spectroscope · full build

AGENT ORCHESTRATOR · JAVA 21 · USER GUIDE & TECHNICAL REFERENCE

the complete **spectroscope** user guide

One headless core, many faces. Every feature of the full build, from the terminal, the web UI, the desktop shell, the Lab, subagents, skills, MCP, voice and vision to the fleet on the bus, explained with real screenshots, and every protocol, event and endpoint documented down to the wire.

7	18	18+	3
MODULES ON ONE STREAM	RUNEVENT TYPES	TOOLS (+ MCP)	LLM PROVIDERS
545	310	3	7
JUNIT TESTS	VITEST TESTS	DESIGNS (DARK · BRIGHT · WHITE)	BUILT-IN SCENARIOS

ABOUT THIS GUIDE

How to read this book

A NOTE ON THE SCREENSHOTS

Every screenshot is a real capture of the running app, taken by the repository's own capture pipeline (`capture_screens.mjs`) and reshot per release. The two editions of this guide carry matching sets: the dark pages show **spectro dark**, the light pages **spectro bright**.

spectroscope is the reference harness of the workshop "build an agent harness from scratch" — and the full build in the `spectroscope/` folder is its final form: everything the ten core stages and six bonus stages construct, plus the conveniences that never fit a single stage. This guide documents that full build. Not the journey — the machine.

The book has five parts, ordered from "I just want to use it" to "I want to know exactly what crosses which wire":

- **Part I — Meet spectroscope**: what it is, how to start it, what it can do.
- **Part II — The faces**: the terminal, the web UI with all its tabs and panels, the Lab, session management, the design system, the desktop app.
- **Part III — The agent's powers**: tools, permissions, hooks, subagents, skills, providers, vision, voice, image generation, MCP and scheduling.
- **Part IV — The file system**: where everything lives on disk and what each file looks like inside.
- **Part V — Under the hood**: the complete technical reference — the architecture dossier, every event, every socket frame, every REST endpoint, every configuration key.

HONESTY RULES OF THIS DOCUMENT

Every screenshot in this guide is a real capture of the running application (English chrome, taken 2026-07-20 with the reproducible script in `docs/guide-assets/capture_screens.mjs`). Every terminal block is real command output. Every name, constant, wire string and default was read from the migration phase-6 source tree (2026-07-20) — not from memory. Where behaviour is a deliberate limitation, the guide says so.

Conventions

- `monospace` marks anything that exists literally in code, config or on the wire: tool names, event types, JSON keys, commands.
- Chips classify tools: `free` needs no permission, `gated` asks first, `gated + prefix rule` asks first and remembers approvals scoped to a prefix.
- German UI labels appear in parentheses where the chrome ships bilingual — the screenshots use the English chrome.
- File paths are relative to the `spectroscope/` root unless they start with `~`.

CONTENTS

The map of this book

PART I

Meet spectroscope

What the harness is, the one idea everything hangs on, how to start it in thirty seconds, and a map of everything it can do.

CHAPTER 1

What is spectroscope?

spectroscope is an agent harness: the machinery around a large language model that turns “a model that can talk” into “an agent that can work” — read files, run commands, ask for permission, delegate to subagents, remember sessions, and show you every step it takes while doing so.

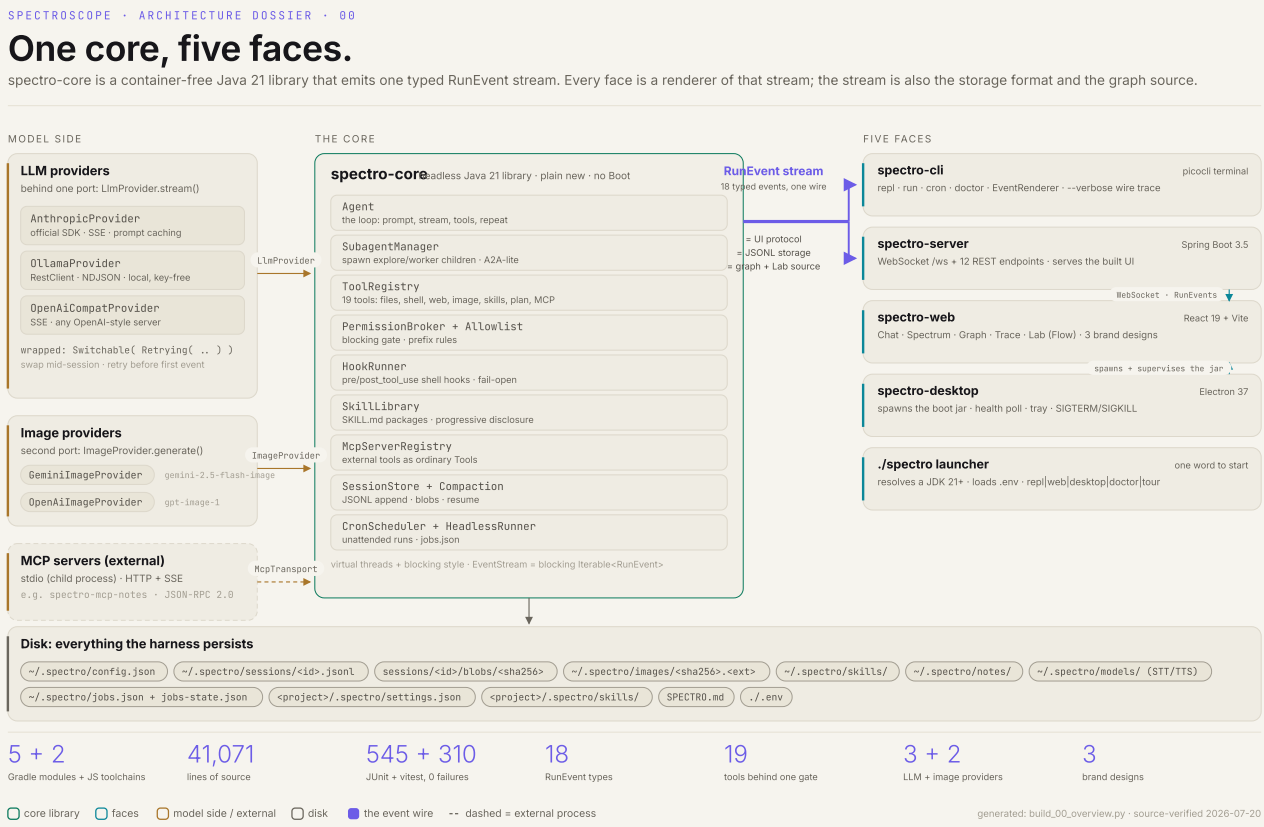
It was built from scratch across a sixteen-stage workshop (Java 21, Gradle, Spring only where it earns its place), and the `spectroscope/` full build is the sum of all of it. Three properties define the design:

One core, five faces

The heart of spectroscope is `spectro-core` — a headless Java library with no container, no UI and no I/O of its own. Everything in it is constructible with plain `new`. When the agent runs, the core produces exactly one thing: a typed stream of **RunEvents**. Everything else in the project is a *renderer* of that stream:

- the **CLI** (`spectro-cli`) renders it as ANSI text with tool cards and a spinner,
- the **web backend** (`spectro-server`) pushes it over a WebSocket,
- the **browser UI** (`spectro-web`) folds it into chat, graph, trace, Lab and panels,
- the **desktop shell** (`spectro-desktop`) wraps that same web UI in a supervised window,
- the **session store** writes it line by line into a JSONL file.

That last renderer is the trick: the same event stream is simultaneously the **UI protocol**, the **storage format** and the **data source for every visualisation**. Replaying an old session is not a separate code path — it is the same array of events, read from a file instead of a socket.



The big picture: the container-free core, the RunEvent stream as the one wire, the five faces, the provider ports and the disk.

Additive forever

The JSONL wire format is shared byte-for-byte with the sibling TypeScript edition of the workshop. That makes the format a contract: new event types and new optional fields may be added, but nothing is ever renamed, removed or reinterpreted, and every consumer silently skips what it does not know. Sessions recorded in 2026 replay forever; sessions recorded by the other edition replay here (proven by a byte-for-byte round-trip test).

Tool inputs are model output

The course maxim that shapes every security decision in this book: whatever a tool receives as input was written by the model, and the model writes what it wants. So every file path passes a sandbox, every mutation passes a permission gate, network egress asks first, hooks can veto calls before the gate, and even the “always allow” button remembers its approval scoped to a prefix instead of a blanket. You will meet this maxim in almost every chapter.

The stack, in one breath

LANGUAGE	Java 21 — virtual threads + blocking style; no reactive frameworks. The stream type is a blocking `Iterable<RunEvent>`, cancellation is a cooperative `CancelSignal`.
BUILD	Gradle 9.6.1, Kotlin DSL, version catalog, five JVM modules + two npm toolchains (Vite/React web UI, Electron shell).
LLM ACCESS	One narrow `LlmProvider` port; behind it Anthropic (official SDK, SSE), Ollama (NDJSON) and any OpenAI-compatible server (SSE) — switchable mid-session.
FRAMEWORKS	Spring *Framework* as a library in the core (RestClient + declarative HTTP interfaces); Spring *Boot* 3.5.3 only in `spectro-server`; picocli in the CLI; React 19 + React Flow in the browser.

TESTS	354 JUnit + 252 vitest, all key-free: fake providers, scripted HTTP servers, process seams. `./gradlew build` needs no API key and no network model.

CHAPTER 2

Quick start

One executable starts everything. The `./spectro` launcher resolves a JDK, loads your `.env`, and dispatches to the face you ask for — you never have to touch the Gradle task zoo.

The launcher

```
$ ./spectro

◆ spectroscope

  repl          interactive agent REPL — /voice push-to-talk, --speak reads aloud
                ./gradlew -q --console=plain :spectro-cli:run --args="..."
  run           headless run: ./spectro run -p "task" [--json] [--image path] [--speak]
                ./gradlew -q --console=plain :spectro-cli:run --args="run -p ..."
  cron          scheduler: list / status / run <id> (~/.spectro/jobs.json)
                ./gradlew -q --console=plain :spectro-cli:run --args="cron list"
  sessions      list the stored JSONL sessions (~/.spectro/sessions)
                ./gradlew -q --console=plain :spectro-cli:run --args="sessions"
  resume        continue a session by id: ./spectro resume <id>
                ./gradlew -q --console=plain :spectro-cli:run --args="--resume <id>"
  doctor        check Java, config layers, provider reachability
                ./gradlew -q --console=plain :spectro-cli:run --args="doctor"
  tour          guided feature tour — menu, tips, settings
                ./gradlew -q --console=plain tour
  web           web face — Spring Boot server + browser UI on :8080
                ./gradlew -q --console=plain :spectro-server:bootRun
  desktop       desktop face — Electron shell + managed Spring Boot process
                ./gradlew :spectro-server:bootJar && (cd spectro-desktop && npm start)
  mcp-notes     build the example MCP server (stdio) into a runnable dist
                ./gradlew -q --console=plain :spectro-mcp-notes:installDist

Extra arguments pass through: ./spectro repl --verbose · ./spectro run -p 'Say OK.' --json
Vision needs a vision model (attach via --image or the web composer); voice: bash scripts/setup-stt.sh
(in) and scripts/setup-tts.sh (out).
MCP: configure external servers in .spectro/settings.json ('mcpServers'); ./spectro mcp-notes builds the
example; in the REPL, /mcp lists them.
```

Before dispatching, the launcher fixes the two classic host problems for you:

- **JDK auto-resolution.** The modules compile with `--release 21`, but many machines default to an older `java`. If `JAVA_HOME` is not already a JDK 21+, the launcher probes `/usr/libexec/java_home`, the Homebrew `openjdk@21` keg (preferred: a stable versioned keg survives brew upgrades), newer kegs up to 25, and the Cellar globs — and exports the first hit. Verified up to JDK 26; a miss prints a note and continues.
- **`.env` loading for every face.** The gitignored `spectroscope/.env` holds API keys and provider switches. The launcher parses it (comments skipped, one quote layer stripped, empty values ignored so `KEY=` can never blank a shell export) and exports the pairs — deliberately including the `desktop` face, which spawns the server through Electron and would otherwise bypass Gradle's own injection.

The thirty-second start

```
cd spectroscope

# 1 — pick a model. Local (no key at all):
#    .env:  SPECTRO_PROVIDER=ollama · SPECTRO_MODEL=qwen3 · SPECTRO_BASE_URL=http://localhost:11434
#    cloud: ANTHROPIC_API_KEY=sk-ant-…      (model defaults to claude-opus-4-8)

# 2 — check the environment
./spectro doctor

# 3 — go
./spectro web        # browser UI on http://localhost:8080
./spectro repl       # terminal REPL
./spectro run -p "Say OK."   # headless one-shot
./spectro desktop    # the Electron app
```

What doctor tells you

`./spectro doctor` is the honest pre-flight: it probes what your configuration actually selects — it loads the config hierarchy with your flags, pings the provider you chose, connects to every configured MCP server, and write-tests the sessions directory. This is real output from the machine this guide was built on:

● $ ./spectro doctor

```
◆ spectro doctor
  ✓ Java 25.0.2
  ✓ config: provider=ollama model=qwen3.5:27b permissionMode=ask autoApprove=2 rule(s)
    layers: user config absent · project settings present
  ✓ ollama at http://localhost:11434 (version 0.24.0)
    images: gemini — GEMINI_API_KEY not set; generate_image will return a readable error
  ✓ skills: 4 installed (brainstorming, test-driven-development, verification, writing-plans)
  ✓ hooks: 0 configured
  ✓ mcp: notes reachable at /Users/you/spectroscope/spectro/spectro-mcp-notes/build/install/spectro-mcp-notes/bin/spectro-mcp-notes (2 tools)
    vision: local provider — attach images only with a vision model (e.g. ollama pull qwen3-vl); a text-only model fails fast
  ✓ voice input: whisper-cli + ggml-small.bin ready (/voice)
    voice output: piper missing · voice missing — run bash scripts/setup-tts.sh to enable --speak
  ✓ sessions dir writable: /Users/you/.spectro/sessions (7 session(s))
  ✓ jobs.json: 0 job(s)

Everything looks good.
```

Green checks fail the exit code when broken; the indented dim lines are information, not failures — a missing image-generation key, for example, is deliberately not unhealthy (the tool will return a readable error instead). Chapter 4 documents all twelve checks.

The provider matrix

PROVIDER	TRANSPORT	NEEDS	DEFAULT MODEL
`anthropic`	official SDK, SSE streaming	`ANTHROPIC_API_KEY`	`claude-opus-4-8`
`ollama`	Spring RestClient, NDJSON	local Ollama on :11434	`qwen3`

`openai`	Spring RestClient, SSE	any OpenAI-compatible server (LM Studio, llama.cpp, vLLM); key optional	`local-model`

Selection rides the settings hierarchy (chapter 30 has every key): defaults < environment (`SPECTRO_PROVIDER` / `SPECTRO_MODEL` / `SPECTRO_BASE_URL`, usually from `./.env` — the BASE layer) < `~/.spectro/settings.json` (user) < launch-dir `.spectro/settings.json` (deprecated) < workspace `.spectro/settings.json` + `settings.local.json` < CLI flags — any settings file outranks the env. “Local Ollama for this checkout” is three uncommented lines in `.env` as long as no settings file overrides them — and in the web UI you can switch provider and model mid-session from the header.

Configuration precedence, low to high. Each layer only fills what the higher layers left unset.

CHAPTER 3

The capability map

Everything the full build can do, on two pages. Each row names the chapter where the details live.

CAPABILITY	IN ONE SENTENCE	CH.
Terminal REPL	Interactive agent with tool cards, spinner, slash commands, voice input and spoken answers.	4
Headless runs	`spectro run -p "…"` — one prompt, exit codes for CI, NDJSON event output with `--json`.	4
Cron scheduling	Jobs in `~/.spectro/jobs.json` run unattended with desktop notifications and audit sessions.	19
Web chat	Streaming chat with markdown answers, collapsible reasoning, tool cards with gate chips, image attachments, push-to-talk.	5
Graph view	Every run as a BPMN-style flow overview or a dagre DAG — live and for any archived session.	6
Trace view	The Wireshark of the harness: every wire frame both directions, filters, Δt, LLM-direction column, three detail modes.	6
Right panel	Agents roster, the agent's live plan, the exact system context per agent, and a sandboxed file browser.	6
The Lab	A step-through debugger for agent runs: dam up the event stream, advance it click by click across three synchronized views.	7
Scenarios	Seven deterministic, compiled demo runs — no LLM, no key — for teaching every mechanism.	7
Sessions	Every run is a JSONL file: list, replay, resume live (with the context re-upload made visible), delete with a two-step guard.	8
Import	Replay foreign files: raw spectroscope JSONL verbatim, Claude Code transcripts through an adapter.	8
Design system	Three brand designs — spectro dark, spectro bright, spectro white — switchable in place, particle/scroll effects, one-step persistence.	9
Desktop app	Electron shell that spawns and supervises the server JVM: health checks, tray, native cron notifications.	10
DE/EN chrome	The whole UI chrome switches between German and English live; content keeps its language.	11
Tool belt	18 built-in tools — files, shell, search, web fetch, plan, skills, spawns — plus every connected MCP tool.	12
Permission system	ask/auto/readonly modes, a blocking broker, prefix-scoped allowlist, “always allow” with optional persistence.	13
Hooks	Config-driven shell guards around every tool call: `pre_tool_use` can block before the gate, `post_tool_use` observes.	13
Subagents	Explore/worker children with their own context, visible A2A task/status/result protocol, four development role tools.	14
Skills	`SKILL.md` packages with progressive disclosure — catalog in the prompt, body on demand.	15

Provider switch	Change LLM backend and model mid-session from the header; history intact; refused cleanly without a key.	16
Resilience	Transient-error retry with backoff (spectroscope owns it, SDK retry off) and Anthropic prompt caching with an honest compaction trigger.	16
Vision	Attach images in the composer or `--image`; blobs stay next to the session file, prompts stay small.	17
Voice in/out	Local whisper.cpp push-to-talk (editable transcript) and piper speech output that reads answers while they stream.	17
Image generation	`generate_image` behind its own provider port (Gemini / OpenAI), content-addressed store, gallery panel.	17
MCP client	External tool servers over stdio or HTTP/SSE, Claude-Desktop-shaped config, at-most-once calls, example notes server included.	18
Context management	Automatic compaction at 100k tokens, forced `/compact`, live context ring with per-part introspection.	28
Wire tracing	`--verbose` mirrors the full agent↔provider protocol to stderr; the Trace tab does it in the browser.	4, 6

PART II

The faces

Five renderers of one event stream: the terminal, the web UI with its tabs and panels, the step-through Lab, the session archive, the design system, and the desktop shell.

CHAPTER 4

The command line

The CLI is the first face the workshop builds and the most direct one: a picocli application whose root command is an interactive REPL, with subcommands for headless runs, scheduling and diagnostics. Everything it prints is a rendering of the same RunEvent stream that the web UI folds and the session store writes.

Commands and global flags

INVOCATION	WHAT IT DOES
`spectroscope`	the interactive REPL (root command, no subcommand)
`spectro run -p "…"`	headless single prompt (see below)
`spectroscope cron [list|status|--once id]`	the scheduler — chapter 19
`spectro sessions`	list stored sessions (a positional, not a subcommand)
`spectroscope --resume <id>`	continue a stored session in the REPL
`spectro doctor`	the environment check
`./spectro tour`	the interactive menu tour of all faces

Global flags on the root command apply to every entry point (subcommands reach them via picocli's `@ParentCommand` — so `flags > env` holds on `run` and `doctor` too):

FLAG	MEANING
`--provider` / `--model` / `--base-url`	override the configured backend for this invocation
`--compaction-threshold <n>`	compaction trigger in input tokens (default 100 000)
`--resume <id>`	rebuild the conversation from a session file and continue it
`--verbose`	mirror the agent↔provider wire protocol to stderr in cyan
`--speak`	read answers aloud while they stream (voice output, chapter 17)

The REPL

Startup prints a banner naming the provider and model (for Ollama it live-probes the server version), the session id and file path, and the essential key bindings. The prompt is a coral `›`. Each turn drives three consumers in order — the JSONL store, the speech renderer, the ANSI renderer — the CLI-side proof of “one stream, many renderers”.

```
◆ spectroscope  full build
  ollama · gpt-oss:20b · ollama 0.24.0 · images: gemini · skills: 4 · allowlist active
  session 20260716-091500-3fa4b2c1 · ~/.spectro/sessions/20260716-091500-3fa4b2c1.jsonl
  /help for commands · /mcp servers · /voice push-to-talk · /speak on|off reads answers aloud · Ctrl+C
aborts a run
```

Rendering rules worth knowing (all in `EventRenderer`):

- **Thinking is dimmed.** A reasoning model's `thinking_delta` stream prints dim under a `· thinking` label, visually apart from the answer.
- **Tool cards.** Each call prints `⚒ name` (coral hammer) with a 100-char input preview; the result line shows `✓/✗ <ms> · output preview`.
- **Permissions inline.** A gated call prints a sand `run <tool>? [y/N]` prompt; only `y` approves. Allowlisted calls print `✓ auto-approved by allowlist` instead — the decision is still recorded in the stream.
- **Subagents in brackets.** Child events render as `[worker-1] …` lines with line-buffered text so parallel children stay readable; A2A task/status/result messages appear as one dim line each.
- **The plan pretty-print.** An `update_plan` call renders `◇ plan (N steps)` with `[x]` done, coral `[~]` running, `[ ]` open.
- **Degradation.** Colors and the braille spinner appear only on a real TTY (and never with `NO_COLOR` or `TERM=dumb`); piped output is plain text.

Slash commands

Handled entirely in the CLI — none of these ever reaches the model:

COMMAND	BEHAVIOUR
`/help`	lists all twelve commands with one-liners
`/cost`	cumulative session tokens: `Session usage: X in / Y out`
`/model`	current provider · model (+ base URL for local providers)
`/sessions`	the stored-session listing, same as `spectro sessions`
`/skills`	installed skills with descriptions (chapter 15)
`/mcp`	connected MCP servers, reachability, and every `mcp__…` tool (chapter 18)
`/think on|off`	toggles reasoning visibility; rebuilds the agent and reloads the session history so nothing is lost
`/voice`	push-to-talk: record → transcribe → editable confirmation (chapter 17)
`/speak on|off`	voice output at runtime; `off` also stops the current sentence
`/compact`	force a context compaction now (`Agent.compactNow()`); prints the result or “Nothing to compact”
`/clear`	fresh agent, fresh session file: `New session: <id>`
`/exit`	leave (empty line or Ctrl-D work too)

Headless runs: `spectro run`

The automation face. One prompt, one process, honest exit codes:

```
# plain: prints only the final answer text
./spectro run -p "Count the Java files under spectro-core."

# NDJSON: every RunEvent on stdout, one per line — byte-identical to the JSONL format
./spectro run -p "Say OK." --json | jq -c '{type}'

# guard rails
./spectro run -p "…" --max-turns 5 --permissions readonly
```

FLAG	MEANING
`-p, --prompt`	required task text
`--json`	NDJSON events on stdout; ALL diagnostics go to stderr, so `| jq` pipelines stay clean
`--max-turns <n>`	cancel when a turn exceeds the limit (stop reason `max_turns`)
`--permissions readonly|auto`	the headless policy. Default **readonly** — "capability before convenience": unattended runs deny mutations unless you opt in
`--image <path>`	attach an image (repeatable; jpg/jpeg/png/webp/gif) — vision, chapter 17
`--verbose` / `--speak`	as on the REPL

Exit code contract: 0 only when the run ended with a regular `end_turn`. Anything else (aborted, max turns, error) exits 1 and names the stop reason and the session id on stderr — the session file is always there for post-mortem inspection.

The twelve doctor checks

#	CHECK	WHAT IT VERIFIES
1	Java runtime	runtime major version ≥ 21
2	Config hierarchy	the layered config loads; shows provider/model/permissionMode/allowlist count and which layers exist. A broken config stops doctor immediately — nothing below would make sense
3	Provider reachability	anthropic: key present · ollama: live version probe · openai: GET `/v1/models` (2 s/3 s timeouts)
4	Image provider	key for gemini/openai; a missing key is informational, not unhealthy
5	Skills	count + names of installed skills
6	Hooks	count + `event:matcher` pairs
7	MCP servers	connects each configured server: reachable + tool count, or a red UNREACHABLE line (never a crash)
8	Vision	informational hint matching your provider (vision model needed locally)
9	Voice input	`whisper-cli` on PATH + the ggml model file present
10	Voice output	piper binary + voice model present
11	Sessions dir	creates the dir, writes and deletes a probe file, counts sessions
12	jobs.json	the cron file parses (loudly invalid otherwise)

Wire tracing with `--verbose`

A decorator around the provider mirrors the whole protocol to **stderr** in cyan: the full outgoing request before each call (message count, tool names, clipped system prompt, every content block) and each incoming provider event as it streams (`← text_delta "…"`, `← tool_call name {…}`,

`← usage 123 in / 45 out`, `← stop end_turn`). Because it writes to stderr, `spectro run --json | jq` keeps a clean stdout. In the REPL the traced instance is shared with subagents — you see the children's requests too.

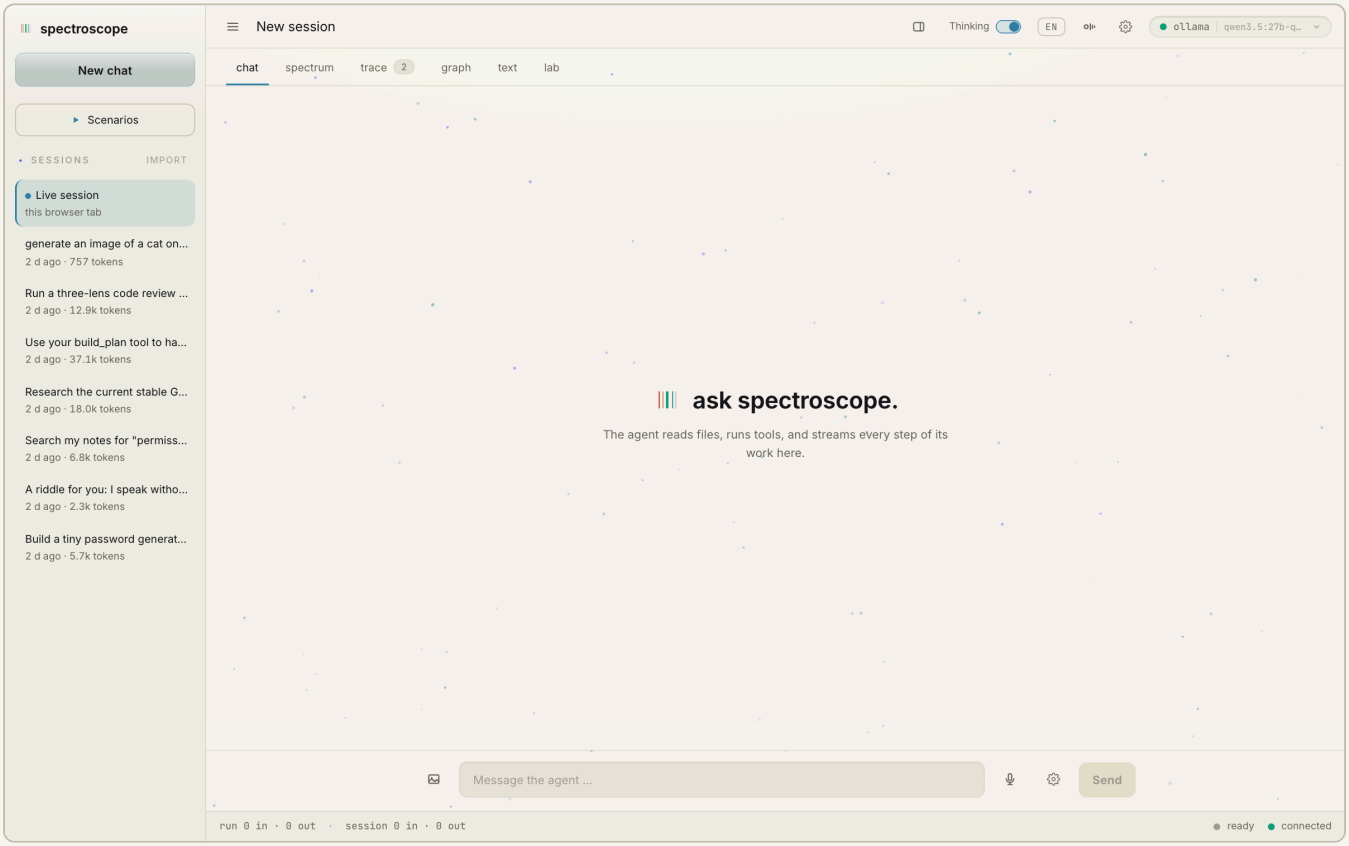
The tour

`./spectro tour` is a menu-driven guide around all faces: launch the REPL, list or resume sessions, do a headless run, drive cron, start the web jar, run doctor — plus a settings screen that captures a provider choice and API key for the session (hidden input, optional save to `./.env` with mode 600) and stage-appropriate “try this” tips. The same file ships in every workshop stage and detects at runtime which features exist; in the full build it shows everything.

CHAPTER 5

The web face: Chat

One Spring Boot jar serves everything: the built React UI, a handful of REST endpoints, and a single WebSocket that carries RunEvents in one direction and a small set of client frames in the other. Open `http://localhost:8080` after `./spectro web` and you are looking at the chat.

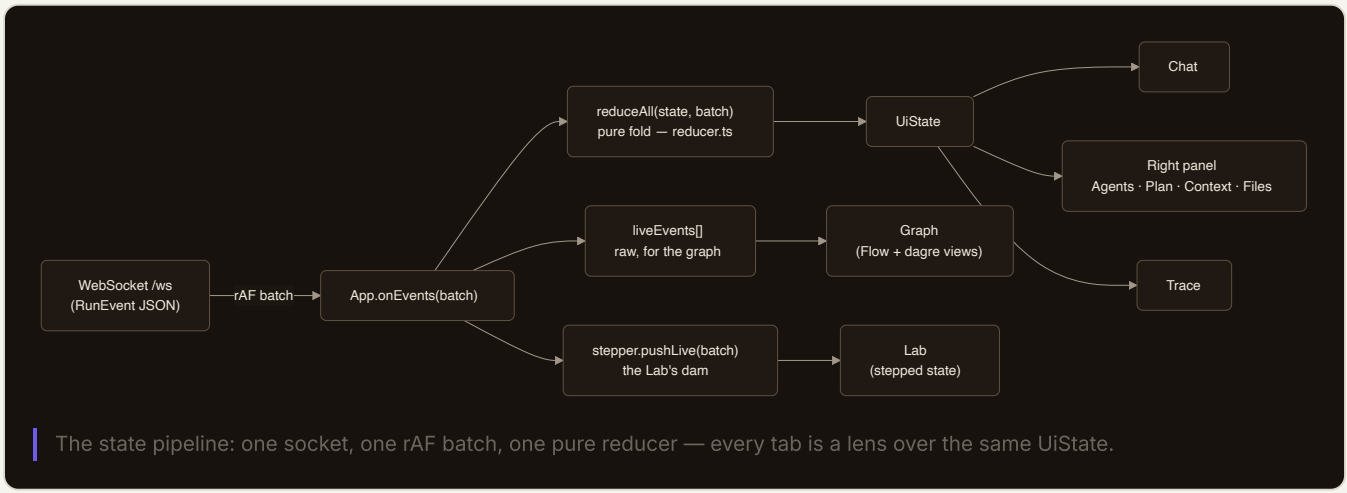


The empty app: sidebar with the session archive, header with thinking toggle · language toggle · design drawer · provider chip, the four tabs, and the composer waiting. “Ask spectroscope.”

The anatomy of the screen

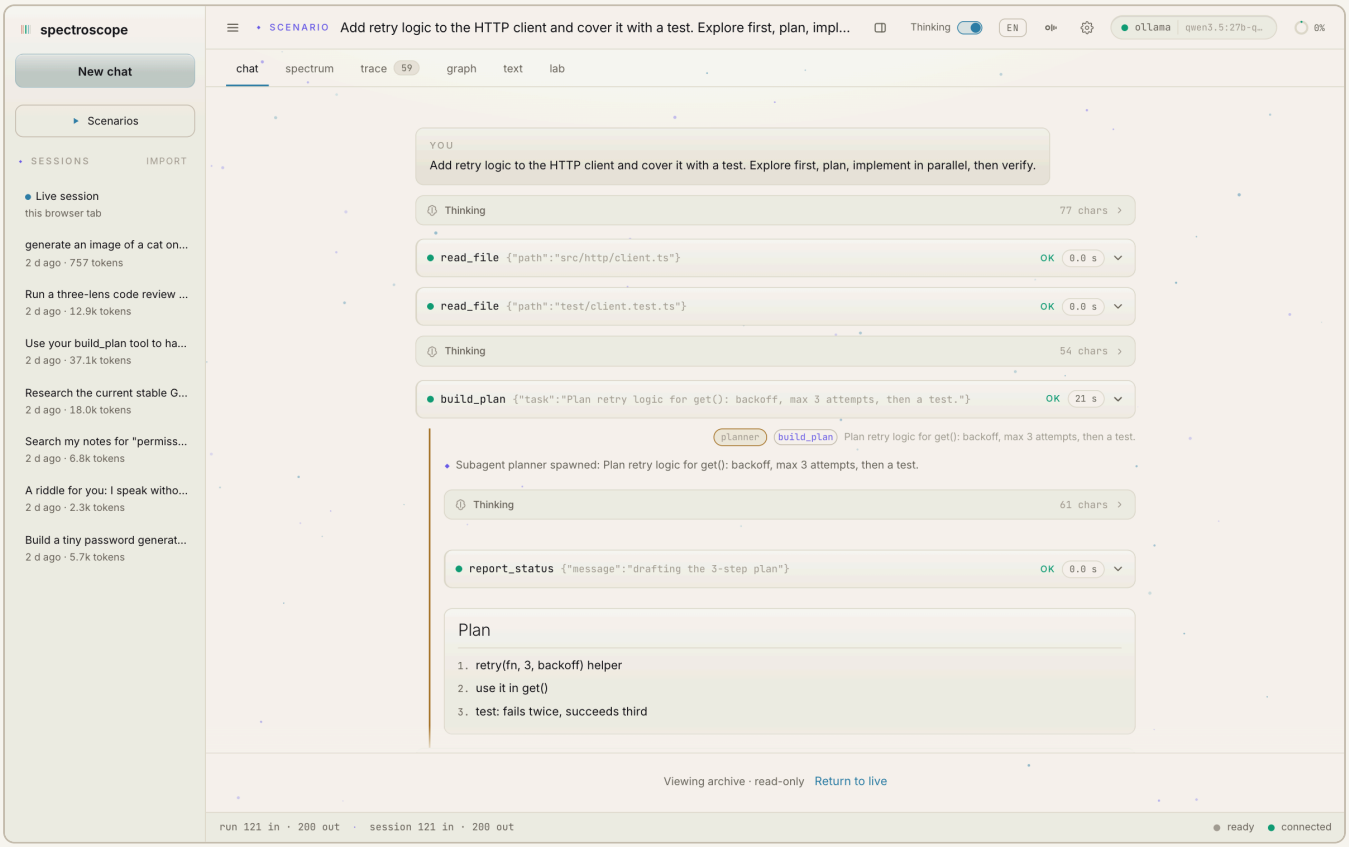
- **Sidebar** (left, resizable, collapsible): “New chat”, the **Scenarios** button (chapter 7), the session archive with an Import button (chapter 8). The first row is always the live session of this browser tab.
- **Header**: session title (the first prompt), image-gallery toggle (appears once images exist), right-panel toggle, the **Thinking** switch, the **DE/EN** toggle, the design drawer, the **provider picker** chip, the **context ring** (appears once tokens flow) and the Stop button while a run is active.
- **Tab bar**: Chat · Graph · Trace · Lab — four lenses over the same state.
- **Footer**: run and session token totals in tabular numerals, a run status dot, and the connection dot.

Behind the scenes, every incoming socket frame is batched once per animation frame (a `text_delta` flood costs one React render per frame, not per token) and folded by one pure reducer into an immutable `UiState`. Live stream and archived replay produce the same state shape from the same code — which is why every view in this chapter works identically for both.



The state pipeline: one socket, one rAF batch, one pure reducer — every tab is a lens over the same UiState.

The conversation

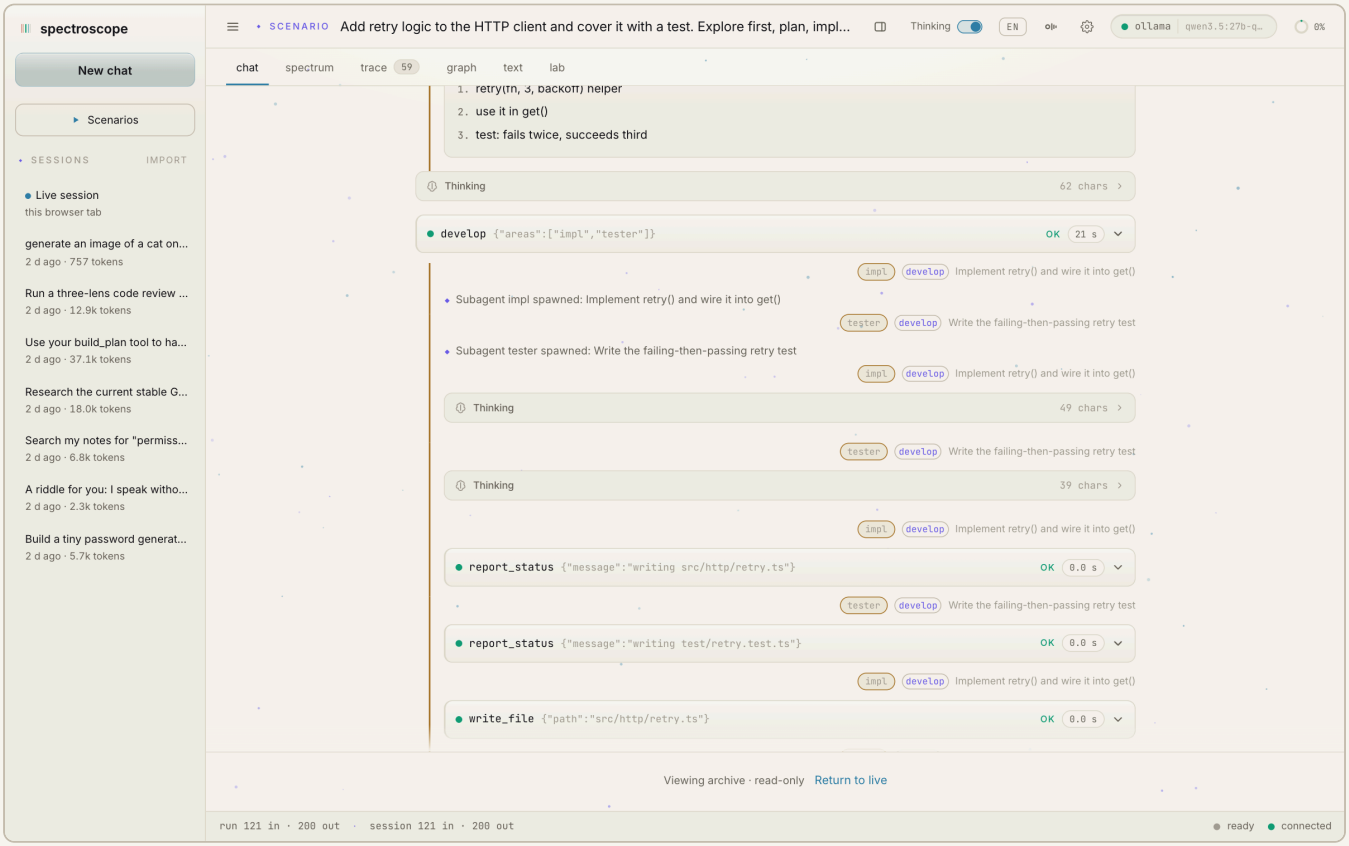


A full conversation (here: the deterministic “coding” scenario): user prompt, collapsible Thinking disclosure, tool cards with gate chips, markdown answer cards.

