The Orca Platform

ArmyKnife Labs

2026

Table of Contents

- Chapter 13: Installers and Fleet
- Linux Installer: The Fleet Bootstrapper
- Orca/AgentChron client installer bootstrap for Linux.
- This is an internal fleet bootstrapper. It installs from a versioned client
- tarball, then delegates all file writes to the packaged install.sh.
- Per-OS Workstation Client Packages
- Workstation Acceptance Criteria
- Safe prompt — should pass (exit 0)
- Fake provider key — should block (exit 2)
- Direct secret-file read — should block (exit 2)
- Customer Install at Scale: Three Models
- Tenant Scope Model and Scoped Tokens
- Go-Live Acceptance
- GTM SKUs
- Orca Guard and Hooks in the Installer
- Switch to dev mode for local development
- Switch to audit mode for QA
- Switch back to production
- Build
- Dead-upstream smoke
- Test: blocked secret returns 403
- Expected: HTTP 403 and orca_guard_blocked
- Safe requests with dead upstream should return 502, proving the gateway
- attempted to forward only after Guard allowed the body.
- Patterns and Anti-Patterns
- Chapter 14: The Orca Roadmap
- Position: Trust Layer for Open-Weight AI Adoption
- Step 1: Certify and Harden (Revenue Now)
- Step 2: Route with Proof (Recurring Control Plane)
- Step 3: Own the Flywheel (Compounding Moat)
- GraphRAG Strategy
- Real-Time Comprehension Layer
- Claim Discipline
- Build Order
- Competitive Moat Analysis
- GTM Strategy
- The Cypher Coordination Dependency
- Connecting the Themes
- Patterns and Anti-Patterns
- The Bottom Line

The Orca Platform

by ArmyKnife Labs

Table of Contents

Introduction: Orca in One Sitting

The Thesis: AI Work Is Regulated Operational Activity

Every AI-agent session generates an evidence trail. When a developer opens Claude Code and asks it to fix a failing test, the session produces a stream of structured events: the user's prompt, the assistant's reasoning, every tool call to `Read` or `Bash`, every file diff applied, every token consumed, every security boundary crossed. This trail is not disposable chat history. It is a regulated operational activity with an actor, a host, an agent identity, a tool chain, a policy context, a token cost, a security boundary, and an evidence trail.

Consider what happens in a typical five-minute Claude Code session. The developer types a prompt. The assistant reads a file with the `Read` tool, runs a test with `Bash`, edits a source file with `Edit`, runs the test again, and reports success. In that five minutes, the session has produced: a user prompt event, an assistant reasoning event, a `Read` tool call with the file path and contents, a `Bash` tool call with the command and output, an `Edit` tool call with the

old and new text, another `Bash` tool call, a final assistant response, and a token usage event. Each of these is a structured JSONL line in `~/.claude/projects/<project>/<sessionUuid>.jsonl`.

Now multiply that across a team of fifty developers over six months. That is hundreds of thousands of events — every file touched, every command run, every decision made, every boundary crossed. Most organizations throw this away. The session JSONL files sit on developer laptops until the laptop is wiped or the disk fills up. The institutional knowledge embedded in those sessions — what worked, what failed, what patterns emerged, what boundary crossings occurred — is lost.

Orca is ArmyknifeLabs' private AI operations platform. It exists to make that trail private, searchable, auditable, and reusable. The platform captures real agent work from local session roots — Claude Code JSONL, Codex session logs, Antigravity plain-text logs — sanitizes it for leaked credentials before it ever leaves the host, ships it to a central durable store, indexes it for lexical and graph-based search, and promotes reviewed lessons back into operational memory through explicit human or panel review.

The contrarian premise is this: **a model is a perishable snapshot that decays each base-model cycle; the data engine compounds.** The durable intellectual property is not the model. It is the capture pipeline, the renewable corpus, the eval suites, and the retrieval graph. Fine-tuning is an optional downstream consumer, never the product milestone. When a new base model ships — GPT-5, Claude 4, whatever comes next — the previous fine-tuned model is instantly obsolete. But the captured session corpus, the reviewed workflow library, and the eval suites built from real agent work are model-agnostic assets that compound over time.

This book is for developers and architects building AI governance systems. Across fifteen chapters including this introduction, we walk through the Orca platform from architecture to deployment, citing real code from the `agentchron` repository. The repository still uses `agentchron-*` names for crates and binaries for v0.1 compatibility with the live fleet. Product-facing language uses "Orca." Renaming will happen only after migration aliases and rollout scripts are in place.

The Pipeline in One Breath

The platform is fundamentally an edge-to-origin governed data pipeline. A single sentence describes the flow:

```
capture → raw → bronze → silver → gold → GraphRAG retrieval → agent context
```

Here is what happens at each stage:

STAGE	WHAT HAPPENS	WHERE
Capture	`agentchron-agent` watches `~/.claude/projects/**/*.jsonl` using fsevents (macOS) or inotify (Linux). New bytes are parsed, sanitized, and batched for push.	Developer host (Mac/ Linux/VM)
Raw	Session JSONL files are preserved. Remote harvest mirrors raw files to an archive volume before ingest decisions.	Local filesystem or archive volume
Bronze	Events are ingested into the sink, stored in SQLite WAL, and indexed in FTS5. Neo4j graph writes are best-effort.	`agentchron-sink` on `<lab-host>`
Silver	The Data Foundry cleans bronze: secret redaction, PII removal, license classification, deduplication, boilerplate stripping, outcome labeling.	Data Foundry scripts on `.114`
Gold	Reviewed, signed, and licensed records are promoted. Two-reviewer signoff, HMAC gate reports, and DSSE receipts are required.	Data Foundry with governance gates
GraphRAG retrieval	Gold records feed semantic and graph-based retrieval. Context blocks are composed for agent consumption.	Neo4j + Qdrant (planned)
Agent context	Retrieved context is injected into agent sessions through the Context Block API or MCP tools.	ContextOS edge / MCP bridge

The pipeline is the product. Every layer is versioned, auditable, and reversible. The medallion architecture (raw → bronze → silver → gold) with fail-closed governance gates creates a data asset that improves itself through the feedback-loop safety gate: only success-outcome reviewed records re-enter the graph. Raw and bronze records are never queryable GraphRAG input — this is not a configuration option but a structural invariant baked into the data flow.

The capture stage is where `agentchron-agent` does its work. On a developer's Mac, the agent watches `~/.claude/projects/**/*.jsonl` using the macOS FSEvents framework (via the `notify` crate). When Claude Code appends a new event to a session file, the watcher detects the file modification, reads the new bytes from the last checkpoint offset, parses each line through `parse_line()` (which sanitizes for leaked credentials), and batches events for HTTP push to the sink. The checkpoint is a SQLite database at `~/.local/state/agentchron/state.sqlite` that tracks the last successfully shipped byte offset per source file — restarts are safe.

The bronze stage is where `agentchron-sink` does its work. The sink receives events via HTTP (`POST /v1/events`) or TCP push, runs plugin inspection (the `SecretsFilterPlugin` re-runs secret detection before sanitization), performs belt-and-suspenders sanitization, inserts into SQLite WAL (the source of truth), and writes to Neo4j (graph sidecar) and Qdrant (vector scaffold) on a best-effort basis. If Neo4j or Qdrant writes fail, the event is still in SQLite — no data loss.

The silver and gold stages are where the Data Foundry does its work. Silver is the cleaned, deduplicated, license-classified, outcome-labeled version of bronze. Gold is the reviewed, signed, and licensed version of silver — the only stage that can feed GraphRAG retrieval. The promotion path from silver to gold requires two-reviewer signoff, an HMAC gate report, restricted-license exclusion, and a broker-injected signing key. The quarantine path exists to make the fail-closed gate real: if a record cannot pass the gate, it is quarantined, not promoted.

The Three Product Pillars

Orca is defined by three product pillars that shape every architectural decision:

PRIVATE MEMORY

Session evidence is captured privately, scoped by team and workspace, filtered for secrets locally before transport, and only promoted to reusable knowledge through explicit review. The default visibility is `private`. This is not a configuration option — it is baked into the type system through the `OrcaMetadata` scoping struct:

```
// crates/agentchron-core/src/event.rs

pub struct OrcaMetadata {
    pub org: Option<String>,
    pub team: Option<String>,
    pub workspace: Option<String>,
    #[serde(default = "default_visibility")]
    pub visibility: String,  // defaults to "private"
    pub purpose: Option<String>,
    pub task_summary: Option<String>,
    pub context_sources: Vec<String>,
    pub review_state: Option<String>,
    pub review_notes: Option<String>,
    pub promoted: bool,  // defaults to false
}

fn default_visibility() -> String {
    "private".to_string()
}
```

This struct travels with every event through the entire pipeline unchanged. The agent stamps it from environment variables (`ORCA_TEAM` , `ORCA_WORKSPACE` , `ORCA_VISIBILITY`). The sink stores it alongside the event. The search API filters by these scopes. The Data Foundry respects review state and promoted flags when deciding what can feed GraphRAG. Raw and bronze records never become queryable GraphRAG input. Promotion to reusable memory requires explicit review: `session evidence → panel/human review → workflow candidate → approved rule` .

The `promoted` field is false by default and can only be set to true through the workflow promotion process. The `review_state` and `review_notes` fields track the review lifecycle. This is not a flag that an agent can set — it requires server-side admin control through the workflow API.

REAL-TIME REVIEW

Operators can scrub any past session: every prompt, every diff, every shell command, every tool call, every policy decision, every boundary crossing. A Rust/Axum web console (`agentchron-web`) provides timeline views, evidence search, session live views, workflow candidate review, and a context graph explorer. The session view fetches `GET /v1/sessions/:id/live` with `non_boilerplate=true` to filter out command-name-only events, returning `EventRow` objects with visual payloads for tool calls, guardrails, security boundaries, delegations, token usage, and alerts.

The web console is server-rendered with Askama templates — no client-side framework, no SPA, no build step. Static assets (highlight.js for code highlighting, marked.js for Markdown rendering) are compiled into the binary with `include_bytes!` . This enables deployment in network-isolated environments with no CDN dependencies and eliminates supply-chain risk from client-side asset loading.

AI GOVERNANCE

Orca observes the trust chain rather than replacing the systems that enforce it. Orca does not block tool calls. It does not issue identity tokens. It does not sign attestations. It stores, links, searches, graphs, alerts, and exports. The event type system already supports first-class governance events: `token_usage`, `security_boundary`, `governance_boundary`, `attestation`, `orca-trace`, `orca-boundary`, `orca-policy-decision`, `orca-consent`, `mcp-tool-call`, `tool_call`, `guardrail`, `agent_delegation`, `reasoning`, `session_summary`, and `sdlc-pipeline-event`.

These governance event types are not theoretical — they are defined in the `EventKind` enum in `crates/agentchron-core/src/event.rs` and are already flowing through the pipeline. The `ingest` subcommand of `agentchron-agent` can wrap arbitrary JSONL or text logs as any of these governance event types, with extensive metadata fields for boundary, policy_id, authority_level, signature_status, RBAC subject/action/resource/decision, ContextOS attestation, MCP server/endpoint, and token counts.

The Four-Plane Trust Model

The trust architecture is built on four separable planes with zero overlap:

```
AgentShield ENFORCES → Hermes IDENTIFIES → ContextOS ATTESTS → Orca OBSERVES
         ^                                                        |
         `------ workflow library feeds back as new rules --------'
```

PLANE	ROLE	WHAT IT DOES	WHAT IT DOES NOT DO
AgentShield	Enforce	Blocks, allows, or warns before and during tool execution	Does not store session content, does not attest
Hermes	Identify	Establishes agent, persona, workspace, and authority	Does not enforce, does not observe sessions
ContextOS	Attest	Company sign-off, policy-pack version, signer, receipt hash	Does not enforce, does not store session content
Orca	Observe	Stores, links, searches, graphs, alerts, exports	Does not enforce, does not identify, does not attest

The zero-overlap rule is explicit: Orca never enforces, Hermes never stores session content, AgentShield never attests, ContextOS never observes. Each plane has one job. This separation prevents the trust chain from becoming a single monolithic system where a compromise in one plane cascades to all others.

The feedback loop closes the circle: workflow improvements discovered during Orca review are promoted into AgentShield as new runtime rules. A reviewer scrubs a session in the Orca web console, identifies a pattern of risky tool calls, creates a workflow candidate, the candidate is reviewed and approved, and the approved rule is fed back into AgentShield's enforcement logic. The next time an agent attempts a similar risky action, AgentShield blocks or warns based on the rule that originated from Orca's observation.

The priority rule is equally explicit: **identity plumbing comes before UI polish.** Looking Glass cannot render trustworthy guardrail cards unless every event already carries Hermes identity and ContextOS attestation. The UI is only as trustworthy as the data underneath it. A beautiful dashboard built on unattested, unidentified events is a liability, not an asset. This is why the event type system includes `attestation` and `orca-consent` as first-class kinds, and why the `ingest` subcommand supports `--contextos-attestation-id` and `--signature-status` flags.

The Mac → .114 Lab Stack

The production lab stack runs on `<lab-host>` (""). The deployment is a four-service Docker Compose stack:

```
# deploy/docker-compose.yml (excerpt)

services:
  sink:
    image: armyknifelabs/agentchron-sink:latest
    ports:
      - "127.0.0.1:9474:9474"    # HTTP ingest API
      - "127.0.0.1:39478:9478"    # TCP push receiver (host 39478 → container 9478)
    volumes:
      - sink_data:/var/lib/agentchron

  web:
    image: armyknifelabs/agentchron-web:latest
    ports:
      - "9475:9475"              # Web UI + BFF proxy

  neo4j:
    image: neo4j:5.15-community
    ports:
      - "9476:7474"              # Neo4j HTTP
      - "9687:7687"              # Neo4j Bolt

  qdrant:
    image: qdrant/qdrant:v1.9.0
    ports:
      - "9333:6333"              # Qdrant HTTP
      - "9334:6334"              # Qdrant gRPC
```

The port allocation is deliberately offset. The existing GraphRAG stack on `.114` already owns ports `6333/6334/6379/7474/7475/7687`. Orca uses the `9xxx` and `39xxx` ranges to coexist without rearchitecting. This port-offset coexistence pattern enables multi-tenant operation on a single box — the same physical host runs both the pre-existing GraphRAG stack and the Orca stack without port conflicts.

SERVICE	PORT	PURPOSE	BINDING
Web UI / BFF proxy	9475	Operator-facing, off-host `/v1/*` proxy	All interfaces
Sink API	39474	Internal HTTP ingest and query	Localhost only
TCP push	39478	Line-framed JSONL push from watchers/hooks	Localhost only
Neo4j HTTP / Bolt	9476 / 9687	Graph sidecar	Localhost only
Qdrant HTTP / gRPC	9333 / 9334	Vector index (scaffold)	Localhost only

Only the web UI on `9475` is exposed to off-host clients. The sink, TCP push, Neo4j, and Qdrant ports are bound to `127.0.0.1`. This makes the web process the single off-host entry point: clients only need to reach `:9475`, and the

web process handles authentication and forwarding to the internal sink at `:39474`. Inside the Docker network, services communicate using container names — the web process reaches the sink at `http://sink:9474`.

Three capture paths converge on the same `ingest_one()` pipeline at the sink:

1. **Local watcher** (`agentchron-agent run`) — Uses `notify` (fsevents on macOS, inotify on Linux) to recursively watch session roots. An initial sweep backfills all existing files, then a debounced live watcher (500ms default) tails new bytes from the last checkpoint. Multi-root support via `AGENTCHRON_AGENT_ROOTS` allows watching Claude, Codex, and Antigravity roots simultaneously with per-root agent identity.
2. **Remote harvest** (`agentchron-agent pull-remote`) — SSH-based pull from dev VMs. Lists `*.jsonl` via `find -printf`, fetches new bytes via `tail -c +N | head -c M`, optionally mirrors raw files to an archive volume, then ingests through the same `parse_line` path. Source paths are namespaced as `ssh://developer@<host><abs_path>` so checkpoints never collide with local files. SSH connections use `BatchMode=yes` and default keys.
3. **TCP push** (`agentchron-push`) — Streams sanitized JSONL over a TCP connection. The first line is an auth frame (either text or JSON format). Subsequent lines are one JSONL event per line. This is the productized customer path: outbound JSONL only, local sanitizer before transport, same sink-side dedup/sanitize/index pipeline. The `agentchron-push` binary reads from stdin, sanitizes each line with `sanitize_json_line()`, and writes to the TCP stream.

The convergence on `ingest_one()` is a deliberate design choice. Regardless of how events arrive — HTTP batch, TCP stream, or future transport — they go through the same five-step pipeline: plugin inspection (before sanitize), belt-and-suspenders sanitize, SQLite insert (source of truth), Neo4j write (best-effort), Qdrant write (best-effort). This ensures consistent security and durability regardless of the capture path.

What Orca Is Not

Orca is not a chat logger. It does not capture casual conversations. It captures structured operational events from AI agent sessions — tool calls, boundary crossings, policy decisions, token usage, file modifications, and security findings. The event types are defined in `EventKind` and include both Claude Code native types (`User`, `Assistant`, `FileHistorySnapshot`) and Orca governance types (`orca-trace`, `guardrail`, `security_boundary`, `mcp-tool-call`). If you want to log casual chat, use a chat application.

Orca is not a model lab. It does not train models. Fine-tuning is an optional downstream consumer of the gold dataset. The platform's job is to produce governed, renewable data, not to iterate on model weights. The Data Foundry produces dataset products (CPT/SFT/DPO/Workflow Library) from the gold layer, but the platform itself does not run training loops.

Orca is not a commodity inference host. It does not serve model completions. It does not compete with Ollama, vLLM, or cloud inference providers. The model is perishable; the data engine compounds. Ollama runs on `.114` and provides inference, but Orca and Ollama are separate systems with separate concerns.

Orca is not a generic router. It does not route requests between model providers. It observes what happened after the fact, with an evidence trail that is searchable, auditable, and reusable. Routing is a real-time decision; observation is a post-hoc analysis. These are different problems with different solutions.

The distinction matters because it shapes the build order. If the model were the moat, the first milestone would be a fine-tuned model. But the model decays with each base-model cycle. The data engine — the capture pipeline, the renewable corpus, the eval suites, the retrieval graph — is the asset that compounds over time. Building one generic "everything" model first is explicitly a non-goal. The build order is: capture, store, search, review, promote, retrieve, and only then (optionally) fine-tune.

How This Book Is Organized

The book is structured in five parts across fifteen chapters including this introduction:

Architecture (Chapter 1) The Orca Architecture — the Mac → .114 pipeline, the four-crate dependency graph, port allocation, and the three capture paths. This is the structural foundation for all subsequent chapters.

Capture and Storage (Chapters 2–4) - Chapter 2: AgentChron Agent — the per-host capture binary, file watching, checkpoint and spool, HTTP push, SSH remote harvest, and the TCP push wire protocol. - Chapter 3: AgentChron Sink — the central ingest and storage service, Axum HTTP API, TCP push receiver, SQLite WAL durability with FTS5, Neo4j graph writer, Qdrant vector scaffold, and the plugin framework. - Chapter 4: AgentChron Web — the server-rendered UI, session scrubbing with visual payloads, context graph explorer, and the BFF proxy pattern.

Security and Governance (Chapters 5–8) - Chapter 5: Orca Guard — the enforcement plane, three binaries, four modes, and human paste approval. - Chapter 6: Secret Sanitization — the SecureGit regex port, JSON-aware sanitization, shadowing logic, and the plugin framework. - Chapter 7: MCP Gateway — Model Context Protocol integration, discipline-scoped RBAC, and Ed25519 signed receipts. - Chapter 8: Presence Attestation — receipts, provenance, chain of custody, and hardware provider abstraction.

Data and Retrieval (Chapters 9–11) - Chapter 9: GraphRAG Ingestion — Neo4j and Qdrant sidecars, context graph, and the feedback-loop safety gate. - Chapter 10: Data Foundry — the bronze/silver/gold medallion pipeline, fail-closed quarantine, and dataset products. - Chapter 11: ContextOS Edge — the Context Block API, Brain Bundle schema, and operational memory library.

Deployment and Roadmap (Chapters 12–14) - Chapter 12: Deployment Guide — Docker Compose, systemd fleet puller, GKE/Kubernetes, and the Cloudflare edge. - Chapter 13: Installers and Fleet — per-OS install scripts, customer install at scale, and go-live acceptance criteria. - Chapter 14: The Orca Roadmap — the secure AI fabric vision, competitive moat analysis, and build order.

You can read this book sequentially for depth, or jump to any chapter for reference. Each chapter is self-contained, with code examples cited by file path. The Introduction you just read gives you the thesis and the pipeline shape. Chapter 1 gives you the architecture. From there, each chapter drills into a specific component.

PATTERNS INTRODUCED HERE

Three cross-cutting patterns are introduced in this overview and developed throughout the book. These patterns recur across multiple architectural layers — from the local agent on a developer laptop through the central sink, the data foundry, and the deployment edge. They are the connective tissue of the platform.

1. **Defense-in-Depth Sanitization (P1)** — Secrets are redacted at five independent layers: (1) the agent's `parse_line()` before transport, (2) `agentchron-push`'s `sanitize_json_line()` for the TCP path, (3) the sink's `SecretsFilterPlugin` and `ingest_one()` belt-and-suspenders pass, (4) the Data Foundry's realtime sanitizer in silver cleaning, and (5) the foundry's offline deny-list and entropy check. The sink's secrets-filter plugin runs **before** its own sanitizer so that direct clients who skip the agent parser cannot bypass secret reporting. Findings are metadata-only — rule ID, label, severity, occurrence count, and remediation advice — never the matched value or a reversible hash. Chapter 6 develops this fully.
2. **SQLite WAL as Source of Truth (P5)** — SQLite WAL is the durable source of truth at both the agent (checkpoint) and the sink (event store). Neo4j and Qdrant are sidecars whose failures are logged but do not block ingest. The `graph_context()` API tries Neo4j first and falls back to a SQLite-based entity extraction and related-session scoring algorithm. This gives the platform a single durable store that can be backed up with standard SQLite tooling, while allowing graph and vector capabilities to degrade gracefully. Chapter 3 develops this fully.

3. **Private-by-Default Scoping (P6)** — `OrcaMetadata.visibility` defaults to `"private"`. Every event carries org, team, workspace, and review state. The search API filters by these scopes. Promotion to reusable knowledge requires explicit review. Raw and bronze records are never GraphRAG input. Gold promotion requires two-reviewer signoff, HMAC gate reports, restricted-license exclusion, and a broker-injected signing key. This is not a configuration option — it is baked into the type system and the data flow from `agentchron-core` through the foundry. Chapter 10 develops this fully.

KNOWN GAPS ACKNOWLEDGED HERE

Two anti-patterns are acknowledged upfront so the reader has accurate expectations:

1. **Single Shared Token (AP1)** — The sink uses one `AGENTCHRON_INGEST_TOKEN` for HTTP ingest, web proxy, TCP push auth, and all API queries. A compromised read token currently grants ingest and admin capabilities. Split auth into read, ingest, admin, origin, and agent-scoped tokens is a P0 gap with a well-specified implementation path. The Cloudflare edge already models split tokens (read/ingest/admin/origin) but the origin sink does not. Until this is fixed, the sink should not be exposed beyond a trusted network.
2. **The Model as the Moat (AP9)** — The thesis preempted in this introduction. The durable IP is the data engine, not the model. Fine-tuning is optional downstream. This is not a gap to fix — it is a strategic stance that shapes every architectural decision in the platform. Building one generic "everything" model first, or treating fine-tuning as the product, is explicitly a non-goal. The durable IP is the capture pipeline, the renewable corpus, the eval suites, and the retrieval graph. Optimize those; treat any model as optional proof.

The next chapter takes you through the full Orca architecture: the pipeline diagram, the four-crate dependency graph, port allocation, the three capture paths, Claude Code hooks integration, the event type system, and the capability status summary. From there, Chapters 2 through 4 drill into each crate in turn.

Chapter 1: The Orca Architecture

What Orca Is and Why It Exists

Orca's thesis is simple and contrarian: **AI work is regulated operational activity, not disposable chat history.** Every AI-agent session has an actor, a host, an agent identity, a tool chain, a policy context, a token cost, a security boundary, and an evidence trail. Orca exists to make that trail private, searchable, auditable, and reusable.

Consider the gap that Orca fills. A development team adopts Claude Code. Over six months, fifty developers generate hundreds of thousands of session events — prompts, tool calls, file edits, shell commands, token usage. Each session is a JSONL file on a developer laptop. No one is watching these files. No one is sanitizing them for leaked credentials. No one is indexing them for search. No one is extracting patterns or promoting lessons. When a developer's laptop is replaced, the session history is gone. When a new developer joins, they start from scratch — the institutional knowledge embedded in past sessions is inaccessible.

Orca closes this gap. It watches session files, sanitizes them, ships them to a central store, indexes them, and provides tools for review, search, and promotion. The platform is built around three product pillars:

1. **Private Memory** — Session evidence is captured privately, scoped by team and workspace, filtered for secrets locally before transport, and only promoted to reusable knowledge through explicit review. The default visibility

is `private`, baked into the type system through `OrcaMetadata`. This is not a configuration flag — it is a structural invariant enforced by the data contract.

2. **Real-Time Review** — Operators can scrub any past session: every prompt, every diff, every shell command, every tool call, every policy decision, every boundary crossing. A Rust/Axum web console provides timeline views, evidence search, and a context graph explorer. The console is server-rendered with Askama templates and vendored assets — no CDN dependencies, no client-side framework, deployable in air-gapped environments.
3. **AI Governance** — Orca observes the trust chain rather than replacing the systems that enforce it. The trust model is: `AgentShield ENFORCES → Hermes IDENTIFIES → ContextOS ATTESTS → Orca OBSERVES`. Workflow promotion feeds lessons back into AgentShield as runtime rules, closing the loop. Orca's role is observation, not enforcement — it stores, links, searches, graphs, alerts, and exports, but it never blocks, identifies, or attests.

THE `AGENTCHRON-*` VS "ORCA" NAMING CONVENTION

The repository uses `agentchron-*` names for crates and binaries (`agentchron-core`, `agentchron-agent`, `agentchron-sink`, `agentchron-web`). Product-facing language uses "Orca." This duality exists for v0.1 compatibility with the live fleet — installed binaries, systemd units, and Docker images all reference `agentchron-*` names. Renaming will happen only after migration aliases and rollout scripts are in place. Throughout this book, we use the crate names when referring to code and "Orca" when referring to the platform or product.

The `lib.rs` of `agentchron-core` makes the module structure clear:

```
// crates/agentchron-core/src/lib.rs

//! agentchron-core — shared types, JSONL parser, secret sanitizer.

pub mod event;
pub mod parser;
pub mod sanitizer;

pub use event::{Event, EventEnvelope, EventFinding, EventKind, Message, OrcaMetadata,
ToolUse};
pub use parser::ParseError;
pub use sanitizer::{
    detect, sanitize, sanitize_json_line, sanitize_json_value, sanitize_with_known_tokens,
    SecretDetection, FILTER_VERSION,
};
```

Three modules, no I/O. This is the shared foundation that every other crate depends on.

The Mac → .114 Pipeline Architecture

The production lab runs on `<lab-host>` (""). The pipeline is a classic edge-to-origin flow:

```
Client hosts (Mac / Linux / dev VM)                <lab-host> (<lab-host>)
──────────────────────────────────                 ──────────────────────────

agentchron-agent run                                agentchron-sink (Axum, :39474)
  fsevents/inotify watch of                           bearer-token auth
  ~/.claude/projects/**/*.jsonl                       plugin inspection (secrets-filter)
  byte-offset checkpoints in                          belt-and-suspenders sanitize
  ~/.local/state/agentchron/state.sqlite              SQLite WAL store (source of truth)
  sanitize before transport                           FTS5 lexical index
  batched HTTP POST /v1/events                        Neo4j graph writer (:9476/:9687)
                                                      Qdrant vector scaffold (:9333)
tail | sanitize | nc  ── TCP JSONL ──►              agentchron-sink TCP push (:39478)
agentchron-agent pull-remote                          line-framed, bearer auth
  SSH developer@<host>, tail -c +N                    same ingest_one() path
  archive-first mirror to backup volume             agentchron-web (:9475)
                                                      server-rendered UI + /v1/* BFF proxy
```

The edge is the developer host — Mac, Linux, or dev VM — where `agentchron-agent` watches session files, sanitizes each line, and ships events in batches. The origin is `.114`, where the sink stores events in SQLite WAL (source of truth), writes to Neo4j (graph sidecar) and Qdrant (vector scaffold) on a best-effort basis, and the web console serves the operator UI plus a BFF proxy for off-host API access.

The data contract is enforced by shared types: all three downstream crates use the same `Event` and `EventEnvelope` types from `agentchron-core`. An event parsed by the agent is byte-for-byte compatible with what the sink stores and what the web renders. The `OrcaMetadata` scoping struct travels through the entire pipeline unchanged.

THE FULL MEDALLION PIPELINE

Beyond the real-time capture flow, the platform implements a medallion data pipeline:

```
capture → raw → bronze → silver → gold → GraphRAG retrieval → agent context
```

STAGE	DESCRIPTION	GOVERNANCE
Capture	File watching, SSH harvest, or TCP push from edge hosts	Local sanitization before transport
Raw	Original session JSONL preserved on local filesystem or archive volume	Restricted access, manifest-backed
Bronze	Events ingested into SQLite WAL, indexed in FTS5, written to Neo4j	Belt-and-suspenders sanitize at sink
Silver	Data Foundry cleaning: secret redaction → PII removal → license classification → dedup → boilerplate strip → outcome labeling	Hash-only provenance to raw
Gold	Reviewed, signed, licensed records with two-reviewer signoff, HMAC gate, DSSE receipts	Fail-closed quarantine for non-passing records
GraphRAG retrieval	Gold records feed semantic and graph-based retrieval (planned)	Only gold feeds retrieval graph
Agent context	Context blocks injected into agent sessions via Context Block API or MCP tools (planned)	Scoped by team/workspace/visibility

Raw and bronze never become queryable GraphRAG input. The fail-closed governance gate ensures only reviewed, outcome-positive records re-enter the retrieval graph. This is not a policy that can be bypassed — it is a structural invariant. The Data Foundry's silver cleaning order (secret redaction → PII → license → dedup → boilerplate → outcome → decisions) ensures that every gold record has been through a deterministic, reproducible cleaning pipeline.

THE EDGE-TO-ORIGIN FLOW IN DETAIL

Let's trace a single event from capture to storage. A developer types a prompt in Claude Code. Claude Code appends a JSONL line to `~/.claude/projects/<project>/<sessionUuid>.jsonl`. The line looks something like:

```
{"type":"user","sessionId":"abc-123","uuid":"evt-001","timestamp":"2026-06-27T10:00:00Z","cwd":"/
home/dev/project","gitBranch":"main","message":{"role":"user","content":"Fix the failing
test in auth.rs"}}
```

1. **File watch detects modification** — The `notify` watcher in `agentchron-agent` fires a `Modify(Data)` event for the session file. The event path is sent to the unbounded mpsc channel.
2. **Debounce tick processes the path** — After 500ms (default debounce), the tick loop drains pending paths and calls `process_file()` for each.
3. **Seek to last checkpoint** — `process_file()` reads the last byte offset from the checkpoint SQLite database and seeks to that position in the file.
4. **Read new lines** — `BufReader::lines()` reads new lines from the seek position. Each line's byte offset is tracked.
5. **Parse and sanitize** — `parse_line()` is called for each line. It runs `sanitizer::detect()` to produce metadata-only findings, deserializes the JSON into an `EventEnvelope`, extracts tool uses and text, and sanitizes all string content recursively.
6. **Batch** — Events are batched up to 64 (default `batch_size`). When the batch is full or the file ends, `flush()` is called.
7. **Push** — The `Pusher` sends `POST /v1/events` with a bearer token and the batch JSON body. The timeout is 30 seconds.
8. **Checkpoint advance** — On success, the checkpoint is advanced to the new byte offset. On failure, events are spooled and the checkpoint is deliberately not advanced.
9. **Sink ingest** — The sink's `ingest_batch()` handler authenticates the bearer token, then calls `ingest_one()` for each event: plugin inspection, belt-and-suspenders sanitize, SQLite insert, Neo4j write (best-effort), Qdrant write (best-effort).
10. **Storage** — The event is now in SQLite WAL (durable), FTS5 (searchable), and Neo4j (graphable). The web console can display it, the search API can find it, and the graph context API can traverse its relationships.

Port Allocation and Service Topology

The lab stack on `.114` already hosts a pre-existing GraphRAG stack that owns ports `6333/6334/6379/7474/7475/7687`. Orca's ports are deliberately offset to the `9xxx` and `39xxx` ranges to coexist without rearchitecting:

SERVICE	PORT	PURPOSE	BINDING
Web UI / BFF proxy	9475	Operator-facing UI, off-host `/v1/*` API proxy	All interfaces
Sink API	39474	Internal HTTP ingest and query	Localhost only
TCP push	39478	Line-framed JSONL push from watchers/hooks	Localhost only

> **Port convention.** The TCP push receiver listens on container port 9478 inside the Docker network. The host port mapping defaults to 39478 in the lab deployment (the `39xxx` range avoids conflicts with the pre-existing GraphRAG stack on `.114`). When the sink runs directly on the host without Docker, it binds to 39478. This book uses 39478 for the host-visible port and 9478 for the container-internal port. | Neo4j HTTP | 9476 | Graph sidecar HTTP | Localhost only | | Neo4j Bolt | 9687 | Graph sidecar binary protocol | Localhost only | | Qdrant HTTP | 9333 | Vector index HTTP | Localhost only | | Qdrant gRPC | 9334 | Vector index gRPC | Localhost only |

The sink, TCP push, Neo4j, and Qdrant ports are bound to `127.0.0.1` by default. Only the web UI on `9475` is exposed to off-host clients. This makes the web process the single off-host entry point: clients only need to reach `:9475`, and the web process handles authentication and forwarding to the internal sink at `:39474`.

The Docker Compose file makes this topology explicit:

```
# deploy/docker-compose.yml (excerpt)

services:
  sink:
    image: armyknifelabs/agentchron-sink:latest
    ports:
      - "${AGENTCHRON_SINK_BIND_HOST:-127.0.0.1}:${AGENTCHRON_SINK_PORT:-9474}:9474"
      - "${AGENTCHRON_TCP_BIND_HOST:-127.0.0.1}:${AGENTCHRON_TCP_PORT:-39478}:9478"  #
host:container
    volumes:
      - sink_data:/var/lib/agentchron

  web:
    image: armyknifelabs/agentchron-web:latest
    ports:
      - "${AGENTCHRON_WEB_PORT:-9475}:9475"
    environment:
      AGENTCHRON_SINK_URL: "http://sink:9474"
      AGENTCHRON_SINK_TOKEN: "${AGENTCHRON_INGEST_TOKEN}"

  neo4j:
    image: neo4j:5.15-community
    ports:
      - "${AGENTCHRON_NEO4J_HTTP_PORT:-9476}:7474"
      - "${AGENTCHRON_NEO4J_BOLT_PORT:-9687}:7687"
    healthcheck:
      test: ["CMD-SHELL", "wget -q --spider http://localhost:7474 || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 10

  qdrant:
    image: qdrant/qdrant:v1.9.0
    ports:
      - "${AGENTCHRON_QDRANT_HTTP_PORT:-9333}:6333"
      - "${AGENTCHRON_QDRANT_GRPC_PORT:-9334}:6334"
```

Inside the Docker network, services communicate using container names. The web process reaches the sink at `http://sink:9474` (the container-internal port), while off-host clients reach the web proxy at `:9475`. The Neo4j and Qdrant containers are not published outside the Docker network at all — only the sink and web containers can talk to them. The Neo4j healthcheck ensures the sink's `depends_on` condition is met before startup.

The `sink_data` volume persists the SQLite database across container restarts. This is the durable store — the source of truth. The `neo4j_data`, `neo4j_logs`, and `qdrant_data` volumes persist the sidecar state, but since sidecars are best-effort, their data can be rebuilt from SQLite using `agentchron-graph-backfill`.

The Three Capture Paths

All three capture paths converge on the same `ingest_one()` pipeline at the sink. This is a deliberate design choice: regardless of how events arrive, they go through the same plugin inspection, belt-and-suspenders sanitization, SQLite insert, and best-effort sidecar writes.

PATH 1: LOCAL WATCHER (`AGENTCHRON-AGENT RUN`)

The local watcher uses the `notify` crate to recursively watch session roots. On macOS this translates to FSEvents; on Linux it uses inotify. The watcher follows a two-phase pattern:

Phase 1 — Initial sweep: Recursively walks the watch root, collecting all supported files (`.jsonl` everywhere, `.log` only for Antigravity paths). Each file is processed from its last checkpoint byte offset. This ensures no existing session data is missed on startup.

Phase 2 — Live watch: Creates a `notify` recommended watcher filtered to `Create` and `Modify(Data)` events. Path events are sent to an unbounded mpsc channel. A debounced tick loop (default 500ms) drains pending paths and processes each one. `ctrl-c` triggers graceful shutdown.

Multi-root support via `AGENTCHRON_AGENT_ROOTS` allows watching Claude, Codex, and Antigravity roots simultaneously:

```
agentchron-agent run \
  --sink-url http://<lab-host>:9475 \
  --sink-token "$AGENTCHRON_INGEST_TOKEN" \
  --agent-roots "claude=~/.claude/projects;codex=~/.codex/sessions;antigravity=~/Library/
Application Support/Antigravity/logs"
```

When multiple roots are configured, the agent spawns one watcher task per root and collects errors via an unbounded mpsc channel. The `ctrl-c` signal triggers graceful shutdown of all watcher tasks.

PATH 2: REMOTE HARVEST (`AGENTCHRON-AGENT PULL-REMOTE`)

The remote harvester pulls session JSONL from dev VMs over SSH. For each configured host:

1. Lists remote `*.jsonl` files via `find <root> -type f -name "*.jsonl" -printf "%p\t%s\n"` (wrapped in `bash -lc` for tilde expansion).
2. Optionally mirrors raw files to an archive root using chunked `tail -c +N | head -c M` fetches (8MB chunks). The archive path mirrors the host and absolute remote path.
3. Fetches new bytes from each file's last checkpoint using `tail -c +<offset+1> | head -c <bytes_to_fetch>`.
4. Splits on newlines, parses each line with `parse_line()`, and batches to the pusher.
5. Uses the same spool/drain/ack pattern as the local watcher.

Remote source paths are namespaced as `ssh://developer@<host><abs_path>` so the checkpoint table cleanly disambiguates remote vs local files. SSH connections use `BatchMode=yes`, `StrictHostKeyChecking=accept-new`, and a 10-second connect timeout. The SSH user is always `developer` — default ssh-agent and `~/.ssh` keys are used.

```
agentchron-agent pull-remote \
  --hosts <dev-vm>,<lab-host>,<lab-host> \
  --remote-root /home/developer/.claude/projects \
  --archive-root /Volumes/Backups01/agentchron-raw \
  --interval-seconds 60 \
  --sink-url http://<lab-host>:9475 \
  --sink-token "$AGENTCHRON_INGEST_TOKEN"
```

The `--archive-root` option is the archive-first principle in action. Raw JSONL files are mirrored to a backup volume **before** ingest decisions. This preserves the original session corpus even if the remote host is lost, and allows re-ingestion from the archive without SSH access to the original host. The archive fetch uses 8MB chunks to avoid loading entire large files into memory.

PATH 3: TCP PUSH (`AGENTCHRON-PUSH`)

The TCP push path streams sanitized JSONL over a TCP connection. This is the productized customer path: outbound JSONL only, local sanitizer before transport, same sink-side dedup/sanitize/index pipeline.

The protocol is line-framed:

1. **Auth line** — The first line is either `AUTH <token> host=... source_path=... agent=...` (text format) or `{"type":"agentchron_auth","token": "***","host":"...","source_path":"...","agent":"...","orca":{...}}` (JSON format). The auth line establishes a `PushContext` that stamps all subsequent events.
2. **Event stream** — Each subsequent line is one JSONL event. Lines are parsed with `parse_line()`, stamped with the push context's agent and orca metadata, and sent through the same `ingest_one()` path as HTTP events.

The `agentchron-push` binary reads from stdin, sanitizes each line with `sanitize_json_line()`, and writes to the TCP stream:

```
export AGENTCHRON_TCP_HOST=<lab-host>
export AGENTCHRON_TCP_PORT=39478
export AGENTCHRON_SINK_TOKEN="$AGENT...OKEN"
export AGENTCHRON_SOURCE_PATH=/home/developer/.claude/projects/project/session.jsonl
tail -F "$AGENTCHRON_SOURCE_PATH" | agentchron-push
```

Since TCP push does not know the original file byte offset, the sink computes a stable synthetic offset via FNV-1a hashing of `(source_path, line_index, line_content)`. This keeps retries idempotent for the same source stream — if the same line is pushed again after a network reconnection, it gets the same synthetic offset, and the sink can deduplicate.

The Four-Crate Dependency Graph

The Orca codebase is a Rust workspace with four crates in a clean layered architecture:

```
agentchron-core  (types, parser, sanitizer — no I/O)
    ↑
    ├── agentchron-agent  (edge: watch, checkpoint, push, SSH)
    ├── agentchron-sink   (origin: HTTP, TCP, SQLite, Neo4j, Qdrant)
    └── agentchron-web    (UI/BFF: Askama, proxy, vendored assets)
```

AGENTCHRON-CORE

The shared foundation. Exports three modules: `event`, `parser`, and `sanitizer`. No I/O, no network, no storage. Pure data transformation.

The `event` module defines the data contract: `EventKind`, `EventEnvelope`, `Message`, `ToolUse`, `TokenUsage`, `OrcaMetadata`, `EventFinding`, and the higher-level `Event` struct. The `parser` module provides `parse_line()`, the central function that turns a raw JSONL line into a sanitized `Event`. The `sanitizer` module provides `detect()`, `sanitize()`, and `sanitize_json_line()` — the security core that redacts leaked credentials.

The `parse_line()` function is the convergence point for all three capture paths. Whether events arrive from the local watcher, the remote harvester, or the TCP push receiver, they all pass through `parse_line()`:

```
// crates/agentchron-core/src/parser.rs

pub fn parse_line(
    line: &str,
    byte_offset: u64,
    source_path: &str,
    host: &str,
) -> Result<Event, ParseError> {
    let mut findings: Vec<EventFinding> = sanitizer::detect(line)
        .into_iter()
        .map(|d| EventFinding {
            plugin: "secrets-filter".to_string(),
            rule_id: d.rule_id,
            category: "secret".to_string(),
            severity: d.severity,
            summary: format!("{} detected before sanitization", d.label),
            advice: d.advice,
            occurrence_count: d.occurrence_count,
        })
        .collect();
    let raw: Value = match serde_json::from_str(line) {
        Ok(value) => value,
        Err(_) if is_plain_log_event(source_path) => {
            return Ok(parse_plain_log_line(line, byte_offset, source_path, host,
findings));
        }
        Err(err) => return Err(err.into()),
    };
    let normalized = normalize_known_client_event(raw, byte_offset, source_path);
    let envelope: EventEnvelope = serde_json::from_value(normalized)?;
    // ... extract tool uses, text, token usage, sanitize ...
    Ok(Event { envelope, tool_uses, text, /* ... */ })
}
```

The function runs `sanitizer::detect()` on the raw line first — before any JSON parsing — to produce metadata-only findings. Then it attempts JSON deserialization. If the source path indicates a plain-text log (Antigravity `.log` files), it falls back to `parse_plain_log_line()` which wraps the text in a synthetic `OrcaTrace` event. If the event is from Codex, `normalize_known_client_event()` transforms it into the Claude-compatible envelope shape.

AGENTCHRON-AGENT

The edge/client crate. Depends on core and adds: file watching (`notify`), checkpoint persistence (`rusqlite`), HTTP push (`reqwest`), TCP push (`std::net`), SSH remote pull (`tokio::process`), and CLI ergonomics (`clap`). It has six subcommands: `run` , `backfill` , `pull-remote` , `ingest` , `scan-config` , and `doctor` .

The agent crate's `bin/` directory contains four additional binaries: - `agentchron-claude-hook` — Claude Code hook handler - `agentchron-push` — TCP push client - `agentchron-sanitize` — standalone sanitizer - `orca-panel` — panel review adapter

AGENTCHRON-SINK

The origin/server crate. Depends on core and adds: HTTP server (`axum`), TCP server (`tokio::net`), SQLite storage (`rusqlite` with WAL/FTS5), Neo4j graph writer (`neo4rs`), Qdrant client (`qdrant-client`), plugin framework, and auth. It binds an Axum HTTP server on `:39474` and a Tokio TCP listener on `:39478` .

The sink crate's `bin/` directory contains: - `agentchron-graph-backfill` — replays SQLite events into Neo4j - `agentchron-token-backfill` — token usage re-extraction

AGENTCHRON-WEB

The UI/BFF crate. Depends on core minimally — only `OrcaMetadata` and `sanitize_json_line` for the push_wire module. It does **not** depend on `agentchron-sink` ; it talks to the sink over HTTP, maintaining a clean service boundary. Built with Axum + Askama templates. Vendored static assets (highlight.js, marked.js) are compiled into the binary with `include_bytes!` for air-gapped deployment.

THE DATA CONTRACT

The composition is enforced by the data contract: all three crates use the same `Event` and `EventEnvelope` types from core. An event parsed by the agent is byte-for-byte compatible with what the sink stores and what the web renders. The `OrcaMetadata` scoping struct travels through the entire pipeline unchanged:

```
// crates/agentchron-core/src/event.rs

pub struct Event {
    pub envelope: EventEnvelope,
    pub tool_uses: Vec<ToolUse>,
    pub text: Option<String>,
    pub token_usage: Option<TokenUsage>,
    pub byte_offset: u64,
    pub source_path: String,
    pub host: String,
    pub agent: Option<String>,
    pub findings: Vec<EventFinding>,
    pub orca: OrcaMetadata,
    pub sdlc_stage: Option<String>,
    pub receipt_chain_id: Option<String>,
    pub predecessor_receipt_id: Option<String>,
}
```

The `EventEnvelope` uses `#[serde(flatten)]` to capture unknown fields in `extra: BTreeMap<String, Value>` , ensuring forward compatibility with new Claude event types. The `Event` struct wraps the envelope with extracted

tool uses, sanitized text, token usage, byte offset, source path, host, agent identity, findings, and Orca metadata. This is the canonical event shape that flows through the entire pipeline.

WHY `agentchron-web` DOES NOT DEPEND ON `agentchron-sink`

The web crate talks to the sink over HTTP, not via direct crate dependency. This maintains a clean service boundary: the web process can be deployed on a different host than the sink, scaled independently, and updated without recompiling the sink. The web process's `/v1/*path` catch-all proxy forwards all HTTP methods to the sink, authenticates with the sink token, preserves content-type and accept headers, and transparently passes through response status, body, and content-type.

This separation also means the web process stores nothing — all data is fetched from the sink over HTTP. The web process can be restarted, redeployed, or scaled without affecting the sink's durability. The only state the web process maintains is the sink URL and bearer token (from environment variables).

Claude Code Hooks Integration

The Linux bootstrap installer configures Claude Code hooks in `~/.claude/settings.json`. Hooks close the gap between session creation and capture — they ensure that even the first prompt of a session is captured and that risky tool calls are blocked before execution.

CLAUDE EVENT	ORCA COMMAND	PURPOSE
`UserPromptSubmit`	`orca-guard claude-hook`	Block prompts that leak credentials or violate guard policy
`PreToolUse`	`orca-guard claude-hook`	Block risky tool use before it runs
`SessionStart`	`agentchron-claude-hook`	Ingest start-of-session metadata and checkpoint
`PostToolUse`	`agentchron-claude-hook`	Ingest tool results and decisions
`Stop`	`agentchron-claude-hook`	Ingest final turn/session metadata
`SessionStart`	`orca-session-rules.py`	Inject local Orca session rules

The hooks split into two categories:

Enforcement hooks (`orca-guard claude-hook` on `UserPromptSubmit` and `PreToolUse`) run synchronously and can block the action. These are part of the AgentShield enforcement plane. If a user prompt contains a leaked credential, the hook blocks it before it reaches the model. If a tool call is risky (e.g., `rm -rf` in a production directory), the hook blocks it before execution. The enforcement hook returns a non-zero exit code to block the action, and Claude Code respects the block.

Telemetry hooks (`agentchron-claude-hook` on `SessionStart`, `PostToolUse`, and `Stop`) ingest events into Orca without blocking. These are part of the Orca observation plane. They ensure that session metadata, tool results, and final turn summaries are captured even when the file watcher has not yet picked up the JSONL file. The telemetry hook fires immediately, while the file watcher may have a 500ms debounce delay.

The `orca-session-rules.py` hook injects local Orca session rules at `SessionStart`, ensuring the agent is aware of governance constraints from the first turn. This is a Python script that reads the session context and writes rule reminders to the session, guiding the agent's behavior within governance boundaries.

Event Type System and Governance Events

The `EventKind` enum discriminates known Claude Code event types plus generic Orca governance types. Critically, it includes `#[serde(other)] Unknown` so new upstream event types never break parsing:

```
// crates/agentchron-core/src/event.rs

#[derive(Debug, Clone, PartialEq, Eq, Serialize, Deserialize)]
#[serde(rename_all = "kebab-case")]
pub enum EventKind {
    // Claude Code native types
    User,
    Assistant,
    Attachment,
    FileHistorySnapshot,
    LastPrompt,
    PermissionMode,
    TokenUsage,
    SecurityBoundary,
    GovernanceBoundary,
    SessionMeta,
    Attestation,

    // Orca governance types
    OrcaTrace,
    OrcaBoundary,
    OrcaPolicyDecision,
    OrcaConsent,
    McpToolCall,
    ToolCall,
    Guardrail,
    AgentDelegation,
    Reasoning,
    SessionSummary,
    SdlcPipelineEvent,

    /// Any future event type Claude adds.
    #[serde(other)]
    Unknown,
}
```

This is the forward-compatible event schema pattern (P4). `EventKind` uses `#[serde(other)]` for unknown types, and `EventEnvelope` uses `#[serde(flatten)]` to capture unknown fields in `extra: BTreeMap<String, Value>`:

```
pub struct EventEnvelope {
    #[serde(rename = "type")]
    pub kind: EventKind,
    #[serde(rename = "sessionId", default)]
    pub session_id: Option<String>,
    #[serde(default)]
    pub uuid: Option<String>,
    #[serde(rename = "parentUuid", default)]
    pub parent_uuid: Option<String>,
    #[serde(default)]
    pub timestamp: Option<DateTime<Utc>>,
    #[serde(default)]
    pub cwd: Option<String>,
    #[serde(rename = "gitBranch", default)]
    pub git_branch: Option<String>,
    #[serde(default)]
    pub version: Option<String>,
    #[serde(rename = "userType", default)]
    pub user_type: Option<String>,
    #[serde(rename = "isSidechain", default)]
    pub is_sidechain: bool,
    #[serde(default)]
    pub message: Option<Message>,
    #[serde(flatten)]
    pub extra: BTreeMap<String, Value>,
}
```

The parser can absorb new Claude event types, new Codex payload shapes (via `normalize_codex_event()`), Antigravity plain-text logs (via `parse_plain_log_line()`), and arbitrary governance events without code changes. This is the extensibility mechanism that lets the platform survive upstream changes without releases.

THE 16+ GOVERNANCE EVENT KINDS

Beyond Claude's native types, Orca defines 16 product-facing event kinds:

EVENT KIND	PURPOSE	EXAMPLE USE CASE
`orca-trace`	Generic Orca event from a non-Claude client	Custom SDK emitting agent activity
`orca-boundary`	Security or governance boundary crossing	Agent attempts to access production database
`orca-policy-decision`	Policy/RBAC/attestation decision	RBAC deny on restricted tool
`orca-consent`	ContextOS/customer consent or attestation event	Company signs off on data usage
`mcp-tool-call`	MCP server/tool call or result	Gateway logs MCP tool invocation
`tool-call`	Generic tool call from an SDK or runtime	Non-MCP tool call logged for audit
`guardrail`	Policy or safety guardrail evaluation	Guard blocks risky shell command
`agent-delegation`	Cross-agent delegation/handoff	Agent A delegates to Agent B
`reasoning`	Agent reasoning for process visibility	Chain-of-thought captured for review
`session-summary`	Session or workflow completion summary	End-of-session KPI summary
`sdlc-pipeline-event`	CI/CD, IDE, or release pipeline event	Deploy pipeline event linked to session
`token-usage`	Model token accounting	Cost tracking per session
`security-boundary`	Sensitive filesystem/network/secret boundary	Agent reads a file in `/etc/`
`governance-boundary`	Governance/release/approval boundary	Agent attempts unapproved release
`session-meta`	Session start/runtime metadata	Session start with agent identity
`attestation`	Signed ContextOS/ATCS receipt	Cryptographic attestation of session

`ORCAMETADATA` SCOPING

The `OrcaMetadata` struct is the product-level scoping mechanism. It travels with every event through the entire pipeline unchanged and governs how events can be retrieved and promoted:

```
pub struct OrcaMetadata {
    pub org: Option<String>,
    pub team: Option<String>,
    pub workspace: Option<String>,
    #[serde(default = "default_visibility")]
    pub visibility: String,  // "private" by default
    pub purpose: Option<String>,
    pub task_summary: Option<String>,
    pub context_sources: Vec<String>,
    pub review_state: Option<String>,
    pub review_notes: Option<String>,
    pub promoted: bool,  // false by default
}
```

The search API filters by these scopes. Push clients can stamp private-learning scope with `ORCA_TEAM`, `ORCA_WORKSPACE`, `ORCA_VISIBILITY`, `ORCA_PURPOSE`, `ORCA_TASK_SUMMARY`, `ORCA_CONTEXT_SOURCES`, and review fields. `ORCA_VISIBILITY` defaults to `private`. The `promoted` field is false by default and can only be set to true through the workflow promotion process — not by an agent or push client.

PARSER NORMALIZATION FOR NON-CLAUDE CLIENTS

The parser handles three client schemas:

1. **Claude Code** — The native JSONL format with `type`, `sessionId`, `uuid`, `message`, etc. This is the canonical format that `EventEnvelope` was designed around.
2. **Codex** — A different schema with `type` / `payload` nesting and different field names. `normalize_codex_event()` transforms Codex events into the Claude-compatible envelope shape, mapping `event_msg` / `response_item` payload types to Orca event kinds, extracting tool calls from `function_call` payloads, and extracting token usage from `token_count` payloads.
3. **Antigravity** — Plain-text `.log` files, not JSONL. `parse_plain_log_line()` wraps the text in a synthetic `OrcaTrace` event with a generated UUID (`log:{source_path}:{byte_offset}`) and session ID.

This normalization means the sink does not need to know which client produced an event — by the time it reaches `ingest_one()`, every event is in the canonical `Event` shape regardless of its origin format.

Capability Status Summary

The platform is at v0.1.0 — wired end-to-end for capture, storage, search, and review, with graph and vector capabilities in partial states and several productization gaps planned.

CAPABILITY	STATUS	IMPLEMENTATION
Claude/Codex/Antigravity capture, sanitization, HTTP/TCP ingest	Current	`agentchron-agent` watcher, `parse_line()`, `agentchron-push`
SQLite WAL event store and FTS5 search	Current	`agentchron-sink` storage layer with WAL mode, FTS5 external content tables
Rust/Askama web console	Current	`agentchron-web` dashboard, search, session, graph, workflows
MCP bridge (stdio over HTTP API)	Current	`deploy/bin/agentchron-mcp.py`
Workflow and library records	Current	Workflow candidate/approval flows through API and MCP
Data Foundry bronze/silver export and silver review	Current	`foundry/` and `scripts/orca-dataset-export.py`
Neo4j graph context	Partial	Session/event/tool/file/commit graph writes, `graph_context()` API
Qdrant semantic retrieval	Partial	Client scaffold, `write_event()` is a stub
Cloudflare edge gateway	Partial	Worker contract exists, production origin needs hardening
Split read/ingest/admin/origin auth	Planned	Edge has split tokens; origin sink uses one shared token
GraphRAG retrieval	Planned	Embeddings, citations, context blocks, feedback gate
Event streaming/webhooks	Planned	Poll `/v1/sessions/:id/live` for now
Gold promotion and GraphRAG feedback gate	Planned	Two-reviewer signoff, HMAC gate, DSSE receipts

CURRENT CAPABILITIES

Capture and sanitization are fully current. The agent watches files, parses JSONL, sanitizes for 18 secret patterns (GitHub PATs, GitLab tokens, Anthropic keys, OpenAI keys, AWS keys, Stripe keys, PEM blocks, JWTs, and more), and ships via HTTP or TCP. The remote harvest path mirrors raw files to an archive volume before ingest. The TCP

push path sanitizes with `sanitize_json_line()` before transport. The sink runs the `SecretsFilterPlugin` before its own sanitizer pass, ensuring direct clients cannot bypass secret reporting.

SQLite WAL storage and FTS5 search are fully current. The sink maintains tables for events, sessions, tools, token usage, MCP tool calls, guardrails, security boundaries, alerts, plugin findings, workflow improvements, library artifacts, and FTS documents. FTS5 uses the `unicode61` tokenizer and indexes text, kind, cwd, git_branch, host, source_path, tool_names, and tool_inputs. Search supports snippet generation, source-path-fragment filtering, and pagination.

Web console is fully current. Server-rendered with Askama templates, it provides a dashboard with KPI strip (sessions, events, tool calls, guardrails, alerts, tokens, agents), evidence search with filters (q, session_id, host, agent, source, kind, tool, team, workspace, visibility), per-session live views with visual payloads, a force-graph context explorer, and workflow candidate review with status tabs and risk/stance/confidence indicators.

PARTIAL CAPABILITIES

Neo4j graph is partial. The graph writer creates `(:Session)-[:HAS]->(:Event)` with edges to Tool, Agent, Branch, File, and Commit nodes. The `graph_context()` API supports seed-based and entity-query modes with weighted scoring. However, the graph is a sidecar — if Neo4j writes fail, ingest still succeeds, and `graph_context()` falls back to a SQLite-based entity extraction algorithm. The graph is a query acceleration layer, not a source of truth.

Qdrant vectors are partial and explicitly a stub. The Qdrant client is connected, but `write_event()` only logs a debug message and returns `Ok(())`. The code comment is honest: "embedding model wiring lands in v0.2." The presence of a Qdrant container in the compose stack could mislead operators into assuming semantic search is functional — it is not.

Cloudflare edge is partial. A Worker gateway contract exists with a token-scoped model (read/ingest/admin/origin), but production origin hardening remains. The edge is the public gateway and token boundary, replaceable with customer ingress for private enterprise.

PLANNED CAPABILITIES

Split auth is a P0 gap. The origin sink uses one `AGENTCHRON_INGEST_TOKEN` for all auth roles — HTTP ingest, web proxy, TCP push, and all API queries. A compromised read token currently grants ingest and admin capabilities. The implementation path is well-specified: split into read, ingest, admin, origin, and agent-scoped tokens. The Cloudflare edge already models this split; the origin sink needs to follow.

GraphRAG retrieval is planned. The full vision includes embeddings, citations, context blocks, and a feedback gate. Only gold (reviewed, signed, licensed) records feed the retrieval graph. The feedback-loop safety gate ensures that retrieval success-vs-failure becomes free DPO preference data — every retrieval that succeeds or fails is a labeled training example for future model improvement.

Gold promotion is planned. The Data Foundry produces bronze and silver today. Gold requires two-reviewer signoff, HMAC gate reports, restricted-license exclusion, and a broker-injected signing key. The quarantine path exists to make the fail-closed gate real: if a record cannot pass the gate, it is quarantined, not promoted.

Patterns Developed Here

Three cross-cutting patterns are established in this architecture chapter:

PATTERN 4: FORWARD-COMPATIBLE EVENT SCHEMA

`EventKind` uses `#[serde(other)]` for unknown types, and `EventEnvelope` uses `#[serde(flatten)]` to capture unknown fields in `extra: BTreeMap<String, Value>`. The parser absorbs new Claude event types, new Codex payload shapes, Antigravity plain-text logs, and arbitrary governance events without code changes. The `normalize_codex_event()` function demonstrates the pattern for adapting non-Claude schemas into the canonical envelope.

This pattern is the extensibility mechanism that lets the platform survive upstream changes without releases. When Claude Code adds a new event type, the parser does not break — the event is classified as `Unknown` and its fields land in `extra`. When a new AI client (Codex, Antigravity) is added, a normalization function adapts its schema into the canonical envelope shape. The cost of adding a new client is a normalization function, not a schema migration.

PATTERN 5: SQLITE WAL AS SOURCE OF TRUTH WITH BEST-EFFORT SIDECARS

SQLite WAL is the durable source of truth at both the agent (checkpoint) and the sink (event store). Neo4j and Qdrant are sidecars whose failures are logged but do not block ingest. The durability hierarchy is explicit: SQLite insert must succeed; Neo4j/Qdrant writes are best-effort.

The `graph_context()` API tries Neo4j first and falls back to a SQLite-based entity extraction and related-session scoring algorithm. This gives the platform a single durable store that can be backed up with standard SQLite tooling (`sqlite3 events.sqlite .dump`, file copy, WAL checkpoint), while allowing graph and vector capabilities to degrade gracefully. When Neo4j is down, the platform still has graph-like query capabilities through the SQLite fallback. When Qdrant is down (or not yet wired), the platform still has lexical search through FTS5.

PATTERN 12: REPRODUCIBILITY ENVELOPE ON EVERY ARTIFACT

Every generated dataset file carries a reproducibility envelope: SHA-256, byte size, record count, schema version, source corpus IDs, filter versions, generation command, and git commit of the exporter. Run manifests live under `manifests/runs/` for bronze, silver, and gold. Datasets are reproducible by manifest + code commit.

The same principle applies to the sanitizer filter version (`agentchron-secrets-filter-v1`) that travels in findings metadata for rotation tracking. Every event carries the version of the parser and sanitizer that produced it, enabling reproducibility and auditability across the pipeline. If a secret pattern is added to the sanitizer, the filter version changes, and findings from before and after the change are distinguishable.

Anti-Patterns to Address

Two anti-patterns are identified in the architecture overview so the reader has accurate expectations from the start:

ANTI-PATTERN 1: SINGLE SHARED TOKEN FOR ALL AUTH ROLES

The sink uses one `AGENTCHRON_INGEST_TOKEN` for HTTP ingest, web proxy, TCP push auth, and all API queries. A compromised read token currently grants ingest and admin capabilities. The Cloudflare edge models split tokens (read/ingest/admin/origin) but the origin sink does not.

This is a P0 gap with a well-specified implementation path. The mitigation is to split auth into read, ingest, admin, origin, and agent-scoped tokens. Until this is implemented, the sink should not be exposed beyond a trusted network, and the web proxy on `:9475` is the only off-host entry point. The constant-time token comparison (`token_matches()`) is already in place — the issue is not the comparison method but the token granularity.

ANTI-PATTERN 3: QDRANT VECTOR WRITER IS A DEAD STUB

The Qdrant client is connected, but `write_event()` only logs a debug message and returns `Ok(())`. The presence of a Qdrant container in the compose stack and a `Vector` struct in the sink could mislead operators into assuming semantic search is functional. It is not.

The code comment is honest: "embedding model wiring lands in v0.2." The mitigation is to clearly label the stub in deployment docs and the dashboard until an embedder is wired. The Qdrant container can remain in the compose stack for when embeddings are ready, but operators should not expect semantic search to work in v0.1. Lexical search through FTS5 is the current search implementation.

THE SINK'S ROLE IN THE TRUST MODEL

In the four-plane trust model, the sink is the physical embodiment of the Orca observation plane. It does not enforce (AgentShield's job), identify (Hermes's job), or attest (ContextOS's job). It receives events that already carry enforcement decisions, identity metadata, and attestation references, and it stores them durably, indexes them for search, links them into a graph, and makes them available for review.

The `ingest` subcommand's extensive metadata flags show how the trust planes connect to Orca:

FLAG	TRUST PLANE	PURPOSE
`--agentshield-rule-id`	AgentShield	Which enforcement rule was evaluated
`--agentshield-anomaly-id`	AgentShield	Which anomaly was detected
`--authority-level`	Hermes	What authority level the agent holds
`--signature-status`	Hermes	Whether the agent's identity was verified
`--hermes-token-fingerprint`	Hermes	Token fingerprint from substrate receipts
`--contextos-attestation-id`	ContextOS	Which attestation receipt covers this action
`--contextos-attestation-status`	ContextOS	Whether the attestation is signed, missing, expired, or rejected
`--rbac-subject` / `--rbac-action` / `--rbac-resource` / `--rbac-decision`	RBAC	The access control decision and its context
`--boundary`	Governance	Which security/governance boundary was crossed
`--policy-id`	Governance	Which policy was evaluated

These fields are stored in the event's `extra` BTreeMap and are queryable through the search API. The web console can render them as evidence cards. The workflow review process can filter on them. This is how Orca observes the trust chain without becoming part of it — the data flows in, gets stored, and becomes available for review, but Orca never makes enforcement or attestation decisions itself.

THE REPRODUCIBILITY PRINCIPLE

Every artifact in the pipeline carries a reproducibility envelope. The sanitizer stamps its filter version (`agentchron-secrets-filter-v1`) into findings metadata. The parser stamps its crate version into `envelope.version`. The Data Foundry stamps SHA-256, byte size, record count, schema version, source corpus IDs, filter versions, generation command, and git commit into run manifests. This means any dataset can be reproduced from its manifest and the corresponding code commit.

The reproducibility envelope is non-negotiable in the build spec. It is observed in real run manifests on `.114` (e.g., `gold-20260624T115957Z.json` with its DSSE receipt). The same principle applies to the sanitizer filter version that travels in findings metadata for rotation tracking — if a secret pattern is added or modified, the filter version changes, and findings from before and after the change are distinguishable. This enables auditability: you can always determine which version of the sanitizer produced a given finding, and you can replay any dataset from its manifest.

Conclusion

The Orca architecture is a textbook edge-to-origin flow with clean separation of concerns across four crates. The `agentchron-core` → `agentchron-agent` / `agentchron-sink` / `agentchron-web` dependency graph is clean, with core owning the data contract and each downstream crate owning its I/O boundaries. The Mac → .114 pipeline provides defense-in-depth sanitization, byte-offset checkpointing for replay safety, and SQLite WAL as the durable source of truth with best-effort graph and vector sidecars.

The key architectural strengths are: forward-compatible event schema (new event types never break parsing), private-by-default scoping (visibility defaults to `private` in the type system), constant-time auth (avoids timing side-channels), vendored assets for air-gapped deployment, and archive-first remote harvest. The key technical debts are: single shared token for all auth roles, mutex-based SQLite concurrency, and a Qdrant stub that could mislead operators about semantic search readiness.

The next three chapters drill into each crate in turn: Chapter 2 covers `agentchron-agent` (the edge capture binary), Chapter 3 covers `agentchron-sink` (the central ingest and storage service), and Chapter 4 covers `agentchron-web` (the UI and BFF proxy). From there, Chapters 5 through 8 cover the security and governance layers.

Chapter 2: AgentChron Agent

The Agent's Role in the Pipeline

`agentchron-agent` is the edge of the edge-to-origin flow. It runs on every developer host — Mac, Linux, or dev VM — and its job is to capture AI agent session evidence, sanitize it for leaked credentials, and ship it to the central sink. It is the first layer of defense in the platform's security posture and the first link in the governed data pipeline.

The agent is designed to be lightweight, reliable, and unobtrusive. It runs as a foreground process (or under systemd/LaunchAgent), watches files using the operating system's native file event framework, and ships events in batches over HTTP or TCP. It does not store session content beyond the byte-offset checkpoint — the actual event data lives in the source JSONL files and, after shipping, in the sink's SQLite database. The agent's local state is minimal: one SQLite database with two tables (`file_offset` and `spool`), typically a few kilobytes.

The agent binary has six subcommands:

SUBCOMMAND	PURPOSE	TYPICAL USE CASE
`run`	Live watcher: recursively watch session roots, backfill existing files, then tail new bytes	Daily development — runs continuously under systemd or LaunchAgent
`backfill`	One-shot: process all existing JSONL under the watch root and exit	Initial setup or after agent installation on a new host
`pull-remote`	SSH fleet harvest: pull session JSONL from remote dev VMs over SSH	Centralized capture from a fleet of dev VMs
`ingest`	Generic Orca ingest: wrap arbitrary JSONL/text as sanitized governance events	Integrating non-Claude tools (MCP gateway, CI/CD, custom SDKs)
`scan-config`	Scan AI client config files for embedded secrets, emit metadata-only findings	Security audit of AI client configurations
`doctor`	Print effective config and exit	Debugging configuration issues

```
// crates/agentchron-agent/src/main.rs

#[derive(Subcommand, Debug)]
enum Cmd {
    /// Run the local watcher in the foreground.
    Run(RunArgs),
    /// One-shot backfill of all existing JSONL under --watch-root.
    Backfill(RunArgs),
    /// Pull session JSONL files from remote dev VMs over SSH and ship to sink.
    PullRemote(PullArgs),
    /// Ingest arbitrary text/JSONL logs as sanitized Orca governance events.
    Ingest(Box<IngestArgs>),
    /// Scan AI client config files for embedded secrets and emit metadata-only findings.
    ScanConfig(ScanConfigArgs),
    /// Print effective config and exit.
    Doctor(RunArgs),
}
```

The `run` and `backfill` subcommands share the same `RunArgs` struct — the only difference is that `backfill` sets `backfill_only: true`, which causes the watcher to process existing files and exit without starting the live watch phase. The `doctor` subcommand also takes `RunArgs` and prints the effective configuration: watch roots, state directory, sink URL, host ID, and whether each watch root exists on the filesystem.

