model turn. A removed tool cannot remain callable. Resource and prompt changes, progress, logging, tasks, cancellation, and custom extension notifications are typed `McpServerEvent` values for product display or SDK consumption.

`resource_subscriptions` contains exact URIs and should be narrow. An empty list still permits supported tools/prompts/resources list-change filters.

Multi-Round Results and Tasks

Tool calls accept complete results, bounded multi-round input-required responses, or asynchronous task handles. Inputs may request sampling, elicitation, or roots only when locally enabled; unknown kinds fail closed.

Every wire request uses `timeout_ms`. `max_task_wait_ms` is an overall deadline across task polling and input rounds (default five minutes), and poll hints are clamped. A failed, cancelled, malformed, or expired task becomes an uncertain external completion for agent reconciliation.

Configuration Reference

Table 15 `[[external_tools]]` fields

Field	Meaning
`id`	Required ASCII id used in `mcp.<id>.<operation>`.
`transport`	Exactly `stdio` or `streamable_http`.
`command` / `url`	Direct argv for stdio or HTTP endpoint; mutually exclusive.
`modes`	`["agent"]`, `["chat"]`, or both. Omission means agent only.
`env_from_env`	Stdio child variable to source-variable mapping. Children otherwise receive only `PATH` and Windows `SystemRoot`.
`headers_from_env`	HTTP header to source-variable mapping; values stay out of TOML.
`timeout_ms`	Per-request and discovery deadline; default 30000.
`max_result_bytes`	Maximum accepted serialized result and raw SSE event; default 1 MiB.
`max_stderr_bytes`	Rolling stdio stderr tail; default 64 KiB.
`roots`	Explicit absolute `file://` roots. Empty means unadvertised.
`sampling` / `max_sampling_tokens`	Opt in to server model calls and cap one request (default 4096).
`elicitation`	Opt in to form/URL elicitation; interactive in chat, declined by the non-interactive agent.
`subscriptions` / `resource_subscriptions`	Enable the listen stream (default true) and exact resource filters.
`tasks` / `max_task_wait_ms`	Enable task/MRTR handling (default true) and its overall deadline (default 300000).
`tools`	Local policy tables keyed by exact remote tool name.

Each policy accepts `effect`, `risk`, `footprint`, and optional `proposal_bindings`. Omitting effect or footprint becomes a high-risk opaque shell declaration and is not admitted by default.

Security Notes

A configured stdio command is user-selected code, and an HTTP server acts outside the candidate workspace. Admission constrains calls but does not sandbox the server. Trust its implementation/operator and grant the narrowest policy.

Result objects are size-checked after protocol decoding and SSE events before decoding. Stderr capture, subscription filters, MRTR rounds, task duration, sampling tokens, and chat tool rounds

are bounded. The official SDK owns framing, correlation, cancellation, and dispatch; Perspt owns transport security, capabilities, admission, product interaction, and replay policy.

Troubleshooting

Symptom	Check
`/mcp` says disabled	Add `"chat"` to `modes` and restart chat.
Protocol is incompatible	Upgrade the server to 2026-07-28; there is no legacy fallback.
No remote tools are admitted	Add exact policies and schema-valid footprints. Resource/prompt operations may still appear when advertised.
Stdio program or credential is missing	Verify direct argv and map variables with `env_from_env`.
HTTP is refused	Use HTTPS outside localhost and check header source variables.
Sampling is rejected	Enable it, stay under the token cap, and omit unadvertised context.
Elicitation is declined in agent mode	Use chat or install an SDK host provider.
A task times out	Increase the overall deadline deliberately or fix the server.

Troubleshooting

API Key Issues

Symptom: "No API key found" error.

Solutions:

1. Check whether the variable is set without printing it: `test -n "$GEMINI_API_KEY" && echo set`
2. Verify spelling: `ANTHROPIC_API_KEY` (not `ANTHROPIC_KEY`)
3. Store it in the config: `perspt config --set api_key="your-key"`
4. Check config file: `~/.config/perspt/config.toml`

Connection Errors

Symptom: "Connection refused" or timeouts.

Solutions:

1. Check internet connectivity
2. For Ollama: ensure `ollama serve` is running
3. Check firewall/proxy settings
4. Try a different provider

Agent Mode Issues

Agent stuck in retry loop:

1. Check tool prerequisites: `which uv`, `which cargo`, `which node`
2. Check LSP is functioning: `ty check .` or `cargo check`
3. Relax the descent gate: `--rho-gate 0.3`
4. Raise the retry allowance: `--rejection-budget 8`
5. Check `perspt status` for gate decisions and denials

High energy despite clean code:

1. Run tests manually: `uv run pytest -v` or `cargo test`
2. Check for LSP diagnostics: `ty check .`
3. Bound runaway turns: `--max-turns 8 --max-calls-per-turn 4`
4. Verify contract compliance

Plugin not detected:

1. Ensure required binaries are installed in PATH
2. Check workspace has expected marker files (`Cargo.toml`, `pyproject.toml`)
3. Run `perspt status` to see ledger counters, last energy, and gate state

TUI Rendering Issues

Symptom: Garbled output, incorrect colors.

Solutions:

1. Ensure terminal supports 256 colors: `echo $TERM`
2. Try a different terminal emulator
3. Fallback to simple CLI: `perspt simple-chat`
4. Check for conflicting terminal multiplexer settings

Degraded Verification

Missing required tools block a hard pass. The runtime records the missing evidence rather than treating it as a clean measurement. Non-required stages may produce advisory findings.

To restore full verification, install the required tools:

```
# Python projects
pip install ty pytest

# Rust projects
rustup component add rust-analyzer

# Node.js projects
npm install -g typescript
```

Session Recovery

If a session is interrupted:

```
# Check what's in progress
perspt status

# Resume the last session (shows trust context first)
perspt resume --last

# Or abort and start fresh
perspt abort
```

If the session store itself refuses to open because of a poisoned DuckDB write-ahead log, back it up and quarantine it (the WAL is never deleted):

```
perspt db repair --db-path <path-to-db> --discard-wal
```

Getting Help

```
perspt --help
perspt agent --help
perspt chat --help
```

For more details, see:

- CLI Reference - Full CLI reference
- Advanced Troubleshooting - Advanced troubleshooting

Dashboard

Watch a live agent run in the browser, or inspect completed sessions. The dashboard shows recorded measurements and decisions from the session store.

Watch a live run

Start the dashboard inside the agent process:

```
perspt agent --dashboard -w ./myproject "Fix the failing parser test"
```

Open `http://127.0.0.1:3000` while the agent runs. The embedded dashboard shares the agent's store connection and stops when the agent exits. Use `--dashboard-port 8080` to select another port.

Use the embedded dashboard for live monitoring. A separate `perspt dashboard` process opens DuckDB read-only and can conflict with an active writer's database lock.

Inspect completed sessions

After the agent exits:

```
perspt dashboard
# Or select a database and port
perspt dashboard --db-path /path/to/perspt.db --port 8080
```

The database must already exist. The standalone dashboard opens it read-only. Recent sessions are listed with pagination.

Find the evidence

Page	What to inspect
Overview and session detail	Session status, task, work directory, node counts, and measurement summaries.
Topology	Recorded graph revisions, node states, and dependency edges.
Energy	One row per candidate measurement with its node, generation, score, and hard-pass flag. Observed scores can rise even when accepted progress descends.
Backlog	Latest node states, backlog energy, and conditional-capacity diagnostics.
Decisions	A flat hash-chained event trace with sequence, kind, summary, and hash.
Governance	Authority, calibration, validator verdicts, and pending audit labels.

The live session statistics use Server-Sent Events every two seconds. These measurements describe the checks that ran; they do not establish requirements beyond their coverage.

Local access

The CLI binds the dashboard to `127.0.0.1` without authentication. It is intended for local inspection; do not forward the port to other machines. The read-only dashboard interface does not promote edits. See Dashboard Setup for setup details.

Concepts

Understanding the foundations of Perspt.

Concept	Scope
SRBN Architecture	The theoretical framework and its experimental implementation in Perspt.
Stability and Measured Progress	How the three Stability papers shape PSP-8, PSP-9 (Final), PSP-10, and agent mode.
Workspace Crates	The fourteen-crate modular architecture.
PSP Process	Perspt Specification Proposals for design and development.

SRBN Architecture

The **Stabilized Recursive Barrier Network (SRBN)** is the idea behind Perspt's experimental autonomous coding agent. It comes from the *Stability is All You Need* paper series.

Models propose; tools measure; the gate decides what progress to keep. A proposal can contain errors that later edits build on. Perspt checks candidate states and records both observations and decisions. The runtime uses the shared contracts in `perspt-sdk`.

> **Theory vs. Implementation**
>
> This page describes both the SRBN paper series and how Perspt's runtime implements it. Where a claim comes from the papers' formal proofs, it is noted as a **paper result**. Where the developer team makes engineering choices that approximate or extend the theory, those are noted as **implementation details**. The coding domain uses measured discrete gates and declares its continuous analytic constants `NotClaimed`. Formal assumptions and illustrative figures do not establish an empirical success rate.

Overview

The SRBN paper models coding tasks as a directed acyclic graph (DAG) of nodes with a sheaf structure that enforces consistency across shared boundaries. The system establishes the following concrete mechanics: each node owns a set of output files (ownership closure), runs a governed tool loop of typed tool calls against an isolated candidate, and must pass the measured acceptance gate on deterministic verifier evidence. Accepted checkpoints are recorded in the ledger. Delivery additionally requires the required checks and promotion approval; multi-node runs verify the merged result.

The runtime environment structures execution in two primary ways:

- **Quadratic energy.** Acceptance is gated on the quadratic residual energy $V(x) = \sum_e w_e \parallel r_e \parallel^2$ (see Lyapunov Energy below).
- **Mutable work graph.** Rather than walking a precomputed topological order, a closed-loop scheduler re-evaluates a dependency-aware ready queue each round and may requeue, split, insert, or replan nodes as verifier evidence arrives. Each individual graph *revision* stays acyclic. Bounded parallelism shipped in v0.6.6: the multi-node dispatcher behind `--max-parallel-nodes` runs ready nodes concurrently, and two nodes with conflicting file footprints never run at the same time.

The core concepts of the control system — ownership closure, typed tool calls, the verifier profiles, node classes, and the Merkle ledger — are described below.

The Control Loop

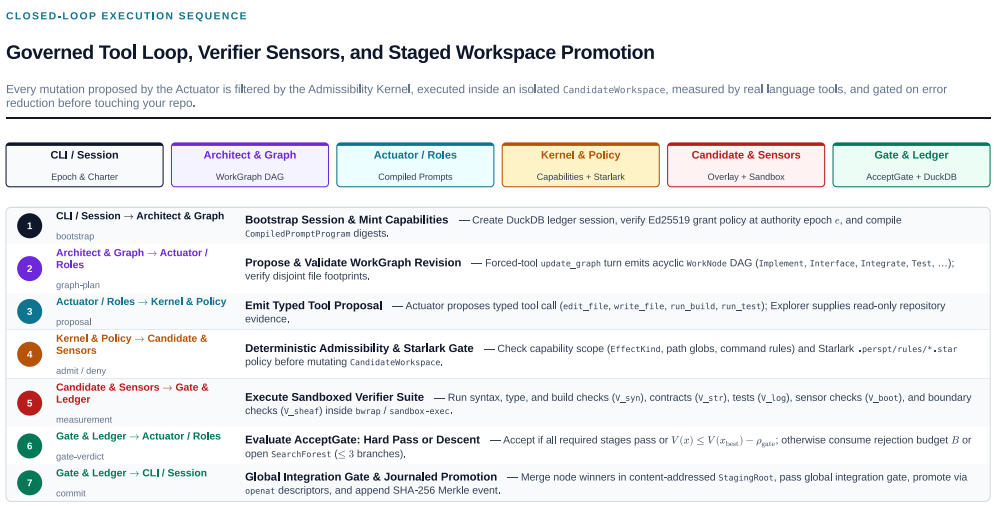


Fig. 4 Sequence of the PSP-9 / PSP-10 governed candidate control loop across Architect, Actuator, Admissibility Kernel, Candidate Overlay, Deterministic Verifier, Accept Gate, and Merkle Ledger.

Bootstrap and the architect turn run once at the start; the remaining phases run **per node inside the closed loop**. The dispatcher re-evaluates the mutable work graph, fills free slots with ready nodes, and runs the governed tool loop for each. A node that fails its gate can open a bounded search forest and be re-attempted, so these phases are *not* a single topological pass:

#	Phase	Description
1	**Session Bootstrap**	Create the durable session, detect the domain package and language plugins, mint epoch-bound capabilities from the grant policy, and compile the prompt programs for the resolved routes and dialects.
2	**Architect Turn**	A governed, forced-tool-choice model turn restricted to the privileged `update_graph` tool proposes the work-graph revision. Acyclicity and completeness are validated deterministically before the revision is accepted; a single-node run keeps one node without an architect call.
3	**Work-Graph Dispatch**	Behind `--max-parallel-nodes`, the dispatcher fills free slots with ready nodes. Conflicting file footprints never run concurrently, and each node attempt is

#	Phase	Description
		bound to a generation seeded from the current staging root, never from unstaged sibling state.
4	**Governed Tool Loop**	Per node, model turns issue typed tool calls. Each call becomes a proposal; the deterministic admissibility kernel decides whether it may affect the candidate; admitted effects mutate an isolated candidate overlay; sandboxed deterministic verifiers re-measure the candidate.
5	**Accept Gate**	A candidate is accepted on a hard verifier pass, or on a measured energy descent of at least ρ_{gate} below the best accepted state. Non-descending candidates consume the shared rejection budget; the gate is evaluated on the re-measured candidate, never on the model's account of it.
6	**Bounded Search** (optional)	On gate failure with budget remaining, a bounded search forest runs up to three isolated branches with distinct strategies against the accepted root. Every branch action reserves its cost before it runs, exact no-goods suppress repeats, and exactly one selected candidate is committed through the same gate.
7	**Staging & Integration**	Each node's winner lands in a content-addressed staging root instead of the user workspace. When dispatch settles, the merged state passes one global integration gate; failure restores the prior staging root.
8	**Promotion**	The integrated winner is written to the user workspace through journaled, descriptor-relative promotion and committed to the hash-chained ledger, from which `perspt replay` and `perspt resume` reconstruct the session.

Lyapunov Energy

The stability of generated code is measured using a Lyapunov energy function, adapted from the paper's sheaf-theoretic formulation into concrete verification barriers. The system evaluates a canonical **quadratic residual energy**: each sensor e emits a residual with a non-negative magnitude $r_e(x)$, and the energy is a weighted sum of their squares.

Let $\mathcal{E}$ be the set of active sensors monitoring the system. For a proposed state x, the total Lyapunov energy $V(x)$ is defined by the quadratic form:

$$V(x) = \sum_{e \in \mathcal{E}} w_e \parallel r_e(x) \parallel^2$$

where $w_e > 0$ represents the positive weight assigned to the residual class of sensor e.

The quadratic expression is continuous in residual magnitudes. The runtime measures discrete workspaces, where residual counts can jump. A smooth illustration does not assert a continuous code-space trajectory.

The five component readouts V_{syn}, V_{str}, V_{log}, V_{boot}, and V_{sheaf} are **derived rollups** of this same energy, grouped by component type, such that the total is the sum of the rollups:

$$V(x) = V_{\text{syn}} + V_{\text{str}} + V_{\text{log}} + V_{\text{boot}} + V_{\text{sheaf}}$$

In this formulation, the rollups themselves carry the squared, weighted residual contributions:

$$V_{\text{comp}} = \sum_{e \in \text{ comp}} w_e \parallel r_e(x) \parallel^2$$

There is no secondary $\alpha/\beta/\gamma$ aggregation pass. The per-class weights w_e are fixed by the domain package's energy model, leaving the core mathematical engine as a pure sum of pre-weighted squares.

7.1.3.1 Components

We enumerate the five component categories and the specific residual classes mapped to them from the *perspt-coding* domain package:

Component	Mapped Residual Classes (Weights)	Operational Semantics
V_syn	• `Syntax` (weight: 4.0) • `Type` (weight: 3.0) • `Build` (weight: 3.0)	Captures syntax errors, build-time compilation failures, and LSP diagnostic warnings. Compiler warnings or type-check diagnostics produce residuals of class `Type`, where the magnitude is the raw diagnostic count. A build failure raises a blocking residual of class `Build`.
V_str	• `ImportGraph` (weight: 2.0) • `SymbolMismatch` (weight: 2.0) • `InterfaceMismatch` (weight: 2.5) • `OwnershipViolation` (weight: 2.0) • `Policy` (weight: 1.0) • `Dependency` (weight: 1.5) • `Manifest` (weight: 1.5) • `Lint` (weight: 0.5) • `Format` (weight: 0.25)	Measures structural contract adherence. If a required symbol defined by a node's public interface signature is absent from the generated output files, the `GoalPresence` sensor registers a blocking `SymbolMismatch` residual, preventing convergence. Policy and format irregularities are likewise squared into this component.
V_log	• `TestFailure` (weight: 2.0) • `Runtime` (weight: 2.0) • `Regression` (weight: 3.0)	Evaluates behavioral outcomes. Unit test failures yield a `TestFailure` residual with a magnitude matching the count of failing test cases. Post-build runtime smoke probes detect process crashes, tracebacks, or numeric anomalies, adding a `Runtime` residual.
V_boot	• `SensorUnavailable` (weight: 1.0) • `ToolFailure` (weight: 1.0)	Registers system infrastructure state. If a required sensor (e.g., an LSP server or test runner) is degraded or unavailable, it raises a `SensorUnavailable` residual, forcing the energy high so the system knows its stability status is indeterminate.
V_sheaf	• `SheafInconsistency` (weight: 2.0)	Evaluates global structural consistency across nodes. Cross-node import/dependency mismatches or signature differences raise a `SheafInconsistency` residual.

7.1.3.2 Convergence Criterion

The system is considered stable if and only if the candidate achieves a **hard pass**: every required verifier and every hard policy constraint passes. A domain may additionally declare an analytic energy floor; a candidate at or below the floor is checkpointed as a classified terminal state but is never reported as verified success. If the state is not stable, the accept-gate evaluates descent.

The single runtime acceptance knob is `--rho-gate`. A state x can be provisionally accepted during convergence if:

$$V(x) \leq V(x_{\text{best}}) - \rho_{\text{gate}}$$

where x_{best} is the best previously accepted state, and ρ_{gate} is the minimum descent gate (default 0.50), ensuring non-trivial progress.

7.1.3.3 Spectral and Independence Diagnostics

The system supports an energy-slope constant $\mu = 2\,\lambda^{+}_{\min}(A)$ — twice the algebraic connectivity (Fiedler value) of the verification graph built from the quadratic energy $V(x) = x^{\top}Ax$. It measures how strongly the verifier ensemble drives the code toward consensus, and its sensitivity to a candidate verifier edge distinguishes an *independent* verifier (which raises the spectral gap) from a *redundant* one. A companion measure, ρ_{eff}, captures effective verifier independence from observed miss correlations.

> **Note**
> `mu` and ρ_{eff} are **diagnostics, not acceptance-gate inputs** — by design they are computed off the critical path and never block a node. The independence statistics are **live**: `independence::compute` folds the `psp9_verdicts` ledger into per-validator miss rates and matched-stratum pair statistics, and ρ_{eff} is reported only when every pair meets the matched-sample floor (a Hoeffding upper confidence bound backs the certified value). `perspt status` and the dashboard's Governance page surface these figures. The spectral eigensolver (`spectral::VerificationGraph`) remains an offline `perspt-sdk` diagnostic.

Node Classes

The following three classes describe the historical PSP-5 design. The current SDK has a broader taxonomy, and the planner currently emits `Implement` nodes. See Agent Mode for the current node classes.

Class	Description
Interface	Define exported signatures, schemas, and seals. Must be committed before dependent Implementation nodes can proceed. Produces an interface-seal digest.
Implementation	Operate on owned files using sealed interfaces from parent nodes. The bulk of code generation happens here.
Integration	Reconcile cross-owner boundaries after all dependent nodes converge. Used for multi-language projects or cross-module wiring.

Ownership Closure

The **ownership closure** rule is a fundamental invariant of PSP-5:

> *Each output file appears in exactly one node's* `output_files` *list.*

This prevents conflicting writes. In the current runtime the guarantee is enforced at dispatch through footprints: two nodes whose declared file footprints conflict never run concurrently, and each node attempt is seeded from the staging root rather than from unstaged sibling state.

Model Roles

The runtime uses one required and two optional model roles. Verification is **not** a model role: the verifiers are deterministic sandboxed sensors (compilers, test runners, linters), never a model call.

Role	Purpose	CLI Flag
Actuator	Proposes governed coding tool calls; the only role that can mutate the candidate. Also serves the architect's `update_graph` turn.	`--model` / `--actuator-model`
Explorer	Optional cheaper read-only repository exploration.	`--explorer-model`
Adjudicator	Optional no-tool conjunctive diff adjudication before promotion.	`--adjudicator-model`

Configure per-role routes via CLI:

```
perspt agent \
  --actuator-model <provider::model> \
  --explorer-model <provider::model> \
  --adjudicator-model <provider::model> \
  --fallback-model <provider::model> \
  "Build a REST API"
```

`--fallback-model` is repeatable and defines the ordered sticky actuator failover route. Persistent per-role routes live in the `[models]` table of `config.toml` as fully qualified `provider::model` values, so identity, calibration, and replay never depend on an ambient default provider. No model name is hard-coded in the runtime.

PSP-10: Model-Conditioned Prompt Programs

PSP-10 replaces free-form prompt templates with compiled prompt programs: typed sections, deterministic composition per model route, and ledgered digests for every model call.

7.1.7.1 Typed Prompt Sections

Prompt text lives in versioned section files under `crates/perspt-core/prompts/<stage>/NN_name.md`, one directory per stage (`session_bootstrap`, `graph_plan`, `repository_explore`, `adjudicate`, `evidence_summarize`), beside a committed `manifest.toml`. The `perspt-prompt-macros` crate compiles every section at build time from the owning crate's `build.rs`: it parses the front matter into a typed `SectionSchema` (id, version, role, priority, size bound, declared variables), runs the codegen validation list, and emits typed section structs into `OUT_DIR`. A malformed section fails `cargo build` with an error naming the offending file — never a session.

7.1.7.2 Stage Composition and Digests

At runtime a `StageComposition` composes one stage's rendered sections for the resolved model route and dialect into a `CompiledPromptProgram`: the ordered messages with per-message rendered hashes, any sections dropped by deterministic budget fitting, the tool-spec hash, and a `program_digest` computed over canonical bytes of the identity-bearing fields. One model call binds a platform program (the universal envelope) and a domain program; the pair forms a `CompiledPromptInvocation` with its own invocation digest.

7.1.7.3 Provenance

Every model call is ledgered: a `prompt_program_compiled` event records each program's sections and digest, and a `prompt_program_invoked` event records the actor, turn, platform digest, domain digest, invocation digest, and tool-spec hash. A session's prompts are therefore reconstructible and diffable after the fact. Validated replacement bundles may substitute section bodies live only behind `--allow-experimental-prompts`; external bodies obey exactly the validation rules the build does.

7.1.7.4 The `perspt prompts` CLI

```
perspt prompts list                    # every compiled section: id, version, stage, role, hash
perspt prompts render <stage>          # compose one stage with fixture variables
perspt prompts lint --bundle <dir>     # validate an external bundle directory
perspt prompts manifest <dir>          # regenerate a library's manifest.toml
perspt prompts explain-session --db-path <db> <session-id>
                                       # the programs a session actually compiled, with digests
```

`perspt context explain-turn` is the companion command for a session's recorded resident-context events (compactions and refusals).

Rejection Budget

There is no per-error-type retry table. The loop runs against a single shared, non-replenishing **rejection budget** B (default 4, `--rejection-budget`): non-descending candidates and recovery attempts consume it, while accepted descents do not. Together with the descent gate this yields the finite-decision bound $\lfloor V_0/\rho_{\text{gate}} \rfloor + B + 1$ on gate decisions per node. When the budget is exhausted, the gate issues a **residual certificate** enumerating the remaining errors and the node terminates as a classified failure rather than looping. In headless mode (`--yes`) the session exits with a non-zero code when the outcome is not verified success.

Typed Tool Proposals

The Actuator does not emit artifact bundles. Every mutation is an individual typed tool call: the call becomes an effect proposal, the deterministic admissibility kernel checks it against the node's capability (path patterns, command patterns, effect kinds), and only an admitted effect executes — against the isolated candidate overlay, never the user workspace. Proposed commands additionally pass the deterministic `sanitize_command` and workspace-bound guards plus the Starlark policy engine, and normally run inside the process sandbox. Explicit native Windows reduced-isolation mode retains those agent checks but runs child processes with host-user authority. The candidate journals a pre-image of every file it touches, so any attempt can be discarded without residue.

Plugin-Driven Verification

Language plugins determine the verification toolchain:

Plugin	LSP Server	Test Runner	Init Command	Required Binaries
Rust	`rust-analyzer`	`cargo test`	`cargo init`	`cargo`, `rustc`
Python	`ty` or `pyright`	`pytest`	`uv init --lib`	`uv`, `python3`
JavaScript	`typescript-language-server`	`npm test`	`npm init -y`	`node`, `npm`

Built-in plugins detect Rust, Python, and JavaScript/TypeScript from workspace markers and supported source files. There is no built-in Go plugin. Multi-language projects activate multiple plugins simultaneously.

Degraded Verification

Missing required evidence blocks a hard pass. A detected fallback can provide a separately identified sensor. If no required syntax, build, or test command can run, the runtime records a blocking `SensorUnavailable` residual. Non-required stages can report advisory findings without blocking the hard gate. A plugin can also declare a stage as a no-op, such as the Python build stage. The runtime does not silently treat a missing required verifier as a zero-error measurement.

Merkle Ledger

All changes are recorded in a DuckDB-backed Merkle ledger:

- **Integrity** - Each commit has a cryptographic hash chaining to its parent
- **Rollback** - Roll back a session's newest completed promotion via `perspt ledger --rollback` (the argument is a session id prefix)
- **Resume** - `perspt resume` continues an interrupted session from its newest durable checkpoint by folding the ledger (see Crash Resume)
- **Audit** - Complete trail of AI-generated changes with energy breakdowns

```
perspt ledger --recent     # View recent commits
perspt ledger --rollback abc123
perspt ledger --stats      # Session statistics
```

Isolation and Resume

Nothing speculative ever touches the user workspace. Three layers keep observed work separated from accepted work, and crash resume preserves the separation.

> **Key Invariant**
> Model-proposed work reaches the user workspace only through the accept gate, the staging root, and the global integration gate — in that order. Everything else is discarded without residue.

7.1.13.1 Candidate Overlays

Each node attempt runs in a `CandidateWorkspace` overlay. Admitted effects mutate the overlay; every touched file's pre-image is journaled; verifiers measure the overlay, not the workspace. A rejected attempt is dropped wholesale.