- **Markdown answers.** Assistant turns render through a dependency-free markdown parser (headings, streaming-safe fenced code with language chip and copy button, nested lists, GFM tables, quotes). Nothing ever becomes raw HTML, and link protocols are vetted — `javascript:` renders as plain text. Model output is untrusted end to end.
- **Thinking, apart from the answer.** While a reasoning model thinks, a pulsing “thinking …” dot is the live indicator; the finished reasoning collapses into a disclosure above the answer showing its size. The header switch turns the stream off for the next run (`set_thinking`).
- **Tool cards.** One collapsed card per call: status dot + name + input preview + **gate chip** + duration. The gate chip is the didactic point — *Gate: allowed / denied / waiting* appears only for permission-gated calls, so the three control paths (free tool, gated tool, hook-blocked call) read differently at a glance.

Expanding shows full input and output JSON with copy buttons.

- **Subagent threads.** Consecutive turns of one child nest under a pastel-edged thread block (header: agent id, role label, task). Interleaving splits threads honestly — the Trace keeps the flat truth. More in chapter 14.
- **Info and error lines.** Spawns and compactions appear as quiet info lines (localized live); a failed run gets an error card with a “Send again” link.



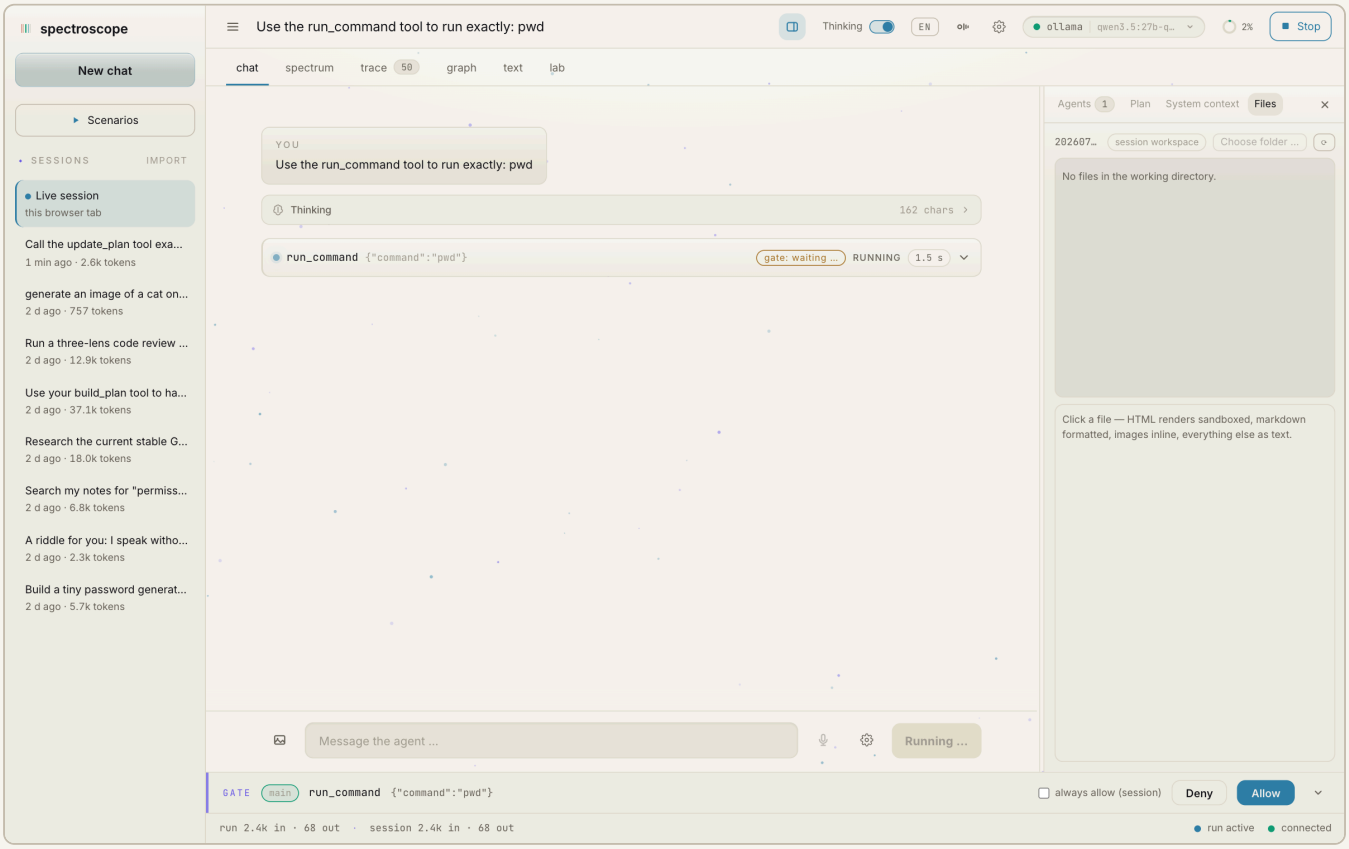
> Subagent threads: two parallel workers from a fan-out, each with its own pastel-edged thread, status lines and tool cards; the A2A result closes each thread.

The composer

- **Enter sends, Shift+Enter breaks the line** — the only shortcut. The textarea autosizes up to 150 px; the Send button reads “Running …” while a run is active (one run at a time per connection).
- **Image attachments.** The image button opens a file picker (jpeg/png/webp/gif, multiple); dragging files anywhere onto the chat works too. Every image is downscaled in the browser before sending (longest edge 1568 px, JPEG re-encode when useful; GIFs pass through to keep animation), previewed as removable chips, and never leaves the page until you hit Send. Attachments cross the socket as base64, land in the session's blob store, and the events carry only references.
- **Push-to-talk.** The mic button records (coral pulsing dot + `m:ss` timer), a second press stops; the audio is transcribed server-side by local whisper.cpp and the text lands *in the input field* — an STT error is reviewable text, never an agent instruction. If STT is not installed the button disables itself with a setup hint. Chapter 17 has the setup.

The permission dialog

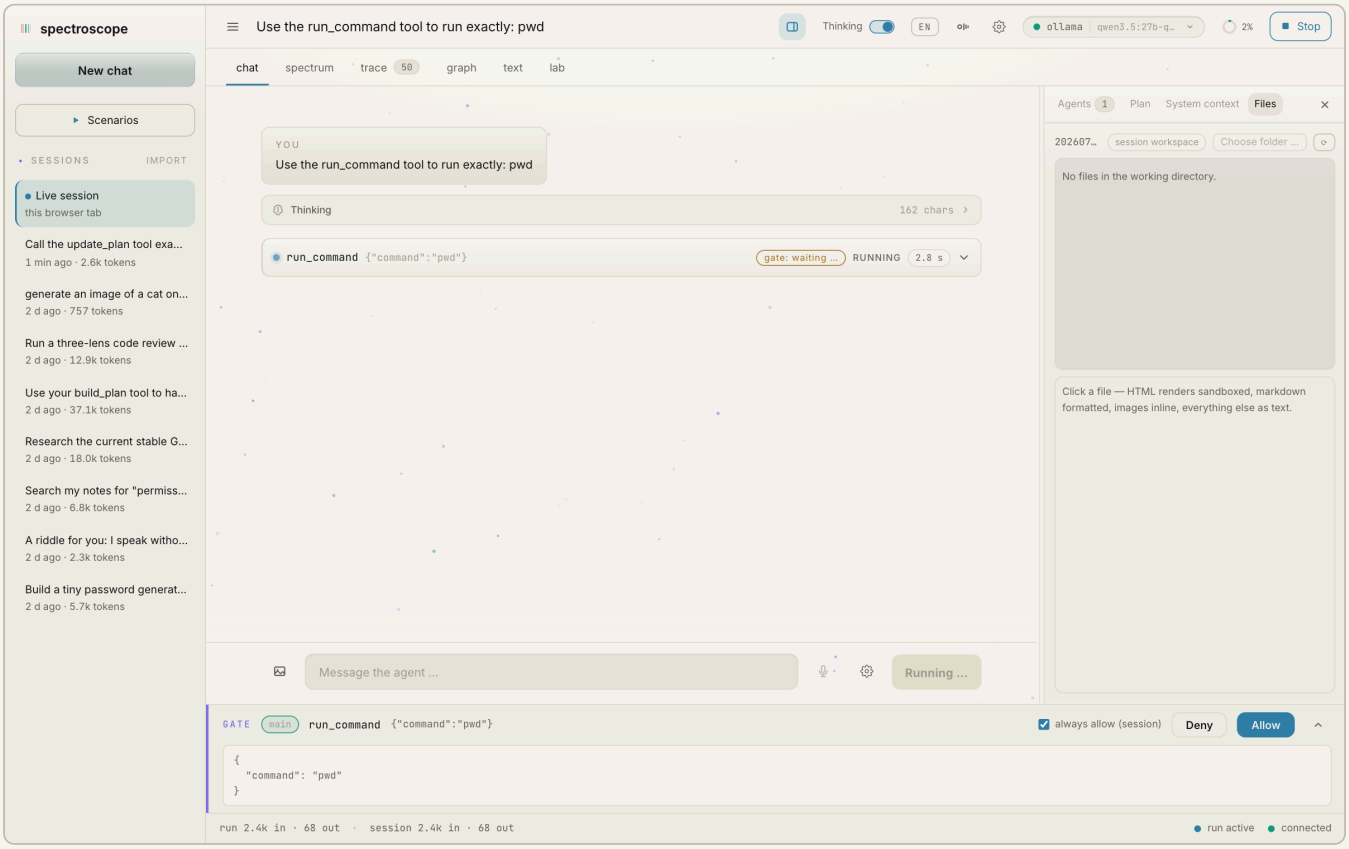
When the agent wants to run a gated tool and no allowlist rule covers it, the run genuinely pauses — the agent's virtual thread parks on a future server-side — and the dialog takes over:



> The serious moment: run_command wants to execute uname -a. The input is shown in full — you approve what you see. Deny is the focused default; Escape denies.

- The input is shown **in full** as pretty JSON — you approve what you see, never a summary.
- **Deny is the safe default**: it holds focus, Escape denies, Tab is trapped, and clicking the backdrop does nothing — a decision must be deliberate.
- When a subagent asks, the dialog names it (“requested by worker-1”).
- Multiple pending requests queue with a counter (“1 of 3”).

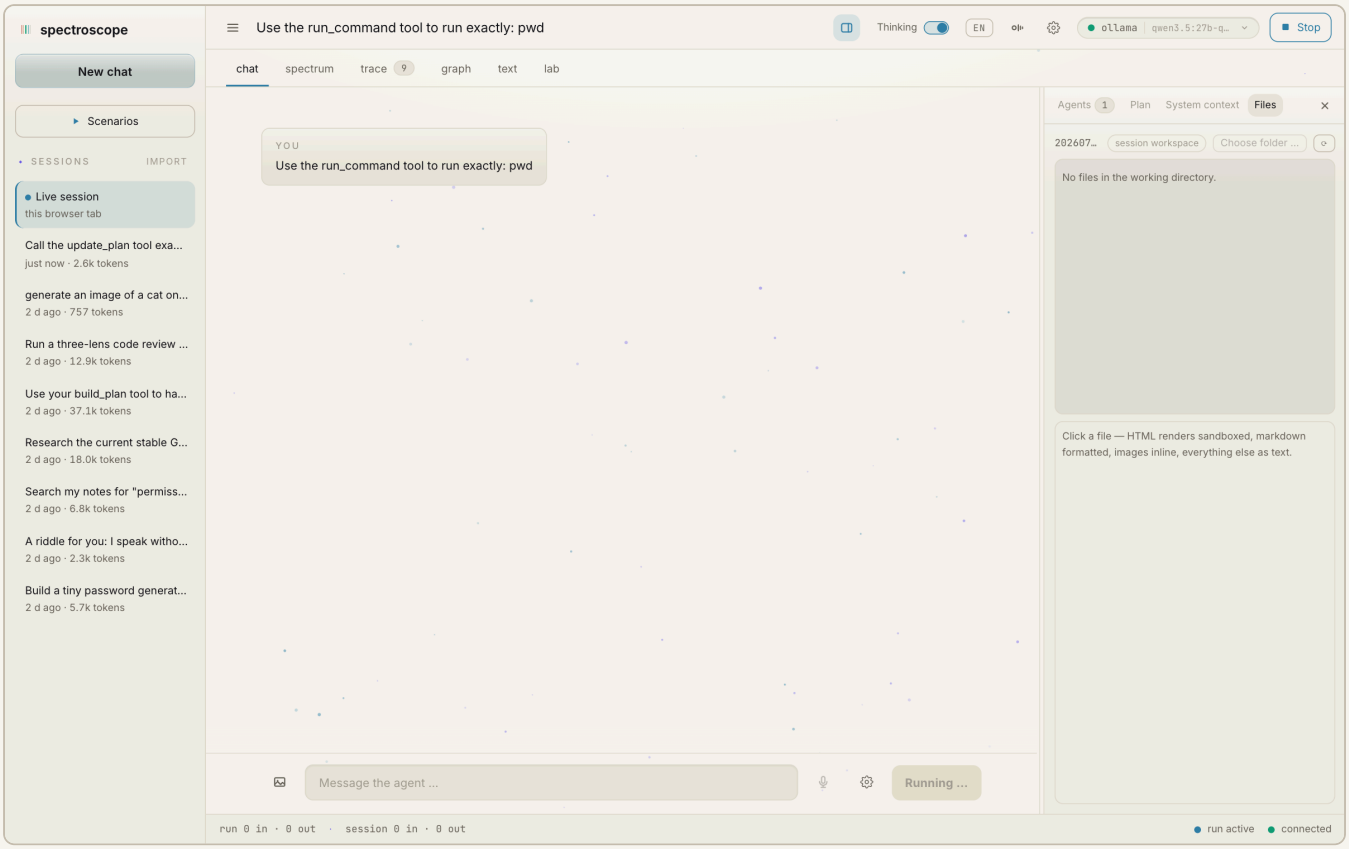
The two checkboxes appear once you tick the first:



> “Always allow: run_command (this session)” — and, gated behind it, “Persist (project settings, .spectro/settings.json)”. Only Allow carries the flags.

Always allow (this session) remembers the approval for this browser tab only. **Persist** appends the rule to the project's `.spectro/settings.json` — the same `autoApprove` list the CLI reads, so a web decision is honored by the terminal on its next run. Remembered rules are **prefix-scoped** for risky tools (a `git status` approval becomes `run_command:git*`, never “all commands”) — chapter 13 has the exact rules. A denied call returns `ERROR: the user denied the execution.` to the model, which reads it and adapts.

Live indicators



> A live run against local Ollama: the model's reasoning streams into the pulsing Thinking disclosure while the header shows the provider chip and the context ring.

- The **context ring** in the header fills with the last reported input tokens against the compaction threshold (green < 70%, amber ≤ 90%, red above); clicking it opens the introspection popover — chapter 28.
- The **usage footer** counts run and session tokens live in tabular numerals; the status dot switches “run active → ready” and names any non-regular stop reason.
- The **connection banner** appears under the header when the socket drops, with a reconnect countdown (exponential backoff 1→15 s) and a “Reconnect now” action. Sessions survive: the JSONL file is the state, the socket is just transport.
- The **Stop button** sends `abort`, which fires the run's CancelSignal — the run ends with `stopReason: "aborted"`, tools and children included.

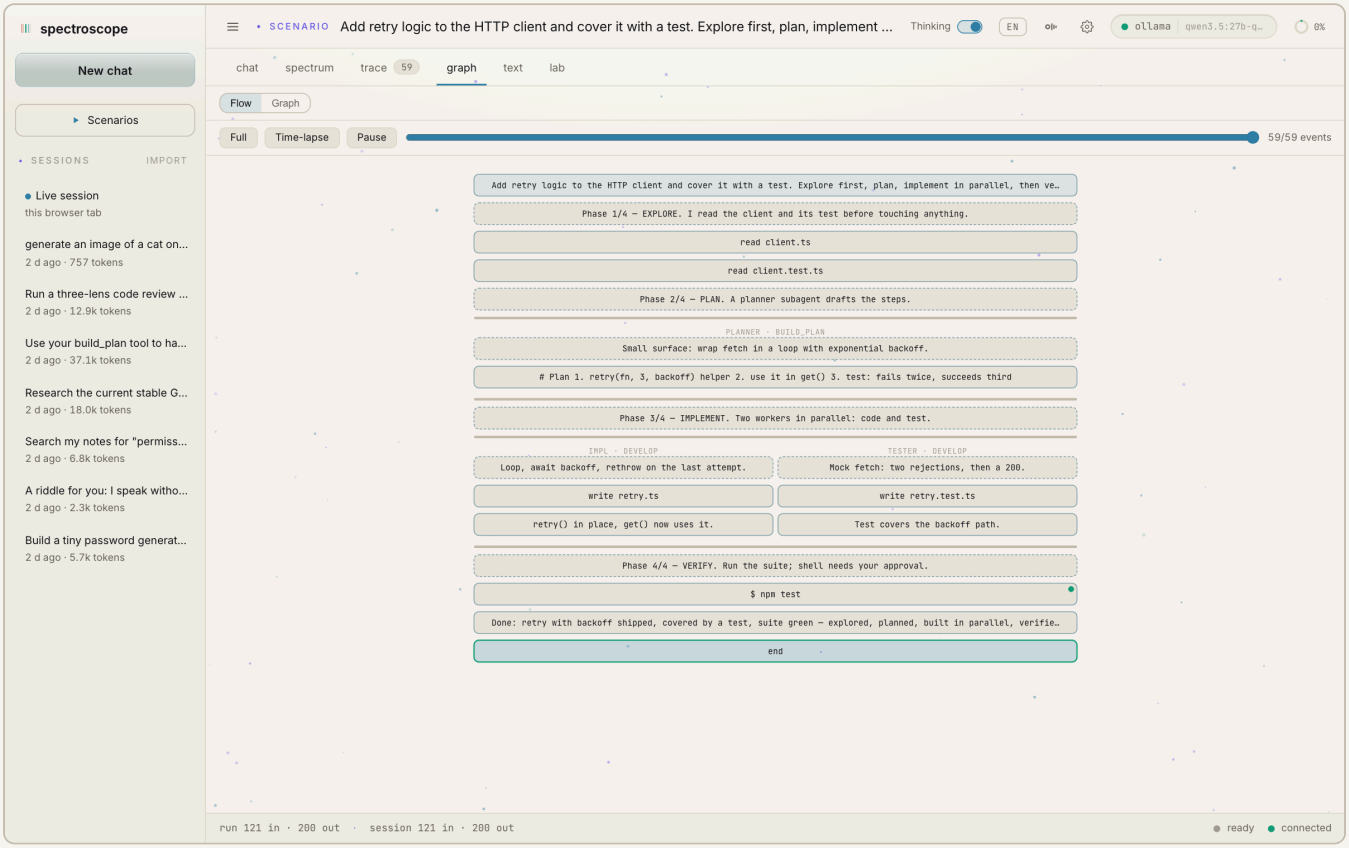
CHAPTER 6

The web face: tabs & panels

Chat is one lens. The Graph tab shows the same run as a process diagram, the Trace tab as wire frames, and the right-docked panel adds the agents roster, the live plan, the exact system context and a sandboxed file browser.

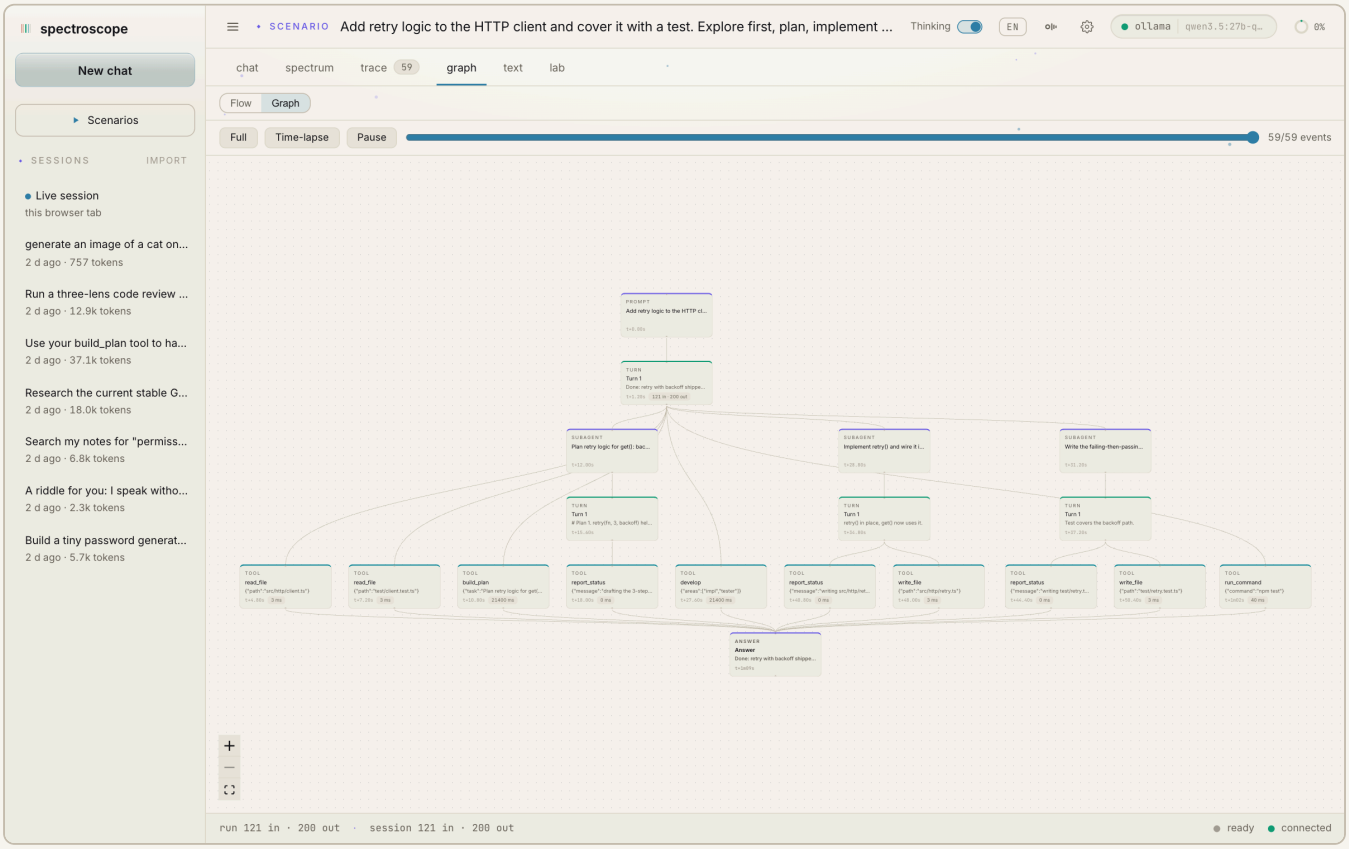
The Graph tab

Two views behind a *Flow* | *Graph* toggle (sticky per browser). For archived sessions a replay bar adds *Full*, *Time-lapse* (advances along the real timestamp gaps, accelerated), *Pause* and a scrubber.



> Flow view: the whole session as a BPMN-style overview — activities on the main spine, one parallel split/join block per subagent wave, gate outcomes as dots.

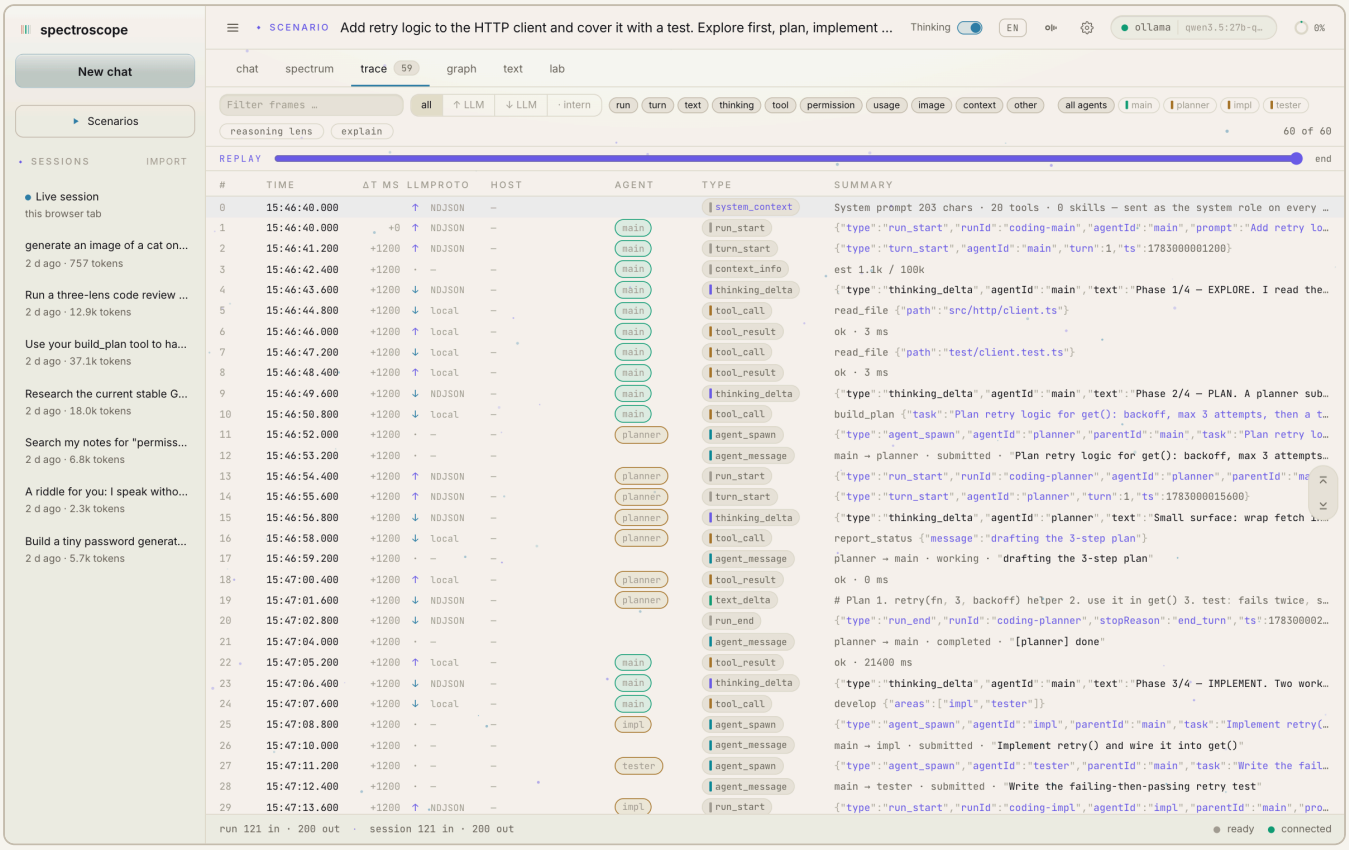
Flow is the minimap for humans: rounded activities (*user / think / tool / say / compact / end*) stack on the main spine; each subagent wave becomes a split block with one column per child and a join bar. Think and say activities accumulate their streamed text, so real runs get sentence labels instead of "token one". Tool activities label themselves honestly — `read <file>`, `$ <command>`, `fetch <url>`, `server · tool` for MCP. Gate outcomes render as small dots (pending / allowed / denied). In a replay, clicking an activity seeks the timeline to its first event.



Graph view: the dagre DAG — prompt, turns, tools, subagent lanes and the answer; every node carries timing, token badges and status.

Graph is the classic DAG (React Flow + dagre): one node per prompt, turn, tool call, subagent and answer, with `t+` relative times, duration and token badges, error badges, and a pulsing outline for running nodes. Sequentially spawned subagent subtrees get their own side-by-side lanes. Clicking a node opens a detail panel with its timing line and its *raw evidence* — exactly the events that formed the node, each as a collapsible JSON tree. The right mouse button pans; the left button only clicks — on both React Flow canvases in the app.

The Trace tab



> The wire view: every frame in both directions with time, Δt, LLM direction, wire protocol, agent, type and a token-highlighted summary. The tab badge counts frames live; the slim rail on the right jumps to the first or the newest frame.

"What Wireshark is to packets, this is to the harness protocol." Every frame lands here — RunEvents inbound, client messages outbound — before the reducer even looks at it, so unknown event types are visible too.

- **Columns:** sequence · wall-clock time with milliseconds · Δt since the previous visible row · **LLM direction** (↑ going to the model with the next request, ↓ produced by the model, · internal) · **proto** · **host** · agent · type · summary.
- **The proto column** names the wire each payload actually rides — the protocol breakdown as a column: `SSE` for the cloud LLM streams (Claude and OpenAI-compatible), `NDJSON` for Ollama, `JSON-RPC` for `mcp__*` tool rows (a result resolves its tool through the callId), `HTTP` for `web_fetch` and image generation, `local` for the file tools, and `–` for everything harness-internal (the gate, the plan, introspection, A2A) that never leaves the process.
- **The host column** names the network counterpart: LLM rows show where the request really goes (`api.anthropic.com`, `localhost:11434` …) — live from the socket-only `provider_info` frame the server sends on connect and after every provider switch, so a switch is a visible trace row, never a silent client-side swap. MCP rows name their server, `web_fetch` the fetched URL's host, image events their backend endpoint; replays only know the provider, so anthropic maps to its fixed endpoint and local backends show an honest `–`.
- **Filters:** free-text (matches type, agent and payload), an LLM-direction segment, and toggleable category chips (run, turn, text, tool, permission, usage, image, context, other).
- **Summaries with token highlighting:** the model's own words in sand, punctuation faint, numbers tabular, the literal `ERROR` red — lossless, the full payload is one click away (collapsible JSON tree + copy).

- **The synthetic `system_context` row:** the system prompt rides with every request but is not a wire event — the view prepends one synthetic ↑ row (from `GET /api/context`) naming prompt size, tools, skills and MCP servers. Display-only: never in the reducer, never in the JSONL.
- **Reading flow:** the tab **opens at the first frame** — the story reads from the beginning. While a frame is open, **Space steps to the next visible one** (it opens, takes focus and centres itself; Enter still toggles a focused row), so the trace taps through like the Lab stepper. A slim **jump rail** on the right edge scrolls to the first or the newest frame.
- **Auto-follow** engages once you scroll down to the live end; while you are reading further up, arrivals accumulate into a "{n} new ↓" pill instead of yanking the view.
- **Three detail modes** per row: *Insight* (the JSON tree), *Compact* (one token-highlighted line per real wire line, horizontally scrollable) and *Raw* (plain text). The `session_resume` marker's detail carries the entire re-uploaded history as its JSONL lines — chapter 8.

The Text tab

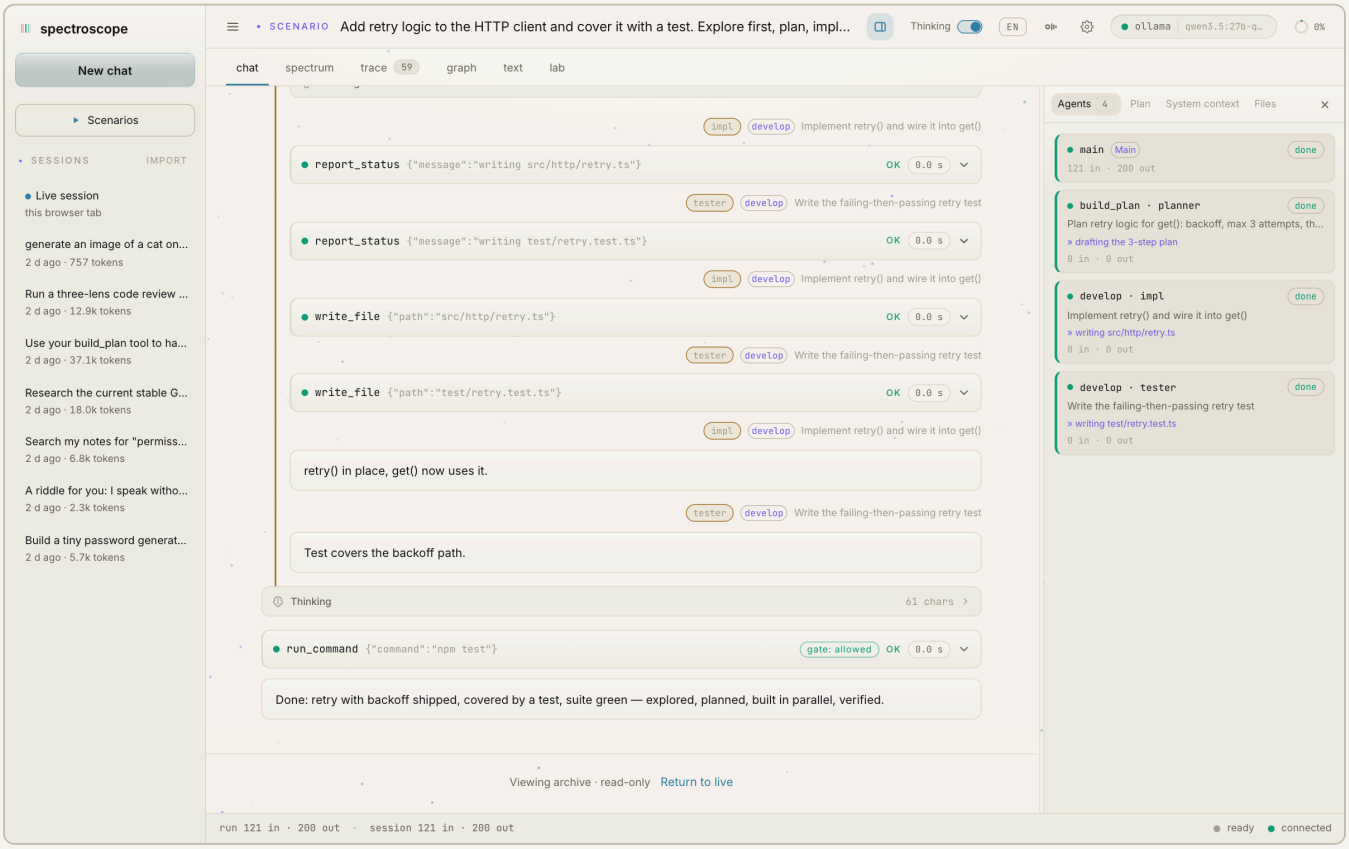
The fourth lens on the same stream — where the trace shows frames, this shows the *text*. Two views behind a *Text* / *JSONL* toggle (sticky per browser), one copy button for the whole thing:

- **Text** is the whole session as one readable feed, with the protocol made visible as literal markers: every reasoning run sits between `<think>` and `</think>` — exactly the tags Ollama really streams inline; Anthropic's thinking blocks render to the same boundaries — followed by the answer text in plain type. Tool use appears as honest indicators with nothing hidden: `[tool_call read_file {"path":"…"}]`, then `[tool_result read_file · 9ms]` with the *full* output underneath. Run boundaries (`[run_start ollama]`, `[run_end end_turn]`), the permission gate, spawns, A2A messages, compaction and errors all keep their place in reading order; subagent lines carry their agent id as a prefix. Prompts render in sand, thinking dimmed italic, markers faint — but it is all ONE selectable text, and the copy button hands it out as plain text.
- **JSONL** is the session exactly as the file on disk stores it: one compact JSON line per wire event, newline per event — the NDJSON view of the truth. Socket-only UI frames (`provider_info`, `workspace_info`) are deliberately absent: they never enter the file, and this view IS the file.

The right panel

The header's panel button docks a resizable panel into the Chat tab with four tabs.

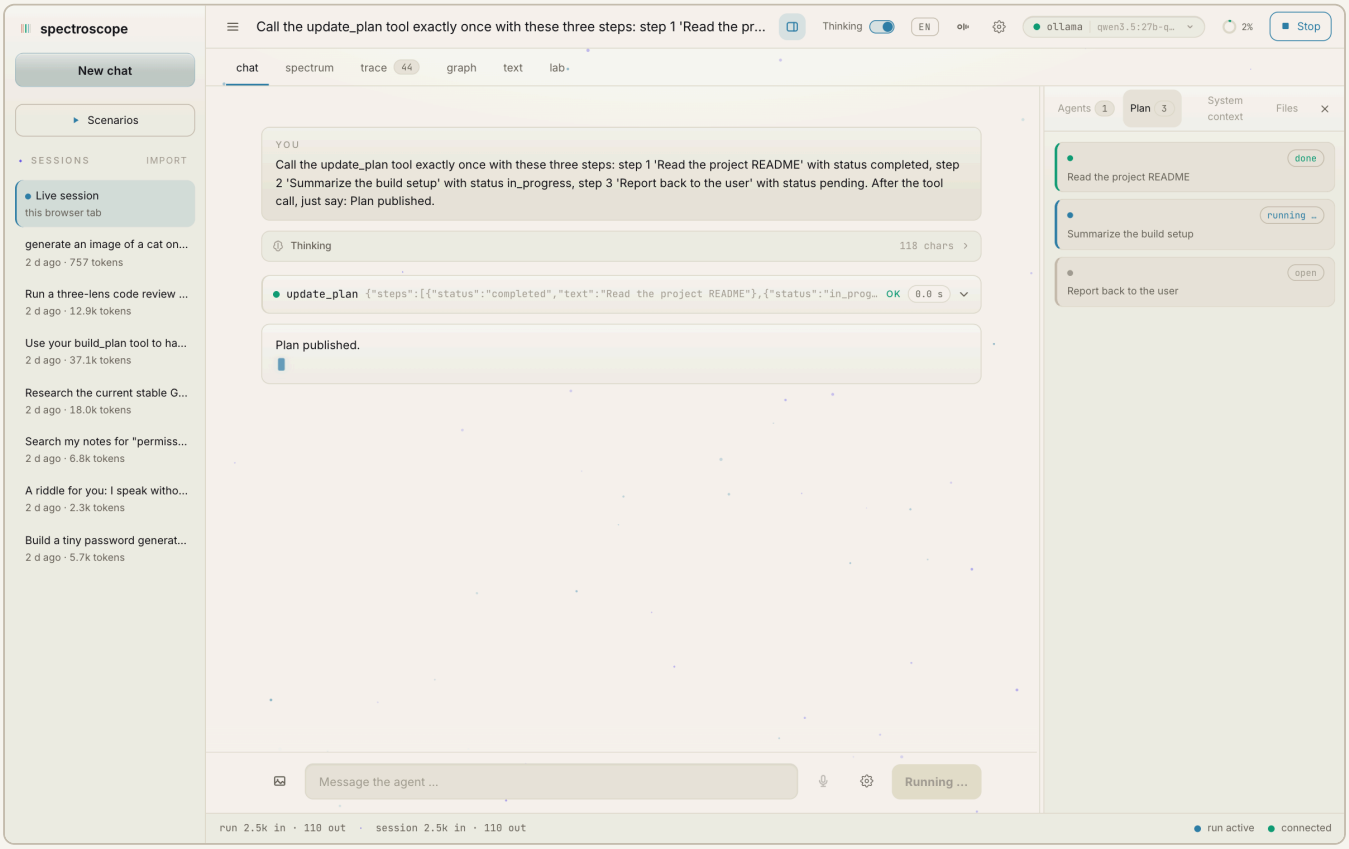
Agents



> The session-wide roster: main plus every subagent that ever ran, with lifecycle badge, task, latest status line and token cost. It survives run ends — only a new chat clears it.

One card per agent — main first, then every child in spawn order — with its state dot and lifecycle badge (*submitted / working / done / failed*), its task, the latest `report_status` line, and its token cost (`1.2k in · 0.3k out`). Selecting a card jumps to the System context tab for that agent.