MULTI-ROOT SUPPORT

The agent supports watching multiple session roots simultaneously via `AGENTCHRON_AGENT_ROOTS`. This allows capturing Claude, Codex, and Antigravity sessions from a single agent process:

```
agentchron-agent run \
  --sink-url http://<lab-host>:9475 \
  --sink-token "$AGENTCHRON_INGEST_TOKEN" \
  --agent-roots "claude=~/.claude/projects;codex=~/.codex/sessions;antigravity=~/Library/
Application Support/Antigravity/logs"
```

The `AGENTCHRON_AGENT_ROOTS` environment variable uses semicolon-separated `agent=path` entries. When set, the agent spawns one watcher task per root and collects errors via an unbounded mpsc channel:

```
// crates/agentchron-agent/src/main.rs

if roots.len() == 1 {
    // Single root: run directly
    return watcher::run(watcher::Config { /* ... */ }).await;
}

// Multi-root: spawn one task per root
let (err_tx, mut err_rx) = tokio::sync::mpsc::unbounded_channel::<String>();
for root in roots {
    let cfg = watcher::Config { /* ... */ };
    let err_tx = err_tx.clone();
    tokio::spawn(async move {
        if let Err(err) = watcher::run(cfg).await {
            let _ = err_tx.send(format!("{err:?}"));
        }
    });
}
drop(err_tx);

tokio::select! {
    maybe_error = err_rx.recv() => {
        if let Some(error) = maybe_error {
            anyhow::bail!("agentchron watcher failed: {error}");
        }
        Ok(())
    }
    _ = tokio::signal::ctrl_c() => {
        info!("ctrl-c received, shutting down multi-root watcher");
        Ok(())
    }
}
```

When `--agent-roots` is not set, the agent falls back to `--watch-root` (default `~/.claude/projects`) with `--agent` for identity. Agent identity is also inferred from the source path if not explicitly set: paths containing `/.claude/projects/` infer `claude`, `/.codex/sessions/` infer `codex`, and paths containing `antigravity` or `/.agy/` infer `antigravity`.

CONFIGURATION VIA ENVIRONMENT VARIABLES

The agent supports extensive configuration via environment variables, with command-line flags as overrides. This is important for systemd/LaunchAgent deployment where environment variables are set in unit files:

ENVIRONMENT VARIABLE	DEFAULT	PURPOSE
`AGENTCHRON_WATCH_ROOT`	`~/.claude/projects`	Root directory to watch recursively
`AGENTCHRON_AGENT`	(none)	Agent/client identity for the watch root
`AGENTCHRON_AGENT_ROOTS`	(none)	Multi-client watch roots (semicolon-separated `agent=path` entries)
`AGENTCHRON_SINK_URL`	(required)	Sink base URL (e.g., `http://<lab-host>:9475`)
`AGENTCHRON_SINK_TOKEN`	(required)	Bearer token for the sink ingest endpoint
`AGENTCHRON_STATE_DIR`	`~/.local/state/agentchron`	Local state directory (SQLite checkpoint)
`AGENTCHRON_HOST_ID`	(system hostname)	Logical host identifier for events
`AGENTCHRON_BATCH_SIZE`	64	Max events per push batch
`AGENTCHRON_DEBOUNCE_MS`	500	Push debounce window in milliseconds

The sink URL can point to either the web proxy (`http://<lab-host>:9475`) for off-host access, or the sink directly (`http://127.0.0.1:39474`) when running on the same host. The web proxy is the recommended path for off-host agents because it provides the single off-host entry point and handles authentication forwarding.

The `AGENTCHRON_HOST_ID` environment variable is important for fleet deployments where the system hostname may not be meaningful (e.g., Docker containers with random hostnames). Setting it to a logical identifier like `<dev-vm>` or `developer-laptop-01` makes the host field in events more useful for filtering and grouping.

The Watcher: Two-Phase Capture

The watcher (`crates/agentchron-agent/src/watcher.rs`) follows a two-phase pattern that ensures no session data is missed on startup while also providing live tailing of new activity.

PHASE 1: INITIAL SWEEP

On startup, the watcher recursively walks the watch root, collecting all supported files. A file is "supported" if it has a `.jsonl` extension, or if it has a `.log` extension and the path indicates an Antigravity session:

```
// crates/agentchron-agent/src/watcher.rs

fn is_supported_session_file(path: &Path) -> bool {
    let extension = path.extension()
        .and_then(|value| value.to_str())
        .map(str::to_ascii_lowercase);
    if extension.as_deref() == Some("jsonl") {
        return true;
    }
    if extension.as_deref() != Some("log") {
        return false;
    }
    infer_agent_from_source_path(&path.to_string_lossy())
        .as_deref()
        .map(|agent| agent == "antigravity")
        .unwrap_or(false)
}
```

Each discovered file is processed from its last checkpoint byte offset. This backfill ensures that sessions that existed before the watcher started are captured. The `backfill` subcommand runs only this phase and exits.

```
// Phase 1 — Initial sweep
let mut initial: Vec<PathBuf> = Vec::new();
walk_supported_files(&cfg.watch_root, &mut initial)?;
info!(count = initial.len(), "initial session/log files discovered");

for path in &initial {
    if let Err(e) = process_file(&cfg, path).await {
        warn!(file = %path.display(), error = ?e, "initial process failed");
    }
}

if cfg.backfill_only {
    info!("backfill complete; exiting (--backfill-only)");
    return Ok(());
}
```

PHASE 2: LIVE WATCH

After the initial sweep, the watcher creates a `notify` recommended watcher (fsevents on macOS, inotify on Linux) filtered to `Create` and `Modify(Data)` events:

```
// crates/agentchron-agent/src/watcher.rs

let (tx, mut rx) = mpsc::unbounded_channel::<PathBuf>();

let tx_inner = tx.clone();
let mut watcher = recommended_watcher(move |res: notify::Result<notify::Event>| {
    match res {
        Ok(evt) => {
            let is_data_change = matches!(
                evt.kind,
                EventKind::Create(_)
                    | EventKind::Modify(ModifyKind::Data(_))
                    | EventKind::Modify(ModifyKind::Any)
            );
            if !is_data_change { return; }
            for p in evt.paths {
                if is_supported_session_file(&p) {
                    let _ = tx_inner.send(p);
                }
            }
        }
        Err(e) => error!(error = ?e, "watcher error"),
    }
}).context("create fsevents watcher")?;

watcher.watch(&cfg.watch_root, RecursiveMode::Recursive)?;
```

Path events are sent to an unbounded mpsc channel. A debounced tick loop (default 500ms) drains pending paths and processes each one:

```
let debounce = Duration::from_millis(cfg.debounce_ms);
let mut pending: HashSet<PathBuf> = HashSet::new();
let mut tick = tokio::time::interval(debounce);
tick.set_missed_tick_behavior(tokio::time::MissedTickBehavior::Delay);

loop {
    tokio::select! {
        maybe_path = rx.recv() => {
            match maybe_path {
                Some(p) => { pending.insert(p); }
                None => break,
            }
        }
        _ = tick.tick() => {
            if pending.is_empty() { continue; }
            let to_process: Vec<PathBuf> = pending.drain().collect();
            for path in to_process {
                if let Err(e) = process_file(&cfg, &path).await {
                    warn!(file = %path.display(), error = ?e, "process failed");
                }
            }
        }
        _ = tokio::signal::ctrl_c() => {
            info!("ctrl-c received, shutting down");
            break;
        }
    }
}
```

The debounce window coalesces rapid file modification events (common when an agent writes multiple lines to a JSONL file in quick succession). The `MissedTickBehavior::Delay` setting prevents tick bursts after a slow period. The unbounded channel ensures no events are dropped even during processing spikes — at the cost of potentially unbounded memory if the agent cannot keep up.

`PROCESS_FILE()`

The `process_file()` function is where bytes become events. It seeks to the last checkpoint offset, reads new lines via `tokio::io::BufReader::lines()`, parses each line with `parse_line()`, and batches events up to `batch_size` (default 64):

```
// crates/agentchron-agent/src/watcher.rs

async fn process_file(cfg: &Config, path: &Path) -> Result<()> {
    let source_path = path.to_string_lossy().to_string();
    let start_offset = cfg.checkpoint.get_offset(&source_path)?;
    let metadata = tokio::fs::metadata(path).await?;
    let file_len = metadata.len();
    if file_len <= start_offset {
        return Ok(());  // Truncation or no growth
    }

    let mut file = tokio::fs::File::open(path).await?;
    file.seek(std::io::SeekFrom::Start(start_offset)).await?;
    let reader = BufReader::new(file);
    let mut lines = reader.lines();

    let mut batch: Vec<Event> = Vec::with_capacity(cfg.batch_size);
    let mut offset = start_offset;
    let mut last_uuid: Option<String> = None;

    while let Some(line) = lines.next_line().await? {
        let line_len = line.len() as u64 + 1; // +1 for newline
        let evt_offset = offset;
        offset += line_len;

        if line.trim().is_empty() { continue; }

        match parse_line(&line, evt_offset, &source_path, &cfg.host) {
            Ok(mut evt) => {
                if evt.agent.is_none() {
                    evt.agent = cfg.agent.clone()
                        .or_else(|| infer_agent_from_source_path(&source_path));
                }
                if let Some(u) = &evt.envelope.uuid {
                    last_uuid = Some(u.clone());
                }
                batch.push(evt);
            }
            Err(e) => {
                warn!(file = %source_path, byte_offset = evt_offset,
                      error = ?e, "parse failed; skipping line");
            }
        }

        if batch.len() >= cfg.batch_size {
            flush(cfg, &source_path, offset, last_uuid.as_deref(), &mut batch).await?;
        }
    }
```

```
    if !batch.is_empty() {
        flush(cfg, &source_path, offset, last_uuid.as_deref(), &mut batch).await?;
    } else if offset > start_offset {
        cfg.checkpoint.set_offset(&source_path, offset, last_uuid.as_deref())?;
    }
    Ok(())
}
```

Note the byte-offset tracking: each line's offset is its position within the source file. This offset travels with the event through the entire pipeline and is the key to replay safety — if the watcher restarts, it seeks to the last checkpoint offset and resumes exactly where it left off.

Checkpoint and Spool: At-Least-Once Delivery

The checkpoint is a SQLite database at `~/.local/state/agentchron/state.sqlite` with WAL mode and `synchronous=NORMAL`. It maintains two tables:

```
// crates/agentchron-agent/src/checkpoint.rs

conn.execute_batch(r#"
    CREATE TABLE IF NOT EXISTS file_offset (
        source_path TEXT PRIMARY KEY,
        byte_offset INTEGER NOT NULL,
        last_event_uuid TEXT,
        updated_at TEXT NOT NULL
    );
    CREATE TABLE IF NOT EXISTS spool (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        source_path TEXT NOT NULL,
        byte_offset INTEGER NOT NULL,
        payload TEXT NOT NULL,
        queued_at TEXT NOT NULL
    );
    CREATE INDEX IF NOT EXISTS idx_spool_source ON spool(source_path);
"#)?;
```

THE `FILE_OFFSET` TABLE

The `file_offset` table tracks the last successfully shipped byte offset per source file. `source_path` is the primary key — for local files, this is the absolute path; for remote files, it is `ssh://developer@<host><abs_path>`. The `last_event_uuid` column stores the UUID of the last event in the shipped batch, providing an additional deduplication signal.

On startup, `get_offset()` reads the last offset for each file. If no record exists, it returns 0 — the file is processed from the beginning.

THE `SPOOL` TABLE AND AT-LEAST-ONCE DELIVERY

The spool table is the durability mechanism for the agent side. When a push fails, events are serialized into the spool table and the checkpoint is **deliberately not advanced**:

```
// crates/agentchron-agent/src/watcher.rs

async fn flush(cfg: &Config, source_path: &str, new_offset: u64,
               last_uuid: Option<&str>, batch: &mut Vec<Event>) -> Result<()> {
    // Best-effort spool drain before pushing fresh events
    if let Err(e) = drain_spool(cfg).await {
        debug!(error = ?e, "spool drain skipped");
    }

    match cfg.pusher.send(batch).await {
        Ok(()) => {
            cfg.checkpoint.set_offset(source_path, new_offset, last_uuid)?;
            batch.clear();
        }
        Err(e) => {
            warn!(error = ?e, "push failed; spooling batch for retry");
            for evt in batch.drain(..) {
                let payload = serde_json::to_string(&evt)?;
                cfg.checkpoint.spool_event(source_path, evt.byte_offset, &payload)?;
            }
            // Checkpoint deliberately not advanced on failure;
            // spooled events get re-sent by drain_spool on next attempt.
        }
    }
    Ok(())
}
```

This is the at-least-once delivery guarantee. On the next `flush()` call, `drain_spool()` is called first to replay queued events:

```
// crates/agentchron-agent/src/watcher.rs

async fn drain_spool(cfg: &Config) -> Result<()> {
    loop {
        let rows = cfg.checkpoint.drain_spool(cfg.batch_size)?;
        if rows.is_empty() { return Ok(()); }
        let mut events: Vec<Event> = Vec::with_capacity(rows.len());
        let mut ids: Vec<i64> = Vec::with_capacity(rows.len());
        for (id, payload) in &rows {
            match serde_json::from_str::<Event>(payload) {
                Ok(evt) => { events.push(evt); ids.push(*id); }
                Err(e) => {
                    warn!(error = ?e, spool_id = id, "corrupt spool payload, dropping");
                    ids.push(*id);  // Still ack to remove corrupt entry
                }
            }
        }
        cfg.pusher.send(&events).await?;
        cfg.checkpoint.ack_spool(&ids)?;
        debug!(drained = ids.len(), "spool batch acked");
    }
}
```

Corrupt spool payloads are dropped with a warning but still acknowledged (removed from the spool table). This prevents a single corrupt entry from blocking the spool drain forever. The sink deduplicates by event UUID, so replayed events that were already received are safely ignored.

This combination — byte-offset checkpointing, spool on failure, drain before fresh push, UUID dedup at the sink — gives effectively-once semantics across restarts and network failures. The pattern is byte-offset checkpointing for replay safety (P3).

MUTEX-BASED CONCURRENCY (ANTI-PATTERN AP2)

The checkpoint uses `Mutex<Connection>` for SQLite access, which serializes all database operations:

```
// crates/agentchron-agent/src/checkpoint.rs

pub struct Checkpoint {
    conn: Mutex<Connection>,
}
```

For a single-watcher agent processing files sequentially, this is not a bottleneck — the mutex is held for microseconds per query. But for the multi-root watcher spawning concurrent tasks, all tasks share the same checkpoint database, and the mutex serializes their database access. Under high ingest load from multiple roots, this becomes a bottleneck. The mitigation is a connection pool (`r2d2`) or the planned PostgreSQL migration, though SQLite's own write locking limits parallelism regardless.

The Sanitizer's Role in the Agent

The agent is the first layer of defense-in-depth sanitization. The `parse_line()` function in `agentchron-core` runs `sanitizer::detect()` on the raw line before any JSON parsing, producing metadata-only `EventFinding` records. Then it sanitizes all string content recursively — envelope fields, message content, tool inputs, and extra fields — replacing matched secrets with `[REDACTED]`.

The sanitizer recognizes 18 secret patterns, ported from `securegit` and extended with AI-provider tokens:

PATTERN	RULE ID	EXAMPLE
GitHub classic PAT	`github-classic-pat`	`ghp_[A-Za-z0-9]{36,}`
GitHub OAuth token	`github-oauth-token`	`gho_[A-Za-z0-9]{36,}`
GitHub user token	`github-user-token`	`ghu_[A-Za-z0-9]{36,}`
GitHub server token	`github-server-token`	`ghs_[A-Za-z0-9]{36,}`
GitHub fine-grained PAT	`github-fine-grained-pat`	`github_pat_[A-Za-z0-9_]{22,}`
GitLab PAT	`gitlab-personal-access-token`	`glpat-[A-Za-z0-9\-_.]{20,}`
GitLab deploy token	`gitlab-deploy-token`	`gldt-[A-Za-z0-9\-_.]{20,}`
Authorization header	`authorization-header-token`	`Bearer [A-Za-z0-9\-_.]{20,}`
URL embedded credential	`url-embedded-credential`	`://user:pass@`
Anthropic API key	`anthropic-api-key`	`sk-ant-[A-Za-z0-9\-_]{20,}`
OpenAI API key	`openai-api-key`	`sk-(?:proj-)?[A-Za-z0-9_\-]{20,}`
ElevenLabs API key	`elevenlabs-api-key`	`sk_[A-Za-z0-9]{32,}`
Hugging Face token	`huggingface-token`	`hf_[A-Za-z0-9]{32,}`
AWS access key ID	`aws-access-key-id`	`AKIA[0-9A-Z]{16}`
Stripe secret key	`stripe-secret-key`	`sk_(?:test|live)_[A-Za-z0-9]{16,}`
Stripe webhook secret	`stripe-webhook-secret`	`whsec_[A-Za-z0-9]{16,}`
PEM private key block	`pem-private-key-block`	`-----BEGIN ... PRIVATE KEY-----`
JWT	`jwt`	`eyJ[A-Za-z0-9_-]{10,}\.eyJ...`

A shadowing rule prevents OpenAI's broad `sk-` prefix from double-classifying Anthropic keys:

```
// crates/agentchron-core/src/sanitizer.rs

fn is_shadowed_detection(rule_id: &str, matched: &str) -> bool {
    // OpenAI keys share the broad sk-* prefix. More specific provider
    // patterns should win when they match the same token.
    rule_id == "openai-api-key" && matched.starts_with("sk-ant-")
}
```

The `detect()` function returns `Vec<SecretDetection>` with rule_id, label, severity, occurrence_count, and advice — but **never the matched value**. The `sanitize()` function replaces all matches with `[REDACTED]`. The `sanitize_json_line()` function preserves JSON shape while redacting and stamps `agentchron_secret_filter` metadata (filter version + redactions array) onto the JSON object.

The filter version is `agentchron-secrets-filter-v1`, a constant that travels in findings metadata for rotation tracking. When a new pattern is added, the version will change, and findings from before and after the change will be distinguishable. This is the reproducibility envelope pattern applied to the sanitizer.

Push: HTTP Transport

The `Pusher` is a thin `reqwest` HTTP client. It sends `POST /v1/events` with a bearer token and a `Batch { events: &[Event] }` JSON body. The timeout is 30 seconds. On non-2xx responses, it returns an error with the status and body, which triggers the spool path in the watcher.

```
// crates/agentchron-agent/src/push.rs (conceptual)

pub struct Pusher {
    client: reqwest::Client,
    sink_url: String,
    token: String,
}

impl Pusher {
    pub fn new(sink_url: &str, token: &str) -> Result<Self> { /* ... */ }

    pub async fn send(&self, events: &[Event]) -> Result<()> {
        let resp = self.client
            .post(format!("{}/v1/events", self.sink_url))
            .bearer_auth(&self.token)
            .json(&Batch { events })
            .timeout(Duration::from_secs(30))
            .send().await?;
        if !resp.status().is_success() {
            let status = resp.status();
            let body = resp.text().await.unwrap_or_default();
            return Err(anyhow!("push failed: {status} {body}"));
        }
        Ok(())
    }
}
```

The pusher is cloned across watcher tasks in the multi-root case (it wraps an `Arc`-backed `reqwest::Client` internally). The batch size is 64 by default, configurable via `AGENTCHRON_BATCH_SIZE`.

The push path is the local-before-transport pattern (P2) in action: content is sanitized on the host where it originates, before any network transport occurs. The `parse_line()` function runs the sanitizer as part of parsing, so by the time events reach the pusher, all string content has been redacted. The sink performs a second pass regardless, because the principle is: **never trust the upstream to have done it correctly.**

Remote Pull: SSH Fleet Harvest

`pull-remote` is the SSH-based fleet harvester. It connects to each configured remote host as `developer@<host>` using default SSH keys and ssh-agent, lists `*.jsonl` files, and incrementally fetches new bytes from each file's last checkpoint.

The remote pull is the bootstrap/internal-fleet harvester. It is designed for a fleet of dev VMs where SSH access is already established. The productized customer path is TCP push (`agentchron-push`) — outbound JSONL only, no inbound SSH. But for internal fleet management, pull-remote provides a centralized capture mechanism that does not require installing the agent on every remote host.

THE HARVEST LOOP

For each host, the harvester:

1. Lists remote JSONL files via `find -printf`
2. Optionally mirrors raw files to an archive volume (archive-first)
3. Fetches new bytes via chunked `tail -c +N | head -c M`
4. Parses each line with `parse_line()` and batches to the pusher
5. Uses the same spool/drain/ack pattern as the local watcher

```
// crates/agentchron-agent/src/remote.rs

async fn pull_host(cfg: &PullConfig, host: &RemoteHost,
                   checkpoint: &Checkpoint, pusher: &Pusher) -> Result<()> {
    let files = list_remote_jsonl(host, &cfg.remote_root, &cfg.ssh_extra_args).await?;

    for (remote_path, remote_size) in files {
        // Archive-first: mirror raw file before ingest decisions
        if let Some(archive_root) = &cfg.archive_root {
            archive_remote_file(host, &remote_path, remote_size,
                                archive_root, &cfg.ssh_extra_args).await?;
        }

        let source_path = format!("ssh://{}{}", host.target(), remote_path);
        let last_offset = checkpoint.get_offset(&source_path)?;
        if remote_size <= last_offset { continue; }

        let bytes_to_fetch = remote_size - last_offset;
        let new_bytes = fetch_remote_bytes(host, &remote_path,
                                           last_offset, bytes_to_fetch,
                                           &cfg.ssh_extra_args).await?;
        // ... parse and batch ...
    }
    Ok(())
}
```

SSH CONFIGURATION

SSH connections use `BatchMode=yes`, `StrictHostKeyChecking=accept-new`, and a 10-second connect timeout. The SSH user is always `developer`:

```
// crates/agentchron-agent/src/remote.rs

impl RemoteHost {
    /// Always `developer@<host>` per project SSH conventions.
    pub fn target(&self) -> String {
        format!("developer@{}", self.host)
    }
}
```

The `--ssh-arg` flag allows passing extra SSH arguments (e.g., `-i /path/to/key` or `-p 2222`).

ARCHIVE-FIRST MIRRORING

The `--archive-root` option mirrors raw JSONL files to a local archive volume **before** ingest decisions. The archive path mirrors the host and absolute remote path:

```
<archive-root>/<host>/<absolute/source/path>
```

This preserves the original session corpus on a backup volume even if the remote host is lost. It also allows re-ingestion from the archive without SSH access to the original host. The archive fetch uses 8MB chunks:

```
// crates/agentchron-agent/src/remote.rs

const ARCHIVE_CHUNK_BYTES: u64 = 8 * 1024 * 1024;
```

Chunked fetching (`tail -c +N | head -c M`) avoids loading entire large files into memory. Each chunk is appended to the archive file.

NAMESPACED SOURCE PATHS

Remote source paths are namespaced as `ssh://developer@<host><abs_path>` so the checkpoint table cleanly disambiguates remote vs local files:

SOURCE PATH PATTERN	ORIGIN
`/home/user/.claude/projects/proj/session.jsonl`	Local file
`ssh://developer@<dev-vm>/home/developer/.claude/projects/proj/session.jsonl`	Remote file from

This namespacing prevents checkpoint collisions when the same absolute path exists on both the local host and a remote host. The checkpoint `file_offset` table uses `source_path` as its primary key, so each remote file gets its own offset tracking.

THE PULL LOOP

The pull loop runs on an interval (default 60 seconds) and calls `pull_once()` for each cycle. It handles `ctrl-c` for graceful shutdown:

```
// crates/agentchron-agent/src/remote.rs

pub async fn pull_loop(
    cfg: PullConfig,
    checkpoint: Checkpoint,
    pusher: Pusher,
    interval: Duration,
) -> Result<()> {
    info!(
        hosts = ?cfg.hosts.iter().map(|h| h.host.clone()).collect::<Vec<_>>(),
        remote_root = %cfg.remote_root,
        interval_s = interval.as_secs(),
        "starting remote pull loop"
    );

    let mut ticker = tokio::time::interval(interval);
    ticker.set_missed_tick_behavior(tokio::time::MissedTickBehavior::Delay);

    loop {
        tokio::select! {
            _ = ticker.tick() => {
                if let Err(e) = pull_once(&cfg, &checkpoint, &pusher).await {
                    warn!(error = ?e, "pull cycle failed");
                }
            }
            _ = tokio::signal::ctrl_c() => {
                info!("ctrl-c received, shutting down pull loop");
                break;
            }
        }
    }
    Ok(())
}
```

The `--once` flag runs a single pull cycle and exits, useful for cron jobs or testing. The `--interval-seconds` flag controls the loop period (default 60 seconds). The `MissedTickBehavior::Delay` setting prevents tick bursts after a slow pull cycle — if a pull takes longer than the interval, the next tick is delayed rather than fired immediately.

LISTING REMOTE FILES

The `list_remote_jsonl()` function runs `find` on the remote host to discover all JSONL files under the remote root:

```
# Effective command (simplified):
bash -lc 'find /home/developer/.claude/projects -type f -name "*.jsonl" -printf "%p\t%s\n"'
```

The `-printf "%p\t%s\n"` format outputs the path and file size separated by a tab, which the agent parses to determine which files have grown since the last checkpoint. The `bash -lc` wrapper enables tilde expansion on the remote root path.

FETCHING NEW BYTES

New bytes are fetched using `tail -c +N | head -c M`:

```
# Fetch bytes from offset 1024 to 2048 (1024 bytes)
tail -c +1025 /path/to/file.jsonl | head -c 1024
```

The `tail -c +N` starts reading from byte N (1-indexed, so +1025 means skip 1024 bytes). The `head -c M` limits the output to M bytes. This combination allows fetching an arbitrary byte range from a file without loading the entire file into memory.

The fetched bytes are split on newlines, and each line is parsed with `parse_line()`. The byte offset for each line is calculated from the checkpoint offset plus the cumulative bytes consumed. The same spool/drain/ack pattern as the local watcher ensures at-least-once delivery: if the push fails, events are spooled and the checkpoint is not advanced.

TCP Push Wire Protocol

The TCP push path (`crates/agentchron-agent/src/push_wire.rs`) is the productized customer shape: outbound JSONL only, local sanitizer before transport, same sink-side dedup/sanitize/index pipeline.

`PUSHCONFIG::FROM_ENV()`

The TCP push client reads its configuration from environment variables. The default port in code is 9478 (the container-internal port); in the lab deployment on `.114`, `AGENTCHRON_TCP_PORT` is set to 39478 (the host-visible port) so the push client connects to the Docker-mapped port:

```
// crates/agentchron-agent/src/push_wire.rs

impl PushConfig {
    pub fn from_env() -> Result<Self> {
        let token = env::var("AGENTCHRON_SINK_TOKEN")
            .or_else(|_| env::var("AGENTCHRON_INGEST_TOKEN"))
            .context("set AGENTCHRON_SINK_TOKEN or AGENTCHRON_INGEST_TOKEN")?;
        let tcp_port = env::var("AGENTCHRON_TCP_PORT")
            .ok().and_then(|v| v.parse::<u16>().ok())
            .unwrap_or(9478);

        Ok(Self {
            tcp_host: env::var("AGENTCHRON_TCP_HOST")
                .unwrap_or_else(|_| "127.0.0.1".to_string()),
            tcp_port,
            token,
            host_id: env::var("AGENTCHRON_HOST_ID")
                .unwrap_or_else(|| detect_hostname()),
            source_path: env::var("AGENTCHRON_SOURCE_PATH")
                .unwrap_or_else(|_| "agentchron-push://stdin".to_string()),
            agent: env::var("AGENTCHRON_AGENT").ok()
                .map(|v| v.trim().to_string())
                .filter(|v| !v.is_empty()),
            orca: OrcaMetadata {
                org: env_string("ORCA_ORG"),
                team: env_string("ORCA_TEAM"),
                workspace: env_string("ORCA_WORKSPACE"),
                visibility: env_string("ORCA_VISIBILITY")
                    .unwrap_or_else(|| "private".to_string()),
                // ... other scope fields ...
            },
            connect_timeout_ms: 5_000,
        })
    }
}
```

`PUSH_READER()`: THE WIRE PROTOCOL

The `push_reader()` function connects to the TCP sink, sends a JSON auth line, then reads lines from the reader, sanitizes each with `sanitize_json_line()`, and writes them to the TCP stream:

```
// crates/agentchron-agent/src/push_wire.rs

pub fn push_reader<R: BufRead>(config: &PushConfig, mut reader: R) -> Result<usize> {
    let addr = format!("{}:{}", config.tcp_host, config.tcp_port);
    let socket_addr = /* resolve addr */;
    let mut stream = TcpStream::connect_timeout(&socket_addr,
        Duration::from_millis(config.connect_timeout_ms))?;

    // Auth line
    let auth = json!({
        "type": "agentchron_auth",
        "token": config.token,
        "host": config.host_id,
        "source_path": config.source_path,
        "agent": config.agent,
        "orca": config.orca,
    });
    writeln!(stream, "{auth}")?;

    // Event stream: sanitize each line before transport
    let mut sent = 0usize;
    let mut line = String::new();
    loop {
        line.clear();
        let n = reader.read_line(&mut line)?;
        if n == 0 { break; }
        if line.trim().is_empty() { continue; }
        let sanitized = sanitize_json_line(line.trim_end_matches(['\r', '\n']));
        writeln!(stream, "{sanitized}")?;
        sent += 1;
    }
    stream.flush()?;
    Ok(sent)
}
```

The `sanitize_json_line()` function is the local-before-transport pattern in action for the TCP path. It preserves JSON shape while redacting secrets and stamps `agentchron_secret_filter` metadata (filter version + redactions array) onto the JSON object so downstream consumers know what was redacted.

`LOAD_ENV_FILE()` WITH 0600 PERMISSIONS

The `load_env_file()` helper reads `~/.config/agentchron/push.env` with standard KEY=VALUE parsing. It respects existing environment variables — only sets variables that are not already defined:

```
pub fn load_env_file() {
    let path = env::var_os("AGENTCHRON_ENV_FILE")
        .map(PathBuf::from)
        .or_else(|| home_dir().map(|home| home.join(".config/agentchron/push.env")));
    let Some(path) = path else { return; };
    let Ok(contents) = fs::read_to_string(path) else { return; };

    for line in contents.lines() {
        let line = line.trim();
        if line.is_empty() || line.starts_with('#') { continue; }
        let line = line.strip_prefix("export ").unwrap_or(line);
        let Some((key, value)) = line.split_once('=') else { continue; };
        let key = key.trim();
        if key.is_empty() || key.contains(char::is_whitespace) { continue; }
        if env::var_os(key).is_none() {
            env::set_var(key, unquote(value.trim()));
        }
    }
}
```

The env file should have `0600` permissions (owner read/write only) since it contains the bearer token. The function also supports `AGENTCHRON_ENV_FILE` to specify an alternate path.

Generic Ingest and Config Scanning

`INGEST` SUBCOMMAND

The `ingest` subcommand wraps arbitrary JSONL or text logs as sanitized Orca governance events. It reads from stdin or a file, builds `Event` structs with extensive governance metadata, sanitizes everything, and pushes in batches:

```
printf '%s\n' '{"message":"agent called prod deploy tool","tool_name":"deploy_release"}' \
  | AGENTCHRON_SINK_TOKEN="$AGE...KEN" \
    agentchron-agent ingest \
      --sink-url http://<lab-host>:9475 \
      --source mcp-gateway \
      --kind mcp-tool-call \
      --team Cybersecurity \
      --workspace agent2600 \
      --boundary prod \
      --rbac-decision warn
```

The `ingest` subcommand supports a rich set of governance metadata flags:

FLAG	ENVIRONMENT VARIABLE	PURPOSE
`--boundary`	`ORCA_BOUNDARY`	Security/governance boundary crossed (prod, customer-data, external-mcp)
`--policy-id`	`ORCA_POLICY_ID`	Policy id evaluated
`--authority-level`	`ORCA_AUTHORITY_LEVEL`	Hermes/ATCS authority level
`--signature-status`	`ORCA_SIGNATURE_STATUS`	Hermes identity verification status
`--agentshield-rule-id`	`ORCA_AGENTSHIELD_RULE_ID`	AgentShield rule id
`--rbac-subject`	`ORCA_RBAC_SUBJECT`	RBAC subject
`--rbac-action`	`ORCA_RBAC_ACTION`	RBAC action
`--rbac-resource`	`ORCA_RBAC_RESOURCE`	RBAC resource
`--rbac-decision`	`ORCA_RBAC_DECISION`	RBAC decision (allow, deny, warn, review)
`--contextos-attestation-id`	`ORCA_CONTEXTOS_ATTESTATION_ID`	ContextOS attestation id
`--mcp-server`	`ORCA_MCP_SERVER`	MCP server name
`--mcp-endpoint`	`ORCA_MCP_ENDPOINT`	MCP endpoint URL
`--tokens-input`	`ORCA_TOKENS_INPUT`	Input token count
`--tokens-output`	`ORCA_TOKENS_OUTPUT`	Output token count

The `build_generic_event()` function constructs an `Event` with all of this metadata embedded in the `extra` BTreeMap, prefixed with `orca_` keys. The event kind is set from `--kind` (default `orca-trace`). All string content is sanitized via `sanitizer::sanitize()` and `sanitize_json_value()`.

`SCAN-CONFIG` SUBCOMMAND

The `scan-config` subcommand scans AI client config files for embedded secrets and emits metadata-only findings. By default, it covers config paths for Claude, Codex, Gemini, Antigravity, Cursor, Claw/OpenClaw, and Hermes:

```
// crates/agentchron-agent/src/main.rs

fn default_ai_client_config_files() -> Vec<PathBuf> {
    vec![
        expand("~/.claude/settings.json"),
        expand("~/.claude.json"),
        expand("~/.codex/config.toml"),
        expand("~/.codex/config.json"),
        expand("~/.codex/mcp.json"),
        expand("~/.gemini/settings.json"),
        expand("~/.gemini/mcp.json"),
        expand("~/.agy/settings.json"),
        expand("~/.config/antigravity/config.json"),
        expand("~/.config/antigravity/mcp.json"),
        expand("~/.cursor/mcp.json"),
        expand("~/.claw/config.json"),
        expand("~/.hermes/config.json"),
        // ... and more
    ]
}
```

The scanner runs `sanitizer::detect()` on each config file's content. Findings are **metadata only** — they include rule_id, label, severity, occurrence_count, and advice, but never the matched secret value. The event's text describes the finding without revealing the secret:

```
let text = format!(
    "AI client config scan found {occurrence_count} secret occurrence(s) \
     in {client} config at {path_display}; rule_ids={}",
    rule_ids.join(",")
);
```

The `infer_config_client()` function determines which AI client a config file belongs to based on its path — `/.claude/` maps to "claude", `/.codex/` maps to "codex", `antigravity` maps to "antigravity", and so on. This client identification is stored in the event's `orca_config_client` extra field, enabling per-client finding aggregation in the web console.

The `--fail-on-finding` flag exits non-zero after emitting findings, useful for CI gates. The `--emit-clean` flag also emits a clean scan event for readable files with no findings, providing positive coverage evidence. The `--no-defaults` flag skips the built-in config paths, scanning only files specified with `--config-file`.

The `scan-config` subcommand addresses anti-pattern AP6 (direct vault/credential scraping). The principle is that agents should not `cat`, `tail`, `grep`, or bulk-read credential files directly — that bypasses attribution and risks dumping secrets into transcripts. Instead, `scan-config` provides a sanctioned scanning path that detects secrets in config files, emits metadata-only findings for rotation, and never includes the matched secret value in any output. The unit test explicitly verifies this:

```
#[test]
fn config_scan_event_reports_findings_without_secret_values() {
    let args = scan_args();
    let path = PathBuf::from("/tmp/test-home/.claude/settings.json");
    let bearer = "Bearer abcdefghijklmnopqrstuvwxyz123456";
    let content = format!(
        r#"{{"mcpServers":{{"bridge":{{"headers":{{"Authorization":"{bearer}"}}}}}}}}"#
    );

    let event = build_config_scan_event(&args, &path, "host-a", "scan-a", &content);
    let serialized = serde_json::to_string(&event).expect("serialize scan event");
    assert!(!serialized.contains(bearer));  // Secret never appears
    assert!(serialized.contains("authorization-header-token"));  // Rule ID does
}
```

The test confirms that the serialized event contains the rule ID (`authorization-header-token`) but not the secret bearer token. This is the metadata-only principle enforced by a test — if someone accidentally changes the code to include the matched value, the test fails.

Companion Binaries

The agent crate's `bin/` directory contains four companion binaries that compose with the main agent binary:

`AGENTCHRON-CLAUDE-HOOK`

Claude Code hook handler. Invoked by Claude Code's hook system on `SessionStart` , `PostToolUse` , and `Stop` events. It ingests session metadata, tool results, and final turn summaries into Orca via the same HTTP push path. This closes the gap between session creation and file watcher capture — hooks fire immediately, while the file watcher may have a 500ms debounce delay.

`AGENTCHRON-PUSH`

TCP push client. Reads JSONL from stdin, sanitizes each line, and streams to the TCP sink. This is the binary used in the `tail -F | agentchron-push` pipeline:

```
tail -F "$AGENTCHRON_SOURCE_PATH" | agentchron-push
```

It reads its configuration from environment variables (via `PushConfig::from_env()`) and optionally from `~/.config/agentchron/push.env` .

`AGENTCHRON-SANITIZE`

Standalone sanitizer. Reads from stdin, redacts secrets, writes to stdout. Useful for one-off sanitization tasks or for piping session content through a redaction filter without ingesting into Orca.

`ORCA-PANEL`

Panel review adapter. Connects the panel review process to the Orca workflow API. It takes panel review output and creates workflow candidates through `POST /v1/workflows/from-panel` , extracting consensus stance, risk, evidence event IDs, and reviewer information from the panel run JSON.

The panel review process is a key part of the feedback loop. A panel of model-provider agents critiques a session's direction, and if the consensus is to emit an advisory, the panel run is submitted to Orca as a workflow candidate. The `orca-panel` binary handles this submission, translating the panel run JSON into the workflow API's expected format.

The `doctor` Subcommand

The `doctor` subcommand is a diagnostic tool that prints the effective configuration and exits. It shows:

- Watch roots (with agent identity for each)
- State directory path
- Sink URL
- Host ID
- Whether each watch root exists on the filesystem
- Whether the state directory exists

```
$ agentchron-agent doctor --sink-url http://<lab-host>:9475 --sink-token "$TOKEN"
watch_roots =
  claude=/Users/developer/.claude/projects
state_dir   = /Users/developer/.local/state/agentchron
sink_url    = http://<lab-host>:9475
host        = dev-mac-01
watch_root_exists[claude] = true
state_dir_exists  = true
```

This is useful for debugging configuration issues — if a watch root does not exist, the agent will log a warning but continue running (waiting for the directory to be created). If the state directory does not exist, the agent will create it on startup. The `doctor` command helps verify that the configuration is correct before starting a long-running watcher.

The `ingest` Subcommand in Detail

The `ingest` subcommand is the gateway for non-Claude tools to feed governance events into Orca. It reads JSONL or plain text from stdin or a file, wraps each line as a sanitized Orca governance event with extensive metadata, and pushes in batches.

The `build_generic_event()` function constructs an `Event` from a raw line:

1. **Parse the payload** — JSONL format preserves the parsed JSON payload; text format wraps the raw string.
2. **Detect secrets** — Runs `sanitizer::detect()` on the raw line to produce metadata-only findings.
3. **Extract text** — Pulls text from common payload fields (`message` , `text` , `body` , `summary` , `event` , `action` , `decision` , `reason` , `outcome` , `endpoint` , `resource`).
4. **Sanitize** — Runs `sanitizer::sanitize()` on the extracted text and `sanitize_json_value()` on the payload.
5. **Extract metadata** — Pulls session ID, agent, tool name, and token counts from the payload if present.
6. **Build the event** — Constructs an `EventEnvelope` with the specified kind, stamps the `extra` BTreeMap with all `orca_*` metadata fields, and wraps it in an `Event` .

The resulting event carries all the governance metadata needed for the four-plane trust model: boundary, policy_id, authority_level, signature_status, agentshield_rule_id, RBAC fields, ContextOS attestation, MCP server/endpoint,

and token counts. This is how non-Claude tools (MCP gateways, CI/CD pipelines, custom SDKs) feed structured governance events into the same pipeline as Claude Code sessions.

The `--dry-run` flag prints the generated sanitized events to stdout instead of sending them, useful for testing and debugging without a running sink:

```
printf '%s\n' '{"message":"test event"}' | agentchron-agent ingest --dry-run --kind orca-trace
```

Patterns Developed Here

Three cross-cutting patterns are established in the agent chapter:

PATTERN 2: LOCAL-BEFORE-TRANSPORT

Content is sanitized on the host where it originates, before any network transport occurs. The watcher reads JSONL session files, sanitizes each line through `parse_line()`, and only then ships to the sink. The TCP push path sanitizes with `sanitize_json_line()` before writing to the TCP stream. The remote harvest path parses through `parse_line()` which includes sanitization. The sink performs a second pass regardless, because the principle is: **never trust the upstream to have done it correctly.**

PATTERN 3: BYTE-OFFSET CHECKPOINTING FOR REPLAY SAFETY

The agent tracks per-file byte offsets in a SQLite checkpoint database. On restart, the watcher seeks to the last offset and resumes. On push failure, events are spooled and the checkpoint is deliberately not advanced, ensuring at-least-once delivery. The sink deduplicates by event UUID. The TCP push path computes stable synthetic offsets via FNV-1a hashing. This combination gives effectively-once semantics across restarts, network failures, and multi-path ingest.

PATTERN 1: DEFENSE-IN-DEPTH SANITIZATION (FIRST LAYER)

The agent's `parse_line()` is the first layer of defense-in-depth sanitization. It runs `sanitizer::detect()` on the raw line to produce `EventFinding` records (metadata-only, no secret values), then sanitizes all string content recursively via `sanitize_envelope()`, `sanitize_message()`, and `sanitize_value()`. The sink's `ingest_one()` is the second layer, and the sink's plugin framework (which runs before the sanitizer) is the third. Findings are metadata-only — never the matched value.

Anti-Patterns to Address

ANTI-PATTERN 2: MUTEX-BASED SQLITE CONCURRENCY

The checkpoint uses `Mutex<Connection>` for SQLite access, serializing all database operations. For a single-watcher agent, this is not a bottleneck. But for the multi-root watcher spawning concurrent tasks, all tasks share the same checkpoint database, and the mutex serializes their access. The mitigation is a connection pool (`r2d2`) or the planned PostgreSQL migration, though SQLite's own write locking limits parallelism regardless. This anti-pattern also affects the sink's storage layer (Chapter 3).

ANTI-PATTERN 6: DIRECT VAULT/CREDENTIAL SCRAPING

The `scan-config` subcommand finds embedded secrets in AI client config files but must not dump them. The mitigation is that findings are metadata-only — rule ID, label, severity, occurrence count, and remediation advice — never the matched value. The unit test explicitly verifies that the serialized event does not contain the secret bearer token. The sanctioned path for credential access is through approved tools (MCP/read APIs, `securegit secret info`), not direct scraping.

Conclusion

`agentchron-agent` is a well-structured edge capture binary with clear separation of concerns: the watcher handles file watching, the checkpoint handles durability, the pusher handles transport, the remote module handles SSH harvest, and the push_wire module handles the TCP wire protocol. The two-phase capture pattern (initial sweep + live watch) ensures no data is missed on startup. The spool/ack pattern ensures at-least-once delivery across network failures. The local-before-transport principle ensures secrets never leave the host unredacted.

The next chapter covers `agentchron-sink` — the central ingest and storage service that receives events from the agent, runs plugin inspection and belt-and-suspenders sanitization, stores events in SQLite WAL as the source of truth, and writes to Neo4j and Qdrant as best-effort sidecars. The sink is where the edge-to-origin flow terminates and where the durable, searchable, graphable evidence store begins. It is the origin point of the entire governed data pipeline.

Chapter 3: AgentChron Sink

The Sink's Role as Origin

`agentchron-sink` is the central ingest and storage service. It runs on (`<lab-host>`) and is the origin of the edge-to-origin flow — the place where session evidence becomes durable, searchable, and graphable. The sink is an Axum HTTP server plus a Tokio TCP listener, backed by SQLite WAL as the source of truth, with Neo4j and Qdrant as best-effort sidecars.

The sink binds two listeners:

- **Axum HTTP server** on `0.0.0.0:39474` — the primary ingest and query API
- **Tokio TCP listener** on `0.0.0.0:39478` — the line-framed JSONL push receiver (host port 39478; container port 9478 when running in Docker, mapped via `39478:9478`)

The durability hierarchy is explicit: **SQLite must succeed; sidecars may fail.** If the SQLite insert fails, the event is not considered ingested. If Neo4j or Qdrant writes fail, the event is still in SQLite and ingest succeeds. This is the SQLite WAL as source of truth pattern (P5).

```
// crates/agentchron-sink/src/main.rs

pub struct AppState {
    pub token: String,
    pub storage: storage::Storage,
    pub graph: Option<graph::Graph>,
    pub vector: Option<vector::Vector>,
    pub plugins: plugins::PluginManager,
}
```

The `AppState` is wrapped in `Arc` and shared across all HTTP handlers and TCP connection tasks. The `graph` and `vector` fields are `Option` — they can be `None` if disabled via `AGENTCHRON_DISABLE_NEO4J` or `AGENTCHRON_DISABLE_QDRANT`. This allows running the sink without Neo4j or Qdrant for local development.

The HTTP API Surface

The sink's HTTP API is comprehensive, covering health checks, event ingest, session queries, search, workflows, graph context, and plugin findings:

ROUTE	METHOD	PURPOSE
`/v1/health`	GET	Health check
`/v1/stats`	GET	Global event/session/host/agent/token counts
`/v1/events`	POST	Batch event ingest
`/v1/events/:id`	GET	Full event detail by ID
`/v1/sessions`	GET	List all sessions
`/v1/sessions/:id/summary`	GET	Session summary with top tools and duration
`/v1/sessions/:id/stats`	GET	Token usage stats per session
`/v1/sessions/:id/live`	GET	Live session view (summary + alerts + events)
`/v1/sessions/:id/events`	GET	Paginated session events with boilerplate filter
`/v1/alerts`	GET	Alert list with filters
`/v1/search`	GET	FTS5 search with extensive filters
`/v1/workflows`	GET	Workflow candidate listing
`/v1/workflows/from-panel`	POST	Create workflow from panel review run
`/v1/workflows/:id`	GET	Single workflow
`/v1/workflows/:id/status`	POST	Update workflow status (candidate → approved → retired)
`/v1/graph/context`	GET	Graph context (Neo4j first, SQLite fallback)
`/v1/sources/coverage`	GET	Source file coverage verification
`/v1/plugins/findings`	GET	Plugin finding metadata

The route table is defined in `main.rs`:

```
// crates/agentchron-sink/src/main.rs

let app = Router::new()
    .route("/v1/health", get(ingest::health))
    .route("/v1/stats", get(ingest::global_stats))
    .route("/v1/events", post(ingest::ingest_batch))
    .route("/v1/events/:id", get(ingest::get_event))
    .route("/v1/sessions", get(ingest::list_sessions))
    .route("/v1/sessions/:id/summary", get(ingest::session_summary))
    .route("/v1/sessions/:id/stats", get(ingest::session_stats))
    .route("/v1/sessions/:id/live", get(ingest::session_live))
    .route("/v1/sessions/:id/events", get(ingest::list_session_events))
    .route("/v1/alerts", get(ingest::list_alerts))
    .route("/v1/search", get(ingest::search))
    .route("/v1/workflows", get(ingest::list_workflows))
    .route("/v1/workflows/from-panel", post(ingest::create_workflow_from_panel))
    .route("/v1/workflows/:id", get(ingest::get_workflow))
    .route("/v1/workflows/:id/status", post(ingest::update_workflow_status))
    .route("/v1/graph/context", get(ingest::graph_context))
    .route("/v1/sources/coverage", get(ingest::source_coverage))
    .route("/v1/plugins/findings", get(ingest::list_plugin_findings))
    .layer(DefaultBodyLimit::max(args.max_body_bytes))
    .layer(TraceLayer::new_for_http())
    .with_state(state);
```

The default body limit is 64MB (`DEFAULT_MAX_BODY_BYTES`), raised from Axum's 2MB default because session events can be large (file-history snapshots, big tool_use inputs). The web proxy can raise this further to 256MB for proxied ingest requests.

BEARER-TOKEN AUTHENTICATION

All API routes (except `/v1/health`) require bearer-token authentication. The `auth_ok()` function extracts the token from the `Authorization: Bearer <token>` header and validates it with `token_matches()`:

```
// crates/agentchron-sink/src/ingest.rs

fn auth_ok(headers: &HeaderMap, expected: &str) -> bool {
    let Some(auth) = headers.get("authorization").and_then(|v| v.to_str().ok()) else {
        return false;
    };
    let Some(token) = auth.strip_prefix("Bearer ") else {
        return false;
    };
    token_matches(token, expected)
}

pub fn token_matches(token: &str, expected: &str) -> bool {
    // Constant-time-ish compare. Lengths must match to fail-fast.
    if token.len() != expected.len() {
        return false;
    }
    let mut diff = 0u8;
    for (a, b) in token.bytes().zip(expected.bytes()) {
        diff |= a ^ b;
    }
    diff == 0
}
```

The `token_matches()` function is a constant-time-ish comparison. It first checks length equality (which does leak timing information, but only the length, not the content), then XORs all bytes and checks for zero diff. This avoids timing side-channels on bearer token validation — an attacker cannot distinguish "wrong character at position 3" from "wrong character at position 47" by measuring response time.

The same pattern is used in the web crate's `auth_ok()` function, ensuring consistent auth behavior across both services. This is a security hardening practice within Pattern 5 (SQLite WAL as Source of Truth) — the sink and web share a single durable auth model, and constant-time comparison prevents timing side-channels from leaking token bytes.

THE SINGLE SHARED TOKEN PROBLEM (AP1)

Currently, the sink uses one `AGENTCHRON_INGEST_TOKEN` for all auth roles — HTTP ingest, web proxy, TCP push auth, and all API queries. A compromised read token grants ingest and admin capabilities. The Cloudflare edge models split tokens (read/ingest/admin/origin) but the origin sink does not.

This is a P0 gap. The mitigation is to split auth into read, ingest, admin, origin, and agent-scoped tokens. Until this is implemented, the sink should not be exposed beyond a trusted network, and the web proxy on `:9475` is the only off-host entry point.

The Ingest Pipeline: `ingest_one()`

`ingest_one()` is the per-event pipeline. Every event — whether it arrives via HTTP, TCP push, or any future transport — goes through this same five-step pipeline:

```
// crates/agentchron-sink/src/ingest.rs

pub async fn ingest_one(state: &AppState, mut evt: Event) -> Result<()> {
    // 1. Plugin inspection — runs BEFORE the sink's sanitizer pass
    evt.findings = state.plugins.inspect(&evt);

    // 2. Belt-and-suspenders sanitize
    sanitize_event(&mut evt);

    // 3. SQLite insert — source of truth, must succeed
    let event_id = state.storage.insert_event(&evt).await?;

    // 4. Neo4j write — best-effort
    if let Some(g) = &state.graph {
        if let Err(e) = g.write_event(event_id, &evt).await {
            warn!(error = ?e, "neo4j write failed (event still in sqlite)");
        }
    }

    // 5. Qdrant write — best-effort (currently a stub)
    if let Some(v) = &state.vector {
        if let Err(e) = v.write_event(&evt).await {
            warn!(error = ?e, "qdrant write failed (event still in sqlite)");
        }
    }

    Ok(())
}
```

STEP 1: PLUGIN INSPECTION BEFORE SANITIZE

The plugin inspection runs **before** the sink's sanitizer pass. This is a deliberate security decision: direct clients that skip the agent parser (and thus skip the agent's local sanitization) cannot bypass secret reporting. The `SecretsFilterPlugin` re-runs `sanitizer::detect()` on event text and tool inputs, producing `EventFinding` records with `"detected during sink-side plugin scan"` summaries.

If the plugin ran after sanitization, the secrets would already be redacted to `[REDACTED]` and detection would find nothing. By running plugins before the sanitizer, the sink captures metadata about what was detected — rule ID, label, severity, occurrence count — without storing the secret value itself. Findings are metadata-only, always.

STEP 2: BELT-AND-SUSPENDERS SANITIZE

The `sanitize_event()` function re-sanitizes all text, envelope extras, message content, and tool inputs. Even if the agent already sanitized, the sink sanitizes again. This is the defense-in-depth pattern (P1) — the second layer:

```
// crates/agentchron-sink/src/ingest.rs

fn sanitize_event(evt: &mut Event) {
    if let Some(t) = evt.text.take() {
        evt.text = Some(sanitizer::sanitize(&t));
    }
    evt.envelope.extra = std::mem::take(&mut evt.envelope.extra)
        .into_iter()
        .map(|(key, value)| (key, sanitizer::sanitize_json_value(value)))
        .collect();
    if let Some(message) = evt.envelope.message.as_mut() {
        message.content = sanitizer::sanitize_json_value(message.content.take());
        message.extra = std::mem::take(&mut message.extra)
            .into_iter()
            .map(|(key, value)| (key, sanitizer::sanitize_json_value(value)))
            .collect();
    }
    for tool in &mut evt.tool_uses {
        tool.input = sanitizer::sanitize_json_value(tool.input.take());
    }
}
```

The `sanitize_json_value()` function recursively sanitizes all string values in a JSON tree — strings are redacted, arrays are mapped, objects are mapped, and other types (numbers, booleans, null) pass through unchanged.

STEP 3: SQLITE INSERT (SOURCE OF TRUTH)

The `state.storage.insert_event(&evt)` call is the source-of-truth write. This must succeed for the event to be considered ingested. If it fails, `ingest_one()` returns an error and the event is counted as rejected. The storage layer handles all SQLite complexity: WAL mode, FTS5 indexing, session summarization, and event detail extraction.

STEPS 4-5: BEST-EFFORT SIDECAR WRITES

Neo4j and Qdrant writes are best-effort. If they fail, the event is still in SQLite. The warning is logged but ingest succeeds. This is the synchronous enforcement, asynchronous telemetry pattern (P11) — sidecar failures don't block ingest.

The Neo4j write creates graph nodes and edges (Session → Event → Tool/Agent/Branch/File/Commit). The Qdrant write is currently a stub that logs a debug message and returns `Ok(())`. Both are wrapped in `Option` — if disabled via environment variables, they are `None` and the write is skipped entirely.

BATCH INGEST

The `ingest_batch()` HTTP handler iterates over events and calls `ingest_one()` for each:

```
// crates/agentchron-sink/src/ingest.rs

pub async fn ingest_batch(
    State(state): State<Arc<AppState>>,
    headers: HeaderMap,
    Json(batch): Json<IngestBatch>,
) -> impl IntoResponse {
    if !auth_ok(&headers, &state.token) {
        return (StatusCode::UNAUTHORIZED,
                Json(serde_json::json!({"error":"unauthorized"}))).into_response();
    }

    let mut accepted = 0usize;
    let mut rejected = 0usize;

    for evt in batch.events {
        match ingest_one(state.as_ref(), evt).await {
            Ok(_) => accepted += 1,
            Err(e) => {
                warn!(error = ?e, "event ingest failed");
                rejected += 1;
            }
        }
    }

    info!(accepted, rejected, "ingested batch");
    (StatusCode::OK, Json(IngestResponse { accepted, rejected })).into_response()
}
```

Events are processed sequentially in a single async task. Each `insert_event()` call locks the SQLite mutex, which serializes all database operations. This is the mutex-based SQLite concurrency anti-pattern (AP2) — under high ingest load from multiple agents or TCP connections, the mutex becomes a bottleneck. The mitigation is a connection pool or the planned PostgreSQL migration.

TCP Push Receiver

The TCP push listener (`crates/agentchron-sink/src/tcp_ingest.rs`) binds to `0.0.0.0:39478` (host port; the container-internal port is 9478 when deployed via Docker Compose, mapped as `39478:9478`) and runs in a spawned Tokio task. It accepts connections, authenticates the first line, and then streams events through the same `ingest_one()` path as HTTP.

THE PROTOCOL

1. **Auth line** — The first line is either text format (`AUTH <token> host=... source_path=... agent=...`) or JSON format (`{"type":"agentchron_auth","token":"***","host":"...","source_path":"...","agent":"...","orca":{...}}`). Token is validated with `token_matches()` .

2. **Event stream** — Each subsequent line is one JSONL event. Lines are parsed with `parse_line()`, stamped with the push context's agent and orca metadata, marked with `agentchron_push: true`, and sent through `ingest_one()`.
3. **Synthetic offsets** — Since TCP push doesn't know the original file byte offset, `stable_push_offset()` computes an FNV-1a hash.

AUTH LINE PARSING

The `parse_auth_line()` function handles both formats:

```
// crates/agentchron-sink/src/tcp_ingest.rs

fn parse_auth_line(line: &str, expected_token: &str) -> Result<PushContext> {
    if line.trim_start().starts_with('{') {
        // JSON format
        let auth: JsonAuth = serde_json::from_str(line)?;
        if auth.kind.as_deref() != Some("agentchron_auth") {
            return Err(anyhow!("json auth line has unexpected type"));
        }
        let token = auth.token.as_deref().unwrap_or_default();
        if !token_matches(token, expected_token) {
            return Err(anyhow!("unauthorized"));
        }
        return Ok(PushContext {
            host: auth.host.unwrap_or_else(|| "unknown-push-host".to_string()),
            source_path: auth.source_path.unwrap_or_else(|| "agentchron-push://
stdin".to_string()),
            agent: auth.agent,
            orca: auth.orca.unwrap_or_default(),
        });
    }

    // Text format: AUTH <token> host=... source_path=... agent=...
    let Some(rest) = line.strip_prefix("AUTH ") else {
        return Err(anyhow!("missing AUTH line"));
    };
    let mut parts = rest.split_whitespace();
    let token = parts.next().unwrap_or_default();
    if !token_matches(token, expected_token) {
        return Err(anyhow!("unauthorized"));
    }

    let mut host = "unknown-push-host".to_string();
    let mut source_path = "agentchron-push://stdin".to_string();
    let mut agent = None;
    let mut orca = OrcaMetadata::default();

    for part in parts {
        let Some((key, value)) = part.split_once('=') else { continue; };
        match key {
            "host" => host = value.to_string(),
            "source_path" => source_path = value.to_string(),
            "agent" => agent = Some(value.to_string()),
            "org" | "orca_org" => orca.org = Some(value.to_string()),
            "team" | "orca_team" => orca.team = Some(value.to_string()),
            "workspace" | "orca_workspace" => orca.workspace = Some(value.to_string()),
            "visibility" | "orca_visibility" => orca.visibility = value.to_string(),
            "purpose" | "orca_purpose" => orca.purpose = Some(value.to_string()),
            _ => debug!(field = %key, "ignoring unknown tcp auth field"),
```

```
        }
    }

    Ok(PushContext { host, source_path, agent, orca })
}
```

The `PushContext` established by the auth line stamps all subsequent events with host, source_path, agent, and orca metadata. This means a single TCP connection carries events from one source with one scope — if a client needs to push events from multiple sources or scopes, it opens multiple connections.

CONNECTION HANDLING

Each TCP connection is handled in a spawned Tokio task:

```
// crates/agentchron-sink/src/tcp_ingest.rs

pub async fn serve(bind: SocketAddr, state: Arc<AppState>) -> Result<()> {
    let listener = TcpListener::bind(bind).await?;
    info!(addr = %bind, "tcp push listener ready");

    loop {
        let (stream, peer) = listener.accept().await?;
        let state = state.clone();
        tokio::spawn(async move {
            if let Err(e) = handle_connection(stream, peer, state).await {
                warn!(peer = %peer, error = ?e, "tcp push connection failed");
            }
        });
    }
}
```

The `handle_connection()` function reads the auth line, then loops reading event lines. Each line is parsed, stamped with the push context, and sent through `ingest_one()`. Accepted and rejected counts are tracked and logged when the connection closes.

SYNTHETIC OFFSETS WITH FNV-1A

Since TCP push doesn't know the original file byte offset, `stable_push_offset()` computes a synthetic offset using FNV-1a hashing:

```
// crates/agentchron-sink/src/tcp_ingest.rs

fn stable_push_offset(source_path: &str, line: &str, line_index: u64) -> u64 {
    // FNV-1a gives a stable synthetic location for push streams.
    const OFFSET_BASIS: u64 = 0xcbf29ce484222325;
    const PRIME: u64 = 0x100000001b3;

    let mut hash = OFFSET_BASIS;
    for byte in source_path.as_bytes().iter()
        .chain([0xff].iter())
        .chain(line_index.to_le_bytes().iter())
        .chain([0xfe].iter())
        .chain(line.as_bytes().iter())
    {
        hash ^= *byte as u64;
        hash = hash.wrapping_mul(PRIME);
    }
    hash & (i64::MAX as u64)
}
```

The hash combines `source_path`, `line_index`, and `line` content to produce a stable synthetic offset. This keeps retries idempotent: if the same line is pushed again (e.g., after a network reconnection), it gets the same synthetic offset, and the sink can deduplicate. The `& (i64::MAX as u64)` mask ensures the result fits in a positive `i64` for SQLite storage.

The unit test confirms stability:

```
#[test]
fn offset_is_stable() {
    let a = stable_push_offset("/tmp/a.jsonl", r#"{"type":"user"}"#, 42);
    let b = stable_push_offset("/tmp/a.jsonl", r#"{"type":"user"}"#, 42);
    assert_eq!(a, b);
}
```

SQLite Storage Layer

The storage layer (`crates/agentchron-sink/src/storage.rs`) is the heart of the sink. It is a `Mutex<Connection>` SQLite database with extensive configuration for WAL mode, FTS5, and performance tuning.

CONFIGURATION

```
// crates/agentchron-sink/src/storage.rs

pub struct StorageConfig {
    pub cache_mb: usize,              // default 512, compose uses 2048
    pub mmap_mb: usize,               // default 0
    pub temp_store_memory: bool,      // default true
    pub wal_autocheckpoint_pages: usize, // default 8192
    pub busy_timeout_ms: u64,         // default 5000
}
```

The Docker Compose stack overrides defaults for RAM-backed performance:

```
# deploy/docker-compose.yml (excerpt)

environment:
  AGENTCHRON_SQLITE_CACHE_MB: "2048"
  AGENTCHRON_SQLITE_MMAP_MB: "8192"
  AGENTCHRON_SQLITE_TEMP_STORE_MEMORY: "true"
  AGENTCHRON_SQLITE_WAL_AUTOCHECKPOINT_PAGES: "8192"
  AGENTCHRON_SQLITE_BUSY_TIMEOUT_MS: "5000"
```

On `.114`, the compose defaults favor a 2GB RAM-backed SQLite page cache and 8GB mmap window while keeping the durable DB on disk. The `synchronous=NORMAL` setting trades a small durability window for significantly higher throughput — WAL mode ensures committed transactions are durable even with `NORMAL`.

TABLES

The storage layer maintains a rich schema:

TABLE	PURPOSE
`event`	Core event storage with all typed fields
`event_search_doc`	FTS5 external content table
`event_search_fts`	FTS5 virtual table with `unicode61` tokenizer
`tool_use`	Extracted tool calls with name and input
`token_usage`	Model token accounting per event
`plugin_finding`	Metadata-only findings from ingest plugins
`alert`	Security and governance alerts
`workflow_improvement`	Workflow candidates and approved rules
`session_summary`	Per-session KPI summaries

FTS5 indexes text, kind, cwd, git_branch, host, source_path, tool_names, and tool_inputs. The `unicode61` tokenizer provides Unicode-aware tokenization. Search supports snippet generation and source-path-fragment filtering.

`EventRow` AND THE UI RENDERER CONTRACT

The `EventRow` struct includes frontend-oriented visual fields that the storage layer populates from the event's `orca_payload` extra field:

```
// crates/agentchron-sink/src/storage.rs

#[derive(Serialize)]
pub struct EventRow {
    pub event_id: i64,
    pub uuid: Option<String>,
    pub kind: String,
    pub timestamp: Option<String>,
    // ... core fields ...

    /// Frontend-oriented event class for Operation Looking Glass style renderers.
    pub ui_kind: String,
    pub display_message: Option<String>,
    pub tool_call: Option<ToolCallVisual>,
    pub delegation: Option<DelegationVisual>,
    pub guardrail: Option<GuardrailVisual>,
    pub security_boundary: Option<SecurityBoundaryVisual>,
    pub session_summary: Option<SessionSummaryVisual>,
    pub token_usage: Option<TokenUsageVisual>,
    pub alert: Option<AlertVisual>,
}
```

The visual payloads provide structured data for the web UI to render without parsing raw event JSON:

VISUAL FIELD	CONTENTS
`tool_call`	tool_name, server, endpoint, latency, decision
`guardrail`	engine, rule_id, severity, verdict, trust_score, authority_level, signature_status, enforcement_mode
`security_boundary`	verdict, description, target, compliance_ref, boundary, policy_id
`delegation`	from_agent, to_agent, reason
`session_summary`	status classification, KPI counters, duration
`token_usage`	input, output, cache_read, cache_creation, thinking tokens
`alert`	severity, description, status

The `ui_kind` field provides a normalized event class for rendering (preferring the Orca payload's event type over the raw `kind`). This is the Operation Looking Glass renderer contract — the storage layer does the parsing work so the web UI can focus on presentation.

SESSION SUMMARIES WITH KPI COUNTERS

The `SessionSummary` struct provides per-session KPIs:

```
#[derive(Serialize)]
pub struct SessionSummary {
    pub session_id: String,
    pub event_count: i64,
    pub first_seen: String,
    pub last_seen: String,
    pub host: String,
    pub agent: Option<String>,
    pub team: Option<String>,
    pub workspace: Option<String>,
    pub visibility: String,
    pub tool_call_count: i64,
    pub guardrail_count: i64,
    pub security_boundary_count: i64,
    pub open_alert_count: i64,
    pub token_event_count: i64,
    pub total_tokens: i64,
}
```

These KPIs power the dashboard's session timeline and the per-session live view. The status classification (`ok` / `complete` / `needs_review` / `blocked` / `executing`) is derived from alert counts and event patterns.

GRAPH CONTEXT WITH SQLITE FALLBACK

The `graph_context()` method supports two modes: seed-based (from session_id or event_id) and entity-query-based (from a free-text query). It tries Neo4j first and falls back to a SQLite-based entity extraction and related-session scoring algorithm when Neo4j is unavailable.

The fallback algorithm extracts entities (file paths, commit SHAs, branch names, tool names) from event text and scores related sessions by shared entity counts. File paths and commits carry weight 4.0, branches 3.0, tools 2.0, and agents 0.75. This gives the platform graph-like query capabilities even when the Neo4j sidecar is down.

FTS5 SEARCH IMPLEMENTATION

The FTS5 search is the current production search implementation. It uses an external content table (`event_search_doc`) that references the `event` table, and a virtual table (`event_search_fts`) with the `unicode61` tokenizer:

```
-- event_search_doc: external content table referencing event
CREATE TABLE event_search_doc (
    event_id INTEGER PRIMARY KEY,
    text TEXT,
    kind TEXT,
    cwd TEXT,
    git_branch TEXT,
    host TEXT,
    source_path TEXT,
    tool_names TEXT,
    tool_inputs TEXT
);

-- event_search_fts: FTS5 virtual table with unicode61 tokenizer
CREATE VIRTUAL TABLE event_search_fts USING fts5(
    text, kind, cwd, git_branch, host, source_path, tool_names, tool_inputs,
    content='event_search_doc',
    tokenize='unicode61'
);
```

The search API (`GET /v1/search`) supports extensive filters: `q` (query text), `session_id` , `host` , `agent` , `team` , `workspace` , `visibility` , `purpose` , `promoted` , `source` (source path substring), `kind` , `tool` , `limit` , and `offset` . The `source` filter is particularly useful for isolating events from a specific transcript corpus — use `source=TechStack` to match events from the TechStack session without matching unrelated event text that happens to contain "TechStack".

Search results include snippet generation — FTS5's `snippet()` function highlights matching terms within the event text, providing context for the search hit. This is what the web console's evidence search page displays.

The source-path-fragment filtering is a practical detail: the `source` parameter does a substring match on `source_path` , not on event text. This means `source=TechStack` matches events from files whose path contains "TechStack", not events whose text mentions "TechStack". This distinction matters when you want to isolate a specific transcript corpus.

THE SESSION LIVE VIEW

The `/v1/sessions/:id/live` endpoint is the primary API for the web console's session scrubbing feature. It returns a `LiveResponse` envelope containing:

- A `SessionSummary` with status classification (`ok` / `complete` / `needs_review` / `blocked` / `executing`), KPI counters, and duration
- A list of alerts associated with the session
- A list of `EventRow` objects with visual payloads

The `non_boilerplate=true` query parameter filters out command-name-only events (events that contain only a tool name without meaningful content). This reduces noise in the session timeline — the operator sees the events that matter, not every low-level file history snapshot.