Test evidence follows an explicit policy. The ordinary `evolving` policy measures the resulting implementation and resulting tests together, which is necessary when a task intentionally changes a contract. A `backward-compatible` run adds a regression view with recognized historical test files restored. An `external-oracle` run adds a separately supplied acceptance overlay that the actuator does not promote. These are additional test-stage views of the same realized candidate, not alternate paths around the syntax, build, or test requirements.

7.1.13.2 Search Forest Branches

A bounded search forest opens at most three branch identities, one branch attempt at a time. Every branch runs the ordinary governed loop in an isolated eager-copy workspace against the same immutable accepted root, and its internal states stay private to the forest. Exactly one selected candidate is committed through a single gate submission, and the committed decision must equal the preview or the forest fails closed. Exact no-goods learned from failed branches suppress equivalent later attempts, and every branch action reserves its cost against the declared limit vector before it runs.

7.1.13.3 Staging and the Integration Gate

With `--max-parallel-nodes` above 1, node winners land in a content-addressed staging root instead of the user workspace. Disjoint footprints are enforced at dispatch; the merged state must pass one global integration gate before promotion, and a gate failure restores the prior staging root. The integration gate uses the same session-bound test policy as the node gates: resulting tests for ordinary evolving development, an additional historical regression view for explicitly backward-compatible work, or a separately protected external acceptance overlay.

7.1.13.4 Crash Resume

`perspt resume` folds the durable ledger rather than deserializing live state. A single-node session re-enters its loop from the newest durable candidate checkpoint with exactly the remaining budgets, its accepted candidate rebuilt from content-addressed artifacts and a fresh capability minted at the current authority epoch (a bumped epoch refuses the resume). A multi-node session resumes through graph dispatch: the staging root is rebuilt from the ledger and every winner still reaches the user workspace only through the global integration gate.

Stability and Measured Progress

Perspt separates a model's proposal from evidence that the proposal works. The runtime measures candidate states with project tools, then decides whether to keep progress or ask for a correction. This page describes the implemented coding contract and how it relates to the stability terminology used in the specifications.

The energy score

A verifier reports a residual: a measured issue such as a compiler diagnostic, failing test, or missing declared symbol. For residual magnitudes $r_e \geq 0$ and positive weights w_e, the SDK computes:

$$V(r) = \sum_e w_e r_e^2$$

The quadratic formula is continuous in the residual magnitudes. The coding runtime, however, measures discrete candidate workspaces. Counts of diagnostics and failing tests can jump between measurements. There is no implemented continuous coordinate system over source-code states and no guarantee that every path through code space is smooth. The coding domain declares its continuous analytic stability constants `NotClaimed`.



Fig. 5 The dots are the decisions that matter. Curves guide the eye between illustrative measurements; they are not a simulation, benchmark, or continuous coding trajectory.

The five reported components group the same weighted contributions:

- **Syntax**: Syntax, type, and build residuals.
- **Structure**: Import, symbol, interface, and other structural residuals.
- **Logic**: Test failures, runtime failures, and regression residuals.
- **Bootstrap**: Missing required sensors and tool failures.
- **Sheaf**: Cross-node consistency residuals.

The components sum to the total without a second weighting step. Capability denials and budget exhaustion are recorded in a separate blocked channel rather than added to this score. See `crates/perspt-sdk/src/energy.rs` and `crates/perspt-coding/src/lib.rs` for the scoring implementation and coding weights.

Keep progress without claiming completion

For the ordinary measured gate:

$$\text{accept}(y) \iff \text{hard_pass}(y) \quad \vee \quad V(y) \leq V(x_{\text{best}}) - \rho_{\text{gate}}$$

A hard pass means all required verifiers and hard constraints passed. Otherwise, the candidate must reduce the best accepted energy by at least ρ_{gate} (default `0.5`). The comparison uses `<=`, so a decrease exactly equal to the margin is sufficient.

Acceptance by descent is an intermediate checkpoint. It does not authorize delivery of incomplete code. Final promotion requires the required checks and approval; multi-node runs also check the merged integration result. Zero measured energy alone is not sufficient evidence that every required stage ran and passed. Likewise, an empty `EnergyScore.hard_violations` list only says no configured residual threshold was exceeded.

The general SDK gate can classify a stop at a declared analytic floor. That terminal outcome is not verified success. The coding domain does not claim an analytic floor. The implemented ordering and outcomes are in `crates/perspt-sdk/src/gate.rs`.

Bound the attempts

With baseline energy V_0, positive descent margin ρ_{gate}, and rejection budget B, the measured node gate has the bound:

$$N_{\text{gate}} \leq \lfloor V_0/\rho_{\text{gate}} \rfloor + B + 1$$

This bounds gate decisions under the declared contract. It does not promise that a model will find a passing candidate or establish a model success probability. Turn, call, and recovery limits also apply. When a run cannot complete within its limits, it stops with a classified outcome and recorded remaining errors.

Observe, accept, and promote

We distinguish three events:

1. **Observe**: The runtime measures a proposed candidate and records its evidence.
2. **Accept**: A hard pass or sufficient descent advances the node's accepted trajectory.
3. **Promote**: The runtime copies a deliverable, approved result into the project.

The ledger records rejected observations as well as accepted checkpoints. Rejected proposals do not become accepted progress and do not modify the project files through promotion. Ledger checkpoints and promotion are not Git commits.

The scope of verification

A required check that cannot run blocks a hard pass. Non-required stages can report advisory failures without changing the hard gate. The default test policy measures the resulting implementation and tests together. Historical regression and externally maintained acceptance evidence are explicit policies described in Configuration Guide.

A hard pass is evidence within the configured checks' coverage. It cannot establish behavior that the checks do not cover. Optional spectral and validator-dependence statistics are diagnostics, not acceptance inputs.

Relation to the stability papers

Perspt's specifications use the *Stability is All You Need* paper series as a design framework for measured progress, bounded correction, and governed effects. The implemented coding runtime uses the discrete measured gate above. Continuous geometric claims require additional assumptions that the coding package does not declare. The specifications preserve the design rationale; SRBN Architecture describes the runtime mechanisms and their limits.

Workspace Crates

Perspt is organized as a fourteen-crate Rust workspace under the `crates/` directory. The program separates user interaction, core execution, security policy, persistence, and domain specifications into distinct libraries.

Eight crates form the running executable program; three crates define the reusable platform SDK and its target domains; `perspt-prompt-macros` is the build-time compiler for the prompt section

libraries; `perspt-benchmark` is an optional, manually run evaluation suite (feature-gated in `perspt-cli`, never part of normal validation); one meta-crate re-exports the libraries.

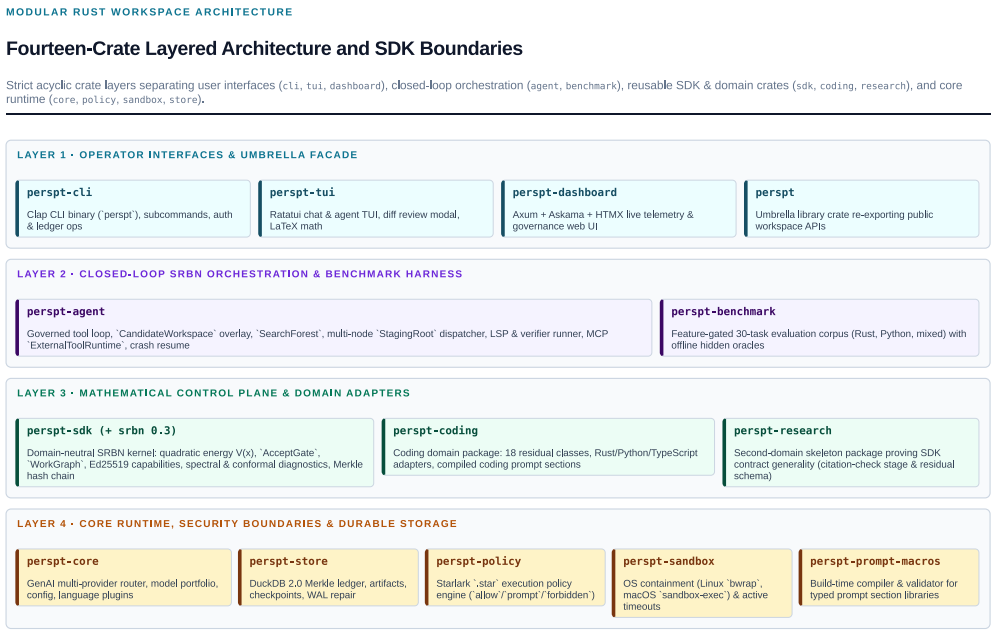


Fig. 6 Layered architecture of the fourteen-crate Perspt workspace across user surfaces, governed runtime & security, domain-neutral SRBN SDK, and evaluation harness.

Crate Summary

Crate	Operational Purpose	Primary Structural Types
perspt-cli	Parses command-line inputs and dispatches system commands.	`Cli`, `Commands`
perspt-core	Formulates shared data structures, configurations, and provider connections.	`GenAIProvider`, `Config`, `TaskPlan`, `NodeClass`, `AgentEvent`
perspt-agent	Executes the governed PSP-9 tool loop and work-graph runtime.	`Psp9AgentRuntime`, `CandidateWorkspace`, `AgentTools`, `GenAiTransport`
perspt-tui	Implements the terminal user interface for interactive review.	`ChatApp`, `AgentApp`, `DiffViewer`, `ReviewModal`
perspt-store	Manages the DuckDB persistence layer for session logs.	`SessionStore`, `SessionRecord`, `Psp9LedgerRow`
perspt-policy	Executes Starlark policies to validate proposed command structures.	`PolicyEngine`, `sanitize_command`
perspt-sandbox	Runs external commands inside isolated subprocess environments.	`SandboxedCommand`, `CommandResult`
perspt-dashboard	Implements a read-only browser-based monitoring engine.	`AppState`, Axum router, SSE stream

Crate	Operational Purpose	Primary Structural Types
perspt-sdk	Defines domain-neutral SRBN stability contracts and gate models.	`ResidualEvent`, `EnergyModel`, `AgentDomainPackage`, `ResidualCertificate`
perspt-coding	Implements the coding-domain adapters, sensors, and mappers.	`CodingDomain`, `LanguageId`, `CodingAdapterRegistry`
perspt-research	Implements the research-domain adapter to demonstrate SDK reuse.	`ResearchDomain`
perspt-prompt-macros	Compiles prompt section files into typed sections at build time.	`StageDecl`, `PromptBuildError`
perspt-benchmark	Runs the optional, manually invoked PSP-10 evaluation ladder.	`BenchmarkSuite`, `BenchmarkRunOptions`

7.3.1.1 perspt-cli

The entry-point command-line utility. It parses command-line flags and subcommands using the `clap` library and dispatches execution control to corresponding subsystems in other workspace crates.

Key structural types include:

- `Cli`: The root structure declaring command-line configuration arguments.
- `Commands`: The enum representing the sixteen subcommands (plus a feature-gated seventeenth):
  - `chat`: Starts the interactive terminal chat mode (the default).
  - `simple-chat`: Simple CLI chat mode without the TUI.
  - `agent`: Runs the governed PSP-9 agent on a task.
  - `init`: Initializes project memory and Starlark policy rules.
  - `config`: Modifies runtime configuration values in `config.toml`.
  - `ledger`: Commands to query, inspect, or rollback Merkle ledger sessions.
  - `status`: Shows current agent status.
  - `audit`: Delayed audit labels and conformal activation.
  - `providers`: Prints the provider capability matrix.
  - `replay`: Deterministic, credential-free audit replay of a session.
  - `abort`: Aborts a PSP-9 session by revoking its authority epoch.
  - `resume`: Resumes a paused or crashed session.
  - `dashboard`: Launches the monitoring Axum server and opens the browser.
  - `db`: Inspects and repairs the local DuckDB store.
  - `prompts`: Inspects and maintains the compiled prompt section libraries.
  - `context`: Explains a session's recorded resident-context events.
  - `benchmark` (`--features benchmark` only): Runs the optional model-backed evaluation tooling.

7.3.1.2 perspt-core

The foundation crate. It defines the central types, data models, and configurations shared globally across all crates. Among workspace crates it depends only on `perspt-sdk` and, at build time, on the `perspt-prompt-macros` compiler.

Key components:

- **Core Types**: Shared configuration, plugin, and workspace structures. (Note: the active PSP-9 work-graph node taxonomy `NodeClass` — `Explore`, `Plan`, `Implement`, `Verify`, `Test`, `Integrate`, `Repair`, `Interface` — lives in `perspt_sdk::workgraph::NodeClass`, while `perspt_core::types::NodeClass` retains the legacy three-way `Interface` / `Implementation` / `Integration` grouping):

```
pub enum NodeClass {
    /// Defines exported signatures, schemas, ownership manifests
    Interface,
    /// Operates on node-owned files plus adjacent sealed interfaces
    Implementation,
    /// Reconciles cross-owner or cross-plugin boundaries
    Integration,
}
```

- **Event Plane**: Defines the event system carrying 33 `AgentEvent` lifecycle variants transmitted across channels to update the TUI and log database records.
- **Provider Gate**: Integrates with the `genai` crate to handle provider-agnostic LLM requests, supporting token streaming, prompt assemblies, and backoff limits.
- **Prompt Libraries**: Hosts the versioned prompt section files under `crates/perspt-core/prompts/` (`session_bootstrap`, `graph_plan`, `repository_explore`, `adjudicate`, `evidence_summarize`), compiled at build time by `perspt-prompt-macros` against the `perspt-sdk` prompt module.

7.3.1.3 perspt-agent

The authoritative PSP-9 runtime executor. It houses the governed tool loop, the work-graph dispatcher, tool definitions, and system sensors. Prompt text does not live here: the section libraries live under `crates/perspt-core/prompts/`.

- **Governed Tool Loop**: The loop in `crates/perspt-agent/src/toolloop/` composes the `perspt-sdk` admissibility kernel, the measured acceptance gate, and the durable ledger. Every model-issued tool call becomes a typed proposal; admissible effects mutate an isolated `CandidateWorkspace` overlay; the `CodingCandidateMeasurer` re-measures the candidate under the configured test-evidence policy; and the gate is evaluated on the measurement, never on the model's account of it. Ordinary development uses the resulting tests, while historical regression and protected external acceptance suites are explicit opt-ins.
- **Work-Graph Dispatcher**: `Psp9AgentRuntime` drives multi-node execution behind `--max-parallel-nodes`. A governed architect turn proposes graph revisions through the privileged `update_graph` tool; the dispatcher fills free slots with ready nodes whose file footprints do not conflict, stages each node's winner into a content-addressed staging root, and promotes the merged result through one global integration gate. `perspt-agent` depends on `perspt-sdk` directly; there is no bridge layer.

7.3.1.4 perspt-tui

The terminal user interface built using the `ratatui` library. It provides high-performance terminal panels optimized for real-time human-in-the-loop review.

- **ChatApp**: An interactive chat client that supports streaming LLM dialogue, rendering tables, and parsing markdown.

- **AgentApp**: Displays active workspace executions. It visualizes the task graph DAG, displays active Lyapunov energy gauges, and opens a `ReviewModal` with unified code diffs for manual validation.

7.3.1.5 perspt-store

The local database storage library built on DuckDB. It manages all persistent schemas for sessions, LLM requests, energy convergence logs, and ledger commits.

Key interface structures:

- `SessionStore`: Encapsulates connection pools to the DuckDB file database.
- **Schema Definitions**: It creates and maintains eleven tables:
  - `sessions`: Top-level session tracking.
  - `schema_migrations`: Applied schema versions.
  - `psp9_ledger_events`: The durable hash-chained canonical event stream.
  - `psp9_artifacts`: Content-addressed artifact bytes keyed by hash.
  - `psp9_authority_epochs`: Per-session authority epoch counters.
  - `psp9_grant_policies`: Persisted signed grant intent.
  - `psp9_external_effects`: External effect intents keyed by idempotency key.
  - `psp9_context_checkpoints`: Durable context checkpoints per covered event root.
  - `psp9_verdicts`: Per-validator verdicts feeding the independence statistics.
  - `psp9_calibration_epochs` and `psp9_calibration_samples`: Conformal calibration state and delayed-audit sample labels.

7.3.1.6 perspt-policy

The security policy engine running the Starlark scripting runtime. It checks proposed mutations before execution to guarantee safety bounds.

Key API signatures:

```
pub enum PolicyDecision {
    Allow,
    Prompt(String),
    Deny(String),
}

impl PolicyEngine {
    pub fn evaluate(&self, command: &str) -> PolicyDecision;
    pub fn is_safe(&self, command: &str) -> bool;
}
```

Each Starlark policy is a single function that receives the command string and returns `"allow"`, `"prompt"`, `"deny"`, or a boolean; the engine folds multiple policies by taking the strictest decision. The crate also exports the deterministic guards `sanitize_command`, `validate_workspace_bound`, and `validate_artifact_mutation`, which run before any Starlark evaluation.

7.3.1.7 perspt-sandbox

Bounded command-execution and process-isolation library. It uses Bubblewrap on Linux and `sandbox-exec` on macOS, fails closed when required isolation is unavailable, and exposes an

explicit best-effort mode for externally isolated embedders and acknowledged native Windows operation.

- `ProcessSandbox` & `BasicSandbox`: Implement `SandboxedCommand` to wrap child processes with OS filesystem/network containment (`bwrap` on Linux, `sandbox-exec` on macOS), active wall-clock timeouts, environment sanitization, and bounded stdout/stderr capture (8 MiB default limit).

7.3.1.8 perspt-dashboard

A read-only local dashboard built using the Axum web framework. It reads execution history directly from the DuckDB store and streams live progress to a local browser page.

- **Server Routing**: Sets up REST routes for historical session analysis.
- **SSE Stream**: Sends live event frames (graph state changes, energy rollups) using Server-Sent Events (SSE).

7.3.1.9 perspt-sdk

The core platform library defining the reusable SRBN stability protocol specifications.

- **Domain-Neutral SDK**: Implements the mathematical formulations for the Measured Acceptance Gate. It has no knowledge of language rules, compilers, or project layouts. It operates purely on mathematical residuals and domain contracts.
- **Domain Integration Trait**: The core trait that custom domain extensions must implement:

```rust
pub trait AgentDomainPackage: Send + Sync {
    fn domain_id(&self) -> DomainId;
    fn detect(&self, workspace: &WorkspaceSnapshot) -> DomainDetection;
    fn residual_schema(&self, scope: &DomainScope) -> ResidualSchema;
    fn energy_model(&self, scope: &DomainScope) -> EnergyModel;
    fn correction_directions(&self, residuals: &[ResidualEvent]) ->
Vec<CorrectionDirection>;
    // PSP-9 agent contract (required):
    fn tool_entries(&self, scope: &DomainScope) -> Vec<ToolEntry>;
    fn hard_gate_policy(&self, scope: &DomainScope) -> HardGatePolicy;
    fn verifier_suite(&self, scope: &DomainScope) -> VerifierSuiteSpec;
    fn safety_barrier(&self, scope: &DomainScope) -> SafetyBarrierDisposition;
}
```

7.3.1.10 perspt-coding

The coding-domain package, implementing the `AgentDomainPackage` trait.

- **Coding Domain Model**: Translates code verification outcomes (compiler syntax diagnostics, type warnings, failed tests, linter reports, and missing symbols) into SDK `ResidualEvent` representations.
- **Language Adapters**: Implements adapters for Rust, Python, and TypeScript to parse raw toolchain output, associate them with residual classes, and supply directed correction instructions (e.g. telling the Actuator what type error was detected and how to resolve it). Adapters are registered in the `LanguageId`-keyed `CodingAdapterRegistry`, so adding a language means registering an adapter, never editing the runtime.

7.3.1.11 perspt-research

A second-domain contract-conformance skeleton proving that `perspt-sdk` is domain-neutral.

- **Research Domain Skeleton**: Implements `AgentDomainPackage` for workspaces containing `references.bib` or `sources.md`. It declares a `citation-check` required stage, a residual schema (`CitationMissing`, `LogicGap`, `SensorUnavailable`, `ToolFailure`), energy weights, and correction directions without coupling `perspt-sdk` to coding-specific types. It is a contract skeleton rather than a complete LaTeX or manuscript-production workflow.

7.3.1.12 perspt-prompt-macros

The build-time compiler for prompt section files (PSP-10). Called from an owning crate's `build.rs`, it parses every `prompts/<stage>/NN_name.md` file, runs the codegen validation list, and emits typed section structs into `OUT_DIR`. A malformed section fails `cargo build` with an error naming the offending file, never a session. The same validation functions serve the runtime bundle scanner and `perspt prompts lint`.

7.3.1.13 perspt-benchmark

Optional PSP-10 evaluation tooling, independent of the runtime mechanism tests. It exposes the `perspt benchmark` subcommand only when `perspt-cli` is built with `--features benchmark`; live runs require configured routes and credentials and never run in CI or normal validation. Suites (`smoke`, `adaptive`, `full`) run a cumulative ladder of arms over matched tasks and publish paired differences with a seeded bootstrap.

PSP Process

A Perspt Specification Proposal (PSP) records a significant design change and its review. Routine fixes use normal pull requests. The catalog preserves both implemented designs and proposals under discussion. A proposal's lifecycle state and its implementation coverage are separate facts.

Find the current state

The PSP registry is generated from record metadata. It is the source for current proposal numbers, titles, and states. We avoid duplicating that registry in the book. The catalog workflow documents the allowed transitions.

The configured states are `unassigned`, `draft`, `discussion`, `accepted`, `active`, `final`, `rejected`, and `withdrawn`. Maintainers assign a number with `guidedog dsc promote`; the proposal remains `draft`. They open discussion separately with `guidedog dsc state` and an explicit reason. Merging a pull request does not number a proposal or change its state. `active` denotes a living specification; `withdrawn` includes superseded proposals.

```
guidedog dsc list --dir=docs/psps
guidedog dsc check --dir=docs/psps
```

Read the design history

The summaries below describe the ideas introduced by each proposal. For current behavior, use Agent Mode and SRBN Architecture.

7.4.2.1 PSP-5: The Core Lifecycle

PSP-5 is the historical specification of the original multi-file agent execution; the operative platform is now defined by PSP-9 and PSP-10. It supersedes PSP-4 and established:

- **Project-first execution** - The task is modeled as a directed acyclic graph (DAG) of nodes instead of a single-file task.
- **Ownership closure** - Each file is owned by exactly one node.
- **Artifact bundle protocol** - Structured operations containing writes, diffs, and commands.
- **Node classes** - Division of nodes into Interface, Implementation, and Integration classes.
- **Five-component energy** - The Lyapunov energy model decomposed into syntactic, structural, logical, bootstrap, and sheaf components.
- **Plugin-driven verification** - Selection of language-specific verification tools.
- **Provisional branches** - Speculative execution isolated from the main workspace until parent nodes commit.
- **Interface seals** - SHA-256 digests of exported signatures to enforce cross-node contracts.

See the full specification at `docs/psps/records/000005-srbn-multi-file-coding-ux.rst`.

7.4.2.2 PSP-7: The Correction Loop

PSP-7 builds upon the foundation of PSP-5 to enforce robust convergence under non-deterministic failures. It introduces:

- **Typed parse pipeline** - A five-layer fail-closed parsing protocol that recovers from malformed model outputs.
- **Retry classification** - Explicit categorization of verification failures to determine whether to retarget, replan, or retry.
- **Prompt compiler with provenance** - Systematic generation of correction prompts using structural error evidence.
- **Manifest policy enforcement** - Protection of root manifests from implicit mutation.
- **Strict budget exhaustion** - Interception of execution before LLM invocation when steps, revisions, or cost limits are exceeded.

See the full specification at `docs/psps/records/000007-robust-typed-correction-loops.rst`.

7.4.2.3 PSP-8: The Reusable Platform

PSP-8 defines the architecture for separating domain-neutral control mechanisms from domain-specific features. It introduces:

- **SDK-First design** - Movement of scheduling, energy gating, capability checks, and replay ledgers to a reusable SDK.
- **Quadratic energy** - Adoption of the quadratic residual energy formula $V(x) = \sum_e w_e \parallel r_e \parallel^2$.
- **Mutable work graph** - A scheduler that modifies, splits, or inserts nodes in the queue as verification evidence arrives.
- **Capability kernel** - Scoped admissibility checks over proposed effects.

See the full specification at `docs/psps/records/000008-srbn-agent-sdk-work-graph.rst`.

7.4.2.4 PSP-9: The Governed Platform

PSP-9 replaces bundle generation with a governed tool loop and defines the operative agent runtime. It introduces:

- **Governed tool loop** - Every model-issued tool call becomes a typed proposal; a deterministic admissibility kernel decides whether it may affect the candidate.
- **Measured acceptance gate** - A candidate is admitted on a hard verifier pass or a measured energy descent of at least ρ_{gate} below the best accepted state.
- **Model portfolio** - Per-role routes (actuator, explorer, adjudicator) with an ordered sticky fallback chain, configured via `[models]` or CLI flags.
- **Durable event ledger** - A hash-chained event stream enabling deterministic, credential-free replay and crash resume.
- **Open language registry** - A `LanguageId`-keyed adapter registry replaces closed per-language enum dispatch.

See the full specification at `docs/psps/records/000009-governed-agentic-platform.rst`.

7.4.2.5 PSP-10: Search and Prompt Programs

PSP-10 extends the governed platform with bounded search trajectories and model-conditioned prompt programs. It introduces:

- **Typed prompt sections** - Prompt libraries compiled at build time from versioned section files, with program and invocation digests recorded in the ledger.
- **Bounded search forests** - On gate failure, isolated branches run against the accepted root under pre-reserved budgets, with exact no-good learning.
- **Resident-context paging** - The conversation becomes content-addressed pages; evicted pages leave tombstones and remain recallable via `context_recall`.
- **Multi-node dispatch** - Concurrent work-graph nodes behind `--max-parallel-nodes`, staged into a content-addressed root and promoted through one global integration gate.

See the full specification at `docs/psps/records/000010-bounded-search-trajectories.rst`.

7.4.2.6 PSP-11: Domain-Neutral Runtime Inversion & Multi-Actor Supervisor

PSP-11 specifies the inversion of the governed candidate engine into `perspt-sdk`, a multi-actor supervisor over shared durable stores, and replayable human interaction gates.

See the full specification at `docs/psps/records/000011-domain-neutral-runtime-inversion.rst`.

Writing a PSP

A developer wishing to propose an architectural modification must adhere to the following protocol:

1. Create a branch from the repository root.
2. Run `guidedog dsc new "Proposal title" --dir=docs/psps` to create an unnumbered record in `docs/psps/records/`.
3. Define the Abstract, Motivation, Specification, Rationale, and Reference Implementation.
4. Run `guidedog dsc index --dir=docs/psps && guidedog dsc check --dir=docs/psps` and submit the proposal for review.
5. Maintainers assign a number and change the record's state through reviewed pull requests, using `guidedog dsc promote` and `guidedog dsc state`.