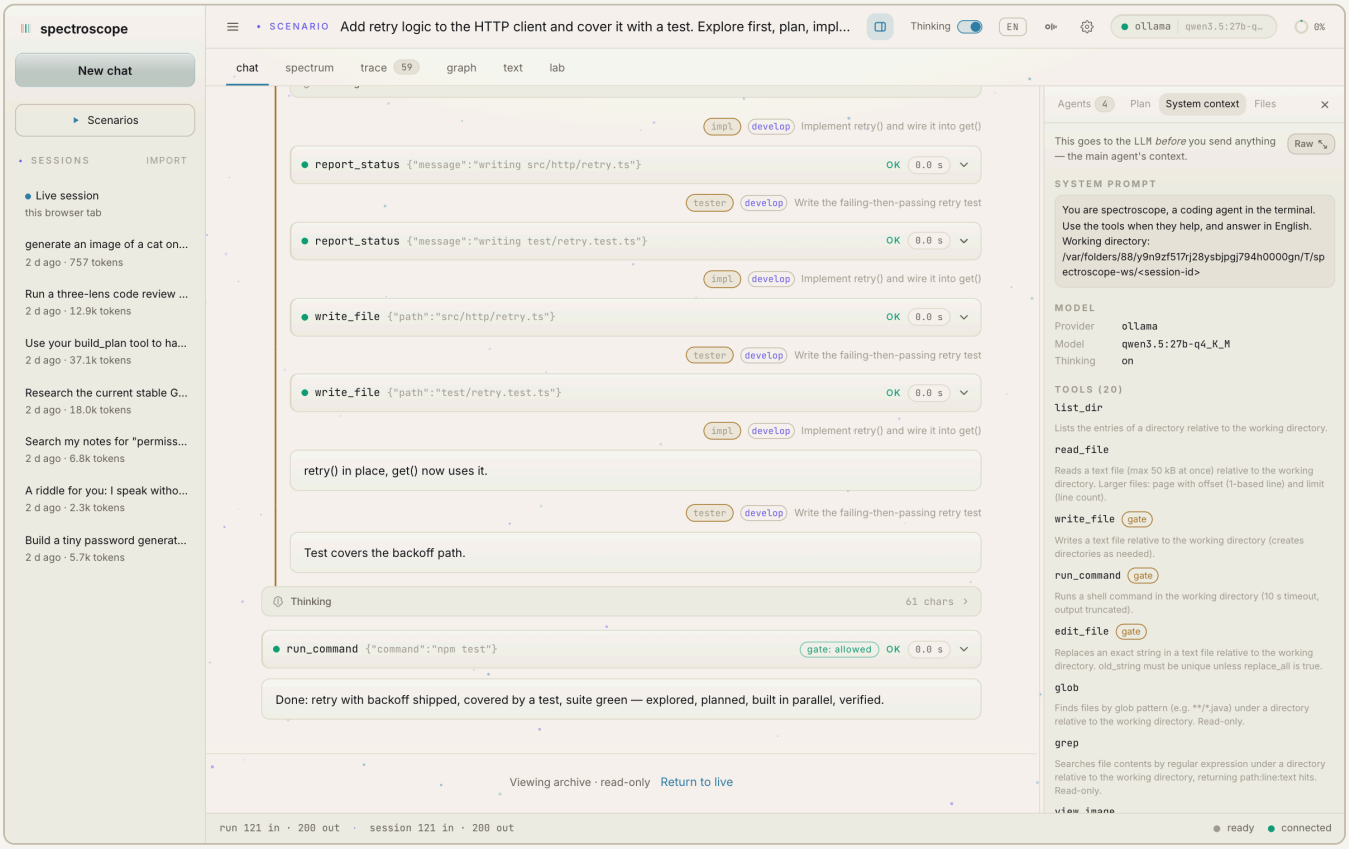
Plan



> The agent's published plan, live: three steps in the three states (done / running… / open). This capture is from a real local-model run — note the first update_plan attempt errored on a wrong field name and the model self-corrected on the second call.

The read-only, latest-wins snapshot of the agent's plan, fed by the permission-free `update_plan` tool: each step with a status dot and badge — *done*, coral *running …*, *open*. The wire statuses stay canonical English (`pending / in_progress / completed`, enforced at the write boundary); only the labels localize. The tab badge counts steps; the CLI pretty-prints the same event.

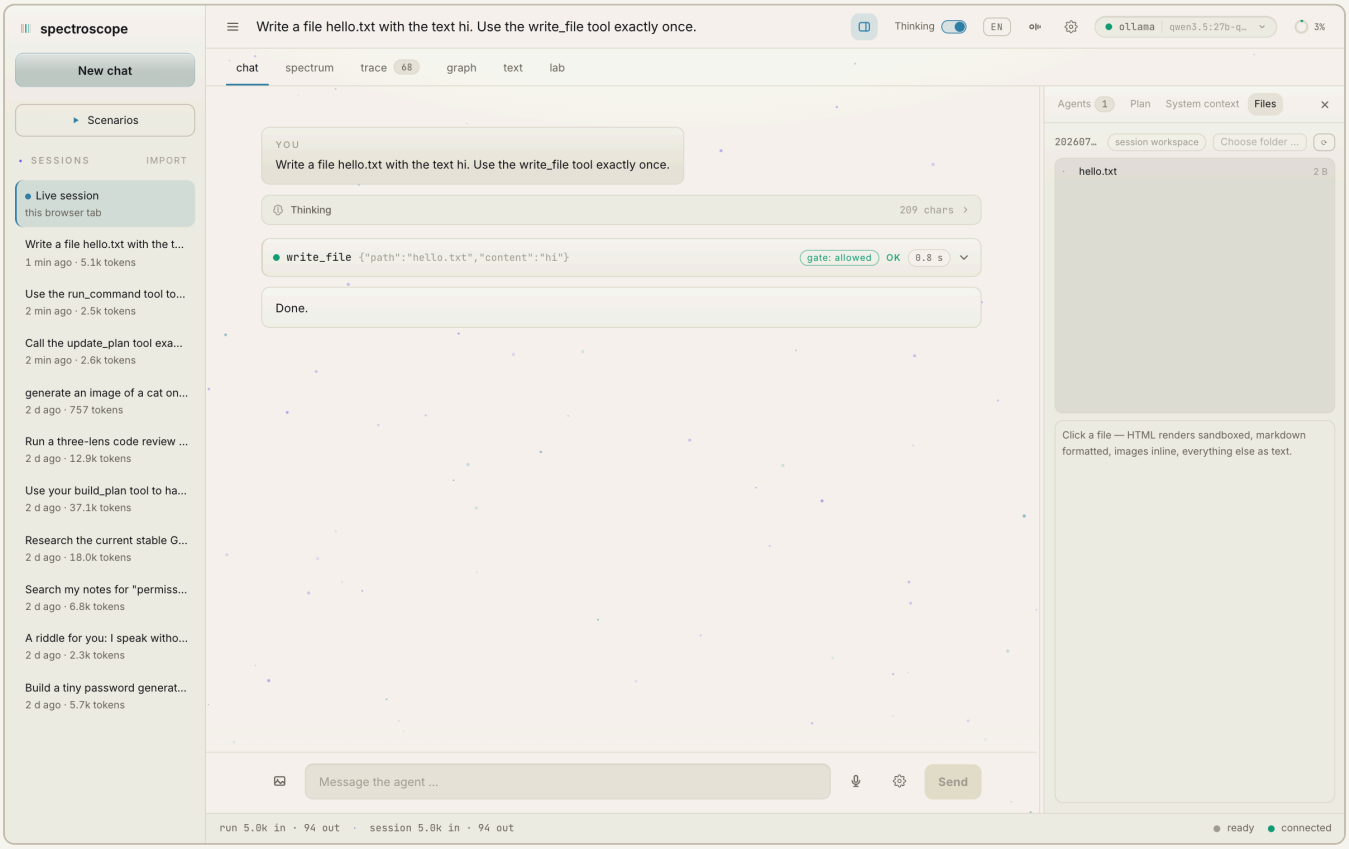
System context



> "What goes to the LLM before any user message": the real system prompt (markdown-rendered), model + thinking state, all tools with gate markers, skills, MCP servers and the subagent role profiles.

The transparency panel: exactly what the LLM receives before any user message. For the **main agent**: the full system prompt (working directory, project `SPECTRO.md` context, the skills catalog), provider/model/thinking, every tool in registration order (gate chip = needs your approval), skills, MCP servers, and the six subagent role profiles. For a **selected subagent**: its role prompt, its task, its reduced tool set — and the note that children never get the spawn tools (nesting ends at depth 1). The data comes from the stateless `GET /api/context`, assembled from the same constants the live tools use — what the panel shows is what the model gets. A fullscreen *Raw* modal shows the whole context untruncated with a character count and copy button.

Files



> The Files tab is the agent's desk: the workspace this session runs in, refreshed live — hello.txt, which the agent wrote through an allowed gate seconds ago, is right there.

A read-only view of the agent's **workspace** — the same sandbox the file tools see. Every session gets its own folder (`<tmpdir>/spectroscope-ws/<session-id>` unless a `workspace` is configured — chapter 30), the server announces it over the socket (`workspace_info`, visible in the trace), and the tab follows it live: what the agent writes appears here, and the repo you started spectroscope in stays clean. The panel **opens itself** on the Files tab when the announcement arrives — the agent's desk appears where its files land. And you can **pick the desk per session**: the “Choose folder ...” button opens the *native* folder dialog on the spectroscope machine (`POST /api/pick-workspace`, macOS; a browser cannot hand out absolute paths, but spectroscope runs locally), and the picked path travels back as the additive `set_workspace` socket message. Only before the first run — once the agent ran, the sandbox and every subagent are anchored, the button locks and a new chat is the way to a different folder. The pin is shared server state, so the tree you browse and the sandbox the tools see are always the same folder, and a resume in the same server process finds the picked folder again. And the browser **remembers your pick**: every new chat lands on the same desk with zero clicks (a dashed “↺ folder” chip shows the memory; clicking it forgets — the next chat gets its per-session temp folder again. Resumes are never touched). Hidden entries and build folders (`.git`, `build`, `node_modules`, `target`, `dist`, `out`) are skipped, depth and entry count are capped honestly (“list truncated” note). Clicking a file previews it by type: HTML runs in a **CSP-sandboxed iframe** (scripts execute in an opaque origin — no cookies, no API, no parent access), markdown renders through the shared component, images inline, everything else as text. Binary answers 415, oversized 413 — and the defense in depth is real: the `.env` with your API key answers 404 *even by direct URL*, because every path segment is re-checked server-side against the hidden/ignored rule.

The image gallery

When the agent generates images (chapter 17), a gallery column opens next to the chat: cards newest-first with the prompt as caption, provider and model badges, full-size link, and a provider dropdown (`gemini` / `openai`) that switches the image backend for the next generation. Bytes never cross the socket — cards load from `GET /api/images/<sha256>.<ext>`, where the content address doubles as the traversal guard (64 hex chars or 400, before any filesystem access).

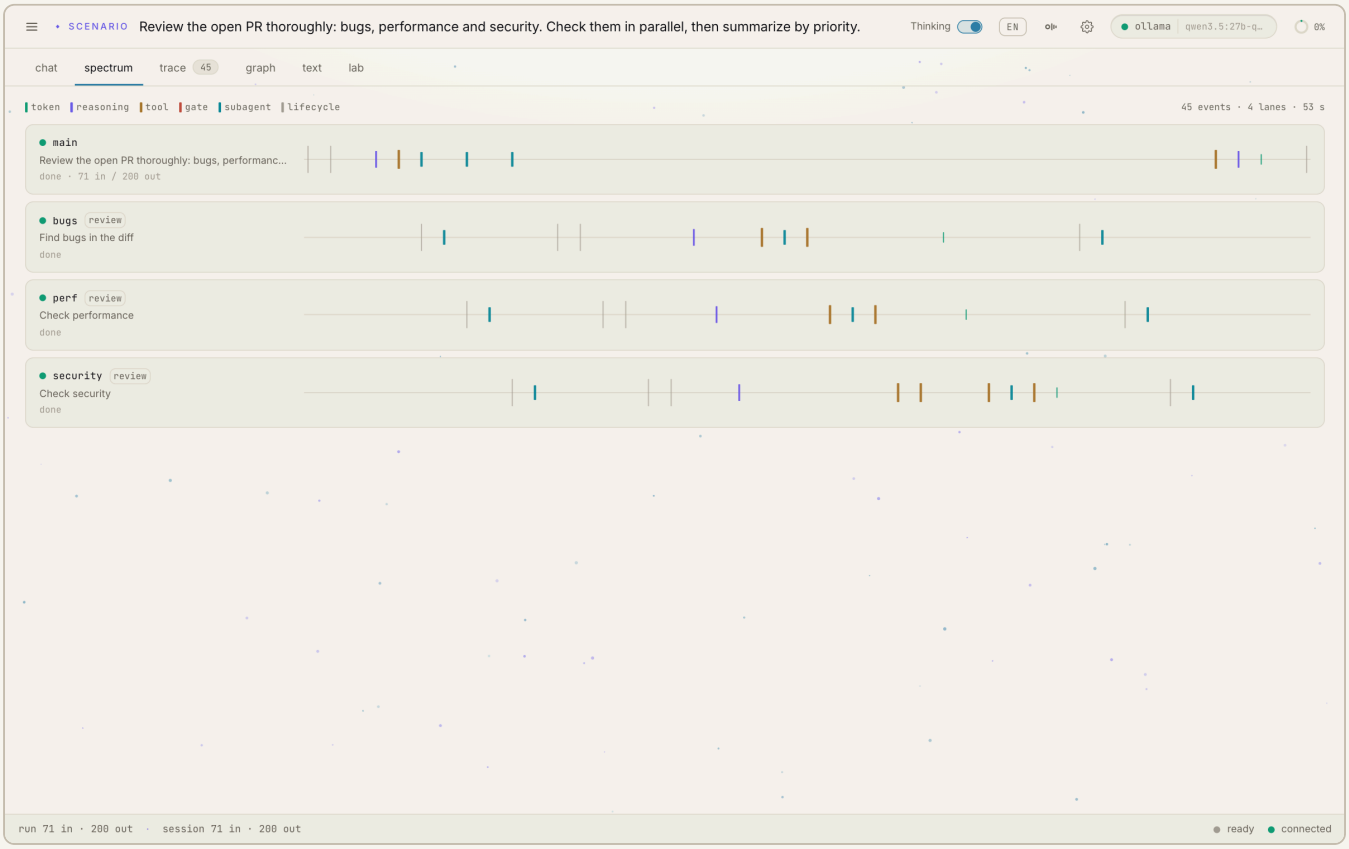
Each card also carries **"→ Workspace"**: it copies the image from the global store into *this session's workspace* — you pick the file name (keep the content-addressed original or type your own; a name without an extension inherits the original's), and the file appears in the Files tab, ready for the agent's tools. The endpoint keeps the house guards: only store names touch the disk, the target name is a single sanitized file name, and an existing file is never overwritten (409).

the spectrum and the new lenses

The design transplant (migration phase 4) brought the brand into the product and four new surfaces with it. All of them read the same event stream you already know; none of them adds a second source of truth.

the spectrum tab

The spectrum is the fleet on one screen. Every agent draws one horizontal lane; every event is a discrete mark on it, colored by type: token teal, reasoning violet, tool amber, gate red, subagent ocean, lifecycle faint. The lane rail carries the agent id, its task, its lifecycle state and its token totals. A pending permission turns the lane's gate mark violet and pulses until the decision lands.

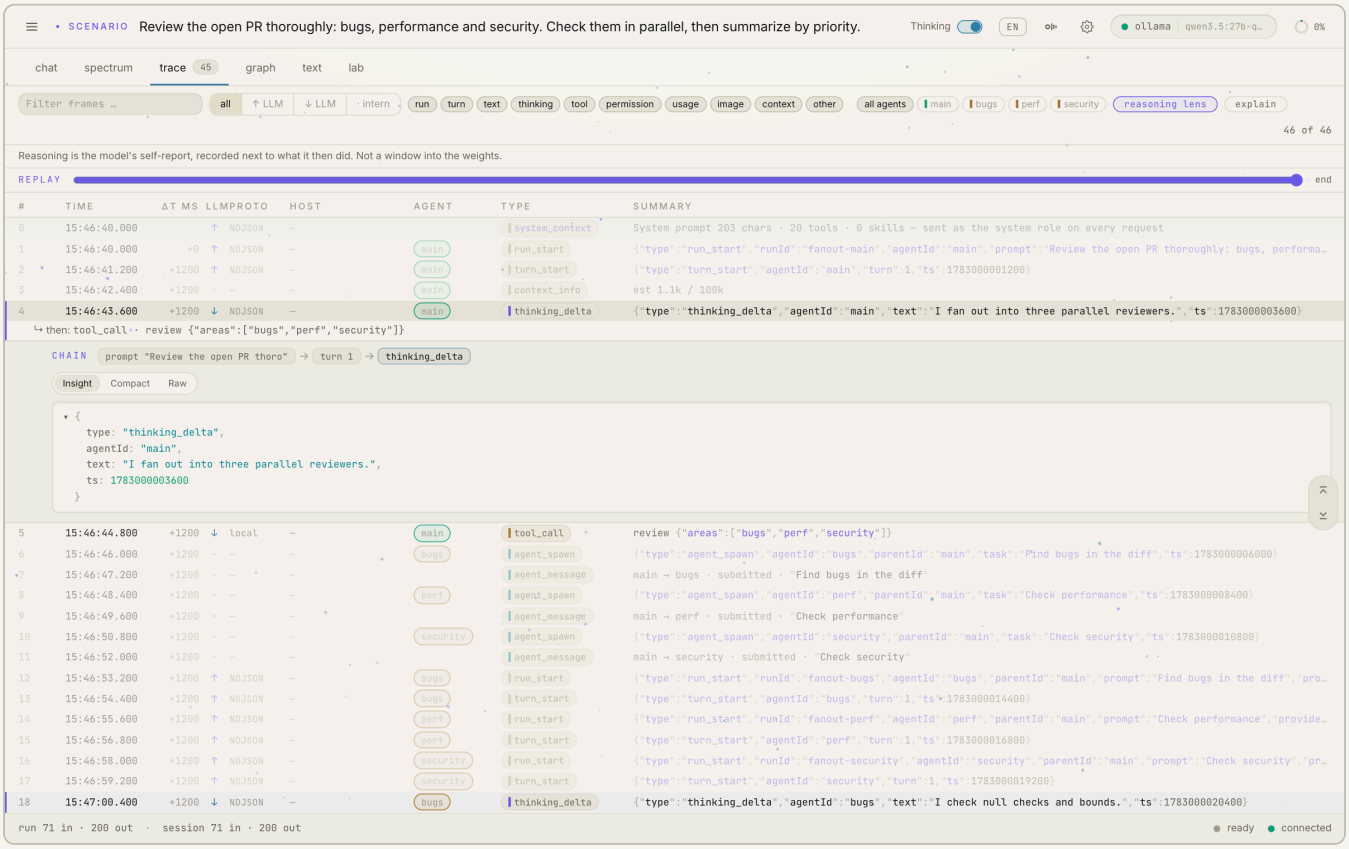


The spectrum tab: one lane per agent, every event a spectral mark. A real capture of a review fan-out with three subagents.

Clicking a lane opens the trace pinned to that agent: the new agent chip row filters the wire view to one lane while session-level frames (decisions, run ends) stay visible. Dense token streams are thinned for display; the counter names the drop, and the JSONL file keeps everything.

the reasoning lens

The reasoning lens is a trace mode for one question: what did the model say it was thinking, and what did it then do? Switched on, thinking frames come to the foreground in violet, everything else steps back, and tool calls and gate frames stay readable as anchors. Every block of thinking ends in a clickable "then:" line that jumps to the action that followed it.



The reasoning lens: thinking foregrounded in violet, the said-vs-did pairing line under the block, the causal chain in the open frame.

HONEST LABELING

Reasoning text is the model's self-report, recorded next to what it actually did. It is not a window into the weights, and the interface says so. With reasoning capture off, the lens tells you that too instead of showing an empty highlight.

The lens state persists with your design preferences, applies to live runs and replays alike, and combines with every other filter.

the causal chain and the replay scrubber

Every open trace frame now shows its walk-back: result, decision, request, call, turn, run, spawn, parent run. Each link is a chip; click it and the trace jumps there. The chain is computed from the recorded stream alone, with no model involved.

The replay scrubber caps the visible stream at any frame. Drag it back and you read the session exactly as far as it had happened, filters included.

the gate surface

Pending permissions are first-class now. Instead of a blocking modal, a gate bar docks above the footer on every tab: the violet line means the run waits on you. Approve or deny inline, keep the session rule checkbox at hand, and expand the bar for the full input and the recorded outcomes of earlier gates. The Lab keeps its own stepper-owned dialog; replays never ask.

the explain panel

The explain button in the trace toolbar docks the why layer next to the stream: a run summary (duration, turns, tool calls by name, gate tallies, tokens, stop reason) and one entry per gate that answers "why did the gate ask" from the permission mode at ask time. Stored sessions carry no mode frames; the panel says "not recorded" instead of guessing. Everything in the panel folds deterministically from the stream.

spectro doctor, the web face

The reference-lamp button in the header opens the calibration page: one measured line per subsystem, from this browser's viewpoint. Server api, live socket, llm backend, session store, default workspace, log level, permission mode. The page re-measures on every open; the CLI's `spectro doctor` checks the machine side.

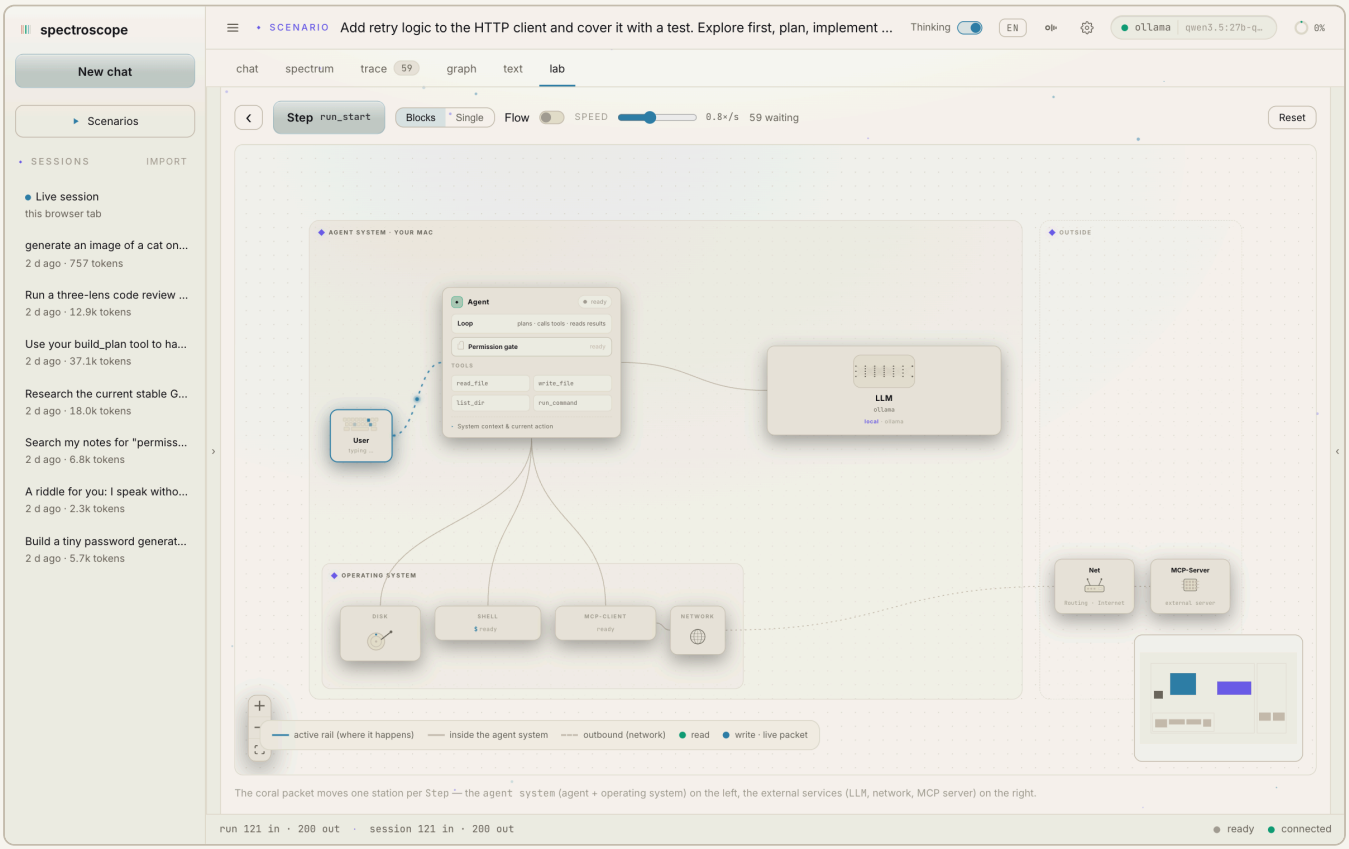
CHAPTER 7

The Lab

The Lab is spectroscope's step-through debugger for agent runs — and its best classroom. Events do not flow through automatically: they queue behind a client-side **dam**, and you advance them click by click while chat, map and JSONL strip move in perfect lockstep. It works on live runs and on any archived replay.

The dam

Live events keep queueing even while the Lab is closed. Each **Step** applies the next event(s) through the *same pure reducer* the chat uses, plus two pure folds underneath: the scene model the Flow map paints, and a strict Petri marking. The invariant is literal: the stepped UI state always equals "fold of everything applied so far". Because the server genuinely waits on permission futures, a token sitting at the gate is the real system state — not a simulation.



A freshly loaded scenario: the dam holds all events ("61 waiting"), the Step button names the next event type, and the Flow map shows the idle system.

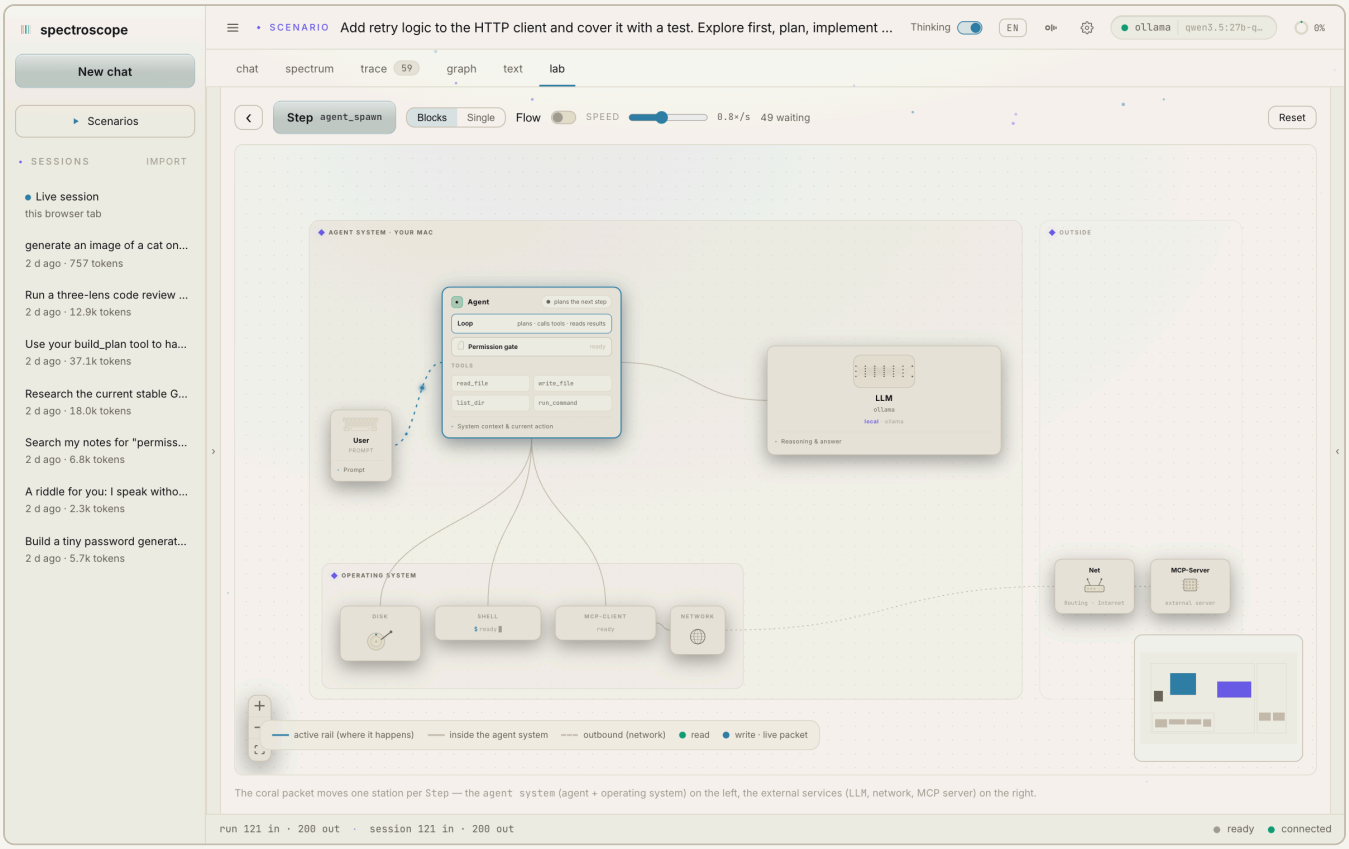
The controls

CONTROL	BEHAVIOUR
Step	applies the next block; the button shows the upcoming event type inline (`Step tool_call`)

‹ Step back	undoes the last step group — chat, map and strip re-fold backwards
Blocks | Single	the grain: one click = one meaningful block (a whole thinking run, a whole answer run, or one event) — or exactly one JSONL line
Flow switch	opens the dam into a paced auto-play — a timer steps at the chosen tempo instead of teleporting to the end
Tempo	0.5×–16×/s slider (default 0.8 steps/s)
Reset	re-queues everything applied

The panes left and right of the map — the fully functional chat and the JSONL strip — are drag-resizable and collapsed by default, so the map gets the room. The strip shows applied lines, the just-fired line highlighted, the dam divider (“▮▮ dam · n waiting”), then the dimmed queue; every row expands to the full event.

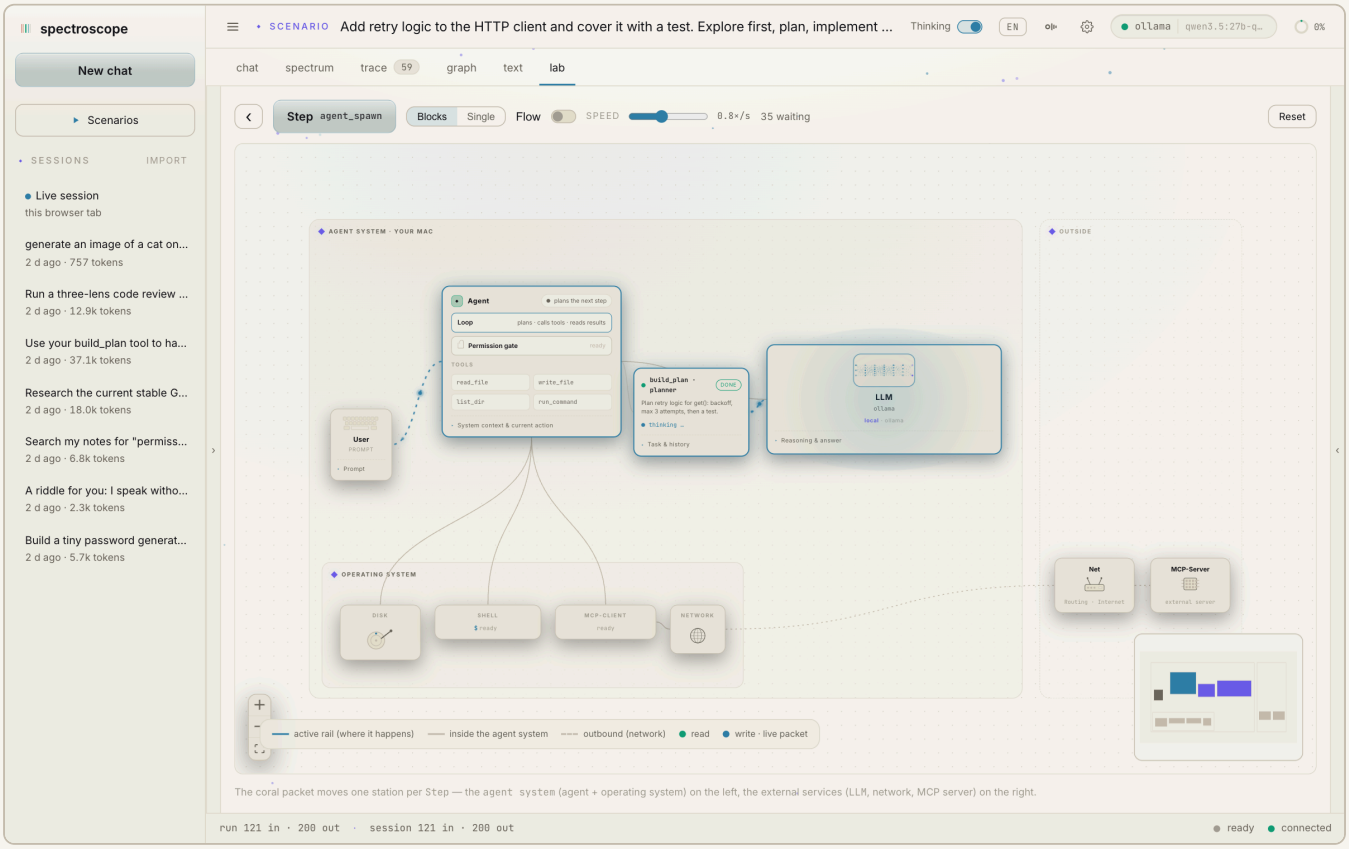
The Flow map (React Flow)



> Mid-run: the coral packet sits on the Agent hub (“plans the next step”), the tool chips and the OS band below it, the LLM zone to the right — with live disclosures on every node.

The default view is a literal systems map. On the left, the **AGENT SYSTEM · YOUR MAC** zone: the user, the **Agent hub** (live activity line, the Loop row, the **Permission gate** row that colors with the gate state, the tool chips) and the **OPERATING SYSTEM** band — disk (animated platter, the touched file as a pill), shell (the running command), MCP client and network stack. On the right, **OUTSIDE**: the LLM (neural-net animation while thinking, model name, reasoning/answer stream slices per agent), the network and the MCP server.

One coral packet rides the rails to wherever the current event happens. The map is **honest about locality**: with a local provider (Ollama) the LLM node sits *inside* your machine and nothing crosses the boundary; switching to a cloud provider re-places it beyond a dashed **network boundary** — live, because the layout follows the selected provider.



> Parallel subagents: each child gets its own loop card inside the machine zone with its own coral packet — two dots flying to the shared LLM = real parallelism, visible.

Every node carries disclosures with the real data: the agent's system context and current tool call, the LLM's per-agent reasoning and answer streams, the disk's file, the shell's command, the MCP client's call. Subagents pop in as their own loop cards with lifecycle badges and disappear when the root run ends.

Under the hood: the Petri marking

The stepper also folds every event into a strict Petri net, for the formally minded: places are the stations a request passes through (*User, LLM, Tool, Gate*), transitions are the RunEvents, and one token moves. A **Log** place accumulates a token per fired transition, so its marking literally counts the JSONL lines — that is the invariant the JSONL strip's highlight rides on. Guards keep the net consistent: a move only fires when the source place holds the token, otherwise it degrades to a pulse (parallel tool calls, odd foreign replays). The marking has no view of its own anymore; it is the Lab's bookkeeping, not its face.

Scenarios: deterministic demo runs

> The scenario picker: seven scripted demo runs, compiled deterministically — no LLM, no API key. Each opens in the Lab with the dam closed.

The sidebar's **Scenarios** button opens seven built-in demo runs. Each is authored once in a bilingual DSL and *compiled* to a deterministic RunEvent stream — real wire format, fixed timestamps, no model involved. Picking one lands in the Lab with the stepper at event zero; it rides the same replay path as any stored session, so every tab renders it.

SCENARIO	WHAT IT DEMONSTRATES
`build_plan · 1 subagent`	dev-tool delegation: a planner child with status updates, verification by main, a gated test run, a *denied* MCP call and graceful continuation
`Review fan-out · 3 subagents`	parallel children (bugs / performance / security) running round-robin, then a prioritized consolidation
`Permission gate · blocked & allowed`	the red path: `rm -rf data/tmp` denied, the agent backs off to read-only listing, then an allowed `git status`
`Disk & shell · read / write`	the disk/shell stations: read → write → list → a gated `npm test`
`Coding · 4 phases, parallel workers`	a realistic dev run: explore → plan (subagent) → implement (two parallel workers writing files) → verify (gated test run)
`Research · consolidate + critical review`	parallel source sweep with MCP searches, a consolidation draft, an adversarial critic child that finds a contradiction, the corrected answer
`Context window · fill & compact`	no subagents — three big reads fill the context ring (ok → warn → high), then a compaction drops the meter and the run continues

WHY SCENARIOS EXIST

Live model behaviour is non-deterministic — a workshop demo that depends on the model calling exactly the right tool at the right moment will betray you. Scenarios make every mechanism of the harness — gates, subagents, MCP, compaction — steppable and repeatable, offline, in either language. (Most screenshots in Part II use the “coding” scenario for exactly this reason.)

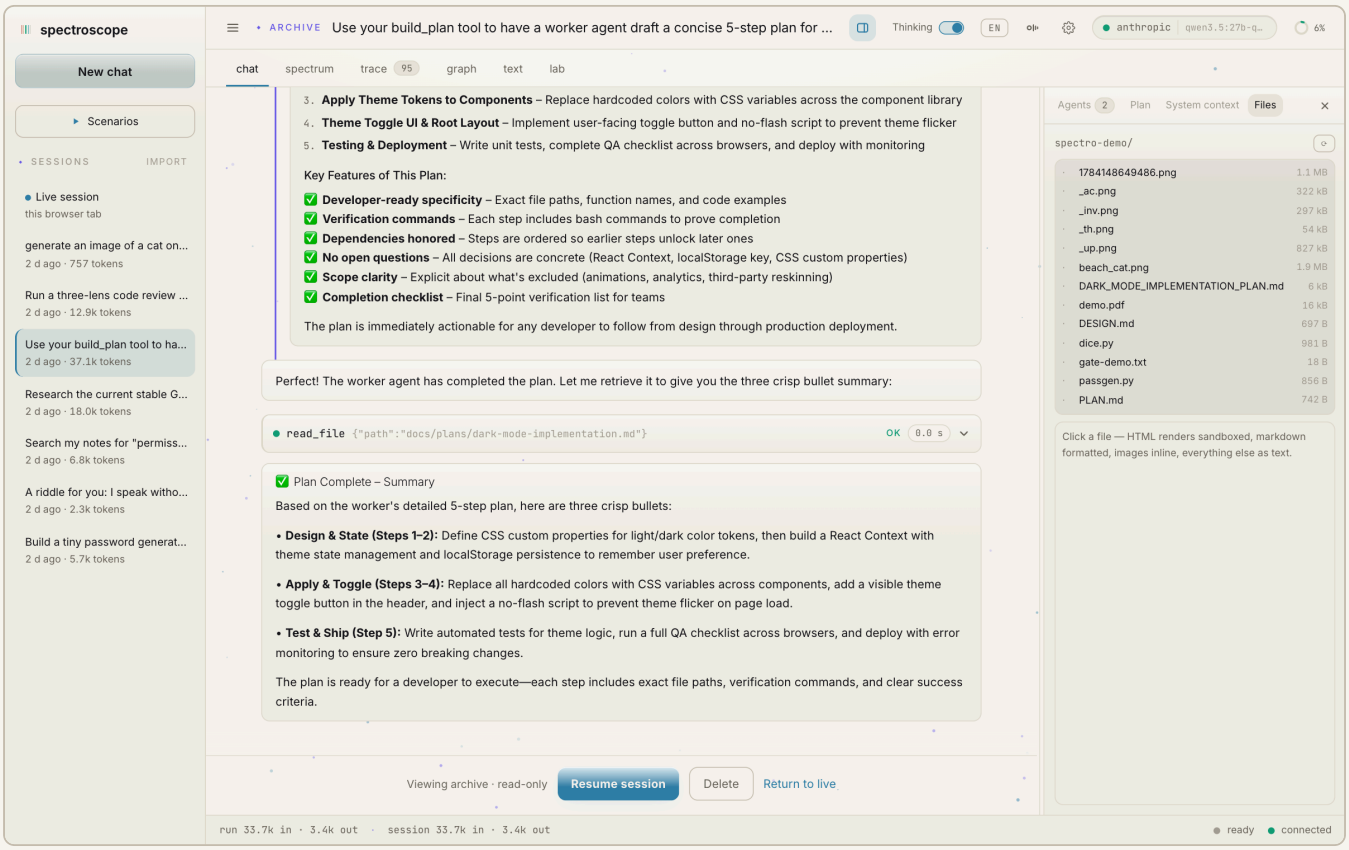
CHAPTER 8

Sessions: replay, resume, delete, import

Every run writes a JSONL session file as it happens — there is no "save" step, the file *is* the state. This chapter is about what you can do with those files from the UI; Part IV shows them on disk, and chapter 22 documents every line.