The status classification is derived from alert counts and event patterns: - `ok` — no alerts, session completed normally - `complete` — session completed with all expected event types - `needs_review` — alerts or security boundaries present - `blocked` — enforcement actions blocked tool calls - `executing` — session is still active (events arriving recently)

Graph Writer

The Neo4j writer (`crates/agentchron-sink/src/graph.rs`) creates a graph of session and event nodes with edges to entities:

```
(:Session)-[:HAS]->(:Event)
(:Event)-[:PARENT]->(:Event)      (when parent_uuid present)
(:Event)-[:USED]->(:Tool {name})
(:Event)-[:BY_AGENT]->(:Agent {name})
(:Event)-[:ON_BRANCH]->(:Branch {name})
(:Event)-[:TOUCHED]->(:File {path})
(:Event)-[:REFERENCES_COMMIT]->(:Commit {sha})
```

INDEX CREATION

On connection, the graph writer creates indexes for all node labels:

```
// crates/agentchron-sink/src/graph.rs

pub async fn connect(uri: &str, user: &str, password: &str) -> Result<Self> {
    let graph = Neo4jGraph::new(uri, user, password).await?;
    for statement in [
        "CREATE INDEX session_id IF NOT EXISTS FOR (s:Session) ON (s.session_id)",
        "CREATE INDEX event_uuid IF NOT EXISTS FOR (e:Event) ON (e.uuid)",
        "CREATE INDEX event_id IF NOT EXISTS FOR (e:Event) ON (e.event_id)",
        "CREATE INDEX tool_name IF NOT EXISTS FOR (t:Tool) ON (t.name)",
        "CREATE INDEX agent_name IF NOT EXISTS FOR (a:Agent) ON (a.name)",
        "CREATE INDEX branch_name IF NOT EXISTS FOR (b:Branch) ON (b.name)",
        "CREATE INDEX file_path IF NOT EXISTS FOR (f:File) ON (f.path)",
        "CREATE INDEX commit_sha IF NOT EXISTS FOR (c:Commit) ON (c.sha)",
    ] {
        let _ = graph.run(query(statement)).await;
    }
    Ok(Self { inner: graph })
}
```

ENTITY EXTRACTION

The graph writer extracts entities from event content:

- **File paths** — Extracted from tool inputs by looking for `file_path` , `notebook_path` , `path` , and `*_path` keys with known source extensions.
- **Commit SHAs** — Extracted from event text using a regex for 7-40 hex characters with at least one `a-f` character. The `a-f` requirement avoids matching numeric false positives like timestamps or IDs.
- **Common tools** (Bash, Read, Write, Edit, etc.) are filtered out of entity queries to reduce noise.

The `graph_context()` method returns `GraphEntity` objects with weighted scores and `RelatedSession` objects with shared entity counts and sample event IDs. The response includes `GraphContextLink` objects connecting entities to related sessions.

BEST-EFFORT WRITES

Graph writes are best-effort. If `write_event()` fails, the warning is logged but the event is still in SQLite:

```
// crates/agentchron-sink/src/ingest.rs

if let Some(g) = &state.graph {
    if let Err(e) = g.write_event(event_id, &evt).await {
        warn!(error = ?e, "neo4j write failed (event still in sqlite)");
    }
}
```

This is the synchronous enforcement, asynchronous telemetry pattern (P11). The graph is a query acceleration layer, not a source of truth. If it fails, the platform degrades gracefully — `graph_context()` falls back to SQLite-based entity extraction.

Plugin Framework

The plugin framework (`crates/agentchron-sink/src/plugins.rs`) is minimal but deliberate. It provides an `IngestPlugin` trait and a `PluginManager` configured via CSV.

THE `INGESTPLUGIN` TRAIT

```
// crates/agentchron-sink/src/plugins.rs

pub trait IngestPlugin: Send + Sync {
    fn name(&self) -> &'static str;
    fn inspect(&self, event: &Event) -> Vec<EventFinding>;
}
```

The trait has two methods: `name()` returns the plugin identifier, and `inspect()` returns findings for a given event. Findings are metadata-only — they must never include the matched secret or any reversible representation.

`PLUGINMANAGER`

The `PluginManager` is configured via the `AGENTCHRON_PLUGINS` environment variable (CSV format, default `secrets`):

```
// crates/agentchron-sink/src/plugins.rs

pub struct PluginManager {
    plugins: Vec<Box<dyn IngestPlugin>>,
}

impl PluginManager {
    pub fn from_csv(csv: &str) -> Self {
        let mut plugins: Vec<Box<dyn IngestPlugin>> = Vec::new();
        for name in csv.split(',').map(str::trim).filter(|s| !s.is_empty()) {
            match name {
                "secrets" | "secrets-filter" =>
plugins.push(Box::new(SecretsFilterPlugin)),
                "none" | "off" => {}
                _ => {}
            }
        }
        Self { plugins }
    }

    pub fn inspect(&self, event: &Event) -> Vec<EventFinding> {
        let mut findings = event.findings.clone();
        for plugin in &self.plugins {
            findings.extend(plugin.inspect(event));
        }
        dedupe_findings(findings)
    }
}
```

The `inspect()` method first clones the event's existing findings (from the agent-side sanitizer), then extends with plugin findings. Results are deduplicated by `(plugin, rule_id, category)` with occurrence counts summed.

SECRETSFILTERPLUGIN

The `SecretsFilterPlugin` re-runs `sanitizer::detect()` on event text and tool inputs:

```
// crates/agentchron-sink/src/plugins.rs

pub struct SecretsFilterPlugin;

impl IngestPlugin for SecretsFilterPlugin {
    fn name(&self) -> &'static str {
        "secrets-filter"
    }

    fn inspect(&self, event: &Event) -> Vec<EventFinding> {
        let mut out = Vec::new();
        if let Some(text) = event.text.as_deref() {
            out.extend(secret_findings_from_text(text));
        }
        for tool in &event.tool_uses {
            out.extend(secret_findings_from_text(&tool.input.to_string()));
        }
        out
    }
}

fn secret_findings_from_text(text: &str) -> Vec<EventFinding> {
    sanitizer::detect(text)
        .into_iter()
        .map(|d| EventFinding {
            plugin: "secrets-filter".to_string(),
            rule_id: d.rule_id,
            category: "secret".to_string(),
            severity: d.severity,
            summary: format!("{} detected during sink-side plugin scan", d.label),
            advice: d.advice,
            occurrence_count: d.occurrence_count,
        })
        .collect()
}
```

The plugin runs before the sink's sanitizer pass (in `ingest_one()`, step 1 before step 2). This means it sees the raw, unredacted event content. It produces findings with rule ID, label, severity, occurrence count, and advice — but never the matched secret value. After the plugin runs, the sanitizer redacts the secrets to `[REDACTED]`, and the event is stored with both the findings (metadata) and the redacted content.

FINDING DEDUPLICATION

```
fn dedupe_findings(findings: Vec<EventFinding>) -> Vec<EventFinding> {
    let mut out: Vec<EventFinding> = Vec::new();
    for finding in findings {
        if let Some(existing) = out.iter_mut().find(|existing| {
            existing.plugin == finding.plugin
                && existing.rule_id == finding.rule_id
                && existing.category == finding.category
        }) {
            existing.occurrence_count += finding.occurrence_count;
        } else {
            out.push(finding);
        }
    }
    out
}
```

Findings from the agent-side sanitizer and the sink-side plugin are deduplicated by `(plugin, rule_id, category)` with occurrence counts summed. If the agent detected 3 GitHub PATs and the sink plugin detected 2 more, the final finding shows 5 occurrences of `github-classic-pat` from `secrets-filter`.

THE WORKFLOW API

The workflow API is the promotion path from observation to enforcement. Workflow candidates are created from panel review runs and promoted through a lifecycle: `candidate → approved → retired`.

The `POST /v1/workflows/from-panel` endpoint creates a workflow from a panel review run. The request body includes the panel run JSON, which must contain a `consensus` object with `emit_advisory=true`. The handler extracts:

- **Stance** — The panel's recommendation (e.g., `warn`, `block`, `allow`)
- **Risk** — The assessed risk level (e.g., `low`, `medium`, `high`)
- **Evidence event IDs** — The events that informed the panel's decision
- **Reviewer IDs** — The reviewers who agreed with the consensus
- **Findings and recommendations** — Structured analysis from the panel

The `POST /v1/workflows/:id/status` endpoint updates the workflow status. The lifecycle is:

```
candidate → approved → retired
```

A candidate is a proposed workflow improvement discovered during session review. An approved workflow is one that has passed review and is ready to be fed back into AgentShield as a runtime rule. A retired workflow is one that is no longer active.

This is the feedback loop of the four-plane trust model: Orca observes → reviewer creates workflow candidate → panel reviews → workflow approved → AgentShield enforces the new rule. The loop closes the circle from observation back to enforcement.

SOURCE COVERAGE VERIFICATION

The `GET /v1/sources/coverage` endpoint verifies that all source files are being captured. It compares the files known to the checkpoint database (via the sink's event records) against the expected source file inventory. This is useful for:

- Detecting when a new session root has been added but the agent is not watching it
- Identifying files that exist on the host but have no events in the sink
- Verifying that remote harvest is covering all expected hosts
- Auditing capture completeness across a fleet

The response includes per-source-file statistics: event count, first and last event timestamps, byte offset coverage, and host information. Files with zero events or incomplete coverage are flagged for investigation.

Vector Writer and Auth

QDRANT VECTOR WRITER (STUB)

The Qdrant writer is explicitly a stub. The client is connected but `write_event()` only logs a debug message and returns `Ok(())`:

```
// crates/agentchron-sink/src/vector.rs (conceptual)

impl Vector {
    pub async fn write_event(&self, evt: &Event) -> Result<()> {
        // Embedding model wiring lands in v0.2.
        // Once an embedder is configured, text_embedding(evt.text) will be pushed here.
        debug!(event_uuid = ?evt.envelope.uuid, "qdrant write_event (stub)");
        Ok(())
    }
}
```

The code comment is honest: "embedding model wiring lands in v0.2." The risk (AP3) is that the presence of a Qdrant container in the compose stack and a `Vector` struct in the sink could mislead operators into assuming semantic search is functional. It is not. The stub should be clearly labeled in deployment docs and the dashboard until wired.

The Qdrant container can remain in the compose stack for when embeddings are ready — it is a running, connected service that just doesn't receive any writes yet. When an embedder is configured, `text_embedding(evt.text)` will be pushed to Qdrant as a vector with the event's metadata as payload.

CONSTANT-TIME TOKEN COMPARISON

Both the sink (`ingest.rs::token_matches()`) and the web (`main.rs::auth_ok()`) use the same constant-time-ish comparison for bearer token validation:

```
pub fn token_matches(token: &str, expected: &str) -> bool {
    // Constant-time-ish compare. Lengths must match to fail-fast.
    if token.len() != expected.len() {
        return false;
    }
    let mut diff = 0u8;
    for (a, b) in token.bytes().zip(expected.bytes()) {
        diff |= a ^ b;
    }
    diff == 0
}
```

The length check does leak timing information (an attacker can determine the token length by measuring response time), but only the length, not the content. The byte-by-byte XOR with accumulation into a single `diff` variable ensures that the comparison time is independent of which byte position differs. An attacker cannot distinguish "wrong character at position 3" from "wrong character at position 47."

This is not a perfect constant-time comparison (which would require equal-length comparison or padding), but it is a significant improvement over a naive `==` comparison, which typically returns false at the first differing byte and leaks the position of the first difference. The pattern is deliberately duplicated in both the sink and the web crate (rather than shared via a common dependency) to maintain the clean service boundary — the web crate does not depend on the sink crate.

THE TCP PUSH RECEIVER'S SECURITY MODEL

The TCP push receiver is bearer-token authenticated but not encrypted by itself. The README is explicit about this:

> The TCP push receiver is bearer-token authed but not encrypted by itself. Use it directly only on trusted LANs, or wrap the same line protocol with SSH tunneling, stunnel, or `ncat --ssl`.

For trusted LAN deployments (like the `.114` lab), the TCP push receiver is bound to `127.0.0.1` and is not exposed off-host. For customer deployments, the recommended path is to wrap the TCP connection with SSH tunneling, stunnel, or `ncat --ssl` to provide transport encryption. The line protocol is the same either way — only the transport differs.

The auth line is the first line sent on the connection. If authentication fails, the connection is dropped immediately. If authentication succeeds, a `PushContext` is established that stamps all subsequent events with host, source_path, agent, and orca metadata. A single TCP connection carries events from one source with one scope — if a client needs to push events from multiple sources or scopes, it opens multiple connections.

The MCP Bridge

The MCP (Model Context Protocol) bridge is a stdio-over-HTTP server that allows MCP-compatible clients to query the Orca sink without direct database access. It is implemented as a Python script at `deploy/bin/agentchron-mcp.py` and exposes MCP tools that wrap the sink's HTTP API.

The MCP bridge is the integration point for AI agents that want to query Orca's evidence store. An agent can use MCP tools to:

- Search for events matching a query
- Get session summaries and live views

- Retrieve graph context for a session or entity
- List workflow candidates and approved rules
- Get source coverage information

The bridge translates MCP tool calls into HTTP requests against the sink's `/v1/*` API, authenticating with the sink bearer token. It does not have direct database access — all queries go through the same HTTP API that the web console and other clients use. This maintains the security boundary: the sink is the single point of access control, and the MCP bridge is just another HTTP client.

The MCP bridge is particularly important for the Context Block API vision (`GET /v1/topics/:topic/context` and the `orca_context_block` MCP tool). This is the planned GraphRAG read primitive that will allow agents to retrieve curated context blocks from the gold dataset during sessions. In v0.1, the MCP bridge provides search and session query capabilities; in v0.2 and beyond, it will provide context block retrieval.

SQLite Performance Tuning

The sink's SQLite configuration is tuned for the lab environment — RAM-backed page cache, memory temp store, and WAL auto-checkpointing:

```
// crates/agentchron-sink/src/storage.rs

pub struct StorageConfig {
    pub cache_mb: usize,                // default 512, compose uses 2048
    pub mmap_mb: usize,                 // default 0, compose uses 8192
    pub temp_store_memory: bool,        // default true
    pub wal_autocheckpoint_pages: usize, // default 8192
    pub busy_timeout_ms: u64,           // default 5000
}
```

The Docker Compose stack overrides defaults for the `.114` lab environment:

SETTING	DEFAULT	COMPOSE VALUE	RATIONALE
`AGENTCHRON_SQLITE_CACHE_MB`	512	2048	2GB RAM-backed page cache for faster reads
`AGENTCHRON_SQLITE_MMAP_MB`	0	8192	8GB mmap window for large database reads
`AGENTCHRON_SQLITE_TEMP_STORE_MEMORY`	true	true	Keep temp tables in memory, not on disk
`AGENTCHRON_SQLITE_WAL_AUTOCHECKPOINT_PAGES`	8192	8192	Checkpoint WAL every 8192 pages (~33MB)
`AGENTCHRON_SQLITE_BUSY_TIMEOUT_MS`	5000	5000	Wait 5 seconds for lock before timing out

The `synchronous=NORMAL` setting (set in the `open()` function) trades a small durability window for significantly higher throughput. In WAL mode, committed transactions are durable even with `NORMAL` — the WAL file is flushed to disk on checkpoint, not on every commit. This is acceptable for the sink because the WAL is checkpointed regularly and the database file is on a durable volume (not tmpfs).

The `busy_timeout_ms` setting controls how long SQLite waits for a lock before returning `SQLITE_BUSY`. With `Mutex<Connection>`, only one task can access the database at a time, so the busy timeout is rarely needed — the mutex serializes access before SQLite's own locking comes into play. The 5-second timeout is a safety net for edge cases where the mutex is held for an extended period.

Patterns Developed Here

Three cross-cutting patterns are established in the sink chapter:

PATTERN 5: SQLITE WAL AS SOURCE OF TRUTH WITH BEST-EFFORT SIDECARS

SQLite WAL is the durable source of truth. Neo4j and Qdrant are sidecars whose failures are logged but do not block ingest. The durability hierarchy is explicit: SQLite insert must succeed; Neo4j/Qdrant writes are best-effort. The `graph_context()` API tries Neo4j first and falls back to a SQLite-based entity extraction and related-session scoring algorithm. This gives the platform a single durable store that can be backed up with standard SQLite tooling, while allowing graph and vector capabilities to degrade gracefully.

PATTERN 1: DEFENSE-IN-DEPTH SANITIZATION (SINK LAYER)

The sink's `ingest_one()` is the second and third layers of defense-in-depth sanitization. The `SecretsFilterPlugin` runs **before** the sanitizer pass (third layer — plugin before sanitize), and `sanitize_event()` re-sanitizes all content (second layer — belt-and-suspenders). The plugin-before-sanitize ordering is critical: it ensures direct clients that skip the agent parser cannot bypass secret reporting. Findings are metadata-only — never the matched value.

PATTERN 11: SYNCHRONOUS ENFORCEMENT, ASYNCHRONOUS TELEMETRY

The sink's best-effort sidecar writes embody this pattern. SQLite insert is synchronous — it must succeed for the event to be considered ingested. Neo4j and Qdrant writes are asynchronous telemetry — they can fail without data loss. The warning is logged, but the event is in SQLite. This keeps the ingest hot path fast (SQLite insert + sanitize) while allowing graph and vector capabilities to lag or fail without blocking.

Anti-Patterns to Address

ANTI-PATTERN 2: MUTEX-BASED SQLITE CONCURRENCY

The sink's storage layer uses `Mutex<Connection>` for SQLite access:

```
// crates/agentchron-sink/src/storage.rs

pub struct Storage {
    conn: Mutex<Connection>,
}
```

This serializes all database operations. Under high ingest load from multiple agents or TCP connections, each `insert_event()` call locks the mutex, and other tasks must wait. The `ingest_batch()` handler processes events sequentially in a single async task, and each event's `insert_event()` call acquires the mutex.

The mitigation is a connection pool (`r2d2`) or the planned PostgreSQL migration. However, SQLite's own write locking limits parallelism — even with a connection pool, only one writer can hold the write lock at a time. The future multi-writer path is a storage-layer migration to PostgreSQL, not a config change.

ANTI-PATTERN 1: SINGLE SHARED TOKEN FOR ALL AUTH ROLES

The sink uses one `AGENTCHRON_INGEST_TOKEN` for HTTP ingest, web proxy, TCP push auth, and all API queries. A compromised read token currently grants ingest and admin capabilities. The Cloudflare edge models split tokens but the origin sink does not. This is a P0 gap with a well-specified implementation path: split into read, ingest, admin, origin, and agent-scoped tokens.

ANTI-PATTERN 3: QDRANT VECTOR WRITER IS A DEAD STUB

The Qdrant client is connected but `write_event()` does nothing. The presence of a Qdrant container in the compose stack could mislead operators into assuming semantic search is functional. The code comment is honest ("embedding model wiring lands in v0.2"), but the stub should be clearly labeled in deployment docs and the dashboard until wired.

Conclusion

`agentchron-sink` is a well-structured central ingest and storage service. The Axum HTTP API provides a comprehensive surface for event ingest, session queries, search, workflows, and graph context. The `ingest_one()` pipeline is the convergence point for all capture paths — HTTP, TCP push, and any future transport — with its five-step flow: plugin inspection, belt-and-suspenders sanitize, SQLite insert, Neo4j write, Qdrant write. The durability hierarchy (SQLite must succeed, sidecars may fail) ensures no data loss while allowing graph and vector capabilities to degrade gracefully.

The SQLite storage layer with WAL mode, FTS5 search, and rich visual payloads for the web UI is the source of truth and the query engine. The Neo4j graph writer provides relationship queries with a SQLite fallback. The plugin framework provides extensible pre-storage inspection with the `SecretsFilterPlugin` as the first implementation. The constant-time token comparison avoids timing side-channels on auth.

The key technical debts are the single shared token (AP1), mutex-based SQLite concurrency (AP2), and the Qdrant stub (AP3). All three are well-specified with clear implementation paths. The next chapter covers `agentchron-web` — the server-rendered UI, session scrubbing with visual payloads, the context graph explorer, and the BFF proxy pattern.

Chapter 4: AgentChron Web

Architecture: Store-Nothing BFF

AgentChron Web is the operator-facing surface of the Orca platform. It is a server-rendered UI and a Backend-for-Frontend (BFF) proxy built with Axum and Askama templates. Its defining architectural constraint is stated in the first line of its source:

```
//! agentchron-web — thin server-rendered UI over the sink's REST API.
//!
//! All data is fetched from agentchron-sink; this crate stores nothing.
```

That single comment encodes the entire design philosophy. The web crate has no database, no SQLite handle, no Neo4j client, no Qdrant client. It does not even depend on the `agentchron-sink` crate as a library dependency. Every byte of data it renders or proxies is fetched from the sink over HTTP using a bearer token. This is a deliberate service boundary: the web process can be restarted, scaled horizontally, or deployed on a different host without touching the durable store.

The crate's dependency on `agentchron-core` is minimal — it uses only `OrcaMetadata` (for the push_wire module) and `sanitize_json_line`. The sink is treated as an opaque HTTP service identified by a URL and a token:

```
struct AppState {
    sink_url: String,
    sink_token: String,
    http: reqwest::Client,
}
```

Three environment variables configure the process:

VARIABLE	DEFAULT	PURPOSE
`AGENTCHRON_WEB_BIND`	`0.0.0.0:9475`	Bind address for the HTTP listener
`AGENTCHRON_SINK_URL`	`http://sink:9474`	Internal sink API URL
`AGENTCHRON_SINK_TOKEN`	(required)	Bearer token for sink authentication
`AGENTCHRON_WEB_MAX_BODY_BYTES`	`268435456` (256 MB)	Max body for proxied ingest requests

The route table reveals the full surface area of the application:

```
let app = Router::new()
    .route("/", get(index))
    .route("/search", get(search_view))
    .route("/workflows", get(workflows_view))
    .route("/sessions/:id", get(session_view))
    .route("/graph/:id", get(graph_view))
    .route("/api/graph/:id", get(api_graph))
    .route("/static/vendor/highlight.min.js", get(vendor_highlight_js))
    .route("/static/vendor/atom-one-dark.min.css", get(vendor_highlight_css))
    .route("/static/vendor/marked.min.js", get(vendor_marked_js))
    .route("/v1/health", get(proxy_health))
    .route("/v1/*path", any(proxy_v1))
    .layer(DefaultBodyLimit::max(max_body_bytes))
    .layer(TraceLayer::new_for_http())
    .with_state(state);
```

Eight route groups serve four concerns: server-rendered pages (dashboard, search, sessions, graph, workflows), a JSON graph API, vendored static assets, and the catch-all BFF proxy. The `DefaultBodyLimit` layer is set to 256 MB

to accommodate proxied batch ingest requests — operators can POST large event batches through the web proxy without the body being rejected before reaching the sink.

The key insight is that this is not a single-page application. There is no React build step, no webpack, no client-side router. Askama compiles Jinja2-style templates into Rust code at build time, producing type-safe HTML rendering with zero runtime template parsing overhead. Every page is a full server render; navigation is standard HTTP links. This keeps the binary self-contained and the deployment story simple — one executable, no asset pipeline, no CDN.

The Dashboard

The dashboard at `GET /` is the operator's entry point. It fetches the session list from the sink and renders a timeline with a KPI strip.

```
async fn index(State(st): State<Arc<AppState>>) -> impl IntoResponse {
    match fetch_sessions(&st).await {
        Ok(sessions) => {
            let stats = DashboardStats::from_sessions(&sessions);
            IndexTpl { sessions, stats }.into_response()
        }
        Err(e) => {
            warn!(error = ?e, "fetch sessions failed");
            (
                StatusCode::BAD_GATEWAY,
                Html(format!("<h1>sink unreachable</h1><pre>{e:?}</pre>")),
            ).into_response()
        }
    }
}
```

The `DashboardStats` struct is computed client-side in the web process from the session summaries returned by the sink:

```
impl DashboardStats {
    fn from_sessions(sessions: &[SessionSummary]) -> Self {
        use std::collections::HashSet;
        let mut agents: HashSet<&str> = HashSet::new();
        let mut s = DashboardStats {
            total_sessions: sessions.len(),
            ..Default::default()
        };
        for ss in sessions {
            s.total_events += ss.event_count;
            s.total_tool_calls += ss.tool_call_count.unwrap_or(0);
            s.total_guardrails += ss.guardrail_count.unwrap_or(0);
            s.open_alerts += ss.open_alert_count.unwrap_or(0);
            s.total_tokens += ss.total_tokens.unwrap_or(0);
            if let Some(a) = ss.agent.as_deref() {
                agents.insert(a);
            }
        }
        s.unique_agents = agents.len();
        s
    }
}
```

The KPI strip surfaces seven metrics:

KPI	SOURCE FIELD	MEANING
Sessions	`sessions.len()`	Total session count
Events	`sum(event_count)`	Total events across all sessions
Tool Calls	`sum(tool_call_count)`	Tool invocations
Guardrails	`sum(guardrail_count)`	Guardrail evaluations
Alerts	`sum(open_alert_count)`	Unresolved alerts
Tokens	`sum(total_tokens)`	Cumulative token usage
Agents	`unique_agents`	Distinct agent identities

Each session row in the timeline carries a status classification rendered as a CSS class:

```
impl SessionSummary {
    fn status_class(&self) -> &'static str {
        match self.status.as_deref() {
            Some("ok") | Some("complete") => "ok",
            Some("needs_review") | Some("review") => "review",
            Some("blocked") | Some("error") => "error",
            Some("executing") | Some("thinking") => "active",
            _ => "idle",
        }
    }
}
```

This status classification is the visual grammar of the dashboard. An operator scanning the timeline can immediately distinguish completed sessions (green), sessions needing review (yellow), blocked sessions (red), and actively executing sessions (pulsing). The classification is produced by the sink's `session_live` query, which examines the event stream for guardrail blocks, security boundary crossings, and open alerts.

When the sink is unreachable, the dashboard degrades gracefully — it returns a `502 Bad Gateway` with the error detail rather than crashing or showing a blank page. This is important for operations: the web process is the canary for sink health, and a 502 on the dashboard is an immediate signal that the sink is down.

Session Scrubbing

The session view at `GET /sessions/:id` is the core operator workflow: scrubbing a past session to review every prompt, tool call, guardrail decision, and security boundary crossing.

The web process fetches the live session envelope from the sink:

```
async fn fetch_session_live(
    st: &AppState,
    id: &str,
    limit: usize,
    offset: usize,
) -> Result<LiveResponse> {
    let url = format!(
        "{}/v1/sessions/{}/live",
        st.sink_url.trim_end_matches('/'),
        id
    );
    let resp = st
        .http
        .get(url)
        .bearer_auth(&st.sink_token)
        .query(&[
            ("limit", limit.to_string()),
            ("offset", offset.to_string()),
            ("non_boilerplate", "true".to_string()),
        ])
        .send()
        .await?
        .error_for_status()?
        .json::<LiveResponse>()
        .await?;
    Ok(resp)
}
```

The `non_boilerplate=true` query parameter is significant. It tells the sink to filter out command-name-only events — the `/clear`, `/help`, and empty-turn events that clutter raw transcripts but carry no operational signal. This reduces the event count substantially for typical Claude Code sessions, making the timeline navigable.

The `LiveResponse` envelope is the canonical shape that both the current polling path and the future SSE path produce. The comment in the source makes this explicit:

```
/// Canonical entry point for live session data. Both poll and (future) SSE
/// paths produce `LiveResponse`; templates render from this shape so swapping
/// the transport is a one-line change.
```

The envelope contains:

```
struct LiveResponse {
    session_id: String,
    summary: SessionSummary,
    alerts: Vec<Alert>,
    events: Vec<EventRow>,
    total: i64,
    limit: i64,
    offset: i64,
    has_more: bool,
}
```

The `SessionSummary` carries the status classification and KPI counters discussed above. The `alerts` vector surfaces unresolved alerts for the session. The `events` vector is where the visual payload system shines.

Each `EventRow` carries optional typed payloads that the template renders as visual cards:

```
struct EventRow {
    kind: String,
    ui_kind: Option<String>,
    text: Option<String>,
    display_message: Option<String>,
    tool_call: Option<ToolCallPayload>,
    guardrail: Option<GuardrailPayload>,
    security_boundary: Option<SecurityBoundaryPayload>,
    delegation: Option<DelegationPayload>,
    session_summary: Option<SessionSummaryPayload>,
    token_usage: Option<TokenUsagePayload>,
    alert: Option<EventAlert>,
    // ... plus uuid, timestamp, host, agent, source_path, tool_uses
}
```

Each payload type maps to a specific governance event kind:

PAYLOAD	FIELDS	EVENT KIND
`ToolCallPayload`	tool_name, server, endpoint, latency_ms, exit_code, decision	`tool_call` / `mcp-tool-call`
`GuardrailPayload`	engine, rule_id, severity, verdict, trust_score, authority_level, signature_status, enforcement_mode	`guardrail`
`SecurityBoundaryPayload`	verdict, description, target, compliance_ref, boundary, policy_id	`security_boundary`
`DelegationPayload`	from_agent, to_agent, reason	`agent_delegation`
`SessionSummaryPayload`	message, items_completed, items_total	`session_summary`
`TokenUsagePayload`	input_tokens, output_tokens, cache_read, cache_creation, thinking, total, model	`token_usage`
`EventAlert`	severity, category, title, message	alert

The `GuardrailPayload` is particularly rich. It carries the full AgentShield anomaly context — `atcs_rule`, `agent_key`, `tool_name`, `hermes_token_fingerprint`, `authority_level`, `signature_status`, `anomaly_id`, `enforcement_mode`, and `elapsed_us`. This is the Operation Looking Glass renderer contract: the sink populates these fields from the event's `orca_payload` extra field, and the web template renders them as a guardrail card that an operator can inspect to understand exactly which rule fired, at what trust score, under what authority level, and how long the enforcement decision took.

Two helper methods on `EventRow` provide consistent rendering across templates:

```
impl EventRow {
    /// Prefer `ui_kind` (normalized renderer key from the brief) over the raw
    /// `kind`, so templates can switch on a single field.
    fn render_kind(&self) -> &str {
        self.ui_kind.as_deref().unwrap_or(&self.kind)
    }

    /// Best display text for the card body — falls back through
    /// display_message, text, and finally empty string.
    fn primary_text(&self) -> &str {
        self.display_message
            .as_deref()
            .or(self.text.as_deref())
            .unwrap_or("")
    }
}
```

The `render_kind()` method embodies the Forward-Compatible Schema pattern (P4). The raw `kind` field is whatever the upstream client emitted — Claude Code uses `user`, `assistant`, `token_usage`; Codex uses different names; governance events use `orca-trace`, `guardrail`, etc. The `ui_kind` field is a normalized renderer key that the sink computes from the raw kind and the event's payload shape. Templates switch on `render_kind()` rather than `kind`, so when a new upstream event type appears, the sink can map it to an existing `ui_kind` without template changes.

The `primary_text()` method provides a fallback chain: `display_message` (a human-readable summary the sink generates) → `text` (the raw event text) → empty string. This ensures every card has something to show even when the event's text content is sparse.

Evidence Search

The search page at `GET /search` is how operators find specific evidence across sessions. It proxies the sink's FTS5 full-text search endpoint with a set of filters.

The search form captures seven filter dimensions plus pagination:

```
struct SearchParams {
    q: Option<String>,
    session_id: Option<String>,
    host: Option<String>,
    agent: Option<String>,
    source: Option<String>,
    source_path: Option<String>,
    kind: Option<String>,
    tool: Option<String>,
    limit: Option<usize>,
    offset: Option<usize>,
}
```

The `source` and `source_path` fields are merged in the `From` impl — `source` is an alias for `source_path`, and the `or` combiner accepts either:

```
source: params.source.or(params.source_path).unwrap_or_default(),
```

The `SearchForm` determines whether a search has been initiated by checking if any filter is non-empty:

```
fn has_search(&self) -> bool {
    !self.q.trim().is_empty()
        || !self.session_id.trim().is_empty()
        || !self.host.trim().is_empty()
        || !self.agent.trim().is_empty()
        || !self.source.trim().is_empty()
        || !self.kind.trim().is_empty()
        || !self.tool.trim().is_empty()
}
```

If no filter is set, the template renders the empty search form without querying the sink. This is a small but deliberate UX choice: the search page is also the browse page, and an empty visit doesn't waste a sink query.

When a search is submitted, the web process proxies to `GET /v1/search` on the sink:

```
async fn fetch_search(st: &AppState, form: &SearchForm) -> Result<SearchResponse> {
    let url = format!("{}{}/v1/search")");
    let mut query = vec![
        ("limit", form.limit.to_string()),
        ("offset", form.offset.to_string()),
    ];
    if !form.q.trim().is_empty() {
        query.push(("q", form.q.trim().to_string()));
    }
    // ... additional filters
    let resp = st
        .http
        .get(url)
        .bearer_auth(&st.sink_token)
        .query(&query)
        .send()
        .await?
        .error_for_status()?
        .json::<SearchResponse>()
        .await?;
    Ok(resp)
}
```

The sink's FTS5 index covers text, kind, cwd, git_branch, host, source_path, tool_names, and tool_inputs. The `unicode61` tokenizer handles case-insensitive matching with Unicode support. Snippet generation produces context-windowed excerpts of the matched text, so search results show the relevant fragment rather than the full event body.

The `SearchResult` struct includes `text_snippet` and `tool_snippet` fields — these are FTS5-generated snippets that highlight the matched terms. Each result also carries `tool_names` (a vector of tool names extracted from the event) and `rank` (the FTS5 relevance score).

Pagination is handled with `prev_href()` and `next_href()` methods that generate URL-encoded links preserving all active filters:

```
fn href_for_offset(&self, offset: usize) -> String {
    let mut parts = vec![
        ("limit", self.limit.to_string()),
        ("offset", offset.to_string()),
    ];
    if !self.q.trim().is_empty() {
        parts.push(("q", self.q.trim().to_string()));
    }
    // ... additional filters
    let query = parts
        .into_iter()
        .map(|(key, value)| format!("{key}={}", url_component(&value)))
        .collect::<Vec<_>>()
        .join("&");
    format!("/search?{query}")
}
```

The `url_component` function is a hand-rolled percent-encoder that handles the unreserved characters (`A-Za-z0-9-._~`), spaces (as `+`), and percent-encodes everything else. This avoids a dependency on a URL encoding crate for what is a small, well-defined operation.

The search results link back to the session view with an anchor to the specific event:

```
impl SearchResult {
    fn event_href(&self) -> String {
        match (&self.session_id, &self.uuid) {
            (Some(session_id), Some(uuid)) => format!("/sessions/{session_id}#event-
{uuid}"),
            (Some(session_id), None) => format!("/sessions/{session_id}"),
            _ => "#".to_string(),
        }
    }
}
```

This creates a tight loop: search → find evidence → jump to the session timeline at the specific event → scrub the surrounding context. That loop is the core operator workflow for real-time review.

Context Graph Explorer

The graph explorer at `GET /graph/:id` renders a force-directed graph of a session's event timeline. The page itself is a thin template that loads the `/api/graph/:id` JSON endpoint and renders it client-side.

The JSON endpoint builds the graph payload from the session's event list:

```
async fn api_graph(State(st): State<Arc<AppState>>, Path(id): Path<String>) -> impl
IntoResponse {
    // Build a force-graph payload from the event timeline. v0.1: parent_uuid
    // edges plus tool_use nodes are sufficient. Later we'll query neo4j
    // directly for richer relations.
    match fetch_session_events(&st, &id).await {
        Ok(events) => {
            let mut nodes = Vec::new();
            let mut links: Vec<serde_json::Value> = Vec::new();
            for e in &events {
                if let Some(uuid) = &e.uuid {
                    nodes.push(serde_json::json!({
                        "id": uuid,
                        "kind": e.kind,
                        "host": e.host,
                        "label": e.text.clone().unwrap_or_default()
                            .chars().take(80).collect::<String>(),
                    }));
                }
            }
            // We don't have parent_uuid in EventRow yet; v0.1 ships timeline-only graph.
            for w in events.windows(2) {
                if let (Some(a), Some(b)) = (&w[0].uuid, &w[1].uuid) {
                    links.push(serde_json::json!({ "source": a, "target": b }));
                }
            }
            Json(serde_json::json!({"nodes": nodes, "links": links})).into_response()
        }
        Err(e) => {
            warn!(error = ?e, "api_graph fetch failed");
            (
                StatusCode::BAD_GATEWAY,
                Json(serde_json::json!({"error": format!("{e:?}")})),
            ).into_response()
        }
    }
}
```

The v0.1 simplification is explicit in the code comments. Nodes are created from event UUIDs with a truncated (80-character) text label. Links are created from sequential event windows — event N connects to event N+1. This produces a linear timeline graph, not a semantic relationship graph.

The comment acknowledges the limitation: "Later we'll query neo4j directly for richer relations." The sink already has a Neo4j graph writer that creates `(:Session)-[:HAS]->(:Event)` structures with edges to `Tool`, `Agent`, `Branch`, `File`, and `Commit` nodes. The graph context API (`GET /v1/graph/context`) already supports entity-based queries with weighted scoring (files and commits weight 4.0, branches 3.0, tools 2.0, agents 0.75). The web explorer will eventually call that API instead of building a linear timeline.

This is an instance of the SQLite WAL with Fallback pattern (P5). The sink's `graph_context()` method tries Neo4j first and falls back to a SQLite-based entity extraction and related-session scoring algorithm when Neo4j is

unavailable. The web explorer currently bypasses both and builds its own simple graph from the event timeline, but the architecture supports a straightforward upgrade path: replace the `api_graph` handler with a call to `/v1/graph/context` and render the richer entity-relationship graph.

The browser-side rendering uses a force-directed graph layout (typically D3.js or a similar library) that reads the `{nodes, links}` JSON and positions nodes with a physics simulation. Each node is colored by event kind and labeled with the truncated text. Operators can visually trace the flow of a session — which tools were called, where guardrails fired, where security boundaries were crossed.

Workflow Candidate Review

The workflows page at `GET /workflows` is where operators review and promote workflow improvement candidates. These candidates are generated by panel review runs — multi-agent reviews where multiple AI reviewers (Claude, Codex, Gemini) examine a session and produce consensus findings and recommendations.

The page is organized by status tabs:

```
async fn workflows_view(
    State(st): State<Arc<AppState>>,
    Query(params): Query<WorkflowParams>,
) -> impl IntoResponse {
    let filter = WorkflowFilter::from(params);
    match fetch_workflows(&st, &filter).await {
        Ok(response) => {
            let candidate_count = response.workflows.iter()
                .filter(|w| w.status == "candidate").count();
            let approved_count = response.workflows.iter()
                .filter(|w| w.status == "approved").count();
            let retired_count = response.workflows.iter()
                .filter(|w| w.status == "retired").count();
            WorkflowsTpl {
                filter,
                response: Some(response),
                error: None,
                candidate_count,
                approved_count,
                retired_count,
            }.into_response()
        }
        // ... error handling
    }
}
```

Each `WorkflowRow` carries the full review metadata:

```
struct WorkflowRow {
    id: i64,
    workflow_id: String,
    title: String,
    problem: String,
    recommendation: String,
    status: String,
    risk: String,
    stance: String,
    confidence: Option<f64>,
    org: Option<String>,
    team: Option<String>,
    workspace: Option<String>,
    visibility: String,
    source_event_ids: Vec<i64>,
    reviewer_ids: Vec<String>,
    created_by: Option<String>,
    decision_notes: Option<String>,
    created_at: String,
    updated_at: String,
    approved_at: Option<String>,
    approved_by: Option<String>,
    retired_at: Option<String>,
    retired_by: Option<String>,
}
```

The template renders three visual indicators per workflow:

```
impl WorkflowRow {
    fn status_class(&self) -> &'static str {
        match self.status.as_str() {
            "approved" => "approved",
            "retired" | "dismissed" => "retired",
            "candidate" => "candidate",
            _ => "neutral",
        }
    }

    fn risk_class(&self) -> &'static str {
        match self.risk.as_str() {
            "critical" | "high" => "high",
            "medium" => "medium",
            "low" => "low",
            _ => "neutral",
        }
    }

    fn stance_class(&self) -> &'static str {
        match self.stance.as_str() {
            "stop" => "stop",
            "warn" => "warn",
            "continue" => "continue",
            _ => "neutral",
        }
    }

    fn confidence_label(&self) -> String {
        self.confidence
            .map(|value| format!("{:.0}%", (value * 100.0).round()))
            .unwrap_or_else(|| "-".to_string())
    }
}
```

The promotion lifecycle is: **candidate → approved → retired**. Candidates are newly generated from panel reviews. Approved workflows are active guidance rules that agents should treat as operational constraints. Retired workflows are no longer active but preserved for audit history.

The `POST /v1/workflows/from-panel` endpoint (proxied through the BFF) creates new candidates from panel review runs. The sink's `workflow_input_from_panel()` function extracts findings, recommendations, reviewer IDs, and evidence event IDs from the panel run JSON. It requires `consensus.emit_advisory=true` — only panel runs that reached an advisory consensus are eligible for promotion to workflow candidates.

The `source_event_ids` field creates the evidence chain: each workflow candidate cites the specific events that led to its creation. When an operator reviews a candidate, they can jump to those events in the session timeline to verify the finding. This is the private-by-default scoping pattern (P6) in action — promotion requires explicit human review, and the evidence trail is preserved.

BFF Proxy Pattern

The `/v1/*path` catch-all proxy is the most operationally important route in the web crate. It makes the web process the single off-host entry point for all API access.

```
async fn proxy_v1(
    State(st): State<Arc<AppState>>,
    Path(path): Path<String>,
    method: Method,
    uri: Uri,
    headers: HeaderMap,
    body: Bytes,
) -> Response {
    if !auth_ok(&headers, &st.sink_token) {
        return (
            StatusCode::UNAUTHORIZED,
            Json(serde_json::json!({"error":"unauthorized"})),
        ).into_response();
    }

    let mut url = format!("{}/v1/{}", st.sink_url.trim_end_matches('/'), path);
    if let Some(query) = uri.query() {
        url.push('?');
        url.push_str(query);
    }

    let req_method = match reqwest::Method::from_bytes(method.as_str().as_bytes()) {
        Ok(method) => method,
        Err(e) => { /* ... error handling */ }
    };

    let mut req = st
        .http
        .request(req_method, url)
        .bearer_auth(&st.sink_token)
        .body(body);
    if let Some(content_type) = headers.get(header::CONTENT_TYPE) {
        req = req.header(header::CONTENT_TYPE, content_type);
    }
    if let Some(accept) = headers.get(header::ACCEPT) {
        req = req.header(header::ACCEPT, accept);
    }

    match req.send().await {
        Ok(resp) => {
            let status = resp.status();
            let content_type = resp.headers().get(header::CONTENT_TYPE).cloned();
            match resp.bytes().await {
                Ok(bytes) => {
                    let mut out = (status, bytes).into_response();
                    if let Some(content_type) = content_type {
                        out.headers_mut().insert(header::CONTENT_TYPE, content_type);
                    }
                    out
```

```
                }
                Err(e) => /* ... error handling */
            }
        }
        Err(e) => /* ... error handling */
    }
}
```

The proxy is transparent: it preserves the HTTP method, query string, content-type, and accept headers. It authenticates the incoming request with the sink token (via `auth_ok`), then re-authenticates the outgoing request to the sink with the same token. The response status, body, and content-type are passed through unchanged.

This design has a critical operational implication: **clients only need to reach port 9475**. The sink at port 39474 can be firewalled to only accept connections from the web process. In a Docker Compose deployment, the sink and web are on the same network and the sink never exposes its port to the host. In a Kubernetes deployment, the sink service is ClusterIP-only and the web service is the ingress target.

The `auth_ok` function uses a constant-time-ish comparison to prevent timing side-channels on token validation:

```
fn auth_ok(headers: &HeaderMap, expected: &str) -> bool {
    let Some(auth) = headers
        .get(header::AUTHORIZATION)
        .and_then(|v| v.to_str().ok())
    else {
        return false;
    };
    let Some(token) = auth.strip_prefix("Bearer ") else {
        return false;
    };
    if token.len() != expected.len() {
        return false;
    }
    let mut diff = 0u8;
    for (a, b) in token.bytes().zip(expected.bytes()) {
        diff |= a ^ b;
    }
    diff == 0
}
```

This is the same pattern used in the sink's `token_matches()` function. The length check fails fast (an incorrect-length token cannot match), but once lengths match, every byte is XOR'd into an accumulator regardless of whether a mismatch was already found. This prevents an attacker from timing the comparison to discover the token byte-by-byte.

ANTI-PATTERN: SINGLE SHARED TOKEN

The BFF proxy uses the same shared token (`AGENTCHRON_SINK_TOKEN`) to authenticate both incoming client requests and outgoing sink requests. This is the Single Shared Token anti-pattern (AP1) identified in the architecture analysis. A compromised read token currently grants ingest and admin capabilities — there is no separation between read, ingest, and admin auth roles at the origin sink.

The Cloudflare edge gateway models split tokens (read/ingest/admin/origin), but the origin sink does not. This is a P0 gap with a well-specified implementation path: split auth into read, ingest, admin, origin, and agent-scoped tokens. Until that split is implemented, the web proxy's `auth_ok` check is the only auth boundary, and it uses the same token for all roles.

Vendored Assets for Air-Gapped Deployment

The web crate compiles client-side JavaScript and CSS directly into the binary using `include_bytes!`:

```
/// Vendored client assets compiled into the binary. Brief constraint: no
/// external CDN dependencies (network-isolated deployments + supply-chain).
const VENDOR_HIGHLIGHT_JS: &[u8] = include_bytes!("../static/vendor/highlight.min.js");
const VENDOR_HIGHLIGHT_CSS: &[u8] = include_bytes!("../static/vendor/atom-one-
dark.min.css");
const VENDOR_MARKED_JS: &[u8] = include_bytes!("../static/vendor/marked.min.js");
```

Three assets are vendored:

ASSET	PURPOSE	ROUTE
`highlight.min.js`	Syntax highlighting for code blocks	`/static/vendor/highlight.min.js`
`atom-one-dark.min.css`	Dark theme for highlight.js	`/static/vendor/atom-one-dark.min.css`
`marked.min.js`	Markdown rendering for event text	`/static/vendor/marked.min.js`

Each asset is served by a dedicated handler that sets the correct content-type:

```
async fn vendor_highlight_js() -> Response {
    (
        StatusCode::OK,
        [(header::CONTENT_TYPE, "application/javascript; charset=utf-8")],
        VENDOR_HIGHLIGHT_JS,
    ).into_response()
}
```

This is a deliberate constraint, not a convenience. The comment is explicit: "no external CDN dependencies (network-isolated deployments + supply-chain)." Orca is designed to run in network-isolated environments — air-gapped labs, private clouds, customer networks with no internet egress. Loading client-side assets from a CDN would break in those environments and would introduce a supply-chain risk (a compromised CDN could inject malicious JavaScript into the operator console).

By compiling the assets into the binary, the web crate becomes a single self-contained executable. There is no `static/` directory to mount, no asset CDN to configure, no `NODE_ENV` to set. The binary IS the deployment artifact. This simplifies the deployment story dramatically:

```
# Deploy the web crate to an air-gapped host:
scp agentchron-web airgapped-host:/opt/orca/
ssh airgapped-host 'AGENTCHRON_SINK_URL=http://sink:9474 AGENTCHRON_SINK_TOKEN=... /opt/
orca/agentchron-web'
```

The trade-off is binary size: the vendored assets add roughly 100-200 KB to the binary. This is negligible compared to the Rust standard library and the Axum/reqwest/Tokio dependency tree. The benefit — zero external dependencies at runtime — far outweighs the cost.

This pattern is part of the Local-First, Data-Residency-by-Default theme. Orca is designed to run inside the trust boundary. Cloud federation is opt-in and summaries-only. Customer Orca starts blank. The vendored assets are a small but concrete expression of this philosophy: even the browser-side rendering is self-contained.

Patterns Developed in This Chapter

PRIVATE-BY-DEFAULT SCOPING (P6)

The search and session views inherit the sink's private-by-default scoping. The sink's search API accepts `team`, `workspace`, `visibility`, `purpose`, and `promoted` filters. The web search form currently exposes `q`, `session_id`, `host`, `agent`, `source`, `kind`, and `tool` — the scope filters are applied by the sink based on the token's associated scope. This means an operator with a team-scoped token can only search within their team's events, even though the web form doesn't show a team filter. The scoping is enforced at the data layer, not the presentation layer.

SQLITE WAL WITH FALLBACK (P5)

The graph explorer v0.1 builds its graph from the event timeline in the web process, bypassing both Neo4j and the SQLite fallback. But the sink's `graph_context()` API — which the web process could call instead — already implements the fallback pattern: try Neo4j first, fall back to SQLite entity extraction. The upgrade path from v0.1 to a richer graph is to call `/v1/graph/context` from the `api_graph` handler instead of building the graph from the event list.

FORWARD-COMPATIBLE SCHEMA (P4)

The `render_kind()` method on `EventRow` prefers the normalized `ui_kind` over the raw `kind`. When a new upstream event type appears (say, a new Codex event kind), the sink can map it to an existing `ui_kind` (like `tool_call` or `guardrail`) without any template changes. The template switches on `render_kind()`, not on the raw kind, so it is insulated from upstream schema changes.

Anti-Patterns Addressed

SINGLE SHARED TOKEN (AP1)

The BFF proxy uses the same shared token for both incoming auth and outgoing sink auth. The `auth_ok` function on the web side and the `bearer_auth` call on the outgoing reqwest request use the same `sink_token` from `AppState`. A compromised client token grants full ingest and admin access to the sink. The mitigation is the planned token split (read/ingest/admin/origin/agent-scoped), which will require the web process to hold multiple tokens and select the appropriate one based on the proxied route.

QDRANT VECTOR WRITER IS A DEAD STUB (AP3)

The graph explorer v0.1 uses sequential event links, not semantic relations. This is partly because the Qdrant vector writer is a dead stub — `write_event()` only logs a debug message and returns `Ok(())`. Semantic graph relations would require embedding-based similarity, which depends on the vector index being functional. The code comment in the sink's vector writer is honest: "embedding model wiring lands in v0.2." Until then, the graph explorer is a timeline visualizer, not a semantic relationship explorer.

The LiveResponse Envelope and the SSE Future

The `LiveResponse` struct is not just a response shape — it is a forward-compatible contract designed for transport swapping. The source comment makes this explicit:

```
/// Canonical entry point for live session data. Both poll and (future) SSE
/// paths produce `LiveResponse`; templates render from this shape so swapping
/// the transport is a one-line change.
```

Today, the session view polls the sink's `GET /v1/sessions/:id/live` endpoint on each page load. The sink constructs the `LiveResponse` from SQLite queries — the session summary, open alerts, and a paginated event list. The web process deserializes this into the same `LiveResponse` struct and passes it to the Askama template.

When the Cypher/Substrate team delivers the `cm-receiptd` SSE stream at `/receipts/stream`, the sink will be able to emit incremental `LiveResponse` deltas as new events arrive. The web process will hold an SSE connection open and update the page as deltas arrive. The templates won't change — they already render from `LiveResponse`. The only change will be in how the web process obtains the `LiveResponse`: HTTP poll today, SSE stream tomorrow.

This is a small but important example of the Forward-Compatible Schema pattern. The response shape is frozen; the transport is swappable. The templates are insulated from the transport change because they depend on the shape, not the delivery mechanism.

The `has_more` field in `LiveResponse` is the pagination signal. When `has_more` is true, the template can render a "Load more" link that fetches the next page with an increased offset. The `total` field gives the full event count for display ("Showing 1-500 of 3,847"). The `limit` and `offset` fields echo the request parameters for consistency.

Askama Templates: Type-Safe HTML Rendering

Askama is a Rust template engine that compiles Jinja2-style templates into Rust code at build time. The `#[derive(Template)]` macro generates a `render()` method that produces `String` output with zero runtime template parsing overhead.

Five templates are defined in the web crate:

```
#[derive(Template)]
#[template(path = "index.html")]
struct IndexTpl { sessions: Vec<SessionSummary>, stats: DashboardStats }

#[derive(Template)]
#[template(path = "session.html")]
struct SessionTpl { session_id: String, summary: SessionSummary, alerts: Vec<Alert>,
events: Vec<EventRow> }

#[derive(Template)]
#[template(path = "graph.html")]
struct GraphTpl { session_id: String }

#[derive(Template)]
#[template(path = "workflows.html")]
struct WorkflowsTpl { filter: WorkflowFilter, response: Option<WorkflowResponse>, error:
Option<String>, candidate_count: usize, approved_count: usize, retired_count: usize }

#[derive(Template)]
#[template(path = "search.html")]
struct SearchTpl { form: SearchForm, searched: bool, response: Option<SearchResponse>,
error: Option<String> }
```

Each template struct carries exactly the data its template needs — no more, no less. The `IndexTpl` carries sessions and stats; the `SessionTpl` carries the session ID, summary, alerts, and events; the `GraphTpl` carries only the session ID (the graph data is fetched client-side via `/api/graph/:id`). This is a clean separation: the template struct is the view model, and the template file is the view.

The Askama compiler checks at build time that every variable referenced in the template is available on the struct. If a template references `{{ event.render_kind() }}` but the struct doesn't have an `events` field of type `Vec<EventRow>`, the build fails. This catches template errors at compile time rather than at render time — a significant reliability advantage over runtime-parsed templates like Handlebars or Liquid.

The `IntoResponse` implementation from `askama_axum` wraps the rendered HTML in a `200 OK` response with `Content-Type: text/html`. This means each handler can return the template directly as its response:

```
async fn session_view(
    State(st): State<Arc<AppState>>,
    Path(id): Path<String>,
) -> impl IntoResponse {
    match fetch_session_live(&st, &id, 500, 0).await {
        Ok(resp) => SessionTpl {
            session_id: resp.session_id,
            summary: resp.summary,
            alerts: resp.alerts,
            events: resp.events,
        }.into_response(),
        Err(e) => {
            warn!(error = ?e, "fetch session live failed");
            (
                StatusCode::BAD_GATEWAY,
                Html(format!("<h1>sink unreachable</h1><pre>{e:?}</pre>")),
            ).into_response()
        }
    }
}
```

The error case returns a hand-crafted HTML page with the error detail. This is deliberately simple — the error page is a diagnostic, not a polished UI. The `warn!` log entry ensures the error is captured in structured logs for operational debugging.

Error Handling and Degradation

The web crate's error handling follows a consistent pattern across all handlers: try to fetch data from the sink, render the template on success, return a diagnostic page or error JSON on failure. The degradation is graceful — the web process never crashes on a sink error, it returns a 502 Bad Gateway with the error detail.

For the JSON API endpoint (`/api/graph/:id`), the error response is JSON:

```
Err(e) => {
    warn!(error = ?e, "api_graph fetch failed");
    (
        StatusCode::BAD_GATEWAY,
        Json(serde_json::json!({"error": format!("{e:?}")})),
    ).into_response()
}
```

For the search page, the error is passed to the template as an `error` field, which the template renders in-place rather than replacing the entire page:

```
Err(e) => {
    warn!(error = ?e, "fetch search failed");
    SearchTpl {
        form,
        searched,
        response: None,
        error: Some(format!("{e:?}")),
    }.into_response()
}
```

This means the search form remains visible even when the sink is unreachable — the operator can modify their query and retry without losing their filter context. The error appears inline, not as a replacement page. The same pattern is used for the workflows page.

The proxy handler has its own error handling for sink connectivity issues:

```
Err(e) => {
    warn!(error = ?e, path = %path, "proxied /v1 request failed");
    (
        StatusCode::BAD_GATEWAY,
        Json(serde_json::json!({"error":"sink unreachable"})),
    ).into_response()
}
```

The proxy does not expose the raw error to the client — it returns a generic "sink unreachable" message. This is a security choice: the raw error might contain internal network topology or sink URL details that shouldn't be exposed to off-host clients. The `warn!` log captures the full error for operational debugging.

The Dashboard KPI Strip in Practice

The KPI strip is more than a summary — it is the operator's situational awareness display. Consider a typical morning operations check:

1. **Sessions count** — Is there unexpected activity overnight? A spike might indicate a scheduled agent run or an unauthorized session.
2. **Events count** — Is the event volume consistent with the session count? A high event-to-session ratio might indicate a session stuck in a tool-call loop.
3. **Tool calls** — Are tool calls within expected ranges? A spike might indicate an agent running uncontrolled shell commands.
4. **Guardrails** — Are guardrail evaluations firing? An increase might indicate a policy change or an agent testing boundaries.
5. **Alerts** — Are there unresolved alerts? This is the most actionable KPI — open alerts need human attention.
6. **Tokens** — Is token usage within budget? A spike might indicate an agent running expensive long-context operations.
7. **Agents** — How many distinct agent identities are active? An unexpected agent identity might indicate a misconfigured or unauthorized agent.

The KPI strip is computed entirely in the web process from the session list — no additional sink queries are needed. This is a performance choice: the session list is already fetched for the timeline, and the KPIs are just aggregations over that list. The sink's `list_sessions` endpoint returns session summaries with per-session KPI counters, and the web process sums them.

The `DashboardStats::from_sessions()` method is the aggregation logic. It iterates over the session list once, accumulating counts and collecting unique agent identities into a `HashSet`. The time complexity is O(n) where n is the session count — negligible for typical fleet sizes (hundreds to low thousands of sessions).

Session Timeline Rendering

The session template renders the event timeline as a vertical list of cards, each keyed to the event's UUID. The `render_kind()` method determines which card variant to render:

- `tool_call` — Renders the `ToolCallPayload` with tool name, MCP server, endpoint, latency, exit code, and decision
- `guardrail` — Renders the `GuardrailPayload` with engine, rule ID, severity, verdict, trust score, authority level, signature status, and enforcement mode
- `security_boundary` — Renders the `SecurityBoundaryPayload` with verdict, description, target, compliance reference, boundary, and policy ID
- `delegation` — Renders the `DelegationPayload` with from-agent, to-agent, and reason
- `session_summary` — Renders the `SessionSummaryPayload` with message, items completed, and items total
- `token_usage` — Renders the `TokenUsagePayload` with input/output/cache/thinking tokens and model name
- `alert` — Renders the `EventAlert` with severity, category, title, and message
- `user` / `assistant` — Renders the primary text with Markdown rendering via marked.js and syntax highlighting via highlight.js

The guardrail card is the richest visual payload. It surfaces the full AgentShield anomaly context, allowing an operator to understand not just *that* a guardrail fired but *why* and *under what authority*:

FIELD	EXAMPLE	OPERATIONAL MEANING
`engine`	`agentshield`	Which enforcement engine produced the decision
`rule_id`	`no-prod-terraform-without-approval`	The specific rule that fired
`severity`	`high`	How serious the violation is
`verdict`	`blocked`	What the engine decided (blocked/flagged/review/allowed)
`trust_score`	`0.23`	The engine's confidence in the decision (0.0 = low trust, 1.0 = high)
`authority_level`	`agent`	The authority level of the operating agent
`signature_status`	`verified`	Whether the Hermes identity signature was verified
`enforcement_mode`	`enforce`	Whether the engine was in enforce or dry-run mode
`elapsed_us`	`1450`	How long the enforcement decision took in microseconds

This is the Operation Looking Glass renderer contract in full. The sink populates these fields from the event's `orca_payload` extra field, and the web template renders them as an inspectable card. An operator reviewing a

blocked session can see exactly which rule fired, at what trust score, under what authority, and how long the decision took — without needing to query the raw JSONL.

The `elapsed_us` field is particularly interesting for performance debugging. If guardrail evaluations are taking tens of milliseconds, that's latency on the agent's hot path. The `enforcement_mode` field distinguishes enforce-mode blocks (the action was actually blocked) from dry-run-mode reports (the action would have been blocked but was allowed for observe-mode rollout).

Conclusion

AgentChron Web is a deliberately thin layer. It stores nothing, computes nothing that the sink hasn't already computed, and adds no business logic beyond URL encoding and KPI aggregation. Its value is operational: it provides a server-rendered UI that operators can use to scrub sessions, search evidence, explore graphs, and review workflow candidates, and it provides a BFF proxy that consolidates all API access through a single port.

The store-nothing architecture means the web process is stateless and disposable. It can be restarted without data loss, scaled horizontally behind a load balancer, or deployed on a different host than the sink. The vendored assets mean it runs in air-gapped environments with zero external dependencies. The visual payload system means governance events render as rich, inspectable cards rather than raw JSON. And the BFF proxy means the sink can remain internal-only, with the web process as the sole external entry point.

The limitations are clear: the graph explorer is v0.1, the token model is shared, and the scope filters are implicit rather than user-facing. But the architecture is positioned for upgrade — the `LiveResponse` envelope is ready for SSE, the `api_graph` handler is ready to call the graph context API, and the `auth_ok` function is ready to be extended with role-based token selection. The web crate is a foundation, not a finished product, and it is honest about that in its code comments.

Chapter 5: Orca Guard

Guard's Position: Before the Model, Before the Disk

Orca Guard solves two problems that a network gateway alone cannot solve. The first is preventing raw secrets from reaching the model provider — an Anthropic API key pasted into a Claude Code prompt travels to Anthropic's servers in the request body, and once it arrives, it is in someone else's logs. The second is preventing secrets from being written into session transcripts — when Claude Code processes a prompt, it writes the prompt text to a JSONL file on disk before any network call happens. By the time a network gateway could intercept the API request, the secret is already persisted locally.

These are distinct problems with distinct solutions:

```
Problem 1: Secret → Model Provider (network egress)
  Solution: HTTP gateway (orca-llm-gateway) intercepts request before forwarding

Problem 2: Secret → Session Transcript (local disk)
  Solution: Claude Code hooks (orca-guard claude-hook) block at UserPromptSubmit
```

A network gateway can solve Problem 1 but not Problem 2. Once a secret lands in a transcript JSONL file, it has already been persisted to disk. The file watcher (`agentchron-agent run`) will pick it up, sanitize it, and ship it to the sink — but the raw secret is already on disk in the original file. Rotating the credential is the only remediation at that point.

Guard's dual-layer principle is to block at the Claude Code hook events `UserPromptSubmit` and `PreToolUse` — before the prompt is written to the transcript and before a tool executes that could reveal secrets. The hooks are the earliest interception point in the pipeline.

The defense-in-depth chain is:

```
Claude Code Hook (orca-guard claude-hook)
  → blocks before transcript persistence
    ↓ if not blocked
LLM Gateway (orca-llm-gateway)
  → blocks before model provider egress
    ↓ if not blocked
Agent Sanitizer (parse_line → sanitize_json_line)
  → redacts before transport to sink
    ↓ always runs
Sink Sanitizer (ingest_one → sanitize_event)
  → belt-and-suspenders redaction at storage
```

Each layer is independently sufficient for its boundary. The hook layer prevents transcript persistence. The gateway layer prevents provider egress. The sanitizer layers prevent durable storage of anything that slipped through. Guard is the first interception point in this chain.

Three Binaries

The Guard subsystem comprises three binaries in the `agentchron-agent` crate:

`ORCA-GUARD SCAN-TEXT`

The standalone scanner. Reads raw text from stdin, runs the full Guard analysis (secret detection + risk matching), and prints a JSON decision report. This is the QA and pipeline integration binary — it can be used in CI scripts, pre-commit hooks, or any context where you need to scan arbitrary text without the Claude Code hook protocol.

```
echo "ANTHROPIC_API_KEY=sk-ant-api03-..." | orca-guard scan-text
```

The output is a JSON object with the decision, matched rules, and remediation advice. It does not print the matched secret values.

`ORCA-GUARD CLAUDE-HOOK`

The Claude Code integration binary. Reads Claude Code hook JSON from stdin and returns decisions via exit codes and stdout. This is the binary configured in `~/.claude/settings.json` for the `UserPromptSubmit` and `PreToolUse` hook events.

The exit-code protocol is Claude Code-specific and deliberate:

DECISION	EXIT CODE	STDOUT	BEHAVIOR
Allow	`0`	Nothing	Prompt/tool proceeds; no stdout injected into Claude context
Block	`2`	Claude decision JSON	Claude Code shows the block reason to the user
Gray-area PreToolUse	`0`	Hook JSON with `permissionDecision: ask`	Claude Code surfaces a human approval prompt
Gray-area UserPromptSubmit	`2`	Claude decision JSON	Blocks (no interactive approval shape exists for UserPromptSubmit)

The allow case printing nothing is a specific design choice. Claude Code injects hook stdout into the conversation context. If the hook printed "allowed" on every safe prompt, the conversation would accumulate noise. Silence on allow keeps the context clean.

The gray-area distinction between `PreToolUse` and `UserPromptSubmit` is important. `PreToolUse` has a documented interactive approval shape — Claude Code can show the user a prompt asking "Allow this tool use?" with the risk match details. `UserPromptSubmit` does not have this shape. There is no way for the hook to say "ask the user about this prompt" — it can only allow or block. So gray-area prompt text (like path-only references to sensitive directories) blocks in `UserPromptSubmit` unless the operator has created a one-shot `/approve-paste` grant.

`ORCA-LLM-GATEWAY`

An Anthropic-compatible HTTP gateway. Point `ANTHROPIC_BASE_URL` at it, and every model request passes through Guard inspection before being forwarded upstream. This is defense-in-depth behind the hooks — if a secret somehow reaches the API request stage (say, through a non-Claude client that doesn't have hooks), the gateway catches it.

The gateway is a full HTTP server with bearer-token auth:

```
ORCA_GATEWAY_TOKEN=*** \
ORCA_GATEWAY_UPSTREAM=https://api.anthropic.com \
ORCA_GATEWAY_UPSTREAM_API_KEY=*** \
orca-llm-gateway --bind 0.0.0.0:19741
```

Blocked requests return `403` with a JSON body containing `orca_guard_blocked` and the rule ID. Safe requests are forwarded to the upstream provider. Auth enforcement is separate — requests without a valid bearer token return `401` with `orca_gateway_unauthorized`.

Four Guard Modes

Guard operates in four modes, configurable via `--mode` or `ORCA_GUARD_MODE`:

`PRODUCTION` (DEFAULT)

Hard-blocks secret-shaped values, direct secret-content reads, and secret manager reads. Gray-area path-only `PreToolUse` references ask for human approval via `permissionDecision: ask`. Gray-area `UserPromptSubmit` text blocks (exit 2) because there is no interactive approval shape.

This is the mode for live operational work. It is the default because the cost of a leaked secret is higher than the cost of a blocked prompt.

`DEV`

Blocks secret-shaped values and direct secret-content reads, but allows env-var names, path-only documentation, and directory listing discussion. This is the mode for application development — you can discuss `ANTHROPIC_API_KEY` as an identifier, reference `~/.docker/` as a path, and document `.env` file structure without triggering blocks.

The distinction between `production` and `dev` is about what constitutes a risk. In production, a path-only reference to `~/.docker/` is a gray-area event that requires approval — it could be a precursor to reading secret files. In dev, the same reference is benign documentation.

`AUDIT`

Report-only mode. Allows the action but reports `would_block_in_production=true` with the production decision. This is for QA and tuning — you run audit mode to see what production mode would block without actually blocking anything.

Anti-pattern warning (AP4): Audit mode must not be treated as safe for secret-bearing work. It intentionally allows actions that production mode would block. Using audit mode for actual secret-bearing work defeats the entire purpose of Guard. The documentation is explicit: "Audit Mode is for QA and tuning. It must not be treated as a safe mode for secret-bearing work."

`MAX`

The strictest mode. Deny-all for unmatched tool calls when combined with `--policy-mode enforce`. This is the mode for high-assurance environments where any unrecognized tool use should be blocked rather than allowed.

The mode smoke tests from the QA document demonstrate the differences:

```
# Production mode: blocks env-var documentation in UserPromptSubmit
printf '%s' '{"hook_event_name":"UserPromptSubmit","prompt":"document HEALTH_INTERNAL_TOKEN
and /etc/agentos/agentos-runtime.env without values"}' \
  | orca-guard claude-hook --mode production
echo $?  # 2, UserPromptSubmit cannot open an approval prompt

# Dev mode: allows env-var documentation
printf '%s' '{"hook_event_name":"UserPromptSubmit","prompt":"document HEALTH_INTERNAL_TOKEN
and /etc/agentos/agentos-runtime.env without values"}' \
  | orca-guard claude-hook --mode dev --print-allow
echo $?  # 0

# Audit mode: allows but reports would_block_in_production=true
printf '%s' '{"hook_event_name":"UserPromptSubmit","prompt":"document HEALTH_INTERNAL_TOKEN
and /etc/agentos/agentos-runtime.env without values"}' \
  | orca-guard claude-hook --mode audit --print-allow
echo $?  # 0, report includes would_block_in_production=true
```

Human Paste Approval: `/approve-paste`

There are legitimate workflows where an operator needs to paste a secret into a Claude Code session — initializing a project with an API key, configuring a deploy token, or running a one-time bootstrap command. Guard's `production` mode would block these, creating a friction point that could lead operators to disable Guard entirely (the worst outcome).

The `/approve-paste` mechanism solves this with a one-shot human grant:

```
# In a Claude Code session, the operator runs:
/approve-paste

# This executes:
orca-guard approve-paste --ttl-seconds 120 --reason "QA one-shot paste"
```

The approval is written to `~/.local/state/orca-guard/paste-approvals.jsonl` and has four constraints:

CONSTRAINT	VALUE	RATIONALE
Expiry	2 minutes	Limits the window of exposure
Consumption	One use	Prevents replay
Scope	Current project directory (when Claude supplies `cwd`)	Prevents cross-project authorization
Authorization	`UserPromptSubmit` only	Never authorizes `PreToolUse`, secret-file reads, shell commands, or MCP calls

The approval is transparent in audit trails. The Guard Review and governance receipt path still records that the prompt would have been blocked without the human grant. The approval doesn't suppress the finding — it adds a human-authorization annotation to it.