How-To Guides

Task-oriented recipes for common operations.

Guide	Summary
Agent Options Reference	Full reference for `perspt agent` flags.
Configuration	Config files, env vars, and precedence.
Set Up Providers	Set up each LLM provider.
Security and Policy Rules	Starlark policies, sandboxing, and security.
Dashboard Setup	Configure and launch the web dashboard.

Agent Options Reference

`perspt agent` runs the governed PSP-9 tool loop, extended by PSP-10 with bounded search forests, resident-context paging, prompt programs, and multi-node dispatch. The CLI exposes only controls consumed by that runtime.

Core Options

Flag	Description
`<TASK>`	Task description.
`-w, --workdir <DIR>`	Workspace root.
`-y, --yes`	Approve final promotion within already-granted authority. It does not grant network, shell, policy, or out-of-workspace effects. A non-interactive run requires `--yes`: promotion approval cannot be prompted without a terminal. `--max-parallel-nodes` above `1` also requires it.
`--model <PROVIDER::MODEL>`	Override the primary actuator route.
`--domain <DOMAIN>`	Domain package to run (e.g. `coding`, `research`); default: detect.
`--exploration-only`	Run only the read-only exploration phase: deterministic map plus an interactive explorer tool loop; nothing is mutated or promoted.
`--allow-dependency-mutation`	Grant governed dependency mutation (`cargo add`, `uv add`, `npm install`).
`--allow-unisolated`	Explicitly permit verifier, inspection, and LSP child processes to run without an OS sandbox. Intended for acknowledged native Windows use or an embedder that isolates the complete Perspt process. Candidate, capability, gate, ledger, and approval con-

Flag	Description
	trols remain active, but the child has the host user's filesystem and network authority.
`--allow-experimental-prompts`	Substitute validated `[prompts]` bundle sections live (Gate AE: experimental until a change record passes paired evaluation).
`--db-path <PATH>`	Path to the PSP-9 ledger database (defaults to the platform data directory).
`--output-summary <FILE>`	Write the terminal PSP-9 run summary as JSON.
`--persistent-grants`	Store signed grant intent. Resume always rechecks the durable authority epoch and must mint a fresh live capability.

Finite Harness

Flag	Description
`--rho-gate <VALUE>`	Required measured energy descent per accepted checkpoint (default `0.5`).
`--max-turns <N>`	Maximum model turns per node (default `12`).
`--max-calls-per-turn <N>`	Maximum direct and nested tool calls per turn (default `8`).
`--rejection-budget <N>`	Shared non-descending and recovery budget (default `4`).
`--max-parallel <N>`	Maximum independent compiler, test, and lint sensors (default `4`).
`--max-parallel-nodes <N>`	Concurrent work-graph nodes (default `1`). Values above `1` require `--yes`.

Model Portfolio

Flag	Description
`--actuator-model <M>`	Primary implementation route.
`--explorer-model <M>`	Optional read-only repository exploration model.
`--adjudicator-model <M>`	Optional no-tool validator of the realized diff. Its verdict is an uncalibrated conjunctive veto, never an energy measurement.
`--fallback-model <M>`	Add an ordered sticky actuator failover route. Repeat the option to add more routes. Role-specific models are never silently reused as actuator fallbacks.

Dashboard

`--dashboard` starts the read-only monitoring server alongside the agent. `--dashboard-port <PORT>` selects its port (default `3000`).

Examples

```
persptagent -y -w /tmp/proj "Create a Python calculator"

persptagent \
  --actuator-model vertex::gemini-3.5-flash-lite \
  --explorer-model vertex::gemini-3.5-flash-lite \
  --fallback-model vertex::gemini-3.5-flash \
  --rho-gate 0.25 \
  --max-turns 16 \
  --max-calls-per-turn 8 \
  --rejection-budget 5 \
  --max-parallel 4 \
  -w ./project "Build a web server"

persptagent --dashboard --dashboard-port 8080 -w ./myapp "Add unit tests"

# Native Windows reduced-isolation mode
persptagent --allow-unisolated -w . "Fix the parser"
```

Configuration

Config File Location

Perspt searches for configuration in this order:

1. `--config <path>` (CLI flag)
2. `~/.config/perspt/config.toml` (Linux), `~/Library/Application Support/perspt/config.toml` (macOS), `%APPDATA%\perspt\config.toml` (Windows)
3. Environment variables
4. Auto-detection

Config File Format

The file is TOML. All fields are optional. `provider` accepts the aliases `provider_type` and `default_provider`; `model` accepts the alias `default_model`.

```
provider = "gemini"
model = "gemini-3.5-flash"
api_key = "AIza..."

# Optional endpoint override for OpenAI-compatible / local / proxy servers
# base_url = "http://localhost:8000/v1"

# Optional per-role routes for `perspt agent`; values stay fully
# qualified as provider::model
# [models]
# architect = "openai::gpt-5.5"
# actuator = "openai::gpt-5-mini"
# verifier = "openai::gpt-5-mini"
# speculator = "openai::gpt-5-mini"
# adjudicator = "openai::gpt-5.5"
```

Agent Runtime Sections

`perspt agent` reads further optional TOML tables, each validated at startup.

`[providers.<id>]` lets several provider credentials coexist in one process. `api_key_env` names an environment variable; Vertex entries may omit a static key and use the per-request ADC token resolver:

```
[providers.anthropic]
api_key_env = "ANTHROPIC_API_KEY"

[providers.local]
adapter = "ollama"
base_url = "http://localhost:11434"
```

`[exploration]` bounds the PSP-10 search forest: sequential eager branches with a hard branch cap of 3. The former `[ensemble]` section was removed by PSP-10 - a present block fails validation with a migration error pointing here:

```
[exploration]
initial_branches = 1
max_branches = 3
distinct_family = true
```

`[[external_tools]]` declares shared latest-only MCP 2026-07-28 servers. The official Rust SDK supplies tools, resources, prompts, completions, roots, sampling, elicitation, subscriptions, MRTR, and tasks. Secret values are never stored here: environment maps contain destination names and source variable names. Omitted `modes` means agent-only; chat must be explicit. Every allowed remote tool needs a local policy, including a schema-valid footprint:

```
[[external_tools]]
id = "docs"
transport = "stdio"
command = ["company-docs-mcp", "--stdio"]
modes = ["agent", "chat"]
env_from_env = { DOCS_TOKEN = "COMPANY_DOCS_TOKEN" }
roots = [{ uri = "file:///absolute/path/to/project", name = "Project" }]
sampling = true
max_sampling_tokens = 4096
elicitation = true
subscriptions = true
tasks = true
max_task_wait_ms = 300000

[external_tools.tools.search]
effect = "search"
risk = "low"
footprint = { selectors = [{ kind = "scoped_argument", family = "company-docs", field =
"query", access = "read" }] }
```

The policy key must equal the remote tool name and its `field` must exist in the advertised input schema. In chat, `/mcp` explains failures and lists the admitted namespaced operations. Older protocol versions have no fallback. See Model Context Protocol (MCP) for both transports, every capability and bound, agent/chat behavior, clipboard paste, and security guidance.

`[prompts]` pins external prompt replacement bundle directories and the paired activation bounds (Gate AE):

```
[prompts]
bundles = ["./prompt-bundles/coding"]
activation_min_tasks = 30
```

`[context]` sets reserves and working-set bounds for the paged resident context; every configured reserve must be positive:

```
[context]
working_set_turns = 8
output_reserve_tokens = 4096
```

`[verification]` sets the test-evidence policy, optional `format` sensor, process-isolation fallback, and per-stage timeouts. `evolving` is the iterative-development default; `backward-compatible` additionally runs recognized historical test files; `external-oracle` additionally runs a protected overlay configured under `[verification.external_oracle]`:

```
[verification]
test_policy = "evolving"
allow_unisolated = false
require_format = true
stage_timeout_secs = 180
```

Keep `allow_unisolated` false for fail-closed execution. Native Windows users may set it true to run compilers, tests, inspection commands, and LSP servers without an OS sandbox; Perspt retains its agent-level gates but child processes have the host user's authority.

Environment Variables

Variable	Provider	Priority
`VERTEX_PROJECT_ID`	Vertex AI	Highest
`GEMINI_API_KEY`	Gemini	2
`OPENAI_API_KEY`	OpenAI	3
`ANTHROPIC_API_KEY`	Anthropic	4
`GROQ_API_KEY`	Groq	5
`COHERE_API_KEY`	Cohere	6
`XAI_API_KEY`	xAI	7
`DEEPSEEK_API_KEY`	DeepSeek	8
(none)	Ollama	Fallback

> **Note**
> When multiple keys are set, the highest-priority provider is used. Override with the `provider` field in the config or by setting only the key you want detected.

CLI Flag and Config Override

Manage the config file from the CLI:

```
# Show the effective config (api_key masked)
perspt config --show

# Set values (structured TOML write)
perspt config --set provider=openai
perspt config --set default_model=gpt-5.5

# Edit in $EDITOR
perspt config --edit

# Override the model per run
perspt chat --model gpt-5.5

# Use a specific config file
perspt --config /path/to/config.toml chat
```

Logging Configuration

```
# Default: error-level logging only (avoids TUI noise)
perspt

# Enable debug logging with RUST_LOG
RUST_LOG=debug perspt simple-chat

# Audit a finished agent run (deterministic, credential-free replay)
perspt replay <SESSION_ID>
```

Set Up Providers

Choose the adapter, set its credentials, then select an explicit route. The Providers table lists all supported adapters and their environment variables. Replace `<model-id>` in these examples with a model available to the account.

Cloud API keys

Set the key for the provider we intend to use:

```
export GEMINI_API_KEY="<api-key>"
perspt chat --model 'gemini::<model-id>'

# Alternatives
export OPENAI_API_KEY="<api-key>"
perspt chat --model 'openai::<model-id>'

export ANTHROPIC_API_KEY="<api-key>"
perspt chat --model 'anthropic::<model-id>'
```

The same route syntax applies to `groq`, `cohere`, `xai`, and `deepseek` with their corresponding credentials. Model names in examples elsewhere in the book illustrate configuration; they do not establish current account access or agent tool support.

Vertex AI

Configure local Application Default Credentials and the project:

```
gcloud auth application-default login
export VERTEX_PROJECT_ID="<project-id>"
export VERTEX_LOCATION="global"
perspt chat --model 'vertex::<model-id>'
```

Perspt resolves tokens per request through ADC. An optional `VERTEX_API_KEY` supplies an explicit bearer-token override. The location variable is `VERTEX_LOCATION`.

Ollama

Start `ollama serve` in another terminal, then install and select a model:

```
ollama pull llama3.2
perspt chat --model ollama::llama3.2
```

No cloud key is needed. An explicit prefix keeps this route local even when cloud keys are present. Use `ollama list` to see installed model IDs. Agent mode additionally needs structured tool-calling support.

Several providers in one agent run

```
[providers.cloud]
adapter = "openai"
api_key_env = "OPENAI_API_KEY"

[providers.local]
adapter = "ollama"
base_url = "http://localhost:11434"

[models]
actuator = "cloud::<coding-model-id>"
speculator = "local::<explorer-model-id>"
```

The `speculator` entry supplies the explorer fallback route. We replace both model placeholders with available models and can override roles per run with `--actuator-model` and `--explorer-model`.

Check the connection

```
perspt providers
perspt providers --probe
```

The second command makes live model requests. A model that supports ordinary chat may still lack the structured operations needed by agent mode.

Security and Policy Rules

Perspt provides multiple layers of security for agent mode.

Starlark Policies (perspt-policy)

The `perspt-policy` crate evaluates Starlark scripts against every command line the agent proposes. Policies can:

- **Deny shell commands** - Block dangerous commands (`rm -rf`, etc.)
- **Require confirmation** - Gate risky commands behind a user prompt

Each `.star` file in the rules directory (`rules/` under the config directory) must define `evaluate`, a function that receives the full command line and returns `"allow"`, `"prompt"`, or `"deny"` (a bool is also accepted). The builtins are `matches_pattern(command, pattern)` (substring match) and `log_policy(message)`:

```
# Example Starlark policy
def evaluate(command):
    for pattern in ["rm -rf", "sudo", "chmod 777"]:
        if matches_pattern(command, pattern):
            log_policy("blocked: " + command)
            return "deny"
    if matches_pattern(command, "git push"):
        return "prompt"
    return "allow"
```

When several files are loaded, the strictest decision wins, and a policy that fails to evaluate denies the command.

Sandbox Isolation (perspt-sandbox)

The `perspt-sandbox` crate provides bounded command execution and OS-backed isolation where a backend is registered:

- **Filesystem scoping** - Commands run in a restricted view of the filesystem
- **Process limits** - Wall-clock timeout and bounded captured output
- **Network control** - Optional network access restriction

Configuration:

```
# Agent with sandbox enabled
perspt agent -w ./project "Task"

# The sandbox restricts commands to the working directory
```

Linux uses Bubblewrap and macOS uses `sandbox-exec`. Native Windows fails closed by default. `--allow-unisolated` is an explicit reduced-isolation override: the agent-level capability and verification machinery stays active, but child processes retain the host user's filesystem and network authority.

Authority Model

PSP-9 authority is capability-based:

- Live capabilities are minted from grants at session start and are bound to the session's authority epoch
- `perspt abort` revokes the epoch durably: in-flight promotions and stale resume intents are refused, and the workspace is untouched
- `--allow-dependency-mutation` is the explicit grant for governed dependency mutation (`cargo add`, `uv add`, `npm install`)
- `--persistent-grants` persists signed grant intent only, never live capabilities: resume must mint fresh ones

Footprint Scheduling

The multi-node dispatcher schedules by write footprints:

- Each node declares the files it writes (a node that declares none holds an opaque whole-workspace footprint)
- Nodes with conflicting footprints never run concurrently
- Promotion is single-flight: it runs only in the dispatcher's completion arm

This prevents conflicting edits and provides a clear audit trail.

Review Modal

In interactive mode (without `--yes`), every node's changes must be manually approved. The review modal shows:

- Full diff of all changes
- Verification results (syntax, build, test, lint), the measured energy, and a degraded flag with reasons when any sensor was skipped
- Options to reject, correct, or edit

For security-sensitive projects, always use interactive mode.

Merkle Ledger

Every committed change is recorded in a content-addressed Merkle tree stored in DuckDB. This provides:

- **Tamper detection** - Hash chain integrity
- **Full auditability** - Every node's input/output is recorded
- **Rollback capability** - Undo a session's newest completed promotion, unless its files changed afterward

```
perspt ledger --recent
perspt ledger --stats
```

Best Practices

1. **Use interactive mode for production code** - Always review diffs
2. **Replay sessions** - `perspt replay <SESSION_ID>` is a deterministic, credential-free audit replay
3. **Use workspace directories** - `-w <dir>` scopes agent writes
4. **Label delayed audit samples** - `perspt audit` lists pending samples and records safe/unsafe labels
5. **Inspect prompt and context provenance** - `perspt prompts explain-session` and `perspt context explain-turn` show what a session compiled and compacted
6. **Review ledger after headless runs** - `perspt ledger --recent`

Dashboard Setup

Configure and launch the Perspt web dashboard for monitoring agent sessions.

Locate the Config File

The Perspt configuration file lives in your platform's config directory:

- **Linux**: `~/.config/perspt/config.toml`
- **macOS**: `~/Library/Application Support/perspt/config.toml`

If migrating from the legacy `~/.perspt/` layout, Perspt will warn and fall back to the old location automatically.

Understand the Access Model

The dashboard binds to `127.0.0.1` only and runs unauthenticated. It is a read-only view of the local session database, suitable for localhost-only use; do not forward the port to other machines.

Change the Default Port

The default port is `3000`. Override via CLI flag:

```
perspt dashboard --port 8080
```

Build from Source

```
cargo build --locked -p perspt-cli --release --features bundled
```

The dashboard is a library embedded in the `perspt` CLI, not a separate binary. Use `perspt agent --dashboard` for live monitoring and `perspt dashboard` after the writer exits.

Launch Locally

```
perspt dashboard
```

The server binds to `127.0.0.1:3000` by default. Open `http://localhost:3000` in your browser.

Point to a Specific Database

By default, the dashboard reads from the platform data directory (`~/.local/share/perspt/perspt.db` on Linux). To use a different database file:

```
perspt dashboard --db-path /path/to/perspt.db
```

Embed in Agent Mode

Instead of running the dashboard as a separate process, you can start it alongside the agent using the `--dashboard` flag:

```
perspt agent --dashboard "Create a REST API"
```

This spawns the dashboard as a background task within the agent process. The embedded server shares the agent's store connection. A separate dashboard process can conflict with an active writer's DuckDB lock.

To use a custom port:

```
perspt agent --dashboard --dashboard-port 8080 "Add tests"
```

The embedded dashboard provides the same interface as the standalone `perspt dashboard` command: overview, session detail, topology, backlog, energy, decisions, and governance pages. It stops automatically when the agent process exits.

> **Note**
> The embedded dashboard is especially useful for headless CI runs (`--yes --dashboard`) where you want a browser view of progress without a separate terminal.

Configuration Guide

Perspt supports zero-config auto-detection, environment variables, a TOML config file, and command-line flags. They are applied in this priority order (highest first):

1. **Command-line arguments**
2. **Configuration file** (`config.toml`)
3. **Environment variables**
4. **Auto provider detection**
5. **Built-in defaults**

Automatic Provider Detection

Set any supported API key environment variable and run `perspt` with no arguments:

Prior-ity	Provider	Environment Variable	Default Model
1	Google Vertex AI	`VERTEX_PROJECT_ID`	`vertex::gemini-2.5-flash`
2	Google Gemini	`GEMINI_API_KEY`	`gemini-3.1-flash-lite-preview`
3	OpenAI	`OPENAI_API_KEY`	`gpt-4o-mini`
4	Anthropic	`ANTHROPIC_API_KEY`	`claude-3-5-sonnet-20241022`
5	Groq	`GROQ_API_KEY`	`llama-3.1-8b-instant`
6	Cohere	`COHERE_API_KEY`	`command-r-plus`
7	XAI	`XAI_API_KEY`	`grok-beta`
8	DeepSeek	`DEEPSEEK_API_KEY`	`deepseek-chat`
9	Ollama	*(none - auto-detected)*	`llama3.2`

```
# Example: set a key and run
export GEMINI_API_KEY="your-key"
perspt               # auto-detects Gemini, uses gemini-3.1-flash-lite-preview
perspt chat --model gemini-3.1-pro   # override model
```

Vertex AI authentication

The Vertex adapter reads `VERTEX_PROJECT_ID` and an optional `VERTEX_LOCATION` (default `global`). Without an explicit bearer-token override, Perspt obtains tokens through Application Default Credentials and resolves them per request. The `gcloud` CLI can configure local ADC:

```
gcloud auth application-default login
export VERTEX_PROJECT_ID="<project-id>"
export VERTEX_LOCATION="global"
perspt chat --model 'vertex::<model-id>'
```

Replace `<model-id>` with a model enabled in the project. `VERTEX_API_KEY` is an optional bearer-token override, not a required static API key. An explicitly supplied token must be refreshed by its owner when it expires.

Supported Models & Naming Conventions

In version 0.6.6, a fully qualified model name resolves its own provider: a `provider::model` prefix (`openai`, `anthropic`, `gemini`/`google`, `vertex`, `groq`, `cohere`, `ollama`, `xai`, or `deepseek`) selects that provider directly, and the model part is passed through verbatim. A bare model name uses the configured or auto-detected provider instead.

```
perspt chat --model vertex::gemini-3.1-pro   # provider from the namespace
perspt chat --model gemini-3.1-pro           # provider from config/detection
```

Configuration File

Perspt reads `config.toml` from the platform config directory, or from an explicit path:

1. Path given via `perspt --config <PATH>`
2. `~/.config/perspt/config.toml` (Linux)
3. `~/Library/Application Support/perspt/config.toml` (macOS)
4. `%APPDATA%\perspt\config.toml` (Windows)

All fields are optional. `provider` accepts the aliases `provider_type` and `default_provider`; `model` accepts the alias `default_model`.

Minimal example:

```
provider = "gemini"
model = "gemini-3.1-pro"
api_key = "your-key"
```

Full example:

```
provider = "openai"
model = "phi-4-npu-ov"
api_key = "your-key"
# Override the endpoint for OpenAI-compatible / local / proxy servers
base_url = "http://localhost:8000/v1"

# Optional per-role routes for `perspt agent`, as fully qualified
# `provider::model` values. When present, the [models] table takes
# precedence over the flat *_model fields.
[models]
architect = "openai::gpt-5.5"
```

```
actuator = "openai::gpt-5-mini"
verifier = "openai::gpt-5-mini"
speculator = "openai::gpt-5-mini"
adjudicator = "openai::gpt-5.5"
```

> **Note**
> `base_url` overrides the endpoint for the active provider. This is useful for Azure OpenAI, proxy servers, local OpenAI-compatible servers, or self-hosted endpoints. You can also set the provider's `*_BASE_URL` environment variable (`OPENAI_BASE_URL`, `OLLAMA_BASE_URL`, …).

> **Note**
> Custom model names that genai does not recognize (for example `phi-4-npu-ov`) are routed to the configured `provider`. You can also target an adapter inline with namespacing, e.g. `openai::phi-4-npu-ov`.

Agent Configuration Blocks

The agent runtime reads four optional TOML blocks. All fields are optional; invalid values fail at startup.

Bounded search (`[exploration]`):

```
[exploration]
initial_branches = 1        # Branches opened before any expansion trigger
max_branches = 3            # Branch identities per forest (hard cap 3)
distinct_family = true      # Prefer a distinct model family on expansion
max_workspace_files = 2048  # Cumulative eager-copy file reservation cap
max_workspace_bytes = 134217728  # Cumulative eager-copy byte reservation cap
```

Prompt bundles (`[prompts]`):

```
[prompts]
bundles = ["./prompt-bundles/tuned"]  # External bundles, pinned at session start
activation_min_tasks = 30    # Minimum paired activation tasks (floor 30; raise-only)
noninferiority_margin = 0.05 # Noninferiority margin epsilon in [0, 0.05]
```

Resident-context reserves (`[context]`):

```
[context]
working_set_turns = 8          # Verbatim turns kept in the working set
synopsis_frame_tokens = 2048   # Token reserve for the synopsis frame
output_reserve_tokens = 8192   # Token reserve for model output
guard_reserve_tokens = 1024    # Guard reserve against overflow
```

Verification acceptance and test evidence (`[verification]`):

```
[verification]
test_policy = "evolving"  # Default: resulting code, tests, and configuration
```

```
allow_unisolated = false # Explicit reduced-isolation process execution
require_format = false    # Declare the plugin format stage as an acceptance sensor
stage_timeout_secs = 180  # Wall-clock limit for every governed verifier stage
test_timeout_secs = 300   # Per-stage override (also syntax/build/lint/format)
```

The test policy defines which test evidence must pass in addition to the coding domain's required syntax and build stages:

`allow_unisolated` is `false` by default. Enable it only when the entire Perspt process is isolated externally or when native Windows coding is more important than OS process isolation. It permits verifiers, read-only inspection commands, and language servers to run with the host user's authority; it does not turn those processes into a sandbox. The CLI also offers the one-run `--allow-unisolated` switch. A config value is preferable when an interrupted native Windows session may need `perspt resume`.

Policy	Acceptance behavior
`evolving`	The default for iterative development. Perspt runs the resulting implementation, resulting project tests, and resulting configuration. Existing tests may be corrected, replaced, or removed when the task intentionally changes their contract; newly written tests participate in the same gate.
`backward-compatible`	Runs the resulting suite and a second regression view in which recognized pre-existing test files are restored. Select it only when the task promises compatibility with those historical expectations.
`external-oracle`	Runs the resulting suite and then overlays separately protected acceptance material onto a private candidate copy. Its configured command is an additional required test-stage verdict. This is intended for CI, security fixes, contractual acceptance, and other work with an independently maintained suite.

An external oracle is explicit and fail-closed:

```
[verification]
test_policy = "external-oracle"
test_timeout_secs = 600

[verification.external_oracle]
path = "/srv/project-acceptance"       # or relative to the agent workspace
command = "cargo test --test acceptance"
```

The directory is copied over a private copy of the candidate only at the measurement boundary. It can contain tests, manifests, runner configuration, or harness scripts, and none of those files are promoted. Keep the directory outside the workspace when its contents must be withheld from the actuator. Configuring the table without `test_policy = "external-oracle"`, or selecting that policy without the table, is a startup error rather than silently ignored configuration.

`evolving` does not claim that model-authored tests are independent proof of semantic correctness. It means the configured project verification suite passed for the new contract. Use protected acceptance evidence when an independent semantic oracle is required.

> **Note**
> The `[ensemble]` section was removed by PSP-10 and is now a hard startup error: the proposal ensemble is replaced by the bounded search forest, and the error message points to `[exploration]`.

Command-Line Flags

Global flags apply to all subcommands:

Flag	Description
`-v, --verbose`	Enable verbose logging
`-c, --config <PATH>`	Path to configuration file
`-h, --help`	Show help
`-V, --version`	Show version

Chat-specific:

```
perspt chat --model <MODEL>
```

Agent-specific (see Agent Options Reference for the full list):

```
perspt agent [OPTIONS] "<TASK>"

# Key options:
--model <MODEL>               # Primary actuator alias
--actuator-model <MODEL>      # Governed tool-call route
--explorer-model <MODEL>      # Optional cheap no-tool exploration
--adjudicator-model <MODEL>   # Optional no-tool diff veto
--fallback-model <MODEL>      # Repeatable sticky actuator fallback
-w, --workdir <DIR>           # Working directory
-y, --yes                     # Auto-approve (headless)
--rho-gate <V>                # Required measured descent
--max-turns <N>               # Finite model-turn budget
--max-calls-per-turn <N>      # Direct and nested call budget
--rejection-budget <N>        # Shared recovery/rejection budget
--max-parallel <N>            # Parallel verifier sensors
--max-parallel-nodes <N>      # Concurrent work-graph nodes (>1 needs --yes)
--exploration-only            # Read-only exploration; nothing mutated
--allow-experimental-prompts # Substitute validated [prompts] bundles live
--domain <ID>                 # Domain package (coding, research); default: detect
--allow-dependency-mutation   # Grant governed dependency mutation
--persistent-grants           # Sign durable grant intent
--db-path <PATH>              # PSP-9 ledger database path
--dashboard                   # Start the web dashboard alongside the agent
--dashboard-port <N>          # Embedded dashboard port (default 3000)
--output-summary <FILE>       # Terminal session summary as JSON
```

Manage configuration interactively:

```
perspt config --show    # Print the effective config (api_key masked)
perspt config --edit    # Open in $EDITOR
```

```
perspt config --set provider=gemini
perspt config --set default_model=gemini-3.1-pro
```

Initialize project-level configuration:

```
perspt init --memory --rules
```

Dashboard Configuration

The `perspt dashboard` subcommand accepts these options:

Flag	Default	Description
`--port`	`3000`	HTTP port for the dashboard server
`--db-path`	Platform default	Path to the DuckDB database file

The dashboard opens the database in **read-only** mode and never writes to it. The server always binds to `127.0.0.1`; cookies are set without the `Secure` flag so plain HTTP works on localhost.