The archive

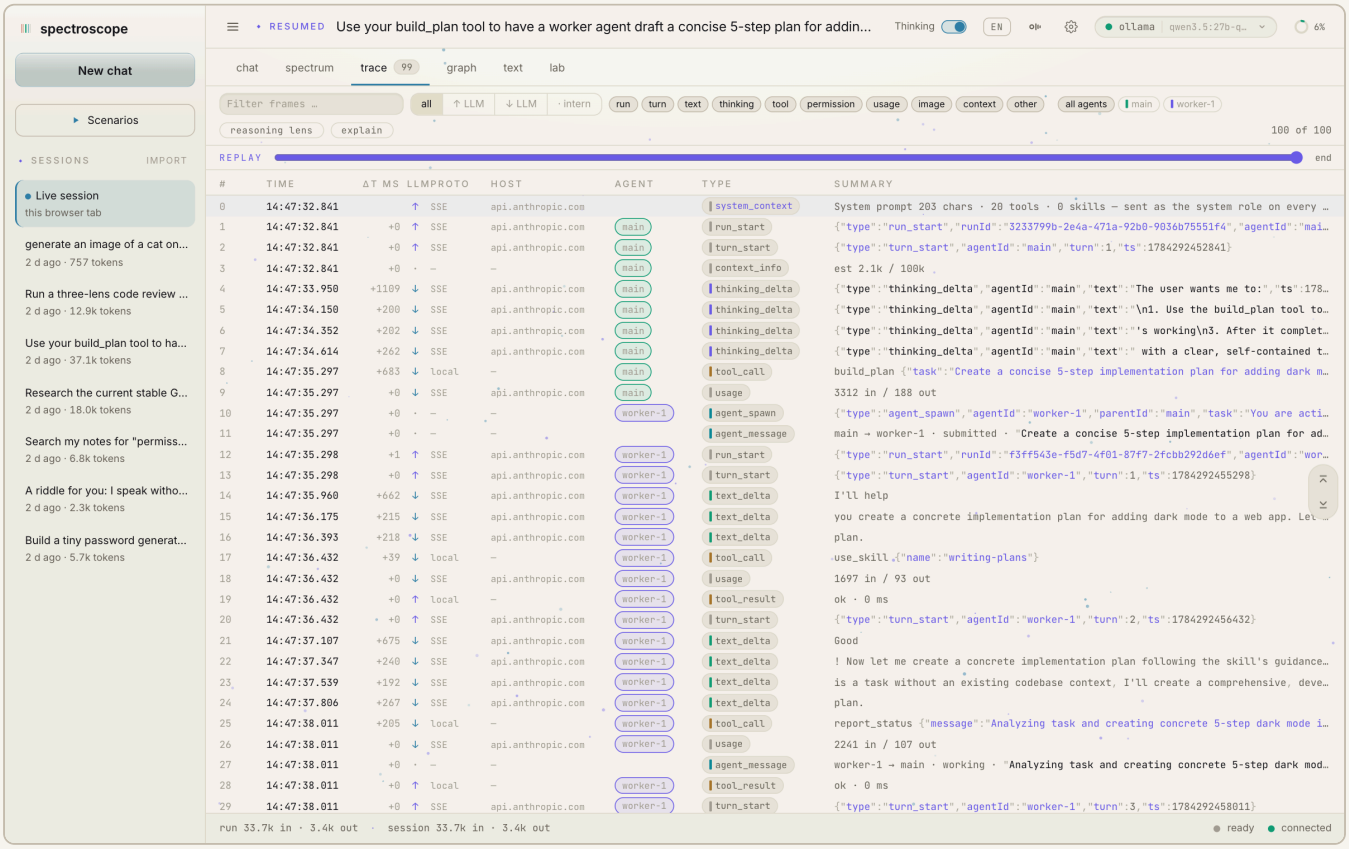
The sidebar lists every stored session newest-first: first prompt, relative age, token total. Opening one is a **replay** — the same events, folded by the same reducer, rendered read-only. Chat, Graph, Trace and Lab all work on it; the composer is replaced by the archive bar.



An archived session: the header eyebrow says Archive, the bar offers "Resume session", the two-step Delete, and "Return to live". All four tabs work on the replay.

Resume — with the re-upload made visible

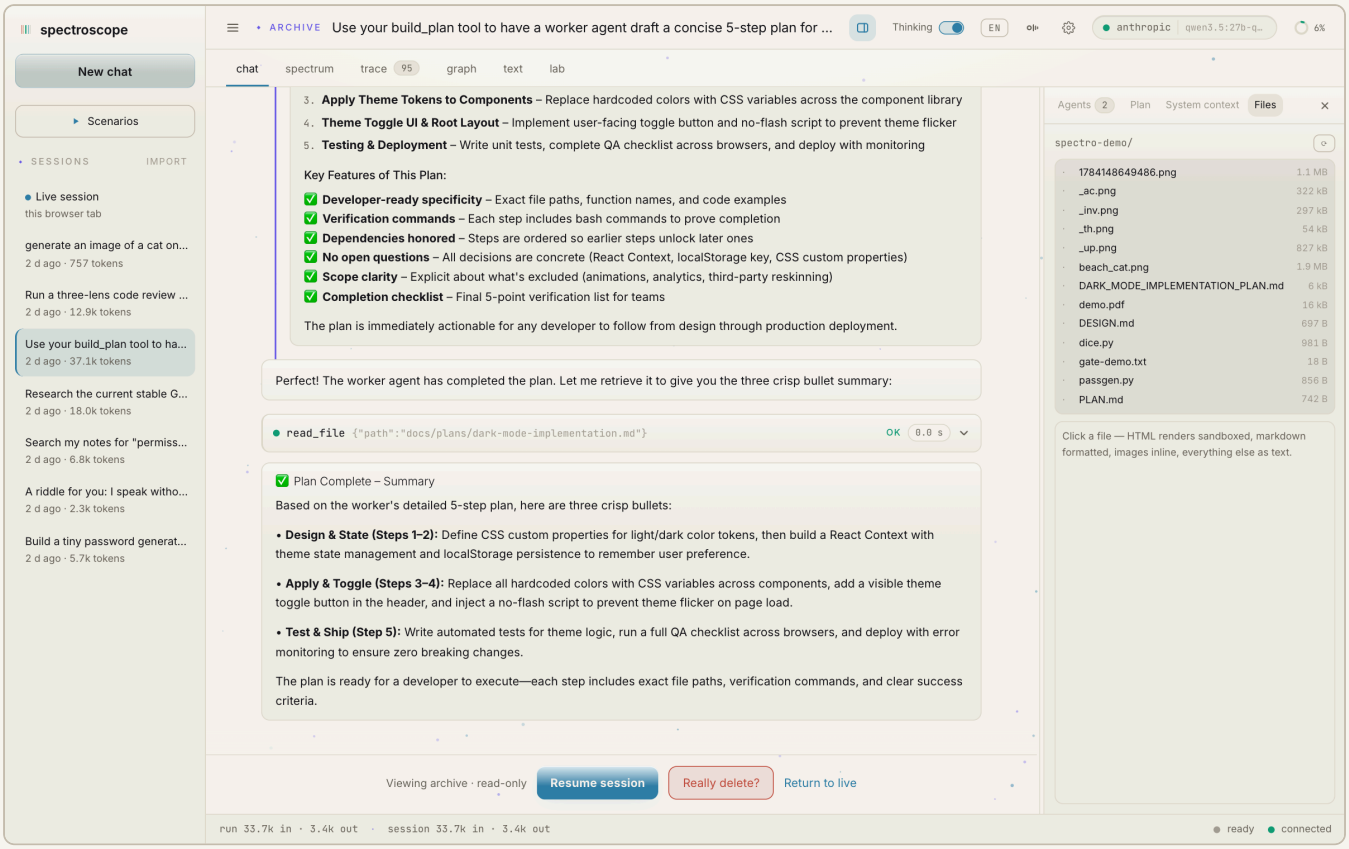
Resume session turns an archive back into the live session: the UI seeds itself from the stored events, the socket reconnects with `?resume=<id>`, the server reconstructs the provider history from the file (main agent only, orphaned tool calls dropped — chapter 22), and new events *append to the same file*. The header eyebrow flips to *Resumed*.



> After a resume, the trace shows the whole reconstructed history followed by the session_resume marker row: “n events loaded · ~t tokens of history ride along with the next request (plus system prompt & tool schemas on top).”

The didactic detail: resuming is not free. Your next prompt re-uploads the whole conversation to the model. The trace makes that visible with a UI-only `session_resume` **marker** between history and new traffic, summarizing how many events and roughly how many tokens now ride along. The estimate mirrors what the server actually reconstructs: main agent only (a subagent's inner steps never ride back up), thinking excluded (it never re-enters the provider history) — and the marker says explicitly that the system prompt and tool schemas come on top, so the next `usage` event's real jump lands where you expect it. The marker's detail view carries the entire re-uploaded history as its JSONL lines. Scenario and import replays offer no resume — there is no file to append to.

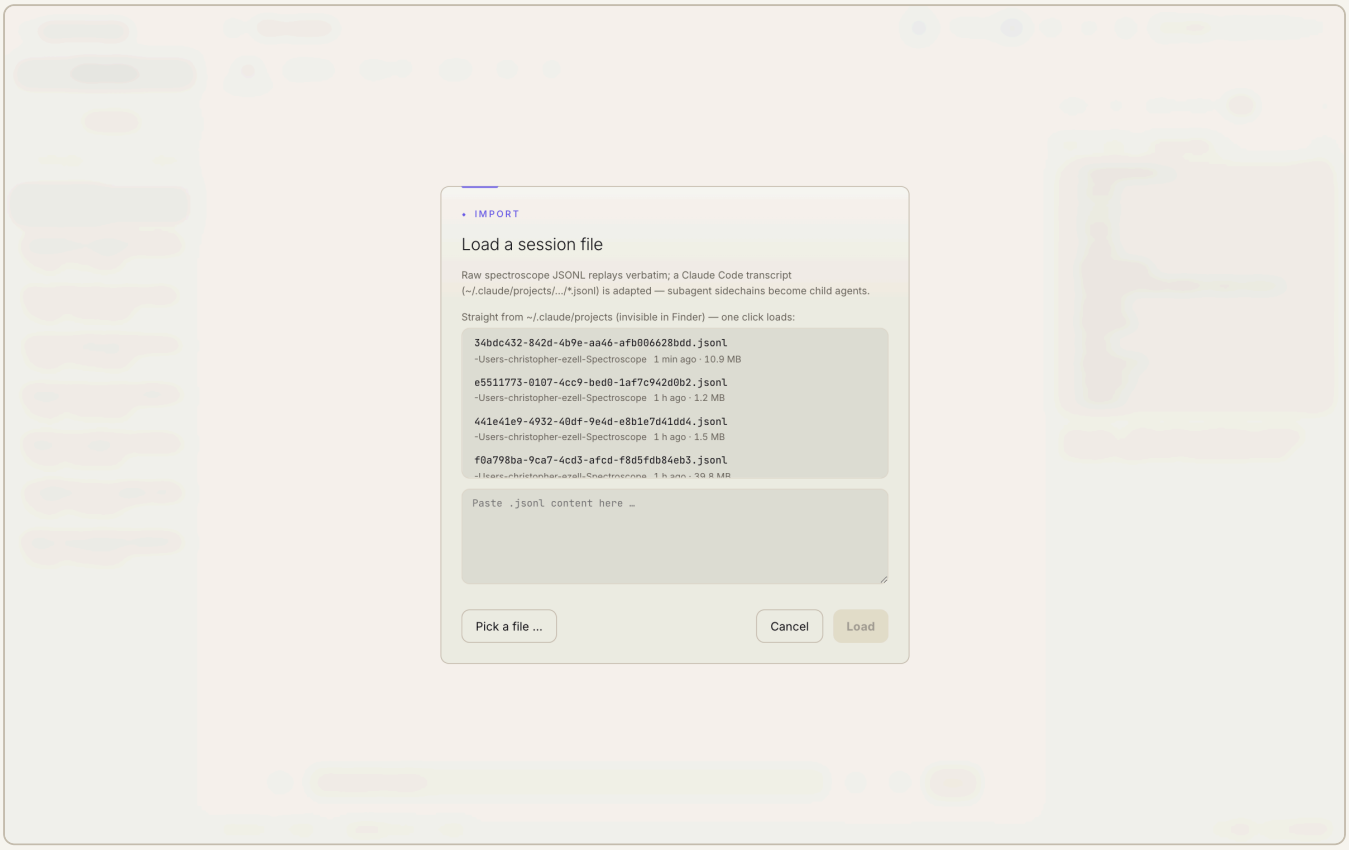
Delete — the two-step danger button



> First click arms: “Really delete?” in error tint. A second click within four seconds deletes; standing still disarms. Nothing else in spectroscope destroys data.

Deleting a session is the first deliberately destructive surface in spectroscope, so it is defended in depth: the button must be clicked twice within four seconds (first click arms, idle disarms); the REST endpoint (`DELETE /api/sessions/{id}`) guards the id shape before touching anything; and the store itself refuses any id that does not resolve to a direct child of the sessions directory — traversal ids delete nothing. Success removes the JSONL *and* the session's blob folder, closes the replay and refreshes the sidebar. The session currently being resumed by the live tab never offers the button.

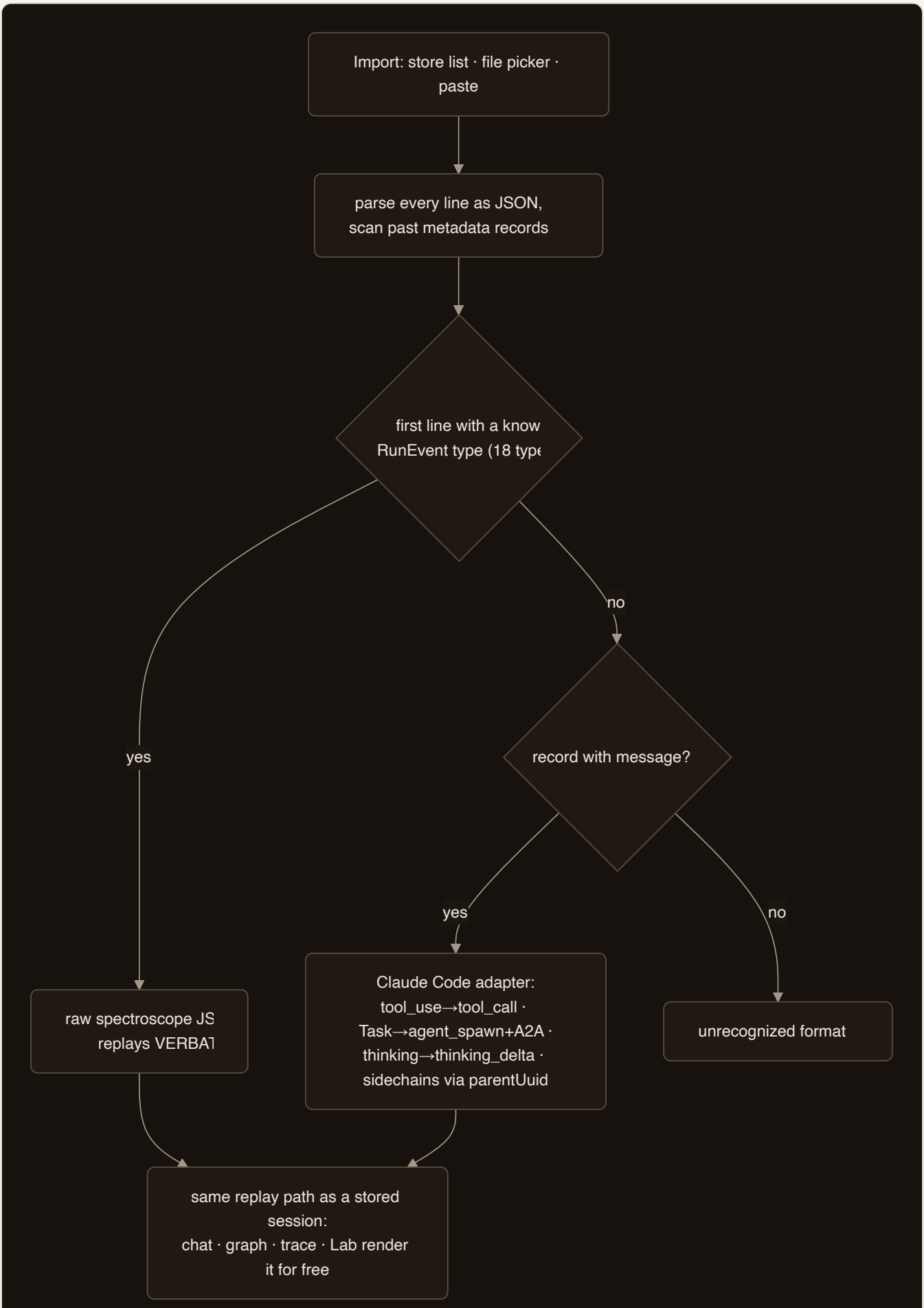
Import



> Three ways in: one-click list of your local Claude Code transcripts (the ~/.claude store is Finder-invisible — a file picker cannot reach it), a file picker, or paste.

The Import dialog replays *foreign* files through the same pipeline:

- **Raw spectroscope JSONL** — recognized by its event types, replayed verbatim. Files from the TypeScript edition work identically (the format is the contract).
- **Claude Code transcripts** — recognized by their `message` records and translated by an adapter: tool uses become `tool_call`s, `Task`/`Agent` spawns become `agent_spawn` + A2A messages, thinking becomes `thinking_delta`, sidechains fold under their owning task. You can watch a real Claude Code session step through the Lab's map.



Import detection: spectroscope JSONL replays verbatim; Claude Code transcripts run through the adapter; both feed the same replay path as stored sessions.

KNOWN LIMITATION

Newer Claude Code versions store subagent transcripts as separate files (`…/subagents/agent-*.jsonl`). Importing a main transcript shows each spawn and its result, but not the child's inner steps — a multi-file merge is a future candidate.

New chat semantics

One socket connection is one server-side session — one agent, one JSONL file. “New chat” therefore closes and reopens the socket: fresh agent, fresh file, fresh state. Two browser tabs are two fully independent sessions (separate agents, files, remembered permission rules); only persisted rules in `.spectro/settings.json` are shared, which is why that writer is serialized.

CHAPTER 9

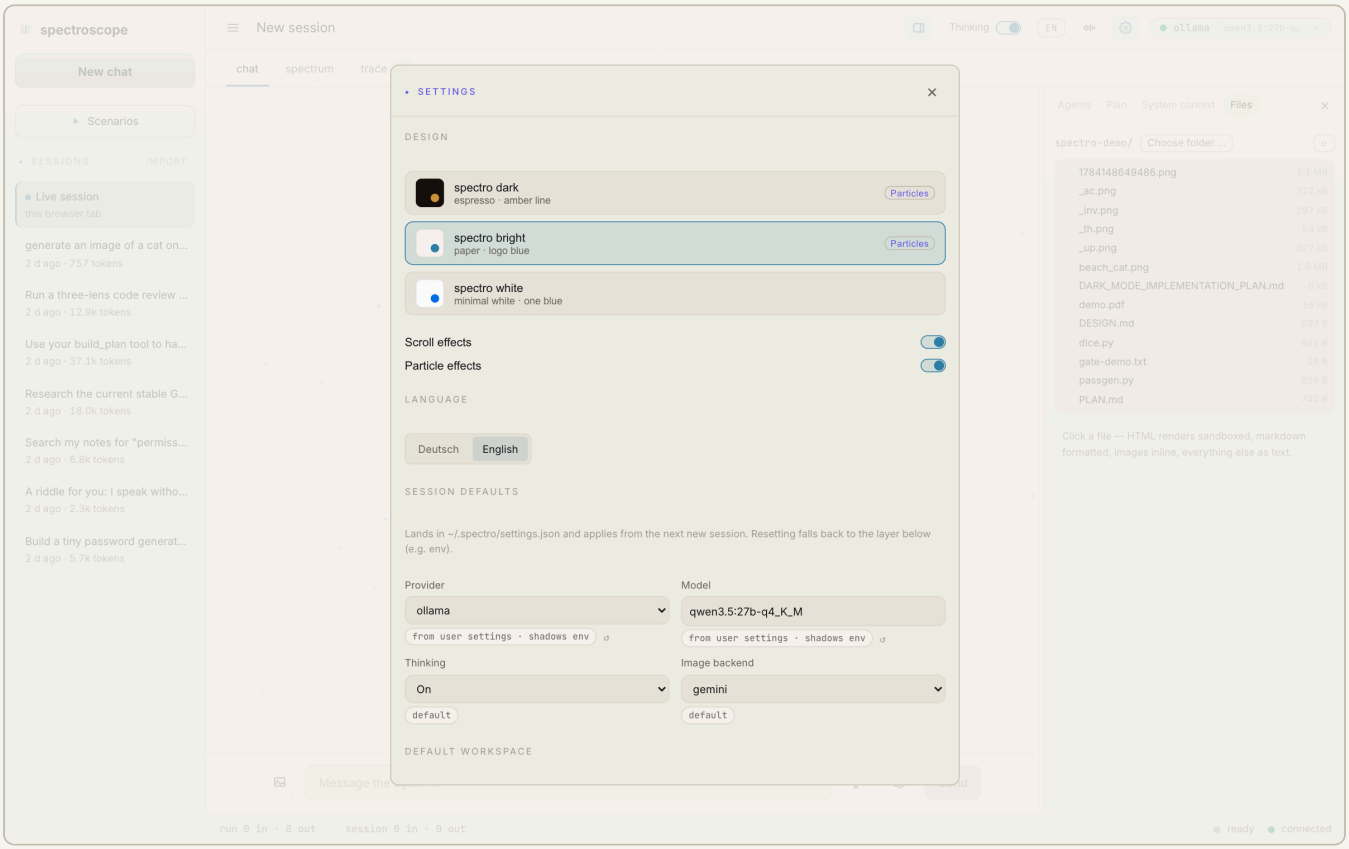
The design system

The whole UI reads its colours, fonts, radii and motion through CSS custom properties. That single indirection makes two things cheap: swapping the entire look (a *design*) and painting effects that always match.

The token base

The master vocabulary is brand-strict: espresso `#17120D` as the ground (never pure black), **the logo's amber line `#CE9440` exclusively for interaction and live elements** (links, buttons, focus, running indicators, the packet), **violet `#8B7CF0` as the editorial accent** (eyebrows, kickers — never clickable), desaturated status colours, and pastel agent accents that are decoration only. Focus is always visible and always the accent; reduced motion switches every animation off globally; numbers render tabular so nothing jitters mid-stream.

Three designs

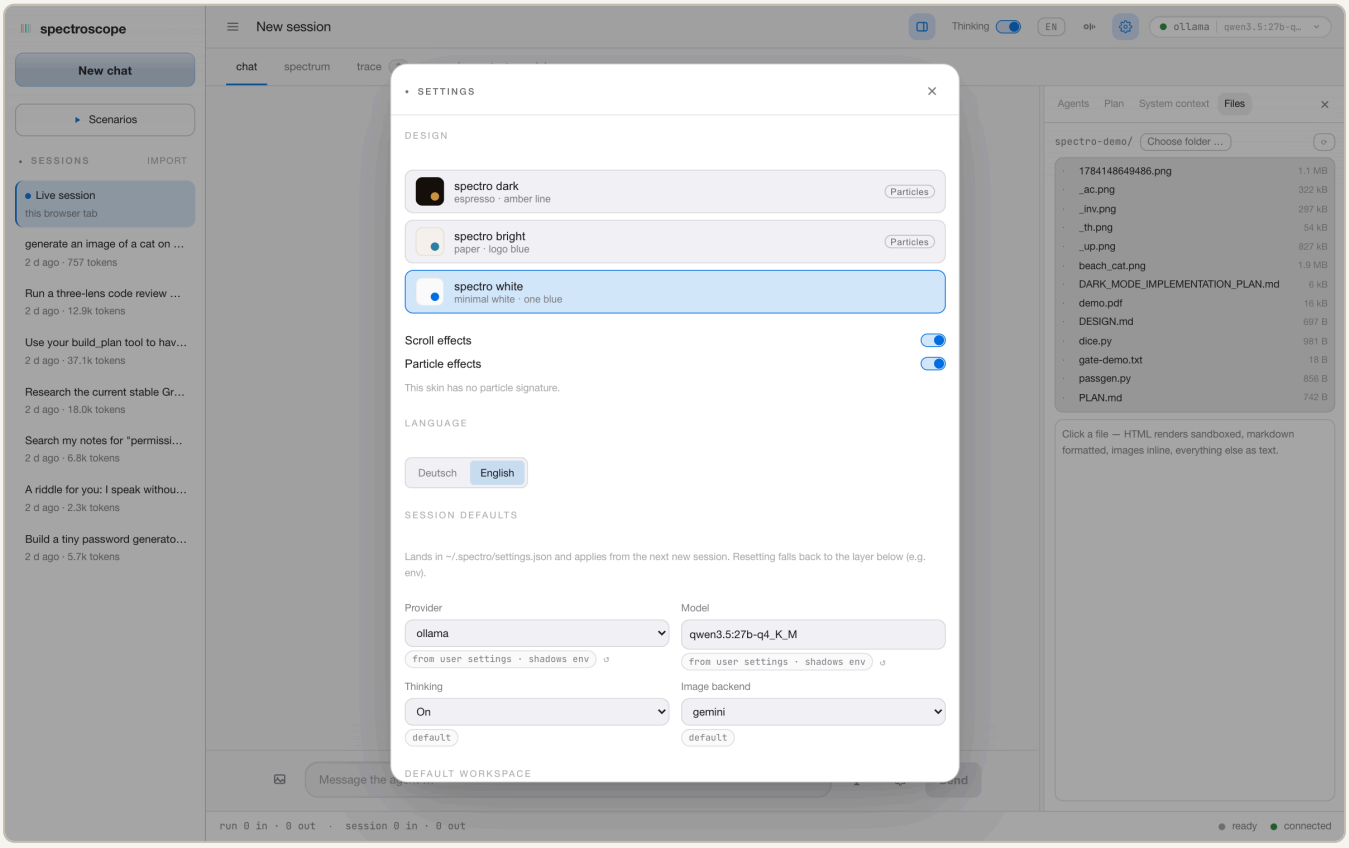


The design section of the settings page: the three designs with swatches, the effect switches beneath — a choice applies live and persists in one step.

A design is a token override set — nothing structural changes. Selecting one flips a `data-design` attribute on `<html>` and the UI re-expresses itself instantly, no reload; a FOUC guard applies the saved design before first paint.

DESIGN	DATA-DESIGN	CHARACTER	GROUND / ACCENT	PARTICLES
spectro dark	`spectroscope`	espresso · amber line (the default)	`#17120D` / `#CE9440`	brand dust
spectro bright	`paper`	LIGHT — paper · the logo blue	`#F6F4EE` / `#2E7EA6`	ink dust
spectro white	`still`	LIGHT — minimal white, gray, one blue	`#fbfbfd` / `#0071e3`	— (deliberately still)

The two **light** designs flip `color-scheme` and re-derive the status colours, agent pastels and the whole shadow set for a light ground — proof that the token vocabulary carries a full polarity flip, not just a palette swap.



> spectro white: the same UI in the quiet all-white design — every component reads var(--token), so the flip is total, live behind the settings page.

The effects layer

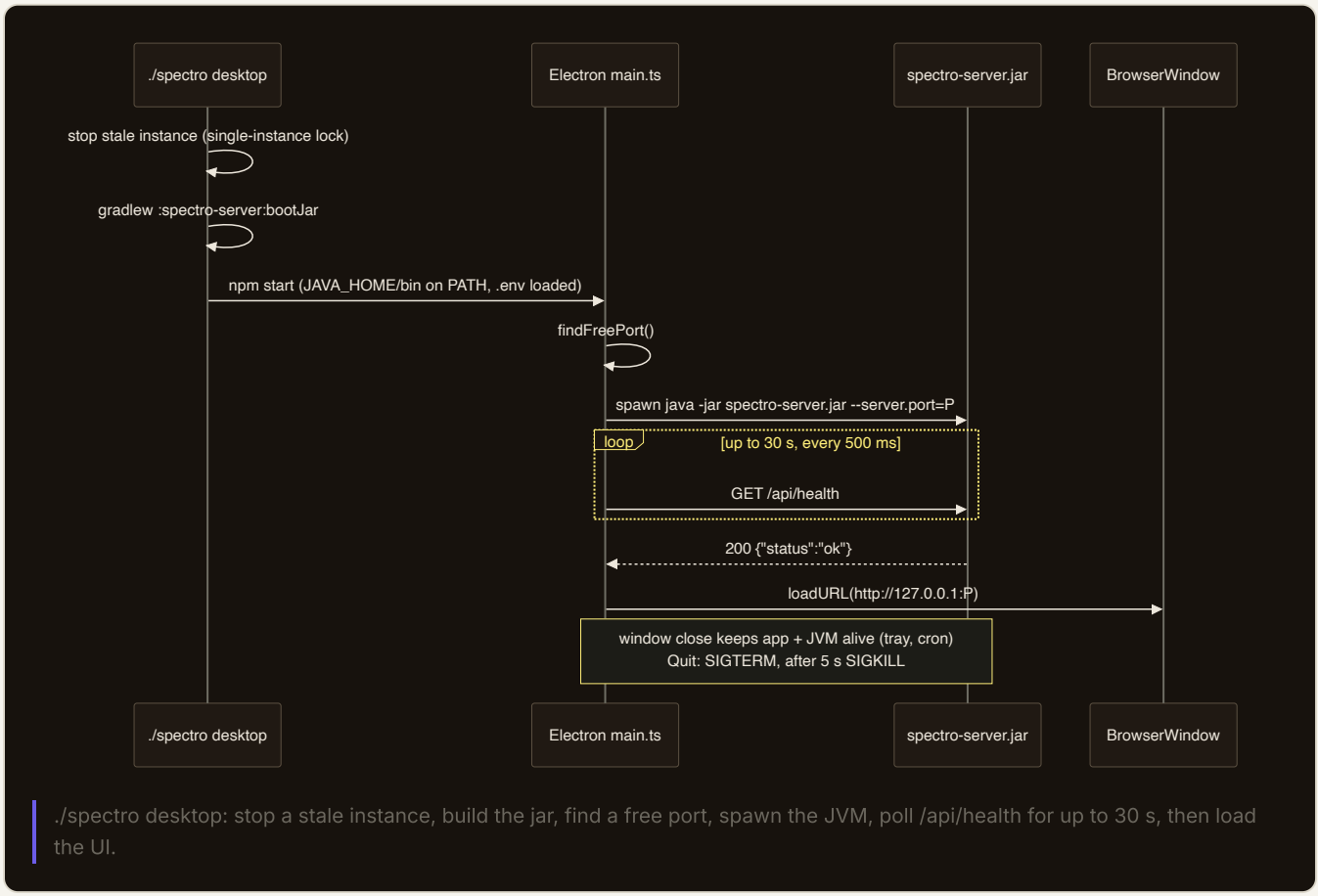
Two independently togglable effects, both reading the *active* design's colours live: the **particle field** (a canvas behind the UI with the brand dust drift — spectro white deliberately renders none) and the **scroll reveal** (panels fade/rise on entry). Both defer to reduced motion; particles pause in hidden tabs. A design choice from the settings page applies and persists in one step; the effect switches work the same way.

CHAPTER 10

The desktop face

A JVM cannot live inside Electron's main process — so the desktop app is honest about it: a thin shell that *supervises* the real server as a child process and points a window at it. The lesson of this face is process management, not IPC.

The supervision sequence



./spectro desktop: stop a stale instance, build the jar, find a free port, spawn the JVM, poll /api/health for up to 30 s, then load the UI.

`./spectro desktop` prepares the ground (stops a stale instance — Electron's single-instance lock would otherwise hand a relaunch to the *old* build; builds the boot jar; clears the macOS quarantine flag from fresh Electron binaries; puts the resolved JDK on the PATH). The Electron main process then finds a free port, spawns `java -jar spectro-server.jar --server.port=P`, polls `/api/health` every 500 ms for up to 30 s, and opens a 1200×800 window on `http://127.0.0.1:P` — the same React UI, talking over the same WebSocket. The renderer gets no Node API at all (`contextIsolation` on, `nodeIntegration` off): it is an ordinary web page.

Tray, notifications, quitting

- **Closing the window does not quit.** The tray (an ebony diamond) keeps shell and server alive so cron jobs keep running. Tray menu: New chat, Cron status, Quit.

- **Native cron notifications.** Every 30 s the shell polls `GET /api/jobs/state` and fires a desktop notification for each job whose status *changed* — click to open the window.
- **Quit is graceful, then firm.** ⌘Q / tray-Quit sends the JVM a SIGTERM (Spring Boot closes sockets and finishes the current JSONL line); if the process still lives after five seconds, SIGKILL follows.
- **Startup failures clean up.** If health never turns green, the shell kills the JVM before showing the error — no headless server left behind. A missing `java` shows “Java 21 is required”.

CHAPTER 11

Two languages

The UI chrome ships fully bilingual — German (the default) and English — switched live by the header toggle. This guide's screenshots use the English chrome.

- **Chrome only, content never.** Sidebar, header, dialogs, panels, tool cards, Lab toolbar, the in-map SVG texts, trace chrome, settings page, footer — all switch live (~330 dictionary entries). Chat content, tool payloads and the JSONL wire never run through translation: a session keeps its own language.
- **Wire values stay English.** Plan statuses (`pending / in_progress / completed`), A2A states and stop reasons are canonical English on the wire — only labels localize. Cross-edition compatibility depends on it.
- **Scenarios compile per language.** Picking a scenario compiles it in the current chrome language; the compiled session then keeps that language, like any other session.
- **Late-bound info lines.** Reducer-folded info lines (spawns, compactions) carry an additive key + variables, so lines already in the transcript re-render when you flip the language.

PART III

The agent's powers

Everything the model may do, and everything that keeps it honest: the tool belt, the permission system, hooks, subagents, skills, providers, the senses — vision, voice, images — MCP and the scheduler.

CHAPTER 12

The tool belt

Eighteen built-in tools, plus one dynamic adapter per connected MCP tool. Every tool obeys one iron contract: **it never throws** — failures come back as a string starting `ERROR:`, which the model reads and self-corrects from. Every model-supplied path passes the sandbox. Every mutation passes the gate.

The shared rules

THE PATH SANDBOX

Every file tool resolves paths relative to the working directory and refuses anything that normalizes outside it (`ERROR: path is outside the working directory`). Tree-walking tools additionally verify each hit's *real* path (symlink-escape guard) and never enter `.git`, `build`, `node_modules`, `target`.

THE CAPS

Files: 50 kB read/edit ceiling. Output: 10 000 chars shared clamp (surrogate-safe truncation). Glob: 200 results. Shell: 10 s timeout, pipe drained on a background thread — a timeout means "genuinely hung", never "output too large". Web: 512 kB streamed body cap before the text clip.

Files & search

`read_file` free — input: { path }

Reads a UTF-8 text file (≤ 50 kB) relative to the working directory. *Try: "Read build.gradle.kts and tell me the dependencies."*

`list_dir` free — input: { path }

Lists a directory, sorted, directories marked with a trailing slash; an empty directory answers `(empty)`.

`glob` free — input: { pattern, path? }

Finds files by glob pattern (`**/*.java`) under a directory — pure Java, pruned walk, symlink-guarded, results sorted and capped at 200. Also granted to explore subagents. *Try: "Find every *.test.ts file under spectro-web."*

`grep` free — input: { pattern, path?, glob? }

Searches file contents by Java regex, returning `path:line:text` hits — no shell involved, so it works even where `run_command` is denied. Skips oversized and binary files; invalid regex answers `ERROR: invalid regex: …`. *Try: "Search for TODO across the Java sources."*

Mutation & shell

`write_file` gated · remembers path — input: { path, content }

Writes a text file, creating parent directories. Success: `Wrote: <path> (<n> bytes)`. An "always allow" remembers the *full path*, never a blanket approval.

`edit_file` gated · remembers path — input: { path, old_string, new_string, replace_all? }

Surgical exact-string replacement. `old_string` must be unique (`ERROR: old_string is not unique (n matches)` otherwise) unless `replace_all`; zero matches and empty `old_string` error readably. A denied edit writes nothing. *Try: "In README.md, change the title X to Y."*

`run_command` gated · remembers first token — input: { command }

Runs a shell command in the working directory via `/bin/sh -c`: stderr merged, output drained concurrently and capped, 10 s timeout, killed on abort. Non-zero exit returns `ERROR: exit code <n>` *plus the output* — exit-code feedback is what lets the model fix its own mistakes. An "always allow" remembers only the first token (`run_command:git*`).

`web_fetch` gated · remembers URL — input: { url }

Fetches an http/https page and returns readable text (scripts/styles dropped, tags stripped, entities decoded, whitespace collapsed, 10 k-char clip). Network egress on model-chosen input is a side effect — hence the gate. Streaming fetch with finite 15 s timeouts and a 512 kB body cap; non-2xx answers `ERROR: web_fetch got HTTP <status>`. Main agent only.

Metadata & delegation

`update_plan` free · main only — input: { steps: [{text, status}] }

Publishes the agent's step list to the Plan panel (chapter 6) as the additive `plan` event. Latest wins — each call replaces the whole plan. The status enum (`pending / in_progress / completed`) is *enforced*: a model improvising `"done"` gets an error instead of polluting the wire. Deliberately not given to subagents — a worker's plan would clobber the display.

`use_skill` free — input: { name }

Returns the full instructions of a named skill (chapter 15). Registered only when skills are installed; children get it too, so dev-tool workers can load their role skill.

`generate_image` gated — input: { prompt }

Generates an image via the configured image provider (chapter 17). Gated because it is a paid cloud call. The image lands content-addressed under `~/.spectro/images/`; the model sees only a short confirmation — the user sees the gallery.

`spawn_agent` / `spawn_agents` free · parent only — input: { type, task } / { agents: [...] }

Starts one subagent (or up to four in parallel) and waits for the result — chapter 14. Free of a gate because the *child's* tools ask for permission individually; delegation is not an escape hatch.

`build_plan` · `write_spec` · `develop` · `test` free · parent only — input: { task }

The four development role tools — thin wrappers over a worker spawn, each pointing its child at a skill (chapter 14).

`report_status` free · children only — input: { message }

The child's side of the A2A protocol: one short progress sentence per milestone, surfacing live in the parent's UI. Telling the parent what you do is never dangerous.