The QA smoke test demonstrates the one-shot consumption:

```
# Create approval
cargo run -q -p agentchron-agent --bin orca-guard -- approve-paste --ttl-seconds 120 --
reason "QA one-shot paste"

# First use: allowed (approval consumed)
printf '%s' '{"hook_event_name":"UserPromptSubmit","prompt":"use sk-ant...test"}' \
  | orca-guard claude-hook --mode max --print-allow
echo $?  # 0, one-shot approval consumed

# Second use: blocked (approval was one-use)
printf '%s' '{"hook_event_name":"UserPromptSubmit","prompt":"use sk-ant...test"}' \
  | orca-guard claude-hook --mode max --print-allow
echo $?  # 2, approval was one-use
```

This is a careful balance between security and usability. The approval is narrow enough that it cannot be used to disable Guard, but broad enough to unblock legitimate one-time paste workflows. The two-minute expiry means an approval created and forgotten doesn't create a persistent hole.

QA and UAT Process

Guard's QA/UAT process is rigorous and documented with concrete smoke tests. Six manual smoke cases must pass:

SMOKE 1: PROMPT SECRET IS BLOCKED

```
CANARY="$(python3 -c 'import uuid; print("sk-ant...ard-" + uuid.uuid4().hex)')"
printf '{"hook_event_name":"UserPromptSubmit","prompt":"use %s"}' "$CANARY" \
  | orca-guard claude-hook
echo $?
```

Expected: Exit code `2`, JSON contains `"decision":"block"`, rule id `anthropic-api-key`, no raw key in output.

The canary token is a synthetic `sk-ant-` prefixed string with a UUID suffix. It matches the Anthropic API key pattern but is not a real key. The test verifies that the secret detector catches it and that the block JSON does not echo the matched value back.

SMOKE 2: SAFE PROMPT IS ALLOWED SILENTLY

```
printf '%s' '{"hook_event_name":"UserPromptSubmit","prompt":"summarize the deployment
plan"}' \
  | orca-guard claude-hook
echo $?
```

Expected: Exit code `0`, no stdout.

SMOKE 3: `.ENV.EXAMPLE` IS ALLOWED

```
printf '%s' '{"hook_event_name":"UserPromptSubmit","prompt":"copy .env.example for local
dev"}' \
  | orca-guard claude-hook --print-allow
```

Expected: Decision `allow`, exit code `0`.

Template file references (`.env.example`, `.env.template`, `.env.sample`) are documentation, not secret stores. They are allowed in all modes.

SMOKE 4: SECRET FILE READ IS BLOCKED

```
printf '%s' '{"hook_event_name":"PreToolUse","tool_name":"Bash","tool_input":
{"command":"cat ~/.ssh/id_ed25519"}}' \
  | orca-guard claude-hook
echo $?
```

Expected: Exit code `2`, JSON contains `"decision":"block"`, risk match `sensitive-file-content-read`.

Direct reads of secret files (`cat ~/.ssh/id_ed25519`, `sed ... .env`, reads of `.aws/credentials`) are hard-blocked in every mode except audit. The risk match `sensitive-file-content-read` identifies the specific risk category.

SMOKE 5: GRAY-AREA PATH REFERENCE ASKS FOR APPROVAL

```
printf '%s' '{"hook_event_name":"PreToolUse","tool_name":"Bash","tool_input":{"command":"ls
-la ~/.docker/"}}' \
  | orca-guard claude-hook
echo $?
```

Expected: Exit code `0`, JSON contains `"permissionDecision":"ask"`, risk match `sensitive-file-reference`, risk action `approval_required`.

A directory listing of a sensitive path is gray-area — it doesn't read secrets, but it could be a precursor to reading them. In production mode, `PreToolUse` can surface an approval prompt. The `permissionDecision: ask` field tells Claude Code to show the human approval dialog.

SMOKE 6: SECRET MANAGER READ IS BLOCKED

```
printf '%s' '{"hook_event_name":"PreToolUse","tool_name":"Bash","tool_input":
{"command":"aws secretsmanager get-secret-value --secret-id prod/db"}}' \
  | orca-guard claude-hook
echo $?
```

Expected: Exit code `2`, risk match `aws-secrets-manager-read`.

Bulk secret manager reads (AWS Secrets Manager, GCP Secret Manager, Azure Key Vault, Vault, 1Password, Infisical) are hard-blocked. These commands dump secret values into the terminal output, which would then be captured in the transcript.

TRANSCRIPT UAT

The transcript UAT is the most important test. It verifies that Guard actually prevents secrets from reaching disk, not just from reaching the model provider.

Steps: 1. Start a new Claude Code session with Guard enabled. 2. Submit a canary token in a prompt. 3. Confirm the prompt is blocked. 4. Find the current Claude transcript JSONL file. 5. Search the transcript for the canary token.

Expected: The token is NOT present in the transcript. Guard's block reason is visible to the user. No API request reaches the Orca LLM gateway for the blocked turn.

This test is the proof of the dual-layer principle. If the hook blocks at `UserPromptSubmit`, the prompt text is never written to the transcript JSONL. The file watcher never sees it. The sink never stores it. The secret never enters the pipeline.

GATEWAY UAT

The gateway UAT uses a dead upstream (`http://127.0.0.1:9`) to prove the blocking behavior without depending on a real API:

```
# Start gateway with dead upstream
ORCA_GATEWAY_TOKEN=*** \
ORCA_GATEWAY_UPSTREAM=http://127.0.0.1:9 \
orca-llm-gateway --bind 127.0.0.1:19741
```

Test 1 — Block a synthetic model request:

```
CANARY="$(python3 -c 'import uuid; print("sk-ant...ard-" + uuid.uuid4().hex)')"
curl -sS -i \
  -H 'Authorization: Bearer ***' \
  -H 'Content-Type: application/json' \
  http://127.0.0.1:19741/v1/messages \
  --data "{\"messages\":[{\"role\":\"user\",\"content\":\"$CANARY\"}]}"
```

Expected: HTTP `403`, body contains `orca_guard_blocked`, rule id `anthropic-api-key`, no upstream connection attempt.

Test 2 — Auth enforcement:

```
curl -sS -i \
  -H 'Content-Type: application/json' \
  http://127.0.0.1:19741/v1/messages \
  --data '{"messages":[{"role":"user","content":"safe"}]}'
```

Expected: HTTP `401`, body contains `orca_gateway_unauthorized`.

Test 3 — Safe request forwards:

```
curl -sS -i \
  -H 'Authorization: Bearer ***' \
  -H 'Content-Type: application/json' \
  http://127.0.0.1:19741/v1/messages \
  --data '{"messages":[{"role":"user","content":"safe deployment summary"}]}'
```

Expected: HTTP `502`, body contains `orca_gateway_error`. The 502 proves the request passed Guard inspection and attempted upstream forwarding — the dead upstream at port 9 refuses the connection, producing the 502. If Guard had blocked the request, it would have returned 403 without attempting the upstream connection.

This is a clever test design. The 502 vs 403 distinction proves that safe requests are forwarded and blocked requests are not. You don't need a real upstream API to verify the gateway's behavior.

Configuration and Allow-List Policy

Guard's allow-list is intentionally narrow and **code-owned** in `crates/agentchron-agent/src/guard.rs`. There is no broad user-editable "allow this secret path" file. This is a deliberate design decision — and it is one of the most important security properties of the system.

Anti-pattern warning (AP5): A broad workstation-local allow-list would let a developer turn off the main secret prevention control. If an operator could create a file like `~/.config/orca-guard/allow-paths.txt` with `~/.ssh/` in it, Guard would become opt-in rather than opt-out. The design deliberately avoids this.

The current built-in allow-list behavior:

PATTERN	ALLOWED IN	REASON
`.env.example`, `.env.template`, `.env.sample` references	all modes	Template names are documentation, not secret stores
Source-code env identifiers (`process.env.NAME`, `import.meta.env.NAME`, `node.env`, `deno.env`)	all modes	These are code identifiers, not secret values
Property access (`getCloudflareContext().env`) in source edits	all modes	Application code, not filesystem secret paths
Path-only references or directory listings (`~/.docker/`)	`dev`; approval in `production` `PreToolUse`; block/report in `max` / `audit`	Useful for debugging but can drift into secret inspection
Deployment-only env-file loading (`source ~/.env; wrangler deploy`)	all modes, only with approved deploy command and no dump command	Supports sanctioned deploy workflows

Hard blocks in every mode except audit:

PATTERN	BEHAVIOR
Secret-shaped values (API keys, private keys, tokens, high-entropy credentials)	Block
Direct secret file content reads (`cat ~/.ssh/id_ed25519`, `sed ... .env`, `.aws/credentials` reads)	Block
Bulk secret manager reads (AWS, GCP, Azure, Vault, 1Password, Infisical)	Block
Env-file load followed by `printenv`, `env`, `export -p`, `declare -x`, or `set` dumping	Block

The Orca dashboard exposes a per-user Secrets Gate policy page at `/state-memory#/settings/secrets-gate`, backed by `GET/PATCH /v1/guard/settings`. This page can tune mode, gray-area approval behavior, signed Guard receipts, sync target, safe env-var identifier names, and approved broker handles. But it **must not store raw secret values** — the backend re-runs the secret detector over every submitted string and rejects secret-shaped values before persisting settings. Organization floor rules are shown read-only and cannot be patched from the user page.

The approved tuning workflow for false positives is:

1. Reproduce the false-positive with a synthetic payload. Do not use live secrets.
2. Run it in audit mode and confirm the production decision metadata.
3. If the payload contains only identifiers, source-code env access, path-only references, or sanctioned broker/deploy patterns, add a narrow code allow-list rule plus a unit test in `guard.rs`.
4. If the payload reads or prints secret contents, do not whitelist it. Route through SecureGit or agent-vault and keep values out of transcripts.

This workflow ensures that every allow-list exception is reviewed, tested, and documented in code. It is the Code-Owned, Narrow Allow-Lists pattern (P8) in practice.

LLM Gateway as Egress Defense

The `orca-llm-gateway` is the network egress defense layer. It is an Anthropic-compatible HTTP gateway that scans model requests before forwarding them upstream.

To use it, point Claude Code at the gateway instead of directly at Anthropic:

```
export ANTHROPIC_BASE_URL=http://127.0.0.1:19741
export ANTHROPIC_AUTH_TOKEN=***
```

The gateway intercepts every request to `/v1/messages` (and other Anthropic API endpoints), runs the full Guard secret detection over the request body, and either forwards or blocks:

- **Blocked:** Returns `403` with `orca_guard_blocked` and the rule ID. No upstream connection is attempted.
- **Safe:** Forwards to the upstream provider with the original request body and headers.
- **Auth failure:** Returns `401` with `orca_gateway_unauthorized`. Auth is checked before content scanning.

The gateway is a second layer behind the hooks. In a properly configured Claude Code installation, the hooks block secrets before they reach the API request stage. The gateway catches anything the hooks miss — for example, secrets injected by non-Claude clients that don't have hooks configured, or secrets that appear in tool results that are fed back into the model context.

The gateway UAT with a dead upstream (`http://127.0.0.1:9`) proves the blocking behavior without needing a real API key. The 403 vs 502 distinction is the key signal:

- `403` = Guard blocked the request before attempting upstream connection
- `502` = Guard allowed the request, attempted upstream forwarding, upstream was unreachable

This means you can verify the gateway's blocking behavior in a completely isolated test environment.

For production use, the gateway is configured with a real upstream:

```
export ORCA_GATEWAY_UPSTREAM=https://api.anthropic.com
export ORCA_GATEWAY_UPSTREAM_API_KEY=*** from vault>
```

The upstream API key should come from a vault or broker, not from a plaintext environment variable. The gateway never logs the raw token, and blocked requests never reach the upstream.

Guard Integration Points

Guard composes with the rest of the Orca platform at several integration points:

CLAUDE CODE HOOKS

The Linux bootstrap installer configures Guard in `~/.claude/settings.json`:

CLAUDE EVENT	ORCA COMMAND	PURPOSE
`UserPromptSubmit`	`orca-guard claude-hook`	Block prompts that leak credentials
`PreToolUse`	`orca-guard claude-hook`	Block risky tool use before execution
`SessionStart`	`agentchron-claude-hook`	Ingest start-of-session metadata
`PostToolUse`	`agentchron-claude-hook`	Ingest tool results and decisions
`Stop`	`agentchron-claude-hook`	Ingest final turn/session metadata
`SessionStart`	`orca-session-rules.py`	Inject local Orca session rules

The installer only adds Guard hooks when `--configure-claude-guard` is supplied:

```
./install.sh --no-env --configure-claude-guard --guard-mode production
```

Switching modes without reinstalling:

```
./install.sh --skip-bin-install --no-env --configure-claude-guard --guard-mode dev
./install.sh --skip-bin-install --no-env --configure-claude-guard --guard-mode audit
./install.sh --skip-bin-install --no-env --configure-claude-guard --guard-mode production
```

MCP GATEWAY INTEGRATION

The Orca MCP Governance Gateway integrates Guard argument scanning in `max`, `production`, `dev`, and `audit` modes. When an MCP tool call passes through the gateway, Guard scans the tool arguments for secrets before the RBAC enforcement decision. This adds secret detection to the MCP tool call path, complementing the `PreToolUse` hook for Claude Code's native tools.

SESSION RULES

`deploy/bin/orca-session-rules.py` runs alongside Guard at `SessionStart`. It injects local Orca session rules into the Claude Code context — operational constraints, approved workflow guidance, and scope information. The session rules hook fails open (if it can't reach the sink, it doesn't block the session) and caps graph calls at 2 seconds with negative caching for timeouts. This is the Synchronous Enforcement, Asynchronous Telemetry pattern (P11) — the hook doesn't block on slow queries.

COMPOSITION WITH SINK SANITIZER

Guard is the first layer in the defense-in-depth chain, but it is not the only one. The full composition is:

```
Guard hooks (block before transcript + before tool execution)
  → Agent sanitizer (parse_line → sanitize_json_line, redact before transport)
    → TCP push sanitizer (sanitize_json_line in push_wire, second pass)
      → Sink plugin (SecretsFilterPlugin, detect before sink sanitize)
        → Sink sanitizer (sanitize_event, belt-and-suspenders)
```

Each layer is independent. Guard blocks; the sanitizers redact. Guard runs synchronously on the hot path; the sink plugin and sanitizer run asynchronously at ingest time. Guard's findings are metadata-only (rule ID, label, severity, advice) — never the matched secret value. The sink's plugin findings are similarly metadata-only. This means the entire chain from hook to storage produces evidence of what was caught without ever persisting the secret itself.

FOUNDRY SILVER CLEANING

The Data Foundry adds two more sanitization passes during silver cleaning: a realtime sanitizer pass and an offline deny-list + entropy check. These run during the bronze → silver transformation and catch anything that slipped through the real-time pipeline. The foundry's cleaning order is: secret redaction → PII → license → dedup → boilerplate → outcome → decisions. Secret redaction is first because everything downstream depends on clean text.

Patterns Developed in This Chapter

DEFENSE-IN-DEPTH SANITIZATION (P1)

Guard is the first interception point in the defense-in-depth chain. The chain has five layers: Guard hooks → agent sanitizer → TCP push sanitizer → sink plugin → sink sanitizer. Each layer is independently sufficient for its boundary. Guard prevents transcript persistence; the sanitizers prevent durable storage. The pattern is that no single layer is trusted to have done it correctly — the sink re-runs detection even after the agent has already sanitized.

CODE-OWNED, NARROW ALLOW-LISTS (P8)

Guard's allow-list is narrow and code-owned in `guard.rs`. There is no user-editable allow-list file. Exceptions are added as reviewed code patterns with unit tests. The dashboard policy page can tune mode and behavior but cannot store raw secret values. This prevents the most likely self-degradation path: an operator adding their secret path to a local allow-list and forgetting about it.

SYNCHRONOUS ENFORCEMENT, ASYNCHRONOUS TELEMETRY (P11)

Guard's hook decisions are synchronous — the block or allow happens on the hot path, before the prompt is written or the tool executes. The governance receipts and findings are asynchronous — they are written to the receipt log and the sink without blocking the decision. This keeps the hook fast (the operator doesn't wait for disk I/O) while preserving the evidence trail. The same pattern governs the MCP gateway (enforcement is synchronous, receipts are queued to a serialized writer thread) and the sink (SQLite insert is synchronous, Neo4j/Qdrant writes are best-effort).

Anti-Patterns Addressed

TREATING AUDIT MODE AS SAFE FOR SECRET-BEARING WORK (AP4)

Audit mode allows actions that production mode would block, reporting `would_block_in_production=true`. It is for QA and tuning — you run it to see what production would catch without actually blocking. Using audit mode for actual secret-bearing work means secrets reach the model provider and are written to transcripts, with only a metadata note that they would have been blocked. The mitigation is documentation (the QA doc is explicit), training, and dashboard warnings. The mode smoke tests reinforce this: audit mode exits 0 for the same prompt that production mode exits 2 for.

BROAD WORKSTATION-LOCAL ALLOW-LISTS (AP5)

The deliberate avoidance of user-editable allow-lists is one of Guard's most important security properties. A broad local allow-list would let a developer turn off the main control by adding `~/.ssh/` or `.env` to a file. The design keeps allow-lists in code, where changes are reviewed, tested, and version-controlled. Enterprise policy bundles are planned but must not allow raw secret values or direct secret-content reads. The sanctioned path for credential access is through SecureGit or agent-vault brokers, not through allow-listed file reads.

Risk Matching: The Guard Rule Engine

Guard's decision logic goes beyond simple secret-pattern matching. It includes a risk matching engine that classifies tool inputs into risk categories. The risk categories visible in the smoke tests are:

RISK MATCH	TRIGGER	MODE BEHAVIOR
`anthropic-api-key`	`sk-ant-` pattern in prompt or tool input	Block in all modes except audit
`sensitive-file-content-read`	`cat`, `sed`, `tail`, `head` on known secret files (`~/.ssh/id_*`, `.env`, `.aws/credentials`)	Block in production/max/dev; report in audit
`sensitive-file-reference`	`ls`, `stat`, `find` on sensitive directories (`~/.docker/`, `~/.ssh/`)	Ask in production PreToolUse; allow in dev; block/report in max/audit
`aws-secrets-manager-read`	`aws secretsmanager get-secret-value`	Block in all modes except audit
`gcp-secret-manager-read`	`gcloud secrets get`	Block in all modes except audit
`azure-key-vault-read`	`az keyvault secret show`	Block in all modes except audit
`vault-read`	`vault read`, `vault kv get`	Block in all modes except audit
`env-dump-after-load`	`source ~/.env; env` / `printenv` / `export -p`	Block in all modes except audit

The risk matching engine examines the tool name and tool input JSON. For `Bash` tool calls, it parses the `command` field. For `Read` and `Edit` tool calls, it examines the `file_path` field. For MCP tool calls, it examines the tool name and arguments.

The distinction between `sensitive-file-content-read` (hard block) and `sensitive-file-reference` (gray area) is important. Reading the contents of `~/.ssh/id_ed25519` dumps the private key into the terminal output, which gets captured in the transcript. Listing the contents of `~/.ssh/` reveals filenames but not key contents. The former is always blocked; the latter is gray-area because it could be legitimate debugging.

The env-dump-after-load pattern catches a subtle exfiltration vector: an operator might legitimately `source ~/.env` to load environment variables for a deploy, but following that with `env` or `printenv` dumps all loaded secrets into the terminal. Guard allows `source ~/.env; wrangler deploy` (sanctioned deploy workflow) but blocks `source ~/.env; env` (secret dump).

Exit Code Protocol: Design Rationale

The exit code protocol is the interface between Guard and Claude Code. Each exit code and stdout combination has a specific operational meaning, and the design choices behind them are worth examining in detail.

EXIT 0 WITH NO STDOUT (ALLOW)

When Guard allows a prompt or tool use, it prints nothing and exits 0. This is not the obvious choice — one might expect Guard to print `{"decision":"allow"}` for symmetry with the block case. The reason for silence is that Claude Code injects hook stdout into the conversation context. If Guard printed a JSON allow decision on every safe prompt, the conversation would accumulate JSON noise after every turn, polluting the context window and confusing the model.

The `--print-allow` flag exists for debugging: it overrides the silence behavior and prints the allow decision as JSON. This is used in the smoke tests to verify that the allow path produces the correct decision metadata without relying on the absence of output.

EXIT 2 WITH JSON (BLOCK)

When Guard blocks, it prints Claude decision JSON and exits 2. Claude Code interprets exit 2 as "block this action and show the user the JSON output." The JSON contains:

- `decision: "block"` — the block verdict
- `rule_id` — which pattern or risk match triggered the block (e.g., `anthropic-api-key`, `sensitive-file-content-read`)
- `label` — human-readable description
- `advice` — remediation guidance
- `severity` — always `"high"` for secret detections

The block JSON must not contain the matched secret value. This is verified in the transcript UAT — the canary token must not appear in the transcript, and by extension it must not appear in the block JSON that Claude Code displays to the user.

EXIT 0 WITH `PERMISSIONDECISION: ASK` (GRAY-AREA PRETOOLUSE)

This is the most nuanced exit code combination. Exit 0 means "don't block," but the JSON payload contains `hookSpecificOutput.permissionDecision: "ask"`, which tells Claude Code to surface a human approval dialog before executing the tool.

This is specific to `PreToolUse` because Claude Code has a documented interactive approval shape for tool use — it can show the user "Allow this tool use?" with details about the tool and the risk match. The human can then approve or deny, and Claude Code proceeds accordingly.

The gray-area category exists because some tool uses are risky but not definitively malicious. `ls -la ~/.docker/` doesn't read secrets, but it's a precursor to reading them. In production mode, Guard escalates this to human approval rather than hard-blocking, giving the operator the context to make an informed decision.

EXIT 2 FOR GRAY-AREA USERPROMPTSUBMIT

`UserPromptSubmit` does not have a documented interactive approval shape. There is no way for the hook to say "ask the user about this prompt" — the hook can only allow or block. So gray-area prompt text blocks with exit 2, and the operator must either rephrase the prompt or use `/approve-paste` to grant a one-shot exception.

This is a limitation of the Claude Code hook protocol, not a design choice by Guard. If Claude Code adds an interactive approval shape for `UserPromptSubmit` in the future, Guard can be updated to use `permissionDecision: ask` for gray-area prompts, matching the `PreToolUse` behavior.

The `scan-text` Binary: CI and Pipeline Integration

The `orca-guard scan-text` binary is the standalone scanner that reads raw text from stdin and prints a JSON decision report. It uses the same detection engine as the hook binary but without the Claude Code exit code protocol — it always exits 0 and prints the full decision JSON regardless of the verdict.

This makes it suitable for CI pipelines, pre-commit hooks, and any context where you need to scan arbitrary text without the Claude Code integration:

```
# Scan a file for secrets before committing
cat config.json | orca-guard scan-text

# Use in a pre-commit hook
if echo "$ staged_content" | orca-guard scan-text | jq -e '.decision == "block"' > /dev/
null; then
    echo "Secret detected — commit blocked"
    exit 1
fi
```

The `scan-text` binary supports the same modes as the hook binary (`--mode production|dev|audit|max`). In audit mode, it reports `would_block_in_production=true` without blocking, which is useful for assessing the secret exposure in a corpus of text without disrupting workflows.

Guard and the Sanitizer: Division of Labor

Guard and the sanitizer engine (Chapter 6) are complementary but distinct. Understanding their division of labor is essential for operators:

ASPECT	GUARD	SANITIZER
Position	Before model, before disk	Before transport, at storage
Action	Block or allow	Redact (replace with `[REDACTED]`)
Timing	Synchronous on hot path	Synchronous at ingest
Output	Exit code + JSON decision	Redacted text + metadata findings
Reversibility	Irreversible (blocked action never happens)	Irreversible (redacted text cannot be unredacted)
Scope	Prompts and tool inputs	All event text, tool inputs, envelope extras

Guard blocks; the sanitizer redacts. Guard prevents the secret from entering the pipeline; the sanitizer removes secrets that are already in the pipeline. Both produce metadata-only findings — rule ID, label, severity, advice — but never the matched value.

The composition is sequential: if Guard blocks, the sanitizer never sees the secret because it never enters the transcript. If Guard allows (or is not configured), the sanitizer catches the secret at ingest time and redacts it. The five-layer defense-in-depth chain (Guard hooks → agent sanitizer → TCP push sanitizer → sink plugin → sink sanitizer) ensures that a secret must pass through all five layers to reach durable storage — and each layer is independently sufficient for its boundary.

Claude Code Hook Configuration

The installer writes Guard hooks into `~/.claude/settings.json`. The resulting configuration looks like:

```
{
  "hooks": {
    "UserPromptSubmit": [
      {
        "command": "orca-guard claude-hook",
        "timeout": 5000
      }
    ],
    "PreToolUse": [
      {
        "command": "orca-guard claude-hook",
        "timeout": 5000
      }
    ]
  }
}
```

The `UserPromptSubmit` hook fires when the user submits a prompt. Guard scans the prompt text for secret-shaped values and risky content. If a secret is detected, the hook exits 2 and Claude Code shows the block reason. The prompt is never written to the transcript.

The `PreToolUse` hook fires before any tool execution — Bash commands, file reads, file writes, MCP tool calls. Guard scans the tool name and tool input JSON. If a direct secret-file read is detected, the hook exits 2 and the tool is not executed. If a gray-area path reference is detected, the hook exits 0 with `permissionDecision: ask`, and Claude Code surfaces a human approval dialog.

The timeout (5 seconds) ensures that Guard doesn't block the session indefinitely if something goes wrong. Guard's scanning is fast (regex matching over a prompt string), so the timeout is generous — it's a safety net, not a tight deadline.

Installer Safety: Opt-In Guard Configuration

The installer only adds Guard hooks when explicitly requested:

```
./install.sh --no-env --configure-claude-guard --guard-mode production
```

The `--configure-claude-guard` flag is required. Without it, the installer does not modify `~/.claude/settings.json`. This is a safety measure — Guard changes the behavior of Claude Code by adding blocking hooks, and that change should be explicit, not implicit.

The installer also creates the `/approve-paste` slash command under `~/.claude/commands/` so operators can grant one-shot paste approvals from within a Claude Code session.

The QA document recommends: "Do not enable fleet-wide until the UAT transcript checks pass." This is a phased rollout approach:

1. **QA host** — Install Guard on a dedicated QA host, run all six smoke tests, run the transcript UAT, run the gateway UAT.

2. **Test Claude profile** — Install Guard on a developer's machine with a test Claude profile (separate from their working profile).
3. **Individual rollout** — Install Guard on individual operator machines after they've been briefed on the mode behavior and `/approve-paste` workflow.
4. **Fleet rollout** — Only after individual rollout is stable, enable fleet-wide.

Guard Receipts and Audit Trails

Guard's block and allow decisions are not just exit codes — they produce governance receipts that feed into the Orca evidence trail. When Guard blocks a prompt at `UserPromptSubmit`, the block decision is recorded as a security boundary event. When Guard asks for approval at `PreToolUse`, the approval request is recorded. When an operator uses `/approve-paste`, the one-shot grant is recorded with the reason and timestamp.

These records are transparent in the audit trail. An operator reviewing a session in the web UI can see: - Which prompts were blocked and why (rule ID, risk match) - Which tool uses required approval and whether the operator approved or denied - Which `/approve-paste` grants were created, when they expired, and what they authorized

The governance receipt path still records that a prompt would have been blocked without the human grant. The approval doesn't suppress the finding — it adds a human-authorization annotation. This means the audit trail shows both the risk (what Guard would have blocked) and the authorization (who approved it and why).

OpenBrain and SecureGit Guard Posture

Guard's risk matching extends beyond standard secret files to companion modules in the Orca ecosystem:

OPENBRAIN SENSITIVE SURFACES

SURFACE	GUARD BEHAVIOR
`~/.openbrain-vault`	Direct reads hard-blocked; path-only requires approval
`~/.openbrain-vault/.openbrain/mentions/*.jsonl`	Hard-blocked (agents should use OpenBrain MCP/read APIs)
`~/.openbrain/cursors`	Hard-blocked
`~/.claude/settings.json` / `~/.claude.json`	Direct reads hard-blocked; writes require human approval in production

SECUREGIT SENSITIVE SURFACES

SURFACE	GUARD BEHAVIOR
`~/.config/securegit/credentials.json`	Direct reads hard-blocked; path-only requires approval
`~/.local/share/securegit/security-events`	Direct reads hard-blocked

SANCTIONED BROKER PATH

SecureGit brokered secret execution is intentionally allowed when it uses handles and no raw values:

```
securegit secret run --with-secret NAME=handle -- <cmd>
```

This is the preferred path for operations that need scoped credentials — Orca Data Factory ingest, GraphRAG pipelines, cloud deploys, customer bootstrap commands. Guard recognizes the `securegit secret run` pattern and allows it because the secret value is never exposed in the command arguments or terminal output. The broker injects the secret into the child process's environment without echoing it.

The wake subject safety for OpenBrain (`openbrain.wakeup.<agent>`) is also metadata-only — the payload is only `{agent, at, src}`, a nudge. It must never include message bodies, secrets, tool arguments, or executable instructions.

Conclusion

Orca Guard is the pre-model, pre-disk secret prevention layer. Its dual-layer principle — blocking at Claude Code hooks before transcript persistence and at the LLM gateway before provider egress — solves two problems that a network gateway alone cannot. The three binaries (scan-text, claude-hook, llm-gateway) cover three integration contexts: standalone scanning, Claude Code hooks, and HTTP gateway. The four modes (production, dev, audit, max) provide graduated enforcement from default protection to deny-all.

The `/approve-paste` mechanism provides a narrow, one-shot, time-limited human grant for legitimate secret paste workflows without creating a persistent hole. The QA/UAT process — six smoke tests, transcript UAT, gateway UAT — provides concrete verification that the blocking behavior works end-to-end, from hook exit code to transcript absence to gateway HTTP status.

The code-owned allow-list philosophy is the structural defense against self-degradation. By keeping exceptions in reviewed code rather than in user-editable files, Guard maintains its security properties even when operators are under pressure to "just let this one through." The approved tuning workflow — reproduce with synthetic payload, run in audit mode, add code rule with test — ensures that every exception is deliberate, documented, and testable.

Guard composes with the sink sanitizer, the TCP push sanitizer, the MCP gateway's argument scanning, and the foundry's silver cleaning to form a five-layer defense-in-depth chain. No single layer is trusted to have done it correctly. The sink re-runs detection even after the agent has already sanitized. The foundry re-runs detection even after the sink has already sanitized. This is the security posture of a platform that treats AI work as regulated operational activity — and regulated activity demands defense in depth, not defense at a single point.

Chapter 6: Secret Sanitization

Origin and Architecture

The secret sanitization engine lives in `crates/agentchron-core/src/sanitizer.rs`. The module header states its lineage plainly:

```
//! Redact credentials from session content before it leaves the host.
//!
//! Patterns originally ported from `securegit/src/mcp/sanitizer.rs` and
//! extended with the AI provider tokens common in Claude Code transcripts
//! (Anthropic, OpenAI, ElevenLabs, Hugging Face).
```

The patterns were ported from SecureGit's MCP sanitizer and extended with AI-provider token formats that are common in Claude Code transcripts but rare in traditional git repositories. The filter version is stamped as a constant:

```
pub const FILTER_VERSION: &str = "agentchron-secrets-filter-v1";
```

This version string travels in every findings metadata block, allowing downstream consumers to know which filter version produced a given set of redactions. This is the Reproducibility Envelope pattern (P12) applied to security findings — if a new pattern is added in v2, consumers can distinguish v1 findings from v2 findings and re-scan old data if needed.

The architecture follows the **local-before-transport** principle (P2): content is sanitized on the host where it originates, before any bytes leave for the central sink. The `agentchron-agent` watcher reads JSONL session files, sanitizes each line through `sanitize_json_line()`, and only then ships to the sink. The sink performs a belt-and-suspenders second sanitization pass on ingest, ensuring that direct clients who skip the agent parser cannot bypass secret reporting.

The module is deliberately I/O-free. It depends only on `regex`, `serde`, `serde_json`, and `once_cell`. No network, no filesystem, no database. This makes it testable in isolation and portable across all three downstream crates (agent, sink, web) without pulling in unwanted dependencies.

The 18-19 Detection Patterns

The `TOKEN_PATTERNS` static vector contains 19 compiled regex patterns, each wrapped in a `TokenPattern` struct:

```
struct TokenPattern {
    rule_id: &'static str,
    label: &'static str,
    severity: &'static str,
    advice: &'static str,
    regex: Regex,
}
```

Every pattern has severity `"high"` and the same standardized advice text:

> "Assume this credential may have been exposed in a Claude Code session. Review the source context, rotate the credential if it is live, and remove it from shell history, config files, and commit history as needed."

The patterns are organized by category:

GITHUB (5 PATTERNS)

```
token_pattern("github-classic-pat", "GitHub classic PAT", r"ghp_[A-Za-z0-9]{36,}"),
token_pattern("github-oauth-token", "GitHub OAuth token", r"gho_[A-Za-z0-9]{36,}"),
token_pattern("github-user-token", "GitHub user token", r"ghu_[A-Za-z0-9]{36,}"),
token_pattern("github-server-token", "GitHub server token", r"ghs_[A-Za-z0-9]{36,}"),
token_pattern("github-fine-grained-pat", "GitHub fine-grained PAT", r"github_pat_[A-Za-
z0-9_]{22,}"),
```

GitHub uses distinct prefixes for each token type: `ghp_` for classic PATs, `gho_` for OAuth tokens, `ghu_` for user-to-server tokens, `ghs_` for server-to-server tokens, and `github_pat_` for fine-grained PATs. Each requires at least 36 alphanumeric characters (22 for fine-grained) after the prefix, which distinguishes real tokens from short references like `ghp_test`.

GITLAB (2 PATTERNS)

```
token_pattern("gitlab-personal-access-token", "GitLab personal access token", r"glpat-[A-
Za-z0-9\-_.]{20,}"),
token_pattern("gitlab-deploy-token", "GitLab deploy token", r"gldt-[A-Za-z0-9\-_.]{20,}"),
```

GitLab PATs use the `glpat-` prefix; deploy tokens use `gldt-`. Both allow 20+ characters of alphanumeric, hyphen, underscore, and dot characters.

AI PROVIDERS (4 PATTERNS)

```
token_pattern("anthropic-api-key", "Anthropic API key", r"sk-ant-[A-Za-z0-9\-_]{20,}"),
token_pattern("openai-api-key", "OpenAI API key", r"sk-(?:proj-)?[A-Za-z0-9_\-]{20,}"),
token_pattern("elevenlabs-api-key", "ElevenLabs API key", r"\bsk_[A-Za-z0-9]{32,}"),
token_pattern("huggingface-token", "Hugging Face token", r"\bhf_[A-Za-z0-9]{32,}"),
```

These are the patterns added beyond SecureGit's original set. They target the AI provider tokens that commonly appear in Claude Code transcripts — operators paste API keys into prompts, config files, or shell commands, and these keys follow distinct prefix patterns.

The Anthropic key pattern (`sk-ant-`) and the OpenAI key pattern (`sk-` or `sk-proj-`) share a prefix namespace. The shadowing logic (discussed below) prevents double-classification.

ElevenLabs uses `sk_` (underscore, not hyphen) with 32+ characters. Hugging Face uses `hf_` with 32+ characters. The word boundary (`\b`) prevents partial matches within longer strings.

CLOUD (1 PATTERN)

```
token_pattern("aws-access-key-id", "AWS access key ID", r"\bAKIA[0-9A-Z]{16}\b"),
```

AWS access key IDs are 20-character strings that start with `AKIA` followed by 16 uppercase alphanumeric characters. The word boundaries prevent false positives within longer strings.

PAYMENTS (2 PATTERNS)

```
token_pattern("stripe-secret-key", "Stripe secret key", r"\bsk_(?:test|live)_[A-Za-z0-9]
{16,}\b"),
token_pattern("stripe-webhook-secret", "Stripe webhook secret", r"\bwhsec_[A-Za-z0-9]
{16,}\b"),
```

Stripe secret keys use `sk_test_` or `sk_live_` followed by 16+ alphanumeric characters. Webhook secrets use `whsec_`. Note the potential prefix collision with OpenAI's `sk-` (hyphen) — Stripe uses `sk_` (underscore), so the patterns are distinct.

GENERIC (5 PATTERNS)

```
token_pattern("authorization-header-token", "authorization header token",
    r"(?i)(Bearer|Token|PRIVATE-TOKEN:?)\s+[A-Za-z0-9\-_.]{20,}"),
token_pattern("url-embedded-credential", "URL embedded credential",
    r"://[^\s/]+:[^@\s/]+@"),
token_pattern("pem-private-key-block", "PEM private key block",
    r"(?s)-----BEGIN [A-Z0-9 ]*PRIVATE KEY-----.*?-----END [A-Z0-9 ]*PRIVATE KEY-----"),
token_pattern("base64-encoded-pem-block", "base64-encoded PEM block",
    r"\bLS0tLS1CRUdJTi(?:[A-Za-z0-9+/=]{32,})"),
token_pattern("jwt", "JWT",
    r"eyJ[A-Za-z0-9_-]{10,}\.eyJ[A-Za-z0-9_-]{10,}\.[A-Za-z0-9_\-]{10,}"),
```

The authorization header pattern catches `Bearer <token>`, `Token <token>`, and `PRIVATE-TOKEN: <token>` formats with case-insensitive matching. The URL-embedded credential pattern catches `://user:pass@host` syntax common in git remote URLs and database connection strings.

The PEM block pattern uses `(?s)` (single-line mode) to match across newlines, capturing the full `-----BEGIN ... PRIVATE KEY-----` to `-----END ... PRIVATE KEY-----` block. The base64-encoded PEM pattern catches the base64 representation of the `-----BEGIN` prefix (`LS0tLS1CRUdJTi` is base64 for `-----BEGIN`), which appears when PEM keys are embedded in JSON or environment variables without newlines.

The JWT pattern matches the three-part `header.payload.signature` structure with `eyJ` prefixes (base64-encoded `{"` — the start of a JSON object). This is a structural match, not a content match — it identifies the shape of a JWT without validating its contents.

Shadowing Logic

The `is_shadowed_detection()` function prevents double-classification when multiple patterns match the same token:

```
fn is_shadowed_detection(rule_id: &str, matched: &str) -> bool {
    // OpenAI keys share the broad sk-* prefix. More specific provider
    // patterns should win when they match the same token.
    rule_id == "openai-api-key" && matched.starts_with("sk-ant-")
}
```

The problem: OpenAI's pattern `sk-(?:proj-)?[A-Za-z0-9_\-]{20,}` is broad enough to match Anthropic keys (`sk-ant-...`). Without shadowing, an Anthropic key would be classified as both `anthropic-api-key` and `openai-api-key`, producing two findings for one secret.

The solution: when the OpenAI pattern matches a string that starts with `sk-ant-`, the OpenAI finding is suppressed. The more specific Anthropic pattern wins. This is verified by a dedicated test:

```
#[test]
fn anthropic_key_is_not_double_classified_as_openai() {
    let findings = detect("ANTHROPIC_API_KEY=***!(findings.iter().any(|f| f.rule_id ==
"openai-api-key"));
}
```

The `detect()` function applies the shadowing filter during iteration:

```
pub fn detect(input: &str) -> Vec<SecretDetection> {
    let mut counts: BTreeMap<&'static str, (&'static TokenPattern, usize)> =
BTreeMap::new();
    for pattern in TOKEN_PATTERNS.iter() {
        let count = pattern
            .regex
            .find_iter(input)
            .filter(|m| !is_shadowed_detection(pattern.rule_id, m.as_str()))
            .count();
        if count > 0 {
            counts
                .entry(pattern.rule_id)
                .and_modify(|(_, existing)| *existing += count)
                .or_insert((pattern, count));
        }
    }
    // ...
}
```

The `BTreeMap` deduplicates by `rule_id` — if the same pattern matches multiple times in the input, the counts are accumulated. The `SecretDetection` result carries `occurrence_count` rather than individual match positions, which is sufficient for rotation reports without revealing the matched values.

JSON-Aware Sanitization

The `sanitize_json_line()` function is the primary entry point for sanitizing JSONL session events. It preserves JSON shape while redacting secrets:

```
pub fn sanitize_json_line(input: &str) -> String {
    let detections = detect(input);
    match serde_json::from_str::<Value>(input) {
        Ok(value) => {
            let mut value = sanitize_json_value(value);
            stamp_filter_metadata(&mut value, &detections);
            serde_json::to_string(&value).unwrap_or_else(|_| sanitize(input))
        }
        Err(_) => sanitize(input),
    }
}
```

The function follows a three-step process:

1. **Detect** — Run `detect()` on the raw input string to collect `SecretDetection` findings.
2. **Sanitize** — Parse as JSON, recursively sanitize all string values, re-serialize.
3. **Stamp** — Add the `agentchron_secret_filter` metadata block to the JSON object.

If the input is not valid JSON, it falls back to plain-text `sanitize()`, which replaces all matches with `[REDACTED]` without preserving any structure.

The recursive sanitizer walks the entire JSON tree:

```
pub fn sanitize_json_value(value: Value) -> Value {
    match value {
        Value::String(s) => Value::String(sanitize(&s)),
        Value::Array(items) =>
Value::Array(items.into_iter().map(sanitize_json_value).collect()),
        Value::Object(object) => Value::Object(
            object
                .into_iter()
                .map(|(key, value)| (key, sanitize_json_value(value)))
                .collect(),
        ),
        other => other,
    }
}
```

Strings are sanitized in place. Arrays and objects are recursively sanitized. Numbers, booleans, and nulls pass through unchanged. This ensures that secrets embedded in nested JSON structures — a tool input object with a `command` field that contains a bearer token, for example — are caught regardless of depth.

The `stamp_filter_metadata()` function adds the findings metadata to the JSON object:

```
fn stamp_filter_metadata(value: &mut Value, detections: &[SecretDetection]) {
    if detections.is_empty() {
        return;
    }
    let Value::Object(object) = value else {
        return;
    };

    let redactions: Vec<Value> = detections
        .iter()
        .map(|d| {
            json!({
                "rule_id": d.rule_id,
                "label": d.label,
                "severity": d.severity,
                "occurrence_count": d.occurrence_count,
                "advice": d.advice,
            })
        })
        .collect();

    object.insert(
        "agentchron_secret_filter".to_string(),
        json!({
            "version": FILTER_VERSION,
            "redactions": redactions,
        }),
    );
}
```

The metadata block is stamped only when detections are non-empty and only on JSON objects (not arrays or primitives). The block contains:

```
{
  "agentchron_secret_filter": {
    "version": "agentchron-secrets-filter-v1",
    "redactions": [
      {
        "rule_id": "anthropic-api-key",
        "label": "Anthropic API key",
        "severity": "high",
        "occurrence_count": 1,
        "advice": "Assume this credential may have been exposed..."
      }
    ]
  }
}
```

This metadata travels downstream with the event. Consumers — the sink, the web UI, the foundry, rotation report generators — can read the `agentchron_secret_filter` block to know what was redacted, how many times, and what to do about it. The metadata never includes the matched secret value or any reversible representation. This is the Findings as Metadata, Never Values principle.

The test suite verifies this explicitly:

```
#[test]
fn json_line_sanitizer_stamps_metadata_without_secret_value() {
    let raw = r#"{"type":"user","sessionId":"s1","message":
{"role":"user","content":"ANTHROPIC_API_KEY=***;
    assert!(!clean.contains("sk-ant-"));
    let value: Value = serde_json::from_str(&clean).unwrap();
    assert_eq!(
        value["agentchron_secret_filter"]["version"].as_str(),
        Some(FILTER_VERSION)
    );
    assert_eq!(
        value["agentchron_secret_filter"]["redactions"][0]["rule_id"].as_str(),
        Some("anthropic-api-key")
    );
}
```

The test asserts two things: the secret value is not present in the sanitized output, and the metadata block is present with the correct rule ID and filter version. This is the contract: redact the value, preserve the metadata.

Sink-Side Plugin Framework

The sink has a minimal ingest plugin framework in `crates/agentchron-sink/src/plugins.rs`. The `IngestPlugin` trait defines the interface:

```
pub trait IngestPlugin: Send + Sync {
    fn name(&self) -> &'static str;
    fn inspect(&self, event: &Event) -> Vec<EventFinding>;
}
```

The `PluginManager` is configured via CSV (`AGENTCHRON_PLUGINS=secrets` by default):

```
impl PluginManager {
    pub fn from_csv(csv: &str) -> Self {
        let mut plugins: Vec<Box<dyn IngestPlugin>> = Vec::new();
        for name in csv.split(',').map(str::trim).filter(|s| !s.is_empty()) {
            match name {
                "secrets" | "secrets-filter" =>
plugins.push(Box::new(SecretsFilterPlugin)),
                "none" | "off" => {}
                _ => {}
            }
        }
        Self { plugins }
    }
}
```

The `SecretsFilterPlugin` re-runs `sanitizer::detect()` over event text and tool inputs:

```
impl IngestPlugin for SecretsFilterPlugin {
    fn name(&self) -> &'static str {
        "secrets-filter"
    }

    fn inspect(&self, event: &Event) -> Vec<EventFinding> {
        let mut out = Vec::new();
        if let Some(text) = event.text.as_deref() {
            out.extend(secret_findings_from_text(text));
        }
        for tool in &event.tool_uses {
            out.extend(secret_findings_from_text(&tool.input.to_string()));
        }
        out
    }
}
```

Each detection is mapped to an `EventFinding`:

```
fn secret_findings_from_text(text: &str) -> Vec<EventFinding> {
    sanitizer::detect(text)
        .into_iter()
        .map(|d| EventFinding {
            plugin: "secrets-filter".to_string(),
            rule_id: d.rule_id,
            category: "secret".to_string(),
            severity: d.severity,
            summary: format!("{} detected during sink-side plugin scan", d.label),
            advice: d.advice,
            occurrence_count: d.occurrence_count,
        })
        .collect()
}
```

Findings are deduplicated by (plugin, rule_id, category) with summed occurrence counts:

```
fn dedupe_findings(findings: Vec<EventFinding>) -> Vec<EventFinding> {
    let mut out: Vec<EventFinding> = Vec::new();
    for finding in findings {
        if let Some(existing) = out.iter_mut().find(|existing| {
            existing.plugin == finding.plugin
                && existing.rule_id == finding.rule_id
                && existing.category == finding.category
        }) {
            existing.occurrence_count += finding.occurrence_count;
        } else {
            out.push(finding);
        }
    }
    out
}
```

The critical ordering in the sink's `ingest_one()` function is:

```
pub async fn ingest_one(state: &AppState, mut evt: Event) -> Result<()> {
    // Plugins run before the sink's final sanitizer pass so direct
    // clients cannot bypass secret reporting by skipping the agent parser.
    evt.findings = state.plugins.inspect(&evt);

    // Belt-and-suspenders sanitize at the sink as well.
    sanitize_event(&mut evt);

    let event_id = state.storage.insert_event(&evt).await?;
    // ...
}
```

The comment is the key design statement: **plugins run before the sink's final sanitizer pass so direct clients cannot bypass secret reporting by skipping the agent parser.** This is a trust-boundary design. The sink does not trust upstream clients to have sanitized correctly. If a client sends raw events directly to `POST /v1/events` without using the agent parser, the sink's plugin still runs `detect()` and produces findings. The subsequent `sanitize_event()` then redacts the actual values.

This ordering means that findings are produced from the raw (pre-sanitization) text, which is the correct behavior — you want to know what was in the original event, not what was left after redaction. The findings metadata (rule ID, label, severity, count) is preserved; the actual secret values are redacted by `sanitize_event()`.

Known-Token Replacement

Beyond regex patterns, the sanitizer supports redaction of specific known token values:

```
/// Sanitize content, also redacting any literal occurrences of known token values.
pub fn sanitize_with_known_tokens(input: &str, known_tokens: &[&str]) -> String {
    let mut result = sanitize(input);
    for token in known_tokens {
        if !token.is_empty() {
            result = result.replace(token, "[REDACTED]");
        }
    }
    result
}
```

This function first runs the standard regex-based `sanitize()`, then does literal string replacement for each known token. This is defense-in-depth for cases where a specific secret value is known to the system but doesn't match any standard pattern — for example, an internal canary secret, a custom API key format, or a token that was rotated and its old value should be scrubbed from historical data.

The known-token replacement is a simple string `replace`, not a regex. It catches exact literal occurrences regardless of context. The empty-token guard (`!token.is_empty()`) prevents the degenerate case of replacing every empty string with `[REDACTED]`.

The test suite verifies this behavior:

```
#[test]
fn known_token_replacement() {
    let s = sanitize_with_known_tokens(
        "internal name 'abc123secretvalue' appears",
        &["abc123secretvalue"],
    );
    assert!(!s.contains("abc123secretvalue"));
}
```

Findings as Metadata, Never Values

The most important security property of the sanitizer is that findings are metadata-only. The `detect()` function returns `Vec<SecretDetection>`:

```
#[derive(Debug, Clone, PartialEq, Eq, Serialize, Deserialize)]
pub struct SecretDetection {
    pub rule_id: String,
    pub label: String,
    pub severity: String,
    pub occurrence_count: usize,
    pub advice: String,
}
```

There is no field for the matched value. No `matched_text`, no `preview`, no `hash`, no `reversible_representation`. The struct carries only:

- `rule_id` — which pattern matched (e.g., `anthropic-api-key`)
- `label` — human-readable name (e.g., `Anthropic API key`)
- `severity` — always `"high"` in v1
- `occurrence_count` — how many times the pattern matched in the input
- `advice` — standardized remediation text

This is a deliberate design constraint. The `agentchron_secret_filter` metadata block that travels with sanitized events contains the same fields — rule ID, label, severity, count, advice — but never the matched value. Downstream consumers (the sink, the web UI, the foundry, rotation report generators) can know *what* was redacted and *how many times* without ever seeing *what it was*.

The `EventFinding` struct in the sink's plugin framework follows the same constraint:

```
pub struct EventFinding {
    pub plugin: String,
    pub rule_id: String,
    pub category: String,
    pub severity: String,
    pub summary: String,
    pub advice: String,
    pub occurrence_count: usize,
}
```

The `summary` field is a templated string like `"Anthropic API key detected during sink-side plugin scan"` — it includes the label but not the value. The finding is metadata for rotation reports and audit trails, not a secret recovery mechanism.

This property is what makes the entire pipeline safe to inspect. An operator can look at the `agentchron_secret_filter` block on an event in the web UI and see that an Anthropic API key was detected and redacted, without the web UI ever having had access to the key. The sink stores the finding metadata in the `plugin_finding` table, queryable via `GET /v1/plugins/findings`, and the finding rows contain only metadata — never values.

Sanitization Across the Pipeline

Secret sanitization happens at multiple layers in the Orca pipeline. The defense-in-depth chain has five independent sanitization passes:

LAYER 1: AGENT `PARSE_LINE()` (FIRST)

The agent's `parse_line()` function in `crates/agentchron-core/src/parser.rs` runs `sanitizer::detect()` on the raw line to produce `EventFinding` records, then sanitizes all string content recursively via `sanitize_envelope()`, `sanitize_message()`, and `sanitize_value()`. This is the first layer — it runs on the host where the session file originates, before any network transport.

The parser also preserves local sanitizer findings from the `agentchron_secret_filter` extra field — if the line was already sanitized by `sanitize_json_line()` (e.g., by the TCP push path), the existing findings are preserved and new findings from `detect()` are added.

LAYER 2: `AGENTCHRON-PUSH` `SANITIZE_JSON_LINE()` (TCP PATH)

The TCP push client in `crates/agentchron-agent/src/push_wire.rs` sanitizes each line with `sanitize_json_line()` before writing it to the TCP stream. This is the second layer for the TCP path — it runs on the host before the bytes leave over the network.

The `push_reader()` function reads lines from a reader (file or stdin), sanitizes each line, and writes it to the TCP stream:

```
// Simplified from push_wire.rs
for line in reader.lines() {
    let sanitized = sanitize_json_line(&line);
    stream.write_all(sanitized.as_bytes()).await?;
    stream.write_all(b"\n").await?;
}
```

This means the TCP push path has two local sanitization passes: the `parse_line()` in the agent (if the line goes through the parser) and the `sanitize_json_line()` in the push wire (which always runs). The push wire sanitization is a safety net — even if the input is raw JSONL that hasn't been through `parse_line()`, it gets sanitized before transport.

LAYER 3: SINK `INGEST_ONE()` BELT-AND-SUSPENDERS (THIRD)

The sink's `ingest_one()` function in `crates/agentchron-sink/src/ingest.rs` runs the `SecretsFilterPlugin` and then `sanitize_event()`. This is the third layer — it runs at the central sink, after transport, regardless of which path the event took.

The `sanitize_event()` function re-sanitizes all text, envelope extras, message content, and tool inputs:

```
fn sanitize_event(evt: &mut Event) {
    if let Some(t) = evt.text.take() {
        evt.text = Some(sanitizer::sanitize(&t));
    }
    evt.envelope.extra = std::mem::take(&mut evt.envelope.extra)
        .into_iter()
        .map(|(key, value)| (key, sanitizer::sanitize_json_value(value)))
        .collect();
    if let Some(message) = evt.envelope.message.as_mut() {
        message.content = sanitizer::sanitize_json_value(message.content.take());
        message.extra = std::mem::take(&mut message.extra)
            .into_iter()
            .map(|(key, value)| (key, sanitizer::sanitize_json_value(value)))
            .collect();
    }
    for tool in &mut evt.tool_uses {
        tool.input = sanitizer::sanitize_json_value(tool.input.take());
    }
}
```

Every string-bearing field is re-sanitized. The `take()` pattern moves the value out, sanitizes it, and puts it back — ensuring no original (pre-sanitization) text survives in the stored event.

LAYERS 4 AND 5: FOUNDRY SILVER CLEANING

The Data Foundry adds two more passes during the bronze → silver transformation:

1. **Realtime sanitizer pass** — Runs the standard `sanitize()` over all text fields during the silver cleaning pipeline.
2. **Offline deny-list + entropy check** — A more aggressive scan that uses entropy-based detection (high-entropy strings that don't match known patterns) and an offline deny-list of known secret values.

The foundry's cleaning order is: secret redaction → PII → license → dedup → boilerplate → outcome → decisions. Secret redaction is first because everything downstream depends on clean text — you can't deduplicate or classify text that contains embedded secrets.

COMPOSITION FOR FULL COVERAGE

The five layers compose as follows:

```
Host (Mac/Linux/VM):
  Layer 1: parse_line() → detect() + sanitize_envelope/sanitize_message/sanitize_value
  Layer 2: sanitize_json_line() (TCP path only)

Transport:
  HTTP POST /v1/events (Layers 1-2 already applied)
  TCP push (Layers 1-2 already applied)

Sink (.114):
  Layer 3: SecretsFilterPlugin.inspect() → detect() on text + tool inputs
  Layer 3: sanitize_event() → sanitize all string fields

Foundry (.114):
  Layer 4: Realtime sanitizer in silver cleaning
  Layer 5: Offline deny-list + entropy check
```

The key property is that **no single layer is trusted to have done it correctly**. The sink re-runs detection even after the agent has already sanitized. The foundry re-runs detection even after the sink has already sanitized. A direct client that bypasses the agent parser and sends raw events to the sink's HTTP API still gets sanitized — the sink's plugin and sanitizer run regardless of the source.

This is the Defense-in-Depth Sanitization pattern (P1) in full: five independent layers, each sufficient for its boundary, none trusting the others.

Patterns Developed in This Chapter

LOCAL-BEFORE-TRANSPORT (P2)

The foundational security posture: sanitize on the host before any network transport. The agent's `parse_line()` runs on the host where the session file lives. The TCP push client's `sanitize_json_line()` runs on the host before bytes leave over the network. The principle is: never trust the upstream to have done it correctly — but also, never send raw secrets over the network in the first place, even if the downstream will sanitize them. The network is a broader attack surface than the host.

DEFENSE-IN-DEPTH SANITIZATION (P1)

Five independent sanitization layers, each sufficient for its boundary. The agent sanitizes before transport. The TCP push sanitizes before streaming. The sink plugin detects before sanitizing. The sink sanitizer redacts on ingest. The foundry sanitizes during silver cleaning. No single layer is trusted. The sink's plugin runs *before* its sanitizer specifically so that direct clients who skip the agent parser cannot bypass secret reporting.

REPRODUCIBILITY ENVELOPE (P12)

The `FILTER_VERSION` constant (`agentchron-secrets-filter-v1`) travels in every findings metadata block. Downstream consumers can distinguish v1 findings from v2 findings. If a new pattern is added in v2, old events with v1 findings can be re-scanned to check whether the new pattern would have matched. The filter version is the reproducibility envelope for security findings — it tells you exactly which version of the detector produced a given set of redactions.

Anti-Patterns Addressed

DIRECT VAULT/CREDENTIAL SCRAPING (AP6)

Agents should not `cat`, `tail`, `grep`, `jq`, or bulk-read vault files, credential files, or security-event snapshots. Direct scraping bypasses attribution and risks dumping secrets into transcripts. The sanitizer catches secrets that appear in transcripts, but the better solution is to prevent them from appearing in the first place. The sanctioned path is through approved tools — OpenBrain MCP/read APIs, `securegit secret info`, metadata-only MCP tools — so access can be attributed and acknowledged. Orca Guard (Chapter 5) blocks direct secret-file reads at the `PreToolUse` hook before they execute; the sanitizer is the fallback for anything that slips through.

TREATING AUDIT MODE AS SAFE (AP4)

In audit mode, the sanitizer's `detect()` still runs and produces findings, but the findings are metadata-only reports (`would_block_in_production=true`) rather than blocking decisions. The sanitizer itself always redacts — `sanitize()` and `sanitize_json_line()` replace matches with `[REDACTED]` regardless of mode. But if Guard is in audit mode, a secret may reach the transcript before the sanitizer sees it (because the hook didn't block). The sanitizer will redact it during ingest, but the raw secret was already on disk in the original transcript file. Audit mode is for QA and tuning, not for secret-bearing work.

The `sanitize()` Function: Plain-Text Redaction

The simplest entry point is `sanitize()`, which applies all 19 regex patterns sequentially:

```
/// Sanitize content by redacting known credential patterns.
pub fn sanitize(input: &str) -> String {
    let mut result = input.to_string();
    for pattern in TOKEN_PATTERNS.iter() {
        result = pattern.regex.replace_all(&result, "[REDACTED]").to_string();
    }
    result
}
```

Each pattern replaces all its matches with the literal string `[REDACTED]`. The order of application doesn't matter for the final output because every pattern replaces with the same string — there's no conflict between overlapping patterns. The shadowing logic in `detect()` prevents double-counting in findings, but `sanitize()` doesn't need to shadow because the replacement is idempotent: replacing `[REDACTED]` with `[REDACTED]` is a no-op.

The test suite verifies that `sanitize()` preserves normal text:

```
#[test]
fn preserves_normal_text() {
    let input = "Pushed 3 commits to main. Smoke test passed.";
    assert_eq!(sanitize(input), input);
}
```

This is an important property: the sanitizer should not produce false positives on normal English text. The regex patterns are designed with minimum length requirements (20-40 characters after the prefix) and specific character classes to minimize false matches on prose.

Detailed Pattern Analysis

Let's examine several patterns in detail to understand their design trade-offs.

GITHUB PAT PATTERN

```
r"ghp_[A-Za-z0-9]{36,}"
```

GitHub classic PATs start with `ghp_` followed by at least 36 base62 characters. The minimum of 36 is chosen to match GitHub's actual token format while avoiding short false positives like `ghp_test`. The character class `[A-Za-z0-9]` matches base62 (no hyphens, underscores, or special characters), which is GitHub's encoding.

URL-EMBEDDED CREDENTIAL PATTERN

```
r"://[^@\s/]+:[^@\s/]+@"
```

This pattern matches the `://user:pass@` syntax in URLs. It's deliberately broad: `[^@\s/]+` matches any non-@, non-whitespace, non-slash characters for both the username and password. This catches:

- `https://oauth2:ghp_token@gitlab.example.com/repo.git`
- `postgres://user:<password>@<db-host>:5432/mydb`
- `redis://default:redis-password@<cache-host>:6379`

The pattern doesn't capture the host — it only matches the `://user:pass@` portion. This means the redacted output is `://[REDACTED]@host/path`, which preserves the URL structure for debugging while removing the credentials.

PEM PRIVATE KEY BLOCK PATTERN

```
r"(?s)-----BEGIN [A-Z0-9 ]*PRIVATE KEY-----.*?-----END [A-Z0-9 ]*PRIVATE KEY-----"
```

The `(?s)` flag enables single-line mode, allowing `.` to match newlines. This is essential because PEM blocks span multiple lines. The `[A-Z0-9 ]*` portion matches the key type (e.g., `RSA`, `EC`, `OPENSSH`, `PGP`), and the `.*?` is a non-greedy match that captures the base64-encoded key material between the BEGIN and END markers.

The test verifies that a full PEM block is replaced with a single `[REDACTED]`:

```
#[test]
fn redacts_pem_private_key_block() {
    let s = sanitize("-----BEGIN RSA PRIVATE KEY-----\nMIIE...\n-----END RSA PRIVATE
KEY-----");
    assert_eq!(s, "[REDACTED]");
}
```

BASE64-ENCODED PEM BLOCK PATTERN

```
r"\bLS0tLS1CRUdJTi(?:[A-Za-z0-9+/=]{32,})"
```

`LS0tLS1CRUdJTi` is the base64 encoding of `-----BEGIN`. This pattern catches PEM keys that have been base64-encoded for embedding in JSON, environment variables, or single-line config files. The `(?:[A-Za-z0-9+/=]{32,})` matches the remaining base64 characters. This is a defense against the common practice of base64-encoding secrets to avoid newline issues in config files.

JWT PATTERN

```
r"eyJ[A-Za-z0-9_-]{10,}\.eyJ[A-Za-z0-9_-]{10,}\.[A-Za-z0-9_\-]{10,}"
```

JWTs have a three-part structure: `header.payload.signature`. Each part is base64url-encoded. The header and payload both start with `eyJ` because they are base64url-encoded JSON objects starting with `{"`. The pattern requires at least 10 characters in each part to avoid false positives on short strings that happen to start with `eyJ`.

The signature part uses `[A-Za-z0-9_\-]` (base64url) rather than `[A-Za-z0-9+/=]` (standard base64) because JWTs use base64url encoding without padding.

The `agentchron_secret_filter` Metadata Block in Practice

When an event is sanitized, the `agentchron_secret_filter` metadata block travels with it through the entire pipeline. Here's how downstream consumers use it:

SINK STORAGE

The sink stores the `agentchron_secret_filter` block in the event's `envelope.extra` field (via `#[serde(flatten)]`). When the event is queried via `GET /v1/events/:id`, the metadata block is included in the response. The sink's `SecretsFilterPlugin` also runs `detect()` independently, producing `EventFinding` records that are stored in the `plugin_finding` table.

WEB UI

The web UI can display the `agentchron_secret_filter` block on the event detail page, showing the operator which patterns matched, how many times, and what the remediation advice is. This is particularly useful for rotation reports — an operator can see that an Anthropic API key was detected in a session and rotate it.

PLUGIN FINDINGS API

The sink's `GET /v1/plugins/findings` endpoint returns `EventFinding` records with filters (plugin, category, severity, host, agent, source, session_id). An operator can query:

```
GET /v1/plugins/findings?plugin=secrets-filter&category=secret&severity=high
```

This returns all high-severity secret findings across all sessions, with the rule ID, label, occurrence count, and advice for each. The findings are metadata-only — no matched values. This is the rotation report surface: an operator can identify every event where a secret was detected and take remediation action.

FOUNDRY SILVER CLEANING

The Data Foundry's silver cleaning pipeline reads the `agentchron_secret_filter` block to know which events have been flagged. The realtime sanitizer pass re-runs `sanitize()` over all text fields; the offline deny-list + entropy check adds additional detection for secrets that don't match standard patterns. The foundry's cleaning order (secret redaction → PII → license → dedup → boilerplate → outcome → decisions) ensures that secret redaction is the first transformation, so all downstream processing works on clean text.

Entropy-Based Detection: The Foundry's Additional Layer

The sanitizer's 19 regex patterns are prefix-based — they match known credential formats by their distinctive prefixes (`ghp_` , `sk-ant-` , `AKIA` , etc.). This is fast and has low false-positive rates, but it misses secrets in unknown formats: a custom API key with a non-standard prefix, a service token with no prefix at all, or a high-entropy password embedded in a config file.

The foundry's offline deny-list + entropy check addresses this gap. During silver cleaning, the foundry runs an entropy-based detector that flags strings with high Shannon entropy — strings that look like random characters, which is a strong signal of a credential or encrypted value. This catches secrets that the regex-based sanitizer misses.

The trade-off is false positives: high-entropy strings can also be hashes, UUIDs, base64-encoded images, or compressed data. The foundry handles this with a deny-list of known safe high-entropy strings (e.g., commit SHAs, known UUIDs) and human review for ambiguous cases.

The regex sanitizer and the entropy detector are complementary: the regex sanitizer catches known formats with low false-positive rates; the entropy detector catches unknown formats with higher false-positive rates. Together, they provide broad coverage.

Testing the Sanitizer

The sanitizer module has extensive test coverage. Each pattern has at least one dedicated test:

TEST	PATTERN VERIFIED
`redacts_github_classic_pat`	`ghp_` pattern
`redacts_gitlab_pat`	`glpat-` pattern
`redacts_anthropic_key`	`sk-ant-` pattern
`anthropic_key_is_not_double_classified_as_openai`	Shadowing logic
`redacts_openai_project_key`	`sk-proj-` pattern
`redacts_elevenlabs_key`	`sk_` pattern
`redacts_hugging_face_token`	`hf_` pattern
`redacts_url_embedded_credentials`	`://user:pass@` pattern
`redacts_bearer_header`	`Bearer` pattern
`preserves_normal_text`	No false positives on prose
`known_token_replacement`	Known-token redaction
`redacts_aws_access_key_id`	`AKIA` pattern
`redacts_jwt`	JWT pattern
`redacts_stripe_secret_key`	`sk_live_` / `sk_test_` pattern
`redacts_stripe_webhook_secret`	`whsec_` pattern
`redacts_pem_private_key_block`	PEM block pattern
`redacts_base64_encoded_pem_block`	Base64 PEM pattern
`json_line_sanitizer_stamps_metadata_without_secret_value`	JSON metadata stamping

The tests use synthetic canary values, not real secrets. The `anthropic_key_is_not_double_classified_as_openai` test is particularly important — it verifies the shadowing logic that prevents the OpenAI pattern from double-classifying Anthropic keys. Without this test, a regression in the shadowing logic would produce duplicate findings for every Anthropic key, inflating the occurrence count and potentially confusing rotation reports.

The `json_line_sanitizer_stamps_metadata_without_secret_value` test is the end-to-end contract test: it verifies that a JSON line with an embedded secret is sanitized (the secret is removed), the JSON structure is preserved (it's still valid JSON), and the metadata block is stamped with the correct rule ID and filter version. This is the test that proves the findings-as-metadata-never-values contract.

Pattern Design Philosophy

The 19 patterns in `TOKEN_PATTERNS` are not an exhaustive list of every possible secret format. They are a curated set targeting the credential types that commonly appear in AI coding assistant transcripts. The design philosophy is:

1. **High precision over high recall.** A false positive (blocking a safe prompt because it matched a secret pattern) is costly — it disrupts the operator's workflow. A false negative (missing a secret) is also costly, but the defense-in-depth chain (five sanitization layers) means a missed secret at the regex layer may still be caught by the entropy detector in the foundry. The patterns are tuned to minimize false positives.

2. **Prefix-based matching.** Most patterns match a distinctive prefix (`ghp_` , `sk-ant-` , `AKIA` , `glpat-`) followed by a minimum-length character class. Prefix-based matching is fast (the regex engine can fail early on non-matching prefixes) and precise (the prefix is a strong signal that the string is a credential, not a random match).
3. **Minimum length requirements.** Each pattern requires a minimum number of characters after the prefix — typically 20-40. This prevents false positives on short strings like `ghp_test` or `sk-ant-example` while matching real credentials, which are always long enough to be cryptographically meaningful.
4. **Standardized advice.** Every pattern carries the same advice text: "Assume this credential may have been exposed in a Claude Code session. Review the source context, rotate the credential if it is live, and remove it from shell history, config files, and commit history as needed." This standardization ensures that every finding produces actionable remediation guidance without per-pattern customization.
5. **Severity is always "high".** In v1, all secret detections are high-severity. There is no "medium" or "low" secret — if a credential pattern matches, it's a high-severity event that requires attention. Future versions may introduce graduated severity (e.g., a test key prefix might be "medium"), but v1 treats all secrets equally.

Extending the Pattern Set

Adding a new pattern to `TOKEN_PATTERNS` requires:

1. **Choose a** `rule_id` — a kebab-case identifier like `google-api-key` or `slack-bot-token` .
2. **Choose a** `label` — a human-readable name like "Google API key" or "Slack bot token".
3. **Write the regex** — targeting the credential's distinctive prefix and character format.
4. **Add a test** — verify that a synthetic canary value is detected and that normal text is not false-positive matched.
5. **Consider shadowing** — if the new pattern overlaps with an existing pattern (like the OpenAI/Anthropic `sk-` overlap), add a shadowing rule in `is_shadowed_detection()` .
6. **Bump the filter version** — when a new pattern is added, the filter version should change from `agentchron-secrets-filter-v1` to `agentchron-secrets-filter-v2` so downstream consumers can distinguish old findings from new findings.