Reference

Precise, authoritative reference for Perspt's interface.

Reference	Scope
CLI Reference	Complete command-line interface documentation.
Key Bindings	TUI keyboard shortcuts and review modal controls.
Advanced Troubleshooting	Advanced diagnostics and known issues.

CLI Reference

```
perspt [OPTIONS] [COMMAND]
```

Global Options

Flag	Description
`-v, --verbose`	Enable verbose logging
`-c, --config <PATH>`	Path to the TOML configuration file
`-h, --help`	Print help
`-V, --version`	Print version

Commands

10.1.2.1 `chat` (default)

Launch the TUI chat interface.

```
perspt chat [--model MODEL]
```

10.1.2.2 `simple-chat`

Launch the plain-text CLI chat.

```
perspt simple-chat [--model MODEL] [--log-file FILE]
```

This command takes no positional prompt. Piped input sends one message per non-empty line; stdout includes session messages as well as responses.

10.1.2.3 `dashboard`

Launch the real-time web monitoring dashboard.

```
perspt dashboard [--port PORT] [--db-path PATH]
```

- `--port` - HTTP port (default `3000`)
- `--db-path` - Path to DuckDB database file (default: platform data directory)

The server always binds to `127.0.0.1`.

See Dashboard Setup for configuration details.

10.1.2.4 `agent`

Run the SRBN autonomous coding agent.

```
perspt agent [OPTIONS] <TASK>
```

- `--dashboard` - Start the web monitoring dashboard alongside the agent
- `--dashboard-port <PORT>` - Port for the embedded dashboard (default `3000`)
- `--allow-unisolated` - Explicit reduced-isolation process execution; required for native Windows coding until a sandbox backend exists

See Agent Options Reference for full agent options.

10.1.2.5 `init`

Initialize project memory and policy rules.

```
perspt init [--memory] [--rules]
```

- `--memory` - Create the `PERSPT.md` project memory file
- `--rules` - Create default Starlark policy rules

10.1.2.6 `config`

View or edit Perspt configuration.

```
perspt config [--show] [--set KEY=VALUE] [--edit]
```

- `--show` - Print the effective config (`api_key` masked)
- `--set KEY=VALUE` - Set a value with a structured TOML write
- `--edit` - Open the config file in `$EDITOR`

10.1.2.7 `ledger`

Query the Merkle ledger.

```
perspt ledger [--recent] [--stats] [--rollback SESSION_PREFIX]
```

- `--recent` - Show recent commits
- `--stats` - Show ledger statistics
- `--rollback SESSION_PREFIX` - Undo the session's newest completed promotion and label it unsafe

10.1.2.8 `status`

Show current session status.

```
perspt status
```

Displays: ledger event count, ledger head, measurement count, last energy, last gate decision, denial count, search forest/branch/no-good counters, validator verdicts, the Φ(W) conditional-capacity diagnostic, validator-independence statistics, pending external effects, and a `perspt replay` hint.

10.1.2.9 `abort`

Abort a PSP-9 session by revoking its authority epoch.

```
perspt abort [--force] [SESSION_ID]
```

- `-f, --force` - Force abort without confirmation
- `SESSION_ID` - Session to abort (defaults to the newest running PSP-9 session)

10.1.2.10 `resume`

Resume an interrupted session.

```
perspt resume [SESSION_ID] [--last] [--db-path PATH]
```

- `SESSION_ID` - Session ID to resume
- `--last` - Resume the most recent session
- `--db-path` - Database file to inspect (defaults to the standard store)

Displays the session id, task, working directory, and status before resuming, and verifies the ledger chain is valid.

10.1.2.11 `audit`

Delayed audit labels and conformal activation (PSP-9).

```
perspt audit [SAMPLE] [--safe] [--unsafe]
```

- `SAMPLE` - Sample id (or unique prefix) to label; omit to list pending samples
- `--safe` - Label the sample as safe
- `--unsafe` - Label the sample as unsafe

10.1.2.12 `providers`

Print the provider capability matrix (PSP-9).

```
perspt providers [--probe]
```

- `--probe` - Run live behavioral probes against every configured model route

10.1.2.13 `replay`

Audit recorded session events without credentials, model calls, or rerunning project checks.

```
perspt replay <SESSION_ID> [--db-path PATH]
```

- `SESSION_ID` - The session id to replay

- `--db-path` - Database file to inspect (defaults to the standard store)

10.1.2.14 `db`

Inspect and repair the local DuckDB store.

```
perspt db repair --db-path PATH [--discard-wal]
```

- `repair` - Quarantine a poisoned WAL after making durable backups
- `--db-path` - Database file to repair
- `--discard-wal` - Explicitly authorize WAL quarantine; the WAL is never deleted

10.1.2.15 `prompts`

Inspect and maintain the compiled prompt section libraries (PSP-10).

```
perspt prompts list
perspt prompts render <STAGE>
perspt prompts lint [--bundle DIR]
perspt prompts manifest <DIR>
perspt prompts explain-session --db-path PATH <SESSION_ID>
```

- `list` - List every compiled section: id, version, stage, role, hash
- `render <STAGE>` - Compose one stage with fixture variables and print it
- `lint` - Run the codegen validation list over an external bundle directory
- `manifest <DIR>` - Regenerate a prompt library's committed `manifest.toml` (explicit)
- `explain-session` - Show the programs a session actually compiled, with digests

10.1.2.16 `context`

Explain a session's recorded resident-context events (PSP-10).

```
perspt context explain-turn --db-path PATH <SESSION_ID>
```

- `explain-turn` - Show a session's recorded context events (compactions, refusals)

10.1.2.17 `benchmark` (optional)

Build `perspt-cli` with the Cargo feature `benchmark` to expose the separate evaluation runner. It is absent from the default CLI build.

```
perspt benchmark validate
perspt --config config.local.toml benchmark run --suite smoke --output report.json
perspt benchmark aggregate report-a.json report-b.json
```

`validate` is credential-free. `run` is explicit and credentialed; its `smoke`, `adaptive`, and `full` suites use production role resolution from the selected Perspt configuration and record the configured topology. `run --tasks <N>` overrides the suite's task count. Coding verification stays deterministic, so a configured verifier route is provenance rather than a model call. Model names and family labels are not benchmark arguments. The benchmark is manual-only and feature-gated — the `perspt-benchmark` crate is a separate optional crate — and a benchmark run is never part of normal validation or CI.

Key Bindings

TUI Chat Mode

Key	Action
Enter	Send message
Ctrl+J / **Ctrl+Enter**	Insert newline in input
Page Up	Scroll chat up 10 lines
Page Down	Scroll chat down 10 lines
Ctrl+Up / **Shift+Up**	Scroll up one line
Ctrl+Down / **Shift+Down**	Scroll down one line
Up / **Down**	Traverse prompt history (or move between input lines)
Home	Move input cursor to beginning of line
End	Move input cursor to end of line
Ctrl+R	Toggle reasoning display
Ctrl+C / **Ctrl+Q**	Quit

Review Modal (Agent Mode)

Key	Action
y	Approve node changes
n	Reject node changes
c	Send correction feedback
e	Open files in editor
d	Toggle diff view
s	Skip node
Left / **Right**	Move action selection
Enter	Confirm selected action
Esc	Dismiss modal

Agent Mode

Key	Action
q	Quit
p	Pause/resume the agent
Tab / **Shift+Tab**	Cycle tabs

Key	Action
1 / 2 / 3	Jump to Dashboard / Tasks / Diff tab
Up / k	Navigate up
Down / j	Navigate down
Page Up / Page Down	Page navigation
Space / Enter	Select item
a	Open the approval modal

Input Editor

Key	Action
Left / Right	Move cursor
Ctrl+B / Ctrl+F	Move cursor left / right
Ctrl+A	Move to beginning of line
Ctrl+E	Move to end of line
Ctrl+W	Delete word backward
Ctrl+U	Delete to beginning of line
Ctrl+K	Delete to end of line
Backspace	Delete character backward
Delete	Delete character forward
Ctrl+D / Ctrl+H	Delete / Backspace character

Advanced Troubleshooting

Diagnostic Commands

```
# Session status with per-node details
perspt status

# Deterministic, credential-free audit replay of a session
perspt replay <SESSION_ID>

# Ledger integrity check
perspt ledger --stats

# Enable verbose logging
RUST_LOG=debug perspt simple-chat 2>debug.log
```

Common Error Patterns

10.3.2.1 Missing API key

When no provider is configured, Perspt detects one from environment keys in a fixed order: Vertex AI project settings first, then `GEMINI_API_KEY`, `OPENAI_API_KEY`, `ANTHROPIC_API_KEY`, `GROQ_API_KEY`, `COHERE_API_KEY`, `XAI_API_KEY`, and `DEEPSEEK_API_KEY`, falling back to a local Ollama setup when none are present. Set the provider environment variable or store a key with `perspt config --set api_key=<KEY>`.

10.3.2.2 `Unsupported provider`

The configured `provider` is not a recognized adapter. Supported values: `openai`, `anthropic`, `gemini` (or `google`), `vertex`, `groq`, `cohere`, `ollama`, `xai`, `deepseek`.

10.3.2.3 Provider failover

When a model route fails, the agent fails over to the next configured fallback route. The failover is charged to the shared recovery pool, so repeated provider failures can exhaust the session's rejection budget.

10.3.2.4 Poisoned DuckDB WAL

If the local DuckDB store fails to open because of a poisoned WAL, run `perspt db repair --db-path <PATH> [--discard-wal]`. The repair makes durable backups first and quarantines the WAL; the WAL is never deleted.

Agent-Specific Issues

Footprint conflict:

Two nodes declared overlapping file footprints. The dispatcher's scheduler never runs conflicting nodes concurrently — the later node waits for the conflicting node to finish. If this persists, simplify the task description.

Global integration failure:

Two nodes may pass separately and fail together. Node winners enter a content-addressed staging root instead of the user workspace, and one global verifier gate runs the full domain suite and the test evidence selected by `[verification] test_policy` on the combined state. Only a hard-passing integration root is promoted, atomically; on failure the prior staging root is restored and the user workspace is left byte-identical.

Degraded verification:

When some checks could not run, the review modal marks a node's stability metrics as degraded and lists the reasons. Address the listed reasons to restore full verification.

Terminal Restoration

If Perspt crashes and leaves the terminal in raw mode:

```
reset
# or
stty sane
```

Perspt restores the terminal (raw mode off, leave alternate screen) only on the normal exit path; after a crash, `reset` will fix it.

Performance

Slow response streaming:

1. Check network latency to the provider
2. Try a faster model (e.g., `gemini-3.5-flash`)
3. Use Groq for the fastest inference

High memory usage in agent mode:

1. Large DAGs with many nodes consume more memory
2. Use `--max-turns` and `--max-calls-per-turn` to bound per-node work
3. Use `--rejection-budget` to bound recovery attempts

API & SDK Reference

Perspt is built as a modular 14-crate Rust workspace (Edition 2024, Rust 1.100+). In addition to the `perspt` CLI binary, you can embed the **`perspt`** `"Omni"` facade crate (`AgentBuilder` and `perspt::prelude::*`) or compose **`perspt-sdk`**, **`perspt-coding`**, **`perspt-research`**, and **`perspt-agent`** directly for coding and non-coding agentic workflows.

Integrating the Perspt Omni SDK

Add the root `perspt` facade crate (or individual crates such as `perspt-sdk` and `perspt-coding`) to your `Cargo.toml`:

```toml
[dependencies]
perspt = { git = "https://github.com/eonseed/perspt" }
```

Build and run a governed agent across coding, research (`ResearchDomain`), or custom domains with `AgentBuilder`:

```rust
use perspt::prelude::*;

# async fn run() -> anyhow::Result<()> {
let runtime = AgentBuilder::new(".")
    .research_domain()
    .approval_policy(ApprovalPolicy::Auto)
    .on_ask_user_async(|q| async move { Ok(format!("answered: {q}")) })
    .build()?;
let _summary = runtime.run("verify paper citations".into()).await?;
# Ok(())
# }
```

You can then score verifier diagnostics with the canonical quadratic Lyapunov energy $V(x) = \sum_{e \in E} w_e \parallel r_e(x) \parallel^2$ and evaluate the acceptance gate in your own Rust code:

```rust
use perspt_coding::CodingDomain;
use perspt_sdk::{
    domain::{AgentDomainPackage, DomainScope},
    energy::score_candidate,
    gate::{evaluate_gate, GateDecision},
    residual::ResidualEvent,
};

fn check_candidate(
    scope: &DomainScope,
    residuals: &[ResidualEvent],
    all_required_checks_passed: bool,
    best_v: f64,
) -> perspt_sdk::error::Result<GateDecision> {
    let model = CodingDomain.energy_model(scope);
    let score = score_candidate(&model, residuals)?;
    let hard_pass = all_required_checks_passed
        && score.hard_violations.is_empty()
```

```
        && score.blocked.is_empty();
    evaluate_gate(hard_pass, score.total, best_v, model.rho_gate)
}
```

The caller must run the required verifiers to determine `all_required_checks_passed`. An empty `hard_violations` list only means no declared residual threshold was exceeded. `AcceptedByDescent` permits intermediate progress; it is not delivery success. This example scores evidence and evaluates a gate; it does not run tools, authorize effects, or promote files.

> **Tip**
> For full Rustdoc-generated API documentation across all workspace crates, open the Rustdoc API Reference or run locally:
>
> ```
> cargo doc --open --workspace --no-deps
> ```

Crate	Responsibility
perspt-core	Types, config, LLM provider, events, plugins.
perspt-sdk	Domain-neutral SRBN control plane: energy, gates, ledger.
perspt-coding	Coding domain package: residuals, adapters, barriers.
perspt-research	Research domain package (executable `citation-check` & `correction_packet`).
perspt-agent	Governed candidate runtime, tool loop, verifier, work-graph dispatch.
perspt-tui	Ratatui terminal UI (chat + agent).
perspt-cli	Clap CLI entry point and subcommands.
perspt-store	DuckDB session persistence.
perspt-policy	Starlark policy engine.
perspt-sandbox	Command sandboxing and isolation.
perspt-dashboard	Axum + Askama + HTMX web dashboard.
perspt-prompt-macros	Build-time compiler for typed prompt section libraries.
perspt-benchmark	Optional, feature-gated manual evaluation tooling.
`perspt`	Root `"Omni"` facade crate (`AgentBuilder`, `prelude`, subcrate re-exports).

`perspt-core`

The foundation crate providing all canonical types, configuration, LLM provider integration, event system, and language plugins.

Modules

Module	Description
`types`	Core shared types: NodeClass, ModelTier, EnergyComponents, TaskPlan, PlannedTask, PlannedContract, PlannedTest, TaskType, CommandContract, SensorStatus, StageOutcome, VerifierStrictness, StructuralDigest, SummaryDigest, ArtifactKind, ContextBudget, RestrictionMap, ContextPackage, ContextProvenance
`config`	`Config` - provider/model/API-key basics plus the `[providers]`, `[models]`, `[external_tools]`, `[verification]`, `[exploration]`, `[prompts]`, and `[context]` tables
`events`	`AgentEvent` (33 variants), `AgentAction`, `NodeStatus`, `ActionType`, channel types
`llm_provider`	`GenAIProvider`, `LlmResponse`, `EOT_SIGNAL`, streaming support
`plugin`	`LanguagePlugin` trait, `PythonPlugin`, `RustPlugin`, `JsPlugin`, `PluginRegistry`
`memory`	`ProjectMemory` from `.perspt/memory.toml`
`normalize`	Model and provider name normalization
`prompts`	`PlatformPromptLibrary` - the platform prompt section library, compiled at build time by `perspt-prompt-macros`
`portfolio`	`ModelPortfolio`, `ProviderHandle` - several concurrently live provider handles
`tools_driver`	`CoreToolCall`, `CoreToolSpec`, `CoreTurnOutput` - the `genai` tool-calling driver
`local_command`	`LocalCommand` - provider-independent commands handled entirely by frontends
`path`	Canonical path resolution for artifact paths
`paths`	Centralized platform-aware path helpers (config/data/project tiers)

Key Types

Config - The main configuration struct (excerpt):

```rust
pub struct Config {
    pub provider: Option<String>,
    pub model: Option<String>,
    pub api_key: Option<String>,
    pub base_url: Option<String>,
    // ... per-tier model overrides, plus the [providers], [models],
    // [external_tools], [verification], [exploration], [prompts],
    // and [context] tables
}
```

EnergyComponents - Lyapunov energy decomposition:

```rust
pub struct EnergyComponents {
    pub v_syn: f32,    // Syntax, type, build, and lint rollups
    pub v_str: f32,    // Contract and structural compliance
    pub v_log: f32,    // Test results
    pub v_boot: f32,   // Sensor availability and tool-execution rollups
    pub v_sheaf: f32,  // Cross-node boundary validation
}
```

AgentEvent - 33 lifecycle events:

`TaskStatusChanged`, `PlanGenerated`, `EnergyUpdated`, `Log`, `NodeCompleted`, `ApprovalRequest`, `Complete`, `Error`, `PlanReady`, `NodeSelected`, `FallbackPlanner`, `VerificationComplete`, `BundleApplied`, `SensorFallback`, `DegradedVerification`, `EscalationClassified`, `SheafValidationComplete`, `GraphRewriteApplied`, `BranchCreated`, `InterfaceSealed`, `BranchFlushed`, `DependentUnblocked`, `BranchMerged`, `ContextDegraded`, `ContextBlocked`, `StructuralDependencyMissing`, `ModelFallback`, `ProvenanceDrift`, `ToolReadiness`, `BudgetUpdated`, `PlanRevised`, `FileDeleted`, `FileMoved`

See Architecture for the complete type inventory.

`perspt-sdk`

The domain-neutral SRBN control plane (PSP-8): residual evidence, the canonical quadratic energy, the measured acceptance gate, capabilities, scheduling, and the replay ledger. Domain packages such as `perspt-coding` provide task-specific residual construction, weights, and correction directions on top.

Modules

Module	Description
`residual`	`ResidualEvent` - the measured reason a state is unsafe or incomplete
`energy`	`score_candidate`/`EnergyModel` - the canonical quadratic residual energy $V(x) = \sum_e w_e \parallel r_e \parallel^2$
`gate`	Measured acceptance gate and finite-decision bound
`kernel`	Adapter over the published `srbn` kernel crate
`domain`	`AgentDomainPackage` - the domain-package contract
`capability`	Capability-constrained admissibility kernel: proposals, never unmediated effects
`admissibility`	The five-clause admissibility kernel (PSP-9)
`command`	Typed command IR and governance tiers (no implicit `sh -c`)
`scheduler`	Dependency-aware parallel ready-queue scheduler
`workgraph`	Mutable, revisioned work graph
`ledger`	Append-only, Merkle-chained event stream with audit replay
`checkpoint`	Context checkpoints: compaction as recorded projection, never deletion
`conformal`	Calibrated risk budgets via conformal risk control
`recovery`	The recovery lattice for classified failures
`routing`	Phase-aware model routing (`ModelRoute`, `ModelTier`)
`model`	Provider-neutral model plane identity and contract types (pure data)
`prompt`	The model-conditioned prompt plane: typed sections, composition, digested programs (PSP-10)
`search`	The bounded search plane: forests, branches, witnesses, limits (PSP-10)

Module	Description
`toolset`	The typed tool catalog with effect/risk/footprint contracts
`independence`	Measured validator dependence, never assumed independence
`spectral`	Spectral energy-slope constant μ from the verification graph
`stability`	Analytic stability claims (`NotClaimed` for the coding domain)
`certificate`	`ResidualCertificate` - the honest stop on exhaustion
`goal`	Goal-presence sensor: the verifier that refuses false stability
`observability`	Read-only dashboard/TUI projections over the event ledger
`benchmark`	Mechanism-check benchmark harness and metrics
`canon`	Domain-tagged canonical byte encoding for replayable digests
`scope`	Glob-like scope patterns shared by capabilities and grant policies
`error`	`SdkError`/`Result` - fail-closed numerical contracts

The crate re-exports the published `srbn`, `srbn_ledger`, and `srbn_serde` kernel crates so consumers depend on one SRBN source.

Key Types

evaluate_gate - The measured acceptance gate:

```rust
pub fn evaluate_gate(
    hard_pass: bool,
    candidate_v: f64,
    best_accepted_v: f64,
    rho_gate: f64,
) -> Result<GateDecision>;

pub fn finite_decision_bound(
    baseline_energy: f64,
    rho_gate: f64,
    rejection_budget: u32,
) -> Result<u64>;
```

EnergyScore - The result of scoring a candidate:

```rust
pub struct EnergyScore {
    pub total: f64,                         // Total energy V
    pub components: EnergyComponents,       // Derived rollups
    pub dominant: Vec<ResidualEventRef>,    // Dominant residuals first
    pub hard_violations: Vec<ResidualClass>,
    pub blocked: Vec<ResidualEventRef>,     // Separate admissibility outcomes
}
```

`perspt-coding`

The coding domain package built on `perspt-sdk`: it declares the coding residual schema, supplies the coding energy model (weights, ρ_{gate}, correction budget), and maps dominant residuals into coding correction directions.

Modules

Module	Description
`barrier`	`OperationalSafetyBarrier` - versioned operational safety channels; correctness evidence stays in `V`, never duplicated into `h`
`diag`	The structured verifier plane: adapters preserve as much source structure as each tool provides
`lang`	Language adapters (Rust, Python, TypeScript) as verifier suites
`prompts`	The coding prompt section library; `branch_correct` renders a typed `CorrectionPacket` for the model
`registry`	`CodingAdapterRegistry`/`LanguageId` - open language-adapter registry keyed by id, never enum dispatch
`runtime`	Generic runtime smoke-probe scheme for exercising built artifacts
`symbols`	Symbol extraction for the goal-presence sensor

Key Types

CodingDomain - The `AgentDomainPackage` implementation:

```rust
pub struct CodingDomain;

impl AgentDomainPackage for CodingDomain {
    fn domain_id(&self) -> DomainId;                 // "coding"
    fn detect(&self, workspace: &WorkspaceSnapshot) -> DomainDetection;
    fn residual_schema(&self, scope: &DomainScope) -> ResidualSchema;
    fn energy_model(&self, scope: &DomainScope) -> EnergyModel;
    fn correction_directions(&self, residuals: &[ResidualEvent])
        -> Vec<CorrectionDirection>;
    // ... hard_gate_policy, verifier_suite, safety_barrier
}
```

LanguageId - Open, stable language identifier:

```rust
pub struct LanguageId(pub String);

pub struct CodingAdapterRegistry {
    adapters: BTreeMap<LanguageId, Box<dyn LanguageAdapter>>,
}
```

The coding domain operates on discrete verifier residuals (compiler, LSP, AST, tests), so its analytic constants remain `NotClaimed`; only the measured discrete gate and the spectral μ apply.

`perspt-research`

The research domain package: a working non-coding domain proving the `perspt-sdk` contracts execute without forking the scheduler, admissibility kernel, residual model, replay ledger, or dashboard event model.

Core Type

ResearchDomain - The `AgentDomainPackage` implementation:

```rust
pub struct ResearchDomain;

impl AgentDomainPackage for ResearchDomain {
    fn domain_id(&self) -> DomainId;   // "research"
    // detect, residual_schema, energy_model, correction_directions,
    // correction_packet, evaluate_workspace, hard_gate_policy,
    // verifier_suite, safety_barrier
}
```

Residual Mapping

Research reuses the SDK residual classes:

Residual Class	Research Meaning
`Syntax`	Malformed BibTeX entries or unbalanced braces
`TestFailure`	Unsupported / contradicted claim
`InterfaceMismatch`	Source mismatch
`ImportGraph`	Unresolved citation key (missing BibTeX or source entry)
`SymbolMismatch`	Mismatched citation label
`ContextDrift`	Stale evidence
`SheafInconsistency`	Cross-source contradiction

Detection looks for bibliography markers (`references.bib`, `refs.bib`, `bibliography.bib`, `sources.md`, `references.md`). Its `evaluate_workspace` verifier executes the `citation-check` stage over `.bib`, source-index, and Markdown/RST/LaTeX prose files, while `correction_packet` folds unresolved citations and syntax errors into targeted correction prompts. The safety barrier is `NotClaimed`.

`perspt-agent`

The governed PSP-9 agent runtime: candidate workspaces, the SRBN tool loop, sandboxed verification, and work-graph dispatch.

Modules

Module	Description
`candidate`	`CandidateWorkspace` - reversible coding candidate overlay with compiler-backed measurement

Module	Description
`exploration`	Deterministic, read-only repository orientation for the runtime
`external_tools`	`ExternalToolRuntime`/`ChatToolSession` - official-SDK MCP 2026-07-28 client for agent and chat: admitted tools plus read-only resource/prompt/completion operations, roots, sampling, elicitation, subscriptions, tasks, typed server events, and rejection diagnostics
`grant`	Persistent grant signing-key resolution
`lsp`	`LspClient` - JSON-RPC stdio client for language servers (the sensor architecture)
`measure`	`CodingCandidateMeasurer` - full verifier suite at gate boundaries, cheap syntax-only pass at mutation boundaries, and the explicit evolving/backward-compatible/external-oracle test-evidence policies
`probe`	`probe_route`/`ProbeReport` - behavioral provider probes (Gate U)
`promote`	Descriptor-relative workspace promotion
`realize`	`SnapshotRealizer`/`ProjectionMismatch` - the gate is evaluated on the realized candidate workspace, never the model's account of it
`runtime`	`Psp9AgentRuntime` - the authoritative PSP-9 agent runtime
`toolloop`	`ToolLoop` - the SRBN tool loop; every model-issued tool call is a governed proposal through the admissibility kernel
`tools`	`AgentTools` - filesystem and search operations plus the open `CandidateHandlerRegistry` the governed candidate dispatches through
`transport`	`GenAiTransport` - the only adapter joining the SDK's provider-neutral contract and `perspt-core`'s `genai` driver
`turn`	Universal actor turn runner shared by every stochastic actor (PSP-10)
`verifier`	Governed verifier sandbox: compiler/test/lint processes run behind a deny-network profile and a read allow-list; an explicit reduced-isolation policy is available on hosts without a backend

Key Types

Psp9ModelRoutes - Explicit model-plane routes for one PSP-9 session:

```rust
pub struct Psp9ModelRoutes {
    pub primary: Option<String>,
    pub actuator: Option<String>,
    pub explorer: Option<String>,
    pub adjudicator: Option<String>,
    pub fallbacks: Vec<String>,
}
```

Psp9RunSummary - Terminal outcome of one run:

```rust
pub struct Psp9RunSummary {
    pub session_id: String,
    pub node_id: String,
```

```
    pub outcome: NodeTerminalOutcome,
    pub turns_used: u32,
    pub ledger_head: String,
    pub promoted_paths: Vec<String>,
}
```

Usage

```
let runtime = Psp9AgentRuntime::from_config(workdir, &config, routes, run_config)?
    .with_database_path(db_path);

let summary = runtime.run(task).await?;
```

`perspt-tui`

Ratatui-based terminal user interface with two application modes.