`mcp__<server>__<tool>` gated · dynamic — schema from the server

Every tool a connected MCP server advertises, wrapped as a first-class spectroscope tool (chapter 18). Always gated: its inputs are model output and its effects are external.

Who gets which tools

AGENT	REGISTRY
`main`	all of the above (17 named + MCP; `use_skill` when skills exist)
`worker-N`	list_dir, read_file, write_file, run_command, edit_file, glob, grep, use_skill*, **report_status**
`explore-N`	list_dir, read_file, glob, grep, **report_status** — read-only *by construction*: everything else simply is not in its registry

* when skills are installed. Children never receive: spawn/dev tools (nesting depth 1 is structural), generate_image, web_fetch, update_plan, MCP tools.

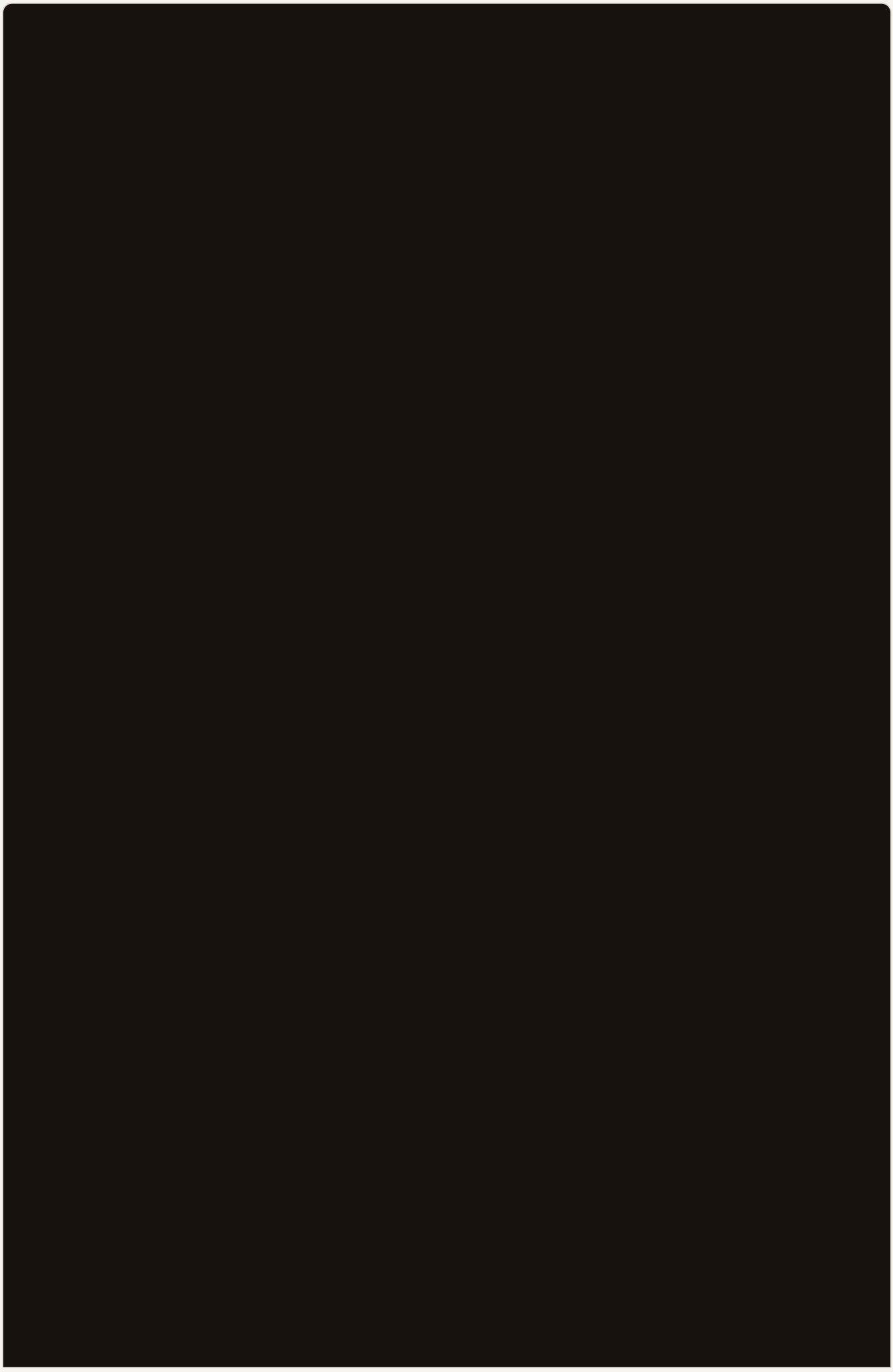
CHAPTER 13

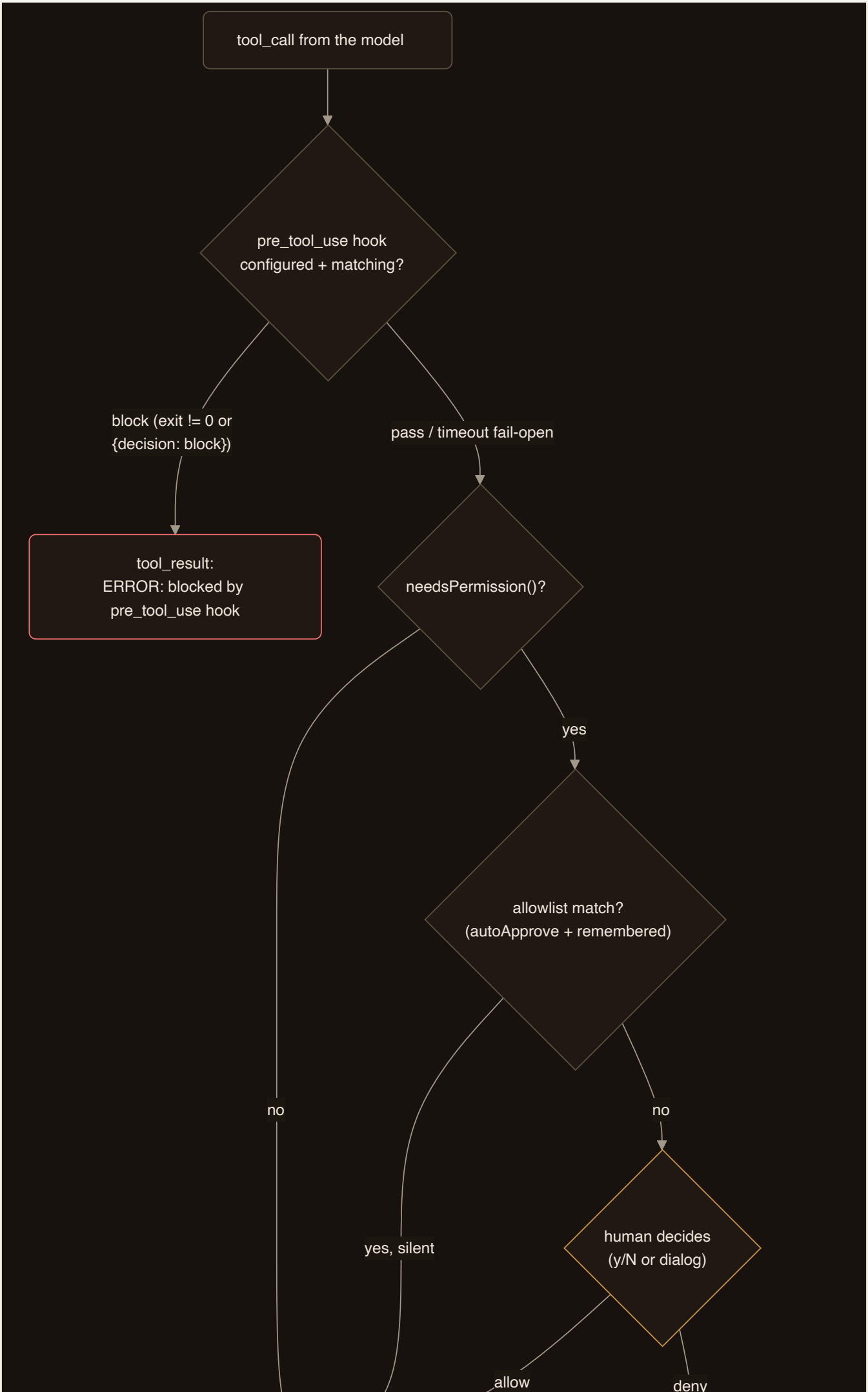
Permissions, allowlist & hooks

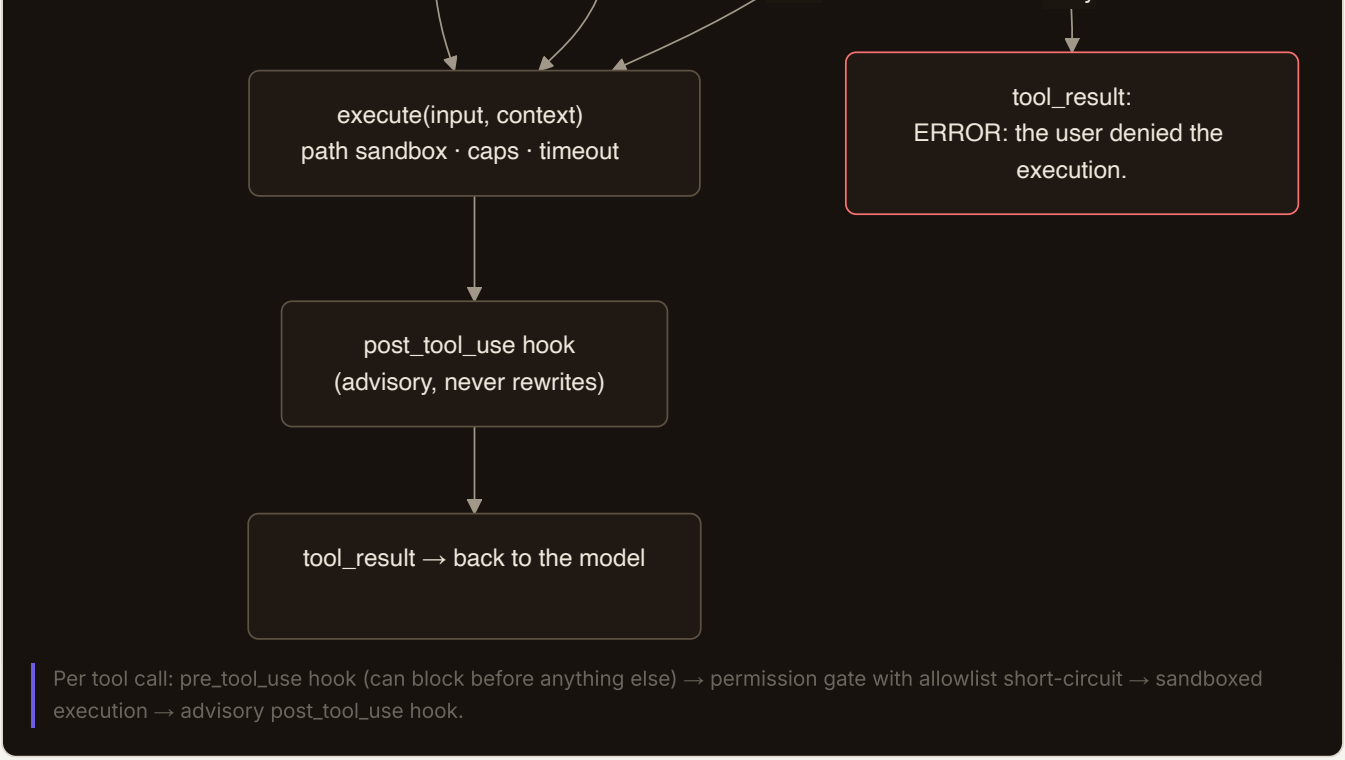
You stay in control through three layered mechanisms: the permission gate (a human decides), the allowlist (you decide once, scoped), and hooks (your scripts decide, before the gate even asks). All three leave an audit trail in the session file.

The guard pipeline

Every single tool call passes this pipeline, in this order — the order is load-bearing:




tool_call from the model
pre_tool_use hook configured + matching?
block (exit != 0 or {decision: block})
pass / timeout fail-open
tool_result: ERROR: blocked by pre_tool_use hook
needsPermission()?
yes
allowlist match? (autoApprove + remembered)
no
no
yes, silent
human decides (y/N or dialog)
allow
deny



Per tool call: pre_tool_use hook (can block before anything else) → permission gate with allowlist short-circuit → sandboxed execution → advisory post_tool_use hook.

Permission modes

MODE	MEANING	WHERE
`ask`	gated tools pause the run until a human decides (terminal y/N, web dialog) — with the allowlist answering first	default; the interactive faces
`readonly`	every gated tool is denied automatically — the agent can look, not touch	headless default (`spectro run`, cron): "capability before convenience"
`auto`	every gated tool is approved automatically	opt-in for unattended runs: `--permissions auto` / per-job `"permissions": "auto"`

Even automatic policies emit `permission_request` / `permission_decision` events — the JSONL always shows who allowed what.

The allowlist

Rules live under `autoApprove` in any settings layer and are consulted before any human is asked:

```
{ "autoApprove": [
    "grep",                      // a bare name approves every call of that tool
    "run_command:git status*",   // a prefix rule approves matching inputs only
    "write_file:docs/notes.md*"
] }
```

Matching reads each tool's **guarded field** — the one input that defines its blast radius. The same table drives what an "always allow" click remembers, so a remembered rule can never silently go dark on the matching side:

TOOL	GUARDED FIELD	AN "ALWAYS ALLOW" CLICK STORES
`run_command`	`command`	the first token: `run_command:git*`
`write_file` / `edit_file`	`path`	the full path: `edit_file:src/Main.java*`
`web_fetch`	`url`	the full URL: `web_fetch:https://example.com*`

everything else	—	the bare tool name

Session-remembered rules live per connection (two tabs never share them). Ticking **Persist** appends the rule to the session's **workspace** project file (`<workspace>/.spectro/settings.json`) through a single serialized writer that preserves every other key and dedupes — falling back to the deprecated launch-dir `.spectro/settings.json` only for a session with no real workspace yet. Because the CLI resolves the same workspace, it honours your web decision there on its next run.

Hooks

Hooks are config-driven shell commands around every tool call — your policy, in your scripting language, versioned with the project. They come *only* from config, never from tool input: a pre-hook is arbitrary shell running before the gate, so it must never be model-controlled.

```
{ "hooks": [
    { "event": "pre_tool_use",          // or post_tool_use — anything else fails LOUDLY
      "matcher": "run_command",         // tool-name glob: * (default), exact, or prefix*
      "command": "./guard.sh",          // any shell string
      "timeoutSeconds": 5 }             // optional, default 10
] }
```

ENVIRONMENT	the hook process sees `SPECTRO_TOOL_NAME`, `SPECTRO_TOOL_INPUT` (compact JSON) and — post only — `SPECTRO_TOOL_RESULT`; cwd is the agent's working directory.
PRE_TOOL_USE	runs **before the permission gate** and blocks on exit ≠ 0 *or* a stdout verdict `{"decision":"block","reason":"…"}`. A block short-circuits: no dialog, no execution — the model receives `ERROR: blocked by pre_tool_use hook: <reason>`. First matching block wins.
POST_TOOL_USE	runs after execution, **advisory only** — exit code ignored, the result is never rewritten. For logging, notifications, formatters.
FAIL-OPEN TIMEOUT	a hanging hook must not wedge every tool call — on timeout the pipeline continues to the normal gate. Safe because the shared runner drains the pipe concurrently: a chatty hook cannot deadlock itself into the timeout.
SUBAGENTS INCLUDED	children run the *same* hook runner — a hook that blocks a tool also blocks it on a child, or delegation would be a bypass.
WHOLE-BLOCK MERGE	like `mcpServers`: a settings layer that defines `hooks` replaces the entire block below it.

EXAMPLE — VETO EVERY RM

In `.spectro/settings.json`:

```
{ "hooks": [ {
  "event": "pre_tool_use", "matcher": "run_command",
  "command": "case \"$SPECTRO_TOOL_INPUT\" in *'rm '*) echo '{\"decision\":\"block\",\"reason\":\"rm
is off limits here\"}' ;; esac"
} ] }
```

Ask the agent to delete a file with `rm` and it instantly receives `ERROR: blocked by pre_tool_use hook: rm is off limits here` — no permission dialog, no execution. `spectro doctor` shows `hooks: 1 configured (pre_tool_use:run_command)`. Verified live.

The audit trail

Every decision is an event: `permission_request` (with the full input) and `permission_decision` (allowed or not) land in the JSONL even for allowlist auto-approvals; hook blocks are visible as their error result. The Trace tab's permission category shows the whole history; the Lab's gate row turns green or red as you step onto it.

CHAPTER 14

Subagents & the A2A protocol

A subagent is not a special construct — it is another `Agent` from the same core, with its own fresh context, its own reduced tool registry, and the same permission gate and hooks as its parent. What makes spectroscope's delegation special is that the protocol between the agents is *visible*: a task/status/result lifecycle you can watch in every view.

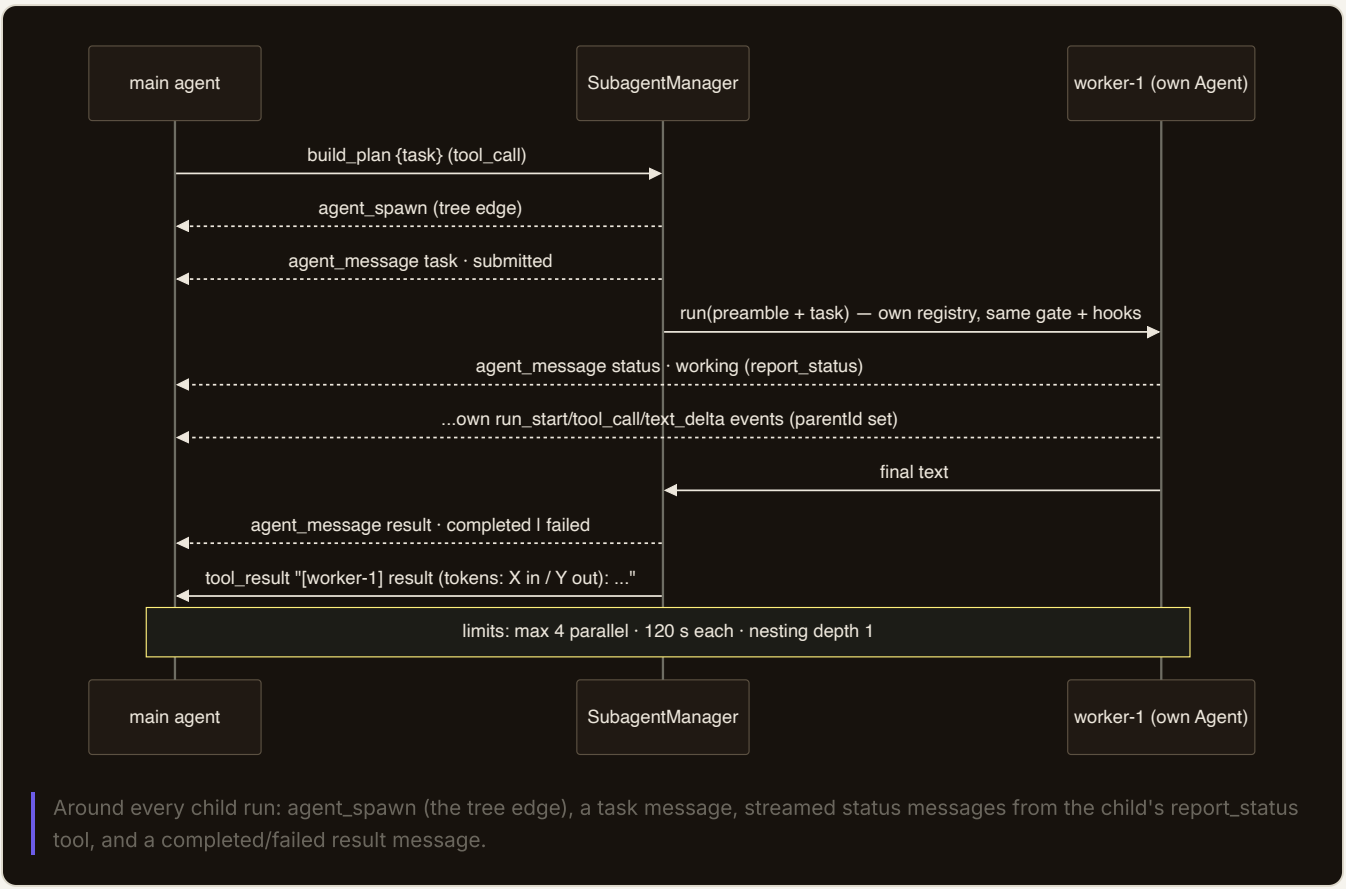
Two roles

ROLE	PURPOSE	TOOLS
`explore`	read-only research: map a directory, find usages, summarize	list_dir, read_file, glob, grep + report_status — **read-only by construction**
`worker`	real subtasks that may change things	the full standard set + use_skill + report_status

Children see *only their task text* — no conversation history — which is the point: a fresh, focused context. Their final text is all the parent receives, wrapped with the token cost so delegation visibly costs something: `[worker-1] result (tokens: 2.1k in / 0.4k out): …`

LIMITS	at most 4 parallel children (schema-enforced and runtime-checked); 120 s per child (then `ERROR: [worker-1] timeout after 120 s — cut the subtask smaller.`); nesting depth 1 — children simply do not have the spawn tools.
CANCELLATION	each child has its own CancelSignal; the parent's cancel cascades — Ctrl+C or Stop ends the whole tree cleanly.
ISOLATION THAT ISN'T	same working-directory sandbox, same permission broker (the dialog names the asking child), same hooks. Delegation never widens what is allowed.

The A2A-lite lifecycle



Around every child run: agent_spawn (the tree edge), a task message, streamed status messages from the child's report_status tool, and a completed/failed result message.

Two additive events carry the protocol — the wire format needed no break. Every spawn emits `agent_spawn` (child id, parent id, task) followed by an `agent_message` with role `task`, state `submitted`. While working, the child's permission-free `report_status` tool publishes role-`status` messages ("working") that surface live in the Agents panel, the chat thread header and the Lab cards. At the end — on *every* path, including timeout and cancellation — a role-`result` message closes the child with state `completed` or `failed`.

Where you see it: chat nests child turns into pastel threads (chapter 5); the Graph tab draws split/join blocks per wave (chapter 6); the Lab gives every child its own loop card with its own packet (chapter 7); the Agents panel keeps the roster with token costs (chapter 6); the CLI renders `[worker-1]`-prefixed lines (chapter 4).

The development tools

Four role tools wrap a worker spawn with a persona and a skill — no new agent types; specialization is prompt + skill:

TOOL	PERSONA	LOADS SKILL	DELEGATES
`build_plan`	senior PLANNER	`writing-plans`	a feature request → a step-by-step implementation plan (a written document)
`write_spec`	requirements ANALYST	`brainstorming`	a rough idea → a design/spec with decisions and trade-offs
`develop`	IMPLEMENTER	`test-driven-development`	a development task, carried out in small verified steps

`test`	TESTER — “Do NOT fix anything.”	`verification`	runs and inspects, reports evidence, changes nothing

The child's prompt is the persona preamble + a standing instruction to report progress via `report_status` + your task text; the A2A task message shows *your* task with the tool name as its label — which is exactly how the UI badges the thread. *Try: “build_plan: draft a plan for adding CSV export to the sessions list.”*

CHAPTER 15

Skills

Skills extend the agent with *data, not code*: markdown instruction packages the agent loads when a task matches. The loop is untouched, no new event type exists — a skill use is an ordinary tool call you can see in every view.

The package format

```
~/.spectro/skills/<folder>/SKILL.md        (user scope)
<project>/.spectro/skills/<folder>/SKILL.md   (project scope — wins by name)

---
name: test-driven-development
description: Red-green-refactor discipline for every change - write the failing
  test first, watch it fail for the right reason, make it pass minimally, then clean up.
---
# The skill body — full markdown instructions…
```

Frontmatter is hand-parsed (`name:`, `description:`); a file without it falls back to the folder name and the first body line. Broken skills are skipped with a warning — they never take the harness down.

Progressive disclosure

Only the *catalog* — one `name: description` bullet per skill — rides in the system prompt under a `## Skills` header, with the standing instruction to call `use_skill` before matching work. The body loads on demand. That keeps the context small however many skills you install, and makes skill usage visible: you can watch the agent fetch its instructions in the chat and trace.

The four shipped skills

SKILL	TEACHES
`brainstorming`	turn a vague idea into an agreed design before code: one question per turn, real alternatives, decisions recorded
`test-driven-development`	red → green → refactor with honest verification rules (no network in tests, one behavior per test)
`writing-plans`	plans a fresh engineer could execute: exact files, verifiable steps, no open questions
`verification`	evidence before claims — never state what you did not observe in this run; report VERIFIED / FAILED / UNVERIFIABLE per claim

`/skills` lists what is installed; `spectro doctor` counts them; the System-context panel shows the catalog exactly as the model sees it. The four development tools of chapter 14 point their children at these skills.

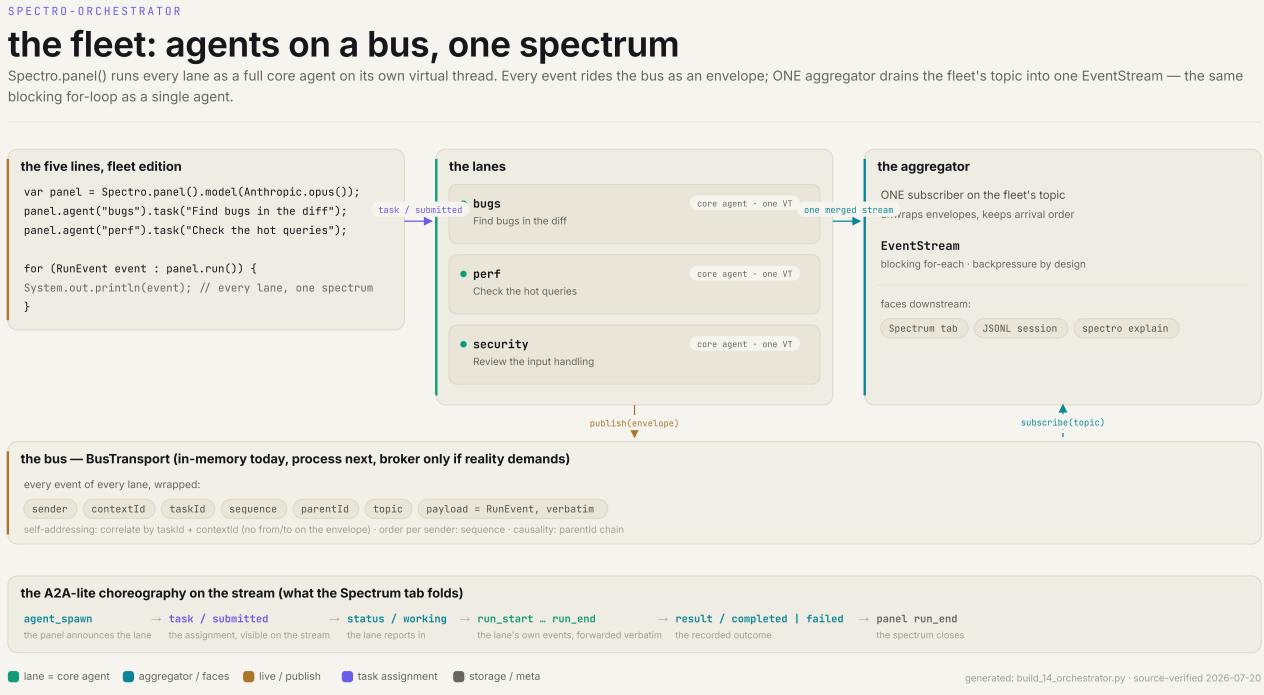
the fleet: Spectro.panel() and the orchestrator

Subagents fan out under one main agent. The fleet is the next level: several full agents, each with its own model, tools and workspace, run in parallel and merge into one spectrum. That is the `spectro-orchestrator` module (migration phase 5), and its facade follows the frozen five-lines style exactly:

```
var panel = Spectro.panel().model(Anthropic.opus());
panel.agent("bugs").task("Find bugs in the diff");
panel.agent("perf").task("Check the hot queries");

for (RunEvent event : panel.run()) {
    System.out.println(event);   // every lane, one spectrum
}
```

Every lane is a complete core agent on its own virtual thread. The panel announces each lane on the stream (`agent_spawn`, then a task message), the lane reports in with a working status, its whole event stream rides along verbatim, and a result message records completed or failed before the panel's own `run_end` closes the spectrum. One broken lane never kills the fleet; it fails loudly and alone.



The fleet architecture: the frozen facade, lanes as core agents, the bus with its envelopes, one aggregator into one stream.

what rides the bus

Under the surface, every event of every lane is wrapped in a bus envelope: sender, contextId, taskId, a per-sender sequence, a causal parentId chain, a topic for session isolation, and the RunEvent itself, verbatim. Addressing is self-addressing: consumers correlate by taskId within a contextId instead of

from/to routing. The transport behind it is a seam. In-memory serves the facade today; a process transport is the designed next step, and a broker only joins if reality ever demands one.

> WHERE IT DOCKS
>
> The bus's future home is the tracing seam: one registry in the core where the JSONL sink, an OTLP exporter and the bus publisher hang side by side. Until that port lands, the panel publishes from the outside. The envelopes and their consumers stay identical either way.

watching a fleet

The spectrum tab folds a panel run with no extra wiring: the panel is a lane, every agent is a lane, task/status/result pulses and gate states land where they belong, and a lane click pins the trace. Export a panel run to JSONL and the import dialog replays the whole fleet, exactly as recorded.

CHAPTER 16

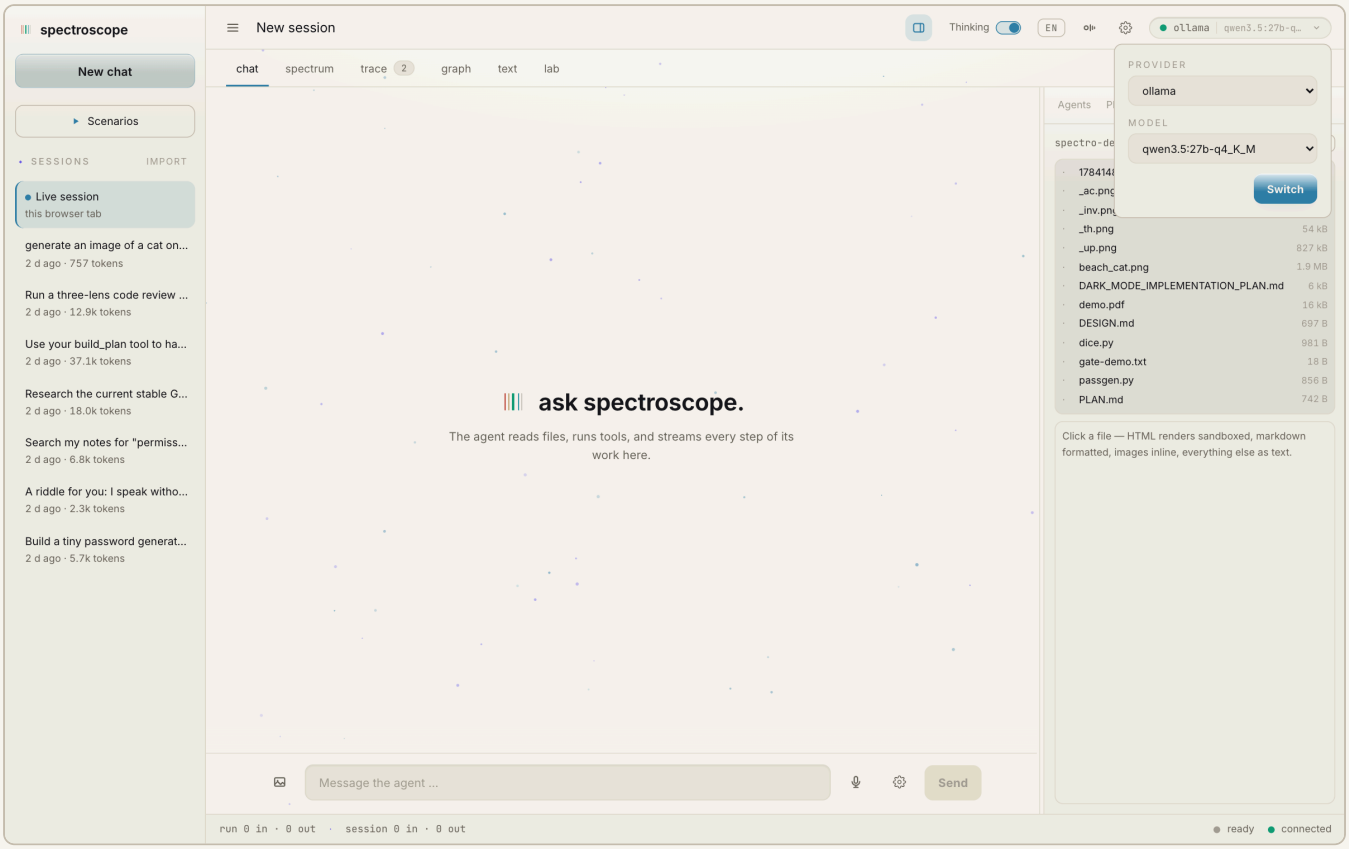
Providers & models

Everything the loop needs from a model sits behind one narrow port: `Iterable<ProviderEvent> stream(request)`. Swap the implementation and the loop never notices — which is why spectroscope can switch between a cloud API and a local model mid-session.

The three backends

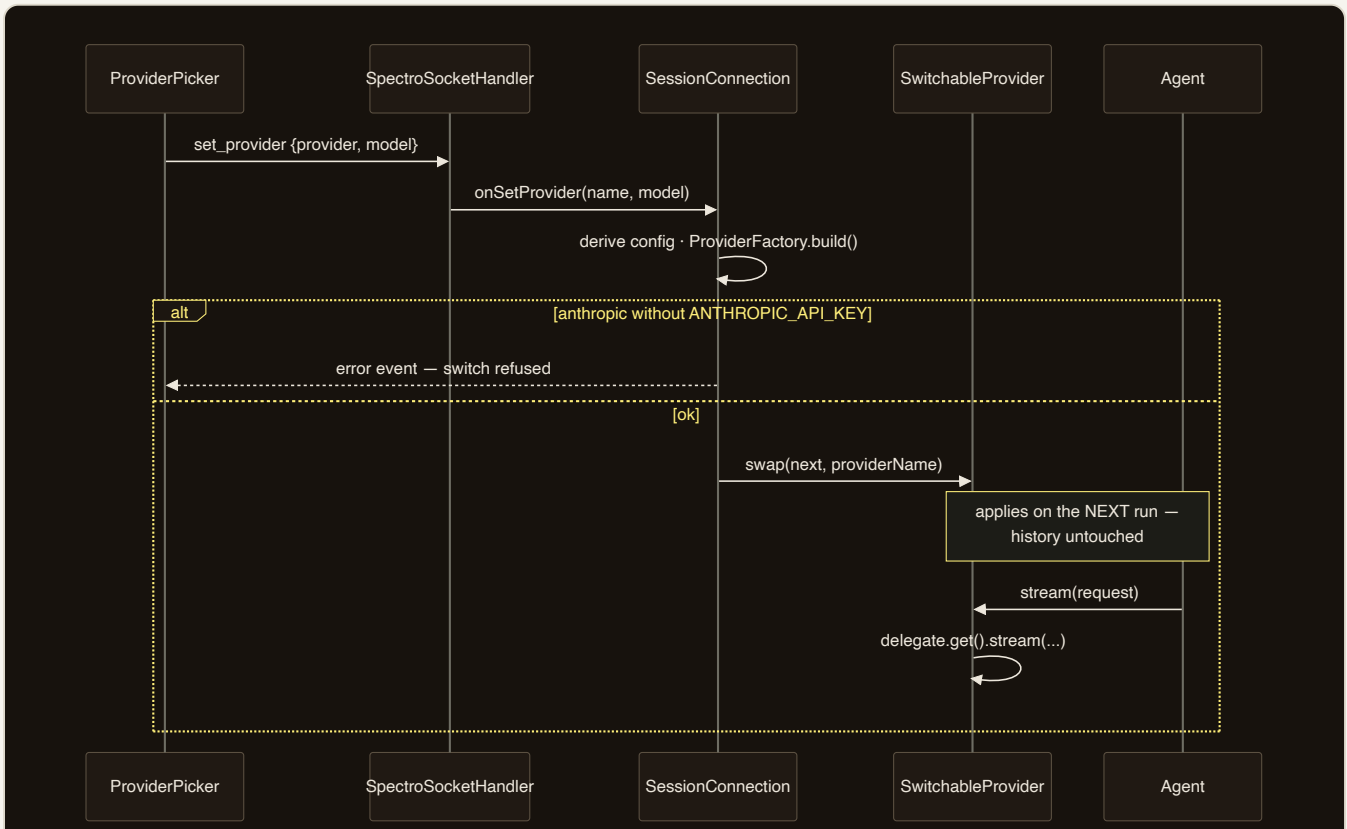
	ANTHROPIC	OLLAMA	OPENAI
transport	official SDK, SSE	NDJSON over `POST /api/chat`	SSE over `POST /v1/chat/completions`
auth	`ANTHROPIC_API_KEY`	none	Bearer optional (LM Studio & friends run keyless)
default model	`claude-opus-4-8`	`qwen3`	`local-model` (the loaded one)
thinking	extended thinking (2048-token budget)	native `message.thinking` + inline `<think>` tag splitting	—
quirks	usage + parsed tool inputs exist only after the stream ends	mints its own call ids; vision fail-fast for non-vision models	assembles fragmented tool-call deltas per index

Switching mid-session



> The header picker: provider select plus a real model dropdown — Ollama's list is live from the server, cloud lists are curated, and a custom model is always typeable.