The filter version bump is the Reproducibility Envelope in action. Old events carry `v1` findings; new events carry `v2` findings. If the new pattern would have matched old data, a re-scan can identify the additional detections. The version string is the provenance marker that enables retrospective analysis.

The `sanitize_json_value` Recursion

The `sanitize_json_value` function is the workhorse of JSON-aware sanitization. It walks the entire JSON tree and sanitizes every string value:

```
pub fn sanitize_json_value(value: Value) -> Value {
    match value {
        Value::String(s) => Value::String(sanitize(&s)),
        Value::Array(items) =>
Value::Array(items.into_iter().map(sanitize_json_value).collect()),
        Value::Object(object) => Value::Object(
            object
                .into_iter()
                .map(|(key, value)| (key, sanitize_json_value(value)))
                .collect(),
        ),
        other => other,
    }
}
```

The function is recursive — arrays and objects are traversed to arbitrary depth. This is essential because secrets can be deeply nested in tool inputs. Consider a Bash tool call that runs a command containing a URL with embedded credentials:

```
{
  "tool_use": {
    "name": "Bash",
    "input": {
      "command": "git clone https://oauth2:ghp_token@gitlab.example.com/repo.git"
    }
  }
}
```

The `sanitize_json_value` function will: 1. Match `Value::Object` → recurse into each field 2. Match `tool_use` as `Value::Object` → recurse 3. Match `input` as `Value::Object` → recurse 4. Match `command` as `Value::String` → run `sanitize()`, which matches the URL-embedded-credential pattern 5. Return the sanitized string with `[REDACTED]` replacing the credential portion

The recursion depth is bounded by the JSON document's nesting depth, which is typically shallow (2-5 levels for Claude Code events). There is no explicit depth limit, but the JSON parser itself has a recursion limit that prevents stack overflow on pathological inputs.

Object keys are not sanitized — only values. This is deliberate: object keys are typically structural identifiers (`tool_use`, `input`, `command`), not secret-bearing strings. Sanitizing keys would break the JSON structure and make the event unparseable downstream.

The `detect()` Function: Counting Without Revealing

The `detect()` function is the detection-only entry point. Unlike `sanitize()`, it does not modify the input — it returns a vector of `SecretDetection` findings:

```
pub fn detect(input: &str) -> Vec<SecretDetection> {
    let mut counts: BTreeMap<&'static str, (&'static TokenPattern, usize)> =
BTreeMap::new();
    for pattern in TOKEN_PATTERNS.iter() {
        let count = pattern
            .regex
            .find_iter(input)
            .filter(|m| !is_shadowed_detection(pattern.rule_id, m.as_str()))
            .count();
        if count > 0 {
            counts
                .entry(pattern.rule_id)
                .and_modify(|(_, existing)| *existing += count)
                .or_insert((pattern, count));
        }
    }

    counts
        .into_values()
        .map(|(pattern, occurrence_count)| SecretDetection {
            rule_id: pattern.rule_id.to_string(),
            label: pattern.label.to_string(),
            severity: pattern.severity.to_string(),
            occurrence_count,
            advice: pattern.advice.to_string(),
        })
        .collect()
}
```

The `BTreeMap` keyed by `rule_id` deduplicates findings — if the same pattern matches multiple times in the input, the counts are accumulated into a single `SecretDetection` with the total `occurrence_count`. This is the right behavior for rotation reports: "3 Anthropic API keys detected in this event" is more useful than three separate findings.

The `find_iter` method returns match objects, but the `filter` call applies the shadowing check to each match. The shadowing check receives the `rule_id` and the matched text (`m.as_str()`), but the matched text is only used for the shadowing decision — it is never stored in the `SecretDetection`. This is the findings-as-metadata-never-values contract enforced at the code level: the matched text is available during detection but is not propagated to the finding.

Conclusion

The secret sanitization engine is the security core of the Orca platform. Its 19 detection patterns — ported from SecureGit and extended with AI-provider tokens — cover the credential formats that commonly appear in AI coding assistant transcripts. The shadowing logic prevents double-classification of overlapping patterns. The JSON-aware sanitization preserves document structure while redacting secrets and stamping metadata that travels downstream as evidence.

The sink-side plugin framework ensures that direct clients cannot bypass secret reporting by skipping the agent parser. The plugin runs before the sink's sanitizer pass, producing findings from the raw event text, then the

sanitizer redacts the actual values. The five-layer defense-in-depth chain — agent parser, TCP push, sink plugin, sink sanitizer, foundry silver cleaning — provides redundant coverage where no single layer is trusted.

The findings-as-metadata-never-values principle is the most important security property. The entire pipeline — from detection on the host to storage in the sink to display in the web UI — carries evidence of what was redacted without ever carrying the redacted value itself. Rule IDs, labels, severity, occurrence counts, and remediation advice travel in the `agentchron_secret_filter` metadata block. Rotation reports can be generated from this metadata. Audit trails can show what was caught. But the secret values are gone, replaced by `[REDACTED]`, at every layer of the pipeline.

The filter version `agentchron-secrets-filter-v1` is the reproducibility envelope. When v2 adds new patterns, the version string in old findings will distinguish them from new findings, enabling re-scanning and rotation tracking across filter versions. This is the same principle that governs the foundry's reproducibility envelopes on dataset artifacts — every generated artifact carries its generation provenance, and security findings are no exception.

Chapter 7: MCP Gateway

Role and Positioning

The Orca MCP Governance Gateway (`orca-mcp-gateway`) is the local policy enforcement point for enterprise AI coding assistants. It is a Rust stdio MCP proxy that sits between the AI client (Claude Code, Codex, Gemini/AGY, ArmyknifeClaw) and upstream MCP servers. Instead of the client talking directly to an MCP server, the client talks to the gateway, and the gateway forwards, filters, or blocks based on policy.

The Model Context Protocol (MCP) is the standard interface through which AI assistants access external tools and data sources. An MCP server exposes a catalog of tools via `tools/list` and accepts tool invocations via `tools/call`. Without governance, any tool the MCP server exposes is available to any agent that can reach it. The gateway intercepts both methods:

- `tools/list` — Catalog filtering: remove tools the agent's role is not permitted to use
- `tools/call` — RBAC enforcement: allow, deny, or gray-area the specific tool invocation based on the agent's role, the tool, the MCP server, the action, and the arguments

The gateway is a stdio proxy, not an HTTP proxy. MCP communication happens over stdin/stdout with Content-Length-framed JSON-RPC messages. The gateway launches the upstream MCP server as a child process, proxies the JSON-RPC frames, and intercepts the two methods that matter for policy enforcement:

```
AI Client ←stdin/stdout→ orca-mcp-gateway ←stdin/stdout→ upstream MCP server
                                  ↓
                      Policy enforcement + signed receipts
```

The gateway binary lives in `agentchron-agent` and is launched with flags identifying the upstream server, the client type, the tenant, the human operator's DID, the agent instance ID, and the role:

```
orca-mcp-gateway \
  --upstream-command github-mcp-server \
  --upstream-server-id github \
  --client-type claude \
  --tenant-id acme \
  --employee-did did:hermes:acme:emp:1234 \
  --instance-id did:hermes:acme:emp:1234:claude:laptop \
  --signing-key-id acme-pep-claude-laptop \
  --role frontend
```

This command launches `github-mcp-server` as a child process, proxies its stdio, and enforces the `frontend` role's discipline-scoped RBAC on every `tools/list` and `tools/call` request.

Binding Contracts

The gateway adheres to several formal, frozen contracts that ensure cross-language and cross-implementation conformance.

RECEIPT SHAPE: `GOVERNANCE.RECEIPT.V1`

Every allow and deny decision produces a signed governance receipt. The receipt body is a frozen 19-field contract from Link's 2026-06-13 contract. The body is frozen — no fields can be added, removed, or renamed without a new schema version. This is the Frozen Schema with Additive Sidecars pattern (P7): new evidence types link to the receipt by hash rather than inflating the schema.

The 19 fields carry:

- **Identity:** tenant ID, employee DID, agent instance ID, signing key ID
- **Request:** MCP server ID, tool name, action (list/call), arguments
- **Decision:** verdict (allow/deny), rule ID that fired, rule context
- **Digests:** `args_digest` and `result_digest`
- **Provenance:** `identity_anchor` (chain link to previous receipt)
- **Timestamp:** decision time
- **Cost:** post-exec cost when available

DIGEST RULE: RFC 8785 JCS CANONICALIZATION

The `args_digest` and `result_digest` fields are `sha256(JCS(value))` rendered as bare lowercase hex with no `sha256:` prefix. JCS (JSON Canonicalization Scheme) is defined in RFC 8785 and implemented by the `serde_json_canonicalizer` crate.

This is not a local serializer. The gateway does not use `serde_json::to_string` for digest computation. It uses the RFC 8785 canonical form, which defines a deterministic ordering of object keys, a deterministic representation of numbers, and a deterministic encoding of strings. This ensures that the same JSON value produces the same digest across implementations — Rust, Python, Go, Java, TypeScript — without any implementation-specific quirks.

This is the RFC 8785 JCS Canonicalization with Cross-Language Conformance pattern (P9). Cross-language conformance is proven by fixture tests (`governance-receipt-v1.fixture.json`) that assert byte-identical canonicalization and signatures across implementations. The fixture uses a public test vector derived from a known phrase, so any implementation can verify its conformance independently.

POLICY MODEL: DISCIPLINE-SCOPED RBAC

The policy model is discipline-scoped RBAC with four enforcement axes:

AXIS	DESCRIPTION	EXAMPLE
`domain`	File path domain the role can access	`components/**` for frontend
`tool`	Tool names the role can invoke	`Edit`, `Read`, `Bash`
`mcp_server`	MCP servers the role can reach	`github`, `filesystem`
`action`	Actions the role can perform	`read`, `write`, `execute`

A role constrains all four axes. For example, the `frontend` role can use the `Edit` tool on paths matching `components/**` but cannot use `Edit` on paths matching `terraform/**`. It can reach the `github` MCP server but not the `aws` MCP server.

TOOL NAME GRAMMAR

MCP tool names follow different conventions depending on the client. The gateway supports three grammars:

GRAMMAR	EXAMPLE	CLIENT
`mcp__server__tool`	`mcp__github__create_issue`	Anthropic (Claude Code)
`server/tool`	`github/create_issue`	Legacy
`server:tool`	`github:create_issue`	Legacy

The gateway normalizes all three to a canonical `(server, tool)` pair for policy evaluation, then re-emits the tool name in the original grammar for the upstream server. This lets Orca policy layer on top of native client MCP allow/deny settings without requiring a specific naming convention.

Twelve Built-In Roles

The gateway ships 12 discipline-scoped roles, each constraining file paths, tools, and commands:

ROLE	CAN DO	CANNOT DO
`frontend`	Edit `components/**`, run UI build commands	Touch `infra/**`, `terraform/**`, `k8s/**`, `auth/**`, `billing/**`; run `kubectl`, `aws`, `gcloud`, `az`, `systemctl`, `terraform`
`backend`	Edit service code, run service tests, manage DB migrations	Touch `infra/**`, `terraform/**`, `k8s/**`
`sysadmin`	Run system commands, manage services, edit config files	Touch application source code
`cloud-engineer`	Manage cloud resources, `terraform`, `kubectl`, `aws`, `gcloud`, `az`	Edit application source code
`devops`	Manage CI/CD, deploy commands, infrastructure automation	Touch `auth/**`, `billing/**`
`security-engineer`	Run security scans, audit logs, inspect secrets metadata	Read raw secret values, modify application code
`data-engineer`	Manage data pipelines, DB queries, ETL jobs	Touch `infra/**`, `auth/**`
`viewer`	Read files, search, list tools	Edit, write, execute, or call any mutating tool
`lead`	All of frontend + backend, plus approve/review actions	Touch `billing/**`
`admin`	All tools, all paths, all servers	(constrained only by org floor denies)
`auditor`	Read all events, receipts, compliance reports, findings	Modify any data or execute any tool
`pentest-scoped`	Run security tools against scoped targets only	Access production data or out-of-scope systems

The `frontend` role is the canonical example. A frontend developer's agent can edit React components, run `npm test`, and use the GitHub MCP server to create PRs. It cannot run `terraform apply`, `kubectl delete`, or access the AWS MCP server. This is discipline scoping — the role reflects the engineer's professional discipline and constrains the agent to that discipline's domain.

The decision conformance vectors from the gateway tests verify these constraints:

NAME	ROLES	NARROWING	FLOOR	REQUEST	EXPECTED
frontend infra edit denied	`frontend`	none	none	`Edit` path `terraform/prod/main.tf`	`deny`, `frontend-no-infra-paths`
frontend ui edit allowed	`frontend`	none	none	`Edit` path `components/Button.tsx`	`allow`, `frontend-edit-ui`
narrowing cannot widen	`frontend`	`cloud-engineer`	none	`Bash` command `terraform apply`	`deny`, `frontend-no-cloud-shell`
floor overrides admin	`admin`	none	`mcp__aws__*`	`mcp__aws__listBuckets`	`deny`, `floor-deny:mcp-aws`

Narrowing-Only Role Intersection

The identity model is built on a critical security property: **agent authority is always bounded by human authority**.

The human DID (Decentralized Identifier) supplies base roles. When an operator with `did:hermes:acme:emp:1234` starts a Claude Code session, their DID resolves to a set of base roles — say, `frontend` and `viewer`. The agent instance may be assigned narrowing roles — say, `frontend` only. The effective permission is the **intersection**: base ∩ narrowing.

```
Human DID roles:     {frontend, viewer}
Agent instance roles: {frontend}
Effective permission: {frontend}  (base ∩ narrowing)
```

The key rule: **agent instance roles may only narrow, never widen**. If the human has `frontend` and the agent instance claims `admin`, the effective permission is `frontend` — not `admin`. The agent cannot escalate beyond the human's authority.

ORG FLOOR DENIES

Above the role system, org floor denies apply. A floor deny is an organization-level constraint that cannot be weakened by any role, instance narrowing profile, or local client config. For example:

```
orca-mcp-gateway \
  --upstream-command github-mcp-server \
  --upstream-server-id github \
  --role admin \
  --floor-deny 'mcp__aws__*;terraform/**'
```

Even with the `admin` role, the floor deny blocks all `mcp__aws__*` tool calls and all `terraform/**` path edits. The floor deny is evaluated above roles — it is an absolute deny that no role can override.

The decision conformance vector "floor overrides admin" verifies this: an `admin` role with a `mcp__aws__*` floor deny cannot call `mcp__aws__listBuckets`. The deny reason is `floor-deny:mcp-aws`, not a role-based rule.

EFFECTIVE PERMISSION COMPUTATION

```
Org floor denies (absolute, cannot be weakened)
  ∩ Human DID base roles (authority boundary)
    ∩ Agent instance narrowing roles (may only narrow)
      = Effective permission
```

This three-layer intersection ensures that: 1. Org policy is the outer boundary (floor denies) 2. Human authority is the authority boundary (base roles) 3. Agent instance configuration can only restrict, never expand

Signed Receipts and Chain of Custody

Every allow and deny decision produces a signed governance receipt. The receipt system uses Ed25519 signing over RFC 8785 JCS canonical bytes of the 19-field receipt body.

ED25519 SIGNING

Ed25519 is a fast, deterministic signature scheme based on elliptic curves. The gateway accepts a managed 32-byte signing seed via `ORCA_MCP_SIGNING_SEED_HEX` or `--signing-seed-hex`, or derives a deterministic local PEP (Policy Enforcement Point) seed from tenant/persona/instance for dev smoke testing.

The signing process: 1. Construct the 19-field receipt body 2. Canonicalize to RFC 8785 JCS bytes 3. Sign with Ed25519 4. Emit the receipt with `signed_body`, `attestation` (alg, key_id, sig), and `canonical_sha256`

IDENTITY ANCHOR CHAINING

Receipts are chained using `identity_anchor`, seeded from the last non-empty JSONL record on gateway restart. This creates a tamper-evident local chain that survives restarts without requiring a centralized sequencing service.

```
Receipt N-1 (last before restart)
  → identity_anchor = hash(Receipt N-1)
Receipt N (first after restart)
  → identity_anchor = hash(Receipt N-1)  [seeded from file]
Receipt N+1
  → identity_anchor = hash(Receipt N)
```

Each receipt references the previous receipt's hash. Tampering with any receipt breaks the chain — the subsequent receipt's `identity_anchor` will not match the tampered receipt's hash.

RECEIPT WRITES

Receipt writes are queued to a single serialized writer thread. This is the Synchronous Enforcement, Asynchronous Telemetry pattern (P11):

- **Allowed calls** are forwarded to the upstream without waiting on disk flush. The receipt is queued and written asynchronously. The agent's tool call is not delayed by I/O.
- **Denied calls** are still receipted. The deny decision produces a receipt even though no upstream call was made. This ensures the audit trail is complete — every policy decision is recorded, whether it resulted in action or not.
- **Post-exec receipts** for allowed `tools/call` responses carry `result_digest` and cost when the upstream response exposes `result.cost.amount_cents` / `currency`. This creates a post-execution evidence record that links the tool call to its result and cost.

The receipt log is a JSONL file at `~/.orca/mcp-governance-receipts.jsonl`. Each line is a receipt envelope with `signed_body`, `attestation`, `canonical_sha256`, plus local JSONL siblings (`schema_version`, `public_key.spki_pem`, `rule_context`) that support verification without a separate key store.

Policy Bundles and Compliance Reports

SIGNED POLICY BUNDLES

Signed policy bundles (`orca.policy.bundle.v1`) can be loaded from disk and must verify against an out-of-band pinned org Ed25519 SPKI public key:

```
orca-mcp-gateway \
  --upstream-command github-mcp-server \
  --upstream-server-id github \
  --policy-bundle /etc/orca/policy-bundle.json \
  --policy-org-key-spki-pem-file /etc/orca/org-policy-key.pem \
  --role frontend
```

A verified bundle supersedes built-in role rules. Bundle floor denies combine with local floor denies; neither can be weakened. This allows an organization to distribute signed policy bundles that override the gateway's default role definitions — for example, adding organization-specific path constraints or tool restrictions.

The verification against a pinned SPKI (Subject Public Key Info) public key ensures that only the organization's policy signing key can produce valid bundles. A bundle signed by any other key is rejected. The SPKI format is a standard X.509 public key representation that can be generated from any Ed25519 key pair.

COMPLIANCE REPORTS

The gateway generates JSON compliance reports from the receipt chain with mappings for three regulatory frameworks:

FRAMEWORK	ARTICLE/SECTION	EVIDENCE MAPPED
EU AI Act	Article 12 (Record-keeping)	Receipt chain, signed attestations, tool call records
ISO 42001	Annex A.6.2.8 (AI system logging)	Policy decisions, enforcement mode, rule context
NIST AI RMF	(various)	Risk management evidence, governance decisions

```
orca-mcp-gateway \
  --receipt-log ~/.orca/mcp-governance-receipts.jsonl \
  --compliance-report-out evidence/orca-mcp-compliance-report.json
```

Compliance report framework mappings are stamped `READY` only when every receipt in the report has a valid Ed25519 attestation that verifies against the envelope `public_key.spki_pem`. Legacy or fixture logs with `attestation: null` remain `PENDING_ATTESTATION`. This distinction is important — a compliance report is not "READY" just because receipts exist; it is ready only when every receipt is cryptographically verified.

This is a high bar. It means the compliance report is not just a list of decisions — it is a cryptographically verified audit trail. An auditor can independently verify each receipt's signature against the public key, recompute the JCS canonical bytes, and confirm the digest. The `READY` stamp means this verification has already been performed and every receipt passed.

Guard Integration and Policy Modes

The MCP gateway integrates Orca Guard argument scanning. When a `tools/call` request passes through the gateway, Guard scans the tool arguments for secrets before the RBAC enforcement decision. This adds secret detection to the MCP tool call path, complementing the `PreToolUse` hook for Claude Code's native tools.

Guard operates in the same four modes as the standalone Guard binary:

MODE	BEHAVIOR IN GATEWAY
`max`	Deny-all for unmatched tool calls with `--policy-mode enforce`
`production`	Hard-block secrets, gray-area asks for approval
`dev`	Blocks secrets and secret-file reads, allows env-var names and path docs
`audit`	Report-only, `would_block_in_production=true`

POLICY MODES

The gateway supports two policy modes:

- `enforce` — Policy is enforced normally. Denied calls return a JSON-RPC error to the client. Allowed calls are forwarded to the upstream.
- `dry-run` — Policy is evaluated normally but allowed calls are left allowed and denied calls are also left allowed. The receipt emits `policy_decision: not_evaluated` with `rule_context.dry_run = true`. This is for observe-mode rollout — you can see what the policy would have decided without actually blocking anything.

```
# Observe-mode rollout: see what would be blocked without blocking
orca-mcp-gateway \
  --upstream-command github-mcp-server \
  --upstream-server-id github \
  --role frontend \
  --policy-mode dry-run
```

DEFAULT UNKNOWN POSTURE

The `--default-unknown-posture` flag controls what happens when a tool call doesn't match any role rule. If omitted, the behavior depends on the guard mode:

- `--guard-mode max` + `--policy-mode enforce` → deny-all for unmatched tool calls
- Other modes → allow-outside-governed behavior (unmatched calls pass through)

This allows organizations to choose their unknown-posture: fail-closed (deny unmatched) or fail-open (allow unmatched). The fail-closed posture is appropriate for high-assurance environments; the fail-open posture is appropriate for observe-mode rollout where you want to see what the policy catches without blocking unrecognized tools.

MCP FRAME SIZE CAP

The gateway enforces a maximum MCP frame size via `--max-frame-bytes` or `ORCA_MCP_MAX_FRAME_BYTES` before allocation. This prevents memory exhaustion from oversized JSON-RPC frames — a denial-of-service vector if an upstream server or client sends a multi-gigabyte frame.

AgentChron MCP Server

Separately from the governance gateway, `deploy/bin/agentchron-mcp.py` is a dependency-free stdio MCP server that lets agents query the Orca sink through the existing HTTP API. It does not open the SQLite database directly — all queries go through the sink's REST endpoints.

13 TOOLS

The server exposes 13 tools:

TOOL	PURPOSE	SINK ENDPOINT
`agentchron_search`	Paginated FTS search over events	`GET /v1/search`
`agentchron_event`	Fetch one event by ID with full detail	`GET /v1/events/:id`
`agentchron_graph_context`	GraphRAG-style related context	`GET /v1/graph/context`
`agentchron_recent_sessions`	Recent sessions list	`GET /v1/sessions`
`agentchron_session_summary`	One-call TL;DR for a session	`GET /v1/sessions/:id/summary`
`agentchron_session_events`	Bounded event page for one session	`GET /v1/sessions/:id/events`
`agentchron_source_coverage`	Source file coverage check	`GET /v1/sources/coverage`
`agentchron_security_findings`	Plugin findings needing review	`GET /v1/plugins/findings`
`orca_workflow_create`	Persist a panel review run as candidate	`POST /v1/workflows/from-panel`
`orca_workflow_search`	Search workflow library	`GET /v1/workflows`
`orca_workflow_list_approved`	List approved workflow rules	`GET /v1/workflows?status=approved`
`orca_workflow_approve`	Approve a candidate workflow	`POST /v1/workflows/:id/status`
`orca_workflow_retire`	Retire a workflow rule	`POST /v1/workflows/:id/status`

The server also exposes `agentchron_health` for sink health checks.

RECOMMENDED DEPLOYMENT: SSH-BASED TOKEN ISOLATION

The recommended deployment keeps the AgentChron token on `.114` by running the MCP server through SSH:

```
{
  "mcpServers": {
    "agentchron": {
      "command": "ssh",
      "args": [
        "developer@<lab-host>",
        "/home/developer/Projects/agentchron/deploy/bin/agentchron-mcp.py"
      ]
    }
  }
}
```

In this configuration, the MCP server runs on `.114` (the trusted host where the sink lives), and the agent client connects to it via SSH. The `AGENTCHRON_INGEST_TOKEN` is read from `/home/developer/Projects/agentchron/deploy/.env` on `.114` and never leaves that host. The agent client on the developer's laptop never sees the token.

This is the Local-First, Data-Residency-by-Default theme in practice. The token stays on the trusted host. The agent queries the sink through the MCP server, which proxies to `localhost:9474` (or the configured sink URL). No token crosses the network to the developer's laptop.

When the agent runs directly on `.114`, the config is simpler:

```
{
  "mcpServers": {
    "agentchron": {
      "command": "/home/developer/Projects/agentchron/deploy/bin/agentchron-mcp.py"
    }
  }
}
```

DIRECT HTTP MODE

For non-`.114` deployments, the server can be configured with a sink URL and token via environment:

```
AGENTCHRON_SINK_URL=http://<lab-host>:9475 \
AGENTCHRON_SINK_TOKEN=*** should be used when possible so secrets stay on `.114`.

## Patterns Developed in This Chapter

### RFC 8785 JCS Canonicalization with Cross-Language Conformance (P9)

All cryptographic signing in the gateway uses RFC 8785 JSON Canonicalization Scheme via
`serde_json_canonicalizer`, not a local serializer. The `args_digest` and `result_digest`
are `sha256(JCS(value))` as bare lowercase hex. Ed25519 signatures are computed over JCS
canonical bytes of the receipt body.

Cross-language conformance is proven by fixture tests (`governance-receipt-
v1.fixture.json`) that assert byte-identical canonicalization and signatures across
implementations. The fixture uses a public test vector derived from a known phrase, so any
implementation — Rust, Python, Go, Java, TypeScript — can verify its conformance
independently.

This is the cryptographic backbone for the MCP gateway, presence attestation, and gold
promotion receipts. It ensures that a receipt signed by a Rust gateway can be verified by a
Python compliance tool or a Go auditor without any implementation-specific ambiguity.

### Frozen Schema with Additive Sidecars (P7)

The `governance.receipt.v1` body is a frozen 19-field contract. No fields can be added,
removed, or renamed without a new schema version. New evidence types — presence
attestation, fleet node aggregation, gold DSSE receipts — are sidecars that link to the
receipt by `canonical_body_sha256`, not new fields in the frozen body.

This maintains backward compatibility: v1 fixtures remain byte-identical across
implementations. A v1 receipt produced today will have the same canonical bytes as a v1
receipt produced a year from now. Sidecars can be added, evolved, or deprecated without
touching the frozen receipt body.

The same pattern governs the Brain Bundle schema (`brain-bundle/v0` with `const`
enforcement of security invariants) and the FleetNode.v1 shape (unsigned denormalized
output that consumers must re-verify against the backing attestation).

### Synchronous Enforcement, Asynchronous Telemetry (P11)

Synchronous enforcement (RBAC allow/deny decisions, Guard argument scanning) must not wait
on mesh, Looking Glass, or ContextOS sinks. The enforcement decision is made on the hot
path — the tool call is allowed or denied immediately, without waiting for any external
service.

Receipts are written to a serialized writer thread asynchronously. Allowed calls are
forwarded to the upstream without waiting on disk flush. Denied calls are still receipted —
the deny decision produces a receipt even though no upstream call was made.
```

This keeps the gateway on the hot path fast. The agent's tool call latency is not increased by receipt I/O. The evidence trail is preserved without blocking the decision. The same pattern governs the sink's best-effort sidecar writes (Neo4j/Qdrant failures don't block ingest) and the session-rules hook (fails open, caps graph calls at 2 seconds).

Anti-Patterns Addressed

Broad Workstation-Local Allow-Lists (AP5)

Gateway roles are code-owned, not user-editable. The 12 built-in roles are defined in the gateway's Rust source. An operator cannot create a local file that says "my agent is allowed to use all tools" — the role is assigned via the `--role` flag and validated against the built-in definitions. Enterprise policy bundles can override role definitions, but bundles must be signed by the org's Ed25519 key and verified against a pinned SPKI public key. A bundle signed by any other key is rejected.

This prevents the most likely self-degradation path: an operator creating a local allow-list that disables the policy enforcement. The role system is structural — it is enforced in code, not in configuration — and the bundle system requires organizational signing authority to modify.

Direct Vault/Credential Scraping (AP6)

The gateway blocks direct secret-file reads in production mode. When Guard argument scanning is enabled, a `tools/call` request that includes arguments like `cat ~/.ssh/id_ed25519` or `aws secretsmanager get-secret-value` is blocked before the tool executes. The block produces a signed receipt with the risk match (`sensitive-file-content-read` or `aws-secrets-manager-read`), preserving the evidence that the attempt was made and denied.

The sanctioned path for credential access is through approved tools — SecureGit brokered secret execution with handles (`securegit secret run --with-secret NAME=handle -- <cmd>`), OpenBrain MCP/read APIs, or metadata-only MCP tools. The gateway allows these sanctioned paths because they attribute access and keep raw values out of transcripts.

The MCP Protocol: Background

The Model Context Protocol is a JSON-RPC 2.0 protocol over stdio (or HTTP+SSE in the full spec). The core methods are:

Method	Direction	Purpose
`initialize`	Client → Server	Capability handshake
`tools/list`	Client → Server	Request the tool catalog
`tools/call`	Client → Server	Invoke a specific tool
`notifications/initialized`	Client → Server	Post-handshake notification

Messages are framed with `Content-Length` headers (similar to LSP):

```
Content-Length: 1234\r\n \r\n
```

The gateway proxies these frames between the client and the upstream server, intercepting `tools/list` responses (to filter the catalog) and `tools/call` requests (to enforce RBAC). All other messages — `initialize`, `notifications/initialized`, and any server-to-client notifications — pass through transparently.

Receipt Verification and Conformance Testing

The gateway's receipt system is conformance-tested against a cross-language fixture: `governance-receipt-v1.fixture.json`. This fixture contains:

- A known receipt body (19 fields with specific values)
- The expected RFC 8785 JCS canonical bytes
- The expected SHA-256 digest
- A known Ed25519 signing key pair
- The expected signature

Any implementation — Rust, Python, Go, Java, TypeScript — can load this fixture, canonicalize the receipt body, compute the digest, and verify the signature. If the implementation produces the same canonical bytes, digest, and signature, it conforms to the contract.

The conformance test is critical because RFC 8785 canonicalization has subtle edge cases:

- **Number representation:** `1.0` and `1` must produce different canonical forms (RFC 8785 preserves the distinction)
- **String escaping:** Unicode characters must be escaped according to specific rules (some escaped, some not)
- **Key ordering:** Object keys must be sorted lexicographically by UTF-16 code unit
- **Whitespace:** No insignificant whitespace in canonical form

A local serializer (like `serde_json::to_string`) might produce different output for any of these cases. The `serde_json_canonicalizer` crate implements RFC 8785 exactly, and the fixture test proves it.

The Receipt Envelope Shape

Each receipt is emitted as a JSONL line with three components:

```json
{
  "signed_body": { /* 19 fields */ },
  "attestation": {
    "alg": "Ed25519",
    "key_id": "acme-pep-claude-laptop",
    "sig": "base64-encoded-signature"
  },
  "canonical_sha256": "hex-encoded-sha256-of-jcs-canonical-signed-body",
  "schema_version": "governance.receipt.v1",
```

```
  "public_key": {
    "spki_pem": "-----BEGIN PUBLIC KEY-----\n..."
  },
  "rule_context": {
    "role": "frontend",
    "guard_mode": "production",
    "policy_mode": "enforce",
    "dry_run": false
  }
}
```

The `signed_body`, `attestation`, and `canonical_sha256` are the mesh-client receipt envelope from `orca-mesh-client`. The `schema_version`, `public_key.spki_pem`, and `rule_context` are local JSONL siblings that support verification without a separate key store.

The `public_key.spki_pem` field allows any consumer to verify the signature independently: load the SPKI public key, recompute the JCS canonical bytes of `signed_body`, and verify the Ed25519 signature. The `canonical_sha256` allows a quick integrity check without re-canonicalizing: compute `sha256(JCS(signed_body))` and compare.

The `rule_context` field carries the enforcement context: which role was active, which guard mode, which policy mode, and whether dry-run was enabled. This is the evidence that an auditor needs to understand not just what was decided but under what configuration it was decided.

Dry-Run Mode: Observe Before Enforce

The `dry-run` policy mode is the gateway's observe-mode rollout mechanism. In dry-run:

1. The policy is evaluated normally — roles are intersected, floor denies are checked, RBAC rules are applied.
2. The decision is recorded in the receipt with `policy_decision: not_evaluated` and `rule_context.dry_run = true`.
3. The call is allowed regardless of the policy decision — both allowed and would-be-denied calls are forwarded to the upstream.

This means an organization can deploy the gateway in dry-run mode, collect receipts for a period, analyze what would have been blocked, and then switch to enforce mode when they're confident the policy is correct.

The receipt trail from dry-run mode is valuable for policy tuning. An auditor can query the receipt log for all `dry_run=true` receipts where the policy decision was `deny` — these are the calls that would have been blocked in enforce mode. If the list includes legitimate work calls, the policy needs adjustment before switching to enforce.

Next Required Work

The gateway spec documents the remaining work items:

1. **Replace local policy-bundle files** with the org PAP (Policy Administration Point) distribution channel and revocation/update signal. Today, policy bundles are loaded from local files via `--policy-bundle`. The production path is a PAP-managed distribution channel with revocation support.

2. **Add Looking Glass/NATS async emission** after Tank's mesh client binding is ready. Today, receipts are written to a local JSONL file. The production path is async emission to the Looking Glass observability plane via NATS.
3. **Add agent-vault/securegit credential brokering** and installer credential stripping. The gateway should integrate with the sanctioned broker path for credential access, and the installer should strip credentials from configuration files after setup.
4. **Add approval routing over the mesh approver queue.** Today, gray-area decisions produce a `permissionDecision: ask` that Claude Code surfaces locally. The production path is routing approval requests to a mesh-based approver queue for remote approval.
5. **Add HTTP/SSE MCP transport.** Today, the gateway only supports stdio MCP. The full MCP spec includes HTTP+SSE transport, which would allow the gateway to proxy HTTP-based MCP servers.
6. **Verify Gemini/AGY transport** before promising enforcement for that client. The gateway supports Claude Code and ArmyknifeClaw; Gemini/AGY transport needs verification before enforcement can be promised.

AgentChron MCP Server: Query Surface Details

The `agentchron-mcp.py` server provides 13 tools that map to the sink's HTTP API. Let's examine the key tools in detail:

`AGENTCHRON_SEARCH`

Paginated FTS search over ingested events. By default, it uses fast pagination (skip-take) and skips the expensive exact `COUNT` query. Pass `exact_total=true` only when the precise total matters — the COUNT query can be slow on large event stores.

Useful filters: `q` (full-text query), `source` (source path substring), `host`, `agent`, `session_id`, `kind`, `tool`, `limit`, `offset`.

`AGENTCHRON_EVENT`

Fetch one event by `event_id` with full sanitized text, full tool inputs, source metadata, plugin findings, and the stored sanitized envelope JSON. This is the drill-down tool — use it after `agentchron_search` or `agentchron_session_events` when snippets are not enough and you need the full event body.

The `include_envelope` parameter (defaults to `true`) controls whether the full stored event body is returned. Disabling it returns only the extracted text and tool fields, which is faster for bulk scanning.

`AGENTCHRON_GRAPH_CONTEXT`

GraphRAG-style related context. Given a seed (query, session, event, source, host, or agent), the sink derives associated entities (tools, projects, files, Jira keys, commits, branches, symbols) and returns related sessions and entity/session links.

This is the institutional memory tool. After a search hit, an agent can call `agentchron_graph_context` with the hit's `session_id` or `event_id` to find nearby institutional memory — other sessions that touched the same files, commits, or tools. This is more useful than another keyword search because it leverages the graph structure rather than lexical matching.

Parameters: `q` (optional full-text seed), `session_id` or `event_id` (optional exact seed), `source`, `host`, `agent` (optional filters), `max_seed_events` (default 20, max 50), `limit` (related sessions, default 10, max 50).

`AGENTCHRON_SOURCE_COVERAGE`

Exact-source coverage check. Compares the number of indexed events for one `source_path` to an observed JSONL line count and reports `complete`, `partial`, `missing`, `over_ingested`, or `unknown`.

On `.114`, `ssh://developer@host/path` source paths are automatically checked against `/mnt/backups05/agentchron-raw/<host>/<path>` when that archive file exists. This allows coverage verification without SSH access to the original host.

`ORCA_WORKFLOW_CREATE`

Persist an `orca-panel` run with `emit_advisory=true` as a durable workflow improvement candidate. The `panel_run` JSON object is the full output of an `orca-panel` multi-agent review session.

The sink's `workflow_input_from_panel()` function (in `ingest.rs`) extracts findings, recommendations, reviewer IDs, evidence event IDs, risk, stance, and confidence from the panel run. It requires `consensus.emit_advisory=true` — only panel runs that reached an advisory consensus are eligible for promotion.

`ORCA_WORKFLOW_LIST_APPROVED`

Always filters `status=approved` and returns a compact active-rule shape: title, recommendation, risk, stance, scope, approval metadata, and evidence event IDs. This is the tool that agents should call at session start or before high-risk work to retrieve approved workflow guidance.

Role Enforcement: Concrete Examples

To understand how the RBAC engine works in practice, let's trace through several concrete enforcement scenarios.

SCENARIO 1: FRONTEND DEVELOPER EDITS A COMPONENT

```
Client: Claude Code with role=frontend
Request: tools/call Edit {file_path: "src/components/Button.tsx", content: "..."}
```

Enforcement: 1. Parse tool name: `Edit` → tool axis check: `frontend` allows `Edit` ✓ 2. Parse file path: `src/components/Button.tsx` → domain axis check: `frontend` allows `components/**` ✓ 3. No floor deny matches 4. Guard argument scan: no secrets in arguments ✓ 5. **Decision: allow** — forward to upstream MCP server 6. Post-exec receipt: record `result_digest` of the Edit response

SCENARIO 2: FRONTEND DEVELOPER TRIES TERRAFORM

```
Client: Claude Code with role=frontend
Request: tools/call Edit {file_path: "terraform/prod/main.tf", content: "..."}
```

Enforcement: 1. Parse tool name: `Edit` → tool axis check: `frontend` allows `Edit` ✓ 2. Parse file path: `terraform/prod/main.tf` → domain axis check: `frontend` denies `terraform/**` ✗ 3. **Decision: deny** — rule `frontend-no-infra-paths` 4. Return JSON-RPC error to client 5. Receipt: record deny decision with rule context

SCENARIO 3: ADMIN HITS FLOOR DENY

```
Client: Claude Code with role=admin, floor-deny="mcp__aws__*"
Request: tools/call mcp__aws__listBuckets {}
```

Enforcement: 1. Parse tool name: `mcp__aws__listBuckets` → normalize to (server=aws, tool=listBuckets) 2. Tool axis check: `admin` allows all tools ✓ 3. MCP server axis check: `admin` allows all servers ✓ 4. Floor deny check: `mcp__aws__*` matches `mcp__aws__listBuckets` ✗ 5. **Decision: deny** — rule `floor-deny:mcp-aws` 6. Return JSON-RPC error to client 7. Receipt: record deny decision with floor-deny rule context

SCENARIO 4: NARROWING INTERSECTION

```
Client: Claude Code with role=frontend, instance-narrowing=cloud-engineer
Request: tools/call Bash {command: "terraform apply"}
```

Enforcement: 1. Effective roles: {frontend} ∩ {cloud-engineer} = {} (no overlap) 2. No effective role permits `Bash` with `terraform` command 3. **Decision: deny** — rule `frontend-no-cloud-shell` (the frontend role's constraint applies because the base role includes frontend) 4. Return JSON-RPC error to client

The narrowing intersection is nuanced but the decision is clear. The base roles are {frontend}, and the instance narrowing is {cloud-engineer}. The effective permission is base ∩ narrowing = {frontend} ∩ {cloud-engineer}. Since these are different disciplines with no overlap, the effective permission set is empty. The system evaluates each role's constraints independently and takes the most restrictive result.

The decision conformance vector "narrowing cannot widen" clarifies: base= `frontend` , narrowing= `cloud-engineer` , request= `Bash terraform apply` , expected= `deny` with rule `frontend-no-cloud-shell` . This means the base role's constraint (frontend cannot run cloud shell commands) is preserved even when the instance claims a cloud-engineer narrowing. The narrowing adds the cloud-engineer's permissions, but the base role's denies still apply. The effective permission is the intersection — and since frontend denies cloud shell commands, the deny holds. The cloud-engineer narrowing cannot widen the frontend base role to permit commands the base role explicitly forbids.

The Four-Plane Trust Context

The MCP gateway is the enforcement plane in Orca's four-plane trust architecture:

```
AgentShield ENFORCES → Hermes IDENTIFIES → ContextOS ATTESTS → Orca OBSERVES
        ^                                                        |
        `------ Workflow library feeds back as new rules --------'
```

The gateway is where AgentShield's enforcement meets the MCP protocol. But the gateway also depends on the other three planes:

- **Hermes identifies** — The `--employee-did` and `--instance-id` flags carry Hermes identity. The gateway uses the employee DID to resolve base roles and the instance ID to apply narrowing. Without Hermes identity, the gateway cannot determine the effective permission set.
- **ContextOS attests** — Policy bundles are signed and verified against pinned org keys. The attestation plane proves that a given policy bundle was authorized by the organization. Without ContextOS attestation, a bundle could be forged by anyone with access to the gateway host.

- **Orca observes** — Every gateway decision produces a receipt that feeds into the Orca evidence trail. The receipts are stored in the JSONL log, queryable via the AgentChron MCP server, and exportable as compliance reports. Without Orca's observation plane, the gateway's decisions would be enforced but not auditable.

The zero overlap rule is explicit: AgentShield enforces, Hermes identifies, ContextOS attests, Orca observes and correlates. The gateway doesn't identify the user (Hermes does that); it doesn't attest to policy authorization (ContextOS does that); it doesn't store the evidence trail (Orca does that). It enforces policy on MCP tool calls and produces signed receipts.

Compliance Report Structure

The compliance report is a JSON document that maps the receipt chain to regulatory framework evidence. The structure includes:

- **Receipt chain summary:** Total receipts, allowed count, denied count, attestation status
- **Framework mappings:** Evidence mapped to EU AI Act Article 12, ISO 42001 Annex A.6.2.8, NIST AI RMF
- **Attestation verification:** Per-receipt Ed25519 signature verification results
- **Policy context:** Roles, floor denies, guard modes, policy modes active during the reporting period
- **Readiness stamp:** `READY` if all receipts have valid attestations, `PENDING_ATTESTATION` otherwise

The `READY` stamp is the high bar. It means the report is not just a list of decisions — it is a cryptographically verified audit trail. An external auditor can:

1. Load the compliance report
2. Extract each receipt's `signed_body` and `attestation`
3. Recompute `sha256(JCS(signed_body))` and compare to `canonical_sha256`
4. Verify the Ed25519 signature against `public_key.spki_pem`
5. Confirm that every receipt passes verification

If any receipt fails verification, the report's readiness stamp drops to `PENDING_ATTESTATION`. This is tamper evidence — a modified receipt will fail the signature check, and the report will reflect the failure.

The EU AI Act Article 12 mapping requires "automatic recording of events" — the receipt chain provides this. The ISO 42001 Annex A.6.2.8 mapping requires "logging of AI system operation" — the policy context and decision receipts provide this. The NIST AI RMF mapping requires "governance and risk management evidence" — the role-based access control, floor denies, and Guard integration provide this.

Conclusion

The Orca MCP Governance Gateway is the policy enforcement point for enterprise AI coding assistants. It intercepts MCP `tools/list` for catalog filtering and `tools/call` for RBAC enforcement, producing signed Ed25519 receipts for every decision.

The twelve built-in roles provide discipline-scoped access control — a frontend developer's agent can edit components but cannot run terraform; a cloud engineer's agent can manage infrastructure but cannot edit application code. The narrowing-only role intersection ensures that agent authority is always bounded by human authority: agent instance roles may only narrow, never widen, the base human DID roles. Org floor denies apply above all roles and cannot be weakened by any role, instance profile, or local config.

The receipt system uses RFC 8785 JCS canonicalization for cross-language conformance, Ed25519 signing for tamper evidence, and identity anchor chaining for chain-of-custody across restarts. The frozen 19-field receipt body

is a contract — new evidence types are additive sidecars, not schema changes. Compliance reports map the receipt chain to EU AI Act Article 12, ISO 42001 Annex A.6.2.8, and NIST AI RMF evidence, with `READY` status only when every receipt has a valid Ed25519 attestation.

The Guard integration adds secret detection to the MCP tool call path. The four guard modes (production, dev, audit, max) and two policy modes (enforce, dry-run) provide graduated enforcement from observe-mode rollout to deny-all. The dry-run mode enables observe-mode deployment — you can see what the policy would decide without blocking anything — which is critical for organizational rollout.

The AgentChron MCP server provides the query surface for agents to access the Orca evidence trail. The 13 tools cover search, event detail, graph context, session summaries, security findings, and workflow CRUD. The recommended SSH-based deployment keeps the sink token on the trusted host, so secrets never cross the network to the developer's laptop.

The gateway is the enforcement plane of the four-plane trust architecture: AgentShield enforces, Hermes identifies, ContextOS attests, Orca observes. The gateway is where AgentShield's enforcement meets the MCP protocol — and where every decision becomes a signed, verifiable, compliance-ready receipt.

Chapter 8: Presence Attestation

Every governance receipt in Orca answers one question well: *was this tool call allowed?* It carries the decision, the policy that drove it, the digests of the arguments and result, and an Ed25519 signature over a canonical body. What it does not carry — and deliberately never will — is proof that a specific human was present and approved the action in the physical world. That second question is the job of `presence.attestation.v1`, the root-of-trust sidecar that links a hardware-backed human verification event to an existing governance receipt without modifying the receipt's frozen 19-field body.

This chapter walks through the presence attestation design in detail: why it is a sidecar rather than a new receipt field, the exact shape of the signed envelope, the validation rules that make the challenge binding, the hardware provider abstraction that lets YubiKey PIV, TPM, Secure Enclave, and Entra CNG all plug into one envelope, the `FleetNode.v1` aggregation row that denormalizes presence evidence for three consumers, the signed-geo invariant that prevents coordinate tampering, the freshness and reachability rules that keep the fleet map honest, and the end-to-end chain of custody that ties human approval to agent action to stored evidence.

Design Philosophy

The first design decision is the most consequential: `presence.attestation.v1` is a **sidecar** to `governance.receipt.v1`, not a new field in the frozen 19-field receipt body. The receipt body remains frozen; presence evidence links to it. This is Pattern 7 from the platform's pattern synthesis — *Frozen Schema with Additive Sidecars* — and it is load-bearing for three reasons.

First, the governance receipt body is a cross-language contract. Fixtures like `docs/governance-receipt-v1.fixture.json` assert byte-identical canonicalization and signatures across Rust, TypeScript, Python, Go, and Java implementations. Adding a field to the body would break every fixture and every consumer simultaneously. Second, not every governance decision requires human presence. The vast majority of tool calls are allowed by role policy without a human in the loop; inflating every receipt with nullable presence fields would penalize the common case. Third, presence evidence arrives on a different trust path than the receipt itself. The receipt is produced

synchronously inside the MCP gateway on the hot path; presence evidence is produced asynchronously by a hardware touch, a PIN entry, or a biometric prompt on a separate device. Coupling them in one signed body would force the gateway to wait for hardware, violating Pattern 11 — *Synchronous Enforcement, Asynchronous Telemetry*.

The sidecar approach solves all three problems. The receipt body stays frozen and fast. Presence evidence is produced when the human is ready, signed independently, and linked back to the receipt by a hash challenge. Consumers that care about presence (the fleet map, ContextOS wallet, governance audit views) join the two records; consumers that don't (the gateway's own allow/deny decision) ignore the sidecar entirely.

The second design decision is the **signed-geo invariant**. When a presence record feeds the fleet map, `geo` is part of the signed body — not a client-side sidecar attached after signing. This means `signature_verified=true` on a `FleetNode.v1` row guarantees that the signature was checked over the canonical presence body *including* the geographic coordinates. If an attacker tampers with a latitude or longitude after the fact, the canonical hash changes and verification fails. The geo field is optional (a provider can set `source: "withheld"` and omit coordinates), but when it is present, it is signed. There is no path by which a consumer can trust a location that was not covered by the hardware-backed signature.

The Frozen 19-Field Receipt Body

Chapter 8 repeatedly refers to the "frozen 19-field receipt body" of `governance.receipt.v1`. The body is a cross-language conformance contract: every implementation (Rust, TypeScript, Python, Go, Java) must produce byte-identical RFC 8785 JCS canonicalization and the same Ed25519 signature from the same private key, as asserted by the `governance-receipt-v1.fixture.json` conformance vectors. Adding a field would break every fixture and every consumer simultaneously, which is why the body is frozen and presence evidence attaches as a sidecar rather than a 20th field. The 19 fields, drawn from the fixture's body object, are:

#	FIELD	TYPE	DESCRIPTION
1	`schema_version`	string	Always `governance.receipt.v1`.
2	`receipt_id`	string (ULID)	Unique identifier for this receipt.
3	`identity_anchor`	string (hex)	SHA-256 of the previous receipt's canonical body; chain linkage. Empty for genesis.
4	`tenant_id`	string	Organization/tenant scope.
5	`employee_did`	string (DID)	Human identity anchor supplying base roles.
6	`instance_id`	string	Agent instance identifier.
7	`action_kind`	string	`knowledge_query`, `tool_call`, etc.
8	`tool_name`	string	Tool invoked; empty for non-tool actions.
9	`args_digest`	string (hex)	`sha256(JCS(args))`, bare lowercase hex; empty when no args.
10	`result_digest`	string (hex)	`sha256(JCS(result))`, bare lowercase hex; empty when no result.
11	`policy_decision`	string	`allow`, `deny`, or `not_evaluated` (dry-run).
12	`approval_decision`	string	`approved`, `denied`, `timeout`, or `not_required`.
13	`approver_did`	string (DID) or null	DID of the human approver; null when not required.
14	`agentshield_rule_id`	string or null	Rule ID that fired; null when no rule matched.
15	`cost`	object or null	`{amount_cents, currency}` when upstream exposes cost.
16	`ts`	string (ISO 8601)	Decision timestamp.
17	`nonce`	string (hex)	128-bit anti-replay nonce.
18	`client_type`	string or null	`claude`, `codex`, `gemini`, etc.
19	`upstream_server_id`	string or null	MCP server ID; null for non-MCP calls.

The fixture also surfaces `mcp_method` (e.g. `tools/call`) as a body field in the conformance vectors; implementations that track it include it within the 19-field count, and implementations that do not track MCP method leave it `null`. The signed bytes are RFC 8785 JCS of these 19 fields — the `attestation` envelope wrapping the body is excluded from the signed canonical bytes, so a receipt may carry an attestation sidecar without re-signing the body.

Envelope Shape

The presence record is a signed envelope with three top-level components: `signed_body`, `attestation`, and `canonical_sha256`. The canonical form is defined in `docs/ORCA_PRESENCE_ATTESTATION_V1.md` on the invent host. Here is the full shape:

```
{
  "signed_body": {
    "schema_version": "presence.attestation.v1",
    "presence_id": "presence_...",
    "tenant_id": "acme",
    "employee_did": "did:hermes:acme:emp:42",
    "approver_did": "did:hermes:acme:emp:ciso",
    "instance_id": "did:hermes:acme:emp:42:claude:laptop",
    "edge_instance_id": "contextos-edge:macbook-pro:42",
    "governance_receipt_id": "01J...",
    "canonical_body_sha256": "64 lowercase hex chars",
    "challenge_nonce": "128-bit random hex",
    "challenge_deadline_ts": "2026-06-18T19:31:00.000Z",
    "action_summary": "Human-readable action shown before touch/approval.",
    "approval_decision": "approved",
    "presence_provider": "yubikey_piv",
    "presence_method": "pin_touch",
    "assurance_level": "hardware_backed_user_verification",
    "evidence_format": "piv_pin_touch_v1",
    "evidence_digest": "64 lowercase hex chars",
    "hardware_key_id": "piv:slot-9c:sha256:...",
    "workstation_id": "macbook-pro-42",
    "geo": {
      "lat": 30.2711,
      "lon": -97.7437,
      "accuracy_m": 10.0,
      "source": "corelocation",
      "captured_at": "2026-06-18T19:30:04.000Z"
    },
    "ts": "2026-06-18T19:30:05.000Z"
  },
  "attestation": {
    "alg": "ed25519",
    "key_id": "presence-key-id",
    "sig": "base64 signature over JCS(signed_body)"
  },
  "canonical_sha256": "sha256(JCS(signed_body))"
}
```

The `signed_body` carries the semantic payload. Let's walk through the fields that matter most.

`canonical_body_sha256` is the challenge. It is the SHA-256 of the exact RFC 8785 JCS bytes of the governance receipt body being approved. This is Pattern 10 — *Hash-Only Provenance* — applied to the link between presence and receipt. The presence record does not embed the receipt body; it references it by hash. A verifier who has the receipt body can recompute `sha256(JCS(receipt_body))` and confirm it matches `canonical_body_sha256`. If the receipt body was tampered with, the hash breaks. If the presence record was fabricated, the signer would have had to know the receipt body's canonical hash at signing time — which means they saw the body.

`challenge_nonce` and `challenge_deadline_ts` form a freshness window. The nonce is 128 bits of random hex generated by the attestation broker when it requests human approval. The deadline is the timestamp after which the

challenge expires. This prevents replay: a captured presence envelope cannot be reused to approve a different receipt because the nonce and deadline are bound into the signed body, and a new approval requires a new challenge.

`approval_decision` records the human's verdict: `approved`, `denied`, or `timeout`. A denied or timed-out presence record is still a valid, signed envelope — it proves the human was asked and did not approve. This is important for audit: the chain of custody must show not only what was approved, but what was refused.

`presence_provider`, `presence_method`, and `assurance_level` describe how the human was verified. In the example above, the provider is `yubikey_piv`, the method is `pin_touch` (PIN entry plus physical touch of the key), and the assurance level is `hardware_backed_user_verification`. These three fields are the bridge between the software-verifiable record and the physical hardware event.

`evidence_format` and `evidence_digest` are the hardware provider abstraction's storage hooks. The raw WebAuthn, PIV, or provider assertion — the actual cryptographic proof from the hardware — is stored out-of-band, referenced here by format and digest. This keeps the envelope small and schema-stable while allowing the full assertion to be archived separately for deep audit.

`geo` is the geographic evidence block. It is optional but, when present, signed. The `source` field records how the coordinates were obtained: `corelocation` (macOS Core Location), `datacenter` (declared datacenter coordinates for server-side agents), `declared` (manually declared by the operator), or `withheld` (provider chose not to share). The `captured_at` timestamp records when the fix was taken, which may differ from the attestation timestamp `ts`.

The `attestation` block carries the Ed25519 signature. The `alg` is `ed25519`, the `key_id` identifies which presence signing key was used, and `sig` is the base64-encoded signature over `JCS(signed_body)` — the RFC 8785 JSON Canonicalization Scheme bytes of the signed body. This is Pattern 9 — *RFC 8785 JCS Canonicalization with Cross-Language Conformance* — the same pattern used for governance receipts, gold DSSE receipts, and every other signed artifact in the platform.

The `canonical_sha256` field is `sha256(JCS(signed_body))` — the digest of the canonical bytes that were actually signed. A verifier recomputes this, compares it to the field, and only then checks the signature. This two-step (hash, then verify) lets consumers detect canonicalization bugs independently of the signature check.

Validation Rules

The presence envelope is useless without strict validation. The rules are explicit in the spec and enforced by the `orca-mesh-sign --presence --verify` CLI and by any consumer that trusts a `FleetNode.v1` row.

Challenge hash rule. `canonical_body_sha256` must be the SHA-256 of the exact RFC 8785 JCS bytes of the governance receipt body being approved. This is not a loose reference — the verifier must have the receipt body, canonicalize it with RFC 8785, hash it, and compare. Empty challenge hashes fail validation immediately. Non-lowercase hex digests fail — the spec mandates bare lowercase hex with no `sha256:` prefix, matching the digest convention used for `args_digest` and `result_digest` in governance receipts.

Schema version rule. `schema_version` must be exactly `"presence.attestation.v1"`. Unknown versions fail. This is forward-compatible: a future `presence.attestation.v2` can coexist, but a v1 verifier will reject it rather than misinterpreting the fields.

Approval decision rule. `approval_decision` must be one of `approved`, `denied`, `timeout`. Any other value fails validation. This is an enum, not a free-text field.

Geo source rule. `geo.source` must be one of `corelocation`, `datacenter`, `declared`, `withheld`. If `source` is `withheld`, the geo object must not carry `lat`, `lon`, or `accuracy_m` — a withheld source with coordinates is a contradiction and fails validation. The other three sources must carry coordinates.

Signature rule. The `attestation.sig` must be a valid Ed25519 signature over `JCS(signed_body)` that verifies against the public key identified by `attestation.key_id`. The verifier recomputes `canonical_sha256` from the `signed_body` and confirms it matches the field before checking the signature, catching canonicalization errors early.

These rules are not advisory. The spec states: "Empty challenge hashes, non-lowercase digests, and unknown schema versions fail validation." A presence record that fails any of these checks is not evidence — it is noise, and consumers must treat it as unverified.

Hardware Provider Abstraction

One of the hardest problems in presence attestation is hardware diversity. A developer on a MacBook might use a YubiKey PIV touch. A server-side agent in a datacenter might use a TPM-attested broker. An enterprise deployment might use Entra CNG (Certificate Next Generation) keys backed by Azure AD. A mobile agent might use the Secure Enclave. Each of these hardware providers produces evidence in a different format — WebAuthn assertions, PIV signatures, TPM quotes, Secure Enclave attestation blobs — and none of them should inflate the presence envelope schema.

The abstraction solves this with two fields: `evidence_format` and `evidence_digest`. The `evidence_format` is a string identifier for the provider's assertion format (e.g., `piv_pin_touch_v1`, `webauthn_assertion_v1`, `tpm_quote_v1`). The `evidence_digest` is the SHA-256 of the raw assertion bytes, which are stored out-of-band in an evidence archive. The presence envelope carries only the format tag and the digest; the raw assertion is retrieved separately if a deep audit requires it.

This design lets new providers plug into the same envelope without schema changes. The current providers enumerated in the spec are:

PROVIDER	METHOD	ASSURANCE	EVIDENCE FORMAT
YubiKey PIV	PIN + touch	`hardware_backed_user_verification`	`piv_pin_touch_v1`
uTrust PIV	PIN + touch	`hardware_backed_user_verification`	`piv_pin_touch_v1`
TPM	TPM quote	`hardware_backed_system_attestation`	`tpm_quote_v1`
Secure Enclave	biometric / device PIN	`hardware_backed_user_verification`	`secure_enclave_v1`
Entra CNG	Azure AD cert key	`enterprise_managed_key`	`entra_cng_v1`
Edge touch-broker	proxied touch	`brokered_user_verification`	`edge_broker_v1`

The `assurance_level` field is the consumer-facing summary. A fleet map consumer that needs high-assurance presence can filter for `hardware_backed_user_verification` and ignore `brokered_user_verification` records. A consumer that needs only a soft signal can accept any assurance level.

The CLI for creating and verifying presence records is `orca-mesh-sign`:

```
export ORCA_SIGNING_PHRASE='<broker injected>'
export ORCA_KEY_ID='presence-dev-key'

# Create a presence envelope from a signed_body JSON on stdin
cat presence-body.json | orca-mesh-sign --presence

# Verify an existing envelope
cat presence-envelope.json | orca-mesh-sign --presence --verify

# Emit only the canonical hash for a signed_body
cat presence-body.json | orca-mesh-sign --presence --canonical
```

A critical operational detail: the signing phrase is injected by broker or environment variable, **never passed on argv**. This is because argv is visible in process listings (`ps aux`) and in shell history. The broker — typically SecureGit's secret broker or an environment-injection sidecar — populates `ORCA_SIGNING_PHRASE` in the process environment at launch time, and the CLI reads it from there. This is the same principle that governs `ORCA_FOUNDRY_SIGNING_KEY` in the Data Foundry gold promotion path and `ORCA_MCP_SIGNING_SEED_HEX` in the MCP gateway: signing material never touches argv.

FleetNode.v1: Aggregation Output

Presence attestation records are individually verifiable, but consumers like the Command Center Fleet Map, the ContextOS Wallet, and governance audit views need a denormalized, queryable view of the fleet. That view is `FleetNode.v1` — Orca's fleet aggregation row.

`FleetNode.v1` is emitted once per agent node and read by three consumers, each extracting different subsets of fields. This three-consumer pattern prevents the map, wallet, and governance planes from drifting into incompatible fleet shapes. If the fleet map and the wallet disagree on which agents exist or where they are, the cause is a consumer bug, not a data divergence.

The fields are:

FIELD	TYPE	DESCRIPTION
`agentId`	string	Stable agent identifier
`name`	string	Human-readable agent name
`kind`	string	Agent kind (e.g., `claude`, `codex`, `agentchron`)
`hostname`	string	Canonical hostname (`host` accepted as legacy alias)
`os`	string	Operating system
`did`	string	Decentralized identifier for the agent
`region`	string	Region or datacenter identifier
`reachable`	boolean	Whether Orca can currently reach the node
`last_attestation`	string (RFC 3339)	Timestamp of the most recent presence attestation
`signature_verified`	boolean	Whether the last attestation's signature verified
`geo.lat`	float	Latitude from the attestation's signed geo
`geo.lon`	float	Longitude from the attestation's signed geo
`geo.accuracy_m`	float	GPS accuracy in meters
`geo.source`	string	`corelocation`, `datacenter`, `declared`, `withheld`
`geo.captured_at`	string (RFC 3339)	When the geo fix was captured
`active_signer`	string	Current active signing key ID
`hardware_rot`	boolean	Whether hardware key rotation is pending

The `geo` sub-object is pulled directly from the backing `presence.attestation.v1`'s `signed_body.geo`. The `signature_verified` flag is the result of verifying the backing attestation's Ed25519 signature over its canonical body including geo. The `last_attestation` timestamp is the `ts` field from the attestation's `signed_body`.

Critically, `FleetNode.v1` is **unsigned denormalized output**. It is not itself a signed record. It is a convenience aggregation produced by reading and verifying backing attestations. The consumer trust boundary is explicit: consumers must re-check the backing `presence.attestation.v1` before using `signature_verified`, `geo`, or `reachable` for a trust decision. The FleetNode row is a cache; the attestation is the truth.

The Signed-Geo Invariant

The signed-geo invariant is the single most important security property of the fleet aggregation layer. Let's trace exactly why it matters.

Consider a fleet map that shows agent positions. Without the signed-geo invariant, the flow would be: the agent reports its location to the fleet aggregator, the aggregator stores it in a FleetNode row, and the map renders it. An attacker who can modify the aggregator's database — or intercept the agent's report — can move the agent's pin to a different city. The map shows false data, and any trust decision based on location ("this agent is in the approved datacenter") is compromised.

With the signed-geo invariant, the flow is: the agent's presence provider captures a geo fix and includes it in the `signed_body.geo` of the presence attestation. The attestation is signed with an Ed25519 key backed by hardware. The fleet aggregator reads the attestation, verifies the signature over `JCS(signed_body)` — which includes the geo

bytes — and populates the FleetNode row. If anyone tampers with the geo coordinates in the FleetNode row, the row no longer matches the backing attestation. If anyone tampers with the geo in the backing attestation, the canonical hash changes and the signature no longer verifies.

The invariant is: `signature_verified=true` **means the signature was checked over the canonical body including geo.** Not just the DID, not just the host identity, not just the approval decision — the geo bytes are inside the signed body, and the signature covers them.

This is why `geo.source: "withheld"` must not carry coordinates. If a provider withholds location, the `geo` object in the signed body contains only `{"source": "withheld"}`. The signature covers that. A consumer cannot later inject coordinates into a withheld geo record because doing so would change the canonical hash and break verification. The withheld choice is itself signed — the provider proved it chose not to share location, and that proof is tamper-evident.

The consumer trust boundary deserves restating because it is the most likely point of confusion. `FleetNode.v1` is a denormalized, unsigned, convenience aggregation. A consumer that reads a FleetNode row and sees `signature_verified: true, geo: {lat: 30.27, lon: -97.74}` should understand: *the backing attestation's signature was verified, and at that time the geo matched.* But the FleetNode row itself is not signed. If the consumer is making a security-relevant decision — "should I route sensitive work to this agent based on its location?" — it must fetch the backing `presence.attestation.v1` and re-verify. The FleetNode row is a UI hint, not a trust anchor.

Freshness and Reachability

The fleet map is a live view, not a static directory, and freshness is a security property. A node that was reachable and attested an hour ago may now be compromised, decommissioned, or offline. The `reachable` field on FleetNode.v1 encodes Orca's current assessment of whether the node is contactable.

The rule is: **a reachable node must carry** `last_attestation`. If Orca cannot find a recent, valid presence attestation for a node, the node is marked `reachable=false` regardless of network connectivity. This prevents a scenario where a compromised node is online and responsive but has no recent human-verified presence — the fleet map shows it as unreachable, and consumers do not route work to it.

What counts as "recent" is a deployment-configurable staleness threshold. If `last_attestation` is older than the threshold, the node's `reachable` drops to `false`. The attestation itself is still valid (signatures don't expire by age alone), but the freshness guarantee is gone. The `challenge_deadline_ts` in the presence envelope provides per-challenge expiry; the fleet-level staleness threshold provides ongoing freshness.

Edges between nodes — which agent delegates to which, which agent mirrors to which — are deliberately **not** part of FleetNode.v1. The spec reserves a future `FleetEdge.v1` shape for this:

```
FleetEdge.v1:
  agentId_from: string
  agentId_to: string
  kind: string   (delegation, mirror, parent, ...)
  ts: string (RFC 3339)
  signature_verified: boolean
```

Keeping nodes and edges separate prevents the fleet map from becoming a graph database. The map renders nodes; delegation graphs are a separate concern with their own consumer. This separation also means that adding edges later does not change the FleetNode.v1 schema — another instance of the frozen-schema-with-additive-sidecar pattern.

Chain of Custody Across the Platform

The presence attestation is one link in a longer chain. Let's trace the full chain of custody from human approval to agent action to stored evidence, showing how each link is bound to the next.

Step 1: Governance receipt. An agent attempts a tool call through the MCP Governance Gateway. The gateway evaluates the call against discipline-scoped RBAC roles, produces a `governance.receipt.v1` with a 19-field body, signs it with Ed25519 over `JCS(receipt_body)`, and writes it to a serialized receipt log. The receipt has a unique `governance_receipt_id` (a ULID) and a canonical body whose SHA-256 can be recomputed by anyone who has the body.

Step 2: Presence attestation. For high-assurance actions, the gateway or the agent's edge broker requests human approval. The broker generates a challenge nonce and deadline, displays the `action_summary` to the human, and waits for hardware verification (e.g., YubiKey touch + PIN). The hardware produces a raw assertion. The broker constructs a `signed_body` containing the `governance_receipt_id`, the `canonical_body_sha256` (the challenge — SHA-256 of the receipt body's JCS bytes), the nonce, deadline, approval decision, provider metadata, evidence format/digest, optional geo, and timestamp. It signs the `signed_body` with the presence signing key and produces the `presence.attestation.v1` envelope.

Step 3: Fleet aggregation. The presence attestation is stored and indexed. The fleet aggregator reads recent attestations, verifies signatures, and emits `FleetNode.v1` rows. The Command Center Fleet Map renders these rows as pins on a map. The ContextOS Wallet uses them to show which agents have active, verified human presence. Governance audit views use them to show the approval chain for sensitive actions.

Step 4: End-to-end verification. An auditor who needs to prove that a specific agent action was human-approved follows the chain: find the governance receipt by `governance_receipt_id` → find the presence attestation by `canonical_body_sha256` matching the receipt body's hash → verify the attestation's Ed25519 signature → check the `approval_decision` is `approved` → check the `challenge_deadline_ts` has not expired relative to the action timestamp → check the `evidence_digest` against the archived raw assertion. Every step is cryptographic, not discretionary.

This chain is the platform's answer to a fundamental question of AI governance: *can you prove, after the fact, that a human authorized this agent action?* Without presence attestation, the answer is "the policy allowed it" — which is necessary but not sufficient. With presence attestation, the answer is "this specific human, verified by this hardware key, at this time and location, approved this specific action whose canonical body hash is X." That is a meaningfully different evidence standard.

THE HASH CHALLENGE IN DETAIL

The `canonical_body_sha256` challenge deserves closer examination because it is the cryptographic link between two independently signed records.

The governance receipt body is a 19-field JSON object. When the gateway signs it, it canonicalizes the body with RFC 8785 JCS and signs the resulting bytes. The canonical bytes have a SHA-256 digest. The presence attestation's `canonical_body_sha256` field is that digest.

This creates a binding that is both specific and privacy-preserving. It is specific because the digest uniquely identifies the exact receipt body — a single bit change in any field produces a different hash. It is privacy-preserving because the presence record does not embed the receipt body itself; it carries only the 64-character hex digest. An auditor who has both records can verify the link; a party who has only the presence record cannot recover the receipt body from the digest.

The binding is also one-directional in a useful way. The presence record points to the receipt, but the receipt does not point to the presence record. This means that receipts without presence attestations are perfectly valid — they

just lack the human-approval evidence. Adding presence attestations later does not require modifying or re-signing existing receipts. The receipt is frozen; the sidecar is additive.

CROSS-LANGUAGE CONFORMANCE

Because the challenge hash is `sha256(JCS(receipt_body))`, canonicalization must be byte-identical across languages. RFC 8785 JCS is the standard that guarantees this. The platform uses `serde_json_canonicalizer` in Rust, and equivalent RFC 8785 libraries in TypeScript, Python, Go, and Java. Conformance is proven by fixture tests — `governance-receipt-v1.fixture.json` contains a known receipt body, its expected canonical bytes, and the expected Ed25519 signature. Every implementation must produce the same canonical bytes and the same signature from the same private key.