Applications

Type	Description
`ChatApp`	Interactive chat with markdown rendering and response streaming
`AgentApp`	Agent dashboard with DAG tree, energy display, and review modal

Widgets and Rendering Modules

Module	Description
`dashboard`	`Dashboard` - main agent dashboard layout with panels
`task_tree`	DAG visualization showing node states and energy
`review_modal`	`ReviewModal` - grouped diff viewer with approve/reject/correct controls
`diff_viewer`	`DiffViewer` - unified diff display with syntax highlighting
`simple_input`	Minimal input widget with full control over key handling
`markdown`	Markdown, table, and LaTeX-math rendering for the chat TUI
`latex`	LaTeX to Unicode transpilation for terminal math rendering
`chat_view`	Message-area and input-box rendering for the chat TUI
`chat_commands`	Chat-local slash commands and conversation export
`theme`	`Theme` - color scheme and styling

Entry Points

```
pub fn run_chat_tui(...) -> Result<()>;
pub fn run_agent_tui_with_runtime(...) -> Result<()>;
```

Terminal setup and teardown live in `tui_runner`:

```rust
pub fn init_terminal(config: &TuiRunnerConfig) -> io::Result<TuiTerminal>;
pub fn restore_terminal(config: &TuiRunnerConfig) -> io::Result<()>;
```

Channels

```rust
pub fn create_app_event_channel() -> (AppEventSender, AppEventReceiver);
pub fn create_telemetry_channel() -> (TelemetrySender, TelemetryReceiver);
```

`perspt-cli`

Clap-based CLI entry point with sixteen subcommands, plus one feature-gated.

Subcommands

Command	Description
`chat`	TUI chat (default)
`simple-chat`	Plain-text streaming chat
`agent`	Governed coding agent; see `perspt agent --help` for current options
`init`	Initialize project memory and policy rules
`config`	View/set/edit configuration (`--show`/`--set`/`--edit`)
`ledger`	Query and manage the Merkle ledger (`--recent`/`--rollback <session-prefix>`/`--stats`)
`status`	Show current agent status
`audit`	Delayed audit labels and conformal activation (PSP-9)
`providers`	Print the provider capability matrix; `--probe` runs live probes
`replay`	Deterministic, credential-free audit replay of a session (PSP-9)
`abort`	Abort a PSP-9 session by revoking its authority epoch
`resume`	Resume a paused or crashed session
`dashboard`	Launch the web monitoring dashboard
`db`	Inspect and repair the local DuckDB store (`repair`)
`prompts`	Inspect and maintain the compiled prompt section libraries (`list`/`render`/`lint`/`manifest`/`explain-session`)
`context`	Explain a session's recorded resident-context events (`explain-turn`)
`benchmark`	Optional model-backed evaluation tooling (`validate`/`run`/`aggregate`); feature-gated behind the `benchmark` feature and run manually, never in CI

See CLI Reference for the complete flag reference.

perspt-store

DuckDB-backed session and ledger persistence.

> **Important**
> Perspt uses **DuckDB**, not SQLite, for its session store.

Core Type

```rust
pub struct SessionStore {
    conn: Mutex<Connection>,  // duckdb::Connection
}

impl SessionStore {
    pub fn new() -> Result<Self>;           // Default path
    pub fn open(path: &Path) -> Result<Self>; // Custom path
    pub fn open_read_only(path: &Path) -> Result<Self>; // Read-only mode
    pub fn default_db_path() -> Result<PathBuf>;    // ~/.local/share/perspt/
}
```

> **Note**
> `open_read_only` uses DuckDB's `AccessMode::ReadOnly` and does **not** call `init_schema()`. This provides read-only access for historical inspection. A separate process can conflict with an active writer; use the embedded `--dashboard` for live runs. The database file must already exist.

Record Types

Type	Description
`SessionRecord`	session_id, task, working_dir, merkle_root, detected_toolchain, status
`Psp9LedgerRow`	session_id, sequence, event_json, prev_hash, hash - one durable PSP-9 ledger record
`Psp9VerdictRow`	session_id, candidate_id, validator_id, stratum, missed, unsafe_label, evidence_hash
`Psp9CalibrationEpochRow`	epoch_id, stratum, target_rho, threshold, state, sample_count
`Psp9ExternalEffectRow`	idempotency_key, intent_hash, intent_json, result_json, status

The ledger accessors on `SessionStore` (append, read, and replay over the `psp9_*` tables) live in `store/psp9_ledger.rs`; `repair.rs` provides `repair_database`/`RepairReport`, the recoverable WAL quarantine behind `perspt db repair`.

DuckDB Tables

The schema is initialized by `init_schema()` through an idempotent transactional migration and comprises 11 tables: `sessions`, `schema_migrations`, `psp9_ledger_events`, `psp9_artifacts`, `psp9_authority_epochs`, `psp9_calibration_epochs`, `psp9_calibration_samples`, `psp9_context_checkpoints`, `psp9_external_effects`, `psp9_grant_policies`, `psp9_verdicts`.

perspt-policy

Starlark-based policy engine for agent action governance.

Core Types

```rust
pub struct PolicyEngine {
    policies: Vec<String>,
    policy_dir: PathBuf,
}

pub enum PolicyDecision {
    Allow,
    Prompt(String),
    Deny(String),
}

pub struct SanitizeResult {
    pub parts: Vec<String>,
    pub warnings: Vec<String>,
    pub rejected: bool,
    pub rejection_reason: Option<String>,
}
```

Functions

Function	Description
`PolicyEngine::new()`	Create engine instance
`PolicyEngine::load_policies()`	Load all .star files from policy directory
`sanitize_command(cmd)`	Parse, validate, and filter a shell command
`validate_workspace_bound(cmd, dir)`	Ensure command stays within working directory
`validate_artifact_mutation(path, root, op)`	Guard destructive artifact operations (delete/move) against protected project files

perspt-sandbox

Process isolation for agent-executed commands.

Core Types

```rust
pub struct CommandResult {
    pub stdout: String,
    pub stderr: String,
    pub exit_code: Option<i32>,
    pub timed_out: bool,
    pub duration: Duration,
}

pub trait SandboxedCommand: Send + Sync {
    fn execute(&self) -> Result<CommandResult>;
    fn display(&self) -> String;
    fn is_read_only(&self) -> bool;
```

```
}

pub enum FilesystemAccess {
    ReadOnly,
    WorkspaceWrite,
}

pub enum IsolationMode {
    Required,
    BestEffort,
}

pub struct ProcessPolicy {
    pub workspace_root: PathBuf,
    pub filesystem: FilesystemAccess,
    pub allow_network: bool,
    pub timeout: Duration,
    pub output_limit: usize,
    pub isolation: IsolationMode,
    pub environment: BTreeMap<String, String>,
}

pub struct ProcessSandbox { /* OS sandbox runner (bwrap on Linux, sandbox-exec on macOS) */ }

pub struct BasicSandbox {
    program: String,
    args: Vec<String>,
    working_dir: Option<String>,
    timeout: Option<Duration>,
    // ...
}
```

Usage

For OS-isolated child execution (used by the governed runtime and verifier), construct a `ProcessSandbox` with a `ProcessPolicy`:

```
let policy = ProcessPolicy::inspection("/path/to/project");
let sandbox = ProcessSandbox::new("cargo", vec!["test".into()], policy)?;
let result = sandbox.execute()?;
assert!(result.success());
```

For direct process execution with active timeout and output limits:

```
let sandbox = BasicSandbox::new("cargo".into(), vec!["test".into()])
    .with_working_dir("/path/to/project".into())
    .with_timeout(Duration::from_secs(60));

let result = sandbox.execute()?;
assert!(result.exit_code == Some(0));
```

`perspt-dashboard`

Real-time web dashboard for Perspt agent monitoring, built with Axum + Askama + HTMX.

Core Types

```rust
pub struct AppState {
    pub store: Arc<SessionStore>,        // shared store; read-only dashboard operations
    pub password: Option<String>,         // optional auth password
    pub session_token: Arc<Mutex<Option<String>>>,
    pub working_dir: PathBuf,
    pub is_localhost: bool,               // controls Secure cookie flag
}

pub enum DashboardError {
    Store(anyhow::Error),     // 503 - DB unavailable
    Template(askama::Error),  // 500 - render failure
    Internal(String),         // 500 - generic
}
```

`DashboardError` implements `IntoResponse` and renders a styled HTML error page. Store errors return `503 Service Unavailable`.

Router

`build_router(state: AppState) -> Router` constructs the full Axum router:

Method	Route	Handler
GET	`/login`	`auth::login_page` - render login form
POST	`/login`	`auth::login_handler` - validate password, set session cookie
GET	`/`	`handlers::overview::overview_handler` - session list
GET	`/sessions/{session_id}`	`handlers::session_detail::session_detail_handler` - session detail
GET	`/sessions/{session_id}/topology`	`handlers::dag::topology_handler` - work-graph topology
GET	`/sessions/{session_id}/backlog`	`handlers::backlog::backlog_handler` - backlog diagnostics
GET	`/sessions/{session_id}/energy`	`handlers::energy::energy_handler` - energy convergence
GET	`/sessions/{session_id}/decisions`	`handlers::decisions::decisions_handler` - decision trace
GET	`/sessions/{session_id}/governance`	`handlers::governance::governance_handler` - governance view
GET	`/sse/{session_id}`	`sse::sse_handler` - SSE event stream

Static assets are served from `crates/perspt-dashboard/static/` under `/static` via `tower_http::services::ServeDir`.

All routes except `/login` and `/static` are behind `auth::auth_middleware`. If no password is configured, all requests pass through.

Auth Middleware

Cookie-based authentication with random session tokens:

- On successful login, generates a 32-character alphanumeric token

- Stores token in `AppState::session_token`
- Sets `perspt_session` cookie: `HttpOnly`, `SameSite=Lax`, `Path=/`, `Secure` (when not localhost)
- Middleware checks cookie value against stored token
- No password configured -> open access mode

SSE Stream

The SSE endpoint pushes named events every 2 seconds:

- `psp9-stats` - live session summary (ledger event count, measurement count, last measured energy)

Each event contains an HTML fragment suitable for HTMX `sse-swap`.

Templates

Askama templates live in `crates/perspt-dashboard/templates/`:

- `base.html` - layout with navigation, HTMX, and DaisyUI theme
- `login.html` - login form
- `session_base.html` - shared per-session layout with tab navigation
- `pages/overview.html` - session list table
- `pages/session_detail.html` - single-session summary
- `pages/dag.html` - work-graph topology view
- `pages/backlog.html` - backlog diagnostics
- `pages/energy.html` - energy component table
- `pages/governance.html` - governance view
- `pages/decisions.html` - flat Merkle-chained PSP-9 event trace

View Models

Each page has a corresponding view model in `src/views/`:

- `OverviewViewModel` - sessions, node counts, budgets
- `SessionDetailViewModel` - single-session summary
- `TopologyViewModel` - work-graph nodes and edges
- `BacklogViewModel` - backlog diagnostics
- `EnergyViewModel` - per-node energy components
- `GovernanceViewModel` - governance evidence
- `DecisionsViewModel` - the flat Merkle-chained PSP-9 event trace

`views/psp9.rs` provides the shared `LedgerProjection` these view models and the SSE stream are built from.

perspt-prompt-macros

Build-time compiler for Perspt prompt section files (PSP-10). Called from an owning crate's `build.rs`, it parses every `prompts/<stage>/NN_name.md`, runs the codegen validation list, and emits typed section structs into `OUT_DIR`. A malformed section fails `cargo build` with an error naming the offending file — never a session.

Core Types

```rust
pub struct StageDecl {
    pub dir_name: String,
    pub separator: String,
}

pub struct PromptBuildError {
    pub file: String,
    pub line: Option<usize>,
    pub message: String,
}

pub struct Generated {
    pub rust_source: String,
    pub sections: Vec<CompiledSection>,
}

pub fn compile_prompt_dir(
    root: &Path,
    stages: &[StageDecl],
) -> Result<Generated, PromptBuildError>;
```

Usage

Consumers (`perspt-core`, `perspt-coding`) call `compile_prompt_dir` from `build.rs` and include the generated source:

```rust
include!(concat!(env!("OUT_DIR"), "/prompt_sections.rs"));
```

The same validation functions (`validate_section_body`, `parse_section_file`) serve the runtime bundle scanner and `perspt prompts lint`, so external replacement bodies obey exactly the rules the build does.

`perspt-benchmark`

Optional PSP-10 evaluation tooling, independent of runtime mechanism tests. Live runs require configured routes and credentials and never run in CI. The crate is feature-gated behind `perspt-cli`'s `benchmark` feature and surfaced as `perspt benchmark` (`validate`/`run`/`aggregate`).

Core Types

```rust
pub enum BenchmarkSuite {
    Smoke,     // One production-topology arm over a small task prefix
    Adaptive,  // The paging/adaptive pair for the default-activation decision
    Full,      // The complete seven-arm diagnostic ladder
}

pub struct BenchmarkRunOptions {
    pub config_path: Option<PathBuf>,
    pub suite: BenchmarkSuite,
    pub task_limit: Option<usize>,
    pub output: Option<PathBuf>,
}
```

Functions

Function	Description
`validate_corpus_command()`	Credential-free corpus validation: structure, coverage floors, and fail-before/pass-after hidden oracles
`run_benchmark(options)`	Run a suite using the configured production topology; model names are never benchmark arguments
`aggregate_reports(paths)`	Aggregate two or more completed reports (also credential-free)

Corpus and Arms

Tasks live in `corpus/<id>/`: a `task.json` (goal, hidden check argv, tags, expectation), a `fixture/` tree the agent works in, a `hidden/` tree of withheld oracle files, and a `solution/` overlay used only by offline corpus validation. The hidden suite runs in a fresh copy of the post-run fixture with `hidden/` overlaid on top, so the oracle and solution are genuinely unseen during evaluation.

Arms are a cumulative ladder over matched tasks: `direct`, `governed`, `packets`, `paging`, `adaptive`, `multi-family`, `integration`. The report publishes paired differences with a seeded 10,000-resample percentile bootstrap; the primary outcome is hidden-test hard pass.

Developer Guide

Internals, architecture, and contribution guide for Perspt developers.

Guide	Scope
Architecture	Crate structure, data flow, and SRBN internals.
Contributing	Dev setup, coding standards, and PR workflow.
Extending Perspt	Add providers, tools, language plugins, and custom domain adapters.
Documentation	Build and review the Guidedog website, figures, and PDFs.
Testing	Test infrastructure, patterns, and conventions.

Architecture

Perspt is a Rust workspace of fourteen crates plus a dev-only `xtask` automation crate. Eight crates make up the running program, four crates form the reusable platform layer (SDK, prompt codegen, and two domain packages), one crate is the optional feature-gated benchmark harness, and one root crate ties them together. Version 0.6.7 (“Omni”) runs on Rust 1.100 (Edition 2024) with the PSP-9 governed candidate runtime, the PSP-10 typed prompt, search, and integration layers, the `perspt` `AgentBuilder` facade, and the executable `ResearchDomain` citation verifier.

Workspace Layout

```
perspt/                       # Root: integration crate (perspt)
+-- crates/
|   +-- perspt-core/          # Types, config, LLM provider, events, plugins, prompts
|   +-- perspt-agent/         # Governed candidate runtime, tool loop, verifier, search
|   +-- perspt-tui/           # Ratatui TUI (chat + agent + review modal)
|   +-- perspt-cli/           # Clap CLI entry point, subcommands
|   +-- perspt-store/         # DuckDB session store
|   +-- perspt-policy/        # Starlark policy engine
|   +-- perspt-sandbox/       # Command sandboxing
|   +-- perspt-dashboard/     # Axum web dashboard
|   +-- perspt-sdk/           # Domain-neutral SRBN platform SDK
|   +-- perspt-prompt-macros/ # Build-time prompt section codegen
|   +-- perspt-coding/        # Coding domain package (first domain)
|   +-- perspt-research/      # Research domain package (second domain)
|   +-- perspt-benchmark/     # Optional credentialed evaluation harness
+-- xtask/                    # PSP code-rule & docs task runner (dev-only)
+-- tests/                    # Integration tests
+-- docs/                     # Guidedog & Typst documentation
```

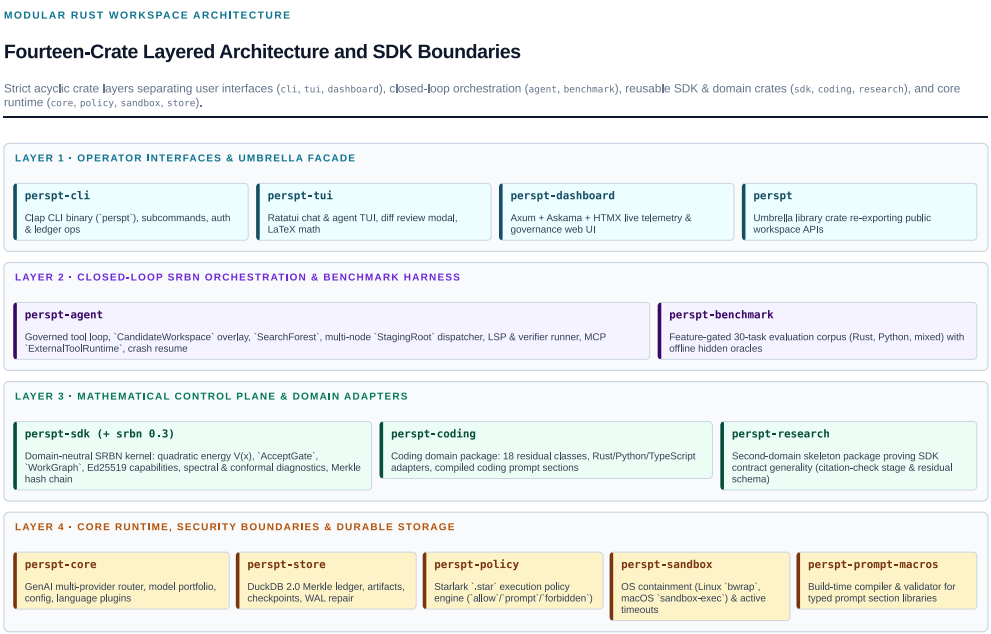
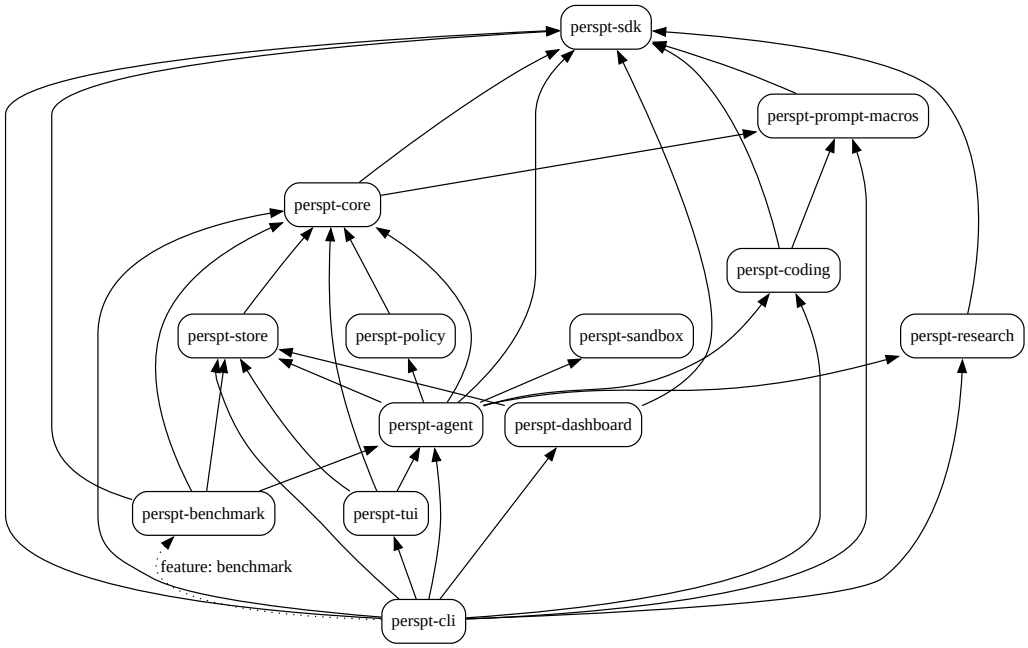


Fig. 7 Layered architecture of the fourteen-crate Perspt workspace.

Dependency Graph



PSP-9 / PSP-10 Overview

PSP-9 replaced the multi-agent orchestrator with a single governed candidate runtime: every model-issued tool call becomes a typed proposal, a deterministic admissibility kernel (in `perspt-`

sdk) decides whether it may affect the reversible candidate workspace, and every gate is evaluated on the re-measured candidate, never on the model's account of it. Every event lands in a hash-chained durable ledger (perspt-store), so sessions replay and resume deterministically. PSP-10 adds typed prompt section libraries with build-time codegen (perspt-prompt-macros), the bounded search forest with exact no-good learning, graph staging behind a global integration gate, and the optional benchmark harness (perspt-benchmark).

Crate: perspt-core

The foundation crate. Re-exports all canonical types.

Modules:

- types - Core shared types, split into submodules types/{context,model,plan,policy,verification,workspace}.rs (see Core Type Inventory below)
- config - Config { provider, model, api_key, ... } plus per-tier model overrides
- events - AgentEvent (33 variants), AgentAction, NodeStatus, ActionType
- llm_provider - GenAIProvider wrapping the genai crate; EOT_SIGNAL
- portfolio - ModelPortfolio with provider handles and declared caps
- plugin - LanguagePlugin trait + PythonPlugin, RustPlugin, JsPlugin
- prompts - Typed prompt section libraries (prompts/*/) compiled at build time by perspt-prompt-macros
- memory - ProjectMemory loaded from .perspt/memory.toml
- normalize - Model and provider name normalization

Key Plugin Types:

```rust
pub trait LanguagePlugin: Send + Sync {
    fn name(&self) -> &str;
    fn detect(&self, path: &Path) -> bool;
    fn get_init_action(&self, opts: &InitOptions) -> ProjectAction;
    fn test_command(&self) -> String;
    fn syntax_check_command(&self) -> Option<String>;
    fn verifier_profile(&self) -> VerifierProfile;
    fn owns_file(&self, path: &str) -> bool;
    // ... ~25 methods total
}
```

Plugins provide verifier profiles with fallback chains:

```rust
pub struct VerifierProfile {
    pub plugin_name: String,
    pub capabilities: Vec<VerifierCapability>,
    pub lsp: LspCapability,
}

pub struct VerifierCapability {
    pub stage: VerifierStage,       // SyntaxCheck | Build | Test | Lint | Format
    pub command: Option<String>,     // Primary command
    pub available: bool,
    pub fallback_command: Option<String>,
    pub fallback_available: bool,
}
```

Crate: perspt-agent

The governed PSP-9/PSP-10 agent runtime.

Modules:

- `runtime` - `Psp9AgentRuntime`: work-graph planning, bounded dispatch, node assembly, adjudication, staging and integration, resume, and the bounded search forest (`runtime/search/`)
- `toolloop` - The SRBN tool loop: each model-issued tool call becomes a typed proposal; the deterministic admissibility kernel decides whether it may affect the candidate
- `candidate` - `CandidateWorkspace`: reversible coding-candidate overlay
- `measure` - `CodingCandidateMeasurer`: full verifier suite at gate boundaries, cheap syntax-only pass at mutation boundaries
- `verifier` - Governed verifier sandbox: compiler/test/lint processes run under a deny-network profile with a read allow-list
- `tools` - `AgentTools` executor and sandboxing, the `CandidateHandlerRegistry` execution plane (`tools/handlers/`), and first-party tool families (`tools/families/{db,system}.rs`)
- `transport` - `GenAiTransport`: the only adapter between the SDK's provider-neutral contract and `perspt-core`'s `genai` driver
- `turn` - Universal actor turn runner (worker, explorer, architect, adjudicator, evidence summarizer, capability probe)
- `grant` - Persistent grant signing-key resolution
- `probe` - `probe_route`/`ProbeReport`: behavioral provider probes
- `promote` - Descriptor-relative workspace promotion on Unix; native write-through replacement plus best-effort reparse-point rejection on Windows
- `realize` - `SnapshotRealizer`: content-addressed workspace states
- `exploration` - Deterministic, read-only repository orientation
- `external_tools` - Official-SDK MCP 2026-07-28 client (stdio/stateless HTTP; tools, resources, prompts, roots, sampling, elicitation, subscriptions, MRTR/tasks), with local admission, replay, and separate agent/read-only-chat lifecycles
- `lsp` - `LspClient` (JSON-RPC over stdio)

Runtime Flow:

`Psp9AgentRuntime::run()` drives one governed session:

1. Exploration - deterministic read-only repository map; `--exploration-only` adds an interactive explorer tool loop and stops before any mutation
2. Planning - the default run uses a deterministic single-node graph. With parallel node dispatch, one governed architect turn can propose a validated graph; an invalid proposal falls back to one node
3. Dispatch - bounded multi-node scheduling (`runtime/dispatch.rs`); `--max-parallel-nodes` sets the concurrency (default 1; above 1 requires `--yes`)
4. Tool loop - per node, the governed loop (`toolloop/`) turns each model tool call into a typed proposal, admits it through the deterministic kernel, and realizes it against the reversible candidate overlay (`candidate.rs`)
5. Measurement - `measure.rs` re-measures the candidate through the plugin verifier suite inside the sandboxed verifier (`verifier.rs`), or through explicitly acknowledged host-user execution in native Windows reduced- isolation mode; the gate is evaluated on the re-measured candidate, never on the model's account of it. The default evolving-test policy measures the resulting tests; backward-compatible and protected external evidence are explicit additions, never implicit assumptions about every task's contract

6. Adjudication - a tool-free validator reviews only the realized diff and records an uncalibrated verdict (`runtime/adjudicate.rs`)
7. Staging and integration - node winners stage into a graph workspace and must pass the global integration gate before descriptor-relative promotion (`runtime/integrate.rs`, `promote.rs`). Unix promotion holds ancestor directory descriptors; Windows uses write-through native replace operations and rejects observed reparse points, without claiming the same race-resistant boundary.
8. Recording - every event lands in the durable hash-chained ledger (`runtime/recorder.rs`); interrupted sessions resume from the newest durable checkpoint with exactly the remaining budgets (`runtime/resume.rs`)

Failed attempts feed the bounded search forest (`runtime/search/`): sequential, at most three branch identities, one branch attempt (quantum) at a time, with exact no-good learning that suppresses only byte-identical repeated attempts.