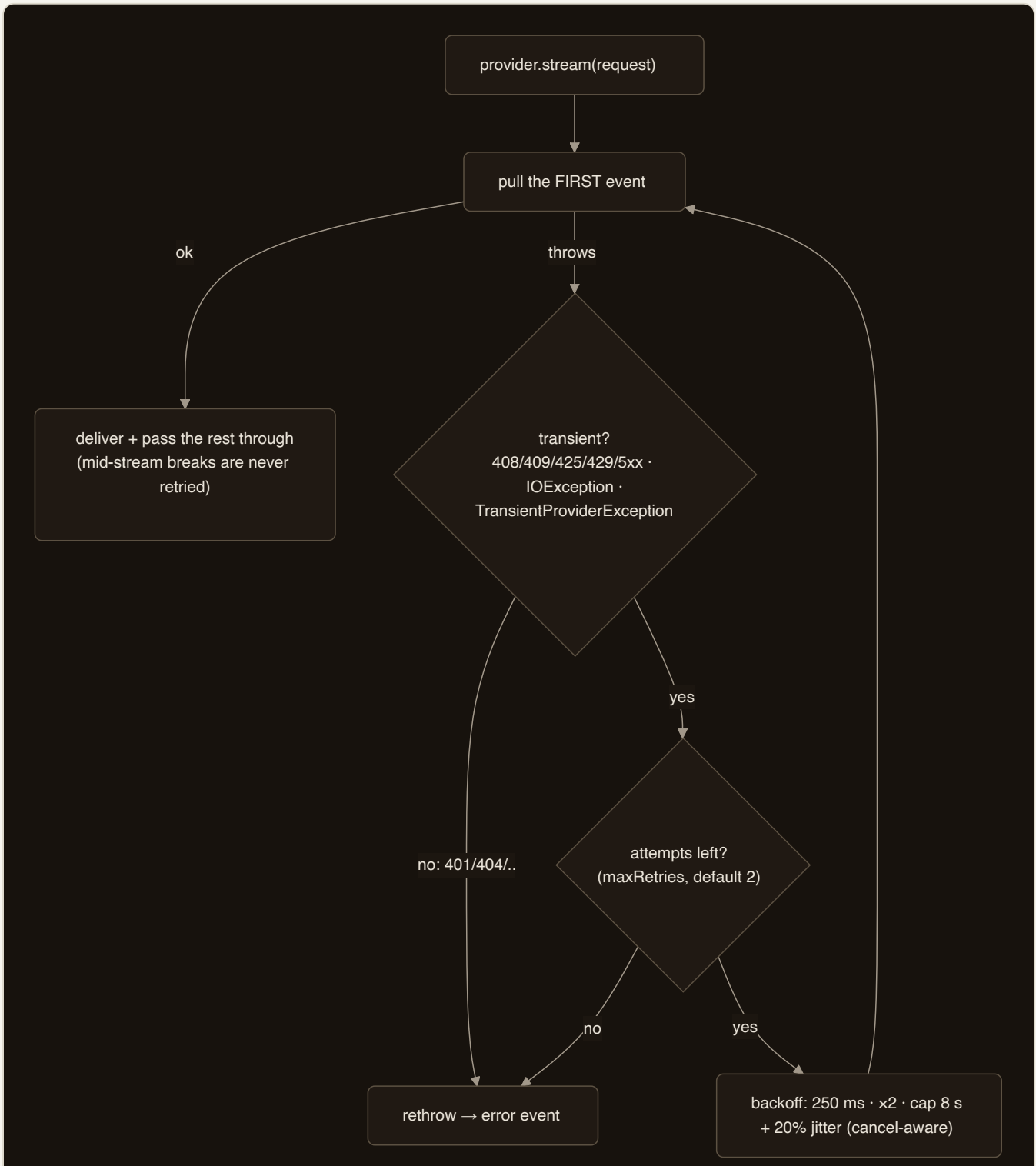
The header chip opens the picker. The model dropdown is honest: for Ollama it lists what is actually installed (live from `/api/tags`), for the cloud providers a curated set, with “Custom model …” always available; an empty list degrades to free text. Apply sends `set_provider` — and because the agent is built once per connection (it carries your history), the switch swaps a delegate *inside* the provider port instead of rebuilding anything. It takes effect on the next prompt, history intact. Selecting Anthropic without a key is refused with a readable error, and the map in the Lab re-places the LLM node inside or beyond the network boundary to match.



set_provider: validate, build the new backend (key check included), swap the delegate. The next run_start reports the new provider — no new event type needed.

Retry — a loop that doesn't tip over

Transient failures (HTTP 408/409/425/429, any 5xx, dropped connections) are retried with exponential backoff and jitter — but **only before the first event** of a response. A stream that breaks mid-answer is never retried (that would duplicate already-emitted text into the transcript); it surfaces as an honest error. Terminal errors — 404 model-not-pulled, 401 bad key — never retry. spectroscope owns retry exclusively: the Anthropic SDK's own retry is disabled so nothing double-retries. Knobs: `maxRetries` (default 2, `SPECTRO_MAX_RETRIES`; 0 disables).



The retry window: establish + first event inside the loop, transient-only, capped backoff with jitter, cancel-aware — and never after the first event.

Prompt caching (Anthropic)

With `promptCaching` on (default), spectroscope places two `cache_control` breakpoints per request: one covering the stable prefix (system prompt + tool list), one on the last *stable* message — the one just before the current turn. Costs and latency drop on every consecutive turn. The wire stays honest: the `usage` event reports the provider's raw token counts, while the cache-read/-creation counts are folded only into the *compaction trigger* — cached tokens still occupy the context window, so the threshold must see them (chapter 28).

The wrapper chain

```
Agent loop
  └─ TracingProvider      (CLI --verbose only: the wire on stderr)
      └─ SwitchableProvider (web face only: the mid-session swap point)
          └─ RetryingProvider (all faces, unless maxRetries=0)
              └─ AnthropicProvider | OllamaProvider | OpenAiCompatProvider
```

All wrapping happens at one chokepoint (`providerFromConfig()`), so the CLI, the server, headless runs and the scheduler behave identically.

CHAPTER 17

Seeing, hearing, speaking, painting

Four bonus-stage senses, all built on the same additive principles: binary data never enters the event stream, external engines sit behind injectable seams, and everything degrades readably when a piece is missing.

Vision — image input

- **Web:** attach images in the composer (picker or drag-and-drop) — they are downscaled client-side, stored content-addressed in the session's blob folder, and restored *before* the prompt text in the provider history (the position matters to the model).
- **CLI:** `spectro run -p "…" --image photo.png` (repeatable).
- **Local models fail fast:** Ollama models without the vision capability are refused with an actionable message ("use qwen3-vl or llava") before any tokens are wasted; cloud models handle images natively.
- Events carry only references (`blobPath`, `sha256`) — session files stay small and `jq`-able; a deleted blob degrades to a placeholder, never a crash.

Voice input — local push-to-talk

One-time setup: `bash scripts/setup-stt.sh` — installs whisper.cpp and ffmpeg (Homebrew on macOS) and downloads the ggml-small model (~460 MB) with a **pinned SHA-256**: a mismatch deletes the file and aborts. Recording is 16 kHz mono WAV (whisper's required format); transcription runs with automatic language detection, so German dictation is not forced through English.

- **REPL `/voice`:** record until Enter, transcribe, then an *editable confirmation* — Enter sends, typing replaces, blank discards. An STT error never reaches the agent unreviewed. The turn is audited by a `voice_input` event that never enters the provider history.
- **Web mic button:** records in the browser, transcribes via `POST /api/transcribe` (the server reuses the same Transcriber), and drops the text into the composer draft. STT not installed → the endpoint answers 503 and the button disables itself with the setup hint.

Voice output — spoken answers

Setup: `bash scripts/setup-tts.sh` (piper + the `en_US-lessac-medium` voice). Enable per run with `--speak`, at runtime with `/speak on|off`, or permanently via `"tts": { "enabled": true }` in `~/.spectro/settings.json` (the legacy `config.json` name is still read beneath it for one release — chapter 20). The renderer buffers streamed text to sentence boundaries and speaks *while the rest still streams* (synthesis runs up to three sentences ahead of strictly-ordered playback). Code blocks are never read aloud — you hear "Code block skipped." Abort stops mid-sentence with no ghost audio. Ollama + piper is a fully offline agent that talks.

Image generation

The `generate_image` tool sits behind its own provider port with two backends — Gemini (`gemini-2.5-flash-image`, `GEMINI_API_KEY`) and OpenAI (`gpt-image-1`, `OPENAI_API_KEY`) — selected by `imageProvider` / `SPECTRO_IMAGE_PROVIDER` or live from the gallery's dropdown. Results land content-addressed under `~/.spectro/images/` (same bytes = same file, shared across sessions); the additive `image_generated` event feeds the gallery panel; the model itself only sees a short confirmation string. Gated, because it is a paid cloud call. A missing key returns a readable `ERROR:` the model can relay — never a crash.

CHAPTER 18

MCP — external tool servers

spectroscope is an MCP client: plug in any Model-Context-Protocol server and its tools become first-class spectroscope tools — gated, traced, replayable. No new event type was needed: MCP calls are ordinary `tool_call` / `tool_result` lines, so graph, trace and archive render them for free.

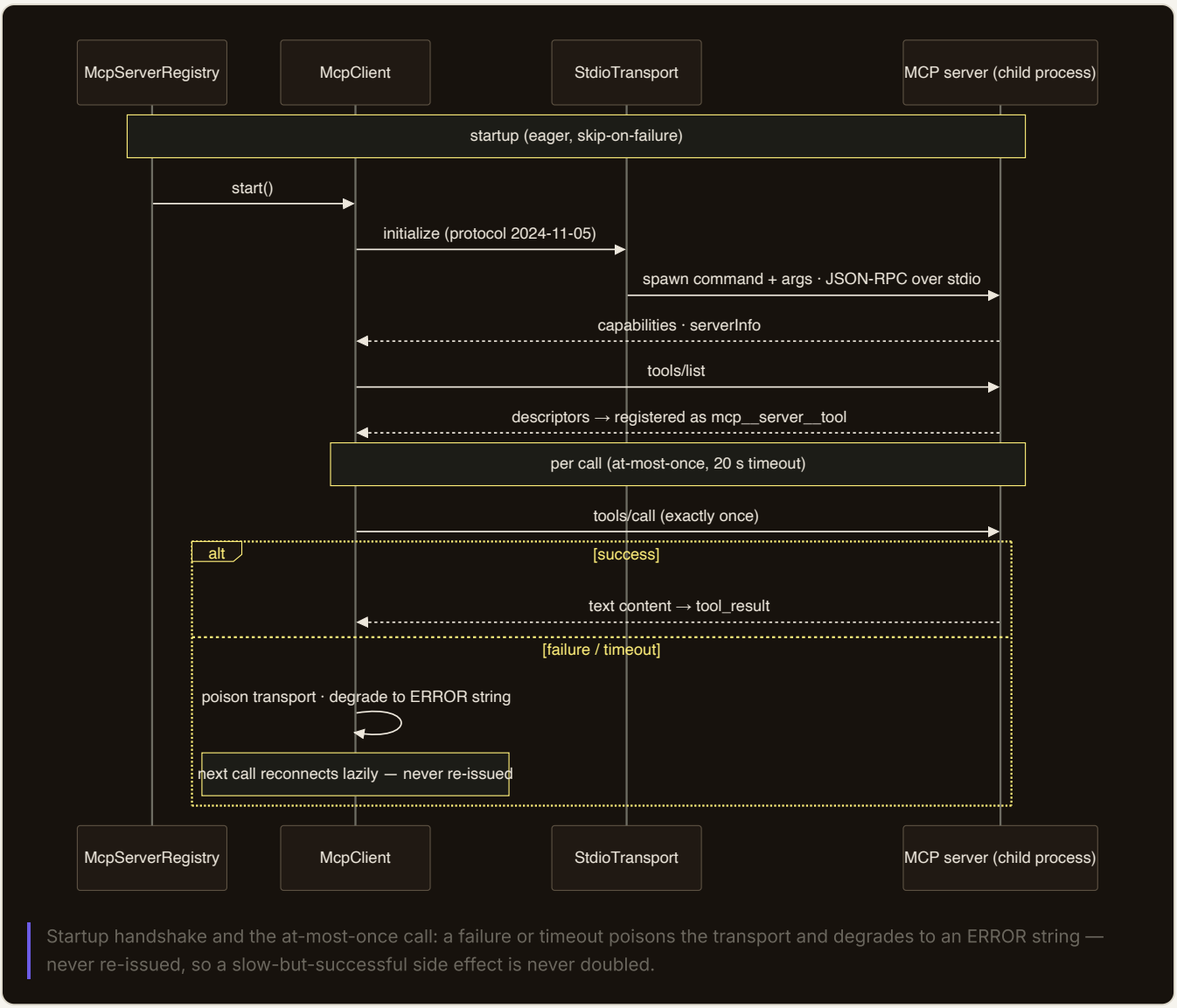
Configuration

The `mcpServers` block has the same shape Claude Desktop and Claude Code use — a config written for those drops straight in:

```
{ "mcpServers": {
    "notes": {                                  // stdio: command (+ args, env)
      "command": "/path/to/spectro-mcp-notes/bin/spectro-mcp-notes",
      "args": ["/path/to/notes-directory"]
    },
    "search": { "url": "http://localhost:9000/mcp", "type": "sse" }   // or HTTP/SSE
} }
```

Servers connect eagerly at startup; one that fails to start is logged and **skipped** — a broken entry never takes the harness down. Each advertised tool registers as `mcp__<server>__<tool>`, always permission-gated: its inputs are model output and its effects are external. `/mcp` lists servers and tools in the REPL; `spectro doctor` probes reachability; the System-context panel names the servers.

Call semantics: at-most-once



Startup handshake and the at-most-once call: a failure or timeout poisons the transport and degrades to an ERROR string — never re-issued, so a slow-but-successful side effect is never doubled.

Every call is issued *exactly once* with a 20-second budget. If the transport died between calls, it is re-established once, lazily. But a call that fails or times out on an established transport is **not retried** — a slow-but-successful `add_note` must not be doubled. Instead the transport is poisoned (the next call reconnects) and the model receives a readable `ERROR: MCP tool '…' on server '…' failed: …`. A dead or slow server degrades; it never crashes the harness.

The example server: spectro-mcp-notes

The repository ships a complete, runnable MCP server as its own module — a stdio JSON-RPC 2.0 program whose only dependency is Jackson ("no heavy MCP SDK, no Lucene"). Build it with `./spectro mcp-notes`. It serves two tools over a directory of plain text files (default `~/.spectro/notes`, six seed notes included):

- `search_notes` — hand-rolled ranked full-text search (term frequency + substring bonus) returning snippets with their source files;
- `add_note` — appends a note as a new file, with collision-safe naming (two identical texts land in two files, never an overwrite).

Try (with the server configured): "Search my notes for 'permissions' and summarize what you find." — the call passes the permission gate like any other tool, and the Lab's map lights the whole MCP chain: client → network stack → boundary → server.

CHAPTER 19

Scheduling

The scheduler turns spectroscope into an unattended worker: jobs in a JSON file, 5-field cron expressions, honest state tracking and desktop notifications.

```
# ~/.spectro/jobs.json — an array of jobs, validated loudly on load
[ { "id": "morning-report",
    "cron": "0 8 * * *",
    "prompt": "Summarize yesterday's session files under ~/.spectro/sessions.",
    "cwd": "/Users/you/projects/demo",
    "permissions": "readonly" } ]        // readonly (default) | auto
```

COMMAND	BEHAVIOUR
`spectroscope cron`	the foreground daemon — arms every job, reschedules after each firing, Ctrl+C stops
`spectroscope cron list`	each job with its expression, policy and next run time (explicit zone)
`spectroscope cron status`	list + last outcome per job: time, ok/failed/skipped, stop reason, session id, result preview
`spectroscope cron --once <id>`	run one job now; exit 0 only on `ok`

- **Overlap guard:** a job still running when its next slot fires records a `skipped (overlap)` state instead of stacking.
- **Headless policy:** jobs default to `readonly`; every run writes a normal session file (full audit) and its outcome into `~/.spectro/jobs-state.json`.
- **Notifications:** macOS notification per finished job (terminal bell elsewhere); the desktop shell (chapter 10) adds native click-to-open notifications.
- **Container-free:** cron-utils computes the times, a single ScheduledExecutorService thread fires them — no Spring scheduling in the core.

IV

PART IV

The file system

Everything spectroscope writes and reads lives in two places: a per-user home under `~/.spectro` and an optional per-project `.spectro/` folder. This part maps every file — and opens one up.

CHAPTER 20

Where everything lives

spectroscope keeps no database, no hidden caches, no binary state. Everything is plain files you can read, version, back up and delete — most of it line-oriented JSON that `jq` understands directly.

The user home: `~/.spectro`

```
~/.spectro/
├── settings.json            user settings layer (optional) — provider, model, autoApprove,
│                            mcpServers, hooks, tts … every key in chapter 30.
│                            (config.json, the legacy name, is still read beneath it
│                            for one release — `spectro doctor --migrate` renames it)
├── sessions/                the session archive — the heart of it all
│   ├── 20260716-091500-3fa4b2c1.jsonl   one session = one append-only event file
│   └── 20260716-091500-3fa4b2c1/        only when images were attached:
│       └── blobs/
│           └── ca96adb5…f90157          attachment bytes, file name = sha256
├── images/                  generated images, shared across sessions
│   └── 3f7a99c2…81d4.png       content-addressed: <sha256>.<png|jpg|webp>
├── skills/                  user-scope skills (chapter 15)
│   └── my-skill/SKILL.md
├── models/                  local speech models (voice setup scripts)
│   ├── ggml-small.bin          whisper.cpp STT model (~460 MB, SHA-pinned)
│   ├── piper/piper           TTS binary
│   └── en_US-lessac-medium.onnx TTS voice
├── notes/                   the example MCP server's note store (one .txt per note)
├── jobs.json                cron jobs — an array, hand-edited (chapter 19)
└── jobs-state.json          last outcome per job id — written by the scheduler
```

Directories appear on demand — a fresh install has none of them until the first session, the first generated image, the first setup script. On this machine, today: `sessions/` (11 files, 228 kB) and `models/` (465 MB — the whisper model dominates).

The launch-dir layer: `<launch dir>/.spectroscope`

```
<launch dir>/.spectro/
├── settings.json            the launch-dir settings layer — the directory spectroscope was
│                            STARTED in. Still read (deprecated compat layer, one
│                            release); its role as THE project file moved to the
│                            workspace's own .spectro/ pair below. The "Persist"
│                            checkbox appends here only when the session has no
│                            real workspace (throwaway temp desk).
└── skills/                  project-scope skills — win over user scope by name
    ├── brainstorming/SKILL.md
    ├── test-driven-development/SKILL.md
    ├── verification/SKILL.md
    └── writing-plans/SKILL.md
```

Two more files matter at the launch-dir root: the gitignored `./.env` (**secrets only** since the precedence flip — API keys, loaded by Gradle and the launcher, never committed; deprecated `SPECTRO_*` lines still work as the env base, but `.env.example` now documents just the keys) and an optional `SPECTRO.md`, whose content is appended to the system prompt as project context — the agent reads it before your first message.

Inside a session file

The session id doubles as the file name: `yyyyMMdd-HHmmss-<uuid8>` — sortable by start time, collision-free. One event per line, compact JSON, UTF-8, append-only: the file is never rewritten, not even by compaction. This is a real session from this machine (a thinking model answering "What is 2+2? Think first." — 43 lines, abbreviated in the middle):

```
{"type":"run_start","runId":"e93e24cf-f415-4873-a21c-cbd73fec673e","agentId":"main",
     "prompt":"What is 2+2? Think first.","provider":"ollama","ts":1783086147207}
{"type":"turn_start","agentId":"main","turn":1,"ts":1783086147208}
{"type":"thinking_delta","agentId":"main","text":"The","ts":1783086147561}
{"type":"thinking_delta","agentId":"main","text":" user","ts":1783086147561}
    … 30 more thinking deltas — the model reasons token by token …
{"type":"text_delta","agentId":"main","text":"2","ts":1783086148244}
{"type":"text_delta","agentId":"main","text":" +","ts":1783086148263}
{"type":"text_delta","agentId":"main","text":" 2","ts":1783086148282}
{"type":"text_delta","agentId":"main","text":" =","ts":1783086148301}
{"type":"text_delta","agentId":"main","text":" 4","ts":1783086148339}
{"type":"text_delta","agentId":"main","text":".","ts":1783086148358}
{"type":"usage","agentId":"main","inputTokens":297,"outputTokens":49,"ts":1783086148379}
{"type":"run_end","runId":"e93e24cf-…","stopReason":"end_turn","ts":1783086148379}
```

Everything you saw in Part II — chat, graph, trace, Lab, plan panel — is a fold over lines like these. A REPL session appends multiple runs into one file; resume appends to the same file again. Chapter 22 documents all 18 line types field by field.

Blobs and generated images

- **Attachments** (what you upload) live next to their session: `sessions/<id>/blobs/<sha256>`. Content-addressed — attaching the same image twice stores one file. Events reference `blobPath` + `sha256` only; a deleted blob degrades to a placeholder on replay, never a crash. Accepted types: jpeg, png, webp, gif.
- **Generated images** (what the agent paints) are shared across sessions in one global store: `~/.spectro/images/<sha256>.<ext>`. The web UI loads them via `GET /api/images/<file>`, where only `64-hex.png|jpg|webp` names are even considered — the content address is the traversal guard.

The workspace: where the agent actually works

The session files above are spectroscope's *records*; the **workspace** is the agent's *desk* — the directory the file tools sandbox against, glob/grep search and `run_command` runs in. Unless a `workspace` is configured (chapter 30), every session gets its own folder under the OS temp dir:

```
<tmpdir>/spectroscope-ws/
├── 20260716-204301-db5f3af7/   one folder per session, keyed by the SESSION id
│   └── add.py                     ← what "write me a script" produces lands HERE,
└── 20260716-204419-39b41a29/      not in the repo you started spectroscope in
```

The key is the session id, so a **resume finds its files again**; a New chat means a fresh desk. The web server announces the folder over the socket (`workspace_info` — a UI-only frame, never in the JSONL) and the Files tab follows it (chapter 6). The launch dir stays anchored: skills, MCP servers, SPECTRO.md and its own (deprecated) settings layer still load from the directory spectroscope was started in — while the workspace contributes its own settings pair on top (below).

Three ways to place the desk, highest wins: **pick it per session** in the web UI (the Files tab's "Choose folder …" button opens the native macOS dialog on the spectroscope machine; only before the first run, pinned in server memory for the session), **configure it** (`workspace` key / `SPECTRO_WORKSPACE` / `--workspace` — chapter 30), or **let spectroscope mint** the per-session temp folder above.

A *real* workspace (picked or configured — not the throwaway temp desk) carries its own settings pair, the top file layers of the hierarchy (chapter 30) and the files the composer gear and the permission dialog's "Persist" checkbox write:

```
<workspace>/.spectro/
├── settings.json            the workspace PROJECT file — travels with that repo.
│                            Team conventions become checked-in settings:
│                            permissionMode, autoApprove, hooks, mcpServers …
│                            Persisted "always allow" rules land here.
├── settings.local.json      machine-local overrides for THIS workspace — absolute
│                            paths, personal rules. Never committed:
└── .gitignore               spectroscope writes it on the first local save; it lists
                             settings.local.json so the local file stays out of git
```

This pair is not just read, it is **executed**: `mcpServers` spawns processes, `hooks` runs shell commands around every tool call, and a permissive `autoApprove` / `permissionMode` can auto-allow gates that would otherwise ask (chapter 30). Pinning a cloned or otherwise foreign folder as your workspace means reviewing its `.spectro/` first — the same way you would review its build scripts before running them.

Lifecycle: create, resume, delete

- **Created** on the first prompt of a connection (web) or at REPL start; headless and cron runs write the same files.
- **Resumed** by `spectroscope --resume <id>` or the web Resume button — the history is reconstructed from the file, new events append.
- **Deleted** only by the guarded two-step (chapter 8): `SessionStore.deleteSession` removes the JSONL and the blob folder — and only for ids resolving to a direct child of the sessions directory. Nothing else in spectroscope ever deletes or rewrites a session line.
- **Crash-safe by construction:** one `write` per event, no open handle, no save step. A torn last line after a crash is silently discarded on read; everything that streamed is on disk.

The jq cookbook

Because the format keeps its rules (one object per line, no binary, camelCase), the shell is a first-class session browser:

```
# watch a session live
tail -f ~/.spectro/sessions/<id>.jsonl | jq -c '{type, agentId, name}'

# tool timings
jq -r 'select(.type=="tool_result") | "\(.durationMs)ms\t\(.callId)"' <id>.jsonl

# token totals
jq -s '[.[] | select(.type=="usage")]
       | {in: map(.inputTokens)|add, out: map(.outputTokens)|add}' <id>.jsonl

# the agent tree
jq -r 'select(.type=="agent_spawn") | "\(.parentId) → \(.agentId): \(.task)"' <id>.jsonl
```

PART V

Under the hood

The complete technical reference: the generated architecture dossier, every event on the wire, the provider port, the WebSocket and REST protocols, the loop internals, context management, MCP internals, every configuration key, and the build itself.

CHAPTER 21

The architecture dossier

Fourteen generated diagrams cover the full build — every class name, wire string, endpoint and count in them was read from this repository, and each comes from a rerunnable Python generator (`docs/diagrams/build_NN_*.py`): change the system, rerun the script, the diagram is current again. The shared visual language: ocean outlines the core, lilac the faces, salmon the model side and external services, moss the disk; sand carries the event wire and stats; coral appears only where a human decides, a run is cancelled or the network boundary is crossed.

The big picture

Chapter 1 already showed diagram 00 (one core, five faces). The build view below adds the honest module structure: four JVM modules, two JS toolchains, and the one deliberate cross-dependency — the server reuses the CLI's Transcriber for `/api/transcribe`.

Diagram 01 — the Gradle build: module dependency edges, version catalog, and the two npm toolchains that live next to the JVM graph on purpose.

The protocol layer

SPECTROSCOPE · ARCHITECTURE DOSSIER · 02

The event protocol.

One sealed interface, 18 records, Jackson-typed by a snake_case "type" property. The same line is the socket frame, the JSONL storage row and the graph input.

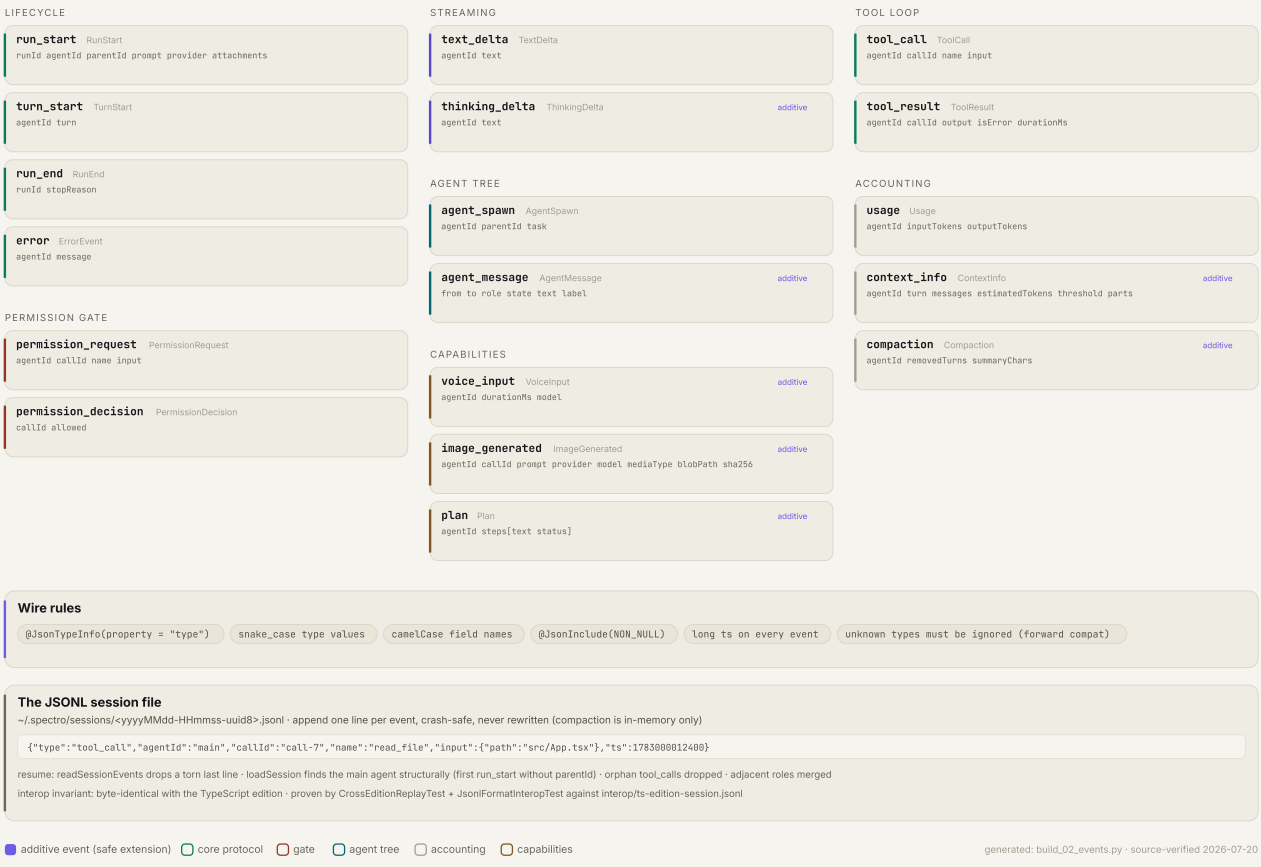


Diagram 02 — the RunEvent protocol: all 18 record types with exact wire strings, additive events marked, the wire rules, and the anatomy of a session file including the cross-edition invariant.

The protocols.

Five wire shapes carry one event model. SSE lives only between the cloud LLM APIs and the provider adapters; inside the harness everything is an in-process stream, the browser gets WebSocket frames, the disk gets JSONL, and agents talk to each other through plain events.

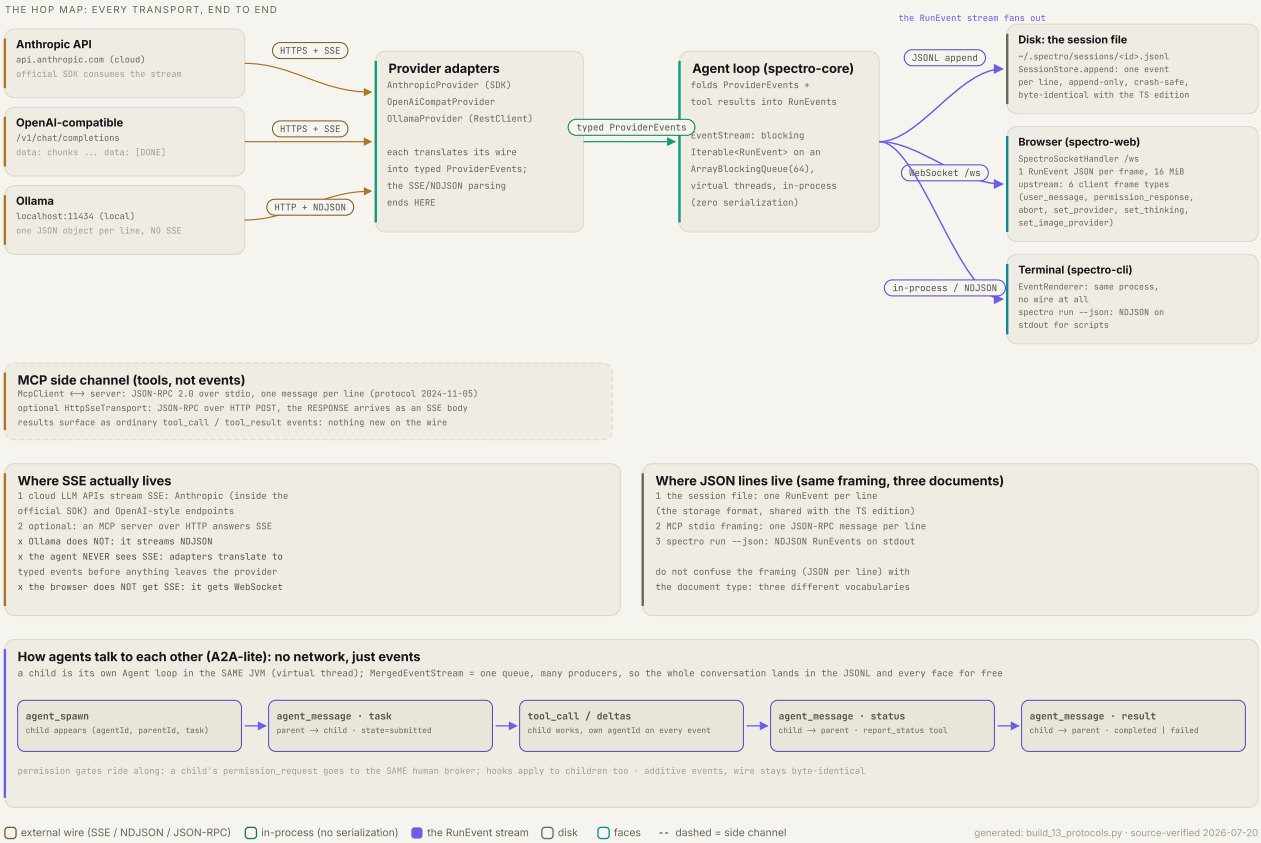


Diagram 13 — protocol breakdown, hop by hop: SSE lives only between the cloud APIs and their adapters (and the optional MCP-HTTP transport); Ollama streams NDJSON; in-process it is the blocking EventStream; WebSocket to the browser; JSONL on disk; JSON-RPC/stdio to MCP servers; A2A between agents is events on the merged stream — no network.

The model side

SPECTROSCOPE · ARCHITECTURE DOSSIER · 03

The provider port.

Everything the loop needs from a model sits behind LlmProvider. Wrappers add runtime switching and retries without the loop noticing; the config layers decide which backend is real.

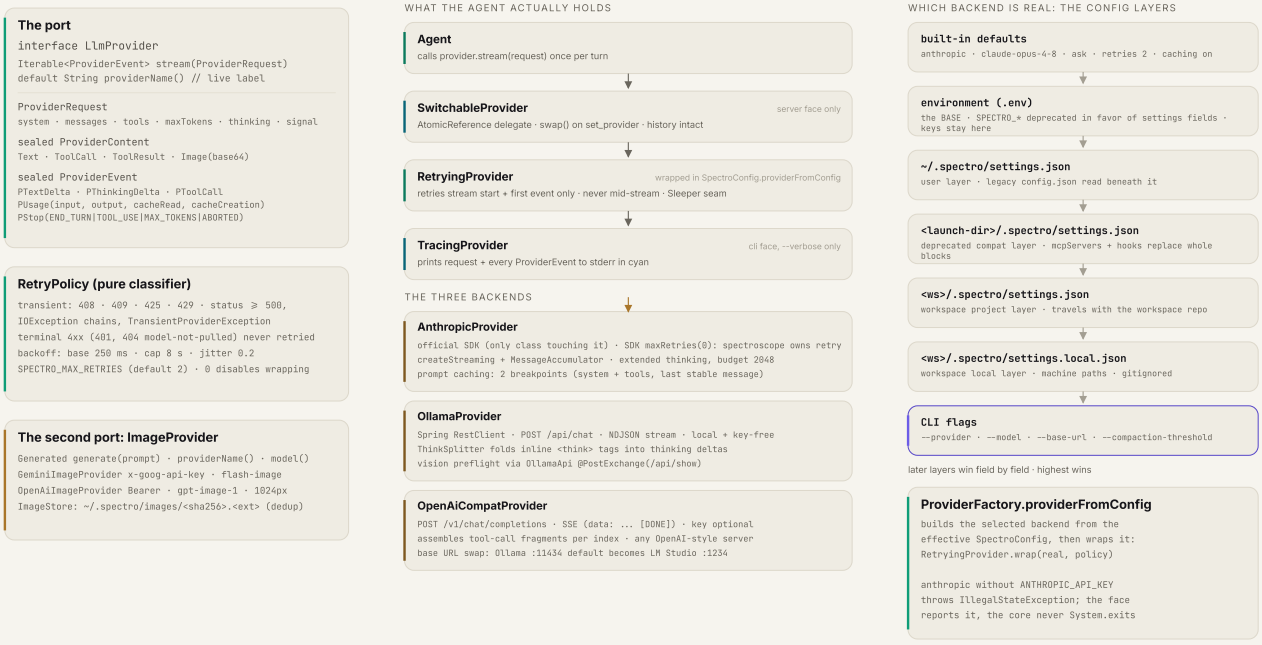


Diagram 03 — the LlmProvider port: the sealed request/event vocabulary, the wrapper chain (Switchable → Retrying → backend), the three backends with their transports, the retry policy table, the ImageProvider port and the config precedence.

The loop and the belt

SPECTROSCOPE · ARCHITECTURE DOSSIER · 04

One turn through the loop.

The agent streams a model turn, guards every tool call through hooks, allowlist and the human gate, feeds the result back, and repeats until the model stops calling tools.

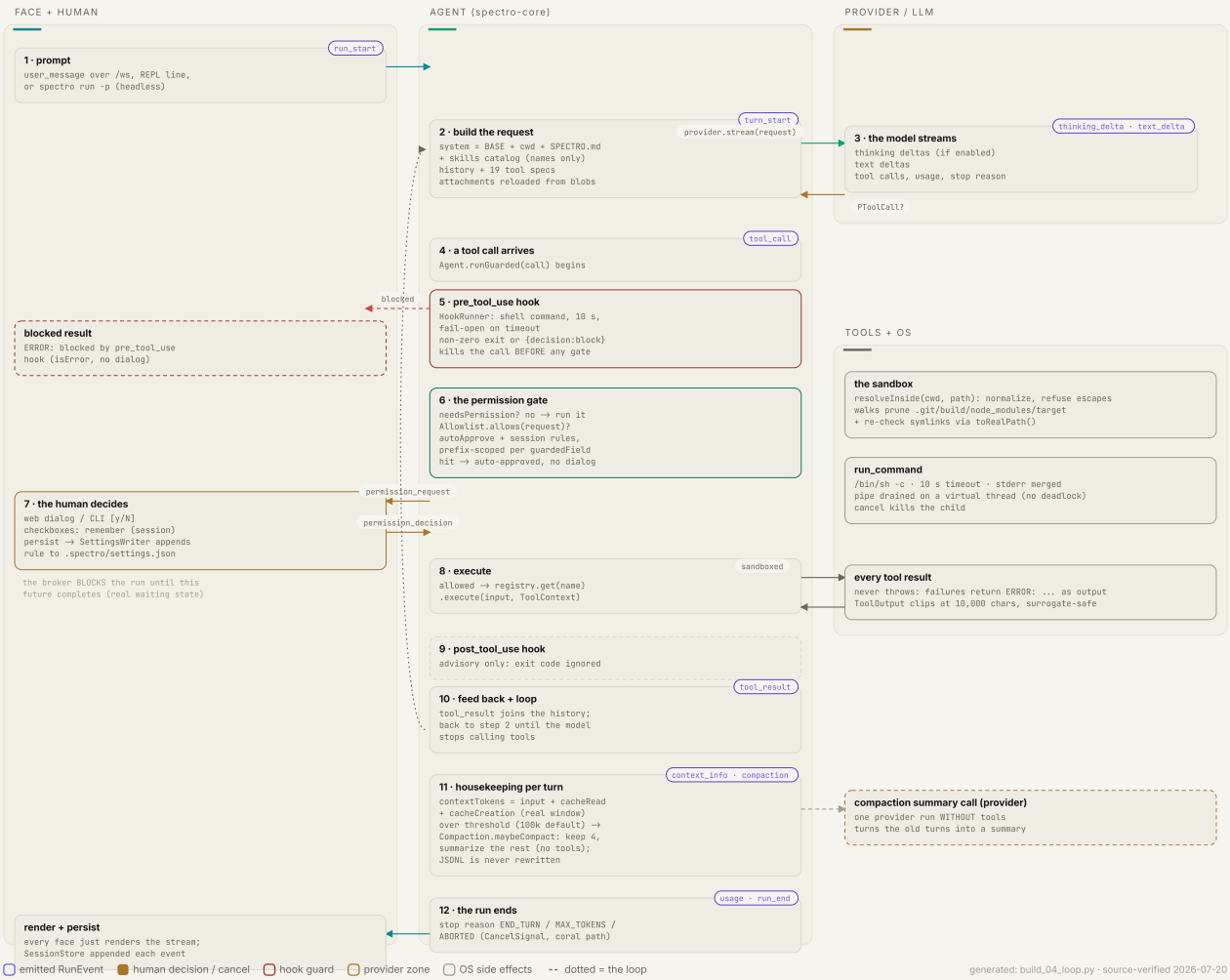


Diagram 04 — one turn as a swimlane: request build, streaming, the pre_tool_use hook, allowlist and human gate, sandboxed execution, the feedback loop into the next turn, compaction and run end.

SPECTROSCOPE · ARCHITECTURE DOSSIER · 05

The tool belt.

19 tools behind one registry and one gate. Read-only tools run free; anything that writes, executes, spends money or leaves the machine asks first. Course maxim: tool inputs are model output, and therefore untrusted.