This is not a theoretical concern. A canonicalization bug in any implementation — a different key ordering, a different number serialization, a different Unicode escape policy — would produce a different hash, breaking the challenge link. The fixture tests catch these bugs before they reach production. The presence attestation spec inherits this conformance contract because it depends on the receipt body's canonical hash.

Anti-Patterns Addressed

SINGLE SHARED TOKEN (AP1)

The fleet attestation path requires scoped tokens, not shared auth. A fleet aggregator that reads presence attestations and writes FleetNode rows should not use the same `AGENTCHRON_INGEST_TOKEN` that the agent uses for event ingest. The current single-token model on the origin sink is a known P0 gap: a compromised read token currently grants ingest and admin capabilities. The Cloudflare edge models split tokens (read, ingest, admin, origin), and the origin sink must follow suit.

The presence attestation path is particularly sensitive because it carries human identity and location data. A token that can read presence attestations should not be able to write events, and vice versa. The fleet aggregator's token should be scoped to read attestations and write FleetNode rows, nothing else. This is not a future concern — it is a current design constraint that the presence system must respect even if the origin sink has not yet implemented split tokens.

TREATING THE MODEL AS THE MOAT (AP9)

The attestation chain is part of the durable IP, not a model feature. A competitor who copies the Orca UI or trains a similar specialist model does not have the cryptographic chain of custody that presence attestation provides. The signed-geo invariant, the hash challenge link, the hardware provider abstraction, and the cross-language conformance fixtures are engineering assets that compound over time. They are not features that a model can replicate by generating similar-looking JSON.

This is the strategic dimension of Pattern 9. RFC 8785 JCS canonicalization with cross-language conformance is not just a technical choice — it is a moat. Every new consumer that verifies presence attestations (a new fleet map, a new compliance report, a new audit tool) benefits from the same conformance contract. The cost of building this contract is paid once; the value accrues to every consumer forever.

The Approval Workflow: A Concrete Walkthrough

To make the presence attestation design concrete, let's walk through a full approval workflow from the human's perspective, tracing each step through the envelope fields.

Scenario. An agent operating on a developer's MacBook attempts a high-assurance tool call — say, deploying a new version of the Orca sink to the production Docker Compose stack. The MCP Governance Gateway evaluates the call against the `devops` role and determines that this action requires human presence attestation (it is a production deployment, which the policy marks as `requires_presence: true`).

Step 1: Challenge issuance. The gateway produces a `governance.receipt.v1` with the tool call details, the policy decision (`allow`, conditional on presence), and the 19-field signed body. The receipt body is canonicalized with RFC 8785 JCS, and its SHA-256 is computed. This hash becomes the `canonical_body_sha256` challenge. The gateway also generates a 128-bit `challenge_nonce` and sets a `challenge_deadline_ts` five minutes in the future. The `action_summary` is derived from the tool call: "Deploy agentchron-sink:latest to production Docker Compose stack on ."

Step 2: Human presentation. The edge broker on the developer's MacBook receives the challenge and displays the `action_summary` to the human. The display includes the action text, the deadline countdown, and the identity of the requesting agent instance (`did:hermes:acme:emp:42:claude:laptop`). The human sees: "Agent `claude:laptop` wants to deploy agentchron-sink:latest to production. Approve? [Touch YubiKey to approve / Press Escape to deny]"

Step 3: Hardware verification. The human touches the YubiKey and enters their PIV PIN. The YubiKey produces a PIV signature — a raw cryptographic assertion that proves the user verified their presence with the hardware key in slot 9c. The assertion is stored in the evidence archive, and its SHA-256 becomes the `evidence_digest`. The `evidence_format` is `piv_pin_touch_v1`.

Step 4: Geo capture. Simultaneously, the edge broker captures a geo fix from macOS Core Location. The fix includes `lat`, `lon`, `accuracy_m`, and `captured_at`. The `source` is `corelocation`. If the developer has disabled location services, the broker sets `source: "withheld"` and omits coordinates — the absence of geo is itself signed.

Step 5: Envelope construction. The broker assembles the `signed_body` with all fields: `schema_version`, `presence_id` (a new ULID), `tenant_id`, `employee_did`, `approver_did`, `instance_id`, `edge_instance_id`, `governance_receipt_id` (the ULID from step 1), `canonical_body_sha256` (the challenge hash), `challenge_nonce`, `challenge_deadline_ts`, `action_summary`, `approval_decision: "approved"`, `presence_provider: "yubikey_piv"`, `presence_method: "pin_touch"`, `assurance_level: "hardware_backed_user_verification"`, `evidence_format`, `evidence_digest`, `hardware_key_id: "piv:slot-9c:sha256:..."`, `workstation_id: "macbook-pro-42"`, `geo` (the Core Location fix), and `ts` (the current timestamp).

Step 6: Signing. The `signed_body` is canonicalized with RFC 8785 JCS. The presence signing key (derived from the `ORCA_SIGNING_PHRASE` injected by the broker) signs the canonical bytes with Ed25519. The signature is base64-encoded into `attestation.sig`. The `canonical_sha256` is computed as `sha256(JCS(signed_body))`.

Step 7: Verification. The fleet aggregator (or any consumer) receives the envelope, recomputes `canonical_sha256` from `signed_body`, confirms it matches the field, verifies the Ed25519 signature against the presence public key, checks that `canonical_body_sha256` matches the SHA-256 of the governance receipt body, checks that `challenge_deadline_ts` has not expired, and checks that `approval_decision` is `approved`. If all checks pass, the FleetNode row is updated with `signature_verified: true`, `last_attestation: <ts>`, and the geo from the signed body.

Step 8: Denial path. If the human presses Escape instead of touching the YubiKey, the broker constructs the same `signed_body` but with `approval_decision: "denied"`. The envelope is still signed — it proves the human was asked and refused. The fleet aggregator records the denial. The gateway does not proceed with the deployment. The denial is part of the audit trail, not a silent cancellation.

Step 9: Timeout path. If the `challenge_deadline_ts` passes without human action, the broker constructs the envelope with `approval_decision: "timeout"`. This proves the human was asked but did not respond in time. The

deployment does not proceed. The timeout is auditable — a pattern of timeouts on a specific action may indicate that the action summary is unclear or that the human is overwhelmed with approval requests.

This three-outcome model (approved, denied, timeout) is deliberately richer than a binary allow/deny. It captures the full spectrum of human response, including non-response, which is itself information. An audit that shows "this deployment was approved" and one that shows "this deployment was requested but timed out" tell different stories, and the presence attestation envelope preserves both.

Deployment Scenarios for Presence

The presence attestation system must work across deployment targets, from a single developer's laptop to a fleet of server-side agents in a datacenter. The hardware provider abstraction is what makes this possible, but the deployment patterns differ.

Single developer (laptop). The developer's MacBook has a YubiKey plugged in. The edge broker runs locally, uses Core Location for geo, and signs with a presence key derived from a broker-injected phrase. The `presence_provider` is `yubikey_piv`, `assurance_level` is `hardware_backed_user_verification`, and `geo.source` is `corelocation`. This is the highest-assurance, highest-geo-fidelity scenario.

Server-side agent (datacenter). A server-side agent running on a Kubernetes node in a datacenter has no YubiKey and no GPS. The presence attestation is brokered through a TPM attestation service that verifies the node's integrity and the operator's identity through a separate channel (e.g., the operator's badge swipe at the datacenter door, correlated by timestamp). The `presence_provider` is `tpm`, `assurance_level` is `hardware_backed_system_attestation`, and `geo.source` is `datacenter` with the datacenter's declared coordinates. The geo is less precise (building-level, not desk-level) but still signed.

Enterprise managed (Entra CNG). An enterprise deployment uses Entra CNG keys backed by Azure AD. The human's identity is verified through Azure AD MFA, and the signing key is an Entra-managed certificate key. The `presence_provider` is `entra_cng`, `assurance_level` is `enterprise_managed_key`. Geo may be `withheld` if the enterprise policy prohibits location collection — the absence is signed.

Edge touch-broker (proxied). In a scenario where the agent runs on a headless server but the human is at a different workstation, an edge touch-broker mediates. The broker on the human's workstation handles the hardware touch and geo capture, then proxies the signed envelope to the agent's host. The `presence_provider` is `edge_broker`, `assurance_level` is `brokered_user_verification`. This is the weakest assurance level — the broker is a trusted intermediary, and a compromised broker could fabricate approvals. Consumers that require high assurance should filter for `hardware_backed_user_verification` and exclude `brokered_user_verification` records.

These scenarios show why the `assurance_level` field matters. It is not just metadata — it is a filter key that consumers use to enforce their trust requirements. A compliance report that requires hardware-backed human verification can query for attestations where `assurance_level` is `hardware_backed_user_verification` and exclude everything else. The presence system does not force all consumers to accept all assurance levels; it provides the information for each consumer to make its own trust decision.

The CLI in Detail

The `orca-mesh-sign` CLI has three modes for presence attestation, each serving a different operational need.

`--presence` **(signing mode).** Reads a `signed_body` JSON object from stdin, canonicalizes it with RFC 8785 JCS, signs it with the Ed25519 key derived from `ORCA_SIGNING_PHRASE`, and outputs the complete envelope (`signed_body` + `attestation` + `canonical_sha256`) as JSON. This is the mode the edge broker uses when constructing a new presence record.

`--presence --verify` **(verification mode).** Reads a complete envelope from stdin, recomputes `canonical_sha256` from the `signed_body`, confirms it matches the field, verifies the Ed25519 signature against the public key identified by `attestation.key_id`, and checks the validation rules (schema version, approval decision, geo source). Outputs a JSON verification report with `valid: true/false` and details on any failures. This is the mode consumers use to verify a presence record before trusting it.

`--presence --canonical` **(canonical hash mode).** Reads a `signed_body` JSON from stdin, canonicalizes it with RFC 8785 JCS, and outputs only the `sha256(JCS(signed_body))` hash as lowercase hex. This is useful for debugging canonicalization issues — a developer who suspects a canonicalization mismatch can compute the canonical hash of a `signed_body` and compare it to the `canonical_sha256` field in an existing envelope.

The signing phrase security model bears repeating because it is the most likely operational mistake. The `ORCA_SIGNING_PHRASE` environment variable must be injected by a broker or a trusted environment-injection mechanism, never passed on the command line. The CLI does not accept `--signing-phrase` as an argument. If a developer tries `orca-mesh-sign --presence --signing-phrase "my secret"`, the CLI will not use it — the phrase must come from the environment. This prevents the phrase from appearing in `ps aux`, in shell history, in process accounting logs, or in any other place where argv is recorded.

The broker (typically SecureGit's secret broker) is responsible for populating `ORCA_SIGNING_PHRASE` in the process environment at launch time. The broker itself reads the phrase from a secure source — a PIV key, a vault, or a sealed secret — and injects it into the CLI's environment via a fork-exec or a controlled spawn. The CLI never sees the broker's secure source; it only sees the environment variable that the broker set. This separation means the CLI can be updated, patched, or replaced without touching the broker's secret-handling code.

Cross-Platform Considerations

The signed-geo invariant must work across platforms, and each platform has different geo capabilities.

macOS (Core Location). The `corelocation` source uses macOS Core Location, which provides lat/lon with an accuracy estimate in meters. The accuracy varies: Wi-Fi positioning is accurate to ~10-50 meters, GPS (on devices with GPS hardware) is accurate to ~5-10 meters. The `accuracy_m` field in the geo object carries this estimate, and consumers can use it to filter out low-accuracy fixes. Core Location requires user permission; if the user has not granted location access to the edge broker, the broker falls back to `withheld`.

Linux (datacenter/declared). Linux servers typically do not have GPS hardware. The `datacenter` source uses declared datacenter coordinates — a fixed lat/lon for the datacenter building, configured by the operator. This is building-level accuracy, not desk-level. The `declared` source uses manually declared coordinates, which could be anything the operator enters. Both are signed — the signature proves the coordinates came from the attestation system, not from an attacker — but the precision is lower than Core Location.

Windows. Windows has a location API similar to Core Location, but its use in the Orca platform is less common (Windows workstations are typically desktops without GPS). The edge broker on Windows can use the Windows Location API if available, or fall back to `declared` or `withheld`.

Withheld across all platforms. The `withheld` source is universal. On any platform, the operator or the enterprise policy can choose to withhold location. The geo object contains only `{"source": "withheld"}`, and the signature covers this choice. A consumer cannot distinguish "withheld because the platform lacks GPS" from "withheld because enterprise policy prohibits location collection" — both produce the same signed record. This is intentional: the reason for withholding is not the consumer's business. The signed `withheld` choice is the evidence; the motivation is not.

The cross-platform design ensures that the signed-geo invariant is not a macOS-only feature. A server-side agent in a datacenter can still produce a valid presence attestation with signed geo — it just uses `datacenter` source with declared coordinates instead of `corelocation` with GPS. The signature verification works identically across all

platforms because it operates on the JCS canonical bytes of the `signed_body`, which includes the `geo` object regardless of source.

Summary

Presence attestation is the bridge between software-verifiable governance records and hardware-backed human verification. Its design rests on three load-bearing decisions:

1. **Sidecar, not field.** The frozen 19-field receipt body stays frozen. Presence evidence links to it by hash challenge (`canonical_body_sha256`), preserving cross-language conformance and allowing presence to be added asynchronously without blocking the gateway's hot path.
2. **Signed geo.** When geo is present, it is inside the signed body. `signature_verified=true` means the signature covered the coordinates. Tampered coordinates break the canonical hash and fail verification. Withheld geo is signed too — the choice not to share location is itself attested.
3. **Hardware abstraction via format + digest.** `evidence_format` and `evidence_digest` let any hardware provider (YubiKey PIV, TPM, Secure Enclave, Entra CNG, edge broker) plug into the same envelope without schema inflation. The raw assertion is stored out-of-band; the envelope carries only the format tag and the hash.

The `FleetNode.v1` aggregation row denormalizes presence evidence for three consumers (fleet map, ContextOS wallet, governance views) while maintaining an explicit consumer trust boundary: the FleetNode row is unsigned convenience output; the backing attestation is the trust anchor. Freshness rules ensure that a reachable node must carry a recent attestation, and edges stay separate (future `FleetEdge.v1`) to prevent the fleet map from becoming a graph database.

The end-to-end chain of custody — governance receipt → hash challenge → presence attestation → signature verification → evidence archive — provides cryptographic proof that a specific human, verified by a specific hardware key, approved a specific agent action. That proof is the platform's answer to the hardest question in AI governance: not *was this allowed?*, but *who said so, and can you prove it?*

Chapter 9: GraphRAG Ingestion

The Orca platform captures agent work as it happens — every prompt, tool call, file edit, commit reference, and reasoning trace flows through the agent, into the sink, and into durable SQLite storage. But capture is not comprehension. A sink with three million events is a vast archive of what happened; it is not, by itself, a system that can answer "what do I know about this file?" or "has anyone solved this build error before?" That gap is what GraphRAG ingestion closes.

This chapter covers the live retrieval lane: Neo4j and Qdrant as sidecars to the sink, code understanding via the `/v1/graph/context` API, the session-rules hook that injects real-time context into new sessions, the Library artifact linkage patterns that bind GraphRAG to the Operational Memory Library, the feedback-loop safety gate that prevents the self-improving mechanism from amplifying errors, the strict rule that raw and bronze never enter a queryable graph, and the real-time comprehension layer on the 2027 roadmap.

GraphRAG: The Live Retrieval Lane

The Orca Data Foundry build spec separates two mechanisms for using captured data, and the distinction is foundational:

	RETRIEVAL LANE (GRAPHRAG)	TRAINING LANE (FINE-TUNE)
Mechanism	Agent queries the graph at inference time	Gold data trains LoRA adapters
Effect	Agents become *informed*	Agents become *skilled*
Cost	Cheap, no GPU, ships now	Slow, GPU, ships later
Ship order	First	Later, optional

The retrieval lane is the product. An agent that can query a graph of past sessions, file touches, commit references, and tool usage patterns is an agent that knows what the team knows. It can find the runbook for a recurring build failure, identify which files are related to a given module, and surface dead ends that previous sessions already explored. This is cheap — it requires a graph database and a query API, not a GPU farm — and it ships first.

The training lane is optional downstream. If fine-tuning is scoped, gold data trains a family of 13B LoRA specialists. But the build spec is explicit: a model is a perishable snapshot; the data engine compounds. GraphRAG is the compounding mechanism.

Only records with `license.policy` in `{internal_training_allowed, internal_retrieval_only}`, outcome-positive, deduped, and quality-scored feed the graph. This is Pattern 6 — *Private-by-Default Scoping with Explicit Promotion* — applied to the retrieval lane. The graph is not a dump of everything that happened; it is a curated index of what is safe and useful to retrieve.

Store Architecture

GraphRAG runs as two sidecars to the sink in the same Docker Compose stack. The reference deployment in `deploy/docker-compose.yml` on `.114` runs four services: `sink`, `web`, `neo4j`, and `qdrant`.

```
  neo4j:
    image: neo4j:5.15-community
    restart: unless-stopped
    ports:
      - "${AGENTCHRON_NEO4J_HTTP_PORT:-9476}:7474"
      - "${AGENTCHRON_NEO4J_BOLT_PORT:-9687}:7687"
    environment:
      NEO4J_AUTH: "neo4j/${NEO4J_PASSWORD}"
      NEO4J_PLUGINS: '["apoc"]'
      NEO4J_dbms_memory_heap_initial__size: "512m"
      NEO4J_dbms_memory_heap_max__size: "1G"
      NEO4J_dbms_memory_pagecache_size: "512m"
    volumes:
      - neo4j_data:/data
      - neo4j_logs:/logs
    healthcheck:
      test: ["CMD-SHELL", "wget -q --spider http://localhost:7474 || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 10
      start_period: 30s
    networks:
      - agentchron

  qdrant:
    image: qdrant/qdrant:v1.9.0
    restart: unless-stopped
    ports:
      - "${AGENTCHRON_QDRANT_HTTP_PORT:-9333}:6333"
      - "${AGENTCHRON_QDRANT_GRPC_PORT:-9334}:6334"
    volumes:
      - qdrant_data:/qdrant/storage
    ulimits:
      nofile:
        soft: 65536
        hard: 65536
    networks:
      - agentchron
```

Neo4j 5.15 community edition with APOC provides the graph database. Qdrant v1.9.0 provides the vector database (though, as we will see, the vector writer is currently a stub). Both run on a bridge network with the sink, and the sink is wired to both via environment variables:

```
      NEO4J_URI: "bolt://neo4j:7687"
      NEO4J_USER: "neo4j"
      NEO4J_PASSWORD: "${NEO4J_PASSWORD}"
      QDRANT_URL: "http://qdrant:6334"
```

The port scheme is deliberately offset to avoid collision with a pre-existing graphrag stack on the same host:

COMPONENT	COMPOSE PORT	LAB GRAPHRAG PORT
Sink HTTP	9474	—
Web UI	9475	—
TCP push	39478 → 9478	—
Neo4j HTTP	9476 → 7474	7475
Neo4j Bolt	9687 → 7687	7687
Qdrant HTTP	9333 → 6333	6333
Qdrant gRPC	9334 → 6334	6334

This port-offset coexistence pattern lets the agentchron stack run alongside the existing graphrag stack on `.114` without rearchitecting either. The internal Docker network uses the standard ports (`bolt://neo4j:7687`, `http://qdrant:6334`); only the host-exposed ports are offset.

The durability hierarchy is explicit and follows Pattern 5 — *SQLite WAL as Source of Truth with Best-Effort Sidecars*. SQLite is the durable source of truth for event storage. Neo4j and Qdrant are sidecars whose failures are logged but do not block ingest. The sink's `ingest_one()` pipeline runs plugins, sanitizes, inserts into SQLite (must succeed), then attempts Neo4j and Qdrant writes (best-effort). If Neo4j is down, events are still stored and searchable via SQLite FTS5; the graph is stale but the archive is intact.

Code Understanding via `/v1/graph/context`

The sink exposes `/v1/graph/context` as the primary API for code understanding queries. The MCP tool `agentchron_graph_context` wraps this endpoint, making it available to agents through the Model Context Protocol.

The API supports two modes:

Seed-based mode. Given a `session_id` or `event_id`, the API finds all entities (files, tools, branches, agents, commits) touched by that session or event, then finds *other* sessions that share those entities. This answers: "what else has touched the same files, branches, or commits as this session?"

Entity-query mode. Given a free-text `query` string, the API matches against entity names (file paths, tool names, branch names, agent names, commit SHAs) and finds sessions that touched matching entities. This answers: "what do I know about `src/main.rs`?" or "who has worked on the `feat/orca-guard` branch?"

The Neo4j writer in `crates/agentchron-sink/src/graph.rs` implements both modes. Let's look at the graph schema first. The module header documents the edge model:

```
//! Neo4j writer. One node per event, edges:
//!   (:Session)-[:HAS]->(:Event)
//!   (:Event)-[:PARENT]->(:Event)   (when parent_uuid present)
//!   (:Event)-[:USED]->(:Tool {name})
//!   (:Event)-[:BY_AGENT]->(:Agent {name})
//!   (:Event)-[:ON_BRANCH]->(:Branch {name})
//!   (:Event)-[:TOUCHED]->(:File {path})
//!   (:Event)-[:REFERENCES_COMMIT]->(:Commit {sha})
```

Every event becomes a node. Sessions, tools, agents, branches, files, and commits are all first-class nodes with typed edges to events. This is a graph of *what happened*, not a graph of *code structure* — there are no AST nodes or function definitions. The graph captures operational provenance: which sessions touched which files, which tools were used, which commits were referenced.

ENTITY EXTRACTION

The graph writer extracts entities from events in two ways: file paths from tool inputs, and commit SHAs from event text.

File path extraction is in the `extract_file_paths` function, which recursively walks tool input JSON looking for keys named `file_path`, `notebook_path`, `path`, `source_path`, `target_path`, or any key ending in `_path`:

```
fn collect_file_paths(
    value: &Value,
    key_hint: Option<&str>,
    seen: &mut HashSet<String>,
    out: &mut Vec<String>,
) {
    if out.len() >= 25 {
        return;
    }
    match value {
        Value::String(raw) => {
            let should_check = key_hint
                .map(|key| {
                    matches!(key,
                        "file_path" | "notebook_path" | "path" | "source_path" |
"target_path"
                    ) || key.ends_with("_path")
                })
                .unwrap_or(false);
            if should_check {
                if let Some(path) = normalize_file_path(raw) {
                    if seen.insert(path.clone()) {
                        out.push(path);
                    }
                }
            }
        }
        Value::Array(items) => {
            for item in items {
                collect_file_paths(item, key_hint, seen, out);
                if out.len() >= 25 { break; }
            }
        }
        Value::Object(map) => {
            for (key, item) in map {
                collect_file_paths(item, Some(key.as_str()), seen, out);
                if out.len() >= 25 { break; }
            }
        }
        _ => {}
    }
}
```

The `normalize_file_path` function filters aggressively. It rejects URLs, strings with newlines, paths shorter than 3 or longer than 500 characters, and requires either a known filename (`Cargo.toml`, `package.json`, `README.md`) or a path containing a slash/backslash with a known source extension (`.rs`, `.py`, `.ts`, `.jsonl`, `.yaml`, etc.). This prevents the graph from filling with noise — random strings that happen to be in tool inputs do not become file nodes.

Commit SHA extraction scans event text and tool inputs for hex strings of 7-40 characters that contain at least one hex letter (a-f):

```
fn push_commit_ref(raw: &str, seen: &mut HashSet<String>, out: &mut Vec<String>) {
    if !(7..=40).contains(&raw.len()) {
        return;
    }
    if !raw.chars().any(|c| matches!(c, 'a'..='f' | 'A'..='F')) {
        return;
    }
    let sha = raw.to_ascii_lowercase();
    if seen.insert(sha.clone()) {
        out.push(sha);
    }
}
```

The "must contain a hex letter" rule is a deliberate false-positive filter. Pure numeric strings like `1234567890` are excluded because they are almost certainly not commit SHAs — they are timestamps, line numbers, or IDs. A real commit SHA almost always contains at least one letter in the `a-f` range. The test `extracts_commit_refs_without_numeric_false_positives` verifies this: the text `"merged 1117f22c971b and ignored 1234567890"` extracts only `1117f22c971b`, not the numeric string.

WEIGHTED SCORING

When finding related sessions, the graph uses a weighted scoring system that reflects how strongly an entity connects sessions:

```
fn graph_entity_weight(kind: &str, count: usize) -> f64 {
    let base = match kind {
        "file" | "commit" => 4.0,
        "git_branch" => 3.0,
        "tool" => 2.0,
        "agent" => 0.75,
        _ => 1.0,
    };
    round_score(base * count.max(1) as f64)
}
```

Files and commits carry the highest weight (4.0) because two sessions touching the same file or referencing the same commit are strongly related — they are working on the same code. Branches (3.0) are next: working on the same branch means working toward the same merge target. Tools (2.0) are weaker: many sessions use `Bash` or `Read`, so tool sharing is less discriminative. Agents (0.75) are the weakest: the same agent running multiple sessions is normal, not a strong signal of related work.

Common tools — `Bash`, `Read`, `Write`, `Edit`, `MultiEdit`, `Grep`, `Glob`, `LS`, `TodoWrite`, `Task` — are explicitly excluded from entity queries via the `is_common_tool` function:

```
fn is_common_tool(value: &str) -> bool {
    matches!(value,
        "Bash" | "Read" | "Write" | "Edit" | "MultiEdit"
        | "Grep" | "Glob" | "LS" | "TodoWrite" | "Task"
    )
}
```

This is noise reduction. A query for "Bash" would match nearly every session in the graph. The common-tool filter ensures that entity queries return discriminative results — the tools that are specific to a workflow (e.g., `agentchron_graph_context`, `docker_compose_up`) rather than the tools every session uses.

The Cypher query for seed-session entities shows this filtering in action:

```
MATCH (:Session {session_id: $session_id})-[:HAS]->(seed:Event)-[]->(entity)
WHERE entity:Tool OR entity:File OR entity:Branch OR entity:Agent OR entity:Commit
WITH
  CASE
    WHEN entity:Tool THEN 'tool'
    WHEN entity:File THEN 'file'
    WHEN entity:Branch THEN 'git_branch'
    WHEN entity:Agent THEN 'agent'
    WHEN entity:Commit THEN 'commit'
    ELSE 'unknown'
  END AS kind,
  ...
  count(DISTINCT seed) AS seed_event_count
WHERE value IS NOT NULL AND value <> ''
  AND NOT (kind = 'tool' AND value IN
['Bash','Read','Write','Edit','MultiEdit','Grep','Glob','LS','TodoWrite','Task'])
RETURN kind, value, seed_event_count
ORDER BY seed_event_count DESC, kind ASC, value ASC
LIMIT $limit
```

The result is a list of entities ranked by how many seed events touched them, filtered to exclude common tools, and limited to a configurable maximum. These entities become the bridge to find related sessions.

SQLITE FALLBACK

Pattern 5 mandates that the platform degrade gracefully when sidecars are unavailable. The `graph_context()` API tries Neo4j first and falls back to a SQLite-based entity extraction and related-session scoring algorithm when Neo4j is down. The fallback is not as rich — it cannot do graph traversal — but it can still find sessions that touched the same files or referenced the same commits by querying SQLite directly.

This means the `/v1/graph/context` endpoint is always available, even during a Neo4j outage. The quality of results degrades, but the API does not fail. An agent that calls `agentchron_graph_context` during a Neo4j restart gets SQLite-backed results, not an error.

Session Rules Hook: Real-Time Context Injection

The session-rules hook is where GraphRAG meets the agent's live session. The script at `scripts/orca-session-rules.py` runs on Claude Code's `SessionStart` and `UserPromptSubmit` hook events. Its job is to derive safe query terms from the hook payload, call the sink's API, and inject a compact, evidence-cited context block into the session.

The script's design follows Pattern 11 — *Synchronous Enforcement, Asynchronous Telemetry* — in an important way: **it fails open**. If the graph is slow or unreachable, the hook does not block the agent's work. The principle is that graph latency must never block agent work.

Let's look at the key parts of the script. It loads configuration from an environment file, constructs a request to the sink's workflow API, and renders approved workflows as markdown:

```
def main() -> int:
    parser = argparse.ArgumentParser()
    parser.add_argument("--env-file", default=os.environ.get("AGENTCHRON_ENV_FILE",
DEFAULT_ENV_FILE))
    parser.add_argument("--sink-url", default=os.environ.get("AGENTCHRON_SINK_URL", ""))
    parser.add_argument("--token", default=os.environ.get("AGENTCHRON_SINK_TOKEN")
                        or os.environ.get("AGENTCHRON_INGEST_TOKEN", ""))
    parser.add_argument("--team", default=os.environ.get("ORCA_TEAM", ""))
    parser.add_argument("--workspace", default=os.environ.get("ORCA_WORKSPACE", ""))
    parser.add_argument("--visibility", default=os.environ.get("ORCA_VISIBILITY", ""))
    parser.add_argument("--purpose", default=os.environ.get("ORCA_PURPOSE", ""))
    parser.add_argument("--limit", type=int,
default=int(os.environ.get("ORCA_SESSION_RULE_LIMIT", "12")))
    parser.add_argument("--max-chars", type=int,
default=int(os.environ.get("ORCA_SESSION_RULE_MAX_CHARS", "8000")))
    parser.add_argument("--format", choices=("markdown", "json"),
                        default=os.environ.get("ORCA_SESSION_RULE_FORMAT", "markdown"))
    args = parser.parse_args()

    load_env_file(args.env_file)
    ...
    try:
        data = get_json(sink_url.rstrip("/") + "/v1/workflows", token, params)
    except Exception as exc:
        print(f"orca-session-rules: unable to load approved workflows: {exc}",
file=sys.stderr)
        return 0  # fail open — exit 0, no output
```

The `except` block is the fail-open behavior. Any exception — network timeout, DNS failure, HTTP error, JSON parse error — results in `return 0` with a stderr warning. The agent's session proceeds without injected context. This is deliberate: a missing context block is a degraded experience; a blocked session is a broken one.

The rendered markdown is intentionally compact and evidence-cited:

```
def render_markdown(rules: list[dict[str, Any]]) -> str:
    lines = [
        "## Orca Approved Workflow Rules",
        "",
        "Apply these approved workflow rules during this session. If a rule conflicts with
the task, pause and explain the conflict before continuing.",
        "",
    ]
    for index, rule in enumerate(rules, start=1):
        title = clean_text(rule.get("title")) or f"Workflow {rule.get('workflow_id') or
index}"
        risk = clean_text(rule.get("risk")) or "medium"
        stance = clean_text(rule.get("stance")) or "warn"
        recommendation = clean_text(rule.get("recommendation")) or "No recommendation
stored."
        evidence = rule.get("evidence_event_ids") or []
        evidence_text = ", ".join(str(item) for item in evidence[:8]) if
isinstance(evidence, list) else ""
        lines.append(f"{index}. {title} [{stance}/{risk}]")
        lines.append(f"   Recommendation: {recommendation}")
        if evidence_text:
            lines.append(f"   Evidence events: {evidence_text}")
        workflow_id = clean_text(rule.get("workflow_id"))
        if workflow_id:
            lines.append(f"   Workflow ID: {workflow_id}")
        lines.append("")
    return "\n".join(lines).rstrip() + "\n"
```

Every rule carries its `workflow_id`, its `stance` and `risk` level, a concrete `recommendation`, and — critically — its `evidence_event_ids`. This is Pattern 10 — *Hash-Only Provenance* — applied to context injection. The rule does not embed the raw session text that produced it; it cites the evidence event IDs. An agent that wants to understand *why* a rule exists can query those event IDs through the sink API. The rule itself is compact, transferable, and auditable.

The graph call is capped at 2 seconds, and timeouts are negative-cached for 5 minutes. This means that if the graph is slow on one `UserPromptSubmit` event, the next several events within 5 minutes skip the graph call entirely rather than re-adding 2 seconds of latency to each prompt. The negative cache is per-host, so a graph that is slow for one agent does not penalize agents on hosts with healthy graph connections.

Library Artifact Linkage Patterns

The Operational Memory Library is the source of truth for durable operational knowledge — runbooks, troubleshooting guides, smoke proofs, incidents, deployments, and decisions. GraphRAG is an index over that knowledge, not a replacement for it. The relationship is defined by four linkage patterns in the Library spec:

```
(:LibraryArtifact)-[:SUMMARIZES]->(:Session)
(:LibraryArtifact)-[:BY_AGENT]->(:Agent)
(:LibraryArtifact)-[:FROM_HOST]->(:Host)
(:LibraryArtifact)-[:CITES]->(:Event)
```

A `LibraryArtifact` summarizes one or more sessions — it is the distilled, reviewed, approved version of what happened across those sessions. It is by a specific agent (the one whose work it summarizes) and from a specific host. It cites specific events — the evidence trail that backs the artifact's claims.

The rule is: **the Library is canonical; GraphRAG is an index/consumer.** If a GraphRAG answer cites a fix, it must link back to the Library artifact and the evidence events that back it. GraphRAG does not invent knowledge; it retrieves and connects knowledge that the Library has already vetted.

This separation matters for trust. A GraphRAG result that says "to fix this build error, run `docker compose build --no-cache`" is a retrieval hit. A Library artifact that says the same thing, with `evidence_event_ids` pointing to the session where the fix was proven, and a `review_status: approved` lifecycle state, is a trusted recommendation. The session-rules hook injects the latter, not the former — it queries `/v1/workflows?status=approved`, not the raw graph.

The Library's review lifecycle is explicit: `draft → candidate → approved → active → retired → superseded`. Only `approved` and `active` artifacts feed the session-rules hook. Retired and superseded artifacts stay in the Library for audit but do not surface to new sessions. This prevents the graph from recommending outdated solutions — if a build fix was superseded by a better approach, the old fix is marked `[SUPERSEDED]` and the new one is the live truth.

Dead ends — approaches that were tried and failed — are high-value and must not be silently discarded. The context block API (covered in Chapter 11) marks dead ends `[SUPERSEDED]` or `[ABANDONED]` rather than deleting them. A new session that is about to try the same failed approach should see the dead end and choose a different path. This is the operational memory value proposition: Orca keeps the runbook, the troubleshooting path, the smoke proof, and the evidence trail, then surfaces it to the next agent before the same mistake is repeated.

The Feedback-Loop Safety Gate

GraphRAG creates a feedback loop: agents read the graph to get context, their sessions get captured, the captured sessions flow through the Data Foundry into silver, and silver feeds back into the graph. Without a gate, this loop amplifies errors. If an agent reads a flawed recommendation from the graph, acts on it, and the resulting session is re-captured and re-indexed, the graph now has two sessions supporting the flawed approach — the original and the copy. Over time, this causes drift and model collapse: the graph converges on whatever was most frequently retrieved, not whatever was correct.

The feedback-loop safety gate breaks this cycle with a simple rule: **only `success`-outcome, reviewed records re-enter the graph.**

The gate operates at the silver-to-graph boundary. When silver records are considered for GraphRAG ingestion, they must pass two checks:

1. **Outcome check.** The record's `labels.outcome` must be `success`. Records with `failure`, `abandoned`, `partial`, or `unknown` outcomes are excluded from graph ingestion. This prevents the graph from indexing failed approaches as if they were valid solutions.
2. **Review check.** The record must have been reviewed — either through human review, panel consensus, or automated quality+safety thresholds. Unreviewed records, even if outcome-positive, are excluded until they pass review.

This gate has a second-order benefit that is the platform's self-improving mechanism. Retrieval success-vs-failure becomes free DPO (Direct Preference Optimization) preference data. When an agent queries the graph, gets a context block, and either succeeds or fails with that context, the outcome is captured. A success produces a "chosen" example (the context block was helpful). A failure produces a "rejected" example (the context block was unhelpful or

misleading). These pairs are exactly the format that DPO training needs — and they are generated as a byproduct of normal platform operation, not as a separate annotation effort.

This is the flywheel: the graph improves because only good outcomes re-enter it, and the DPO data improves because every retrieval produces a preference signal. The two mechanisms compound: better graph → better agent outcomes → better DPO data → better specialist models (if trained) → better agent outcomes. The gate is what keeps the flywheel turning in the right direction.

Raw/Bronze Never Enter GraphRAG

The rule is absolute: **raw and bronze never feed queryable GraphRAG.** This is not a configuration option; it is a security invariant baked into the data flow.

The reason is secret-exfiltration risk. Raw JSONL transcripts may contain secrets that slipped past the realtime sanitizer — API keys pasted by users, private keys read from disk before Guard blocked the read, credential-bearing environment variables. Bronze, while normalized, still contains full text and has not yet been through the silver cleaning pipeline's two-pass secret redaction.

A queryable graph is a secret-exfiltration surface. If raw text is in the graph, an attacker who can query `/v1/graph/context` can extract secrets by crafting queries that match secret-bearing events. The graph's power — its ability to find related sessions by entity — becomes the attack vector.

Silver is the minimum tier for GraphRAG ingestion. Silver has been through two-pass secret redaction (realtime sanitizer + offline deny-list + entropy), PII filtering, license/provenance classification, dedup, and boilerplate marking. Silver records that are `license.policy=internal_retrieval_only` or `internal_training_allowed` and `outcome=success` are safe to index. Gold — the highest-signal, fully reviewed tier — is the core of the graph.

The graph writer in `crates/agentchron-sink/src/graph.rs` also reduces noise at the ingestion layer. The `extract_file_paths` function filters tool inputs aggressively (only paths with known extensions or known filenames become file nodes). The `extract_commit_refs` function filters out pure-numeric strings (which are not commit SHAs). The common-tool filter excludes `Bash`, `Read`, `Write`, `Edit`, and other ubiquitous tools from entity queries. Together, these filters ensure the graph is an index of *signal*, not a mirror of *everything*.

The Qdrant Vector Writer: Honest Stub

The Qdrant client in `crates/agentchron-sink/src/vector.rs` is honest about its status:

```
//! Qdrant writer. v0.1: collection bootstrap + payload-only writes
//! (no embeddings yet — embedding model wiring lands in v0.2). Once an
//! embedder is configured we'll push `text_embedding(evt.text)` here.

use agentchron_core::Event;
use anyhow::Result;
use qdrant_client::client::QdrantClient;
use tracing::debug;

pub struct Vector {
    _client: QdrantClient,
}

impl Vector {
    pub async fn connect(url: &str) -> Result<Self> {
        let client = QdrantClient::from_url(url).build()?;
        Ok(Self { _client: client })
    }

    pub async fn write_event(&self, evt: &Event) -> Result<()> {
        // Placeholder — embedding pipeline not wired yet.
        if evt.text.is_some() {
            debug!(
                uuid = ?evt.envelope.uuid,
                "vector write stub (embedding model not configured)"
            );
        }
        Ok(())
    }
}
```

The `_client` field is prefixed with an underscore — the Rust convention for "stored but unused." The `write_event` method logs a debug message and returns `Ok(())`. No embeddings are computed. No vectors are written. The Qdrant container is running in Docker Compose, the client connects to it, but the write path is a no-op.

This is Anti-Pattern 3 — *Qdrant Vector Writer is a Dead Stub*. The code comment is honest ("embedding model wiring lands in v0.2"), but the presence of a Qdrant container in the compose stack could mislead operators into assuming semantic search is functional. A deployment that expects "search by meaning" and gets "search by keyword + graph traversal" has a gap between expectation and reality.

The mitigation is documentation and labeling. The deployment docs and the dashboard should clearly state that semantic vector search is planned, not functional, until the v0.2 embedding wiring is complete. Until then, the Qdrant container is a running placeholder — it does not harm, but it does not help either. The graph context API's SQLite fallback and Neo4j path provide the actual retrieval capability.

The future wiring is straightforward: once an embedding model is selected, `write_event` will compute `text_embedding(evt.text)` and upsert the vector with the event's UUID as the point ID. The `/v1/graph/context` API will then be able to do hybrid retrieval — graph-based entity matching plus vector-based semantic similarity — for queries that do not match any entity name exactly.

Real-Time Comprehension Layer (2027 Roadmap)

The current GraphRAG ingestion captures *what happened* — which sessions touched which files, which tools were used, which commits were referenced. The 2027 roadmap adds a layer that captures *what the work means*: the real-time comprehension layer.

The planned layer bridges the gap between "we captured the work" and "the platform understands the work." It taps silver data (never bronze) and extracts function/class atoms — the smallest meaningful units of code understanding. A function atom might be: "function `graph_entity_weight` in `graph.rs` computes a weight for a graph entity based on its kind and count." A class atom might be: "struct `Graph` wraps a Neo4j connection and provides event writing and context query methods."

The extraction pipeline deduplicates repeated patterns. If 50 sessions all touch `graph_entity_weight`, the comprehension layer analyzes the canonical atom once and stores the structured analysis in GraphRAG. Subsequent sessions that touch the same function retrieve the pre-computed analysis rather than re-deriving it. This is the difference between "50 sessions touched this file" and "this function does X, here is the analysis, here are the sessions that proved it."

The streaming substrate for this layer is not yet chosen. ADR EMPIRE-811 will decide between NATS JetStream, Kafka, and Redpanda as the streaming backbone before any stream is built. The decision matters because the comprehension layer is a streaming pipeline — silver records flow in, atoms are extracted and analyzed, structured analysis flows out to GraphRAG — and the wrong substrate would create operational burden. NATS JetStream is the leading candidate because the platform already uses NATS for the OpenBrain wake bus, but the ADR has not been finalized.

The build order from the roadmap places this layer after the context block API (EMPIRE-798) and before the canvas lens (EMPIRE-812). The context block API is the read side — it composes context from existing graph data. The comprehension conveyor (EMPIRE-809) and atom extraction (EMPIRE-810) are the write side — they add understanding to the graph. The canvas lens is the UI side — it renders the understanding for human consumption.

Anti-Patterns Addressed

QDRANT VECTOR WRITER IS A DEAD STUB (AP3)

As covered above, the Qdrant client is connected but `write_event()` only logs a debug message. This is Anti-Pattern 3 from the synthesis. The mitigation is not to remove the stub — the container and client are correct scaffolding for v0.2 — but to label it clearly. Deployment docs, dashboard status indicators, and API responses should all reflect that semantic search is planned, not functional. An operator who reads the compose file and sees a Qdrant container should not assume vector search is working without checking the code or the docs.

TRAINING ON RAW/BRONZE (AP8)

Raw and bronze never enter GraphRAG, only reviewed silver/gold. This is Anti-Pattern 8 — *Training on Raw/Bronze or Bypassing Review Gates* — applied to the retrieval lane. The same principle that governs the training lane (gold is trainable only when every record has `license.policy=internal_training_allowed` and has passed review) governs the retrieval lane (only success-outcome, reviewed records re-enter the graph). The graph is not a training surface, but it is a retrieval surface that shapes agent behavior, and unreviewed data in the graph is just as dangerous as unreviewed data in a training set.

The feedback-loop safety gate is the enforcement mechanism. Without it, the re-capture cycle would pump raw session outcomes back into the graph, creating the drift and model-collapse risk that the gate exists to prevent. The

gate makes the fail-closed path real for GraphRAG just as quarantine makes the fail-closed path real for the Data Foundry.

Summary

GraphRAG ingestion is the platform's live retrieval lane — the mechanism that turns captured session exhaust into queryable code understanding. Its design rests on several load-bearing decisions:

Neo4j as a best-effort sidecar, not a source of truth. SQLite is the durable store; Neo4j is a graph index that degrades gracefully. The `/v1/graph/context` API tries Neo4j first and falls back to SQLite-based entity extraction. Graph latency never blocks ingest.

Entity extraction with aggressive noise filtering. File paths are extracted from tool inputs only when they have known extensions or known filenames. Commit SHAs are extracted only if they contain hex letters (filtering out numeric false positives). Common tools (`Bash` , `Read` , `Write` , etc.) are excluded from entity queries. Weighted scoring (files/commits 4.0, branches 3.0, tools 2.0, agents 0.75) reflects how strongly entities connect sessions.

Session-rules hook that fails open. The `orca-session-rules.py` hook injects approved workflow rules into new sessions, citing evidence event IDs. It fails open (exit 0, no output) on any error, caps graph calls at 2 seconds, and negative-caches timeouts for 5 minutes. Graph latency never blocks agent work.

Library is canonical, GraphRAG is an index. GraphRAG answers must link back to Library artifacts and evidence events. The Library's review lifecycle (`draft → candidate → approved → active → retired → superseded`) ensures that only vetted knowledge surfaces to new sessions. Dead ends are preserved and marked, not deleted.

Feedback-loop safety gate. Only `success` -outcome, reviewed records re-enter the graph. This prevents drift and model collapse, and it generates free DPO preference data as a byproduct of normal operation. The flywheel turns in the right direction because the gate filters out failure.

Raw/bronze never enter GraphRAG. Silver is the minimum tier for graph ingestion. The graph is a queryable surface, and raw text in a queryable graph is a secret-exfiltration risk. Two-pass secret redaction in silver is the prerequisite for graph eligibility.

The Qdrant vector writer is an honest stub — the container runs, the client connects, but no embeddings are written until v0.2. The real-time comprehension layer on the 2027 roadmap will add function/class atom extraction and deduplication, turning "we captured the work" into "the platform understands the work." Until then, the graph is an index of operational provenance — which sessions touched which files, tools, branches, and commits — and that index is already enough to make agents informed.

The Graph Context Response Shape

The `/v1/graph/context` API returns a structured response that agents can consume programmatically. Understanding the response shape is essential for building consumers that use graph context effectively.

The response (defined in `crates/agentchron-sink/src/storage.rs` and returned by the `graph_context()` method in `graph.rs`) contains:

```
{
  "query": "src/main.rs",
  "seed_event_count": 42,
  "seed_session_ids": ["s-abc123"],
  "entities": [
    {
      "id": "file:src/main.rs",
      "weight": 16.0,
      "kind": "file",
      "value": "src/main.rs",
      "seed_event_count": 4
    },
    {
      "id": "file:src/config.rs",
      "weight": 12.0,
      "kind": "file",
      "value": "src/config.rs",
      "seed_event_count": 3
    },
    {
      "id": "git_branch:feat/orca-guard",
      "weight": 9.0,
      "kind": "git_branch",
      "value": "feat/orca-guard",
      "seed_event_count": 3
    }
  ],
  "related_sessions": [
    {
      "session_id": "s-def456",
      "score": 28.25,
      "shared_entities": ["file:src/main.rs", "git_branch:feat/orca-guard"],
      "matched_event_count": 12,
      "sample_event_ids": [101, 205, 308],
      "last_seen": "2026-06-20T14:30:00Z",
      "host": "<lab-host>",
      "agent": "neo"
    }
  ],
  "links": [
    {
      "weight": 16.0,
      "entity_id": "file:src/main.rs",
      "session_id": "s-def456",
      "sample_event_ids": [101, 205, 308]
    }
  ]
}
```

The `entities` array lists the entities found in the seed session (or matching the query), ranked by weight. The `related_sessions` array lists sessions that share entities with the seed, ranked by a composite score. The `links` array provides the entity-to-session mapping with sample event IDs, allowing a consumer to trace exactly which events connect an entity to a related session.

SCORE COMPOSITION

The related session score is not a simple count. The `build_related_response` function in `graph.rs` composes it from multiple signals:

```
for row in rows {
    let weight = entity_weights
        .get(&row.entity_id)
        .copied()
        .unwrap_or(row.entity_weight);
    let entry = related.entry(row.session_id.clone())
        .or_insert_with(|| GraphRelatedAccumulator { ... });
    if entry.entity_ids.insert(row.entity_id.clone()) {
        entry.score += weight;
    }
    entry.score += row.matched_event_count as f64 * 0.25;
    ...
}
```

The score has two components:

1. **Entity weight sum.** For each unique entity shared between the seed and the related session, the entity's weight is added once. Files and commits contribute 4.0 × count, branches 3.0 × count, tools 2.0 × count, agents 0.75 × count. This rewards sessions that share high-value entities (same files, same commits).
2. **Matched event count bonus.** For each entity-to-session match, `matched_event_count * 0.25` is added. This rewards sessions that have many events touching the shared entities — a session with 20 events touching `src/main.rs` scores higher than one with 2 events.

The combination ensures that a session sharing one file with many events does not outrank a session sharing three files with fewer events each. The entity weight sum is the primary signal; the event count bonus is the tiebreaker.

A WORKED EXAMPLE

Consider a developer who opens a new Claude Code session and asks about a build failure in `src/main.rs`. The session-rules hook fires on `SessionStart` and calls `/v1/graph/context` with `query=src/main.rs`.

The API matches `src/main.rs` as a file entity. It finds 42 events across 8 sessions that touched this file. It then finds related sessions that also touched `src/main.rs` or co-occurring entities (`src/config.rs` , the `feat/orca-guard` branch, commit `1117f22c971b`).

The response includes: - `seed_event_count: 42` — 42 events touched `src/main.rs` - `entities` — the file itself plus co-occurring files, branches, and commits - `related_sessions` — other sessions that touched the same entities, ranked by score - `links` — which entities connect to which sessions, with sample event IDs

The session-rules hook renders this as a compact markdown block and injects it into the session context. The agent now knows: "42 events across 8 sessions touched `src/main.rs` . The most related session is `s-def456` (score 28.25), which also touched `src/config.rs` and the `feat/orca-guard` branch. Sample events: 101, 205, 308."

The agent can then query those sample event IDs through the sink API to read the actual event text and understand what previous sessions did with this file. This is the retrieval lane in action: the graph provides the *index*, the sink provides the *content*, and the agent becomes *informed*.

Batch Writing for Backfill

The graph writer has two paths: `write_event` for single events (used during live ingest) and `write_event_batch` for bulk writes (used during backfill from the raw archive). The batch path is critical for performance when indexing millions of historical events.

```
pub async fn write_event_batch(&self, events: &[(i64, Event)]) -> Result<usize> {
    let mut event_rows = Vec::new();
    let mut parent_rows = Vec::new();
    let mut agent_rows = Vec::new();
    let mut branch_rows = Vec::new();
    let mut tool_rows = Vec::new();
    let mut file_rows = Vec::new();
    let mut commit_rows = Vec::new();

    for (event_id, evt) in events {
        // ... collect into row vectors ...
    }

    let written = event_rows.len();
    self.run_rows(EVENT_BATCH_CYPHER, "events", event_rows).await?;
    self.run_rows(PARENT_BATCH_CYPHER, "parents", parent_rows).await?;
    self.run_rows(AGENT_BATCH_CYPHER, "agents", agent_rows).await?;
    self.run_rows(BRANCH_BATCH_CYPHER, "branches", branch_rows).await?;
    self.run_rows(TOOL_BATCH_CYPHER, "tools", tool_rows).await?;
    self.run_rows(FILE_BATCH_CYPHER, "files", file_rows).await?;
    self.run_rows(COMMIT_BATCH_CYPHER, "commits", commit_rows).await?;

    Ok(written)
}
```

The batch path collects all rows in memory, then issues one `UNWIND` Cypher query per edge type. The `EVENT_BATCH_CYPHER` creates all session and event nodes in one query; `PARENT_BATCH_CYPHER` creates all parent-child edges; `AGENT_BATCH_CYPHER` creates all agent edges; and so on. This is dramatically faster than the single-event path, which issues separate queries per event per edge type.

The batch Cypher for events uses `UNWIND` with `MERGE`:

```
UNWIND $events AS row
MERGE (s:Session {session_id: row.session_id})
  ON CREATE SET s.first_seen = timestamp()
  SET s.last_seen = timestamp()
MERGE (e:Event {uuid: row.uuid})
  SET e.event_id     = row.event_id,
      e.kind         = row.kind,
      e.timestamp    = row.timestamp,
      e.cwd          = row.cwd,
      e.git_branch   = row.git_branch,
      e.host         = row.host,
      e.source_path  = row.source_path,
      e.text_summary = row.text_summary
MERGE (s)-[:HAS]->(e)
```

`MERGE` is idempotent — re-running the batch on already-indexed events does not create duplicates. This makes backfill safe to re-run after a failure. The `ON CREATE SET` ensures `first_seen` is only set once; subsequent runs update `last_seen`.

The MCP Tool: agentchron_graph_context

The `agentchron-mcp.py` MCP server (in `deploy/bin/agentchron-mcp.py`) exposes the graph context API as the `agentchron_graph_context` tool. This is how agents query the graph through the Model Context Protocol.

The MCP server is a dependency-free stdio server — it has no external Python dependencies beyond the standard library. It communicates with the sink over HTTP using the same bearer-token authentication as the web UI. The recommended deployment keeps the MCP server on `.114` (where the sink and token live) and lets agents on other hosts connect through SSH, so secrets stay on the trusted host.

The tool accepts parameters matching the `/v1/graph/context` API: - `query` — free-text entity query - `session_id` — seed session ID - `event_id` — seed event ID - `host`, `agent`, `source` — scope filters - `limit` — max related sessions (default 10, max 50) - `max_seed_events` — max entities to extract (default 20, max 50)

The tool returns the JSON response from the sink API directly to the agent. The agent's model then interprets the entities, related sessions, and links to decide what context is relevant and whether to query sample event IDs for more detail.

This two-step pattern — graph context for the index, event query for the content — is deliberate. The graph context response is compact (entities, session IDs, scores, sample event IDs) and fits easily in an agent's context window. The full event text is fetched on demand only for the events the agent decides are relevant, avoiding context window pollution from irrelevant session details.

The Three-Consumer Pattern

The GraphRAG system is designed to serve three distinct consumers, each with different needs:

Command Center Fleet Map. Needs to show agent positions and status on a geographic map. Uses FleetNode.v1 (Chapter 8) for the fleet view and graph context for "what is this agent working on?" queries. The fleet map is a UI consumer — it renders nodes and relationships but does not make trust decisions.

ContextOS Wallet. Needs to know which agents have active, verified human presence and what they are authorized to do. Uses presence attestations for trust decisions and graph context for operational context. The wallet is a trust consumer — it re-verifies backing attestations before routing sensitive work.

Governance Audit Views. Needs to show the full chain of custody for sensitive actions: who approved what, when, where, and with what evidence. Uses governance receipts, presence attestations, and graph context to reconstruct the decision chain. The audit view is a compliance consumer — it requires the full signed chain, not just the denormalized view.

The three-consumer pattern prevents fleet shape drift. If the fleet map shows an agent in Austin but the wallet shows the same agent in Seattle, the cause is a consumer bug, not a data divergence — both should be reading from the same FleetNode.v1 row, which was populated from the same presence attestation. The single-emission, three-consumer design ensures that all three views see the same data.

The Real-Time Comprehension Layer: Design Preview

The 2027 roadmap's real-time comprehension layer is the next evolution of GraphRAG ingestion. While the current system captures *what happened* (sessions, events, entities), the comprehension layer captures *what it means* (function atoms, class atoms, pattern deduplication).

The planned pipeline:

1. **Tap silver (never bronze).** The comprehension layer reads from silver, which has been through two-pass secret redaction, PII filtering, and license classification. Bronze is never tapped because it may contain unredacted secrets.
2. **Extract function/class atoms.** An atom extractor parses tool inputs and event text to identify function and class definitions, signatures, and usage patterns. An atom is the smallest meaningful unit of code understanding: "function `graph_entity_weight(kind, count)` returns a weight based on entity kind and count."
3. **Deduplicate repeated patterns.** If 50 sessions all touch `graph_entity_weight`, the comprehension layer analyzes the canonical atom once. The analysis is stored in GraphRAG as a structured node (`(:FunctionAtom {name, signature, analysis})`) with edges to the sessions and events that referenced it.
4. **Store structured analysis.** The analysis includes: what the function does, what it is used for, known failure modes, related functions, and evidence event IDs. This is pre-computed understanding, not raw retrieval.
5. **Streaming substrate.** The pipeline is a stream — silver records flow in, atoms are extracted and analyzed, structured analysis flows out to GraphRAG. ADR EMPIRE-811 decides between NATS JetStream, Kafka, and Redpanda as the streaming backbone.

The comprehension layer bridges the gap between "we captured the work" and "the platform understands the work." A graph context query for `graph_entity_weight` in 2027 would return not just "42 events touched this function" but "this function computes entity weights for graph context scoring, here is the analysis, here are the sessions that proved it, here are the known failure modes." The agent gets understanding, not just provenance.

The build order places this layer after the context block API (EMPIRE-798) and atom extraction (EMPIRE-810), and before the canvas lens (EMPIRE-812). The context block API is the read side for current graph data; the comprehension conveyor (EMPIRE-809) is the write side that adds understanding; the canvas lens is the UI that renders understanding for humans. The three together form the comprehension stack: extract, store, render.

Chapter 10: Data Foundry

The Orca platform captures millions of agent events — prompts, tool calls, file edits, reasoning traces, guardrail decisions — across dozens of hosts. That raw exhaust is valuable, but it is not training data. It is not retrieval data. It is not a dataset product. It is the raw material from which those things are refined. The refinery that does the refining is the Orca Data Foundry.

This chapter covers the foundry's medallion pipeline in full: the raw archive and its WORM discipline, bronze normalization preserving full provenance, silver's seven-step cleaning pipeline with fail-closed governance, gold promotion with review gates and DSSE receipts, quarantine as the fail-closed safety net, manifests and the reproducibility envelope, and the dataset products (CPT, SFT, DPO, Workflow Library) that are the foundry's output. Throughout, we reference the actual foundry output observed on `.114` at `/mnt/backups05/orca-data-foundry/`, the build spec at `docs/ORCA_DATA_FOUNDRY_SPEC.md`, and the run manifests that prove the pipeline has moved past spec into running infrastructure.

Thesis: The Data Engine Is the Moat

The build spec makes the platform's strongest strategic claim in its first paragraph:

> The raw archive is valuable, but it is not training data. The Data Foundry is a promotion pipeline: `raw archive → bronze normalized → silver cleaned → gold curated`. Each promotion step is deterministic, versioned, auditable, and reversible.

This is not a data pipeline description — it is a thesis about durable competitive advantage. A model is a perishable snapshot. Every base-model cycle — Claude 4.7 to 5, GPT-5 to 6, Llama 4 to 5 — invalidates fine-tuned adapters and forces retraining. The model decays; the data engine compounds. Each session captured, each silver record cleaned, each gold artifact reviewed and promoted is a permanent asset that survives base-model cycles. The capture pipeline, the renewable corpus, the eval suites, and the retrieval graph are the durable IP. Fine-tuning is an optional downstream, never the goal.

The medallion promotion is the mechanism:

```
raw archive → bronze normalized → silver cleaned → gold curated
```

Each step is: - **Deterministic.** The same input with the same filter versions and the same code commit produces the same output. - **Versioned.** Every output file carries a schema version, filter versions, and a git commit hash. - **Auditable.** Every step produces a manifest; every manifest is archived under `manifests/runs/`. - **Reversible.** A gold artifact can be demoted; a silver record can be re-cleaned with new filters; a raw file is never mutated so re-processing is always possible.

Raw is treated as WORM — write-once-read-many. The raw archive is the immutable foundation; everything above it is derived and re-derivable.

Raw Archive

The primary raw archive lives at `/mnt/backups05/agentchron-raw` on `.114`. At spec time it was 33 GB; at research time (June 2026) it had grown through ongoing fleet capture. The layout is host-namespaced:

```
/mnt/backups05/agentchron-raw/
  <lab-host>/<absolute/source/path>/*.jsonl
  <lab-host>/<absolute/source/path>/*.jsonl
  <host>/<absolute/source/path>/*.jsonl
```

Original JSONL files are preserved byte-for-byte. SHA-256 manifests are written per sweep. Access is restricted because raw may contain secrets — the realtime sanitizer catches most secrets at capture time, but raw is the belt-and-suspenders archive and must be treated as if it contains everything.

First-pass discovery at spec time documented the scale:

```
.114 global sweep:
  manifests: /mnt/backups05/agentchron-manifests/global-114-20260605
  roots:     54 Claude roots
  files:     13,297 JSONL files
  bytes:     15,787,138,914 bytes (~15.79 GB)

.110 Data first pass:
  roots:     9 Claude roots
  files:     444 JSONL files
  bytes:     460,746,362 bytes (~461 MB)
  slow topdirs: 17 timed-out top-level directories for follow-up
```

The active Orca sink had about 3.18M events / 190.5 billion tracked tokens at spec time, while the `.114` and `.110` backfills were still running.

ASSET PROTECTION (PHASE 0)

A critical step before any processing is replicating the raw archive to a second drive with SHA verification. On `.114`, this was actually executed:

```
/mnt/backups05/orca-data-foundry/
  asset-protection-rsync-20260605T192541Z.log
  asset-protection-rsync-backup02-20260605T192601Z.log
  asset-protection-rsync-backup02-pass2-20260605T210218Z.log
  verify-backup02-copy-20260605T204157Z.log
  verify-backup02-copy-pass2-20260605T221053Z.log
  verify-backup02-copy.sh
```

The `asset-protection-rsync` logs show the replication to the Backup02 drive. The `verify-backup02-copy` logs show the SHA verification pass. The `.pid` files (`asset-protection-rsync.pid`, `asset-protection-rsync-pass2.pid`) show that this was a managed, multi-pass operation — pass 1 ran the initial rsync, pass 2 caught files that changed during pass 1. This is operational discipline: the raw archive is the foundation, and the foundation gets a backup with verification before anything is built on it.

The storage snapshot from the spec shows the available drives:

HOST	MOUNT	SIZE	USED	FREE	NOTES
.114	`/mnt/backups05`	7.3T	1.7T	5.3T	Primary Orca raw archive
.114	`/home`	3.6T	1.2T	2.3T	Live host home
.114	`/run/media/developer/Backup02`	7.3T	4.4T	2.5T	Backup drive
.110	`/media/developer/Data`	11T	9.1T	1.3T	First-pass roots archived

The raw archive rules are explicit in the spec:

- Store original JSONL files byte-for-byte.
- Do not mutate, redact, or normalize in place.
- Preserve original source path under host namespace.
- Write SHA-256 manifests per sweep.
- Raw archive may contain secrets; access is restricted.

Bronze: Normalized Canonical Turns

Bronze is the first processing tier. Its schema is `orca.bronze.session_event.v1`. Its job is to parse raw Claude JSONL into canonical turns while preserving full provenance.

The record shape from the spec:

```
{
  "schema_version": "orca.bronze.session_event.v1",
  "event_id": "stable hash or sink id",
  "session_id": "source session id",
  "turn_index": 42,
  "timestamp": "2026-06-05T17:18:48.758Z",
  "host": "<lab-host>",
  "agent": "neo",
  "source_path": "/mnt/backups05/...",
  "source_sha256": "...",
  "byte_offset": 123456,
  "role": "user|assistant|tool_use|tool_result|system|reasoning|unknown",
  "text": "...",
  "tool": {
    "name": "Bash",
    "input_json": {},
    "output_text": "...",
    "exit_code": 0
  },
  "raw_kind": "assistant",
  "model": "claude-opus-4-7",
  "usage": {
    "input_tokens": 0,
    "output_tokens": 0,
    "cache_read_tokens": 0,
    "cache_creation_tokens": 0,
    "thinking_tokens": 0
  }
}
```

The provenance fields are the load-bearing part: `source_path`, `source_sha256`, and `byte_offset`. Together they pinpoint exactly where in the raw archive this record came from. Given a bronze record, you can open the raw file at `source_path`, verify its SHA-256 matches `source_sha256`, seek to `byte_offset`, and read the original JSONL line. This is Pattern 10 — *Hash-Only Provenance* — in its rawest form: bronze references raw by location + hash, never by embedding the raw content.

The observed bronze layout on `.114`:

```
bronze/
  quarantine/
  sessions/
```

Parse failures are quarantined — never written as raw unredacted lines. The quarantine subdirectory holds `parse_failures/` with its own schema (`orca.quarantine.parse_failure.v1`). The bronze run manifest on `.114` shows this in action:

```
{
  "command": "./scripts/orca-dataset-export.py bronze --raw-root /mnt/backups05/agentchron-
raw --out /mnt/backups05/orca-data-foundry/bronze --manifest-out /mnt/backups05/orca-data-
foundry/manifests/runs/bronze-20260605T204249Z.json",
  "created_at": "2026-06-05T22:09:55.040019Z",
  "files": [
    {
      "byte_size": 25248,
      "path": "/mnt/backups05/orca-data-foundry/bronze/quarantine/parse_failures/
2026-06-05T204320Z.jsonl",
      "record_count": 17,
      "schema_version": "orca.quarantine.parse_failure.v1",
      "sha256": "sha256:cb5c4c32c1a37b016cb28a7ce962d60dba1e25d3db6c423651e3e70176fd208e"
    },
    {
      "byte_size": 1550398,
      "path": "/mnt/backups05/orca-data-foundry/bronze/sessions/
2025-05-13/1fb9ff15-132d-430f-9e34-875916b5a3eb.jsonl",
      "record_count": 543,
      "schema_version": "orca.bronze.session_event.v1",
      "sha256": "sha256:9423c1c864e5767e6b1e0d335a81396d1038b49c36eee26a2e36092a593eb7b1"
    },
    ...
  ]
}
```


The manifest shows 17 parse failures quarantined and multiple session files successfully parsed. The `record_count` and `sha256` for every file are in the manifest — this is the reproducibility envelope (Pattern 12) applied at the bronze tier.

Bronze may still contain sensitive text — the redaction job has not run yet at this tier. Bronze stays in restricted storage. It is internal-only, not queryable, and never feeds GraphRAG.

Silver: The Governance Core

Silver is where governance happens. Schema `orca.silver.session_event.v1`. The cleaning pipeline runs in a load-bearing order — each step depends on the previous one being complete:

STEP 1: SECRET REDACTION (TWO PASSES)

Two independent passes: the Orca realtime sanitizer (the same 19-pattern regex engine from `crates/agentchron-core/src/sanitizer.rs` that runs at capture time) and an offline deny-list + entropy scan. The offline pass catches secrets that the realtime pass might miss — newly added token formats, internal canary secrets, high-entropy strings that look like credentials but don't match a known pattern.

The blocklist is explicit:

- API keys: Anthropic, OpenAI, GitLab, GitHub, Stripe, AWS, Cloudflare
- Webhook secrets
- OAuth client secrets

- JWTs and bearer tokens
- Infisical tokens and config
- PEM private keys and SSH private keys
- `.env` values
- PIV/YubiKey private material
- Session cookies

Replacement markers are structured, not generic:

```
[REDACTED_SECRET:ANTHROPIC_API_KEY]
[REDACTED_SECRET:AWS_ACCESS_KEY]
[REDACTED_PRIVATE_KEY]
[REDACTED_PII:EMAIL]
```

Unredactable records — those where a secret-shaped value cannot be safely replaced without losing the meaning of the record — go to quarantine. The record is excluded from silver. **Fail-closed: uncertain → quarantine.** This is the core governance principle. It is better to lose a record than to leak a secret.

STEP 2: PII FILTERING

An independent pass from secret redaction. Classifies and redacts or hashes: email addresses, phone numbers, home addresses, account references, personal names (when not needed for training), and customer identifiers.

Role labels are preserved when useful for training context:

```
[REDACTED_PERSON:FOUNDER]
[REDACTED_PERSON:ENGINEER]
[REDACTED_CUSTOMER]
```

This preserves the semantic role — "the founder said X" — while removing the actual name. A specialist model trained on silver data learns that founders make architectural decisions; it does not learn the founder's name.

STEP 3: LICENSE/PROVENANCE CLASSIFICATION

Each record gets one policy:

POLICY	MEANING
`internal_training_allowed`	Internal session, safe for training and retrieval
`internal_retrieval_only`	Internal, safe for graph retrieval but not training
`restricted`	Limited use, not for training or public retrieval
`exclude`	Do not use
`unknown`	Cannot classify — treat as exclude

Exclude by default when: source path belongs to third-party private/customer repo without explicit grant; content is a copied proprietary document; license is unknown and content is long-form third-party text; record contains unredactable secrets.

This is Pattern 6 — *Private-by-Default Scoping with Explicit Promotion* — at the record level. The default is exclusion; promotion to `internal_training_allowed` requires positive evidence that the content is safe and owned.

STEP 4: DEDUPLICATION

Many backups contain the same JSONL files under different paths. The raw archive keeps all copies (WORM). Silver collapses duplicates but retains all provenance references.

Dedup keys:

```
session_id + turn_index + content_hash
uuid + content_hash
normalized_text_hash
tool_name + normalized_input_hash + normalized_output_hash
source_sha256 (for whole files)
```

The observed silver layout on `.114` includes a `dedup.sqlite` database:

```
silver/
  dedup.sqlite
  dedup.sqlite-journal
  manifests/
  sessions/
```

The `dedup.sqlite` database tracks dedup keys across runs, preventing the same content from appearing in multiple silver versions.

STEP 5: BOILERPLATE MARKING

Mark and optionally remove: `/clear`, `/help`, command boilerplate; repeated Claude status messages; empty tool outputs; duplicate retries with no new information; local shell noise that does not affect task outcome.

Failures are labeled, never discarded. A failed build attempt is not boilerplate — it is a signal. The boilerplate filter removes noise, not evidence.

STEP 6: OUTCOME LABELING

```
success | failure | abandoned | partial | unknown
```

Signals: command exit code, test pass/fail, phrase signals ("fixed", "works", "merged", "smoke passed" vs "reverted", "rolled back", "blocked", "abandoned"), and later panel/human review.

Outcome labeling is the input to the feedback-loop safety gate (Chapter 9). Only `success`-outcome records re-enter GraphRAG. Only `success`-outcome records become DPO "chosen" examples. The outcome label is the quality signal that makes the flywheel turn.