Crate: `perspt-store`

DuckDB-backed persistence. **Not SQLite.**

```
pub struct SessionStore {
    conn: Mutex<Connection>,  // duckdb::Connection
}
```

12.1.6.1 DuckDB Schema and Tables

The schema (`crates/perspt-store/src/schema.rs`) is applied through an idempotent transactional migration; `schema_migrations` records the applied version. Eleven tables:

Table 54 Database Tables

Table Name	Key Columns	Purpose
`sessions`	`session_id` (PK), `task`, `working_dir`, `merkle_root`, `detected_toolchain`, `status`	Life cycle of agent sessions
`schema_migrations`	`version` (PK), `applied_at`	Applied schema versions
`psp9_ledger_events`	`(session_id, sequence)` (PK), `event_json`, `prev_hash`, `hash`	The durable canonical event stream; hash-chained
`psp9_artifacts`	`content_hash` (PK), `content`, `byte_len`, `media_type`	Content-addressed artifact bytes
`psp9_authority_epochs`	`session_id` (PK), `epoch`	Single-writer authority fencing per session
`psp9_context_checkpoints`	`(session_id, covered_event_root)` (PK), `checkpoint_json`	Durable conversation checkpoints for resume
`psp9_external_effects`	`(session_id, idempotency_key)` (PK), `intent_hash`, `status`	External (MCP) effect intents and results, idempotent
`psp9_grant_policies`	`policy_id` (PK), `session_id`, `policy_json`, `revoked`	Capability grant policies
`psp9_verdicts`	`(session_id, candidate_id, validator_id)` (PK), `stratum`, `missed`, `unsafe_label`	Adjudication verdicts per candidate

Table Name	Key Columns	Purpose
`psp9_calibration_epochs`	`epoch_id` (PK), `stratum`, `target_rho`, `threshold`, `state`, `sample_count`	Conformal calibration epochs
`psp9_calibration_samples`	`(epoch_id, sample_id)` (PK), `score`, `unsafe_label`, `audit_selected`	Calibration samples; `unsafe_label` stays NULL until the delayed audit label arrives

Row types live in `store/rows.rs` (`SessionRecord`) and `store/psp9_ledger.rs` (`Psp9LedgerRow`, `Psp9VerdictRow`, `Psp9CalibrationEpochRow`, `Psp9ExternalEffectRow`). `repair.rs` backs `perspt db repair` for recovering a database with a poisoned WAL.

Crate: `perspt-tui`

Ratatui-based terminal UI with two modes:

- **ChatApp** - Interactive chat with streaming, LaTeX math transpilation, ASCII table wrapping, and markdown saving
- **AgentApp** - Agent dashboard with work-graph tree, energy display, review modal

The agent TUI is entered through `run_agent_tui_with_runtime` (`agent_app.rs`), which owns the `Psp9AgentRuntime` for the session and wires its event and action channels.

Key components:

Component	Purpose
`Dashboard`	Main agent dashboard layout
`TaskTree`	Work-graph visualization with node states
`ReviewModal`	Grouped diff viewer with approve/reject/correct
`DiffViewer`	Unified diff display
`telemetry`	TUI-side energy component display types
`FrameRateLimiter`	60fps cap, adaptive rendering

Crate: `perspt-policy`

Starlark policy evaluation:

```rust
pub struct PolicyEngine {
    policies: Vec<String>,
    policy_dir: PathBuf,
}

pub enum PolicyDecision {
    Allow,
    Prompt(String),
    Deny(String),
}
```

Utility functions:

- `sanitize_command(cmd)` -> `SanitizeResult` (split, validate, filter)
- `validate_workspace_bound(cmd, working_dir)` - Ensure commands stay in scope

- `validate_artifact_mutation(path, workspace_root, operation)` - Protect root project files from delete/move

Crate: `perspt-sandbox`

Process isolation with OS containment (`bwrap` on Linux, `sandbox-exec` on macOS) and active timeout enforcement:

```rust
pub trait SandboxedCommand: Send + Sync {
    fn execute(&self) -> Result<CommandResult>;
    fn display(&self) -> String;
    fn is_read_only(&self) -> bool;
}

pub struct ProcessSandbox {
    program: String,
    args: Vec<String>,
    policy: ProcessPolicy, // ReadOnly/WorkspaceWrite, allow_network, Required/BestEffort
}

pub struct BasicSandbox {
    program: String,
    args: Vec<String>,
    working_dir: Option<String>,
    timeout: Option<Duration>,  // Active: spawn + poll + kill on deadline
    // ...
}
```

Core Type Inventory

All canonical shared types live in `perspt_core::types`, re-exported from `types/mod.rs`:

Workspace (``types/workspace.rs``):

Type	Description
`NodeClass`	Interface, Implementation (default), Integration
`VerifierStrictness`	Default (compile + tests), Strict (adds lint), Minimal (syntax only)

Model (``types/model.rs``):

Type	Description
`ModelTier`	Architect, Actuator, Verifier, Speculator; `default_model()` maps each tier to its recommended default route

Energy (``types/context.rs``):

Type	Fields
`EnergyComponents`	v_syn (syntax/type/build/lint), v_str (contracts), v_log (tests), v_boot (sensor availability & tool execution), v_sheaf (cross-node); `total()` is the plain sum of the rollups

Task Planning (``types/plan.rs``):

Type	Description
`TaskPlan`	Container for `Vec<PlannedTask>` with ownership-closure, acyclicity, and implicit-dependency validation

Type	Description
`PlannedTask`	id, goal, context_files, output_files, dependencies, task_type, contract, command_contract, node_class, dependency_expectations
`TaskType`	Code, Command, UnitTest, IntegrationTest, Refactor, Documentation
`PlannedContract`	interface_signature, invariants, forbidden_patterns, tests
`PlannedTest`	name, criticality (informational string label)
`DependencyExpectation`	required_packages, setup_commands, min_toolchain_version
`CommandContract`	command, expected_exit_code, expected_files, forbidden_stderr_patterns, working_dir

Verification and Context (``types/verification.rs``):

Type	Description
`SensorStatus`	Available | Fallback { actual, reason } | Unavailable { reason }
`StageOutcome`	stage, passed, sensor_status, output
`ArtifactKind`	Signature, Schema, SymbolInventory, InterfaceSeal
`StructuralDigest`	Content hash of a compile-critical structural artifact (signatures, schemas, seals)
`SummaryDigest`	Condensed summary with hash; `SummaryKind` is IntentSummary, VerifierResults, or DesignRationale
`ContextBudget`	byte_limit (100KB default), file_count_limit (20)
`RestrictionMap`	Per-node context boundary: owned_files, sealed_interfaces, structural/summary digests, dependency commits
`ContextPackage`	The bounded, reproducible context assembled for a node
`ContextProvenance`	Audit trail of the digests and files a node's context used

Policy (``types/policy.rs``):

Type	Description
`CommandPolicyDecision`	Allow, Deny, RequireApproval
`ManifestMutationPolicy`	Allow, Deny

Events System

The event system uses unbounded tokio channels:

```
// In perspt_core::events::channel
pub type EventSender = UnboundedSender<AgentEvent>;
pub type EventReceiver = UnboundedReceiver<AgentEvent>;
pub type ActionSender = UnboundedSender<AgentAction>;
pub type ActionReceiver = UnboundedReceiver<AgentAction>;
```

`AgentEvent` has 33 variants (`crates/perspt-core/src/events.rs`):

- **Planning**: `PlanReady`, `PlanGenerated`, `PlanRevised`, `FallbackPlanner`

- **Execution**: `NodeSelected`, `BundleApplied`, `NodeCompleted`
- **Verification**: `VerificationComplete`, `DegradedVerification`, `SensorFallback`
- **Sheaf**: `SheafValidationComplete`
- **Branches**: `BranchCreated`, `InterfaceSealed`, `BranchFlushed`, `BranchMerged`, `DependentUnblocked`
- **Escalation**: `EscalationClassified`, `GraphRewriteApplied`
- **Context**: `ContextDegraded`, `ContextBlocked`, `StructuralDependencyMissing`, `ProvenanceDrift`
- **Budget**: `BudgetUpdated`
- **File Ops**: `FileDeleted`, `FileMoved`
- **UI**: `ApprovalRequest`, `TaskStatusChanged`, `EnergyUpdated`, `Log`
- **Lifecycle**: `Complete`, `Error`, `ModelFallback`, `ToolReadiness`

The sheaf and branch variants are vestigial: the TUI still renders them for older ledger streams, but the PSP-9 runtime no longer emits them.

Data Flow

```
User Input
    |
[perspt-cli]  Parse args (clap)
    |
[perspt-core]  Config + Provider init
    |
+---+---+
|       |
chat    agent
|       |
[tui]   [perspt-agent]
|       |
|       +-- Psp9AgentRuntime
|       |     +-- exploration        -> read-only repository map
|       |     +-- plan               -> architect turn (update_graph)
|       |     +-- dispatch           -> bounded multi-node scheduler
|       |     +-- tool loop          -> proposals -> admissibility kernel
|       |     +-- candidate overlay  -> reversible workspace mutations
|       |     +-- verifier/measure   -> sandboxed deterministic verification
|       |     +-- adjudicate         -> diff-only validator verdict
|       |     +-- integrate/promote  -> staging, integration gate, promotion
|       |     +-- recorder           -> hash-chained ledger events
|       |
|       +-- GenAiTransport            (provider-neutral model plane)
|       +-- CandidateHandlerRegistry  (open execution plane + tool families)
|       +-- ExternalToolRuntime       (governed MCP)
|       +-- LspClient                 (JSON-RPC stdio)
|       |
|       +-- EventSender --> [perspt-tui AgentApp / perspt-dashboard]
|       +-- ActionReceiver <-- [perspt-tui ReviewModal]
|
[perspt-store]  DuckDB persistence (ledger, checkpoints, verdicts)
[perspt-policy]  Starlark rule evaluation
[perspt-sandbox]  Process isolation
```

Streaming Contract

Both chat and agent mode use the same streaming protocol:

1. LLM requests stream chunks over `mpsc::UnboundedSender<String>`
2. End-of-response signaled by `EOT_SIGNAL` (`<|EOT|>`)
3. Provider sends EOT - UI never adds its own
4. UI batches channel messages, handles first EOT, ignores duplicates

5. Streaming buffer updates the last assistant message live
6. Pending inputs queue until EOT is received

> **Warning**
> Never block the UI select loop. Spawn LLM work on tokio tasks and send results via the channel.

Contributing

Development Setup

```
# Clone
git clone https://github.com/eonseed/perspt.git
cd perspt

# Build
cargo build

# Run tests
cargo test

# Lint
cargo clippy -- -D warnings

# Format check
cargo fmt -- --check
```

Project Structure

```
crates/
  perspt-core/           # Types, config, LLM, events, plugins, prompts
  perspt-agent/          # Candidate runtime, tool loop, verifier, dispatch
  perspt-tui/            # Chat + Agent TUI
  perspt-cli/            # CLI entry (clap)
  perspt-store/          # DuckDB persistence
  perspt-policy/         # Starlark policies
  perspt-sandbox/        # Command sandboxing
  perspt-dashboard/      # Axum web dashboard
  perspt-sdk/            # Domain-neutral SRBN platform SDK
  perspt-prompt-macros/  # Build-time prompt section codegen
  perspt-coding/         # Coding domain package
  perspt-research/       # Research domain package
  perspt-benchmark/      # Optional credentialed evaluation harness
  perspt/                # Root integration crate
xtask/                   # PSP code-rule & docs task runner (dev-only)
tests/                   # Integration tests
docs/                    # Guidedog & Typst documentation
```

Coding Standards

1. **Clippy clean** - `cargo clippy -- -D warnings` must pass
2. **Formatted** - `cargo fmt` with default settings
3. **Tests pass** - `cargo test` must pass all tests
4. **No ``println!`` in UI paths** - Use the event system or `env_logger`

5. **PSP code rules** - Files <= 1408 lines, functions <= 70 code lines, lines <= 108 columns; enforced by `./check-rules.sh check`
6. **Streaming safety** - Never block the UI select loop; spawn on tokio tasks

Commit Messages

- Describe what changed, not the sequence
- Do NOT include phase numbers or commit sequence numbers
- Keep the subject line under 72 characters

```
# Good
Add sheaf validation for cross-language boundaries

# Bad
Commit 3/7: Phase 2 - Add sheaf validation
```

PR Workflow

1. Create a feature branch from `master` (or from the preceding feature branch for an explicitly stacked pull request).
2. Make changes with passing tests
3. Run the pull-request gate:

   ```
   cargo fmt --all -- --check
   cargo clippy --locked --workspace --all-targets -- -D warnings
   cargo test --locked --workspace --all-targets -- --test-threads=1
   cargo check --locked -p perspt-cli --features benchmark
   cargo doc --locked --workspace --no-deps
   ./check-rules.sh check
   ```

   These commands use the exact DuckDB 2.0.0 shared library. CI downloads the official checksummed library matching the pinned Rust crate; local systems should install the same version. Release binaries alone use `bundled`. The final `cargo check` covers the optional benchmark command without running any credentialed evaluation.
4. Push and open a PR
5. Address review feedback

CI Tiers

The workflows separate quick review feedback from target-platform evidence:

- Every non-draft pull request runs one Ubuntu job containing format, Clippy, credential-free tests, optional-feature compilation, Rustdoc, and the PSP code rules. Stacked pull requests are supported because CI does not filter on the PR's base branch.
- A GitHub `merge_group` runs the same Ubuntu gate plus Windows and macOS tests against the temporary merge-queue commit. Enable a branch ruleset that requires the merge queue and the `PR gate (Ubuntu)`, `PSP code rules`, and `Target OS` checks to make this evidence a pre-merge requirement.
- Pushes to `master` or `develop` run the target-OS matrix as a safety net for direct merges. Maintainers can also use **Actions -> CI -> Run workflow** to request the Windows/macOS matrix before merging when a queue is not enabled. That manual target-only run does not repeat the Ubuntu quality gate.

The macOS job exercises its governed verifier backend. The native Windows job tests chat/TUI and shared functionality, proves strict coding mode fails closed, and runs a Rust verifier fixture only after explicit reduced-isolation opt-in.

Superseded PR runs are cancelled. Merge-queue and protected-branch runs are never cancelled because their result is evidence for a specific commit. PDF generation is release work rather than a pull-request gate.

Documentation

Documentation is built in pure Rust and Typst using `guidedog` and `cargo xtask`:

```
# Build Typst assets, PSP catalog, Perspt Book HTML/PDF, and unified site
cargo run -p xtask -- docs

# Build Perspt Book HTML strictly
guidedog build html docs/perspt_book -W -n

# Validate and build PSP records (HTML + Typst PDF)
guidedog dsc check --dir=docs/psps
guidedog dsc build --dir=docs/psps
guidedog build html docs/psps -W -n

# Live preview
cargo run -p xtask -- serve
```

CI treats `guidedog` warnings as errors (`-W -n`) so documentation validation is deterministic. One documentation workflow publishes the unified Guidedog site, Typst PDFs, PSP catalog, and Rustdoc API as a single GitHub Pages artifact.

See Documentation for the publication toolchain, source map, and review checks.

Extending Perspt

Adding a Language Plugin

Language plugins implement the `LanguagePlugin` trait from `perspt-core`.

The built-in registry contains Rust, Python, and JavaScript/TypeScript. Go is an example of an extension we could add, not a shipped plugin.

1. Start from the trait in `crates/perspt-core/src/plugin/contract.rs` and an existing implementation such as `plugin/js.rs`.
2. Implement detection, LSP configuration, initialization and tooling actions, test and run commands, and the verifier profile.
3. Declare real availability checks and explicit no-op stages. A missing required stage must block a hard pass.
4. Register the plugin in `PluginRegistry` at the composition root and add the corresponding structured adapter to `CodingAdapterRegistry`.
5. Test detection, diagnostics, missing tools, and fail-before/pass-after verification with a local fixture.

Designing Domain Adapters

The platform SDK establishes a modular separation of concerns: the core SRBN control plane (in `perspt-sdk`) owns scheduling, residual scoring, capability checking, ledger tracking, and telemetry monitoring, while the domain adapter owns the domain-specific logic.

To write a custom domain extension (for example, to support research manuscript compilation, cloud deployment stability, or databases), developers must implement the `AgentDomainPackage` trait.

12.3.2.1 Implementing the AgentDomainPackage Trait

A package implements `AgentDomainPackage` from `crates/perspt-sdk/src/domain.rs`. We can inspect a complete, small implementation in `crates/perspt-research/src/lib.rs`:

```
use perspt_research::ResearchDomain;
use perspt_sdk::{AgentDomainPackage, DomainScope, WorkspaceSnapshot};

let domain = ResearchDomain::new();
let workspace = WorkspaceSnapshot::new(
    "/work/paper",
    vec!["references.bib".into()],
);
assert!(domain.detect(&workspace).activated);
let model = domain.energy_model(&DomainScope::default());
model.validate()?;
```

The required methods declare the domain identity, detection evidence, residual schema, energy model, correction directions, tool entries, hard-gate policy, verifier suite, and safety-barrier position. Every emitted consistency residual needs an explicit positive weight. Use `CorrectionDirection::new(class, instruction)` to construct a correction from structured residual evidence.

A contract declaration does not implement a verifier. The research package is a skeleton that declares a `citation-check` stage; source discovery and citation verification are not complete research workflows. A new domain also needs executable measurement and effect handlers wired into the runtime. We leave analytic stability and safety claims undeclared unless we can supply their required evidence.

12.3.2.2 Registering the Domain Adapter

Once implemented, register your custom package inside the `DomainRegistry` at the composition root in `crates/perspt-cli/src/commands/agent.rs`:

```
let mut domains = perspt_sdk::DomainRegistry::new();
domains.register(std::sync::Arc::new(ResearchDomain));
```

An explicit `--domain` id wins; otherwise the registry selects the best detection, falling back to the coding domain. Every domain must implement the same interface to maintain compatibility with the ledger and dashboard.

Adding an Agent Tool

Tools live under `crates/perspt-agent/src/tools/` (catalog and executor in `mod.rs` and `executor.rs`, sandboxing in `sandbox.rs`, builtin handlers in `handlers/`, first-party families in `families/`). The extension point is the open execution plane: a *tool family* contributes catalog entries (`perspt_sdk::ToolEntry`) and registers one `CandidateToolHandler` per tool name in the

`CandidateHandlerRegistry`. The governance wrapper (catalog -> validate -> budget -> certify -> execute -> re-certify) is uniform and lives outside the handlers.

```
use perspt_agent::{
    CandidateHandlerRegistry, CandidateToolHandler, CandidateWorkspace,
};
use perspt_agent::toolloop::EffectOutcome;

struct MyProbe;

#[async_trait::async_trait]
impl CandidateToolHandler for MyProbe {
    async fn apply(
        &self,
        workspace: &CandidateWorkspace,
        call: &perspt_sdk::ProviderToolCall,
        entry: &perspt_sdk::ToolEntry,
    ) -> anyhow::Result<EffectOutcome> {
        // Realize the effect against the reversible candidate.
    }
}
```

Register the family at the composition root (`crates/perspt-cli/src/commands/agent.rs`), through the same public path the shipped system and DB explorer families use:

```
let mut handlers = CandidateHandlerRegistry::with_builtins();
handlers.register("my_probe", Arc::new(MyProbe))?;
runtime = runtime
    .with_tool_family(vec![my_probe_entry()])  // catalog entries
    .with_tool_handlers(handlers);             // execution surface
```

Duplicate registration fails closed, so a family can never silently shadow a builtin. Handlers never decide admission, budgets, or certification — the admissibility kernel already did; they only realize the effect against the reversible candidate.

Adding a Prompt Section Library

Actor prompts are typed section libraries under `crates/perspt-core/prompts/` (`session_bootstrap`, `graph_plan`, `repository_explore`, `adjudicate`, `evidence_summarize`). A library is a directory of ordered Markdown sections with YAML frontmatter:

```
---
id: session_bootstrap/role
version: 1
role: system
required: true
max_bytes: 256
vars:
  domain_id: { type: "BoundedText<64>" }
---
You are a governed {{domain_id}} agent.
```

`perspt-prompt-macros` compiles the sections at build time (`frontmatter.rs` parses, `validate.rs` runs the codegen validation list, `emit.rs` generates the section constructors). After any section change:

1. `perspt prompts lint` - run the validation list (`--bundle` points it at an external bundle directory)
2. `perspt prompts manifest crates/perspt-core/prompts` - regenerate the committed `manifest.toml` with fresh content hashes

Adding a Benchmark Corpus Task

The optional evaluation corpus lives in `crates/perspt-benchmark/corpus/` (30 hidden-oracle tasks; see its `README.md`). Each task directory holds:

- `task.json` - user goal, shell-free hidden-check argv, expectation, one language tag (`rust`, `python`, `mixed`), one scale tag, and capability tags
- `fixture/` - the only tree copied into the agent's workspace
- `hidden/` - the oracle overlay copied only after the agent finishes
- `solution/` - a withheld reference overlay used only by validation

Validate with the credential-free authoring gate:

```
cargo run -p perspt-cli --features benchmark -- benchmark validate
```

The gate refuses fewer than 30 tasks, an imbalanced language/scale mix, insufficient capability coverage, a fixture that already passes its oracle, or a reference solution that does not pass.

Adding an LLM Provider

The `genai` adapter in `crates/perspt-core/src/llm_provider.rs` handles providers. To add a new provider:

1. Add the adapter kind to `str_to_adapter_kind()`
2. Map the env var in `new_with_config()`
3. Add the provider's default model to `detect_provider_from_env()` in `perspt-core/src/llm_provider.rs`
4. Update the auto-detection priority in config

Unknown provider identifiers must remain errors; a new adapter must not introduce an implicit fallback to another provider. Update explicit route resolution, environment detection, and the portfolio tests together.

Adding a Starlark Policy

Create a `.star` file under `rules/` in the platform configuration directory. Each file defines `evaluate(command)` and returns `"allow"`, `"prompt"`, or `"deny"`:

```
# Starlark, not a Python runtime hook.
def evaluate(command):
    if matches_pattern(command, "sudo"):
        return "deny"
    return "allow"
```

The engine evaluates command policies; it does not call invented file-write hooks. Capability and workspace checks govern file effects separately. See Security and Policy Rules for policy loading and precedence.

Testing

Test Infrastructure

Perspt uses Rust's standard test framework with `#[tokio::test]` for async tests.

```
# Run all tests
cargo test

# Run tests for a specific crate
cargo test -p perspt-core
cargo test -p perspt-agent
cargo test -p perspt-store

# Run a specific test
cargo test test_name

# Run with output
cargo test -- --nocapture
```

Test Organization

Location	Type	Description
`crates/*/src/*.rs`	Unit tests	`#[cfg(test)] mod tests` blocks
`tests/`	Integration tests	Cross-crate integration
`crates/perspt-store/`	Store tests	DuckDB schema, CRUD, ledger
`crates/perspt-agent/tests/`	Mechanism checks	23 `mc_*.rs` suites plus `psp9_runtime.rs` covering the PSP-9/PSP-10 runtime mechanisms (tool loop, dispatch, search, recovery)
`crates/perspt-sdk/tests/`	Mechanism checks	`mechanism_checks.rs` and `mc_prompt_activation.rs` for the platform SDK

Test Patterns

Temporary Store:

Use `SessionStore::open()` against a temporary directory for tests that need persistence without touching the user database:

```
#[test]
fn test_session_roundtrip() {
    let dir = tempdir().unwrap();
    let store = SessionStore::open(&dir.path().join("test.db")).unwrap();
    store
        .create_session(&SessionRecord {
            session_id: "s1".into(),
            task: "demo".into(),
            working_dir: "/tmp/demo".into(),
            merkle_root: None,
            detected_toolchain: None,
            status: "active".into(),
        })
        .unwrap();
    assert!(store.get_session("s1").unwrap().is_some());
}
```

Plugin Testing:

Test plugin detection and verifier profiles:

```
#[test]
fn test_python_plugin_detection() {
    let plugin = PythonPlugin;
    let dir = tempdir().unwrap();
    std::fs::write(dir.path().join("pyproject.toml"), "").unwrap();
    assert!(plugin.detect(dir.path()));
}
```

Energy Computation:

Test energy calculation with known inputs:

```
#[test]
fn test_energy_total() {
    let energy = EnergyComponents {
        v_syn: 1.0,
        v_str: 0.0,
        v_log: 2.0,
        v_boot: 0.0,
        v_sheaf: 0.0,
    };
    assert_eq!(energy.total(), 3.0);
}
```

Event Testing:

Test event channel communication:

```
#[tokio::test]
async fn test_event_flow() {
    let (tx, mut rx) = event_channel();
    tx.send(AgentEvent::Log { message: "test".into() }).unwrap();
    let event = rx.recv().await.unwrap();
    assert!(matches!(event, AgentEvent::Log { .. }));
}
```

Quality Gates

All PRs must pass:

```
cargo fmt --all -- --check
cargo clippy --locked --workspace --all-targets -- -D warnings
cargo test --locked --workspace --all-targets -- --test-threads=1
cargo check --locked -p perspt-cli --features benchmark
cargo doc --locked --workspace --no-deps
./check-rules.sh check
```

The PR gate runs once on Ubuntu. It downloads the official DuckDB 2.0.0 shared library, verifies its published checksum, and compiles the optional `benchmark` CLI without running live model evaluation. Release builds, not CI tests, use the expensive `bundled` feature. A merge-queue commit, protected-branch push, or manual full run adds Windows and macOS tests. Platform-neutral formatting, Clippy, Rustdoc, and PSP rules are not repeated on every operating system. macOS exercises the native governed verifier. Windows covers native chat/TUI and shared code, verifies strict coding mode fails closed, and separately exercises the explicit reduced-isolation verifier.

GitHub merge queues are the pre-merge path: the `merge_group` event checks the proposed change combined with the latest target branch and any changes ahead of it in the queue. Without an enabled merge-queue ruleset, maintainers must use the CI workflow's manual full run when pre-merge cross-platform evidence is required; the protected-branch push remains a post-merge fallback.

Benchmarks Are Not Tests

`perspt benchmark` is not part of `cargo test` or CI. The `perspt-benchmark` crate is gated behind the non-default `benchmark` feature of `perspt-cli`, requires configured model credentials, and is run manually to evaluate and compare Perspt against other coding agents. Its own unit tests are credential-free and run with the ordinary workspace suite.

Panic Safety

The TUI runner restores the terminal on its normal exit path. If a crash leaves the terminal in raw mode, use `reset` or `stty sane`. The integration fixture `tests/panic_handling_test.rs` covers panic and error scenarios.

Documentation

The website, book, and proposal catalog are published with Guidedog. Python, Sphinx, pip, and a virtual environment are not part of this publication workflow. The source format remains reStructuredText.