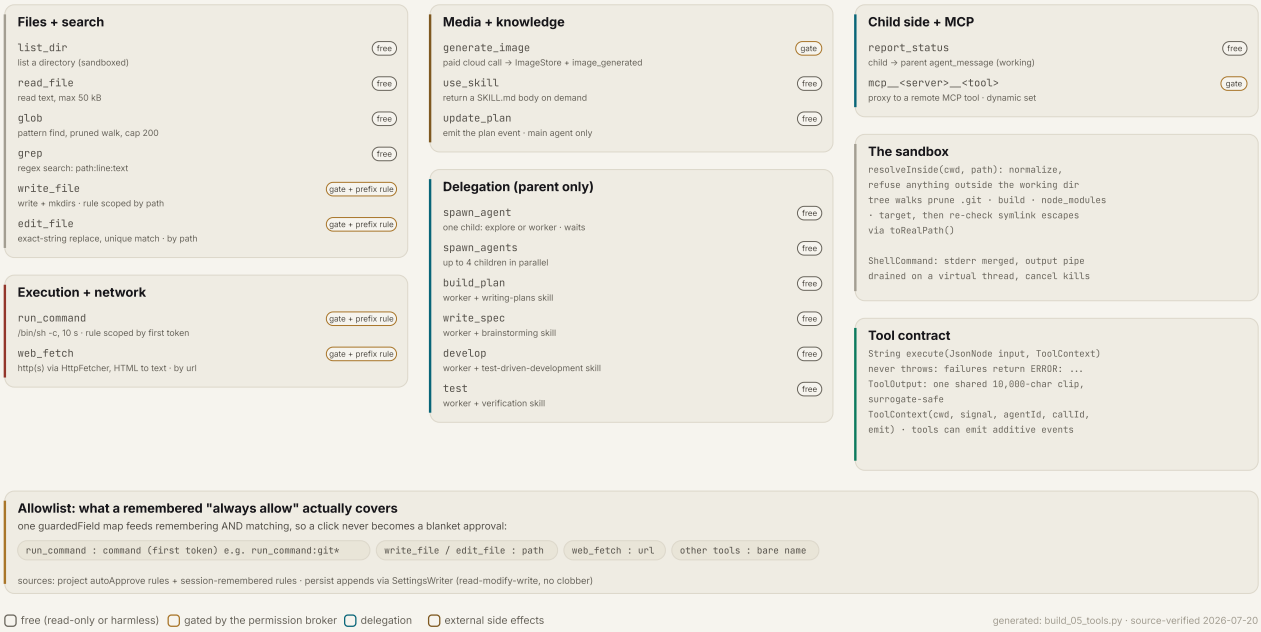


Diagram 05 — the tool belt: every tool grouped by family with its gate class (free / gated / gated + prefix rule), the sandbox rules, the tool contract, and the allowlist's guardedField mapping.

Delegation

SPECTROSCOPE · ARCHITECTURE DOSSIER · 06

Delegation: agents spawning agents.

The parent agent spawns typed children through ordinary tools. Every child is its own loop with its own toolset and CancelSignal; an additive agent_message lifecycle makes the protocol between them visible.

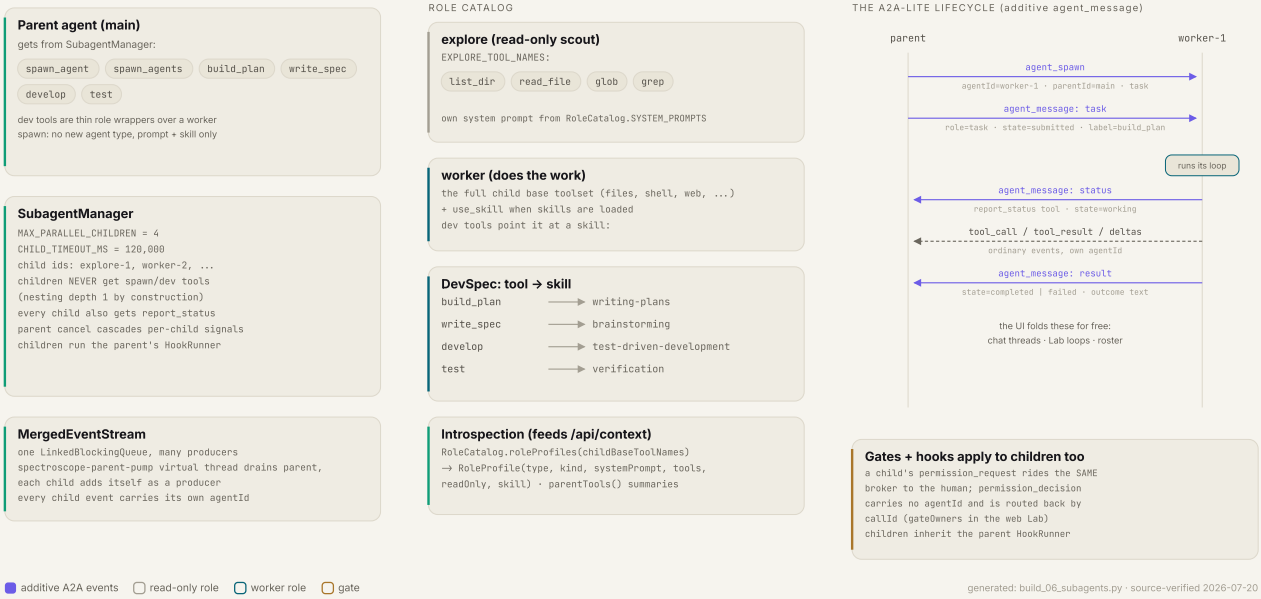


Diagram 06 — subagents: the manager's limits, the explore/worker role profiles, the dev-tool-to-skill table, and the A2A-lite lifecycle as a parent/child sequence.

The web face

SPECTROSCOPE · ARCHITECTURE DOSSIER · 07

The web backend.

Spring Boot on 127.0.0.1:8080. One TextWebSocketHandler speaks the RunEvent wire; a handful of read-only REST endpoints serve replay, context, models and the workspace. One jar serves the built UI.

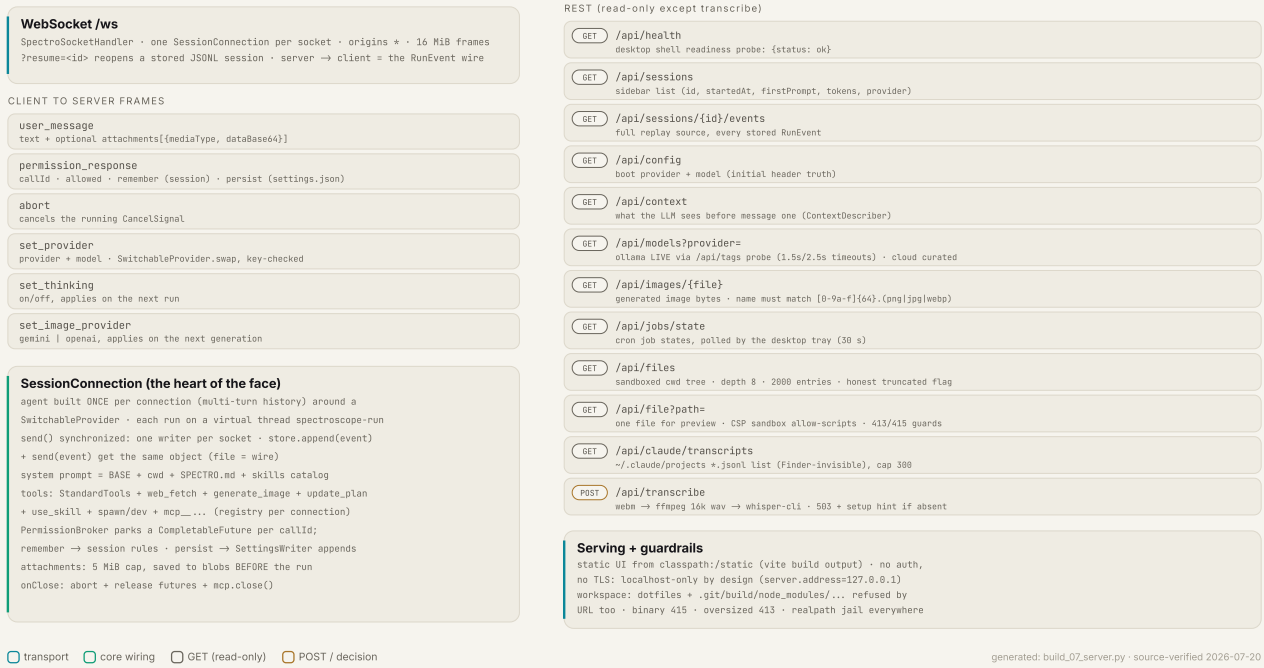


Diagram 07 — spectro-server: the WebSocket frame vocabulary in both directions, all REST endpoints with their guards, and SessionConnection's responsibilities.

SPECTROSCOPE · ARCHITECTURE DOSSIER · 08

The browser face.

Events arrive on one socket, are batched per animation frame and folded by one pure reducer. Every tab is a different reader of the same UiState, so live runs, archives, imports and scenarios all render identically.

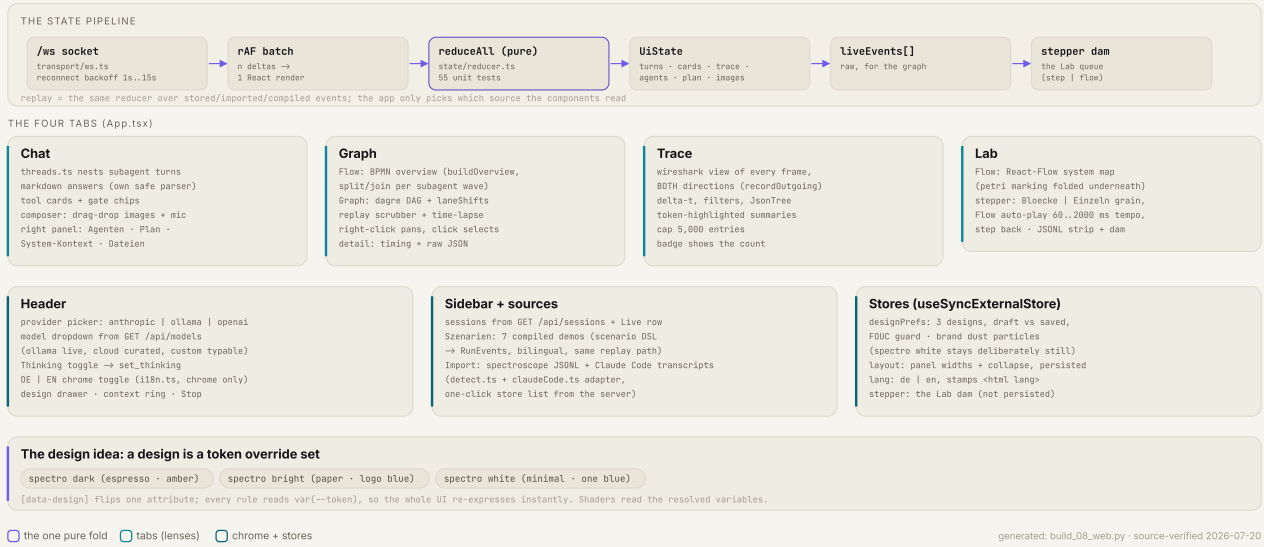


Diagram 08 — spectro-web: the rAF-batched pure-reducer pipeline, the tabs as lenses over one UiState, the header and sidebar anatomy, the persistence stores and the design system.

Runtime and integration

SPECTROSCOPE · ARCHITECTURE DOSSIER · 09

Start it: launcher, desktop, doctor.

One executable per checkout fixes the two host problems (JDK version, .env) before any command runs. The desktop face is a supervisor: Electron manages the Spring Boot JVM as a child process.

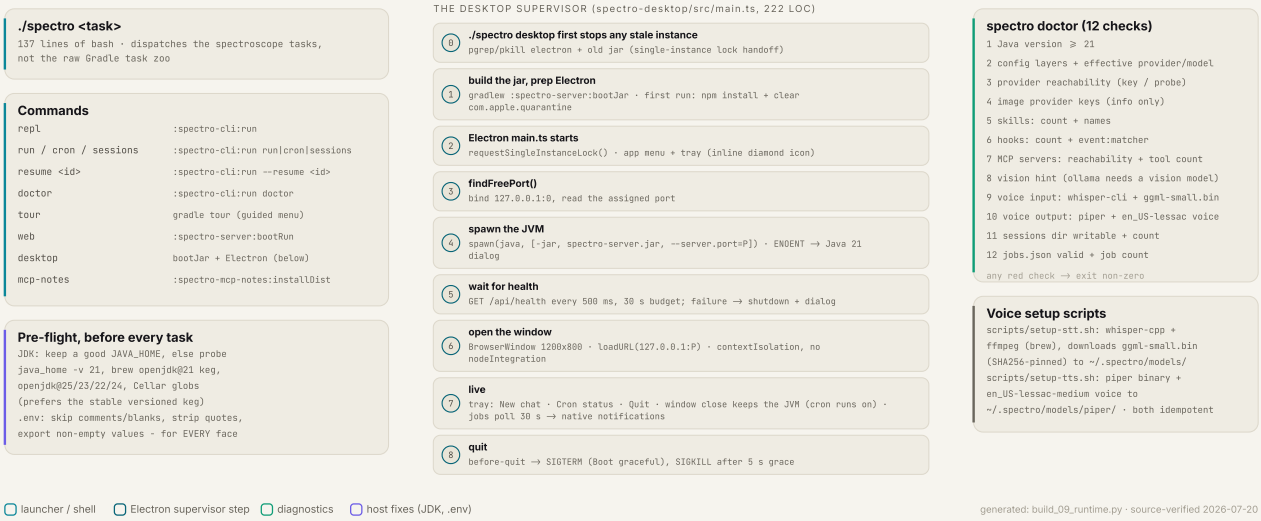


Diagram 09 — runtime: the ./spectro command table and its pre-flight (JDK resolution, .env), the nine-step Electron supervision sequence, the twelve doctor checks and the voice setup scripts.

SPECTROSCOPE · ARCHITECTURE DOSSIER · 10

MCP: external tools, no new events.

A container-free MCP client wraps every remote tool as an ordinary spectroscope Tool. Calls surface as plain tool_call / tool_result, so graph, trace and JSONL show them for free and the wire stays byte-identical.

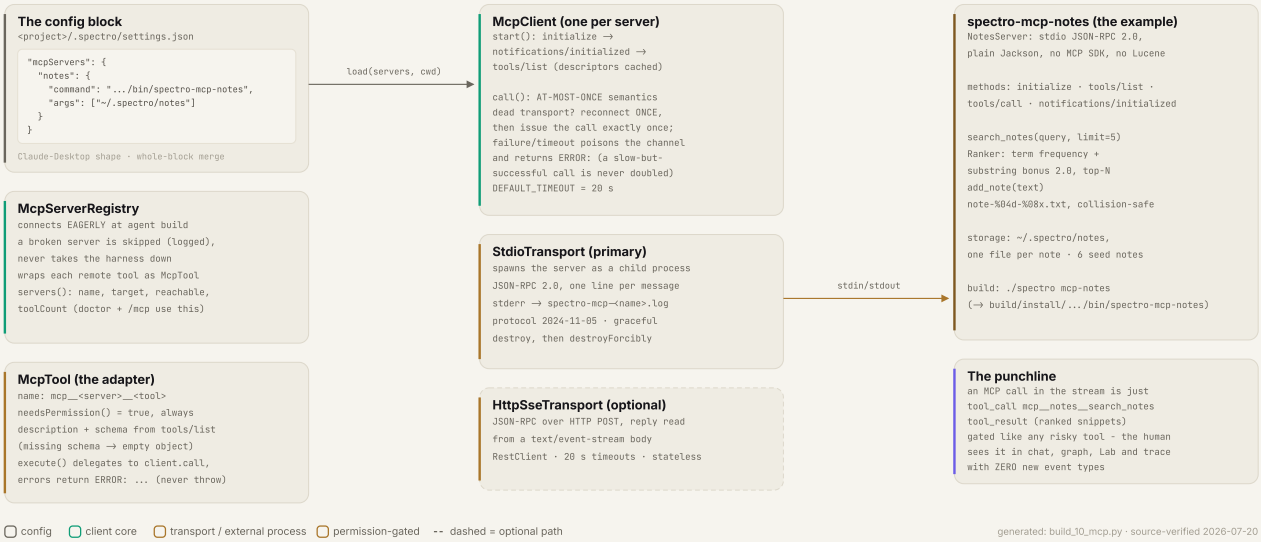


Diagram 10 — MCP: the config block, the registry's eager-connect/skip-on-failure policy, the at-most-once client, both transports, the McpTool adapter and the spectro-mcp-notes example server.

The codebase itself

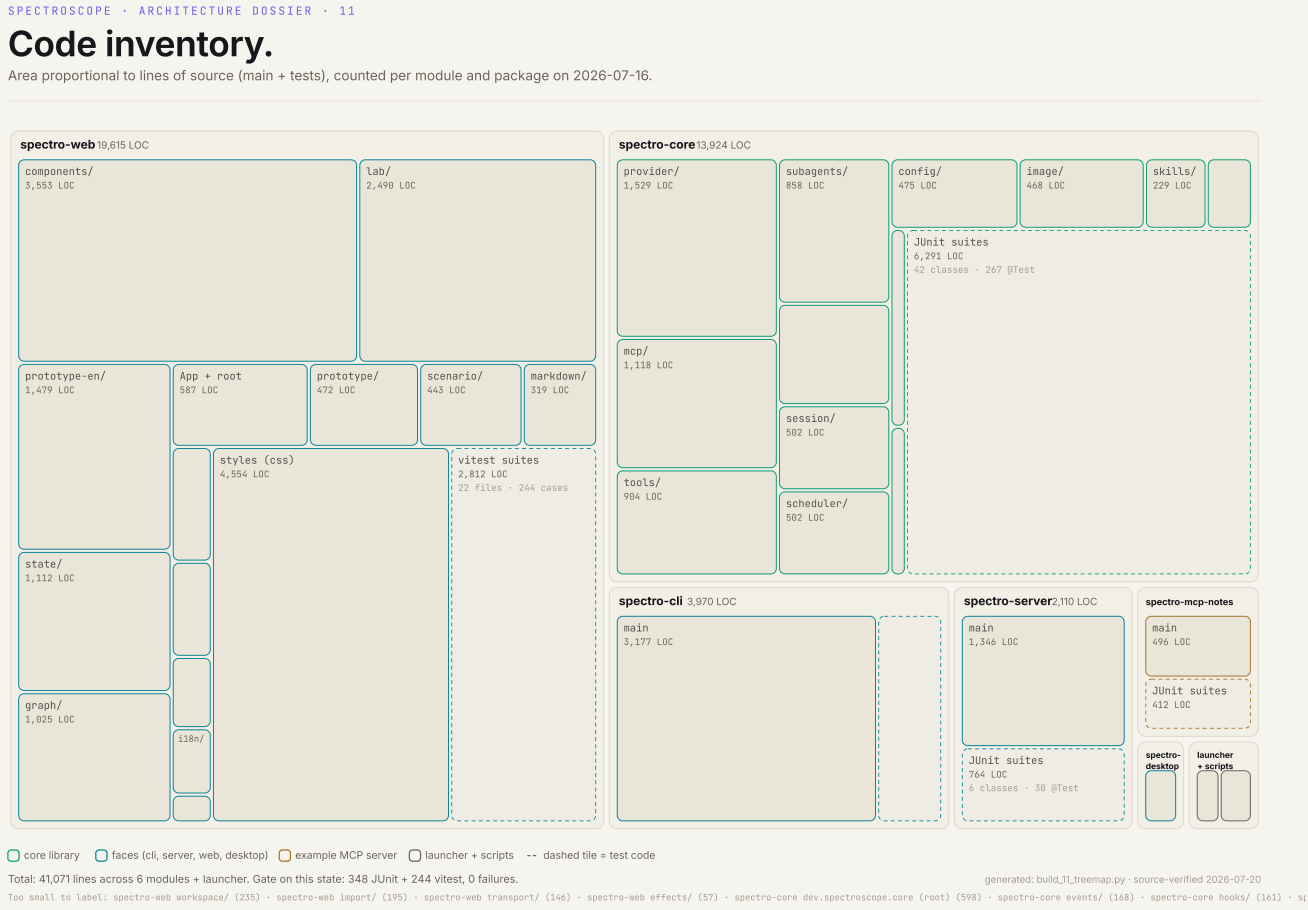


Diagram 11 — the code inventory as a treemap: area proportional to lines of source per module and package, test code as dashed tiles, measured counts in the footer.

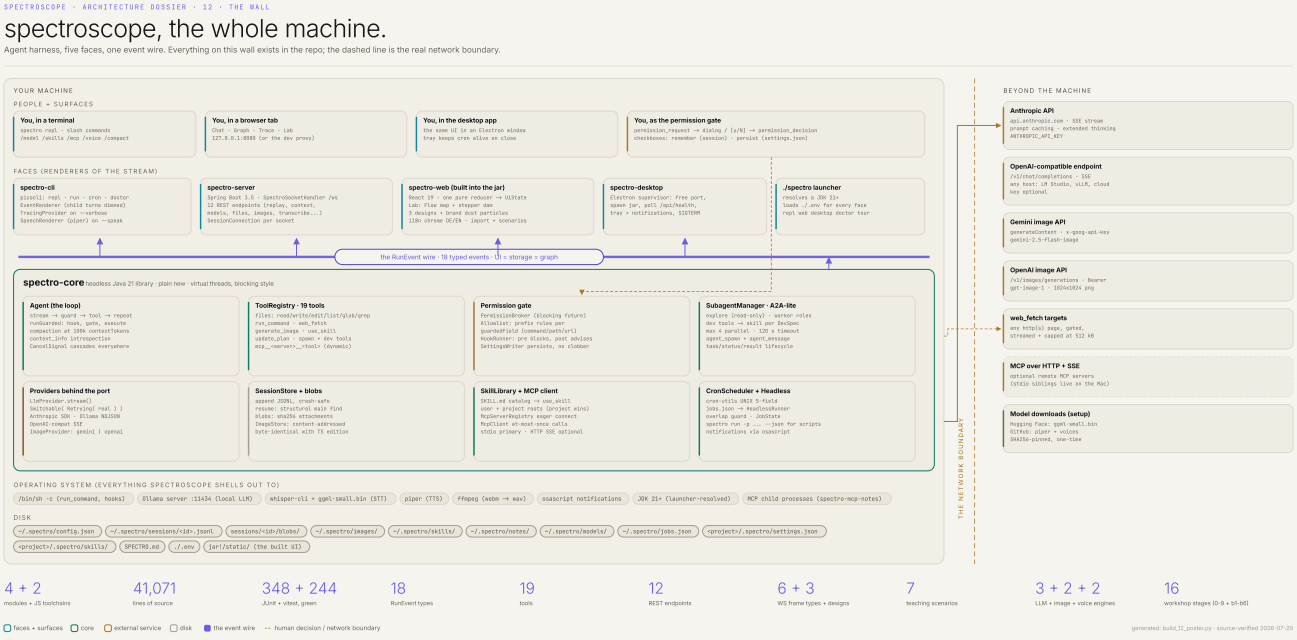


Diagram 12 — the wall poster: people, faces, core, operating system, disk, and everything beyond the network boundary in one picture.

CHAPTER 22

The RunEvent protocol

Eighteen event types, one sealed Java interface, one JSON line each. This chapter is the complete wire reference — every field, every optionality, every rule. The format is binding and shared byte-for-byte with the TypeScript edition (`docs/concept/JSONL-FORMAT.md` is the contract document).

The wire rules

1. **snake_case types, camelCase fields.** `"type":"tool_call"` carries `callId`, never `call_id`.
2. **One line = one event**, compact JSON, UTF-8, `\n`-terminated. No arrays, no comments, no blank lines.
3. **Append-only.** A written line is never modified, reordered or deleted — compaction included.
4. **Additive evolution only.** New types and new optional fields may appear; renaming, removing or reinterpreting is forbidden. Old files replay forever.
5. **Tolerant consumers.** Unknown types and fields are skipped, never errors; an unparsable (torn) trailing line is silently discarded.
6. **Null optionals are absent.** Optional fields are omitted from the JSON entirely (`@JsonInclude(NON_NULL)`), matching the TS edition.
7. **Structured tool input.** `input` travels as a JSON object, never as a serialized string — consumers parse trees, never string-match JSON.
8. **Only data.** No binary, no object references, no cycles; every line round-trips losslessly.

IDS — `sessionId` = file name (`yyyyMMdd-HHmmss-<uuid8>`) · `runId` = UUID per run · `agentId` = `"main"` or `explore-1` / `worker-2` … · `callId` links tool_call ↔ permission_request/decision ↔ tool_result.

TIMESTAMPS — `ts` = epoch milliseconds, monotone within a file (write order).

The catalog

`run_start` — since stage 3

FIELD	TYPE	
`runId`	string	UUID per run
`agentId`	string	`"main"` or child id
`parentId`	string?	only on child runs — the structural marker of a subagent
`prompt`	string	the user/task text
`provider`	string? *additive*	kept live-accurate across mid-session switches
`attachments`	Attachment[]? *additive*	image references: `{kind, mediaType, blobPath, sha256}` — bytes never enter the event

`turn_start` — since stage 3

`agentId`, `turn` (1-based; the loop caps at 15). On resume a turn boundary flushes the reconstruction buffer.

`text_delta` — since stage 3

`agentId`, `text` — one streamed chunk of the answer. Consecutive deltas concatenate into the assistant text on resume.

`thinking_delta` — additive · stage 7+

`agentId`, `text` — one chunk of the model's reasoning. A sibling of `text_delta` but a separate stream: rendered apart, and it **never re-enters the provider history** (the resume fold ignores it).

`tool_call` — since stage 3

`agentId`, `callId`, `name`, `input` (JSON object). MCP calls (`mcp__server__tool`) and skill uses are ordinary tool calls — no dedicated types exist.

`permission_request` — since stage 3

`agentId`, `callId`, `name`, `input` — emitted when a gated tool is about to run in ask mode; same `callId` as the pending call.

`permission_decision` — since stage 3

`callId`, `allowed` (boolean). One of only two types **without an** `agentId` — consumers route it back via `callId`. Emitted for every decision, allowlist auto-approvals included (the audit trail).

`tool_result` — since stage 3

`agentId`, `callId`, `output` (always a string, clipped surrogate-safely), `isError` (true iff the output starts `ERROR:`), `durationMs`.

`agent_spawn` — additive · stage 5

`agentId` (the *child*), `parentId`, `task`. The tree edge — the whole agent hierarchy lives in these two id fields.

`compaction` — additive · stage 4

`agentId`, `removedTurns`, `summaryChars`. An audit line only: compaction replaces the *in-memory* history — the file is never rewritten.

`voice_input`

additive · bonus 2

`agentId`, `durationMs`, `model` (e.g. `"ggml-small"`). Written *before* the `run_start` of a spoken turn; pure provenance — a resumed voice turn is byte-identical to a typed one.

`usage`

since stage 3

`agentId`, `inputTokens`, `outputTokens` — per agent, raw provider counts. The wire never includes cache arithmetic (that lives only in the compaction trigger) — the token truth.

`run_end`

since stage 3

`runId`, `stopReason` — no `agentId`. Stop reasons: `end_turn` (regular), `max_tokens`, `tool_use` (edge case), `aborted` (cancel/Ctrl+C), `max_turns` (the 15-turn brake), `error` (preceded by an `error` event).

`error`

since stage 3

`agentId?`, `message` — the failure channel, followed by `run_end {stopReason:"error"}`. Also the web server's own error path (a refused provider switch, a too-large attachment) — always a first-class event, never a separate frame type.

`image_generated`

additive · bonus 4

`agentId`, `callId`, `prompt`, `provider`, `model`, `mediaType`, `blobPath` (relative to `~/.spectro`, e.g. `images/3f7a….png`), `sha256`. The event carries the reference; the bytes live in the content-addressed store.

`context_info`

additive · full build, opt-in

`agentId`, `turn`, `messages`, `estimatedTokens`, `threshold`, `parts[]` (`{label, chars, estTokens}` — system prompt / tool schemas / conversation). Emitted once per turn when introspection is on; sizes are chars/4 *estimates* — `usage` stays the truth. Drives the context ring.

`agent_message`

additive · full build (A2A-lite)

`from`, `to`, `role`, `state`, `text`, `label?` — note the addressing is from/to, not agentId. Roles: `task` (parent→child, state `submitted`), `status` (child→parent, `working`, fed by `report_status`), `result` (child→parent, `completed` / `failed`). `label` names the dev tool (`build_plan`…) and is absent on plain spawns.

`plan`

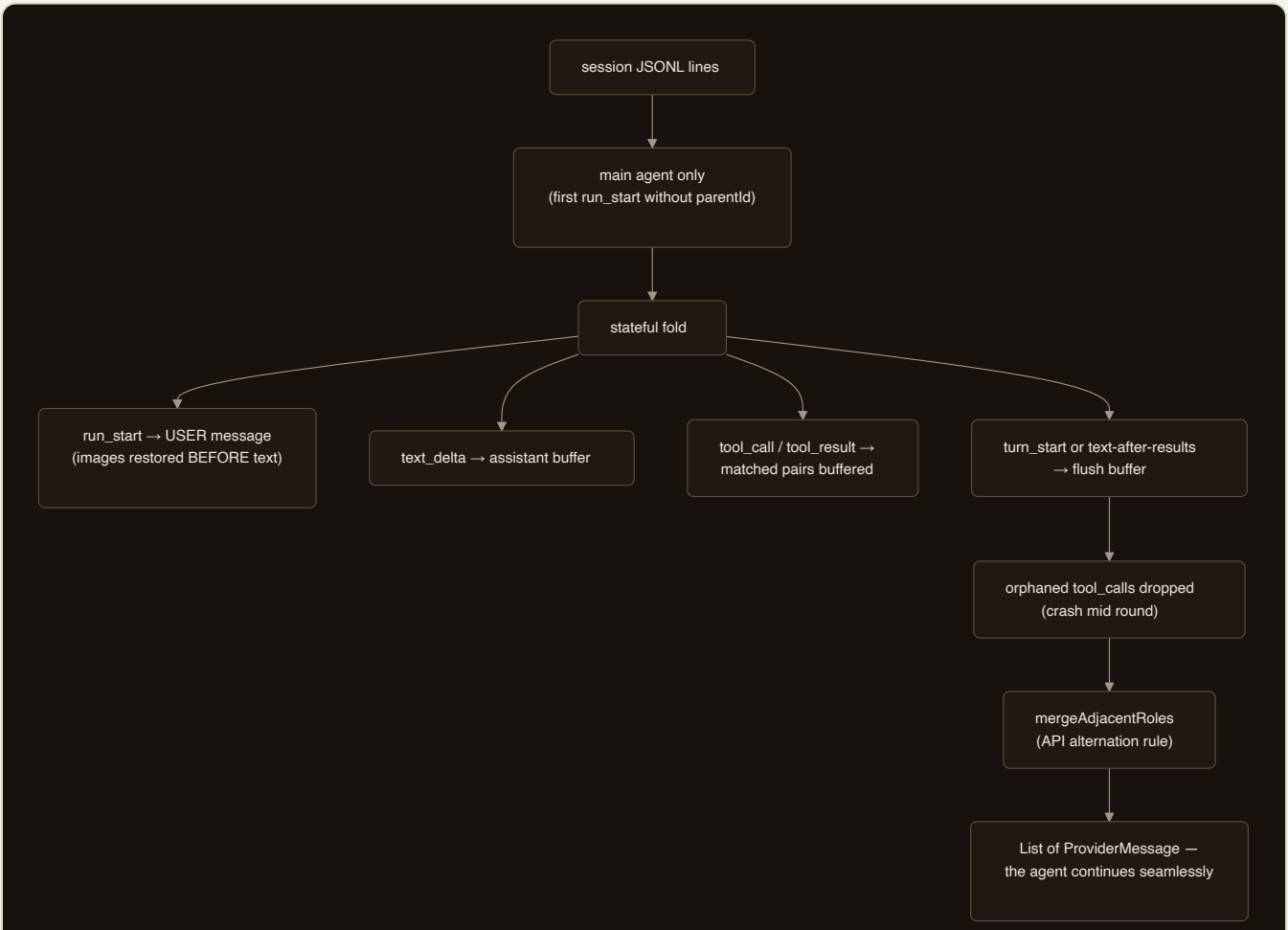
additive · full build

`agentId`, `steps[]` (`{text, status}`). Latest-wins full replacement from `update_plan`; statuses are the canonical English enum `pending / in_progress / completed`, enforced at the write boundary.

The cross-edition proof

The invariant is tested, not asserted: `CrossEditionReplayTest` replays the TypeScript edition's canonical example session verbatim — every line parses into a typed event, **re-serialization reproduces the other edition's bytes line for line**, and the resume fold reconstructs a usable history even though that file uses compact ids (`a0`) instead of `"main"`. Two interop hardenings came out of that work and are now permanent behaviour: the main agent is found *structurally* (first `run_start` without `parentId`), and text arriving after buffered tool results flushes the reconstruction even when a foreign file omits `turn_start`.

Resume: lines become a conversation



The resume fold: four line types contribute (run_start, text_delta, tool_call, tool_result); everything else is display material. Orphaned calls are dropped, roles merged.

Only four line types rebuild the provider history — everything else (`usage`, permissions, `thinking_delta`, `compaction`, `voice_input`…) is display and audit material. Child events never leak into the history: their results are already inside the parent's `tool_result`. Orphaned tool calls (a crash mid-round) are dropped because the provider API would reject the history; attachment images are restored *before* their prompt text, at their original position.

CHAPTER 23

Streams & cancellation

The Java-21 answer to the TypeScript edition's async generators: a blocking `Iterable` over a bounded queue, produced by a virtual thread, plus a cooperative cancel signal. No reactive framework anywhere.

EventStream

```
public interface EventStream extends Iterable<RunEvent>, AutoCloseable {
    void cancel();
    @Override void close();   // idempotent; also cancels
    static EventStream start(CancelSignal signal, Consumer<Consumer<RunEvent>> body) { … }
}
```

- **Backpressure for free:** a bounded queue (64) — a slow renderer blocks the producer instead of ballooning a buffer.
- **The producer** runs on a virtual thread named `spectroscope-agent`; a `finally` always enqueues the end sentinel, so a consumer's for-each can never hang, even when the body throws.
- **The sentinel** is a private instance recognized by reference identity — never handed out, never serialized.
- `close()` also interrupts a producer parked on a full queue — try-with-resources cleans up even when a consumer bails early.

MergedEventStream

Subagents write into a shared merge: one unbounded queue, many producers (the parent pump plus one forwarder per child), exactly one consumer. Unbounded on purpose — child producers must never block behind a slow renderer. This is why A2A needs no network: the “protocol between agents” is just events interleaving on the one stream that every renderer and the session store already consume.

CancelSignal

```
public final class CancelSignal {
    public synchronized void cancel();                  // idempotent, runs listeners
    public boolean isCancelled();
    public synchronized Runnable onCancel(Runnable l);  // returns a deregistration handle
}
```

The loop checks it at safe points; providers close their HTTP streams on it; running shell commands are force-killed by a per-call listener that deregisters itself afterwards (the handle return — a run's listener list must not grow per tool call). `onCancel` fires immediately if already cancelled, closing the classic registration race at spawn time. Children get their own signal, cascaded from the parent's — one Ctrl+C ends the whole tree with `run_end {stopReason:"aborted"}`, not a stack trace.

CHAPTER 24

The provider wire

Between the agent loop and any model sits one sealed vocabulary. The loop consumes five event kinds and produces one request record — everything provider-specific stays inside the adapters.

The port

```
public interface LlmProvider {
    Iterable<ProviderEvent> stream(ProviderRequest request);   // blocking, lazy
    default String providerName() { return null; }             // live label for run_start
}

record ProviderRequest(String system, List<ProviderMessage> messages,
                       List<ToolSpec> tools, int maxTokens, boolean thinking,
                       CancelSignal signal)

record ProviderMessage(Role role /* USER | ASSISTANT */, List<ProviderContent> content)

sealed interface ProviderContent
        permits TextContent, ToolCallContent, ToolResultContent, ImageContent

sealed interface ProviderEvent permits PTextDelta, PThinkingDelta, PToolCall, PUsage, PStop
record PUsage(int inputTokens, int outputTokens,
              int cacheReadTokens, int cacheCreationTokens)
record PStop(StopReason reason /* END_TURN | TOOL_USE | MAX_TOKENS | ABORTED */)
```

Contract points: all three backends return *lazy* iterables (tokens reach the UI while the HTTP stream is still open); `PUsage.inputTokens` stays the provider's raw count (it feeds the wire `usage` event) while the two cache fields ride separately for the compaction trigger only; a cancel mid-stream surfaces as `PStop(ABORTED)`, never as an exception.

Backend mappings

Anthropic (SSE via the official SDK)

- SDK client built with `maxRetries(0)` — spectroscope owns retry.
- Deltas translate one-to-one; **usage and fully-parsed tool inputs exist only in the accumulated final message** — the classic SDK trap — so `PToolCall`s, `PUsage` and `PStop` emit after the stream is exhausted.
- Extended thinking with a 2048-token budget (clamped below `maxTokens`); thinking requests carry no sampling parameters.
- Vision: image blocks are reordered before text within a user message.
- Cancel closes the SDK stream; the resulting error maps to `ABORTED` iff the signal fired.

Ollama (NDJSON)

- `POST /api/chat` read line by line; typed wire records (`ChatRequest(model, stream, messages, tools, options, think)`); tool results become `role:"tool"` messages; images ride as raw base64 arrays.
- No call ids from the server — the adapter mints `ollama-call-<nanos>`.
- Token counts arrive only in the final `done:true` chunk.
- **Thinking, two ways:** the native `message.thinking` field, plus a streaming state machine that splits inline `<think>` tags even when a tag is torn across chunk boundaries.
- **Vision fail-fast:** before sending images, `POST /api/show` checks the model's capabilities — a text-only model is refused with an actionable message instead of a confusing 400.
- Error classification at the source: retryable HTTP → `TransientProviderException`; terminal (404 model-not-pulled, 401) → `IllegalStateException` — never retried.

OpenAI-compatible (SSE)

- `data:` lines until `data: [DONE]`; keep-alive comments skipped; `stream_options.include_usage` puts usage in the final chunk.
- **Fragmented tool calls** accumulate per index until the turn closes; arguments arrive as a JSON *string* and are re-parsed (unparseable → `{"raw": …}`).
- Bearer auth only when a key is present — LM Studio and friends run keyless.
- If the base URL still equals the Ollama default, it swaps to `http://localhost:1234` (LM Studio's port).

CHAPTER 25

The WebSocket protocol

One endpoint (`/ws`), one connection per browser tab, one agent per connection. Server→client traffic is *nothing but serialized RunEvents* — there is no separate frame vocabulary in that direction. Client→server is six small frames.

Client → server

FRAME	SHAPE	SEMANTICS
`user_message`	`{type, text, attachments?: [{mediaType, dataBase64}]}`	starts a run (refused with an error event while one is active). Attachments are decoded and stored *before* the run starts; 5 MB cap per attachment; supported: jpeg/png/webp/gif
`permission_response`	`{type, callId, allowed, remember?, persist?}`	answers a parked permission future. `remember` adds a session rule, `persist` appends it to `.spectro/settings.json`
`abort`	`{type}`	fires the current run's CancelSignal
`set_provider`	`{type, provider, model?}`	mid-session backend switch; validated; refused without a key; applies on the next run
`set_image_provider`	`{type, provider}`	`gemini` | `openai`; effective on the next generation
`set_thinking`	`{type, enabled}`	reasoning visibility; applies on the next run

Nuances the names hide: the client frame is `permission_response` — `permission_decision` is the *RunEvent* that flows back and into the file. Cancel is spelled `abort` on the wire. Resume is *not* a frame: it is the `?resume=<id>` query parameter at connect time (kept across auto-reconnects); archive replay is pure REST.

Server → client

Every event of a run goes to the socket *and* the session file — the same object. Server-side errors (invalid frame, refused switch, oversized attachment) arrive as first-class `error` RunEvents. One synchronized writer per connection; a dead socket swallows sends — "a dead socket is not a run failure; the JSONL already has it."