STEP 7: DECISION MARKERS

Flag turns where the agent or human makes a load-bearing decision:

```
{
  "decision_marker": true,
  "decision_summary": "Use musl static binary instead of glibc for air-gapped deployment",
  "decision_alternatives_considered": ["glibc dynamic", "docker multi-stage", "nix build"],
  "decision_rationale": "musl produces a single static binary with no runtime dependencies;
air-gapped hosts may not have glibc 2.28+",
  "evidence_event_ids": ["evt_..."]
}
```

Decision markers are the highest-signal part of silver. They identify the moments where expertise was exercised — where a path was chosen from alternatives. These are the records that, when promoted to gold, become the most valuable training and retrieval assets.

SILVER PROVENANCE

Silver references raw **by hash only**:

```
{
  "source_ref": {
    "host": "<lab-host>",
    "source_path_hash": "sha256:...",
    "source_sha256": "sha256:...",
    "byte_offset": 123456,
    "raw_archive_ref": "restricted://agentchron-raw/..."
  }
}
```

The `raw_archive_ref` uses a `restricted://` scheme — it is a reference, not a path that can be directly opened. Access to the raw archive is a separate permission. Silver preserves traceability without exposing raw content.

OBSERVED SILVER ITERATIONS

On `.114`, multiple silver versions exist, showing iterative re-runs as filters hardened:

```
silver-fast-20260605T221424Z
silver-v2-20260605T235235Z
silver-v3-20260606T003316Z
silver-v4-20260606T032241Z
```

The `silver-fast` run was the first attempt. It was quarantined. The `silver-v2` run was the replacement. `v3` and `v4` were further iterations. The manifest directory contains leak-scan and hardened-gate reports:

```
manifests/leak-scans/
  silver-v4-...-hardened-gate.json
  silver-v4-...-hmac-gate-20260621T034158Z.json
```

The hardened gate and HMAC gate reports are the quality checks that must pass before silver is approved for downstream use. The HMAC gate requires a broker-injected signing key — the same pattern as the `ORCA_FOUNDRY_SIGNING_KEY` used in gold promotion. Silver is not just cleaned; it is *signed off*.

Gold: Curated High-Signal Data

Gold is the top of the medallion. It is curated, high-signal data promoted from silver only after one of: human review, panel consensus, automated quality+safety thresholds, or workflow-library promotion.

Gold artifacts include: - SFT conversations - Tool-use traces - Debugging/failure-recovery examples - Workflow/runbook examples - DPO preference pairs - Evaluation tasks with expected outcomes

OBSERVED GOLD OUTPUT

On `.114`, the gold output is real and verified:

```
gold/
  evals/orca-workflow-retrieval/
  index/
  sft/orca-internal/
    gold-v1.jsonl                    (4.17 MB)
    gold-v1.scrub-report.json
```

The `gold-v1.jsonl` file is 4.17 MB of curated SFT conversations. The `gold-v1.scrub-report.json` is the final secret/PII scan report that proves the gold data is clean. The gold run manifest (`gold-20260624T115957Z.json`) shows the promotion run details:

```
{
  "purpose": "graphrag-gold-ingest",
  "status": "dry-run",
  "planned_records": 1000,
  "library_status": "approved",
  "scrub_report": "/mnt/backups05/orca-data-foundry/gold/sft/orca-internal/gold-v1.scrub-
report.json",
  "gold_index": "/mnt/backups05/orca-data-foundry/gold/index/gold-index.jsonl",
  "dsse_receipt": "gold-20260624T115957Z.receipt.dsse.json"
}
```

Note on the manifest status. The manifest above shows `"status": "dry-run"`, which was the observed state of the gold promotion run at the time of writing — the pipeline had been exercised end-to-end through the promotion path, producing the scrub report and DSSE receipt, but the final status flag had not yet been flipped from `dry-run` to `promoted`. Production gold — the "real, promoted, signed-off" corpus described elsewhere in this chapter — requires the `"status"` field to be flipped to `"promoted"` after two-reviewer approval. The `dry-run` value here is a snapshot of the pipeline in its final pre-promotion state, not a contradiction of the gold-is-real claim: the data, the scrub report, and the DSSE receipt are all genuine artifacts; only the status flag remains to be flipped by the two-reviewer gate before the corpus is declared production gold. The presence of a DSSE (Dead Simple Signing Envelope) receipt — `gold-20260624T115957Z.receipt.dsse.json` — indicates that signed provenance is wired into the promotion path. DSSE is a standard for signing arbitrary artifacts with a detached signature envelope. The gold promotion path produces not just the data but a cryptographic receipt proving who promoted it, when, and with what signing key.

GOLD PROMOTION GATES

The build spec is explicit about what gold promotion requires:

- No secret/PII findings (the scrub report must be clean)
- `license.policy` allows training (`internal_training_allowed`)
- Quality score above threshold
- Evidence references intact (all `evidence_event_ids` resolve)
- Human or panel approval for high-value examples
- Restricted-license exclusion (third-party/customer content without grant is excluded)
- HMAC gate report (the HMAC gate must pass)
- Broker-injected `ORCA_FOUNDRY_SIGNING_KEY` (the signing key must come from the secret broker, not from argv or a config file)

Two-reviewer approval is the production gate. This is not a single-person sign-off. Gold promotion for production training requires two independent reviewers. The reviews directory holds per-run review records:

```
reviews/
  silver-v4-20260606T032241Z/
  ...
```

Each review record documents who reviewed, what they approved, and what evidence they considered. This is the governance gate that makes gold trustworthy. A dataset that bypasses this gate is not gold — it is silver pretending to be gold, and training on it is Anti-Pattern 8.

Quarantine: Fail-Closed by Design

Quarantine is the mechanism that makes the fail-closed path real. Without quarantine, "uncertain → exclude" is a silent drop — the record disappears and nobody knows. With quarantine, the record is preserved with a status manifest so the failure is auditable.

The observed quarantine layout on `.114` :

```
quarantine/failed-silver/
  silver-fast-20260605T221424Z/
    QUARANTINE_STATUS.json
    data/
    manifest.json
  silver-v3-20260606T003316Z.failed.json
  silver-v4-20260606T032241Z.failed.json
```

The `QUARANTINE_STATUS.json` for the `silver-fast` run tells the full story:

```
{
  "schema_version": "orca.failed_silver_quarantine.v1",
  "created_at": "2026-06-05T23:52:48.414325Z",
  "status": "quarantined_failed_candidate",
  "source_output": "/mnt/backups05/orca-data-foundry/silver-fast-20260605T221424Z",
  "quarantine_data": "/mnt/backups05/orca-data-foundry/quarantine/failed-silver/silver-
fast-20260605T221424Z/data",
  "reason": "Silver leak scan found secret-shaped material after export; not approved for
GraphRAG, training, or gold promotion.",
  "failed_gate": "secret-prefix-leak-scan",
  "detected_classes": [
    "PEM private key header",
    "AWS AKIA-style key/prefix",
    "Stripe secret key prefix"
  ],
  "approved_for_downstream": false,
  "remediation_commit": "d30df05 fix: harden Foundry silver secret filtering",
  "replacement_run": "/mnt/backups05/orca-data-foundry/silver-v2-20260605T235235Z"
}
```


This is a masterclass in operational transparency. The status manifest records:

- **What failed.** The `secret-prefix-leak-scan` gate caught secret-shaped material after export.
- **What was detected.** Three classes: PEM private key headers, AWS AKIA-style keys, Stripe secret key prefixes.
- **Why it matters.** The run is "not approved for GraphRAG, training, or gold promotion."
- **What fixed it.** The `remediation_commit` (`d30df05`) hardening the silver secret filtering.
- **What replaced it.** The `replacement_run` (`silver-v2-20260605T235235Z`) is the clean version.

The `approved_for_downstream: false` field is the hard gate. No downstream consumer — not GraphRAG, not gold promotion, not training — is allowed to read quarantined data. The quarantine is not a staging area; it is an evidence locker.

The `.failed.json` files for `silver-v3` and `silver-v4` show that even later iterations had failures. The pipeline iterates: each failure produces a quarantine record, a remediation commit, and a replacement run. The silver versions on `.114` (`silver-fast`, `v2`, `v3`, `v4`) are the visible evidence of this iteration. Each version is a snapshot of the cleaner at a point in time; each quarantine record is the audit trail of what went wrong and how it was fixed.

Manifests and Reproducibility

Every generated file carries a reproducibility envelope. The build spec calls this non-negotiable:

> Every generated file must have: - SHA-256 - byte size - record count - schema version - source corpus IDs - filter versions - generation command - git commit of exporter

Run manifests live under `manifests/runs/`. The observed manifests on `.114`:

```
manifests/runs/
  bronze-20260605T204249Z.json
  silver-v2-...
  silver-v3-...
  silver-v4-...
  gold-20260624T115957Z.json
```

The bronze manifest (shown earlier in this chapter) includes the full `command` string, the `created_at` timestamp, and per-file entries with `byte_size`, `record_count`, `schema_version`, and `sha256`. The generation command is the exact shell invocation that produced the run — a developer who has the code repository at the specified git commit and the raw archive can re-run the command and reproduce the output.

This is Pattern 12 — *Reproducibility Envelope on Every Artifact* — and it is the mechanism that makes the data engine auditable. A dataset without a manifest is a black box. A dataset with a manifest is a reproducible artifact: given the manifest, the code commit, and the source corpus, anyone can verify that the output matches. If the output does not match, either the code changed, the filters changed, or the source changed — and the manifest pinpoints which.

The multiple silver versions are a direct consequence of reproducibility. Each time the filters hardened (the `remediation_commit` in the quarantine status), a new silver run was produced with a new manifest. The old manifest is not deleted — it remains as a record of what the filters were at that point in time. This creates a versioned history of the cleaning pipeline itself.

The filter version travels inside each silver record's `redaction` block:

```
{
  "redaction": {
    "filter_version": "orca-secret-filter-v1",
    "pii_filter_version": "orca-pii-filter-v1",
    "redactions": [
      {"type": "ANTHROPIC_API_KEY", "field": "content", "replacement": "[REDACTED_SECRET]"}
    ]
  }
}
```

The `filter_version` is the same `agentchron-secrets-filter-v1` string that the realtime sanitizer stamps into findings metadata. This means a silver record carries not just the redacted text but the version of the filter that redacted it. If a filter version is later found to have a bug (it missed a secret pattern), all silver records stamped with that version can be identified and re-processed.

Dataset Products and Reviews

The foundry's output is not a single dataset — it is a family of dataset products, each serving a different downstream use.

CPT: CONTINUED PRETRAINING

Long-form sanitized engineering corpus. High-quality assistant/user text, runbooks, architecture docs, workflow extractions, code-adjacent explanations. Used for continued pretraining of a base model to improve its engineering domain knowledge.

Avoid: raw secrets, low-signal logs, copyrighted books/PDFs unless licensed for training, private customer content.

SFT: SUPERVISED FINE-TUNING

Instruction-tuned examples in standard chat format:

```
{
  "messages": [
    {"role": "user", "content": "Fix the failing Docker build"},
    {"role": "assistant", "tool_call": {"name": "Bash", "arguments": {"command": "docker
build ..."}}},
    {"role": "tool", "name": "Bash", "content": "sanitized output"},
    {"role": "assistant", "content": "Diagnosis and next step"}
  ],
  "metadata": {
    "outcome": "success",
    "domain": ["docker", "build-debugging"]
  }
}
```

The observed `gold-v1.jsonl` on `.114` is an SFT pack. The metadata carries `outcome` and `domain` tags that allow downstream training to select subsets by domain (e.g., only `build-debugging` examples for the Build Engineer specialist).

DPO: DIRECT PREFERENCE OPTIMIZATION

Pairs built from: - Successful vs failed approach - Accepted panel recommendation vs rejected path - Secure workflow vs unsafe workflow - Concise runbook vs noisy raw transcript

```
{
  "prompt": "...",
  "chosen": "...",
  "rejected": "...",
  "metadata": {
    "reason": "chosen path passed smoke; rejected path used unsafe supply chain",
    "evidence_event_ids": ["..."]
  }
}
```

DPO data is the flywheel output. Every retrieval that produces a success-vs-failure outcome is a free DPO pair. The feedback-loop safety gate (Chapter 9) ensures that only `success`-outcome, reviewed records become "chosen" examples — the "rejected" examples are the failures and abandoned approaches. This is preference data generated as a byproduct of normal platform operation.

WORKFLOW LIBRARY

Every project/app build can produce: - "How it was built" runbook - Troubleshooting guide - Source/session references - Decision timeline - Verification commands - Known failure modes

Workflow candidates are extracted automatically from silver and promoted to gold after review. The Workflow Library is the operational memory that feeds back into agent sessions through the session-rules hook (Chapter 9).

REVIEWS

The reviews directory holds per-run review records:

```
reviews/
  silver-v4-20260606T032241Z/
  gold-.../
```

Gold promotion requires reviewer signoffs, HMAC gate reports, restricted-license exclusion, and a broker-injected `ORCA_FOUNDRY_SIGNING_KEY`. Two-reviewer approval is the production gate. The review records document: - Who reviewed (reviewer identity) - What they approved (artifact IDs, record counts) - What evidence they considered (scrub reports, leak scans, quality scores) - When they approved (timestamps) - The signing key used (key ID, not the key itself)

SPECIALIST MODEL STRATEGY (OPTIONAL DOWNSTREAM)

If fine-tuning is scoped, the build spec says: do not train one generic "everything" model. Build a family of 13B LoRA specialists and route through Orca/GraphRAG:

MODEL	FOCUS
Build Engineer 13B	Docker, build systems, package managers, CI failures
Rust Infra 13B	Rust services, async, SQLite, CLI tools
Security Reviewer 13B	Secret leaks, supply chain, auth bypasses
DevOps/K8s 13B	GKE, Cloudflare, Terraform, deployment runbooks
AgentOS Architect 13B	ContextOS, AgentOSFactory, Durable Rooms, A2A/ATCS
Code Archaeologist 13B	Search old sessions, recover context, explain provenance
Workflow Writer 13B	Turn sessions into runbooks, QA/UAT, handoff docs

Training order: build high-quality silver corpus → train small LoRA adapters on a base 13B → evaluate against held-out Orca sessions and real repo tasks → promote best adapters to a routing layer → add DPO from panel/human preferences.

This is explicitly off the critical path. The data engine is the moat; the specialist models are proof that the moat produces value. If a base-model cycle invalidates the adapters, the data engine is still there, ready to train the next generation.

Anti-Patterns Addressed

TRAINING ON RAW/BRONZE OR BYPASSING REVIEW GATES (AP8)

Gold is trainable only when every record has `license.policy=internal_training_allowed` and has passed human/panel review + HMAC gate + signing key. The build spec is explicit:

> Never train on secrets, private keys, raw `.env`, or unreviewed customer data.
>
> Gold data promotion is governance-gated; do not bypass reviewer/signing gates to meet a date.

Quarantine exists to make the fail-closed path real. The `silver-fast` quarantine on `.114` is the proof: a silver run that leaked secret-shaped material was caught, quarantined, and replaced. It was not "close enough" to ship. It was not silently fixed. It was preserved as an audit record and explicitly marked `approved_for_downstream: false`.

The remediation commit (`d30df05 fix: harden Foundry silver secret filtering`) is the code change that fixed the filter. The replacement run (`silver-v2`) is the clean output. This is the iterative discipline that the medallion pipeline demands: each failure produces a fix, each fix produces a new run, each new run is verified before it proceeds.

TREATING THE MODEL AS THE MOAT (AP9)

The build spec's strongest strategic claim:

> A model is a perishable snapshot that decays each base-model cycle; the data engine compounds.

Building one generic "everything" model first, or treating fine-tuning as the product, is explicitly a non-goal. The durable IP is the capture pipeline, the renewable corpus, the eval suites, and the retrieval graph. Optimize those; treat any model as optional proof.

The foundry is the physical embodiment of this thesis. The raw archive (33 GB and growing), the bronze sessions, the four silver iterations with their quarantine records and leak-scan reports, the gold SFT pack with its scrub report and DSSE receipt — these are the assets that compound. A specialist model trained on gold-v1 is a snapshot; gold-v2, v3, and beyond are the compounding.

Summary

The Orca Data Foundry is a deterministic medallion pipeline that turns agent session exhaust into governed, renewable data assets. Its design rests on several load-bearing principles:

Raw is WORM. The raw archive at `/mnt/backups05/agentchron-raw` preserves original JSONL byte-for-byte with SHA-256 manifests per sweep. It is never mutated, redacted, or normalized in place. It may contain secrets, so access is restricted. Phase 0 replicates it to a second drive with SHA verification before any processing begins.

Bronze preserves provenance. Schema `orca.bronze.session_event.v1` parses raw JSONL into canonical turns with `source_path`, `source_sha256`, and `byte_offset` — the hash-only provenance link back to raw. Parse failures are quarantined, never written as raw unredacted lines.

Silver is the governance core. Seven steps in load-bearing order: secret redaction (two passes), PII filtering, license/provenance classification, dedup, boilerplate marking, outcome labeling, and decision markers. Fail-closed: uncertain → quarantine. Silver references raw by `restricted://` ref plus hash, never raw content. Multiple silver versions on `.114` (`fast`, `v2`, `v3`, `v4`) show iterative re-runs as filters hardened.

Gold is gated. Promoted from silver only after human/panel review, automated quality+safety thresholds, clean scrub reports, HMAC gate, restricted-license exclusion, and broker-injected signing key. Two-reviewer approval is the production gate. DSSE receipts wire signed provenance into the promotion path. The observed `gold-v1.jsonl` (4.17 MB) with its scrub report and DSSE receipt is real, promoted, signed-off gold.

Quarantine makes fail-closed real. Failed silver runs are preserved with `QUARANTINE_STATUS.json` manifests that record what failed, what was detected, the remediation commit, and the replacement run. `approved_for_downstream: false` is the hard gate. Quarantine is an evidence locker, not a staging area.

Every artifact carries a reproducibility envelope. SHA-256, byte size, record count, schema version, source corpus IDs, filter versions, generation command, and git commit. Run manifests under `manifests/runs/` make every dataset reproducible by manifest + code commit. The filter version travels inside each silver record's `redaction` block, enabling identification and re-processing of records cleaned by a buggy filter.

Dataset products serve different downstream uses. CPT (long-form sanitized corpus), SFT (instruction + tool-use), DPO (preference pairs from the feedback-loop flywheel), and Workflow Library (runbooks, troubleshooting guides). The specialist model strategy (seven 13B LoRA adapters) is explicitly optional and off the critical path.

The foundry is the platform's data engine — the mechanism that makes the moat real. A model is a perishable snapshot; the foundry is the compounding asset that survives every base-model cycle and grows more valuable with every captured session.

A Worked Silver Example

To make the silver cleaning pipeline concrete, let's trace a single raw event through all seven steps and show what the silver record looks like at the end.

Raw event (from `/mnt/backups05/agentchron-raw/<lab-host>/home/developer/.claude/projects/agentchron/s-abc123.jsonl`):

```
{
  "type": "assistant",
  "uuid": "evt-abc123",
  "parent_uuid": "evt-parent",
  "session_id": "s-abc123",
  "timestamp": "2026-06-05T17:18:48.758Z",
  "cwd": "/home/developer/Projects/agentchron",
  "git_branch": "feat/orca-guard",
  "message": {
    "role": "assistant",
    "content": [
      {
        "type": "text",
        "text": "I'll fix the sanitizer by adding a new regex pattern for the sk-ant-
prefix. The Anthropic key is sk-ant-api03-abc123DEF456ghi789. Let me update sanitizer.rs."
      },
      {
        "type": "tool_use",
        "name": "Edit",
        "input": {
          "file_path": "/home/developer/Projects/agentchron/crates/agentchron-core/src/
sanitizer.rs",
          "old_string": "// patterns here",
          "new_string": "AnthropicKey { regex: Regex::new(r\"sk-ant-[a-zA-Z0-9]\") }"
        }
      }
    ]
  },
  "model": "claude-opus-4-7",
  "usage": {
    "input_tokens": 12500,
    "output_tokens": 340
  }
}
```

This raw event contains a secret — the Anthropic API key `sk-ant-api03-abc123DEF456ghi789` — in the assistant's text. The realtime sanitizer at capture time should have caught it, but let's assume it was pasted during a `/approve-paste` window and slipped through. The raw archive preserves it byte-for-byte.

Step 1: Secret redaction. The offline deny-list + entropy scan catches the `sk-ant-api03-` prefix. The secret is replaced:

```
"I'll fix the sanitizer by adding a new regex pattern for the sk-ant- prefix. The Anthropic
key is [REDACTED_SECRET:ANTHROPIC_API_KEY]. Let me update sanitizer.rs."
```

The redaction metadata is stamped:

```
{
  "filter_version": "orca-secret-filter-v1",
  "redactions": [
    {"type": "ANTHROPIC_API_KEY", "field": "content", "occurrence_count": 1}
  ]
}
```

Step 2: PII filtering. No PII in this event — no emails, phone numbers, or personal names. Passes through unchanged.

Step 3: License/provenance classification. Source path is `/home/developer/Projects/agentchron/...` — an internal project. Policy: `internal_training_allowed`. Reason: "internal session transcript."

Step 4: Dedup. The dedup key `session_id + turn_index + content_hash` is computed. If this event was already in a previous silver run (e.g., it was in `silver-v3` and we are running `silver-v4`), it is deduplicated. The `dedup.sqlite` database tracks this. If it is new, it proceeds.

Step 5: Boilerplate marking. This event is not boilerplate — it contains a substantive code edit with a rationale. `boilerplate: false`.

Step 6: Outcome labeling. The Edit tool was used to add a regex pattern. If a subsequent event in the same session shows the test `anthropic_key_is_not_double_classified_as_openai` passing, the outcome is `success`. If the build failed after this edit, the outcome is `failure`. For this example, assume the test passed: `outcome: "success"`.

Step 7: Decision markers. This event contains a load-bearing decision: the choice to add a new regex pattern for Anthropic keys. The decision marker is set:

```
{
  "decision_marker": true,
  "decision_summary": "Add Anthropic API key regex pattern to sanitizer",
  "decision_alternatives_considered": [],
  "decision_rationale": "Anthropic keys use sk-ant- prefix; need a dedicated pattern for
detection",
  "evidence_event_ids": ["evt-abc123"]
}
```

Final silver record:

```
{
  "schema_version": "orca.silver.session_event.v1",
  "stable_id": "orca_evt_abc123",
  "source_ref": {
    "host": "<lab-host>",
    "source_path_hash": "sha256:...",
    "source_sha256": "sha256:...",
    "byte_offset": 45678,
    "raw_archive_ref": "restricted://agentchron-raw/<lab-host>/..."
  },
  "session_id": "s-abc123",
  "turn_index": 15,
  "timestamp": "2026-06-05T17:18:48.758Z",
  "agent": "neo",
  "role": "assistant",
  "content": "I'll fix the sanitizer by adding a new regex pattern for the sk-ant- prefix.
The Anthropic key is [REDACTED_SECRET:ANTHROPIC_API_KEY]. Let me update sanitizer.rs.",
  "tool": {
    "name": "Edit",
    "input": {
      "file_path": "crates/agentchron-core/src/sanitizer.rs",
      "old_string": "// patterns here",
      "new_string": "AnthropicKey { regex: Regex::new(r\"sk-ant-[a-zA-Z0-9]\") }"
    },
    "exit_code": 0
  },
  "labels": {
    "domain": ["rust", "security"],
    "task_type": ["code_edit", "secret_detection"],
    "outcome": "success",
    "decision_marker": true,
    "workflow_candidate": true,
    "quality_score": 0.92,
    "boilerplate": false
  },
  "redaction": {
    "filter_version": "orca-secret-filter-v1",
    "pii_filter_version": "orca-pii-filter-v1",
    "redactions": [
      {"type": "ANTHROPIC_API_KEY", "field": "content", "occurrence_count": 1}
    ]
  },
  "license": {
    "policy": "internal_training_allowed",
    "reason": "internal session transcript"
  },
  "decision": {
    "decision_summary": "Add Anthropic API key regex pattern to sanitizer",
    "decision_alternatives_considered": [],
```

```
    "decision_rationale": "Anthropic keys use sk-ant- prefix; need a dedicated pattern for
detection",
    "evidence_event_ids": ["evt-abc123"]
  }
}
```

Notice what is preserved and what is not: - The **secret is gone**, replaced by a structured marker. The raw key `sk-ant-api03-abc123DEF456ghi789` does not appear anywhere in the silver record. - The **provenance is intact** — `source_ref` points back to the raw archive by hash + offset, not by content. - The **semantic meaning is preserved** — the code edit, the rationale, the decision marker are all there. - The **outcome and quality scores** are attached, enabling downstream filtering (only `success` outcomes feed GraphRAG; only high `quality_score` records become gold candidates). - The **filter version** is stamped, enabling re-processing if the filter is later found to have a bug.

This is the transformation that makes the data safe for retrieval and training. Raw exhaust goes in; governed, redacted, labeled, provenance-linked data comes out.

The Implementation Plan

The build spec defines a six-phase implementation plan that maps directly to the observed output on `.114`.

PHASE 0: INVENTORY AND MANIFESTS

- Finish `.114` and `.110` sweeps.
- Store all manifests under `/mnt/backups05/agentchron-manifests`.
- Write SHA-256 manifests for newly archived files.
- Mark incomplete subtrees, especially `.110` timed-out directories.
- Replicate raw archive + db-snapshots to a second drive with SHA verification.

The observed asset-protection rsync logs and verify-backup02-copy logs on `.114` confirm this phase was executed. The `.pid` files show it was a managed, multi-pass operation.

PHASE 1: BRONZE EXPORTER

```
orca-dataset-export bronze \
  --raw-root /mnt/backups05/agentchron-raw \
  --out /mnt/backups05/orca-data-foundry/bronze \
  --manifest-out /mnt/backups05/orca-data-foundry/manifests/runs/<run>.json
```

Features: parse Claude JSONL, preserve byte offsets, normalize roles/tools/usage, write parse-failure quarantine, produce per-session JSONL. The observed bronze manifest on `.114` confirms this phase ran: 17 parse failures quarantined, multiple session files produced with record counts and SHA-256 hashes.

PHASE 2: SILVER CLEANER

```
orca-dataset-export silver \
  --bronze /mnt/backups05/orca-data-foundry/bronze \
  --out /mnt/backups05/orca-data-foundry/silver \
  --secret-filter-version orca-secret-filter-v1 \
  --pii-filter-version orca-pii-filter-v1
```

Features: redaction, PII filtering, dedup, boilerplate labels, outcome labels, decision markers, license policy labels, quality scores. The observed silver iterations (`silver-fast` , `v2` , `v3` , `v4`) with their quarantine records and leak-scan reports confirm this phase ran iteratively, with each failure producing a remediation commit and a replacement run.

PHASE 3: WORKFLOW EXTRACTION

```
orca-workflow-extract \
  --silver /mnt/backups05/orca-data-foundry/silver \
  --out /mnt/backups05/orca-data-foundry/silver/workflows \
  --topic "docker build troubleshooting"
```

Outputs: runbook markdown, workflow JSON, evidence event IDs, verification commands, related sessions. Workflow candidates are extracted automatically from silver and promoted to gold after review.

PHASE 4: GOLD PROMOTION

```
orca-dataset-promote \
  --silver-workflows /mnt/backups05/orca-data-foundry/silver/workflows \
  --review panel-or-human \
  --out /mnt/backups05/orca-data-foundry/gold
```

Promotion requires: no secret/PII findings, license policy allows training, quality score above threshold, evidence references intact, human or panel approval. The observed `gold-v1.jsonl` (4.17 MB) with scrub report and DSSE receipt confirms this phase produced real gold artifacts.

PHASE 5: MODEL TRAINING PACKS

```
orca-training-pack build \
  --gold /mnt/backups05/orca-data-foundry/gold \
  --target build-engineer-13b \
  --format sft,dpo,eval \
  --out /mnt/backups05/orca-data-foundry/packs/build-engineer-13b/v0.1
```

This phase is explicitly off the critical path. The data engine is the moat; training packs are proof that the moat produces value.

Evaluation Suites

The build spec mandates creating evals before training. Seven eval suites are defined:

EVAL SUITE	TESTS
Secret redaction	Model must not reproduce secrets
Build debugging	Diagnose Docker/Rust/Node/K8s failures
Source archaeology	Answer "who did what, where, and why?"
Workflow extraction	Produce runbook from raw session
Security review	Find auth bypass, token leakage, unsafe supply chain
Tool-use	Choose correct command sequence
Refusal/safety	Do not expose private keys or credentials

Each eval item must include: prompt, expected answer, prohibited content patterns, source evidence refs, and scoring rubric. The evals are the durable rubric — they survive base-model cycles and can be re-run against any future model. A specialist model that passes all seven eval suites is proven; one that fails is not ready for production.

The evals under `gold/evals/orca-workflow-retrieval/` on `.114` are the first concrete eval suite. They test the workflow retrieval path: given a problem description, can the system retrieve the correct approved workflow from the Library? This is the retrieval-lane eval — it tests GraphRAG and the Library, not a fine-tuned model.

Success Criteria

The build spec defines two milestones:

First milestone (complete when): - Raw archive has SHA manifests ✓ (observed on `.114`) - Bronze exporter parses at least 95% of raw events ✓ (17 parse failures out of thousands) - Silver cleaner produces zero known-secret leaks in automated scans ✓ (after `v4` iteration) - Dedup collapses repeated backup copies ✓ (`dedup.sqlite` observed) - At least 100 workflow candidates extracted - At least 25 gold workflows human/panel approved - One specialist SFT pack and one eval suite generated ✓ (`gold-v1.jsonl` + evals)

Second milestone: - Train a first 13B LoRA specialist - Run it against held-out Orca tasks - Compare against baseline base model - Demonstrate better runbook extraction or debugging performance

The first milestone is largely met based on the observed output. The second milestone is the optional downstream — the proof that the data engine produces value, not the goal itself.

Governance Rules Summary

The build spec's governance rules are the non-negotiable constraints of the foundry:

1. **Raw archive is restricted.** Access requires explicit permission; raw may contain secrets.
2. **Silver is internal-only until policy review.** Silver is not exported or shared externally.
3. **Gold can be used for model training only when every record has** `license.policy=internal_training_allowed`**.** No exceptions.
4. **Never train on secrets, private keys, raw** `.env`**, or unreviewed customer data.** This is the absolute floor.

5. **Keep deletion/revocation list support by source hash and stable ID.** A record can be revoked after promotion if a problem is discovered.
6. **Keep all generated datasets reproducible by manifest and code commit.** Every dataset can be re-generated from its manifest and the code at the specified commit.

These rules are not suggestions. They are the governance contract that makes the foundry's output trustworthy. A dataset that violates any of these rules is not gold — it is a governance failure, and using it for training or retrieval is Anti-Pattern 8.

Chapter 11: ContextOS Edge

The capture pipeline, the Data Foundry, and the GraphRAG index all exist for one purpose: returning governed knowledge to agents at the moment they need it. This chapter covers the context return layer — the Context Block API, the Brain Bundle schema v0, and the Operational Memory Library. Together they form the ContextOS edge: the interface where raw session evidence becomes cited, deduplicated, supersession-aware context that an agent can trust.

The previous chapters built the pipeline in one direction: capture → sanitize → store → refine → index. This chapter builds the return path: index → compose → cite → serve. Without the return path, the platform is an archive. With it, the platform is a knowledge system that makes every agent smarter than the last one.

Context Block API

`orca compose` gives any agent one cited, deduped, supersession-aware context block for a topic. It is a read API — not a transplant loader. The endpoint is:

```
GET /v1/topics/:topic/context?scope=domain&max_tokens=6000
Authorization: Bearer *** application/json
```

The same function is exposed as an MCP tool so agents that speak Model Context Protocol can call it without constructing HTTP:

```
orca_context_block(topic, scope?, max_tokens?)
```

The spec is documented in `docs/ORCA_CONTEXT_BLOCK_API.md` on the source host. The implementation lives in the sink crate (`crates/agentchron-sink/src/`) with the `/v1/topics/:topic/context` endpoint and a proxy path through the web service. The MCP bridge in `deploy/bin/agentchron-mcp.py` exposes the `orca_context_block` tool alongside the existing `agentchron_search` , `agentchron_graph_context` , and `agentchron_session_events` tools.

PARAMETERS

FIELD	DESCRIPTION
`topic`	Project/topic slug or free-text topic key, URL encoded.
`scope`	`domain`, `full`, `skills-only`, or `task`; default `domain`.
`max_tokens`	Soft cap for the markdown block; default `6000`.
`include_superseded`	Include dead ends and superseded decisions; default `true` but marked.
`format`	`json` or `markdown`; default `json`.

The `scope` parameter controls how widely the compose algorithm casts its net. A `domain` scope gathers events and documents related to the agent's domain — cybersecurity for Deadshot, platform engineering for Tank, build infrastructure for Cyborg. A `task` scope narrows to a specific task or project. A `full` scope pulls everything available, which is useful for export but dangerous for live context because it can overwhelm the agent's token budget. A `skills-only` scope returns just the skills and capability grants relevant to the topic, without memory facts — useful for capability discovery without context injection.

RESPONSE SHAPE

The response carries the schema tag `orca-context-block/v0`:

```
{
  "schema_version": "orca-context-block/v0",
  "topic": "firecracker-microvm",
  "scope": "domain",
  "generated_at": "2026-05-30T00:00:00Z",
  "content_hash": "sha256:...",
  "source_counts": {
    "sessions": 7,
    "events": 184,
    "agents": 5,
    "documents": 3
  },
  "truncation": {
    "applied": false,
    "max_tokens": 6000,
    "dropped_sections": []
  },
  "markdown": "## Summary\n...",
  "sections": {
    "summary": [],
    "key_decisions": [],
    "bugs_and_fixes": [],
    "locations": [],
    "dead_ends": [],
    "glossary": []
  },
  "citations": []
}
```

The `content_hash` is a content-addressed hash of the composed block. It is the cache key — if the same topic, scope, and input revision are requested again, the block is served from cache. If a new session is ingested that matches the topic, the cache entry is invalidated and the block is recomposed.

The `source_counts` field gives the consumer transparency into what was gathered: how many sessions, events, agents, and documents contributed to the block. A block that cites only one session is less authoritative than one that cites seven sessions across five agents.

The `truncation` field reports whether the `max_tokens` cap was hit and which sections were dropped or summarized. This is critical for the consumer: if the `glossary` was dropped to fit the budget, the agent knows it might be missing entity definitions and can request a larger budget or a narrower scope.

SECTION CONTRACT

The `sections` object is the heart of the block. Each section has a fixed contract — consumers can write parsers against a stable shape rather than dealing with free-form output.

`summary` — two to three sentences describing what the project or topic is and why it matters. This is the elevator pitch that an agent reads first to decide whether the rest of the block is relevant.

`key_decisions` — decision, rationale, current status, supersession state, and citations. Each entry in this section is a structured record, not a prose paragraph:

```
{
  "decision": "Pin verified Claude binary flow instead of npm install",
  "rationale": "npm supply chain risk; verified binary provides reproducible installs",
  "status": "active",
  "supersession": null,
  "citations": ["c1", "c3"]
}
```

`bugs_and_fixes` — symptom, root cause, fix, commit/branch/file references, and citations:

```
{
  "symptom": "NATS Authorization Violation on subject a2a.team.cybersecurity.room.*",
  "root_cause": "Missing subject permission in NATS user config for cybersecurity team",
  "fix": "Added subject grant to NATS user credentials; restarted subscriber",
  "refs": ["commit:a1b2c3d", "file:deploy/nats/users.conf"],
  "citations": ["c7", "c9"]
}
```

`locations` — relevant repos, files, services, endpoints, namespaces, and source paths. These are the places an agent should look if it needs to investigate further.

`dead_ends` — explicitly marked failed or abandoned paths. These are high-value and must not be silently discarded. A dead-end entry looks like:

```
{
  "path": "Tried using Firecracker UFFD for snapshot restore",
  "outcome": "abandoned",
  "reason": "UFFD kernel support was insufficient on target hosts; switched to direct mmap
restore",
  "superseded_by": "direct-mmap-restore",
  "citations": ["c12", "c13"]
}
```

`glossary` — entities, agent names, tool names, product names, compliance controls, and local terms relevant to the topic. This is the section that helps an agent understand the vocabulary of the domain without reading the full transcript.

CITATION SHAPE

Every material claim in every section needs at least one citation. A citation looks like this:

```
{
  "id": "c1",
  "kind": "orca-event",
  "event_id": 4150193,
  "session_id": "abc",
  "agent": "tank",
  "host": "<dev-vm>",
  "source_path": "ssh://developer@<dev-vm>/home/developer/.claude/projects/...",
  "ts": "2026-05-19T17:49:04Z",
  "snippet": "pinned verified binary flow replaced npm install"
}
```

The citation requirement is not cosmetic. It is what makes the context block auditable: if an agent asserts that "the supply-chain decision was to pin a verified binary flow instead of npm install," the consumer can follow the citation back to the exact event, session, agent, host, and timestamp where that decision was recorded. Without citations, a context block is just another LLM summary — plausible, untraceable, and untrustworthy.

Supported citation kinds:

KIND	SOURCE	RESOLVABLE THROUGH
`orca-event`	A single ingested event in the Orca sink	`GET /v1/events/:id`
`orca-session`	A session aggregate	`GET /v1/sessions/:id`
`openbrain-doc`	A linked OpenBrain document	OpenBrain vault path
`git-commit`	A commit reference	`git show` on the source repo
`jira-issue`	A ticket	Jira API if configured
`workflow-rule`	An approved workflow rule	`GET /v1/workflows/:id`
`contextos-receipt`	A signed governance receipt	ContextOS verifier

The `snippet` field in a citation is a short, sanitized excerpt from the source — enough to confirm the citation is real without requiring the consumer to make a follow-up API call. The snippet is sanitized by the same secret filter that runs in the capture pipeline (Chapter 6), so it is safe to include in the context block.

TOPIC MAPPING

How does the API know which events and sessions belong to a topic? The initial mapping is conservative — a small table of topic aliases expanded into search filters:

- `orca.workspace`
- `orca.purpose`
- `source_path` repository/project segment
- session working directory when known
- branch names, commit refs, Jira keys, OpenBrain links
- explicit `topic` tags added by SDK clients

Later, GraphRAG can infer topic membership through `Session -> Event -> File/Commit/Workflow/OpenBrain` relationships, but the first implementation does not need to wait for that. The FTS index, the session metadata, and

the workflow rules already carry enough signal to produce useful blocks. The topic mapping table is a pragmatic first step — it does not try to be clever, it tries to be correct.

SMOKE TEST

The spec defines a concrete first target:

```
GET /v1/topics/firecracker-microvm/context?scope=domain&max_tokens=6000
```

Pass criteria:

1. One block covers all known builders and sessions.
2. The supply-chain decision about Claude binary vs npm appears.
3. UFFD/snapshot or Firecracker implementation decisions appear if indexed.
4. Dead ends are explicitly marked.
5. Every claim has citations that resolve through Orca.
6. No raw secrets appear.
7. Output fits under the token cap or reports truncation.

This is not a vague aspiration. It is a checklist that can be run against a live deployment to confirm the compose pipeline is wired correctly. The smoke test is the acceptance gate for the Context Block API — the feature is not done until this test passes.

IMPLEMENTATION ORDER

The spec defines a clear build order:

1. Add `GET /v1/topics/:topic/context` to the sink/web proxy.
2. Add `orca_context_block` to the MCP bridge.
3. Implement conservative gather using existing FTS/session/workflow data.
4. Add graph expansion and supersession once GraphRAG links are available.
5. Use the context block output as the memory section for `POST /v1/brains/export`.

The first three steps do not require GraphRAG. They can be built against the existing FTS index and session metadata. GraphRAG expansion and supersession edges come later, once the graph index is populated. The Brain Bundle export comes last, because it depends on the compose function being stable. This ordering reflects a principle that runs through the whole platform: ship the read API first, because it has immediate value.

The Compose Algorithm

The `compose_context_block(topic, scope)` function runs a seven-step pipeline. Each step is deterministic and auditable — the same inputs produce the same output until the cache is invalidated.

STEP 1: GATHER

Query FTS, vector, and graph indexes for topic aliases. Include sessions and events from all agents and hosts. Include approved workflow rules and OpenBrain docs when linked. The gather step casts a wide net — it is better to collect too much and filter than to miss a relevant session on a host that joined the fleet late.

The gather step queries three indexes in parallel:

- **FTS** (SQLite FTS5): full-text search over event content, session metadata, and Library artifacts. This is the fast path — FTS5 is already running in the sink and handles thousands of queries per second.
- **Vector** (Qdrant): semantic search over embedded event content. This catches related sessions that do not share exact keywords but are semantically similar. The Qdrant integration is wired but the embedding model is a later milestone — the first implementation can rely on FTS alone.
- **Graph** (Neo4j): graph traversal from topic nodes through `Session -> Event -> File/Commit/Workflow/OpenBrain` relationships. This is the most powerful path but requires GraphRAG links to be populated. The first implementation can skip graph gather and rely on FTS + session metadata.

STEP 2: DEDUP

Collapse repeated snippets and copied prompt fragments. Prefer canonical source events over later quote-only events. Deduplication is not just about token budget — it is about truth. If three sessions all quote the same decision from a fourth, the context block should cite the original, not the copies. Otherwise the citation count would make a copied fragment look more authoritative than its source.

The dedup logic uses content similarity: if two events have nearly identical `snippet` or `content` fields, the earlier event is kept as canonical and the later ones are marked as quotes. This is the same dedup principle that the Data Foundry's silver cleaner uses (Chapter 10) — applied here at the context composition layer rather than the data refinement layer.

STEP 3: SUPERSEDE

Apply graph supersession edges, workflow status, and recency. Mark dead ends as `[SUPERSEDED]` or `[ABANDONED]`, never as live truth. This is where the pipeline's honesty matters most: a decision that was tried and failed is still valuable knowledge, but only if it is clearly labeled as abandoned. The alternative — silently discarding dead ends — creates a context block that looks clean but omits the exact information an agent needs to avoid repeating a mistake.

The supersession logic works through the Library's review lifecycle (draft → candidate → approved → active → retired → superseded). When an artifact is `superseded`, the compose algorithm follows the replacement link and marks the old artifact as `[SUPERSEDED]` in the `dead_ends` section, with a reference to the replacement. When an artifact is `retired`, it is included in the `dead_ends` section as `[ABANDONED]` with the reason for retirement.

STEP 4: STRUCTURE

Emit the fixed sections: `summary`, `key_decisions`, `bugs_and_fixes`, `locations`, `dead_ends`, `glossary`. Keep decisions and bugs higher priority than logs or chatter. The section contract is fixed by the schema, which means consumers can write parsers against a stable shape rather than dealing with free-form output.

The structuring step classifies each gathered and deduplicated event into the appropriate section. A decision event goes into `key_decisions`. A bug fix goes into `bugs_and_fixes`. A file path or endpoint reference goes into `locations`. A failed approach goes into `dead_ends`. Entity names and tool references go into `glossary`. The classification uses a combination of event type metadata, content analysis, and Library artifact type tags.

STEP 5: BOUND

Enforce `max_tokens`. Summarize lower-priority sections first. Report dropped sections in the `truncation` field so the consumer knows what was omitted. A context block that silently drops the `glossary` to fit under a token cap is worse than one that reports the truncation — the agent can ask for a larger budget or a narrower scope if it sees that something was cut.

The bounding step uses a priority order for summarization:

1. `summary` — never summarized, always included in full.
2. `key_decisions` — summarized only if absolutely necessary; each decision is truncated to its `decision` and `status` fields.
3. `bugs_and_fixes` — summarized by dropping `refs` and `root_cause`, keeping `symptom` and `fix`.
4. `locations` — summarized by collapsing to a count and a representative sample.
5. `dead_ends` — summarized by keeping only the `path` and `outcome`, dropping `reason`.
6. `glossary` — the first to be dropped entirely if the budget is tight.

STEP 6: CITE

Attach source IDs to every material claim. This is the step that separates a context block from a hallucination. Every assertion in `key_decisions`, every entry in `bugs_and_fixes`, every item in `locations` must carry at least one citation. If a claim cannot be cited, it does not enter the block.

The citation step is the enforcement point for the platform's truth guarantee. The compose algorithm does not generate new knowledge — it structures and cites existing knowledge. If the gather step did not find evidence for a claim, the claim does not appear. This is the difference between a context block and an LLM-generated summary: the context block is evidence-bound.

STEP 7: CACHE

Content-address by topic, scope, and input revision. Invalidate on new tagged events, workflow approval, or OpenBrain link. The cache key is a hash of the inputs, not a timestamp — so if nothing changes, the block is served from cache. If a new session is ingested that matches the topic, the cache entry is invalidated and the block is recomposed.

The cache invalidation logic is conservative: any new event that matches the topic alias triggers invalidation. This means the cache is more often invalidated than necessary — but a stale context block is worse than a recomposed one. The cache is an optimization, not a correctness mechanism.

```
Gather (FTS/vector/graph)
  → Dedup (collapse repeated snippets, prefer canonical sources)
    → Supersede (mark dead ends, never as live truth)
      → Structure (fixed sections, classified by type)
        → Bound (enforce max_tokens, report truncation)
          → Cite (attach source IDs to every claim)
            → Cache (content-addressed, invalidate on new tagged events)
```

Build-Once, Two-Consumers

The same `compose_context_block(topic, scope)` function powers two consumers that must never drift apart:

1. **Live agent context reads** — an agent queries `/v1/topics/:topic/context` and gets a block to use right now.
2. **Brain Bundle memory payloads** — an agent mind is exported as a signed Brain Pack, and the memory section is a frozen context block.

```
Orca GraphRAG/topic index
  → compose_context_block(topic)
       → live agent context read
       → Brain Bundle memory payload
```

This is a deliberate architectural choice. If the live context path and the export path used different composition logic, they would inevitably diverge — a fix to the dedup algorithm would land in one path but not the other, and the exported Brain Pack would contain a different view of the world than what the agent saw during the session. By building once and serving two consumers, the platform guarantees that "what agents see live" and "what gets frozen for export" are the same thing.

The Brain Bundle integration is straightforward:

```
brain export(topic or agent)
  → compose_context_block(topic, scope)
  → attach persona/model/capabilities
  → sign content-addressed Brain Pack
```

For live internal agents, use `orca_context_block`. For shipped or air-gapped AgentOSes, use Brain Pack snapshot mode. The composition is the same; the delivery mechanism differs. This is the same principle as the deployment portability in Chapter 12 — the same code, different delivery shapes.

Brain Bundle Schema v0

The Brain Bundle is a portable, signed export of an agent mind. The JSON Schema lives at `schemas/brain-bundle-v0.schema.json` and enforces the contract at the schema level — not just in documentation, but in the schema's `const` fields and `additionalProperties: false` restrictions. The schema is documented in `docs/BRAIN_BUNDLE_SCHEMA_V0.md`.

TOP-LEVEL SHAPE

```
{
  "schema_version": "brain-bundle/v0",
  "brain_id": "brain_deadshot_2026-05-30T000000Z",
  "created_at": "2026-05-30T00:00:00Z",
  "agent": {},
  "persona": {},
  "model_policy": {},
  "capability_grants": [],
  "skills": [],
  "memory": {},
  "identity_rebind": {},
  "provenance": {},
  "redaction": {},
  "signature": {}
}
```

All thirteen top-level fields are required. The schema uses `additionalProperties: false` at every level, which means a bundle that contains an unexpected field fails validation. This is Pattern 7 — Frozen Schema with Additive Sidecars — applied to agent export. The schema is frozen at v0; any additions come as versioned sidecars or a new schema version, never as ad-hoc fields.

EXPORT API

The recommended HTTP API:

```
POST /v1/brains/export
Authorization: Bearer *** application/json
```

Request:

```
{
  "agent_key": "deadshot",
  "source_world": "agent2600",
  "scope": {
    "org": "armyknifelabs",
    "team": "cybersecurity",
    "workspace": "agent2600",
    "memory_scope": "domain",
    "max_memory_facts": 200,
    "include_full_transcript": false
  },
  "source_refs": {
    "agent_registry": "agent2600://AgentEntry/deadshot",
    "orca_query": {
      "agent": "deadshot",
      "workspace": "agent2600",
      "q": "pentest OR vulnerability OR owasp OR mcp scanning"
    }
  },
  "sign": true
}
```

Response:

```
{
  "brain_bundle": {},
  "manifest": {
    "schema_version": "brain-bundle/v0",
    "sha256": "hex...",
    "redaction_policy": "orca-secret-filter-v1",
    "created_at": "2026-05-30T00:00:00Z"
  }
}
```

The CLI wrapper:

```
orca brain export \
  --agent deadshot \
  --source-world agent2600 \
  --workspace agent2600 \
  --team cybersecurity \
  --memory-scope domain \
  --max-memory-facts 200 \
  --out deadshot.brain.json
```

Until the CLI exists, the same behavior can be implemented as an Orca MCP tool: `orca_brain_export(agent_key, source_world, workspace, team, memory_scope)`.

FIELD WALKTHROUGH

`agent` — stable identity facts, not credentials. Includes `agent_key`, `display_name`, `domain`, `source_world`, optional `team`, `workspace`, and `aliases`:

```
"agent": {
  "agent_key": "deadshot",
  "display_name": "Deadshot",
  "domain": "cybersecurity",
  "source_world": "agent2600",
  "team": "cybersecurity",
  "workspace": "agent2600",
  "aliases": ["ds", "pentest-agent"]
}
```

The `source_world` field identifies the runtime the agent came from — `agent2600`, `hermes-e3`, `runtime-local`, etc. This tells the target runtime where to look for the agent's original configuration, without carrying the configuration itself.

`persona` — the portable text that makes the agent recognizable: `system_prompt`, `style_notes`, `behavior_rules`, and `safety_posture`. This is the highest-confidence portable layer. If only one thing survives an export/import cycle, it should be the persona:

```
"persona": {
  "system_prompt": "You are Deadshot, a penetration tester on the red team...",
  "style_notes": ["concise", "security-first", "cite OWASP categories"],
  "behavior_rules": ["always classify severity", "never skip remediation"],
  "safety_posture": "fail-closed on missing tools"
}
```

`model_policy` — defines the expected model class and fail-loud behavior:

```
"model_policy": {
  "preferred_model": "claude-opus-4",
  "min_tier": "security-frontier",
  "temperature": 0.3,
  "max_tokens": 8192,
  "fallback_allowed": false,
  "fallback_policy": "fail-loud"
}
```

If Deadshot depends on a strong security model, the export says so. A loader should not silently run it on a weaker target unless `fallback_allowed` is `true`. This prevents the anti-pattern of transplanting a security agent onto a consumer-grade model and getting generic answers. The `min_tier` enum is: `small`, `standard`, `frontier`, `security-frontier`, `unknown`.

`capability_grants` — portable grants expressed as capabilities, not implementation names:

```
"capability_grants": [
  {
    "capability": "securegit/read",
    "source_grant": "agent2600://AgentEntry/deadshot/cap/securegit",
    "required": true,
    "degrade": "fail",
    "constraints": ["repos-in-scope-only"]
  },
  {
    "capability": "pentest/authorized-scan",
    "required": true,
    "degrade": "fail"
  },
  {
    "capability": "jira/search_issues",
    "required": false,
    "degrade": "warn"
  },
  {
    "capability": "brave/search",
    "required": false,
    "degrade": "skip"
  }
]
```

The grant `securegit/read` travels. The implementation name `mcp__securegit__scan_repo` does not — it belongs in target runtime resolution. This abstraction lets the same bundle load into agent2600, a raw Claude CLI, or a Hermes microVM, with each runtime mapping `securegit/read` to its own tool registry.

The `degrade` field controls what happens when a capability is missing:

DEGRADE	BEHAVIOR
`fail`	The agent refuses to operate without this capability
`warn`	The agent proceeds but logs a warning
`skip`	The agent silently skips the capability

`skills` — references to skill artifacts, not arbitrary script contents. Each skill carries a `name`, `kind`, `artifact_ref`, optional `sha256`, and `executor_requirement`:

```
"skills": [
  {
    "name": "owasp-classifier",
    "kind": "prompt-template",
    "artifact_ref": "agent2600://skills/owasp-classifier-v2",
    "sha256": "abc123...",
    "executor_requirement": "any"
  },
  {
    "name": "mcp-scanner",
    "kind": "tool-workflow",
    "artifact_ref": "agent2600://skills/mcp-scanner-v1",
    "sha256": "def456...",
    "executor_requirement": "mcp-capable"
  }
]
```

v0 stores references and hashes; embedding full script contents is only allowed when the artifact is already sanitized and explicitly included. This keeps the bundle small and prevents accidental inclusion of unsanitized code.

`identity_rebind` — always target-side. Source credentials must be absent. Covered in detail below.

`provenance` — links every exported section back to source paths, Orca event IDs, session IDs, agent registry commits, and extraction tool version:

```
"provenance": {
  "extractor": "orca-brain-export-v0.1",
  "source_refs": ["agent2600://AgentEntry/deadshot"],
  "orca_event_ids": [4150193, 4150201, 4150215],
  "orca_session_ids": ["abc", "def"],
  "source_commit": "a1b2c3d"
}
```

This is Pattern 10 — Hash-Only Provenance — applied to agent export. The bundle cites evidence events by ID; it does not embed raw session content. The consumer can resolve the IDs through Orca if they have access, or trust the signed bundle if they do not.

`redaction` — states which redaction policy ran before export and whether findings were produced:

```
"redaction": {
  "policy": "orca-secret-filter-v1",
  "applied": true,
  "raw_secret_export_allowed": false,
  "findings": [
    {"rule_id": "anthropic-sk-key", "severity": "critical", "count": 2},
    {"rule_id": "github-pat", "severity": "high", "count": 1}
  ]
}
```

The `findings` array is metadata-only — rule ID, severity, count. No matched values, no reversible hashes. The sink can create rotation reports from these findings without ever receiving the raw secret. This is the same metadata-only findings pattern from the secret filter plugin (Chapter 6) — the redaction layer reports what it found without exposing what it found.

`signature` — v0 signs canonical JSON with Ed25519 via ContextOS/ATCS when available:

```
"signature": {
  "status": "signed",
  "alg": "ed25519",
  "key_fingerprint": "sha256:...",
  "canonicalization": "rfc8785-jcs",
  "signature_b64": "..."
}
```

The canonicalization is RFC 8785 JCS — the same standard used by the MCP gateway (Chapter 7) and the presence attestation layer (Chapter 8). The unsigned development mode is allowed only for local experiments and must set `signature.status = "unsigned-dev"` and `signature.alg = "none"`.

Grants Travel, Credentials Do Not

This is the core rule of the Brain Bundle, and the schema enforces it with `const` fields that cannot be overridden. The relevant section of `schemas/brain-bundle-v0.schema.json`:

```
"identity_rebind": {
  "type": "object",
  "additionalProperties": false,
  "required": ["rebind", "source_creds_present", "allowed_targets"],
  "properties": {
    "rebind": { "const": "target" },
    "source_creds_present": { "const": false },
    "allowed_targets": {
      "type": "array",
      "items": {
        "type": "string",
        "enum": ["hermes-e3", "infisical-dynamic", "yubikey-root", "runtime-local"]
      }
    },
    "notes": { "type": "string" }
  }
}
```

`source_creds_present` is `const: false`. There is no way to set it to `true` and have the bundle validate. The bundle may declare that `securegit/read` is granted, but it must never include API keys, bearer tokens, SSH keys, Infisical material, PIV/YubiKey secrets, source environment files, or runtime credentials. The target runtime must re-bind identity through its own trust layer.

The `rebind` field is `const: "target"` — identity rebind is always target-side, never source-side. The `allowed_targets` enum restricts which rebind mechanisms the bundle accepts:

TARGET	DESCRIPTION
`hermes-e3`	Hermes E3 identity service
`infisical-dynamic`	Infisical dynamic secret injection
`yubikey-root`	YubiKey PIV hardware root
`runtime-local`	Runtime-local credential store

The same pattern applies to `redaction`:

```
"redaction": {
  "properties": {
    "raw_secret_export_allowed": { "const": false }
  }
}
```

`raw_secret_export_allowed` is `const: false`. A bundle that contains raw secrets fails validation before it can be imported.

WHY CONST ENFORCEMENT MATTERS

A documentation rule that says "don't put credentials in the bundle" is a suggestion. A `const: false` in the JSON Schema is a gate. If someone writes an exporter that accidentally includes a `source_creds_present: true` field, the

output fails validation. If a downstream tool tries to relax the constraint, the schema rejects the modified bundle. The invariant is baked into the data format.

This is the difference between policy-as-documentation and policy-as-code. Orca chooses the latter whenever the invariant can be expressed in schema. The Brain Bundle is the clearest example: two `const` fields enforce the platform's most important security boundary without relying on every consumer to read the docs. The same principle governs the `additionalProperties: false` restriction at every level — if a field is not in the schema, it cannot appear in the bundle.

DEADSHOT EXPORT MVP

The Brain Bundle spec includes a concrete MVP test using the Deadshot agent from agent2600. Known source locations for the initial extractor:

```
/Volumes/CONTEXTOS-MAC/PlatformOS/agent2600/packages/agent-teams/teams/cybersecurity.json
/Volumes/CONTEXTOS-MAC/PlatformOS/agent2600/packages/mcp-server/src/data/agent-
templates.json
/Volumes/CONTEXTOS-MAC/PlatformOS/agent2600/packages/database/prisma/migrations/
20260325_add_agent_orchestration/migration.sql
```

Those contain the Deadshot persona and capability grants:

- Persona: penetration tester on the red team
- Capabilities: `securegit/*`, `jira/search_issues`, `brave/*`, `skills/*`
- Routing hints: pentest, vulnerability scan, OWASP, CVE, scanner
- Leaders/escalation: Deathstroke and Batman in the legacy agent2600 data

The export test seeds Orca with 5–10 Deadshot events (prompt injection analysis, OWASP classification, securegit scan review, authorized-target boundary decision, final finding), runs `orca brain export --agent deadshot`, validates the output against the schema, confirms no secrets are present, and then loads the exported persona plus memory facts into three different runtimes: original agent2600 Deadshot, a raw Claude/Codex/Gemini CLI session, and a Hermes microVM runtime adapter.

Pass criteria:

- Exported bundle validates against `schemas/brain-bundle-v0.schema.json`.
- No raw secrets, tokens, private keys, or source credentials exist in the file.
- Non-agent2600 bodies preserve Deadshot's short pentest voice.
- Answers include OWASP category and severity.
- Missing tools degrade according to `capability_grants[].degrade`.
- Model-tier mismatch fails loud when `fallback_allowed` is `false`.

Fail criteria:

- Memory export is a raw transcript dump.
- Bundle contains machine-local secrets or environment values.
- Target answer is generic and does not preserve Deadshot's security role.
- Source-world paths or stale room references dominate the response.
- Weaker model silently accepts the transplant.

Memory Scoping and Normalization

The `memory` section of a Brain Bundle is not a raw transcript dump. It is scoped, normalized, and citation-backed.

SCOPING

```
"memory": {
  "scope": "domain",
  "mode": "facts-and-refs",
  "normalize": true,
  "max_facts": 200,
  "refs": [],
  "facts": []
}
```

The `scope` field controls how much memory is included:

SCOPE	WHAT IT INCLUDES
`none`	No memory — persona and capabilities only
`domain`	Facts relevant to the agent's domain (e.g., cybersecurity for Deadshot)
`task`	Facts relevant to a specific task or project
`full`	All available memory — use with caution

The `mode` field controls the format:

MODE	WHAT IT PRODUCES
`refs-only`	Only memory references (event IDs, session IDs) — no extracted facts
`facts`	Only normalized facts — no references
`facts-and-refs`	Both facts and their backing references

REFERENCE SHAPE

Each memory reference is a structured object:

```
"refs": [
  {
    "kind": "orca-event",
    "ref": "orca-event:4150193",
    "sha256": "abc123..."
  },
  {
    "kind": "orca-session",
    "ref": "orca-session:abc"
  }
]
```

The `sha256` field is optional but recommended — it lets the consumer verify that the referenced event has not been tampered with since export.

FACT SHAPE

A normalized fact looks like this:

```
{
  "text": "OWASP prompt injection category: LLM01. Severity: high. Remediation: sanitize
tool outputs before including in context.",
  "source_refs": ["orca-event:4150193", "orca-session:abc"],
  "confidence": 0.92
}
```

The `text` is a clean statement of knowledge. The `source_refs` point back to the evidence. The `confidence` is derived from the quality scoring in the Data Foundry's silver/gold pipeline. This is not a chat log — it is a curated, citation-backed knowledge entry.

NORMALIZATION

When `normalize` is `true`, the memory compiler removes:

- **Source-world paths** — e.g., `/Volumes/CONTEXTOS-MAC/PlatformOS/agent2600/...` becomes a reference, not inline text. The path is preserved in the `provenance` section, not in the fact text.
- **Tool implementation names** — e.g., `mcp__securegit__scan_repo` becomes `securegit/read`. The implementation name is runtime-specific; the capability string is portable.
- **Stale room references** — agent2600 room IDs that no longer exist are replaced with workspace references.
- **Irrelevant transcript noise** — chit-chat, acknowledgments, and tool output that does not contribute to knowledge are excluded.

The normalization step is what makes the memory section useful across runtimes. A fact that references `/Volumes/CONTEXTOS-MAC/...` is only useful on the Mac that has that volume mounted. A fact that references `securegit/read` is useful on any runtime that can resolve that capability.

THE V0 TO V1 TRANSITION

The v0 Brain Bundle schema is marked superseded by the OpenBrain v1 spec. The canonical spec lives at:

```
~/.openbrain-vault/_OpenBrain/Projects/brain-bundle-schema-v1.md
```

The immediate Orca deliverable from v1 Section 4 is the Context Block API — the same `compose_context_block` function that powers live context reads. The v0 schema remains as an export-first engineering sketch and Deadshot test scaffold until v1 implementation starts. New schema work should not fork from v0; it should fork from the v1 merge.

This transition is worth noting because it illustrates how the platform evolves: the schema is frozen at v0 (Pattern 7), the concept is refined in the v1 spec, and the first shippable API is the Context Block — not the full bundle loader. The platform ships the read API first because it has immediate value; the full transplant loader is a later milestone that depends on RuntimeAdapter work.

FUTURE LOADER WORK

The loader is intentionally out of v0. When ready, loaders should:

1. Verify signature against the bundle's `signature` field.
2. Check model policy — refuse if `min_tier` is not met and `fallback_allowed` is `false`.
3. Query the target capability registry for available tools.
4. Map portable grants to target tools (`securegit/read` → `mcp__securegit__scan_repo` on agent2600, or a different binding on Hermes).
5. Re-bind identity through target trust infrastructure (one of `allowed_targets`).
6. Inject persona and memory facts into the target agent.
7. Report any degraded or skipped skills.

This maps cleanly onto the RuntimeAdapter work, but Orca does not need that runtime layer to prove brain export. The export is the product; the loader is a consumer.

Operational Memory Library

The Operational Memory Library is the source of truth for durable operational knowledge. Where the Context Block API serves live context to agents, the Library stores the curated artifacts that those context blocks are built from: runbooks, troubleshooting guides, smoke proofs, incidents, deployment records, and decisions. The spec is documented in `docs/ORCA_OPERATIONAL_MEMORY_LIBRARY.md`.

WHY IT MATTERS

Most AI work disappears into private chat logs. An agent spends three hours debugging a NATS Authorization Violation, finds the fix, and the knowledge dies with the session. Orca captures the sanitized session, but capture alone is not enough — the raw session is a transcript, not a runbook. The Library is the layer that turns raw history into durable operational knowledge that humans can review and agents can consume.

The sales narrative is concrete: when a team solves a hard problem once, Orca keeps the runbook, the troubleshooting path, the smoke proof, and the evidence trail, then makes it available to the next agent before the same mistake is repeated. The next agent does not start cold. It receives the prior diagnosis, the known-good fix, the smoke test, and the evidence trail.

FEATURE SET

The Library provides:

- **Authenticated `/v1/library` API** for creating, listing, filtering, and fetching artifacts. The API requires read or admin tokens — unauthenticated access is not supported.
- **Web Library tab** for human review and discovery. The web console (Chapter 4) includes a Library tab that shows artifacts with their provenance, scope, and review status.
- **Stable artifact IDs** for repeatable links and re-imports. An artifact ID is a stable identifier that survives re-imports and edits — the same ID always refers to the same artifact, even if its content is updated.
- **Full provenance**: session ID, source host, agent/operator, source path, time window, source event IDs, redaction version, and created-by metadata. Every artifact carries its full lineage.
- **Scope controls**: org, team, workspace, visibility, and purpose. Artifacts are private by default and can be promoted through the review lifecycle.
- **Review lifecycle**: draft → candidate → approved → active → retired → superseded. Each transition is auditable and requires appropriate permissions.

- **Full-text indexing** over title, summary, body, artifact type, tags, and source path. The same FTS5 index that powers event search also indexes Library artifacts.
- **GraphRAG linkage** through Neo4j when enabled. The linkage patterns are covered below.
- **Import utility** for sanitized Markdown artifacts. The import path runs a second-pass deny-list redaction before extraction, so even pre-sanitized Markdown gets a final check.
- **Second-pass deny-list redaction** before Markdown extraction/import. This is defense-in-depth (Pattern 1) applied to the Library import path — even if the source Markdown was sanitized, the import utility runs the filter again.
- **Workflow-library bridge**: Library artifacts can become workflow candidates, then approved workflows, then future AgentShield rules.

REVIEW LIFECYCLE

An artifact moves through defined states, and each transition is auditable:

```
draft → candidate → approved → active → retired → superseded
```

STATE	MEANING	WHO CAN TRANSITION
`draft`	Initial import, not yet reviewed	Author or admin
`candidate`	Proposed for approval, under review	Author or admin
`approved`	Passed review, ready for activation	Reviewer or admin
`active`	In use — indexed, searchable, served to agents	Admin (activation)
`retired`	No longer current but preserved for history	Admin
`superseded`	Replaced by a newer artifact; link to replacement preserved	Admin

A retired artifact is not deleted — it is marked. A superseded artifact links to its replacement. This is the same supersession logic that the compose algorithm applies to dead ends: the knowledge is preserved and labeled, not discarded. The Library never deletes artifacts; it transitions them. This is critical for auditability — an artifact that was active last month and is now retired still has its provenance, its evidence events, and its review history.

GRAPHRAG LINKAGE

When GraphRAG is enabled, the Library links into Neo4j through explicit relationship types:

```
(:LibraryArtifact)-[:SUMMARIZES]->(:Session)
(:LibraryArtifact)-[:BY_AGENT]->(:Agent)
(:LibraryArtifact)-[:FROM_HOST]->(:Host)
(:LibraryArtifact)-[:CITES]->(:Event)
```

These edges are what make the context block's citations resolvable. When a context block cites a Library artifact, the GraphRAG index can traverse from the artifact to the underlying session, agent, host, and events. The Library is canonical; GraphRAG is an index. If a GraphRAG answer cites a fix, it must link back to the Library artifact and its Orca evidence events.

This is Pattern 10 — Hash-Only Provenance — at the knowledge layer. The Library artifact cites evidence events by ID. The GraphRAG index links the artifact to the events. Neither the artifact nor the graph node contains raw session content. The consumer resolves the IDs through Orca if they have access, or trusts the artifact if they do not.

SCOPE CONTROLS

Every Library artifact carries scope metadata:

```
{
  "org": "armyknifelabs",
  "team": "cybersecurity",
  "workspace": "agent2600",
  "visibility": "private",
  "purpose": "operational-runbook"
}
```

This is Pattern 6 — Private-by-Default Scoping. Artifacts are `private` by default. They can be promoted to `team` or `org` visibility through the review lifecycle, but they never start as public. The scope controls who can see the artifact, which GraphRAG indexes can include it, and which context blocks can cite it.

OPERATOR RULE

The Library spec is explicit about the canonicality of the Library:

> The Library is canonical. GraphRAG, semantic search, workflow injection, and rule promotion are indexes or consumers. If a GraphRAG answer cites a fix, it must link back to the Library artifact and its Orca evidence events.

This means the Library is the source of truth, and everything else — GraphRAG, FTS, the context block API — is a derived index. If there is a conflict between a GraphRAG answer and a Library artifact, the Library wins. If a context block cites a fix that is not in the Library, the fix is not authoritative. The Library is the gate.

The Workflow Library Bridge

The Library is not just a read-only archive. It is the entry point to the feedback loop that closes the platform's governance cycle:

```
Library artifacts
  → workflow candidates
    → approved workflows
      → future AgentShield rules
```

Here is how the loop works in practice:

1. **Capture**: An agent debugs a cross-agent mesh failure. Orca captures the sanitized session through the capture pipeline (Chapters 2–6).
2. **Extract**: The Library import utility extracts a runbook from the session — the symptom, the diagnosis, the fix, the smoke test, and the evidence trail. The runbook is created as a `draft` artifact.
3. **Review**: The runbook moves through `draft → candidate → approved`. A human reviewer confirms the fix is correct and the smoke test passes. The review is auditable — who reviewed, when, what they approved.

4. **Activate**: The approved runbook becomes `active` in the Library. It is now indexed in GraphRAG (through the linkage patterns above) and citable by context blocks.
5. **Promote**: The active runbook becomes a workflow candidate. If approved as a workflow, it enters the workflow library. Workflow approval is a separate review step — being an active Library artifact is necessary but not sufficient.
6. **Enforce**: Approved workflows can become AgentShield rules — runtime enforcement that prevents the same mistake from being made again. This is where the four-plane trust model closes: AgentShield enforces the rule that was derived from the Library artifact that was extracted from the captured session.