Where content lives

Source	Purpose
`docs/site/`	Landing page, layout template, and landing-page CSS.
`docs/perspt_book/`	Tutorials, user guides, concepts, and references.
`docs/psps/`	Design proposals, lifecycle metadata, and generated registry.
`docs/shared/_static/`	Shared styles, local math rendering assets, and generated figures.
`docs/diagrams/`	Editable Typst figure sources.
`docs/perspt_book/_templates/`	Book layout and shared cover artwork.
`docs/conf.toml`	Website mounts for the landing page, book, PDF, and proposal catalog.

The `sphinx.ext.graphviz` and `sphinx.ext.mathjax` strings in configuration are compatibility identifiers interpreted by Guidedog. They do not load Python extensions. Python snippets, Starlark highlighting, Python-language agent examples, and benchmark fixtures are separate from the documentation publisher.

Build and preview

```
guidedog --version

# HTML only, using prepared artwork when Typst is unavailable
cargo run -p xtask -- docs --no-pdf
```

```
# Website, book PDF, and proposal PDFs
cargo run -p xtask -- docs

# Recompile diagram and cover sources with the standalone Typst CLI
cargo run -p xtask -- assets

# Live local preview
cargo run -p xtask -- serve
```

Guidedog includes the PDF and Graphviz engines. The standalone Typst CLI regenerates the checked-in SVG figures and cover previews. After changing a figure, regenerate its assets before reviewing the site. CI publishes those prepared assets, so source and output must stay together.

Validate a change

```
guidedog dsc check --dir=docs/psps
guidedog build html docs/perspt_book -W -n
guidedog build html docs/psps -W -n
cargo run -p xtask -- docs
```

We review the landing page and representative book pages at desktop and narrow widths in both themes. We check the PDF cover, figures, captions, and code examples after layout changes. The complete publication helper `./generate-docs.sh` also builds and stages Rustdoc. CI pins the publisher version and checksum in `.github/actions/guidedog/action.yml`.

Keep the documentation tied to behavior

- **Commands**: Check `crates/perspt-cli/src/main.rs` and each command implementation. Help text alone does not establish stdin or stdout behavior.
- **Scoring and gates**: Check `crates/perspt-sdk/src/energy.rs` and `gate.rs`. An accepted descent is intermediate progress; a hard pass is the delivery success condition.
- **Verification**: Check `crates/perspt-agent/src/measure.rs` and `crates/perspt-core/src/plugin/`. Required stages and advisory stages have different consequences.
- **Promotion and rollback**: Check the runtime and ledger commands. A ledger checkpoint is not a Git commit, and rollback is not arbitrary time travel.
- **Domain support**: Check registered plugins and domain packages. A skeleton contract is not a complete user workflow.
- **Proposal states**: Use the generated registry rather than copying lifecycle states into tutorials. A proposal can describe intended behavior beyond the implementation.

Use short, complete sentences and task-based examples. We put the command and expected decision before the internal vocabulary. We label conceptual plots as illustrations and tie empirical claims to recorded evidence. The quadratic energy formula is continuous in residual magnitudes; the coding runtime measures discrete candidate states.

Changelog

Version 0.6.7 - “Omni”

This release upgrades the workspace toolchain to Rust 1.100.0 (Edition 2024), modernizes the dependency stack, and delivers the `"Omni"` multi-domain agent SDK facade so coding, research, and custom non-coding domains run natively through the same SRBN control plane.

Toolchain, Dependencies & Refactoring

- **Rust 1.100.0 & Edition 2024** - Updated `rust-toolchain.toml`, workspace `rust-version = "1.100"`, `edition = "2024"`, and CI/Docs/Release workflows; adopted Edition 2024 `if let` chains and pattern ergonomics.
- **Scoped ``genai 0.7.4`` Provider Clients** - Eliminated all process-global `std::env::set_var` and `remove_var` mutations in `perspt-core::llm_provider` and tests in favor of explicit per-client `ProviderClientOptions` and custom `base_url` resolution.
- **Major Crate Upgrades** - Upgraded `duckdb 2.0` (`1.20000.2` + DuckDB `v2.0.0`), `rmcp 3.10.0`, `toml 1.2.1`, `uuid 1.29.2`, `syn 3.0.10`, `tui-markdown 0.3.13`, and updated integration harnesses for Cargo 1.100 target artifact paths while removing unused legacy dependencies.
- **Modular CLI & Runtime Architecture** - Extracted `#[derive(clap::Args)] pub struct AgentArgs` and decomposed `commands::agent::run` into focused helpers; decomposed oversized helpers in `perspt-agent` within strict PSP-3 complexity bounds.

Omni-Domain Agent SDK & Bug Fixes

- **``perspt`` “Omni” Facade** - Added `AgentBuilder` and `perspt::prelude` re-exporting core agentic types, domain packages (`CodingDomain`, `ResearchDomain`), async/sync `ask_user` hooks, custom governed tool registration, delegated subagents, and TUI/channel streaming.
- **Workspace Domain Evaluation Hook** - Added `WorkspaceSnapshot` text snapshots, `DomainEvaluation`, and `AgentDomainPackage::evaluate_workspace` in `perspt-sdk`, wired into `CodingCandidateMeasurer` (incremental and full gate passes when build/test plugins are absent).
- **Executable ``ResearchDomain`` (``citation-check``)** - Implemented deterministic BibTeX syntax parsing, Markdown/RST/LaTeX citation verification against `.bib` and source indexes, `evaluate_workspace`, and typed `correction_packet` self-healing in `perspt-research`.
- **Async Interactive Hooks & Correction Refinement** - Added `AsyncFnInteractionHandler` in `perspt-agent`, and fixed human-in-the-loop `RequestCorrection` refinement (`ToolLoop::run_requiring_turn` / `CandidateSeed::require_model_turn`) so feedback turns always execute before re-gating.

Version 0.6.6 - “□□□ (Shen DaiJun)”

In memory of a mother who always gave and never asked anything in return.

This release implements PSP-9’s governed agentic platform: every stochastic action is a proposal, every effect is mediated by a deterministic kernel, every accepted state is measured

on the realized candidate, and every reliability claim is conditional on recorded evidence — following the *Stability is All You Need* papers I, II, and III. On top of it, PSP-10 makes the loop's prompts, resident context, and search governed artifacts of the same record.

Governed Tool Loop & Admissibility Kernel (PSP-9):

- **SRBN Tool Loop** - New `perspt-agent::toolloop` implements Paper II's harness contract as an executable loop: model-issued tool calls become Paper III proposals, the five-clause admissibility kernel (Definition 3.2) decides each one, and the acceptance gate reads the re-measured candidate — never the model's account of it. An adversarial "lying model" fixture proves a model claiming success while every effect is denied can never reach a hard pass.
- **Five-Clause Kernel** - `check_full_admissibility` replaces hardcoded contract/barrier passes with registered evaluators; an absent clause is recorded absent, never true. Promotion is one transaction debiting exactly the barrier's certified increment `c_t` — the allowance and the debit are the same number.
- **Operational Safety Barrier** - `perspt-coding` registers six versioned policy channels (protected paths, sandbox escape, secret exposure, network reachability, dependency policy, resource limits) with exact deterministic increments; correctness evidence stays in `V`.
- **Recovery Lattice** - One shared, non-replenishing cascade budget across all five levels (Definition 8.1): exhaustion forces strict escalation, containment is unconditional, and every cascade terminates within `b + k` steps in one of Theorem 6's four terminal classes.

Model Portfolio & Transport:

- **Multi-Provider Portfolio** - The `[providers]` table holds several credentialed routes in one process; `perspt providers` prints the capability matrix with declared-vs-probed honesty. A one-entry portfolio keeps single-provider setups unchanged.
- **ModelTransport Port** - The SDK declares the object-safe transport port; `perspt-core` provides the `genai` tool-calling driver; `perspt-agent::transport` is the single adapter — the tool loop cannot name a vendor type or credential (Gate S).
- **Portfolio Routing** - Fully qualified `provider::model` routes with capability objectives, decorrelated-family review preference, and cross-provider failover chains. Failover is recovery, not retry, and draws on the shared cascade pool.
- **Typed Tool Catalog** - Declarative footprints (`writes ⊆ reads` by construction), capability-filtered tool specs, and an open `LanguageId` adapter registry. `sed_replace`/`awk_filter` leave the model-facing catalog; `edit_file` fails closed on ambiguity. External tool servers register under the kernel and can never exceed session authority.

Measured Certificates:

- **Realizability Interface** - `srbn` upgraded 0.1.1 → 0.3.0 (`stabilize_realized`, restore-best policy, hash-chain ledger). Coding uses unmeasured realization with `projection_mismatch` telemetry; the analytic Theorem 12.4 path is admitted only when every hypothesis — including Theorem 12.1's step-size bounds — is represented by evidence.
- **Independence Statistics** - Matched-stratum pairwise misses, degenerate marginals using direct joint misses, conservative Hoeffding upper bounds, and certification gating; `ρ_eff` reports `n` beside it and the correlation-driven energy attenuation helper is removed.
- **Conformal Feasibility Floor** - A declared budget `ρ` fixes `n ≥ 1/ρ − 1` labeled samples before anything can be autonomously accepted; reject-all is reported as *insufficient calibration*, never

as a satisfied budget. Calibration is keyed by full exchangeability strata with shadow/active/stale readiness epochs.

- **Audit Replay** - `perspt replay <session>` folds the durable event stream deterministically with tamper detection and no provider credentials; context compaction preserves an exact `ControlFrame` and unresolved tool calls, and a stale checkpoint is rebuilt, never patched.

Recovery, Resume & Calibration:

- **Recovery Ladder** - The runtime now holds Theorem 6's higher rungs live: level 2 revises the work graph with the exhausted attempt's evidence and re-runs at a new generation, level 3 hands the node to the configured architect route, and level 4 durably revokes the session's authority epoch — also wired to `perspt abort` and the TUI quit path.
- **Exact Mid-Loop Resume** - Every gate acceptance writes a durable candidate checkpoint (control frame plus content-addressed file artifacts). `perspt resume` rebuilds the accepted candidate from it, re-mints a fresh capability from the grant intersection at the current durable epoch, and re-enters the loop with exactly the remaining budgets; a bumped epoch refuses the resume.
- **Delayed Audit Labels & Conformal Activation** - Every promotion persists an audit-selected, unlabeled calibration sample; `perspt audit` ingests delayed labels (single-assignment) and recomputes the stratum threshold over labeled samples only. At the finite sample floor a new immutable epoch activates, after which hard-pass candidates above `theta` commit without a prompt with the certified accept ledgered; any other state backs off to approval.
- **Grant Ceilings Enforced** - `GrantPolicy::mint` intersects every live capability with the grant ceilings (effects, paths, commands, deny-by-default network, approval strictness) and fails closed; persisted grant intent is signed over canonical bytes and verified against a local trust anchor on resume, never the embedded key.
- **Behavioral Provider Probes** - `perspt providers --probe` observes each configured route on a live two-tool round trip (tool calling, multi-tool selection, parallel batching, schema validity) with evidence labelled `behavioral`; all three configured routes passed live.
- **Exploration-Only Mode** - `perspt agent --exploration-only` runs an interactive explorer tool loop under a strictly read-only capability: every call passes the kernel and mutation attempts are recorded denials.
- **Parity Gate Passed** - The governed tool loop ran against the ungoverned whole-file ablation baseline live: 100% hard-pass with zero false stability, failures preserved. No measurement justifies a typed whole-file tool yet; PSP-9 status moves to Final. The `parity_bench` harness is retired by PSP-10; evaluation moves to the optional `perspt-benchmark` crate, run manually.
- **Governance Surfaces** - A dashboard Governance page shows authority epochs, signed grant intent, calibration epochs, pending delayed audits, and validator verdicts; `perspt status`, `perspt ledger`, and `perspt abort` now read and act on the PSP-9 surfaces instead of printing placeholders.

PSP-9 Completion (all phases Done):

- **Open Execution Plane** - Tool execution moved behind an exact-name handler registry with a conformance fixture proving a new family registers at the composition root (catalog entries + handlers + derived grants) without editing the loop, the candidate, or the node assembly. The legacy 15-arm executor dispatch and its ungoverned `sed_replace`/`awk_filter`/`run_command` paths are deleted; grants derive from the assembled catalog intersected with an explicit withheld set.

- **Shipped Tool Families** - A read-only system explorer (`sys_info`, `sys_processes`, `sys_disk`, `sys_env` — variable names only, values withheld) and a local DB explorer (`db_list`, `db_schema`, `db_query` — SELECT-only over an in-memory read-only DuckDB view with statement allowlisting and row/byte caps) under two new deliberate read-only effect kinds, `SystemProbe` and `DataRead`.
- **Domain Registry Wired** - The runtime selects its domain package from the open `DomainRegistry` (`--domain` or detection); the research domain is registered and reachable; the closed `CodingLanguage` enum is retired in favor of the `LanguageId` adapter registry; `PluginRegistry` accepts registrations.
- **Governed Dependency Mutation** - `mutate_dependencies` resolves through each language plugin's dependency commands (`cargo add`, `uv add`/`uv lock``+``uv sync`, `npm install`) behind an explicit `--allow-dependency-mutation` grant, with Manifest/Lockfile footprints, package-name injection guards, pre-image journaling, and write-ahead external-effect bracketing; manifest and lockfile promote like any governed mutation.
- **Ledger-Folded Conversation (Gate O)** - Model context is a pure fold over digest-chained conversation deltas recorded before they are applied; the durable checkpoint's control frame carries the rolling fold digest, and resume refolds the ledger and refuses a conversation that is not derivable from the record.
- **Commuting Tool-Call Batches (Gate P)** - Same-turn calls are planned over declared footprints: commuting mutators run in the model's own arrival order, precise-footprint reads execute concurrently, and a colliding pair is returned as a `ToolBatchConflict` observation, never given an invented order.
- **Bounded Multi-Node Dispatch (Gate P)** - `--max-parallel-nodes` (default 1, requires `--yes`) enables a governed forced-choice `update_graph` planning turn and a dispatcher running per-node lifecycle futures on borrowed structured concurrency with cancellation tokens, one shared non-replenishing recovery pool, first live `GraphWrite` lease use, and single-flight promotion; `--max-parallel` keeps its verifier-fan-out meaning.
- **Proposal Ensembles Superseded (Gate M/T)** - The live proposal ensemble (`[ensemble]` / `--ensemble-width`) is removed by PSP-10 in favor of the bounded search forest: each branch is still one ledgered gate decision in its own reversible overlay with selection strictly by measured energy, and a lingering `[ensemble]` block is now a fatal startup error pointing to `[exploration]`.
- **Certified Pairwise Bounds (Gates R/T)** - The deterministic verifier suite records a verdict row beside the adjudicator's for the same candidate and stratum; `perspt audit` labels verdicts with the calibration sample in one pass; the independence estimator runs over labeled matched verdicts and surfaces per-pair statistics in `perspt status` and the governance dashboard — Certified with `n` or the literal `insufficient evidence`.
- **Latest MCP Client Wired Into Runtime and Chat (Gates K/L/U)** - The custom compatibility transport was replaced by the official Rust SDK and a strict MCP 2026-07-28 `server/discover` lifecycle with no legacy fallback. Stdio and stateless Streamable HTTP now cover tools, resources/templates, prompts/completions, explicit roots, opt-in sampling/elicitation, typed notifications, `subscriptions/listen`, MRTR, and bounded tasks. Remote tools remain locally admitted, bracketed, and replayed; protocol context primitives use reserved read-only operations.
- **MCP TUI and Boundary Hardening** - `/mcp` and `/help` expose status, policy rejection, and elicitation actions; chat accepts interactive form/URL requests while the non-interactive agent declines them. Sampling preserves the local model route and forwards generation controls,

subscription changes re-admit catalogs, task and sampling budgets are bounded, stdio children inherit no ambient credentials beyond executable-resolution variables, and bracketed paste now inserts Unicode/multiline clipboard content into the chat input.

- **Diagnostics and Rollback (Gate R) -** The dashboard renders the work-graph revision lineage (Topology), a Backlog page with the arriving-potential gauge Φ(W) labeled as conditional diagnostics, and the governance independence pair table; `perspt status` shows Φ(W), the withheld `topology_gap`, and certification state; `perspt ledger --rollback <session>` restores promoted pre-images through the hardened descriptor-relative path, extends the hash chain with the rollback event, and labels the candidate UNSAFE — the delayed-label source for calibration and independence.
- **Promotion TOCTOU Hardening** - Promotion, resume recovery, and rollback share one engine. Unix uses `openat`/`renameat`/`unlinkat` with `O_NOFOLLOW` component walks and held directory descriptors. Native Windows uses write-through replacement and rejects observed reparse points in its explicitly weaker reduced-isolation mode. Rollback failures are reported, never swallowed. The scheduler now tracks refined generations exactly.
- **Recovery Continuity** - Transient provider errors (429, 5xx, timeouts) are retried with bounded exponential backoff inside the transport, per Gate P's rule that rate-limit waits are throughput delays; recovery-ladder rungs reseed from the previous attempt's best accepted candidate state instead of re-paying the whole cost (`ladder_reseeded`); and containment caused by exhausted transport preserves the session's authority epoch so `perspt resume` remains viable after an outage, while governance containment still revokes.
- **Legacy Retirement** - ~19,000 lines of orphaned PSP-5 orchestrator code, dead TUI entry points, the legacy dashboard LLM/Sandbox pages, and the empty legacy status branch are removed; remaining pages read the PSP-9 ledger.

Typed Prompts, Paged Context & Bounded Search (PSP-10):

- **Typed Prompt Sections & Codegen** - The new `perspt-prompt-macros` crate compiles typed prompt sections into prompt programs; every compiled program and every model call is ledgered with ordered section provenance, the tool-surface hash, and its digest. `perspt prompts` (`list`, `render`, `lint`, `manifest`, `explain-session`) inspects and maintains the section libraries, and `perspt agent --allow-experimental-prompts` substitutes validated `[prompts]` bundle sections live — experimental until a change record passes paired evaluation (Gate AE).
- **Resident-Context Paging** - Model context is a paged resident set with reserves declared in `[context]`: evicted pages stay addressable through the `context_recall` tool, every compaction and refusal is a recorded context event, and `perspt context explain-turn` explains a session's recorded events.
- **Bounded Search Forests** - After a gate failure the runtime can open a bounded search forest (`[exploration]`; at most three branch identities, sequential eager branches): exact no-goods are learned from rejected candidates, and every branch's workspace reservation is durable and pre-action with cost-free refusals.
- **Multi-Node Dispatch with Integration Gate** - `--max-parallel-nodes` dispatches work-graph nodes into per-node staging overlays behind one global integration gate, so concurrent nodes promote through a single measured decision point.
- **Evolving Test Evidence** - Production coding gates now default to `[verification] test_policy = "evolving"`: the resulting implementation, tests, and project configuration are measured

together, so an intentional contract change is not forced to satisfy a superseded test. `backward-compatible` explicitly adds the restored historical regression view, while `external-oracle` overlays separately protected acceptance material on a private candidate copy and adds its command as a blocking test-stage verdict. The selected policy is shared by node, search-preview, and integration measurements and recorded at session start; benchmark hidden oracles remain withheld and fixed.

- **Crash Resume** - `perspt resume` rebuilds a crashed session from the durable record, judging the work graph by its latest recorded state and refolding the conversation by digest chain.
- **Optional Benchmark Crate** - The new `perspt-benchmark` crate (perspt-cli Cargo feature `benchmark`, non-default) evaluates and compares Perspt with other coding agents over a 30-task hidden-oracle corpus: suites `smoke` (1 arm, 8 tasks), `adaptive` (2 arms), and `full` (7 arms). It is credentialed and run manually only — never part of `cargo test` or CI.

Engineering Discipline:

- **Reproducible Toolchain** - Added `rust-toolchain.toml` pinned to Rust 1.97.1 while retaining edition 2021; workspace packages inherit the same MSRV.
- **DuckDB 2.0.0 Boundary** - Exact-pinned `duckdb` at `=1.20000.0`, added dynamic ABI rejection, transactional versioned migrations, catalog-checked nullability changes, and post-DDL checkpoints.
- **Recoverable WAL Repair** - `perspt db repair --db-path PATH --discard-wal` records WAL size and SHA-256, makes durable timestamped database/WAL backups, quarantines the WAL, verifies recovery read-only, and restores the original WAL on failure.
- **Shared MCP Runtime Foundation** - Introduced the governed external-runtime boundary, mode filtering, local effect/footprint admission, namespacing, bounded observations, environment-referenced secrets, replay-only observations, and explicit reconciliation. The current release supersedes its initial tool-only transport with the official latest SDK client described above.
- **PSP-5 API Retirement** - Removed the legacy orchestrator and Merkle-ledger facade from the public agent crate and refused resume of PSP-5 sessions; their existing rows remain inert forensic data.
- **PSP Code Check** - New `xtask` checker enforces file ≤ 1408 lines, function ≤ 70 code lines (measured with `syn`, not brace counting), and line ≤ 108 columns, with a shrink-only baseline ratchet wired into CI (`./check-rules.sh`).
- **Workspace Decomposition** - The 5,889-line orchestrator, `types.rs`, `plugin.rs`, `store.rs`, `tools.rs`, `ledger.rs`, and `verification.rs` split into focused submodules; workspace version fields now inherit from `[workspace.package]`.
- **Native Windows Capability Contract** - TUI chat, simple-chat, MCP, and the coding workflow compile natively. Coding process execution stays fail-closed by default and requires `--allow-unisolated` or `[verification] allow_unisolated = true` until a restricted-token/ACL sandbox backend exists; CI tests strict refusal and the acknowledged native verifier path separately.
- **Audit-Driven Hardening** - A full-code audit closed a candidate-restore gap (files created after a checkpoint now roll back exactly via a pre-image journal), separated read paths from the promotable mutated set, routed promotion itself through the five-clause kernel, made the policy engine consume canonical command lines, inverted network-scope delegation to deny-by-default, and made ledger appends O(1) via stage-then-commit.

Version 0.6.2 - “Hózhó”

Hózhó (Navajo) — A state of perfect balance, harmony, and continuous self-improvement.

This iteration focuses on bringing greater stability, balance, and refinement to the core architecture. It introduces the full platform SDK implementation alongside the mathematical constructs outlined in the *Stability is All You Need* paper series (Papers I, II, and III) and formalizes PSP-8.

Stability Is All You Need & Platform SDK (PSP-8):

- **SDK Platform Crates** - Introduced `perspt-sdk`, `perspt-coding`, and `perspt-research` to separate domain-neutral orchestration logic from target execution domains.
- **Measured Acceptance Gate** - Implemented the PSP-8 spectral acceptance gate using symmetric eigensolvers (via `nalgebra`) to compute energy-slope convergence guarantees.
- **Goal Verification Loop** - Integrated target validation kernels with directed LLM corrections and structured `GoalVerdict`/`TaskType` contracts.
- **Symbol Extraction & Verification** - Added runtime parsing, dependency tracing, and semantic verification to enforce correct output states.

Core Capabilities & Providers:

- **Vertex AI Support** - Added seamless Google Cloud Vertex AI provider support with automatic OAuth2 Bearer token resolution via `gcp_auth` ADC.
- **29 LLM Adapters** - Expanded provider configurations to handle 29 unique genai adapters, modernizing model defaults (e.g., `gemini-3.5-flash`, `gpt-5.5`, `claude-fable`).
- **Pipenv Integration** - Added support for `pipenv` environment isolation to the Python execution plugin.

Documentation Overhaul:

- **Stability Foundations** - Reworked core concept and developer documentation to align with the theoretical foundations of the Stability paper series.
- **Sphinx Book Polish** - Redesigned the Sphinx book layout featuring a retro scientific textbook LaTeX cover, custom plots, and enhanced tables.
- **Refined Clinical Tone** - Rewrote the user manual, tutorials, and configuration guides in a formal, clinical documentation style.

Workspace Maintenance & Upgrades:

- **Ratatui Upgrade** - Bumped `ratatui` dependency to `0.30.2` in `perspt-tui` to support modern terminal layouts and components.
- **TUI Textarea Fork Migration** - Migrated from the unmaintained `tui-textarea` to `tui-textarea-2` version `0.11.0` to resolve API compatibility conflicts.
- **Dependency Upgrades** - Bumped `rustyline` to `18.0`, `toml` to `1.1`, and `tower-http` to `0.7` across the workspace.
- **GitHub Actions Security** - Upgraded GitHub Action workflows to use `actions/checkout@v7` for improved stability and security.

Version 0.6.1 - “AKU”

AKU - sharp fixes from sharp ears.

Config Coherency & Schema-Driven Settings:

- **TOML Config Schema & Resolution** - Added a robust TOML configuration schema with smart config-driven provider resolution.
- **Unified Provider Binding** - Bound the configured provider cohesively across all modes including `chat` (TUI), `simple-chat` (CLI), and the autonomous `agent` mode.
- **Refined Config Commands** - Redesigned *perspt config* command to be fully structured, interactive, and improved key initialization/init prompts.

TUI & CLI Slash Commands:

- **CLI Simple-Chat Enhancements** - Integrated `rustyline` file-based history and introduced interactive slash commands inside simple-chat.
- **TUI Input Navigation & Persistence** - Added persistent history paths and implemented fully UTF-8 safe input navigation in TUI inputs.
- **Conversation Persistence** - Added new conversation save commands to easily export dialogues to local files.

Advanced Terminal UI Rendering:

- **LaTeX Math Transpilation** - Integrated real-time LaTeX mathematical transpilation into the markdown rendering pipeline.
- **Intelligent ASCII Table Wrapping** - Implemented self-wrapping logic for ASCII tables in the chat UI to ensure readable presentation on narrow views.

Documentation:

- **Comprehensive Feature Guides** - Fully documented TUI/CLI slash commands, config schemas, and keyboard shortcut matrices in the user guides.
- **PSP-8 Stability Documentation** - Added a dedicated Perspt Book chapter explaining how the three *Stability is All You Need* papers shape agent mode, the SDK-first roadmap, and future domain plugins.

Workspace Maintenance:

- **Crate Version Alignment** - Bumped version of all workspace crates to `0.6.1`.

Version 0.6.0 - “kukuza”

Nurturing the foundation, empowering the core.

Workspace-Wide Dependency Upgrades:

- **duckdb Upgrade** - Bumped `duckdb` requirement to `=1.10503.1` in the workspace root.
- **askama Upgrade** - Bumped `askama` requirement to `0.16` in `perspt-dashboard`.
- **diffy Upgrade** - Bumped `diffy` requirement to `0.5` in `perspt-agent`.
- **genai Upgrade** - Bumped `genai` requirement to `0.6.1` in `perspt-core`.
- **starlark Upgrade** - Bumped `starlark` requirement to `0.14` in `perspt-policy`.

API Alignments & Adaptations:

- **genai API Alignment** - Updated `all_model_names` inside `llm_provider.rs` to pass `()` for the new `ProviderConfig` parameter.
- **starlark API Alignment** - Adapted policy loading in `engine.rs` to use `Module::with_temp_heap` since `Module::new` was deprecated and made private in `0.14`.