Threading model

- The Tomcat thread only parses frames; every run executes on a virtual thread (`spectroscope-run`).
- Permission waits *park* the agent's virtual thread on a `CompletableFuture` keyed by `callId` — cheap, and the browser dialog answers with the same id. On disconnect every pending future completes with *deny*, so no thread ever hangs.
- Two tabs = two connections = two agents, two session files, two independent remembered-rule lists. Only the persisted settings file is shared — its writer is serialized.

- Text frames up to 16 MB (base64 images need the headroom; the Spring default of 8 kB would close the socket with status 1009).

CHAPTER 26

The REST API

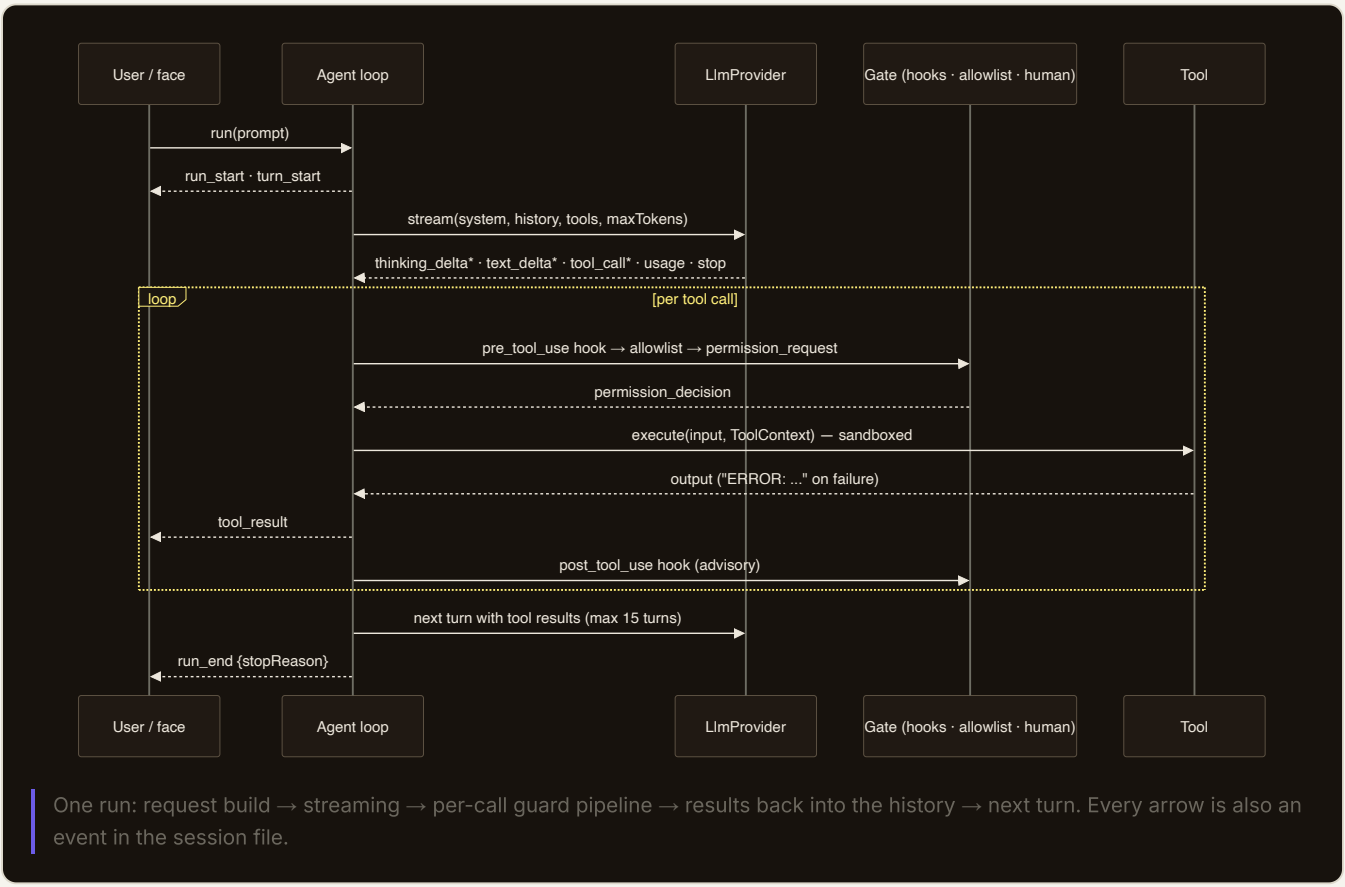
Read-only by design — the socket carries every mutation — with exactly two exceptions: the transcription POST and the guarded session DELETE. Bound to 127.0.0.1; auth and TLS are a deliberate course boundary.

ENDPOINT	RETURNS	NOTES & GUARDS
`GET /api/health`	`{"status":"ok"}`	the desktop shell's startup probe
`GET /api/sessions`	`[{id, startedAt, firstPrompt, tokens, provider}]`	fold over the sessions dir; broken files skipped
`GET /api/sessions/{id}/events`	the session's RunEvents as a JSON array	replay + resume seeding; torn last line dropped
`DELETE /api/sessions/{id}`	204 · 400 · 404	**the one destructive endpoint**: id-shape guard (`[A-Za-z0-9][A-Za-z0-9-]*`) → 400; unknown → 404; the store additionally refuses ids that do not resolve to a direct child of the sessions dir. Removes JSONL + blob folder
`GET /api/config`	`{provider, model}`	boot truth; live switches are client-side overlay
`GET /api/context`	system prompt, tools, skills, MCP servers, thinking, provider/model, subagent role profiles	stateless — assembled from the same constants the live agent uses; no agent built, MCP not connected
`GET /api/models?provider=`	`[string]`	ollama: live from `/api/tags` with finite 1.5 s/2.5 s timeouts (a stalled Ollama must not pin a worker); cloud: curated; errors → `[]`
`GET /api/images/{file}`	image bytes	name must match `[0-9a-f]{64}\.(png|jpg|webp)` — 400 *before* any filesystem access; the content address is the traversal guard
`GET /api/jobs/state`	`{jobId: JobState}`	cron outcomes; corrupt file → `{}` ; polled by the desktop shell
`GET /api/files`	the sandboxed workspace tree	hidden + ignored dirs skipped; depth ≤ 8; ≤ 2000 entries with an honest `truncated` flag
`GET /api/file?path=`	one file for preview	`resolveInside` + per-segment hidden/ignored re-check (the `.env` answers 404 even by direct URL); every document served with `Content-Security-Policy: sandbox allow-scripts` ; 413 oversized, 415 binary
`POST /api/transcribe`	`{text}`	browser audio → ffmpeg 16 kHz mono → whisper-cli; 503 with a setup hint when STT is not installed; the transcript goes to the composer, never to the agent
`GET /api/claude/transcripts`	`[{path, project, file, size, modifiedAt}]`	lists `~/.claude/projects/**/*.jsonl` (newest first, ≤ 300) — the folder is Finder-invisible, hence this endpoint
`GET /api/claude/transcripts/content?path=`	raw JSONL	must end `.jsonl` ; canonical real-path must stay inside the real base (no traversal, no symlink escape); 64 MB cap

CHAPTER 27

The agent loop, hop by hop

The whole harness reduces to one loop in one class: build a request, stream the response, execute the tool calls behind the guards, feed the results back, repeat — at most fifteen times. Everything else in this book hangs off one of these hops.



One run: request build → streaming → per-call guard pipeline → results back into the history → next turn. Every arrow is also an event in the session file.

The turn, precisely

1. **Run start.** Fresh UUID, `run_start` with the *live* provider label (a switched backend reports itself truthfully). Attachments convert to image content *before* the prompt text in the first user message.
2. **Per turn** (1…15):
   a. `turn_start`; with introspection on, a `context_info` estimate of what the next call will carry;
   b. the compaction check (chapter 28) — a no-op below the threshold;
   c. the request: system prompt, full history copy, tool specs, maxTokens (default 32 000), thinking flag, the cancel signal;
   d. streaming: text deltas emit immediately; thinking deltas emit but never join the history; tool calls collect; usage updates the context measure; the stop reason records;
   e. abort check — a cancelled run ends here with `aborted`;
   f. history append: the assistant message (text + tool-call blocks) goes in *before* any results — the API's ordering rule;

    g. no tool calls → `run_end` with the mapped stop reason; otherwise:
    h. the tool round: every call passes the guard pipeline below, in order; all results of the round return to the model in *one* user message — denials and errors included, as data the model can read and react to.

3. **Turn 15 exceeded** → `run_end {stopReason:"max_turns"}` — the runaway brake.
4. **Any exception** → `error` + `run_end {stopReason:"error"}` (or `aborted` if the signal fired). The stream terminates on every path.

The guard pipeline (per tool call)

Chapter 13 showed it from the user's side; this is the implementation order inside `runGuarded`, and it is load-bearing:

1. `pre_tool_use` **hooks** — before anything else, so a policy veto costs no dialog. First matching block wins; timeout fails open.
2. **The gate** — only for `needsPermission()` tools: `permission_request` → the broker blocks the virtual thread (terminal read, or a parked future in the web) → `permission_decision` — always emitted, allowlist answers included. Denial returns the literal `ERROR: the user denied the execution.`
3. **Execution** — `tool.execute(input, context)`; the context carries the sandbox root, the cancel signal, the caller's agent id, the call id and an `emit` sink through which artifact tools publish their additive events (`plan` , `image_generated` , `agent_message`).
4. `post_tool_use` **hooks** — after the fact, advisory, never rewriting.

The headless variant

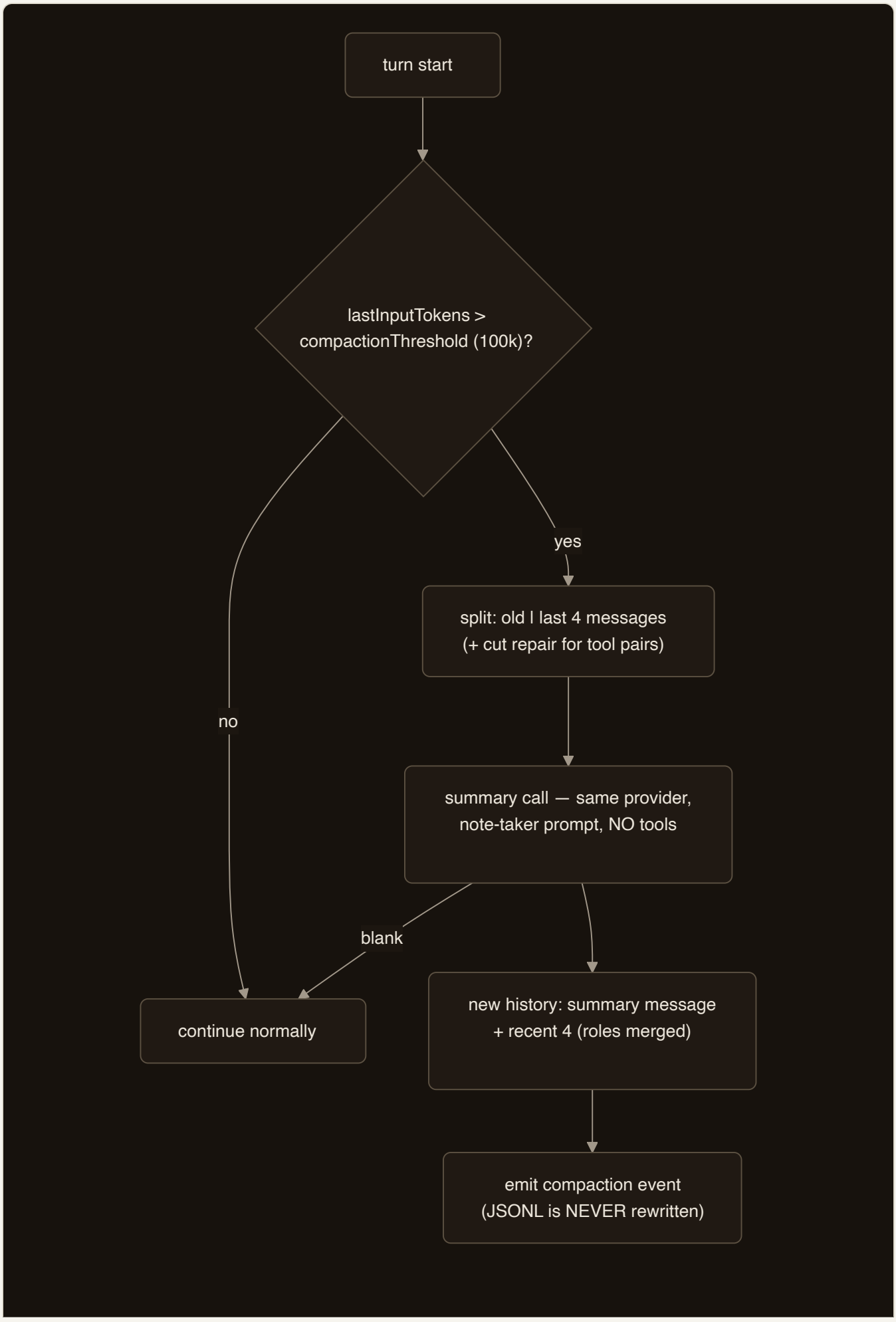
`spectro run` and cron jobs share a runner with a fixed system prompt for unattended operation (“there is no human at the terminal: do not ask questions … if a tool is denied, do not retry it”), a constant-policy broker (readonly denies, auto allows), standard tools only (never the spawn tools), an external turn brake, and a result record (final text, stop reason, session id, exit-ok). Every headless run writes a normal session file — automation is fully auditable after the fact.

CHAPTER 28

Context: compaction, caching, introspection

Three mechanisms manage the finite context window: compaction shrinks the history when it grows past the threshold, prompt caching makes the stable prefix cheap, and introspection shows you both at work.

Compaction


turn start
lastInputTokens > compactionThreshold (100k)?
yes
no
split: old | last 4 messages (+ cut repair for tool pairs)
summary call — same provider, note-taker prompt, NO tools
blank
continue normally
new history: summary message + recent 4 (roles merged)
emit compaction event (JSONL is NEVER rewritten)

> Above the threshold: split the history, summarize the old part with the same model (no tools), swap in "summary + last 4 messages". In memory only — the file grows, never shrinks.

- **Trigger:** the last turn's context size vs. `compactionThreshold` (default 100 000 input tokens) — checked at the start of each turn. The measure is honest under caching: `inputTokens + cacheRead + cacheCreation`, because cached tokens still occupy the window even though the provider bills them separately.
- **Mechanism:** everything except the last four messages is summarized by the same provider (note-taker system prompt, summary-only instruction, no tools); the new history is one summary message plus the recent four, with a cut repair so no tool result loses its call. A failed summary is an `error` event and an unchanged history — never a broken run.
- **The file never shrinks.** A `compaction` audit line is appended; a later resume reconstructs the *full* history from the file and compacts again if needed — the file is always the unabridged truth.
- **Force it:** `/compact` in the REPL calls the same mechanism regardless of the threshold.

Prompt caching

Chapter 16 covered the two breakpoints (system + tools; the last stable message). The reference detail: only text and tool-result blocks can carry a breakpoint (others pass through, best effort); the index is recomputed fresh every call, so a history reshaped by compaction simply gets a new breakpoint; and the wire `usage` event never includes the fold — a cache hit must not change the bytes an old consumer sees.

Introspection

With introspection on (both interactive faces enable it), every turn emits `context_info`: message count, an estimated token total against the threshold, and labeled parts — system prompt, tool schemas, conversation — each measured in chars and estimated at chars/4. The web header's **context ring** renders it live (green < 70 %, amber ≤ 90 %, red above); its popover shows the parts table with the honest caveat that the estimate is an estimate — the `usage` line is the truth. The "Context window" scenario (chapter 7) demonstrates the whole arc: fill → warn → compact → continue.

CHAPTER 29

MCP internals

Chapter 18 explained MCP for users; this is the wire level. The `dev.spectroscope.core.mcp` package is container-free: hand-rolled JSON-RPC 2.0 records, a transport seam, a client with strict delivery semantics, and a registry that treats broken servers as skippable.

JSON-RPC framing

- Protocol version pinned: `2024-11-05`; client info `spectroscope / 1.0`.
- **stdio transport** (primary): one JSON object per line; stdout is protocol, stderr goes to a per-server log file under the system temp dir. Handshake: `initialize` → `notifications/initialized` → `tools/list`. Reads are bounded (20 s) by a watcher thread; **a timed-out channel is poisoned** — the uninterruptible reader is unblocked by closing the streams, and the channel refuses further requests until reconnect.
- **HTTP/SSE transport** (optional): each JSON-RPC frame POSTs to the configured URL; the reply's `data:` lines are concatenated per the SSE spec (a plain JSON body is accepted too); 20 s connect/read timeouts; stateless.
- Tool results: the `text` of all `content[]` blocks joined by newlines; unexpected shapes return the raw JSON rather than failing.

Client semantics

- **At-most-once**: `tools/call` is issued exactly once per tool call. Dead transport *before* the call → one lazy reconnect. Failure or timeout *during* the call → poison + readable `ERROR:`, never a re-issue (a slow-but-successful `add_note` must not run twice).
- **Registry policy:** eager connect at startup, log-and-skip on failure, descriptors cached, config order preserved; handles (name, target, reachable, tool count) feed `spectro doctor` and `/mcp`. Closed on shutdown.
- **Adapter hygiene:** a server-supplied schema that is missing or not an object is replaced by `{"type":"object"}` — untrusted server output must not crash the provider advertisement.
- **Scope:** MCP tools bind to the session at connect time, independent of LLM-provider switches; children never see them.

The reference server

`spectro-mcp-notes` doubles as executable documentation of the server side: a single-threaded stdio loop answering `initialize`, `tools/list`, `tools/call`; JSON-RPC errors `-32601` / `-32603` where they belong; tool-level failures as MCP-style `isError` results (still JSON-RPC successes). Its store is a directory of text files with collision-safe creation; its ranker a few dozen lines of term frequency + substring bonus — search as a small program, not a search engine. Eighteen tests include one that spawns the real child JVM and speaks the protocol end to end.

CHAPTER 30

Configuration reference

Seven layers, seventeen keys (three of them whole-block), plus a small handful of true environment-only secrets. Missing files are fine at every layer; malformed JSON fails loudly — a broken config is a programming error, not a fallback case.

The layers

```
built-in defaults
  <  environment  SPECTRO_*                       (usually injected from ./.env —
                                                   now the BASE, directly above defaults)
  <  ~/.spectro/settings.json                     (user; config.json read beneath it,
                                                   for one release)
  <  <launch-dir>/.spectro/settings.json          (deprecated compat layer)
  <  <workspace>/.spectro/settings.json           (project — travels with the workspace)
  <  <workspace>/.spectro/settings.local.json     (machine-local, gitignored)
  <  CLI flags    --provider --model --base-url --compaction-threshold --workspace
```

The flip (owner decision 2026-07-18, "settings-productization"): env used to sit just below the flags, outranking every settings file — editing `./.env` was the loudest way to configure spectroscope, and a stale line nobody remembered could silently shadow a deliberate choice made in the Settings page. Now env is the BASE, one step above the built-in defaults: it still seeds the starting configuration (a good fit for bootstrap, CI, or a one-shot experiment), but ANY settings file — user, project or local — outranks it. *"Die env ist die Basis, die Settings geben den Ton an."* Later layers win field by field — except `autoApprove`, `mcpServers` and `hooks`, which merge as **whole blocks**: a layer that defines the block replaces it entirely (no per-entry merge). Unknown JSON keys are ignored everywhere, so additive fields never break older builds.

Two resolution moments. Config resolves at two different times, and not every field is meaningful at both. The *process* moment (server boot, `spectro doctor`, CLI startup, a stateless REST call) has no session yet, so only `defaults < env < user < launch-dir < flags` applies — this is where `workspace` itself resolves, and the one-per-process `logLevel`. The *session* moment (the agent build, the composer gear, a per-connection re-apply) joins the session's own resolved workspace pair — project, then local — directly below launch-dir: two concurrent sessions with different workspaces can legitimately run different providers or permission modes. A workspace scope may not set `workspace` itself (a folder must not repoint the agent elsewhere) or `logLevel` (one log file per process) — both fail loudly if you try.

Workspace-supplied config is executed, not just read. A workspace's own `.spectro/settings.json` can set `mcpServers`, `hooks`, `autoApprove` and `permissionMode` — and every one of those takes effect the moment the workspace resolves: an `mcpServers` entry spawns its process, a `hooks` entry runs a shell command around every tool call, and a permissive `autoApprove` / `permissionMode` can auto-allow gates that would otherwise ask. Pinning a folder as your workspace therefore runs whatever its checked-in settings declare — the course maxim *"tool inputs are model output and therefore untrusted"* extends one layer further to *folder-supplied config is untrusted too*. Review a cloned or otherwise foreign repository's `.spectro/` before pointing spectroscope at it, the same way you would before running its build scripts.

The settings API. `GET /api/settings[?session=]` returns the resolved configuration alongside per-field **provenance** (which layer won, which lower layers were shadowed) and the raw, non-empty layers — without `?session=` it is the process-moment view, with one the session-relative view. `PUT /api/settings/{user|project|local}` writes a schema-validated partial patch to exactly one scope; a `null` value removes that key. Secret-shaped keys (`*_API_KEY`, `*_TOKEN`) are always rejected — a settings file is never a place to paste a key.

Two gears, two scopes. The header gear (the Settings page) edits the GLOBAL, user-scope fields only: session defaults (provider/model/thinking/image backend), the default workspace (origin badge + a "reset" that falls back to the layer below), `logLevel`, and the machine-tool paths (`chromeBinary` / `imageModel` / `sttModel`). A second, compact gear sits in the composer row and edits THIS session's workspace: permission mode as a keyboard-navigable text list (live immediately via `set_permission_mode`, persisted to the workspace project file once one is pinned), always-allow rules (add and delete), machine-local overrides, and raw-JSON editors for `mcpServers` / `hooks`. An unpinned session — still in its throwaway temp folder — shows the composer gear's project sections disabled: there is no durable folder to write to yet.

Doctor never lets a shadowed variable go unnoticed. Every run, `spectro doctor` walks the same provenance and prints one line for every `SPECTRO_*` variable that IS set but is no longer the effective source, naming both the variable and the settings layer that now wins:

```
env SPECTRO_MODEL is set but shadowed by user settings (effective model comes from user)
```

On first boot after the flip, if no user settings file exists yet, spectroscope materializes one FROM the current env base (secrets excluded, only the fields the env actually sets) — so day one changes nothing functionally, and from then on that file is the truth the doctor line, the Settings page and every face agree on. `spectro doctor --migrate` renames the old `~/.spectro/config.json` to its new name, `settings.json`, whenever the new name is not yet present.

Deprecation table

Every `SPECTRO_*` variable below still works exactly as before — as the env BASE — but is deprecated in favor of the settings field it feeds:

ENV VARIABLE (DEPRECATED)	SETTINGS FIELD
`SPECTRO_PROVIDER`	`provider`
`SPECTRO_MODEL`	`model`
`SPECTRO_BASE_URL`	`baseUrl`
`SPECTRO_WORKSPACE`	`workspace`
`SPECTRO_THINKING`	`thinking`
`SPECTRO_IMAGE_PROVIDER`	`imageProvider`
`SPECTRO_IMAGE_MODEL`	`imageModel`
`SPECTRO_STT_MODEL`	`sttModel`
`SPECTRO_CHROME`	`chromeBinary`
`SPECTRO_MAX_RETRIES`	`maxRetries`
`SPECTRO_PROMPT_CACHING`	`promptCaching`
`SPECTRO_LOG_LEVEL`	`logLevel`

Not deprecated — these never had a settings-field counterpart: `compactionThreshold` and `permissionMode` (aside from its live socket message) are flag/UI-only, no env form ever existed for them; the API keys in "Environment-only variables" below are true secrets and never graduate into a settings file.

Every key

KEY	TYPE · DEFAULT	ENV BASE (DEPRECATED)	MEANING
`provider`	`anthropic` | `ollama` | `openai` · `anthropic`	`SPECTRO_PROVIDER`	the LLM backend; unknown values fail loudly
`model`	string · `claude-opus-4-8`	`SPECTRO_MODEL`	if provider is ollama/openai and no layer set a model, the default flips to `qwen3` / `local-model`
`baseUrl`	URL · `http://localhost:11434`	`SPECTRO_BASE_URL`	ignored by anthropic; openai swaps an untouched Ollama default to `api.openai.com` when `OPENAI_API_KEY` is set (a key means the cloud), else to `:1234` (LM Studio) — an explicit URL always wins
`compactionThreshold`	int · `100000`	— (flag only)	compaction trigger in input tokens
`permissionMode`	`ask` | `auto` | `readonly` · `ask`	—	chapter 13; headless `run` has its own `--permissions`; live via the composer gear's `set_permission_mode`
`workspace`	path · *unset*	`SPECTRO_WORKSPACE` / `--workspace`	the agent's working directory (file tools, glob/grep, run_command); unset = a per-session folder `<tmpdir>/spectroscope-ws/<session-id>` — a resume finds its files again, and the project you started spectroscope in stays clean; process-global — a workspace scope may not set this (a folder must not repoint the agent)
`autoApprove`	string[] · `[]`	—	allowlist rules: `tool` or `tool:prefix*` — whole-block; lives in the workspace project file once one is pinned
`imageProvider`	`gemini` | `openai` · `gemini`	`SPECTRO_IMAGE_PROVIDER`	image-generation backend
`thinking`	bool · `true`	`SPECTRO_THINKING`	reasoning-stream visibility (REPL `/think`, web header toggle)
`mcpServers`	object · `{}`	—	Claude-Desktop-shaped server map — whole-block (chapter 18)
`maxRetries`	int · `2`	`SPECTRO_MAX_RETRIES`	transient retry budget; 0 disables; garbage values fail readably
`promptCaching`	bool · `true`	`SPECTRO_PROMPT_CACHING`	Anthropic cache_control breakpoints; no-op elsewhere
`logLevel`	`error` | `warn` | `info` | `debug` | `trace` · `info`	`SPECTRO_LOG_LEVEL`	operator-log detail (chapter 31); process-global — one log file per JVM, refused in a workspace scope, user scope only
`imageModel`	string · *unset*	`SPECTRO_IMAGE_MODEL`	overrides the image backend's own default model; machine-tool path — belongs in user or workspace-local scope
`sttModel`	path · *unset*	`SPECTRO_STT_MODEL`	absolute path to the whisper.cpp model (default `~/.spectro/models/ggml-small.bin`); machine-tool path
`chromeBinary`	path · *unset*	`SPECTRO_CHROME`	override for the system-Chrome binary `browse_page` discovers automatically; machine-tool path
`hooks`	array · `[]`	—	pre/post tool hooks — whole-block (chapter 13)

`tts`	`{enabled, voice}` · off, `en_US-lessac-medium`	—	CLI-side voice output block in the same user settings file (not a `SpectroConfig` field)

Every `SPECTRO_*` name in the "env base" column still works exactly as before, but only as the BASE layer — see the deprecation table above for the field each one now defers to, and run `spectro doctor` to see which of yours are currently shadowed. The three "machine-tool path" fields are session-scoped like any other field, but discouraged from a committed project file — the header gear and the composer gear's local-overrides section are where they belong.

Environment-only variables

Everything that used to live only in the environment has graduated to a settings field (deprecation table above). What is left is **secrets** — spectroscope never writes an API key to a settings file, and the `PUT` endpoints refuse to accept one outright (any `*_API_KEY` / `*_TOKEN`-shaped key is rejected):

VARIABLE	CONSUMED BY
`ANTHROPIC_API_KEY`	the Anthropic SDK; checked up front — a missing key refuses provider construction with a readable message
`OPENAI_API_KEY`	optional Bearer for the chat provider; required by the OpenAI image backend
`GEMINI_API_KEY`	the Gemini image backend
`TAVILY_API_KEY`	the Tavily tier of `web_search`; absent falls back to the keyless DuckDuckGo tier

Three `.env` injection paths, one parse rule. Gradle's run tasks (CLI and server) and the `./spectro` launcher all read the project root's gitignored `.env` identically: comments and blank lines skipped, first `=` splits, one quote layer stripped, **empty values excluded** — a `KEY=` line can never blank a variable you exported in your shell. The launcher's copy exists because the desktop face bypasses Gradle entirely. All JavaExec tasks also run from the project root, so the project settings layer is always found.

The single writer. Only `SettingsWriter` ever writes a settings file — two entry points, the same serialized read-modify-write discipline: `appendAutoApprove` (the permission dialog's “Persist”/“dauerhaft” checkbox) appends one allowlist rule to the workspace project file; `patch` (behind every `PUT /api/settings/*` call, so behind both gears) applies a schema-validated partial patch to any of the three files, rejecting secrets and process-globals in the wrong scope before anything touches disk. Both preserve every other key already in the file, and a `local` write also ensures the workspace's `.spectro/.gitignore` lists `settings.local.json`. Everything else only reads.

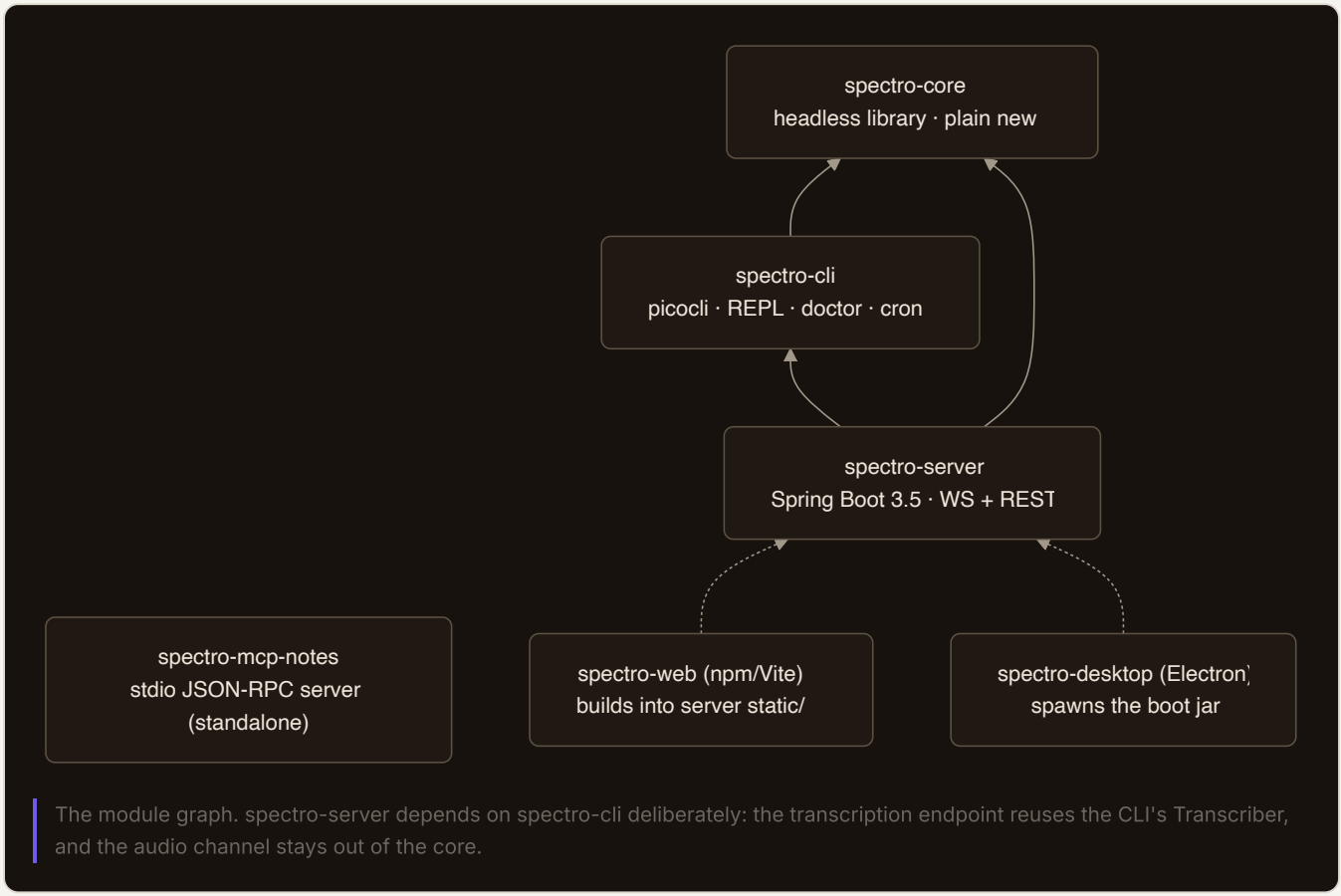
SPECTRO.md

A `SPECTRO.md` in the working directory is appended to the system prompt as a “Project context” section — provider-neutral, visible in the System-context panel. The counterpart of a CLAUDE.md, for spectroscope.

CHAPTER 31

Build & test inventory

Four JVM modules, two npm toolchains, one version catalog — and a test suite that never needs an API key.



The module graph. spectro-server depends on spectro-cli deliberately: the transcription endpoint reuses the CLI's Transcriber, and the audio channel stays out of the core.

Modules

MODULE	ROLE	KEY FACTS
`spectro-core`	the headless library	java-library; Jackson is `api` (JsonNode appears in public contracts); the Anthropic SDK is `implementation` — no other class may import it; Spring *Framework* as a library (RestClient + HTTP interfaces); cron-utils
`spectro-cli`	terminal face	application; picocli; slf4j-nop keeps output clean; registers the `tour` task; stdin wired for the REPL
`spectro-server`	web backend	Spring Boot 3.5.3 (only here); starters web + websocket; depends on core *and* cli; serves the built UI from `static/`
`spectro-mcp-notes`	example MCP server	standalone; Jackson only; its test spawns the real child JVM
`spectro-web`	browser UI	npm/Vite; React 19, React Flow 12, dagre; `npm run build` writes straight into the server's resources

`spectro-desktop`	desktop shell	npm/Electron 37; TypeScript strict; zero runtime dependencies (health polling via built-in fetch)

Version pins

LIBRARY	VERSION	
`com.anthropic:anthropic-java`	2.34.0	owns SSE; client built with maxRetries(0)
`jackson-databind`	2.17.2	the one `api` dependency
`info.picocli:picocli`	4.7.6	CLI framework
`org.springframework:spring-web`	6.2.8	Framework as library — no Boot in the core
`com.cronutils:cron-utils`	9.2.1	computes slots; a ScheduledExecutorService fires them
`org.junit.jupiter`	5.10.2	
Gradle wrapper	9.6.1	checked in
Spring Boot	3.5.3	spectro-server only
react / @xyflow/react / dagre	19.1 / 12.11 / 0.8.5	spectro-web
electron / electron-builder	37.2 / 26.0	spectro-desktop

The test suite

MODULE	TESTS	WHAT THEY PIN
spectro-core	270	event round-trips and wire shapes, the loop against fake providers, sessions/resume/delete, compaction, subagents (parallelism, timeout), scheduler, all three provider mappings against scripted local servers, MCP client/transports, hooks, allowlist, retry
spectro-cli	33	renderer, overrides plumbing, allowlist, voice seams
spectro-server	33	REST guards (incl. the delete endpoint), a full WebSocket round-trip against an Ollama mock, workspace sandbox, transcripts controller
spectro-mcp-notes	18	store, ranker, stdio protocol in-JVM and as a real child process
JUnit total	**354**	plus **252 vitest** for the web UI (reducer, threads, stepper, scene models, scenario compiler, importer, markdown parser, layout stores)
spectro-web (vitest)	252	

Every test runs key-free and network-model-free: fake providers, scripted HTTP servers, injectable process seams, and a redirected `user.home` so no test can ever touch your real `~/.spectro`. Gate: `./gradlew build` + `cd spectro-web && npm test`.

APPENDIX A

Troubleshooting

SYMPTOM	CAUSE & FIX
`zsh: operation timed out: ./spectro`	The folder arrived as a download — every file carries the macOS quarantine flag, and Gatekeeper hangs in an online notarization check on exec. Fix: `xattr -cr <path>/spectroscope` (read-only `.git/objects` complaints are harmless).
`ollama unreachable` in doctor	The Ollama server is not running — open the app or `ollama serve`. Probe: `curl -s http://localhost:11434/api/tags`.
Gradle: "release version 21 not supported"	Your default JDK is older than 21. The launcher normally solves this; without it, install `brew install openjdk@21` or set `JAVA_HOME`.
git hangs / `Operation timed out` on `git diff`	On a sync filesystem (OneDrive/iCloud) the fsmonitor daemon can wedge. Fix: `git config core.fsmonitor false`.
Model without vision error	You attached an image while a text-only local model is selected. `ollama pull qwen3-vl` (or llava) and switch, or use a cloud provider.
Mic button disabled with a tooltip	STT is not installed — `bash scripts/setup-stt.sh`; or the browser denied microphone access.
`Voice output disabled: piper, voice or audio player missing`	`bash scripts/setup-tts.sh` (note: its checksums are pin-on-first-run — the script prints the sums it computed and asks you to verify and paste them).
Anthropic switch refused in the picker	`ANTHROPIC_API_KEY` is not set for the *server* process — put it in `spectroscope/.env` and restart `./spectro web`.
Desktop window blank after a rebuild	An old instance held Electron's single-instance lock. `./spectro desktop` stops stale instances first — use it instead of a bare `npm start`.
MCP server UNREACHABLE in doctor	The configured `command` path does not exist on this machine (the settings file carries machine-specific absolute paths) or the dist was never built: `./spectro mcp-notes`.
`SPECTRO_MAX_RETRIES must be an integer`	Exactly what it says — the env var refuses garbage loudly instead of silently defaulting.

APPENDIX B

Reproducing this guide

This document is generated — never hand-edited — and every asset in it is reproducible from the repository.

SCREENSHOTS	`docs/guide-assets/capture_screens.mjs` — Playwright on the system Chrome against a running `./spectro web`. Deterministic: EN chrome seeded before load, scenario replays for all feature states, two live local-model runs for the plan tab and the permission dialog (the delete shot only *arms* the button and lets it disarm). Output: `docs/guide-assets/shots/*.png`.
MERMAID DIAGRAMS	`docs/guide-assets/mermaid-src/*.mmd` rendered to SVG by `render_mermaid.mjs` (mermaid 11, dark theme on the spectroscope tokens) — the guide inlines the SVGs so the PDF needs no JavaScript.
ARCHITECTURE SVGS	generated by `docs/diagrams/build_NN_*.py`; rerun with `for g in build_*.py; do python3 $g; done`.
TERMINAL CAPTURES	real output of `./spectro` and `./spectro doctor` on 2026-07-16, ANSI stripped at build time.
ASSEMBLY	`python3 docs/guide-assets/build_user_guide.py` — inlines the fonts (from the server's static assets), all images as data URIs, and the content parts from `docs/guide-assets/parts/` into the self-contained `docs/USER-GUIDE.html`.
PDF	headless Chrome: `--headless --no-pdf-header-footer --print-to-pdf=docs/USER-GUIDE.pdf docs/USER-GUIDE.html`. Dark pages incl. margins come from the thead/tfoot page-spacer technique with `@page margin 0`.
SOURCE OF TRUTH	all facts verified against the tree at the phase-5 migration state (2026-07-20); gate at capture time: 540 JUnit + 310 vitest, 0 failures.

Credits and colophon

spectroscope is designed and built by **Christopher Ezell**, with Claude (Anthropic) doing much of the heavy lifting across code, design and copy. It began as the reference harness of a build-an-agent-harness workshop and grew into its own product; the brand, the orchestrator and this guide came after.

The guide is generated, never hand-written. The type is Inter and JetBrains Mono, both embedded so the pages read the same on any machine. The architecture plates are drawn by Python generators; the sequence and flow figures are Mermaid; the screenshots are real captures of spectro web. Everything assembles into one self-contained HTML file and a PDF, in an espresso-dark and a paper-light edition. The wire format is shared byte-for-byte with the TypeScript edition, so a session recorded by one is replayable by the other.