Specification & Ecosystem Maintenance:

- **PSP-7 Finalization** - Formally transitioned the PSP-7 specification to `Final` status under the PSP-000001 process.

Version 0.5.9 - “xinli guanghua”

Perfecting the essence until the work needs no words to shine.

PSP-7: Robust Correction Loop Contracts:

- **Structured Artifact Bundle Format** - Switched correction prompt from free-form `File: ...` output to a strict JSON `{ artifacts: [], commands: [] }` schema. Includes exact target paths from evidence so the LLM targets the correct files, reducing parse failures.
- **AgentTools Integration** - Routed correction commands through `execute_correction_command()` to integrate with plugin policy, user approval gates, and tool failure tracking.
- **Typed Parse Pipeline** - Replaced Option-based bundle extraction with a 5-layer fail-closed typed parse pipeline. Added `RetryClassification` (Retarget, MalformedRetry, SupportFileViolation, Replan) population in `CorrectionAttemptRecord`.
- **Manifest Policy Enforcement** - Added semantic validation to prevent implicit mutation of root manifests (Cargo.toml, package.json) unless explicitly listed as output targets, while preserving legal support files.
- **Strict Budget Exhaustion** - Widened budget exhaustion checks from cost-only to `any_exhausted()` to properly respect step and revision caps before attempting LLM calls.

LLM Provider Maintenance:

- **genai Upgrade** - Bumped `genai` dependency from 0.5.1 to 0.5.3 (stable patch release with bug fixes).
- **Dead Code Cleanup** - Removed `generate_response_with_history()` and `generate_response_with_options()` which had zero callers across the workspace and were the only methods leaking `genai::ChatOptions` into the public API surface.
- **Clippy Fixes** - Fixed *clippy::unnecessary-sort-by* and applied *clippy::collapsible-match* auto-fixes for Rust 1.95 compatibility.

Version 0.5.8 - “Qualitaetsveredelung”

Orchestration State Overhaul Release

> “Qualitaetsveredelung - the craft of refining what exists until its quality speaks for itself. Not new features, but the quiet discipline of making every state transition truthful, every metric honest, and every dead path removed.”

Orchestration Correctness (Refs: #112, #113, #114, #116):

- **SessionOutcome enum** - New `SessionOutcome` type (Success, PartialSuccess, Failed) derived from actual completed/escalated node counts. The `Complete` event now carries truthful outcomes instead of unconditional `success: true`.

- **NodeOutcome enum** - `execute_node()` returns `Result<NodeOutcome>` where `NodeOutcome` is `Completed` or `Escalated`, replacing the previous `Result<()>` that could not distinguish outcomes.
- **Correct session outcome derivation** - `run_orchestration()` and `run_resumed_inner()` track completed/escalated counts per node and derive the final `SessionOutcome` accordingly.
- **Always-on LLM telemetry** - `call_llm_with_logging()` now records token usage (in/out), latency, and estimated cost via `record_llm_usage()` after every LLM call, regardless of `--log-llm`. The flag now only controls verbose prompt/response text persistence.
- **Budget envelope persistence** - `upsert_budget_envelope()` called after each `BudgetUpdated` event to persist cost/step tracking to the database.
- **Sandbox-aware context retrieval** - `ContextRetriever` in `step_speculate()` uses `effective_working_dir(idx)` (the node's sandbox directory) instead of the workspace root. Sandbox file tree listings included in actuator and correction prompts for better generation grounding.

Type-Safe State Management (Refs: #114):

- **NodeState::from_display_str()** - Case-insensitive canonical parser with legacy aliases ("running" -> Coding, "stable" -> Completed, "retrying" -> Retry). Replaces all ad-hoc string parsing across the codebase.
- **NodeState helpers** - `is_success()` (true only for Completed), `is_active()` (true for Coding, Verifying, Planning, Retry, SheafCheck, Committing), and `Display` impl producing lowercase labels.
- **CLI state cleanup** - All string-based state comparisons in `status.rs`, `agent.rs`, and `resume.rs` replaced with `NodeState::from_display_str()` and type-safe helper methods.

Dead Code Elimination:

- Removed 16 unused functions across `perspt-core`, `perspt-store`, `perspt-agent`, `perspt-policy`, `perspt-tui` (~234 lines)
- Downgraded `canonicalize` to `pub(crate)` in `perspt-policy`
- Removed orphaned `sha2` dependency from `perspt-store`

Bug Fixes (Refs: #107, #111):

- **Session status stuck at RUNNING** - Status now persisted in `end_session()` with `COALESCE`-based finalization guarantee (#111)
- **LLM token counts always zero** - Real provider token usage (prompt + completion tokens) extracted from `genai ChatResponse::usage` and persisted per request (#107, #110)

Tests:

- 7 new tests covering `NodeState` parsing round-trips, `SessionOutcome` equality, `NodeOutcome` discriminants, and session outcome derivation from completed/escalated counts
- Total test count: 359 (up from 352)

Documentation:

- Updated README: fixed crate count (8 -> 9), deduplicated dashboard command, added missing agent flags (`--single-file`, `--verifier-strictness`, `--output-plan`, all `--*-fallback-model`), aligned contributing commands with CI gates
- Updated Perspt Book: SRBN architecture (Phase 7 -> Commit & Outcome), CLI reference (`logs` always shows token metrics), developer architecture guide (orchestrator lifecycle,

NodeOutcome, SessionOutcome in type inventory and data flow), workspace crates (removed dead `is_safe_for_auto_exec`)
- Updated PSP-5: execution flow steps 8-11 with Completed/Escalated paths and SessionOutcome derivation, headless output with `OUTCOME` line, added Orchestration State Overhaul implementation appendix

Version 0.5.7 - “navikaran”

Dashboard UX Polish Release

> “Bridging the purpose of Ikigai with the momentum of Kaizen - renewal through continuous, intentional refinement.”

Dashboard UX Improvements (PSP-6 continued):
- **Custom DaisyUI 5 themes** - `perspt-light` and `perspt-dark` themes with orange/pink oklch palette (WCAG AA compliant), powered by `@plugin "daisyui/theme"` blocks
- **Theme toggle** - Navbar button with sun/moon icons, localStorage persistence, and migration from legacy theme names
- **Friendly session names** - Deterministic human-readable names (e.g. “bold-hawk”) derived from session UUIDs via hash-indexed adjective+noun arrays
- **Breadcrumb friendly names** - All six session sub-pages show friendly name with UUID-on-hover tooltip
- **Session card layout** - Stacked vertical cards with `btn-outline` sub-page buttons replacing ghost buttons
- **Task text formatting** - `whitespace-pre-line` rendering for readable multi-line task descriptions
- **Collapse arrow fix** - `pe-10` padding on DAG and LLM collapse summaries to prevent arrow overlap with text
- **Decisions page resilience** - All six store queries use `unwrap_or_default()` instead of `?` early-return, preventing 503 errors on partial data
- **Paginated overview** - 20 sessions per page with DaisyUI `join` pagination controls, backed by `list_sessions_paginated()` and `count_sessions()` store methods
- **Login page theme** - Updated to `perspt-light` default

CI & Build:
- **Node.js in CI** - Added `actions/setup-node@v4` (Node 22) to CI test matrix and release workflows so `npx @tailwindcss/cli` runs on all runners

Store:
- `list_sessions_paginated(limit, offset)` - LIMIT/OFFSET SQL for paginated session listing
- `count_sessions()` - Total session count for pagination controls

Version 0.5.6 - “ikigai”

SRBN Sandbox Revision Flow Release

"A reason for being - the happiness of always being busy with what you love."

Real-Time Web Dashboard (PSP-6):

- **perspt-dashboard crate** - Axum 0.8 + Askama 0.15 + HTMX 2 + Tailwind v4/DaisyUI 5 web interface for monitoring agent execution
- **Read-only store access** - `SessionStore::open_read_only()` with DuckDB `AccessMode::ReadOnly` for safe concurrent reads alongside the agent
- **Six monitoring pages** - Overview (sessions), DAG (task graph), Energy (Lyapunov components), LLM (request telemetry), Sandbox (provisional branches), Decisions (escalations, sheaf validations, rewrites, plan revisions, repairs, verifications)
- **SSE live updates** - Server-Sent Events stream node statistics every 2 seconds
- **Password authentication** - Random token, HttpOnly/SameSite cookie, Secure flag on non-localhost deployments
- **``perspt dashboard`` CLI command** - Launches the dashboard server on a configurable port
- **12 integration tests** - Route smoke tests, SSE content-type, auth flow
- **Store extensions** - `get_session_energy_history()`, `get_all_sheaf_validations()`, `get_all_repair_footprints()`

SRBN Sandbox Revision Flow (PSP-5 Phases 3-12):

- **PlanningPolicy** - Adaptive agent gating with 5 policies (LocalEdit, FeatureIncrement, LargeFeature, GreenfieldBuild, ArchitecturalRevision). `needs_architect()` and `needs_speculator()` gate agent tier activation
- **FeatureCharter auto-creation** - Policy-derived file/module/revision limits created before architect planning so the plan gate has bounds to enforce
- **Speculator lookahead gating** - Speculator tier only activates for LargeFeature, GreenfieldBuild, and ArchitecturalRevision policies
- **BudgetEnvelope session restore** - Step/cost/revision caps restored from DB during `resume` so interrupted sessions honour the original limits
- **Bundle path normalization** - `filter_bundle_to_declared_paths` uses `normalize_artifact_path` for correct comparison of path variants (e.g. `./src/main.rs` vs `src/main.rs`)
- **NodeState::Superseded** - New terminal state for plan amendment (Phase 14 preparation). Updated `is_terminal()`, `parse_node_state()`, and `NodeStatus` conversion
- **Orchestrator module extraction** - `orchestrator.rs` split into 9 submodules: `mod.rs`, `bundle.rs`, `commit.rs`, `convergence.rs`, `init.rs`, `planning.rs`, `repair.rs`, `solo.rs`, `verification.rs`
- **Centralized prompts** - All 15 agent prompts consolidated in `prompts.rs` with constants and `render_*` helpers; duplicates removed from `agent.rs`
- **RepairFootprint-backed correction** - `build_correction_prompt` uses `RepairFootprint` for precise, grounded repair context
- **Greenfield bootstrap ordering** - Plugin-driven project initialization with correct pre-sheafify plugin re-detection
- **Provisional branch lifecycle** - Sandbox-first execution with branch creation, merge, and flush cascade
- **Escalation classification** - 5 categories with 9 rewrite actions and graph surgery support

Documentation:

- Updated architecture docs with PlanningPolicy, FeatureCharter, and NodeState::Superseded documentation
- Added Planning Policy and Feature Charter sections to SRBN architecture guide
- Updated workspace crates docs with orchestrator submodule structure
- Added speculator lookahead and budget restore documentation to agent mode guide
- Fixed energy weight default (gamma=2.0) in advanced features guide
- Refreshed all version references to 0.5.6

Version 0.5.5

PSP-5 Cross-Platform and CI Stabilization Release

Cross-Platform Fixes:

- **Windows sandbox path normalization** - `list_sandbox_files` now returns forward-slash-separated relative paths on all platforms
- **Windows workspace-bound validation** - `validate_workspace_bound` correctly detects absolute paths with Windows drive prefixes (`C:\...`) and normalizes backslash path separators before POSIX shell tokenization
- **Clippy and fmt CI compliance** - Resolved `items_after_test_module` and `useless_vec` warnings that were failing CI on all platforms

Build and CI Improvements:

- **Removed accidental eval workspace member** - `.perspt-eval/rust_cli` removed from workspace members; `.perspt-eval/` added to `.gitignore`
- **Stabilized cargo doc** - Added `doc = false` to CLI bin target to prevent output collision with the `perspt` library crate
- **Removed deprecated atty dependency** - Replaced with `std::io::IsTerminal` for TTY detection
- **Lockfile refresh** - Cleared hard `cargo audit` vulnerability failures via dependency updates

Documentation:

- Updated workspace coding instructions to match current multi-crate architecture
- Refreshed all version references across the Perspt Book and Sphinx configuration

Version 0.5.4

PSP-5 Compliance Release

SRBN Agent Enhancements:

- **Per-node error recovery** - Retry logic respects `ErrorType` classification (`Compilation`, `ToolFailure`, `ReviewRejection`) with separate counters
- **Multi-file extraction** - Actuator reliably extracts artifact bundles with multiple files from LLM responses
- **Multi-artifact bundles** - Bundle protocol correctly handles write, diff, and command artifacts in a single node
- **Plugin-driven project initialization** - All plugins use `uv init --lib` (Python), `cargo init` (Rust), `npm init` (JS) for proper project scaffolding

- **Degraded verification mode** - When tool binaries are missing, falls back to heuristic verification with clear warnings
- **Sheaf validation** - 7 validator classes for cross-node contract verification

Bug Fixes:

- Fixed `uv init` to `uv init --lib` in `plugin.rs` (2 locations) and `test_runner.rs` (1 location) for correct `src/` layout with `[build-system]`
- Removed dead `test_check_workspace_requirement` test from `orchestrator.rs`

Documentation:

- Complete rewrite of the Perspt Book for PSP-5 accuracy
- Updated developer guide with full type inventory and architecture diagrams
- Added tutorials for headless mode, Python ETL, Rust CLI, and scientific computing
- Updated all model names and provider defaults to current versions

Version 0.5.3

- SRBN energy convergence improvements
- Provisional branch lifecycle management
- Interface seal and flush cascade support
- DuckDB migration from SQLite

Version 0.5.2

- Initial SRBN orchestrator implementation
- Per-tier model selection (`--architect-model`, etc.)
- Merkle ledger with DuckDB backend
- Plugin system for Python, Rust, and JavaScript
- Agent TUI with dashboard, task tree, and review modal
- Starlark policy engine
- Basic sandbox command execution
- 10 CLI subcommands (chat, agent, simple-chat, init, config, ledger, status, abort, resume, logs)

Version 0.5.1

- Multi-provider chat support
- TUI markdown rendering with code blocks
- Simple CLI mode for scripting
- Streaming response protocol with EOT signal

Version 0.5.0

- Initial release
- Basic chat interface with OpenAI
- Configuration auto-detection from environment

License

Perspt is released under the GNU Lesser General Public License v3.0 (LGPL v3).

LGPL v3 License

Copyright (c) 2025 Ronak Rathoer, Vikrant Rathore

This program is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

What This Means

The LGPL v3 is a copyleft license that provides strong protection for software freedom while allowing linking with proprietary software. Here's what it means in practical terms:

Allowed: What you CAN do

- **Use** Perspt for any purpose, including commercial projects
- **Modify** the source code to fit your needs
- **Distribute** copies of Perspt
- **Link** Perspt as a library in proprietary software
- **Combine** Perspt with software under different licenses
- **Create** derivative works based on Perspt

Required: What you MUST do

- **Provide source code** for any modifications to Perspt itself
- **Include** the LGPL v3 license text with distributions
- **Preserve** copyright notices and license information
- **Allow users** to replace the Perspt library with modified versions
- **Make modified source** available under LGPL v3 terms

Not Provided: What we DON'T provide

- **Warranty** - The software is provided "as is"
- **Liability coverage** - We're not responsible for any damages
- **Support guarantees** - While we strive to help, support is provided on a best-effort basis

Third-Party Licenses

Perspt depends on several open source libraries, each with their own licenses:

Core Dependencies

Crate	License	Description
tokio	MIT	Async runtime for Rust
ratatui	MIT	Terminal user interface library
serde	MIT/Apache-2.0	Serialization framework
clap	MIT/Apache-2.0	Command line argument parser
anyhow	MIT/Apache-2.0	Error handling library
thiserror	MIT/Apache-2.0	Error derive macros

LLM Integration

Crate	License	Description
genai	MIT/Apache-2.0	Unified LLM provider interface
reqwest	MIT/Apache-2.0	HTTP client library

Terminal and UI

Crate	License	Description
crossterm	MIT	Cross-platform terminal library
unicode-width	MIT/Apache-2.0	Unicode character width calculation
textwrap	MIT	Text wrapping and formatting

Development Dependencies

Crate	License	Description
criterion	MIT/Apache-2.0	Benchmarking library
mockall	MIT/Apache-2.0	Mock object library
tempfile	MIT/Apache-2.0	Temporary file management

License Compatibility

The LGPL v3 is compatible with most other open source licenses:

Compatible Licenses: - Apache License 2.0 - BSD licenses (2-clause, 3-clause) - ISC License - MIT License - Public Domain (CC0) - GPL v3+ (can be upgraded to GPL)

Special Considerations: - GPL v2: Not directly compatible due to version differences - Proprietary licenses: Can link with LGPL libraries but must allow library replacement - Copyleft licenses: LGPL provides weaker copyleft than GPL

Commercial Use

Perspt can be freely used in commercial projects:

Allowed Commercial Uses

- **Internal tools** - Use Perspt as part of your development workflow
- **Linked libraries** - Link Perspt as a library in commercial software
- **Service offerings** - Provide Perspt as part of consulting or hosting services
- **Modified library versions** - Create modified versions for internal use
- **Enterprise solutions** - Build enterprise tools that use Perspt

Requirements for Commercial Use

1. **Include LGPL license text** in your distribution
2. **Maintain copyright notices** from the original code
3. **Provide source code** for any modifications to Perspt itself
4. **Allow library replacement** - users must be able to replace the Perspt library
5. **No trademark usage** without permission (see below)

No additional fees, registrations, or permissions are required.

Trademark Policy

While the source code is LGPL v3 licensed, trademarks are handled separately:

“Perspt” Name and Logo: - The name “Perspt” and any associated logos are trademarks - You may use the name in accurately describing the software - Commercial use of the name/logo as your own brand requires permission - Modified versions should use different names to avoid confusion

Acceptable Uses: - “Built with Perspt” - “Based on Perspt” - “Powered by Perspt” - “Fork of Perspt”

Requires Permission: - Using “Perspt” as your product name - Using Perspt logos in your branding - Implying official endorsement

Contributing and License

By contributing to Perspt, you agree that:

1. **Your contributions** will be licensed under the same LGPL v3 License
2. **You have the right** to license your contributions under LGPL v3
3. **You understand** that your contributions may be used commercially
4. **You retain copyright** to your contributions while granting broad usage rights

Contributor License Agreement (CLA)

For substantial contributions, we may request a Contributor License Agreement to:

- Ensure you have the right to contribute the code
- Provide legal protection for the project and users
- Allow for potential future license changes if needed
- Clarify the rights and responsibilities of contributors

License FAQ

Q: Can I use Perspt in my proprietary software? A: Yes, LGPL v3 allows linking with proprietary software. You must provide the library source and allow replacement.

Q: Can I modify Perspt and sell the modified version? A: Yes, but you must provide the source code for your modifications under LGPL v3.

Q: Do I need to open source my modifications? A: Yes, any modifications to Perspt itself must be made available under LGPL v3.

Q: Can I remove the copyright notices? A: No, you must preserve the copyright notices and license information in all copies.

Q: What if I only use parts of the code? A: The LGPL v3 license still applies to any substantial portions you use.

Q: Can I change the license of my derivative work? A: You can license your own code separately, but Perspt parts must remain LGPL v3.

Q: Do I need to attribute Perspt in my application? A: Yes, you must include the LGPL v3 license and copyright notices.

Getting Legal Advice

This page provides general information about the LGPL v3 License and is not legal advice. For specific legal questions:

- **Consult** with a qualified attorney
- **Review** the full license text carefully
- **Consider** your specific use case and jurisdiction
- **Seek** professional legal counsel for commercial decisions

Reporting License Issues

If you believe there's a license violation or have questions about licensing:

- **Email**: legal@perspt.dev
- **GitHub Issues**: License Questions
- **Include** specific details about the concern or question

We take licensing seriously and will investigate all reports promptly.

See also

- Acknowledgments - Credits and thanks to contributors
- Contributing - How to contribute to the project
- GNU Project - Official LGPL v3 License text

Acknowledgments

Perspt is built on the shoulders of giants. We extend our gratitude to the many open-source projects, libraries, and communities that made this project possible.

Theoretical Foundation

The SRBN engine at the heart of Perspt's experimental agent mode is based on the *Stability is All You Need* paper series:

Paper I: *Lyapunov-Guided Hierarchies for Long-Horizon LLM Reliability*
Paper II: *SRBN-Control Beyond Harness Engineering*
Paper III: *The SRBN Platform Contract*

The specifications draw on the series for measured residual energy, bounded correction, and a separation between model proposals and governed effects. Perspt serves as an experimental coding implementation of these ideas. The coding domain uses measured discrete gates and declares its continuous analytic constants `NotClaimed`. Paper assumptions and illustrative figures do not establish a measured success rate for Perspt.

Core Dependencies

AI and LLM Integration

genai

The foundation of Perspt's multi-provider support. This exceptional crate provides unified interfaces to multiple AI providers and supplies the adapters used by Perspt's configured model routes.

- **Project**: genai
- **License**: MIT/Apache 2.0
- **Impact**: Enables seamless integration with OpenAI, Anthropic, Google, Groq, Cohere, XAI, DeepSeek, and Ollama providers

serde & serde_json

Rust's premier serialization framework, powering Perspt's configuration management and API communication.

- **Project**: serde
- **License**: MIT/Apache 2.0
- **Impact**: JSON configuration parsing, API request/response handling

User Interface and Terminal

ratatui

The modern, feature-rich TUI framework that powers Perspt's interactive terminal interface.

- **Project**: ratatui
- **License**: MIT
- **Impact**: Rich terminal UI, markdown rendering, scrollable chat interface

crossterm

Cross-platform terminal manipulation library enabling consistent behavior across operating systems.

- **Project**: crossterm
- **License**: MIT
- **Impact**: Keyboard input handling, terminal control, cross-platform compatibility

Async Runtime and Concurrency

tokio

The asynchronous runtime that enables Perspt's responsive, non-blocking architecture.

- **Project**: tokio
- **License**: MIT
- **Impact**: Async/await support, concurrent LLM requests, responsive UI

Error Handling and Utilities

anyhow

Elegant error handling that makes Perspt's error messages helpful and actionable.

- **Project**: anyhow
- **License**: MIT/Apache 2.0
- **Impact**: Comprehensive error context, user-friendly error messages

clap

Command-line argument parsing that makes Perspt easy to use and configure.

- **Project**: clap
- **License**: MIT/Apache 2.0
- **Impact**: CLI interface, help generation, argument validation

Documentation Tools

Guidedog

The Rust-native documentation, RFC (*guidedog dsc*), and static site engine (*guidedog ssg*) powering the Perspt Book, PSP Catalog, and unified site.

- **Project**: Guidedog
- **Impact**: Native reStructuredText & MyST compilation, RFC governance, multi-mount static site generation

Typst

The modern typesetting system used to compile the Perspt Book PDF, PSP technical papers, and vector SVG architecture diagrams.

- **Project**: Typst

- **License**: Apache 2.0
- **Impact**: High-contrast editorial PDF typesetting, vector diagrams, mathematical notation

Development Tools

Rust Language
: The systems programming language that makes Perspt fast, safe, and reliable.
  - **Project**: Rust
  - **License**: MIT/Apache 2.0
  - **Impact**: Memory safety, performance, excellent tooling ecosystem

cargo
: Rust's package manager and build system that makes development smooth and dependency management effortless.
  - **Project**: Part of Rust toolchain
  - **License**: MIT/Apache 2.0
  - **Impact**: Dependency management, build automation, testing framework

Community and Inspiration

AI Provider Communities

OpenAI
: For creating GPT models and establishing many of the patterns that define modern AI interaction.

Anthropic
: For Claude models and their pioneering work in AI safety and helpful, harmless, and honest AI systems.

Google
: For Gemini models and their contributions to accessible AI technology.

Groq
: For ultra-fast inference infrastructure and democratizing AI speed.

Cohere
: For enterprise-grade language models and excellent developer tools.

XAI
: For Grok models and advancing conversational AI capabilities.

DeepSeek
: For their contributions to the open-source AI ecosystem.

Ollama
: For making local AI model hosting accessible and user-friendly.

Open Source Ecosystem

GitHub
: For providing the platform that enables collaborative development and open-source sharing.

crates.io
: Rust's package registry that makes sharing and discovering Rust libraries effortless.

docs.rs
: For automatically generating and hosting documentation for Rust crates.

Terminal and CLI Inspiration

The terminal and CLI interface draws inspiration from many excellent tools:

- **htop** - For showing how terminal UIs can be both beautiful and functional
- **tmux** - For terminal multiplexing concepts and keyboard navigation patterns
- **vim/neovim** - For modal editing concepts and efficient keyboard shortcuts
- **fzf** - For demonstrating responsive, interactive terminal interfaces

Rust Community Projects

Many patterns and approaches in Perspt were learned from studying excellent Rust projects:

- **ripgrep** - For performance optimization and user experience design
- **bat** - For beautiful terminal output and syntax highlighting
- **exa/eza** - For modern CLI design and colored output
- **gitui** - For TUI application architecture and event handling

Testing and Quality Assurance

Users and Beta Testers
: The early adopters and users who provided feedback, reported bugs, and suggested improvements.

Security Researchers
: For responsible disclosure of security issues and helping make Perspt more secure.

Documentation Reviewers
: For helping improve the clarity and completeness of this documentation.

Special Thanks

AI Safety Research Community
: For ongoing work to make AI systems more reliable, interpretable, and aligned with human values.

Open Source Contributors
: To everyone who contributes to open-source projects, from major features to documentation fixes.

Rust Community
: For creating and maintaining an inclusive, helpful community that makes Rust development a joy.

Terminal Enthusiasts
: For keeping the art of terminal-based applications alive and pushing the boundaries of what's possible in text-based interfaces.

Contributing Back

Perspt aims to be a good citizen of the open-source ecosystem. We contribute back by:

Open Source Release
: Perspt itself is released under the LGPL v3 license, allowing anyone to use, modify, and distribute it.

Documentation Standards
: This comprehensive documentation serves as an example of thorough project documentation.

Best Practices Sharing
: Through blog posts, talks, and code examples, we share what we've learned building Perspt.

Upstream Contributions
: When we find bugs or missing features in dependencies, we contribute fixes and improvements back to those projects.

License Information

Perspt is licensed under the LGPL v3 License. For complete license information, see License.

All dependencies are used in accordance with their respective licenses. We are grateful to all the authors and maintainers who choose to share their work under permissive open-source licenses.

Get Involved

Want to contribute to Perspt or the broader ecosystem?

Report Issues
: Help improve Perspt by reporting bugs, suggesting features, or improving documentation.

Contribute Code
: See our Contributing guide for how to contribute code improvements.

Share Knowledge
: Write blog posts, create tutorials, or give talks about your experience with Perspt.

Support Dependencies
: Consider contributing to the open-source projects that Perspt depends on.

Spread the Word
: Help others discover Perspt and the amazing ecosystem of Rust and AI tools.

—

Thank you to everyone who makes open-source software development possible. Your contributions, large and small, make projects like Perspt possible.