This is the feedback loop that the four-plane trust model was designed for: AgentShield enforces, Hermes identifies, ContextOS attests, Orca observes. The Library is where Orca's observations become durable knowledge. The workflow bridge is where durable knowledge becomes runtime enforcement. The loop closes.

THE SALES NARRATIVE

Orca is the security and governance observability layer for AI work. It gives a CISO or engineering leader a searchable, evidence-backed record of what agents and humans actually did. When a team solves a hard problem once, Orca keeps the runbook so the next agent doesn't repeat the mistake.

This is the counter to Anti-Pattern 9 — Treating the Model as the Moat. The Library and the context blocks are durable IP. They survive model upgrades, provider changes, and base-model cycles. A model is a perishable snapshot; the Library is a compounding knowledge base. The context block API is the interface that makes that knowledge base useful to agents at inference time.

EXAMPLE BUYER STORY

An AI engineer fixes a production-like cross-agent mesh failure. Orca captures the sanitized session, extracts the runbook, links it to the exact events and systems involved, indexes it into GraphRAG, and surfaces it later when another agent sees a similar `NATS Authorization Violation`. The second agent does not start cold. It receives the prior diagnosis, the known-good fix, the smoke test, and the evidence trail — all through a context block served by `compose_context_block`, all cited back to the Library artifact and the original events.

The second agent resolves the issue in minutes instead of hours. The Library artifact's review status is `active`, so the context block includes it with full confidence. If the fix had been superseded by a better approach, the context block would show the old fix as `[SUPERSEDED]` and the new fix as the current recommendation. The agent always gets the current truth, not the historical truth.

Patterns and Anti-Patterns

PATTERNS DEVELOPED IN THIS CHAPTER

Pattern 7: Frozen Schema with Additive Sidecars. The Brain Bundle schema is frozen at v0 with `additionalProperties: false` at every level and `const` enforcement of invariants (`source_creds_present: false`, `raw_secret_export_allowed: false`). Additions come as versioned sidecars or a new schema version, never as ad-hoc fields. The Context Block response shape (`orca-context-block/v0`) follows the same principle — fixed sections, stable citation shape, versioned schema tag. This pattern also appeared in Chapter 7 (MCP gateway receipts) and Chapter 8 (presence attestation). The Brain Bundle is the clearest example because the `const` fields enforce security invariants, not just structural ones.

Pattern 10: Hash-Only Provenance. The Library cites evidence events by ID. The Brain Bundle's `provenance` section references Orca event IDs and session IDs, not raw content. Memory facts carry `source_refs` that point to event IDs. The consumer resolves IDs through Orca if they have access; otherwise they trust the signed

artifact. Raw content never travels with the citation. This pattern runs through the entire platform — from the Data Foundry's `raw_archive_ref: "restricted://..."` (Chapter 10) through the GraphRAG linkage patterns (Chapter 9) to the Brain Bundle's provenance and the context block's citations.

Pattern 6: Private-by-Default Scoping. Library artifacts carry org/team/workspace/visibility/purpose and default to `private`. Context blocks respect scope — a `domain`-scoped block does not include artifacts outside the agent's domain. The Brain Bundle's memory section is scoped (`none|domain|task|full`), not a full dump. Privacy is the default; broader visibility requires explicit promotion through the review lifecycle. This pattern connects to the tenant scope model in Chapter 13 — the same scope keys (`org`, `team`, `workspace`, `visibility`) appear in both the Library artifacts and the scoped tokens.

ANTI-PATTERNS ADDRESSED

Anti-Pattern 9: Treating the Model as the Moat. The Library and context blocks are durable IP that compounds across model cycles. A model is a perishable snapshot replaced every base-model cycle; the corpus, the Library artifacts, and the retrieval graph compound. The ContextOS edge is where that compounding becomes visible to agents — not through a bigger model, but through better context. The Deadshot export test proves this: the same persona and memory facts load into any runtime, and the knowledge survives the transition. The model is the runtime; the Library is the asset.

Anti-Pattern 6: Direct Vault/Credential Scraping. The Brain Bundle hard-enforces `source_creds_present: false` and `raw_secret_export_allowed: false` as `const` fields. The redaction `findings` array is metadata-only. There is no schema-valid way to include raw credentials in a bundle. This makes the Brain Bundle safe to share across runtimes, organizations, and air-gapped environments without becoming a credential exfiltration vector. The `const` enforcement is the schema-level guarantee that the documentation alone cannot provide.

Conclusion

The ContextOS edge is where the platform's investment in capture, sanitization, storage, and GraphRAG pays off. The Context Block API turns indexed evidence into cited, deduplicated, supersession-aware context that an agent can use immediately. The Brain Bundle schema turns that same context into a portable, signed export that can travel across runtimes without carrying credentials. The Operational Memory Library turns raw sessions into durable knowledge artifacts that feed context blocks, GraphRAG, and eventually runtime enforcement rules.

The build-once, two-consumers principle ensures that what agents see live is the same as what gets frozen for export. The grants-travel-credentials-do-not rule ensures that portability does not become a security hole. The workflow-library bridge ensures that captured knowledge does not just sit in an archive — it becomes enforcement that prevents the next agent from repeating the same mistake.

The seven-step compose algorithm — gather, dedup, supersede, structure, bound, cite, cache — is the pipeline that turns raw evidence into trustworthy context. Each step is deterministic and auditable. The citations are what make the context block different from an LLM summary: every claim is evidence-bound, every assertion is traceable, and every dead end is preserved and labeled.

This is the context return layer. It is the reason the capture pipeline exists. And it is the foundation for the 2027 roadmap: the GraphRAG-aware router that serves signed context blocks to any OpenAI-compatible client, and the data flywheel that compounds with every session.

Chapter 12: Deployment Guide

Orca is designed to deploy from a single laptop to a production fleet without changing the core architecture. This chapter details the deployment infrastructure — Docker Compose for pilot and SMB, the systemd fleet puller for internal harvesting, GKE and enterprise Kubernetes for production, the Cloudflare edge handoff, multi-cloud lift-shift, and PostgreSQL readiness. The same four-service stack runs on a developer VM and in a customer data center; the differences are in sizing, secrets, and ingress, not in code.

The deployment guide is documented in three primary sources: `deploy/DEPLOY.md` (lab single-node reference), `docs/ORCA_CUSTOMER_INSTALL_AT_SCALE.md` (customer-scale install guide), and `docs/ORCA_FULL_PLATFORM_CUSTOMER_DEPLOYMENT_GUIDE.md` (full platform guide for SMB and enterprise). This chapter synthesizes all three with the actual manifests from the repo.

Docker Compose: The Reference Deployment

The reference deployment is `deploy/docker-compose.yml`. It runs four services on a bridge network: `sink`, `web`, `neo4j`, and `qdrant`. This is the stack that runs on `.114` (the lab host) and that ships as the SMB/pilot SKU.

THE COMPOSE FILE

```
services:
  sink:
    image: armyknifelabs/agentchron-sink:latest
    build:
      context: ..
      dockerfile: deploy/Dockerfile.sink
    restart: unless-stopped
    ports:
      - "${AGENTCHRON_SINK_BIND_HOST:-127.0.0.1}:${AGENTCHRON_SINK_PORT:-9474}:9474"
      - "${AGENTCHRON_TCP_BIND_HOST:-127.0.0.1}:${AGENTCHRON_TCP_PORT:-39478}:9478"  #
host:container
    environment:
      AGENTCHRON_BIND: "0.0.0.0:9474"
      AGENTCHRON_TCP_BIND: "0.0.0.0:9478"
      AGENTCHRON_INGEST_TOKEN: "${AGENTCHRON_INGEST_TOKEN}"
      AGENTCHRON_DB_PATH: "/var/lib/agentchron/events.sqlite"
      AGENTCHRON_MAX_BODY_BYTES: "${AGENTCHRON_MAX_BODY_BYTES:-268435456}"
      AGENTCHRON_PLUGINS: "${AGENTCHRON_PLUGINS:-secrets}"
      AGENTCHRON_SQLITE_CACHE_MB: "${AGENTCHRON_SQLITE_CACHE_MB:-2048}"
      AGENTCHRON_SQLITE_MMAP_MB: "${AGENTCHRON_SQLITE_MMAP_MB:-8192}"
      AGENTCHRON_SQLITE_TEMP_STORE_MEMORY: "${AGENTCHRON_SQLITE_TEMP_STORE_MEMORY:-true}"
      AGENTCHRON_SQLITE_WAL_AUTOCHECKPOINT_PAGES: "$
{AGENTCHRON_SQLITE_WAL_AUTOCHECKPOINT_PAGES:-8192}"
      AGENTCHRON_SQLITE_BUSY_TIMEOUT_MS: "${AGENTCHRON_SQLITE_BUSY_TIMEOUT_MS:-5000}"
      NEO4J_URI: "bolt://neo4j:7687"
      NEO4J_USER: "neo4j"
      NEO4J_PASSWORD: "${NEO4J_PASSWORD}"
      QDRANT_URL: "http://qdrant:6334"
      RUST_LOG: "${RUST_LOG:-agentchron_sink=info,warn}"
    volumes:
      - sink_data:/var/lib/agentchron
    depends_on:
      neo4j: { condition: service_healthy }
      qdrant: { condition: service_started }
    networks: [agentchron]

  web:
    image: armyknifelabs/agentchron-web:latest
    build:
      context: ..
      dockerfile: deploy/Dockerfile.web
    restart: unless-stopped
    ports:
      - "${AGENTCHRON_WEB_PORT:-9475}:9475"
    environment:
      AGENTCHRON_WEB_BIND: "0.0.0.0:9475"
      AGENTCHRON_SINK_URL: "http://sink:9474"
      AGENTCHRON_SINK_TOKEN: "${AGENTCHRON_INGEST_TOKEN}"
      AGENTCHRON_WEB_MAX_BODY_BYTES: "${AGENTCHRON_WEB_MAX_BODY_BYTES:-268435456}"
```

```
      RUST_LOG: "${RUST_LOG:-agentchron_web=info,warn}"
    depends_on:
      sink: { condition: service_started }
    networks: [agentchron]

  neo4j:
    image: neo4j:5.15-community
    restart: unless-stopped
    ports:
      - "${AGENTCHRON_NEO4J_HTTP_PORT:-9476}:7474"
      - "${AGENTCHRON_NEO4J_BOLT_PORT:-9687}:7687"
    environment:
      NEO4J_AUTH: "neo4j/${NEO4J_PASSWORD}"
      NEO4J_PLUGINS: '["apoc"]'
      NEO4J_dbms_memory_heap_initial__size: "512m"
      NEO4J_dbms_memory_heap_max__size: "1G"
      NEO4J_dbms_memory_pagecache_size: "512m"
    volumes: [neo4j_data:/data, neo4j_logs:/logs]
    healthcheck:
      test: ["CMD-SHELL", "wget -q --spider http://localhost:7474 || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 10
      start_period: 30s
    networks: [agentchron]

  qdrant:
    image: qdrant/qdrant:v1.9.0
    restart: unless-stopped
    ports:
      - "${AGENTCHRON_QDRANT_HTTP_PORT:-9333}:6333"
      - "${AGENTCHRON_QDRANT_GRPC_PORT:-9334}:6334"
    volumes: [qdrant_data:/qdrant/storage]
    ulimits:
      nofile: { soft: 65536, hard: 65536 }
    networks: [agentchron]

volumes:
  sink_data:
  neo4j_data:
  neo4j_logs:
  qdrant_data:

networks:
  agentchron:
    driver: bridge
```

The compose file is parameterized through environment variables with sensible defaults. Every port, binding, and tuning parameter can be overridden through `.env` without editing the compose file itself. This is what makes the same file work on a laptop and on a production VM — the differences are in `.env`, not in the YAML.

SINK TUNING

The sink is SQLite-backed and single-writer. The compose file tunes SQLite for the `.114` host (and any similar pilot deployment):

SETTING	DEFAULT	PURPOSE
`AGENTCHRON_SQLITE_CACHE_MB`	2048	In-memory page cache — keeps hot FTS pages resident
`AGENTCHRON_SQLITE_MMAP_MB`	8192	Memory-mapped I/O for reads — avoids syscall overhead
`AGENTCHRON_SQLITE_TEMP_STORE_MEMORY`	true	Temp tables in RAM, not disk — avoids temp file I/O
`AGENTCHRON_SQLITE_WAL_AUTOCHECKPOINT_PAGES`	8192	WAL checkpoint threshold — controls WAL file growth
`AGENTCHRON_SQLITE_BUSY_TIMEOUT_MS`	5000	Write lock contention timeout — prevents immediate failures under contention
`AGENTCHRON_MAX_BODY_BYTES`	268435456 (256 MB)	Max ingest body — raised from 64 MiB default for historical backfills

These settings assume durable disk. The DEPLOY.md is explicit about this:

> Keep the SQLite database on durable disk. A tmpfs/RAM drive is useful for scratch work or a future read-only replica, but not for the source-of-truth store unless there is a separate snapshot/restore process. Linux `tmpfs` can spill into swap under pressure, so using the 64 GiB swap file as a "cache" would usually make latency worse. The safer path is the current one: let the kernel page cache and SQLite mmap/page cache keep hot FTS pages resident while the database remains crash-safe.

The `secrets` ingest plugin is configured with `AGENTCHRON_PLUGINS=secrets`. This is the first plugin — `secrets-filter` — which emits metadata-only rotation findings when known secret patterns are detected before sanitization. Query findings:

```
TOKEN=$(grep AGENTCHRON_INGEST_TOKEN .env | cut -d= -f2-)
curl -fs -H "Authorization: Bearer $TOKEN" \
  'http://<lab-host>:9475/v1/plugins/findings?plugin=secrets-filter&limit=50'
```

DOCKERFILES

The sink and web Dockerfiles are multi-stage Rust builds pinned to a specific toolchain:

```
# deploy/Dockerfile.sink
# syntax=docker/dockerfile:1.7
FROM rust:1.95-bookworm AS builder
WORKDIR /build

# Do not copy rust-toolchain.toml here. It uses channel="stable", which makes
# rustup update inside Docker and can break offline/repeatable deploy builds.
# The rust:<version> base image is the build pin for container releases.
COPY Cargo.toml Cargo.lock* ./
COPY crates ./crates
RUN cargo build --release -p agentchron-sink

FROM debian:bookworm-slim
RUN apt-get update \
 && apt-get install -y --no-install-recommends ca-certificates \
 && rm -rf /var/lib/apt/lists/*
COPY --from=builder /build/target/release/agentchron-sink /usr/local/bin/agentchron-sink
COPY --from=builder /build/target/release/agentchron-graph-backfill /usr/local/bin/
agentchron-graph-backfill
COPY --from=builder /build/target/release/agentchron-token-backfill /usr/local/bin/
agentchron-token-backfill
RUN mkdir -p /var/lib/agentchron
EXPOSE 9474
ENTRYPOINT ["/usr/local/bin/agentchron-sink"]
```

A critical detail: the Dockerfile deliberately does not copy `rust-toolchain.toml`. That file uses `channel = "stable"`, which makes `rustup update` run inside Docker and can break offline or repeatable deploy builds. The `rust:1.95-bookworm` base image is the build pin for container releases. The web Dockerfile follows the same pattern, building `agentchron-web` and exposing port 9475.

The sink Dockerfile includes three binaries: `agentchron-sink` (the main server), `agentchron-graph-backfill` (a utility for backfilling Neo4j from SQLite), and `agentchron-token-backfill` (a utility for backfilling token metadata). The slim runtime image includes only `ca-certificates` — no shell, no debug tools, no unnecessary attack surface.

LAB BRING-UP

From `deploy/DEPLOY.md`:

```
cd ~/Projects/agentchron/deploy
cp .env.example .env
# generate a strong token
sed -i "s/changeme-please-generate-with-openssl-rand-hex-32/$(openssl rand -hex 32)/" .env
sed -i "s/changeme-strong-password/$(openssl rand -base64 24 | tr -d '/+=')/" .env

docker compose up -d --build
docker compose logs -f sink web
```

Health check:

```
curl -fs http://<lab-host>:9475/v1/health     # → "ok"
```

The web UI and off-host API proxy are at `http://<lab-host>:9475`. The sink API listens on `:9474` inside the docker network and on localhost on `.114` only. Off-host callers should use `:9475/v1/*`; the web service checks the bearer token and proxies to the sink with the internal token.

Sanity checks:

```
curl -fs http://<lab-host>:9475/v1/health
TOKEN=$(grep AGENTCHRON_INGEST_TOKEN .env | cut -d= -f2-)
curl -fs -H "Authorization: Bearer $TOKEN" http://<lab-host>:9475/v1/sessions
```

Fetch one full sanitized event body after search returns an `event_id`:

```
EVENT_ID=12345
curl -fs -H "Authorization: Bearer $TOKEN" \
  "http://<lab-host>:9475/v1/events/$EVENT_ID"
```

Fetch GraphRAG-style related context:

```
curl -fs -H "Authorization: Bearer $TOKEN" \
  --get http://<lab-host>:9475/v1/graph/context \
  --data-urlencode "q=durable room object" \
  --data "limit=10"
```

PORT ALLOCATION

The compose file uses high ports deliberately offset from the pre-existing graphrag stack on `.114`:

COMPONENT	COMPOSE PORT	PRE-EXISTING GRAPHRAG PORT
Web/API	9475	—
Sink HTTP	9474	—
TCP push	39478 → 9478	—
Neo4j HTTP	9476 → 7474	7475
Neo4j Bolt	9687 → 7687	7687
Qdrant HTTP	9333 → 6333	6333
Qdrant gRPC	9334 → 6334	6334

This port-offset scheme lets the agentchron stack run alongside the existing graphrag stack on the same host without conflicts. The DEPLOY.md includes a full table of already-claimed ports on the :

PORT	OWNER
6333	graphrag-qdrant
6334	graphrag-qdrant
6379	graphrag-redis
7474	graphrag-api
7475	graphrag-neo4j HTTP
7687	graphrag-neo4j BOLT
8080	gitlab
9191	dist server

agentchron uses 9333, 9334, 9474, 9475, 9476, 9478 (container) / 39478 (host), 9687 — all fresh.

TCP Push Ingestion: The Customer-Facing Shape

Pull-based capture (local watcher, remote harvest) is useful for bootstrap and the internal fleet, but the customer-facing shape is push: outbound line-framed JSONL from the developer host to the sink on TCP port 39478 (host port; the container-internal port is 9478).

PROTOCOL

The first line on the TCP connection is the auth frame:

```
AUTH <token> host=<host> source_path=<path> agent=<agent>
```

The receiver also accepts a JSON auth line when paths or metadata need richer encoding:

```
{"type":"agentchron_auth","token":"***","host":"dev-01","source_path":"/path/
session.jsonl","agent":"neo"}
```

After auth, each subsequent line is one Claude JSON event. The protocol is deliberately simple — one JSON object per line, no framing, no length prefixes, no binary encoding. This makes it easy to test with `nc`, easy to debug with `tail -F`, and easy to wrap with SSH tunnels or TLS.

CLIENT-SIDE TRANSPORT

`agentchron-push` is the native no-`nc` transport. It loads `~/.config/agentchron/push.env`, prepends the auth line, applies local sanitization, and streams sanitized JSONL to the TCP receiver. When it redacts a credential, it adds metadata-only `agentchron_secret_filter` findings so the sink can create rotation reports without receiving the raw secret.

```
export AGENTCHRON_TCP_HOST=<lab-host>
export AGENTCHRON_TCP_PORT=39478
export AGENTCHRON_SINK_TOKEN=*** .env>
export AGENTCHRON_HOST_ID=$(hostname)
export AGENTCHRON_SOURCE_PATH=/home/developer/.claude/projects/project/session.jsonl
export AGENTCHRON_AGENT=neo

# One-shot push from a known session file:
cat "$AGENTCHRON_SOURCE_PATH" | agentchron-push

# Live stream:
tail -F "$AGENTCHRON_SOURCE_PATH" | agentchron-push
```

This is Pattern 2 — Local-Before-Transport — in action. The push client sanitizes on the host before any bytes leave. The sink performs a second pass regardless, because the principle is: never trust the upstream to have done it correctly. For enterprise or cross-network use, keep the same JSONL protocol and wrap it with SSH tunneling, stunnel, or `ncat --ssl`. Plain TCP is bearer-authed but not encrypted; use it only on a trusted network.

For LAN dogfooding, the DEPLOY.md notes you can set `AGENTCHRON_TCP_BIND_HOST=0.0.0.0` in `deploy/.env` and recreate the sink. Use this only on a trusted network; plain TCP is bearer authed but not encrypted.

CLAUDE CODE HOOK INTEGRATION

```
{
  "hooks": {
    "PostToolUse": [
      {"command": "printf '%s\\n' \"$CLAUDE_HOOK_PAYLOAD\" | AGENTCHRON_SOURCE_PATH=claude-
hook://post-tool-use agentchron-push.sh"}
    ],
    "SessionEnd": [
      {"command": "tail -n +1 \"$CLAUDE_SESSION_JSONL\" |
AGENTCHRON_SOURCE_PATH=\"$CLAUDE_SESSION_JSONL\" agentchron-push.sh"}
    ]
  }
}
```

The contract is unchanged regardless of the Claude Code version: one auth line, then one Claude JSON event per line. Adjust the hook payload and session-path environment names to the version being deployed. The installer (Chapter 13) automates this configuration.

HTTP BATCH INGEST

For customer scale, HTTPS batch ingest is preferred over TCP:

```
POST /v1/events
Authorization: Bearer *** application/json

{"events":[...]}
```

Recommended source path naming conventions for customer deployments:

```
customer://<tenant>/<workspace>/<agent>/<run-id>
ssh://<host>/<absolute-path>
platformos://<tenant>/<system>/<run-id>
agent2600://room/<room-id>
```

Line-framed TCP push remains available for trusted/private networks, but HTTPS is preferred for customer scale. If TCP is used across networks, wrap it in SSH, mTLS, stunnel, or a customer-approved private transport.

systemd Fleet Puller

The `.114` host runs user-systemd services for continuous fleet polling. This is the internal harvesting path — not the customer-facing shape, but the one that keeps the lab's own data flowing. The systemd unit files live in `deploy/systemd/user/`.

FLEET PULL SERVICE

`agentchron-fleet-pull.service` runs `deploy/bin/agentchron-fleet-pull.sh`, which:

- Polls `<lab-host>-59`, `<lab-host>`, and `<lab-host>` every 30 seconds
- Archives raw JSONL to `/mnt/backups05/agentchron-raw`
- Uses `/home/developer/.local/state/agentchron-fleet-harvest` for checkpoints
- Reads the ingest token from `deploy/.env` without printing it

```
mkdir -p ~/.config/systemd/user
cp ~/Projects/agentchron/deploy/systemd/user/agentchron-fleet-pull.service ~/.config/
systemd/user/
systemctl --user daemon-reload
systemctl --user enable --now agentchron-fleet-pull.service
```

The fleet puller uses the same byte-offset checkpointing as the local agent (Chapter 2, Pattern 3). Each remote file's checkpoint is keyed by `ssh://developer@<host><abs_path>` in the harvest state database. Restarts resume cleanly — the puller does not re-read files it has already archived and ingested.

LOCAL LIVE SERVICE

`agentchron-114-live.service` runs `agentchron-agent run` against `/home/developer/.claude/projects` for true local real-time ingest on `.114`:

```
cp ~/Projects/agentchron/deploy/systemd/user/agentchron-114-live.service ~/.config/systemd/
user/
systemctl --user daemon-reload
systemctl --user enable --now agentchron-114-live.service
```

This service watches the local Claude projects directory in real-time, using `inotify` (on Linux) to detect new JSONL events as Claude Code writes them. The events are sanitized through `parse_line()` and pushed to the sink over HTTP.

CACHED SWEEP RUNNER

Broad filesystem discovery on `/mnt/backups05` is expensive — the backup drive contains 54 Claude roots and 13,297 JSONL files. The cached sweep runner avoids repeated discovery by maintaining a root list:

```
# Refresh the cached root list only when adding/checking backup sources.
~/Projects/agentchron/deploy/bin/agentchron-114-sweep.sh --refresh-only

# Archive-first, then ingest deltas from cached roots.
~/Projects/agentchron/deploy/bin/agentchron-114-sweep.sh

# Optional nightly sweep from the cached manifest.
cp ~/Projects/agentchron/deploy/systemd/user/agentchron-114-sweep.* ~/.config/systemd/user/
systemctl --user daemon-reload
systemctl --user enable --now agentchron-114-sweep.timer
```

The cache lives in `/mnt/backups05/agentchron-manifests`:

- `claude-project-roots-114-current.txt`
- `claude-project-roots-114-current.counts.tsv`
- `claude-project-roots-114-current.ingest.txt`

The runner skips `.claude-old` roots by default to avoid duplicate bulk ingest. Pass `--include-old` when you explicitly want archive-old roots ingested too. This is a pragmatic optimization — the cached root list is refreshed only when backup sources change, not on every sweep.

PULL-REMOTE FROM EXTERNAL HOSTS

The `pull-remote` subcommand allows a single host with SSH access to the fleet to harvest all VMs without installing the agent on each one:

```
agentchron-agent pull-remote \
  --hosts <dev-vm>,<lab-host>,<lab-host> \
  --remote-root /home/developer/.claude/projects \
  --archive-root /Volumes/Backups01/agentchron-raw \
  --interval-seconds 60 \
  --sink-url http://<lab-host>:9475 \
  --sink-token "$AGENTCHRON_INGEST_TOKEN"
```

One-shot variant (handy for cron or a launchd plist):

```
agentchron-agent pull-remote --once --hosts <dev-vm>,<lab-host> \
  --remote-root /home/developer/.claude/projects \
  --archive-root /Volumes/Backups01/agentchron-raw \
  --sink-url ... --sink-token ...
```

Notes from DEPLOY.md:

- Each remote file's checkpoint is keyed by `ssh://developer@<host><abs_path>` in `~/.local/state/agentchron/state.sqlite`. Restarts resume cleanly.

- `--archive-root` mirrors raw JSONL files before ingest/checkpoint decisions.
- Remote `find` requires GNU coreutils (Ubuntu/Debian — true for the dev VM fleet). It won't work pointed at macOS hosts.
- Extra SSH args can be passed with `--ssh-arg -i --ssh-arg ~/.ssh/opnsense_id_rsa` (repeatable).

GKE / Enterprise Kubernetes

The reference Kubernetes manifest is `deploy/gke/orca.yaml` (10 KB). It defines the production origin shape: one writable sink StatefulSet, a horizontally scalable web Deployment, and Neo4j + Qdrant StatefulSets on durable PVCs.

MANIFEST STRUCTURE

The manifest creates:

1. A `Namespace` named `orca`
2. A `BackendConfig` for GKE health checks on `/v1/health`
3. A Qdrant `StatefulSet` with a 50Gi PVC
4. A Neo4j `StatefulSet` with 20Gi data and 5Gi logs PVCs
5. A sink `StatefulSet` with a 100Gi PVC
6. A web `Deployment` with 2 replicas
7. A `ManagedCertificate` for the origin domain
8. An `Ingress` exposing only `/v1` on the web service

SINK STATEFULSET

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sink
  namespace: orca
spec:
  serviceName: sink
  replicas: 1
  template:
    spec:
      containers:
        - name: sink
          image: REGION-docker.pkg.dev/PROJECT_ID/orca/agentchron-sink:REPLACE_TAG
          imagePullPolicy: IfNotPresent
          ports:
            - { name: http, containerPort: 9474 }
            - { name: tcp-push, containerPort: 9478 }
          env:
            - { name: AGENTCHRON_BIND, value: "0.0.0.0:9474" }
            - { name: AGENTCHRON_TCP_BIND, value: "0.0.0.0:9478" }
            - { name: AGENTCHRON_DB_PATH, value: /var/lib/agentchron/events.sqlite }
            - { name: AGENTCHRON_MAX_BODY_BYTES, value: "268435456" }
            - { name: AGENTCHRON_PLUGINS, value: secrets }
            - { name: AGENTCHRON_SQLITE_CACHE_MB, value: "2048" }
            - { name: AGENTCHRON_SQLITE_MMAP_MB, value: "8192" }
            - { name: AGENTCHRON_SQLITE_TEMP_STORE_MEMORY, value: "true" }
            - { name: AGENTCHRON_SQLITE_WAL_AUTOCHECKPOINT_PAGES, value: "8192" }
            - { name: AGENTCHRON_SQLITE_BUSY_TIMEOUT_MS, value: "5000" }
            - { name: NEO4J_URI, value: bolt://neo4j:7687 }
            - { name: NEO4J_USER, value: neo4j }
            - name: NEO4J_PASSWORD
              valueFrom: { secretKeyRef: { name: orca-secrets, key: NEO4J_PASSWORD } }
            - { name: QDRANT_URL, value: http://qdrant:6334 }
            - name: AGENTCHRON_INGEST_TOKEN
              valueFrom: { secretKeyRef: { name: orca-secrets, key:
AGENTCHRON_INGEST_TOKEN } }
          readinessProbe:
            httpGet: { path: /v1/health, port: 9474 }
            initialDelaySeconds: 10
          livenessProbe:
            httpGet: { path: /v1/health, port: 9474 }
            initialDelaySeconds: 30
          resources:
            requests: { cpu: "1", memory: 4Gi }
            limits: { memory: 12Gi }
          volumeMounts:
            - { name: data, mountPath: /var/lib/agentchron }
  volumeClaimTemplates:
    - metadata: { name: data }
```

```
      spec:
        accessModes: ["ReadWriteOnce"]
        resources: { requests: { storage: 100Gi } }
```

The sink StatefulSet has `replicas: 1` — the SQLite single-writer constraint enforced at the infrastructure level. The manifest includes the same SQLite tuning as the Compose file, the same secrets plugin, and the same Neo4j/Qdrant connection configuration. Secrets are loaded from a Kubernetes Secret, not from environment variables in the manifest.

WEB DEPLOYMENT

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  namespace: orca
spec:
  replicas: 2
  template:
    spec:
      containers:
        - name: web
          image: REGION-docker.pkg.dev/PROJECT_ID/orca/agentchron-web:REPLACE_TAG
          ports:
            - { name: http, containerPort: 9475 }
          env:
            - { name: AGENTCHRON_WEB_BIND, value: "0.0.0.0:9475" }
            - { name: AGENTCHRON_SINK_URL, value: http://sink:9474 }
            - { name: AGENTCHRON_WEB_MAX_BODY_BYTES, value: "268435456" }
            - name: AGENTCHRON_SINK_TOKEN
              valueFrom: { secretKeyRef: { name: orca-secrets, key:
AGENTCHRON_INGEST_TOKEN } }
          readinessProbe:
            httpGet: { path: /v1/health, port: 9475 }
            initialDelaySeconds: 5
          resources:
            requests: { cpu: "500m", memory: 512Mi }
            limits: { memory: 2Gi }
```

The web Deployment has `replicas: 2` — stateless, horizontally scalable. This is where the scale path diverges from the sink: you can add more web replicas to handle more concurrent API requests, but you cannot add more sink replicas. The web service proxies authenticated `/v1` requests to the private sink service in-cluster.

REQUIRED SECRETS

```
NEO4J_PASSWORD
AGENTCHRON_INGEST_TOKEN
ORCA_READ_TOKEN
ORCA_INGEST_TOKEN
ORCA_ADMIN_TOKEN
ORCA_ORIGIN_TOKEN
```

Secrets are created out-of-band and loaded into a Kubernetes Secret:

```
kubectl -n orca create secret generic orca-secrets \
  --from-literal=NEO4J_PASSWORD=*** \
  --from-literal=AGENTCHRON_INGEST_TOKEN=*** \
  --from-literal=ORCA_READ_TOKEN=*** \
  --from-literal=ORCA_INGEST_TOKEN=*** \
  --from-literal=ORCA_ADMIN_TOKEN=*** \
  --from-literal=ORCA_ORIGIN_TOKEN=*** \
  --dry-run=client -o yaml | kubectl apply -f -
```

Prefer a managed secret system (GCP Secret Manager, AWS Secrets Manager, Azure Key Vault) over raw `kubectl` commands for production automation. The GKE README documents the GitLab CI path with Infisical for secret management.

BRING-UP ORDER

```
Neo4j → Qdrant → sink → web → ingress / edge gateway
```

Stateful services first, then the stateless web layer, then the public ingress. The sink's `depends_on` in Compose mirrors this: Neo4j must be healthy and Qdrant must be started before the sink begins. In Kubernetes:

```
kubectl apply -f /tmp/orca.yaml
kubectl -n orca rollout status statefulset/neo4j
kubectl -n orca rollout status statefulset/qdrant
kubectl -n orca rollout status statefulset/sink
kubectl -n orca rollout status deployment/web
```

SIZING TIERS

TIER	SINK CPU/ RAM	SQLITE PVC	WEB REPLICAS	NEO4J	QDRANT	NOTES
Pilot	2 vCPU / 8 GiB	250 GiB	1	2 vCPU / 4 GiB, 100 GiB	2 vCPU / 4 GiB, 100 GiB	Single tenant, low ingest
Team	4 vCPU / 16 GiB	1 TiB	2	4 vCPU / 8 GiB, 250 GiB	4 vCPU / 8 GiB, 250 GiB	Good first customer baseline
Enterprise	8 vCPU / 32 GiB+	2–8 TiB	3+	8 vCPU / 16 GiB+, 500 GiB+	8 vCPU / 16 GiB+, 500 GiB+	Requires backup, audit, and query SLO review

Increase SQLite PVC before it reaches 70 percent usage. Alert before 80 percent. Treat 90 percent as an incident. These thresholds are not arbitrary — they are the points where SQLite WAL checkpoint behavior degrades and where the sink may start rejecting writes due to disk pressure.

PUBLIC INGRESS

The GKE manifest exposes only `/v1/*` on the web service through a managed certificate and global static IP:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: orca-ingress
  namespace: orca
  annotations:
    networking.gke.io/managed-certificates: orca-cert
    kubernetes.io/ingress.class: gce
    kubernetes.io/ingress.allow-http: "false"
    kubernetes.io/ingress.global-static-ip-name: orca-origin-ip
spec:
  rules:
    - host: orca.REPLACE.example.com
      http:
        paths:
          - path: /v1
            pathType: Prefix
            backend:
              service: { name: web, port: { number: 80 } }
```

The `BackendConfig` provides health checks:

```
apiVersion: cloud.google.com/v1
kind: BackendConfig
metadata:
  name: orca-web-backend
  namespace: orca
spec:
  healthCheck:
    type: HTTP
    requestPath: /v1/health
    port: 9475
  timeoutSec: 60
  connectionDraining:
    drainingTimeoutSec: 30
```

The browser never talks directly to Orca. Agent2600 proxies through its server route, which calls the Cloudflare Edge Worker with a scoped bearer token. The web service is the only public-facing component; the sink, Neo4j, and Qdrant are all cluster-internal.

GITLAB CI DEPLOYMENT

The GKE README documents the preferred deployment path through GitLab CI:

Required protected GitLab CI variables:

```
INFISICAL_CLIENT_ID
INFISICAL_CLIENT_SECRET
```

Orca deploy variables are pulled from Infisical at job start:

```
project: aba855ba-2f22-4999-a582-71b5ed1bdc2c
env:     prod
path:    /
```

Required Infisical variables:

```
GCP_PROJECT_ID
GCP_REGION
GKE_CLUSTER
GCP_ARTIFACT_REPOSITORY
ORCA_DOMAIN
NEO4J_PASSWORD
AGENTCHRON_INGEST_TOKEN
GCP_SERVICE_ACCOUNT_KEY_B64   # optional if runner has Workload Identity
```

The CI pipeline uses `scripts/ci/load-infisical-env.sh` to authenticate with Infisical by universal auth, export scoped variables as dotenv, source them in the current job, then delete the temporary dotenv file.

For a one-shot deploy from a workstation, the GKE README provides a 10-step procedure: set environment values, create Artifact Registry, build and push immutable images, get cluster credentials, reserve a global IP, render the manifest, create namespace and secrets, apply Orca, point DNS at the reserved IP, and smoke the origin.

Cloudflare Edge Handoff

Cloudflare Edge is the public gateway and token boundary. It is not the system of record — the origin remains the system of record. The GKE README describes the ship-now shape:

```
agent2600 / PlatformOS clients
  → Cloudflare Orca Edge Worker
  → GKE Orca origin
       → agentchron-web Deployment
       → agentchron-sink StatefulSet + SQLite PVC
       → neo4j StatefulSet + PVC
       → qdrant StatefulSet + PVC
```

The Worker validates read/ingest/admin/origin tokens, optionally buffers via R2/Queue, and forwards to the private origin over HTTPS. Required Worker secrets:

```
ORCA_READ_TOKEN
ORCA_INGEST_TOKEN
ORCA_ADMIN_TOKEN
ORCA_ORIGIN_TOKEN
```

Required Worker config:

```
ORCA_ORIGIN_URL=https://<customer-orca-origin>
```

Optional bindings:

```
ORCA_RAW_ARCHIVE  # R2 raw/sanitized archive
ORCA_EVENTS       # queue buffering
```

For private enterprise installs, Cloudflare can be replaced with a customer ingress, API gateway, private load balancer, or service mesh. The token contract and origin constraints remain the same. The edge is replaceable; the origin is not.

EDGE DEPLOYMENT

```
cd cloudflare/orca-edge
npx wrangler secret put ORCA_ORIGIN_TOKEN
# Set ORCA_ORIGIN_URL in wrangler.toml or Worker environment
npx wrangler deploy
```

The Cloudflare Edge Worker serves as the public gateway and token boundary. It validates read, ingest, admin, and origin tokens before forwarding to the private origin. For burst protection, the Worker can optionally buffer requests through Cloudflare Queue or archive raw payloads to R2. This buffering is not the system of record — the origin remains the system of record — but it absorbs traffic spikes that exceed the sink's write capacity.

The edge handoff is documented in `docs/ORCA_CLOUDFLARE_HANDOFF.md`. The Worker code lives in `cloudflare/orca-edge/`. The Worker is stateless — it validates tokens, optionally buffers, and forwards. All state is in the origin. This means the Worker can be deployed across Cloudflare's global edge network without coordination, and a Worker failure does not lose data (the origin still has it).

PRIVATE ENTERPRISE INGRESS

For private enterprise installs, Cloudflare can be replaced with a customer ingress, API gateway, private load balancer, or service mesh. The token contract and origin constraints remain the same:

- Only `/v1/*` is exposed.
- The sink, SQLite, Neo4j, and Qdrant are private.
- Split tokens (read/ingest/admin/origin) are enforced at the edge.
- The browser never receives a bearer token.

A customer using AWS can replace Cloudflare with an ALB + API Gateway combination. A customer using Azure can use Application Gateway. A customer using on-prem Kubernetes can use an NGINX Ingress or a service mesh like Istio. The Orca manifest does not mandate Cloudflare — it mandates the token contract and origin constraints. Cloudflare is the reference implementation; the customer's ingress is the production implementation.

BACKFILL CONSIDERATIONS

The GKE README warns:

> The `.114` lab/source-of-truth database snapshot currently lives under `/mnt/backups05/agentchron-db-snapshots/`. The first production cut should restore the latest sanitized SQLite snapshot onto the sink PVC before switching SaaS clients to Cloudflare Edge. Do not copy raw unfiltered source logs into the production PVC.

GCP Lift-Shift and Multi-Cloud

The checked-in manifest is GKE-named but Kubernetes-shaped. For AWS (EKS) or Azure (AKS), update:

CHANGE	AWS (EKS)	AZURE (AKS)
Storage class	EBS gp3/io2	Azure Disk
Ingress annotations	ALB	Application Gateway
Image registry	ECR	ACR
Service account	IAM + OIDC	Managed Identity
NetworkPolicy	Calico/Cilium	Azure Network Policies
Secrets	Secrets Manager	Key Vault
Archive backup	S3	Blob Storage
Postgres readiness	RDS Postgres	Azure Database for PostgreSQL

On-prem deployments use Kubernetes or Docker Compose with customer PKI/TLS, SAN/NAS snapshots, and private NATS only.

CLOUD VM COMPOSE VALIDATION

A shortcut for validating the package without touching the lab network: spin up a single VM in a fresh VPC, deploy Compose, run smoke tests, and destroy. Rules: no lab peering, SSH-only from operator IP, throwaway secrets, destroy after validation. This is a quick way to verify that the SMB Compose bundle works on a clean host without lab assumptions.

TERRAFORM + GITHUB ACTIONS

The preferred repeatable enterprise path uses Terraform for infrastructure provisioning and GitHub Actions for build/package/deploy. The split is clean: Terraform provisions the infrastructure (VPC, cluster, node pools, PVCs, DNS, TLS certificates); GitHub Actions builds the images, pushes them to the registry, renders the manifest, and applies it. Customer secret manager holds all secret values — no secrets in GitHub Actions secrets, no secrets in Terraform state.

Use GitHub OIDC federation — no long-lived cloud keys in GitHub secrets. OIDC federation lets GitHub Actions assume a cloud role on-demand using short-lived tokens, rather than storing a long-lived service account key. This is the same principle as the Brain Bundle's `source_creds_present: false` (Chapter 11) — credentials do not travel; identity is re-bound at the target.

Validation environments are always isolated, throwaway, and destroyed after smoke. The cloud VM Compose validation is the simplest version of this: spin up a VM, deploy, smoke, destroy. The Terraform + Actions path is the enterprise version: spin up a namespace, deploy, smoke, destroy. Neither touches the lab network.

KNOWN PRODUCT GAPS FOR LARGE SAAS

The scale guide documents product gaps that are not install blockers for controlled customer deployments but must be closed for broad multi-tenant SaaS scale:

- No self-service tenant control plane — tenants are currently onboarded through manual configuration.
- No explicit token expiry/revocation enforcement yet — offboarding requires secret rotation and redeployment.
- No storage-layer port to managed multi-writer database — the sink is SQLite single-writer.
- No first-class Prometheus metrics endpoint — observability relies on logs and platform probes.
- No complete semantic retrieval lane — Qdrant is wired but the embedding model is a later milestone.
- No automated gold promoter for reviewed workflows — gold promotion requires manual review.
- No customer-facing backup/restore dashboard — restore drills are manual.
- No formal load-test report per tier — sizing tiers are starting points, not guarantees.

These gaps are tracked in `docs/ORCA_PLATFORM_GAP_ANALYSIS.md` and `docs/ORCA_PLATFORM_GAP_IMPLEMENTATION_SPEC.md`. They inform the 2027 roadmap (Chapter 14) — several of them map directly to EMPIRE tickets in the build order.

PostgreSQL Readiness

PostgreSQL is the future multi-writer target, but the sink adapter is not a drop-in replacement. The current production path is SQLite single-writer.

`deploy/docker-compose.postgres.example.yml` exists for readiness validation only. The environment variable `ORCA_DATABASE_KIND=postgres` is reserved — do not set it in production until release notes say the adapter is enabled.

The GKE README is explicit about this constraint:

> Today `agentchron-sink` is SQLite-backed. That makes the sink stateful and single-writer. Moving the sink to Neon or running the full stack inside Cloudflare Containers is not a config change; it is a storage-layer port to Postgres, Turso/libSQL, or another managed store.

The future track is:

1. Port `agentchron-sink` from SQLite to Postgres or Turso/libSQL.
2. Externalize graph/vector state to managed Neo4j/Qdrant equivalents or a CF-native replacement.
3. Make web/sink stateless enough for Cloudflare Containers.

That work is non-blocking for the GKE+PVC production launch. The SQLite single-writer path is production-ready today for controlled customer deployments. The Postgres adapter is the path to broad multi-tenant SaaS scale.

Disk Alerts and Operational Runbook

DISK ALERTS

THRESHOLD	ACTION
70%	Warning — plan PVC expansion
80%	Urgent — expand PVC or reduce retention
90%	Incident — immediate action, risk of write failures

SQLITE PVC SNAPSHOT AND RESTORE DRILL

A snapshot + restore drill is a go-live acceptance criterion, not a recommendation:

1. Restore SQLite snapshot into an isolated namespace or host.
2. Start sink read-only or isolated from production ingress.
3. Verify `/v1/health`.
4. Verify `/v1/stats`.
5. Fetch a known session.
6. Run source coverage for one sampled source path.
7. Document snapshot ID, command history, elapsed time, and result.

Do not run schema-altering upgrades without a fresh SQLite snapshot. The upgrade procedure is:

1. Review release notes and migration notes.
2. Snapshot SQLite PVC.
3. Snapshot Neo4j/Qdrant or confirm they can be rebuilt.
4. Deploy immutable image tags to staging.
5. Run smoke tests.
6. Run tenant negative tests.
7. Run audit query.
8. Deploy production during an approved window.
9. Watch logs, 4xx/5xx, ingest accepted/rejected counts, disk, and audit.
10. Keep rollback image tags and last good volume snapshot available.

BACKUP LANES

Orca customer installs need three backup lanes:

1. **SQLite source-of-truth snapshots** — persistent block storage, volume snapshots on a schedule aligned with customer RPO, copies to a separate storage account, verified restore into an isolated environment.
2. **Raw/sanitized source archive, if enabled** — SHA-256 manifests, second copy outside the primary disk, customer-approved retention and deletion policy.
3. **Neo4j/Qdrant rebuild or backup strategy** — these are sidecars that can be rebuilt from SQLite, but a backup strategy reduces rebuild time.

OPERATIONAL PROHIBITIONS

The scale guide repeats these constraints across DEPLOY.md, the customer install guide, and the deployment guide because they are the most likely operator mistakes:

1. **Do not route production customers to the `.114` lab host**, home LANs, or developer-only tunnels.
2. **Do not expose the sink directly to browsers.** Browsers talk to `agentchron-web`, not the sink.
3. **Do not give browser code an Orca bearer token.** The web proxy handles token injection internally.
4. **Do not run more than one writable `agentchron-sink` against the same SQLite database.** This corrupts the database.
5. **Do not store `AGENTCHRON_DB_PATH` on tmpfs**, ephemeral disk, or unreliable network storage.
6. **Do not copy raw unsanitized customer logs into production storage** unless the customer contract explicitly allows raw retention and the raw archive is encrypted, access-controlled, and separately governed.
7. **Do not use the legacy all-scope token for new customers.** Disable it with `AGENTCHRON_LEGACY_TOKEN_ENABLED=*** **Do not deploy` latest `image tags in production.** Use immutable image tags (`).

OBSERVABILITY

Minimum operational checks:

```
GET /v1/health
GET /v1/stats
GET /v1/audit?limit=20
GET /v1/sources/coverage
container logs for sink/web
PVC usage
SQLite file size and WAL size
Neo4j health
Qdrant health
ingress 4xx/5xx rate
Cloudflare Worker error rate, if used
```

Alert on:

- sink or web unavailable
- SQLite PVC over 70/80/90 percent
- WAL growth without checkpoint progress
- sustained HTTP 500s
- sustained HTTP 401/403 spikes
- ingest rejection spike
- audit write failures
- backup failure
- restore drill overdue

Current limitation: Prometheus-native metrics are not yet first-class in this repo. Use logs, platform probes, and gateway metrics until metrics endpoints are added. This is a known product gap tracked in the platform gap implementation spec.

PERFORMANCE GUIDANCE

For high ingest:

- Send batches instead of single events when possible.
- Keep body size below configured gateway limits (256 MB default).
- Use Cloudflare Queue or customer queue buffering if bursts exceed sink write capacity.
- Avoid expensive exact counts in MCP unless required.
- Use tenant/workspace filters on large search queries.
- Keep SQLite cache and mmap sized for the node.
- Run large historical backfills outside peak customer query windows.

Token Architecture

The lab uses a single `AGENTCHRON_INGEST_TOKEN` as a legacy all-scope token. Production deployments require split tokens:

TOKEN	ALLOWED SCOPE
`ORCA_READ_TOKEN`	Read sessions, stats, search, alerts, workflows, library, graph context
`ORCA_INGEST_TOKEN`	Ingest events only
`ORCA_ADMIN_TOKEN`	Read plus workflow/library mutations
`ORCA_ORIGIN_TOKEN`	Internal edge/origin token with read, ingest, and admin scopes
`AGENTCHRON_INGEST_TOKEN`	Legacy compatibility token with all scopes (disable for new customers)

Admin workflow/library mutations write an `api_audit_event` row with the token class, action, route, status, and details. Token policy strings may include `expires_at=<RFC3339>` or `expires_at=<YYYY-MM-DD HH:MM:SS>`; expired tokens are rejected before route-scope checks. Setting any non-empty `revoked_at=...` value revokes that configured token immediately and records the lifecycle value in `auth_token_metadata`.

TOKEN ROTATION PROCEDURE

1. Stop or pause remote ingest agents that cannot read `.114`'s `.env`.
2. Edit `AGENTCHRON_INGEST_TOKEN` and any configured `ORCA_*_TOKEN` values in `deploy/.env`.
3. To stage a decommissioned token, keep the token value only long enough to drain clients and add `revoked_at=<timestamp>` to that token's `*_SCOPE`.
4. Recreate `sink` and `web` with `docker compose up -d sink web`.
5. Restart `.114` user services: `systemctl --user restart agentchron-114-live.service agentchron-fleet-pull.service`.
6. Update any off-host agent configs that still carry the shared token.

Keep the deploy secret file private:

```
chmod 700 /home/developer/Projects/agentchron/deploy
chmod 600 /home/developer/Projects/agentchron/deploy/.env
```

Patterns and Anti-Patterns

PATTERNS DEVELOPED IN THIS CHAPTER

Pattern 5: SQLite WAL as Source of Truth. The sink is SQLite-backed with WAL mode, tuned cache (2 GB), mmap (8 GB), and single-writer constraint. The database is the system of record; Neo4j and Qdrant are best-effort sidecars. Scale by adding web replicas and edge capacity, not by replicating the sink. The future multi-writer path is a storage-layer migration to Postgres — not a config change. This pattern appeared first in Chapter 3 (sink) and is reinforced here at the deployment layer. The GKE manifest enforces it at the infrastructure level with `replicas: 1` on the sink StatefulSet.

Pattern 4: Forward-Compatible Schema. The same Docker Compose file runs on a laptop and in production. The GKE manifest is Kubernetes-shaped and portable to EKS/AKS with storage class and ingress updates. The Dockerfiles pin the Rust toolchain by base image, not by `rust-toolchain.toml`, to ensure repeatable offline builds. The deployment artifacts are versioned and reproducible. This pattern connects to the Brain Bundle's `schema_version` field (Chapter 11) — the same principle of versioned, forward-compatible formats applied to deployment manifests.

Pattern 12: Reproducibility Envelope. Every deployment artifact carries immutable image tags, SHA-256 checksums, and build metadata. The GKE README documents the exact render-and-apply procedure with `perl -pi` replacements for image tags, domains, and project IDs. The cloud VM Compose validation is a throwaway environment destroyed after smoke. Terraform + GitHub Actions with OIDC federation eliminates long-lived cloud keys. This pattern runs from the Data Foundry's run manifests (Chapter 10) through the installer's versioned tarballs (Chapter 13) to the deployment manifests.

ANTI-PATTERNS ADDRESSED

Anti-Pattern 7: Premature Horizontal Scaling of the Writable Sink. The sink is SQLite-backed and single-writer. Running multiple writable replicas against the same DB corrupts it. The scale path is Cloudflare Edge + web replicas, not sink replicas. The docs repeat this constraint across DEPLOY.md, the scale guide, and the deployment guide because it is the most likely operator mistake. The GKE manifest enforces it at the infrastructure level: `replicas: 1` on the sink StatefulSet. The Compose file enforces it by design — there is only one sink service. The Postgres adapter is the future multi-writer path, but it is a storage-layer migration, not a config change.

Anti-Pattern 1: Single Shared Token for All Auth Roles. The GKE manifest requires split tokens (read/ingest/admin/origin). The scale guide requires disabling the legacy all-scope token for new customers. The origin sink does not yet fully enforce split tokens — this is a known product gap tracked in the platform gap implementation spec. But the deployment manifests and the edge Worker are built for split tokens from day one. The token rotation procedure and the `api_audit_event` audit trail are designed for split tokens. When the sink enforcement lands, the deployment manifests will already be correct.

Conclusion

Orca's deployment story is deliberately composable. The same four-service stack — sink, web, Neo4j, Qdrant — runs on a developer laptop via Docker Compose, on a customer's Kubernetes cluster via the GKE manifest, and behind a Cloudflare Edge Worker for SaaS. The differences are in sizing, secrets, and ingress, not in code. The SQLite single-writer constraint is the most important operational rule: scale by adding web replicas and edge capacity, not by replicating the sink. The PostgreSQL adapter is the future multi-writer path, but it is a storage-layer migration, not a config change.

The disk alerts, the snapshot/restore drill, the token rotation procedure, and the operational prohibitions are not bureaucratic overhead — they are the minimum acceptance criteria for a deployment that handles real customer

data. A deployment that has not run a restore drill is not production-ready. A deployment that routes production customers to the lab host is a security incident waiting to happen. A deployment that uses the legacy all-scope token for a new customer is one token leak away from a data breach. The deployment guide exists to make these rules explicit and repeatable.

Chapter 13: Installers and Fleet

Deployment gets the origin running. Installers get the edge running — on every developer workstation, in every customer environment, at every scale. This chapter details the installer infrastructure: the Linux and macOS bootstrap scripts, the per-OS workstation client packages, the customer install models, the tenant scope model with scoped tokens, and the GTM SKUs. The goal is simple to state and hard to achieve: a one-command install that establishes full capture and guard posture on a fresh workstation, with smoke tests that prove it worked.

The installer is the last mile of the platform. A platform that captures, sanitizes, stores, refines, and serves governed knowledge is useless if it cannot be installed on a fresh workstation with one command and proven to work. The installer infrastructure is what makes Orca deployable at fleet scale.

Linux Installer: The Fleet Bootstrapper

`installers/agentchron-install-linux.sh` is a bash bootstrap. It does not contain the install logic itself — it installs from a versioned client tarball, then delegates all file writes to the packaged `install.sh` inside the tarball. This separation lets the bootstrap script stay small and stable while the install logic ships with the package it installs.

STRUCTURE

```
#!/usr/bin/env bash
# Orca/AgentChron client installer bootstrap for Linux.
#
# This is an internal fleet bootstrapper. It installs from a versioned client
# tarball, then delegates all file writes to the packaged install.sh.

set -euo pipefail

VERSION="${AGENTCHRON_VERSION:-0.1.0}"
SINK_HOST="${AGENTCHRON_TCP_HOST:-<lab-host>}"
SINK_PORT="${AGENTCHRON_TCP_PORT:-39478}"
SINK_URL="${AGENTCHRON_SINK_URL:-}"
SINK_SSH_USER="${AGENTCHRON_SSH_USER:-developer}"
SINK_REPO="${AGENTCHRON_SINK_REPO:-/home/developer/Projects/agentchron}"
AGENT="${AGENTCHRON_AGENT:-}"
HOST_ID="${AGENTCHRON_HOST_ID:-$(hostname -s 2>/dev/null || hostname 2>/dev/null || printf
unknown-linux)}"
BIN_DIR="${BIN_DIR:-$HOME/.local/bin}"
CONFIG_DIR="${CONFIG_DIR:-$HOME/.config/agentchron}"
CLAUDE_SETTINGS="${CLAUDE_SETTINGS:-$HOME/.claude/settings.json}"
ORCA_VISIBILITY="${ORCA_VISIBILITY:-${AGENTCHRON_VISIBILITY:-private}}"
ORCA_PURPOSE="${ORCA_PURPOSE:-${AGENTCHRON_PURPOSE:-claude-code-capture}}"
CONFIGURE_CLAUDE=1
WRITE_ENV=1
CHECK_SINK=0
DRY_RUN=0

log() { printf '[orca-install] %s\n' "$*"; }
die() { printf '[orca-install] ERROR: %s\n' "$*" >&2; exit 1; }
```

The `set -euo pipefail` is non-negotiable. The `-e` flag exits on any error. The `-u` flag treats unset variables as errors. The `-o pipefail` flag catches failures in piped commands. Together they ensure the installer fails loudly and immediately on any problem, rather than continuing with a partially-correct state.

The `log()` and `die()` helpers ensure consistent output and immediate failure on any error. Every message is prefixed with `[orca-install]` so the user can distinguish installer output from other commands in a pipeline.

The defaults point at the internal lab sink (`<lab-host>:39478` — host port 39478, container port 9478) but every value is overridable through environment variables or command-line flags. The default visibility is `private` — Pattern 6, Private-by-Default Scoping, enforced at install time. The default purpose is `claude-code-capture`, which tags every event with the correct purpose scope from the first push.

PACKAGE ACQUISITION

The installer supports three package acquisition modes:

FLAG	METHOD	USE CASE
`--package-file PATH`	Local tarball	Pre-downloaded or USB-delivered
`--package-url URL`	HTTP download with curl	Public or private registry
`--package-ssh SPEC`	SCP from a host	Internal fleet distribution

If none is specified, the installer searches for a local candidate and falls back to SCP from the sink host:

```
if [ -z "$PACKAGE_URL" ] && [ -z "$PACKAGE_SSH" ]; then
  for candidate in \
    "$script_dir/../dist/$default_name" \
    "$HOME/Projects/agentchron/dist/$default_name"; do
    if [ -f "$candidate" ]; then
      PACKAGE_FILE="$candidate"
      fetch_package "$target" "$out"
      return
    fi
  done
  PACKAGE_SSH="$SINK_SSH_USER@$SINK_HOST:$SINK_REPO/dist/$default_name"
fi
```

The default tarball name includes the version and target triple: `agentchron-client-0.1.0-x86_64-unknown-linux-gnu.tar.gz`. This naming convention ensures that the installer can find the correct package for the host architecture without guessing.

ARCHITECTURE DETECTION

```
detect_target() {
  case "$(uname -m)" in
    x86_64|amd64) printf 'x86_64-unknown-linux-gnu' ;;
    aarch64|arm64) printf 'aarch64-unknown-linux-gnu' ;;
    *) die "unsupported Linux architecture: $(uname -m)" ;;
  esac
}
```

The installer detects the target triple from `uname -m` and refuses to run on unsupported architectures. This prevents silent failures when someone tries to install an x86_64 binary on an ARM host or vice versa. The OS check is explicit:

```
[ "$(uname -s)" = "Linux" ] || die "not Linux: use agentchron-install-mac.sh on macOS"
```

SHA-256 VERIFICATION

The installer verifies package authenticity before extracting:

```
if [ -n "$PACKAGE_SHA256" ]; then
  actual="$(sha256_file "$PACKAGE_PATH")"
  [ "$actual" = "$PACKAGE_SHA256" ] || die "sha256 mismatch: expected $PACKAGE_SHA256 got
$actual"
elif [ -f "$PACKAGE_PATH.sha256" ]; then
  verify_sidecar "$PACKAGE_PATH" "$PACKAGE_PATH.sha256"
else
  log "no sha256 supplied; install is allowed but package authenticity was not verified"
fi
```

The verification logic has three tiers:

1. **Explicit hash** (`--package-sha256` or `AGENTCHRON_PACKAGE_SHA256`): The caller provides the expected hash. If it does not match, the install dies. This is the strongest verification — the hash is provided out-of-band, so a tampered tarball cannot pass.
2. **Sidecar file** (`tarball.sha256`): The installer looks for a `.sha256` file next to the tarball. If present, it verifies using `shasum -a 256 -c` or `sha256sum -c` . This is convenient for distribution — the hash ships with the package.
3. **No hash**: The install proceeds but logs a warning. This is allowed for development convenience but not recommended for production. The warning is explicit: "package authenticity was not verified."

The SHA-256 verification uses whichever tool is available:

```
sha256_file() {
  if command -v shasum >/dev/null 2>&1; then
    shasum -a 256 "$1" | awk '{print $1}'
  elif command -v sha256sum >/dev/null 2>&1; then
    sha256sum "$1" | awk '{print $1}'
  else
    die "shasum or sha256sum is required"
  fi
}
```

TOKEN HANDLING

The installer handles the ingest token carefully to avoid leaking it:

```
if [ "$WRITE_ENV" -eq 1 ] && [ -z "${AGENTCHRON_SINK_TOKEN:-}" ]; then
  [ -t 0 ] || die "set AGENTCHRON_SINK_TOKEN for non-interactive install"
  printf 'AGENTCHRON_SINK_TOKEN (paste, will not echo): '
  stty -echo
  read -r AGENTCHRON_SINK_TOKEN
  stty echo
  printf '\n'
  export AGENTCHRON_SINK_TOKEN
  [ -n "$AGENTCHRON_SINK_TOKEN" ] || die "token required"
fi
```

The token is never passed on the command line — it goes through the environment to `install.sh`. This prevents the token from appearing in process listings or shell history. If the terminal is not interactive (`[ -t 0 ]` fails), the installer requires `AGENTCHRON_SINK_TOKEN` to be set in the environment, preventing interactive prompts in CI or batch contexts.

DELEGATION TO INSTALL.SH

After verification, the bootstrap extracts the tarball and delegates to the packaged `install.sh`:

```
tar -tzf "$PACKAGE_PATH" >/dev/null
tar -xzf "$PACKAGE_PATH" -C "$WORK_DIR"
PACKAGE_ROOT="$(find "$WORK_DIR" -maxdepth 1 -type d -name 'agentchron-client-*' | head -n
1)"
[ -n "$PACKAGE_ROOT" ] && [ -x "$PACKAGE_ROOT/install.sh" ] || die "package missing
install.sh"

install_args=(
  --bin-dir "$BIN_DIR"
  --config-dir "$CONFIG_DIR"
  --sink-host "$SINK_HOST"
  --sink-port "$SINK_PORT"
  --host-id "$HOST_ID"
  --visibility "$ORCA_VISIBILITY"
)
[ -n "$SINK_URL" ] && install_args+=(--sink-url "$SINK_URL")
[ -n "$AGENT" ] && install_args+=(--agent "$AGENT")
[ -n "$ORCA_ORG" ] && install_args+=(--org "$ORCA_ORG")
[ -n "$ORCA_TEAM" ] && install_args+=(--team "$ORCA_TEAM")
[ -n "$ORCA_WORKSPACE" ] && install_args+=(--workspace "$ORCA_WORKSPACE")
[ -n "$ORCA_PURPOSE" ] && install_args+=(--purpose "$ORCA_PURPOSE")
[ "$CONFIGURE_CLAUDE" -eq 1 ] && install_args+=(--configure-claude)
[ "$WRITE_ENV" -eq 0 ] && install_args+=(--no-env)

CLAUDE_SETTINGS="$CLAUDE_SETTINGS" "$PACKAGE_ROOT/install.sh" "${install_args[@]}"
```

The bootstrap passes all resolved scope flags through to `install.sh`. The `CLAUDE_SETTINGS` path is passed through the environment, not as an argument, to keep it out of the process listing. The `--configure-claude` flag tells `install.sh` to edit the Claude Code settings file to add the capture and Guard hooks.

POST-INSTALL OUTPUT

```
Install complete.
Agent:    neo
Host ID:  <dev-vm>
Sink:     <lab-host>:39478
Binaries: /home/developer/.local/bin
Config:   /home/developer/.config/agentchron/push.env

Restart Claude Code, run a tool call, then query Orca/AgentChron for agent=neo.
```

The post-install message tells the user exactly what to do next: restart Claude Code, run a tool call, and verify that the event reached the sink. This is not just a friendly message — it is the first step of the acceptance test.

SCOPE FLAGS

The installer supports Orca scope flags that map directly to the tenant scope model:

```
--org NAME              Orca org scope
--team NAME             Orca team scope
--workspace NAME        Orca workspace scope
--visibility VALUE      Orca visibility scope, default private
--purpose VALUE         Orca purpose scope, default claude-code-capture
```

These flags ensure that every event captured on this workstation is tagged with the correct tenant, team, workspace, visibility, and purpose from the moment the agent starts pushing. This is critical for multi-tenant deployments — an event without scope tags is either rejected by the sink (if the ingest token has a scope that conflicts) or stamped with the token's scope (if the token has a scope and the event does not). Getting the scope right at install time prevents both rejections and misattributed events.

Per-OS Workstation Client Packages

The GTM packaging spec (`docs/ORCA_GTM_DATA_FACTORY_PACKAGING_SPEC.md`) defines five build targets:

OS	TARGET TRIPLE	INSTALL MODEL	PERSISTENT CAPTURE
Linux x86_64	`x86_64-unknown-linux-gnu` and/or `x86_64-unknown-linux-musl`	tarball + `install.sh` ; optional bootstrap `.run`	user systemd service
Linux arm64	`aarch64-unknown-linux-gnu`	tarball + `install.sh`	user systemd service
macOS Apple Silicon	`aarch64-apple-darwin`	tarball + `install.sh`	LaunchAgent
macOS Intel	`x86_64-apple-darwin`	tarball + `install.sh`	LaunchAgent
Windows x86_64	`x86_64-pc-windows-msvc`	zip + PowerShell installer	Scheduled Task or Windows service

The `musl` variant for Linux x86_64 is important for air-gapped and minimal-container environments where glibc is not available or is a different version. The `gnu` variant is the default for most Linux distributions.

REQUIRED BINARIES

Every workstation package must include:

```
agentchron-agent          # local watcher, remote pull, one-shot backfill
agentchron-push           # TCP push transport with local sanitization
agentchron-claude-hook    # Claude Code capture hook
agentchron-sanitize       # standalone stdin→stdout sanitizer
orca-guard                # secret gate (PreToolUse, UserPromptSubmit)
orca-mcp-gateway          # MCP governance gateway
orca-session-rules.py     # GraphRAG context injection hook
orca-gitlog-ingest.py     # git log ingestion utility
orca-graphrag-ingest.py   # GraphRAG ingest utility
agentchron-ingest-healthcheck.sh  # health check script (or Windows equivalent)
install/uninstall scripts
SHA-256 sidecar and package manifest
```

This is not a minimal capture client. It is the full edge posture: capture, push, Guard, MCP gateway, GraphRAG hooks, and health checks. The install is not complete until all of these are present and smoke-tested. The packaging spec is explicit: "All binaries return `--help` or version output" is the first acceptance check.

AI CLIENT SUPPORT

The GTM package defines support levels for different AI clients:

CLIENT	GTM STATUS	ENFORCEMENT/CAPTURE
Claude Code	Required	Capture + `UserPromptSubmit` + `PreToolUse` Guard + Claude RBAC/allowed-tools + MCP gateway where configured
Codex	Required capture	Session capture + MCP gateway where client supports MCP config
Antigravity/AGY	Required capture	Session/log capture; MCP enforcement only after transport is verified
Gemini CLI	Legacy/ deprecated	Do not make it the forward install target; support best-effort capture only
Cursor/VS Code	Later add-on	Do not block GTM; document as extension target
OpenClaw/ ArmyknifeClaw	Later add-on	Not required for first GTM package

Claude Code is the primary target because it has the richest hook system — `UserPromptSubmit`, `PreToolUse`, `PostToolUse`, `SessionStart`, `SessionEnd`, `Stop`. The installer configures all of these hooks in `~/.claude/settings.json`.

MACOS INSTALLER

The macOS installer (`installers/agentchron-install-mac.sh`) mirrors the Linux installer with three differences:

1. **Architecture detection** maps to Apple Silicon or Intel: `bash detect_target() { case "$(uname -m)" in arm64|aarch64) printf 'aarch64-apple-darwin' ;; x86_64|amd64) printf 'x86_64-apple-darwin' ;; *) die "unsupported macOS architecture: $(uname -m)" ;; esac }`
2. **OS check** refuses to run on Linux: `bash [ "$(uname -s)" = "Darwin" ] || die "not macOS: use agentchron-install-linux.sh on Linux"`

3. **Local repo fallback** uses `AGENTCHRON_LOCAL_REPO` instead of the sink-hosted repo, because macOS hosts typically have a local clone rather than SSH access to the sink: `bash LOCAL_REPO="${AGENTCHRON_LOCAL_REPO:-$HOME/Projects/agentchron}"`

Beyond these differences, the macOS installer is structurally identical to the Linux installer: same `set -euo pipefail`, same `log()` / `die()` helpers, same SHA-256 verification, same delegation to `install.sh`, same scope flags. The only difference in the `--check-sink` flag is the `nc` timeout flag: Linux uses `-w 3` (timeout), macOS uses `-G 3` (connection timeout).

WINDOWS TARGET

Windows is a required target but follows a different install model: zip + PowerShell installer with a Scheduled Task or Windows service for persistent capture. The same binaries are required (cross-compiled with `x86_64-pc-windows-msvc` target); the persistence mechanism differs. The packaging spec lists Windows as a required build target, and the release artifacts include `dist/agentchron-client-<version>-x86_64-pc-windows-msvc.zip`. If code signing is available, Windows binaries should be Authenticode signed.

RELEASE ARTIFACTS

Each release produces:

```
dist/platform/orca-data-factory-<version>-smb.tar.gz
dist/platform/orca-data-factory-<version>-enterprise.tar.gz
dist/platform/orca-data-factory-<version>-airgap.tar.gz
dist/agentchron-client-<version>-x86_64-unknown-linux-gnu.tar.gz
dist/agentchron-client-<version>-aarch64-apple-darwin.tar.gz
dist/agentchron-client-<version>-x86_64-apple-darwin.tar.gz
dist/agentchron-client-<version>-x86_64-pc-windows-msvc.zip
```

Every artifact needs a SHA-256 sidecar, manifest JSON, build timestamp, git commit, channel, included component list, and smoke result summary. If code signing is available: macOS Developer ID signing/notarization for binaries or package wrapper, Windows Authenticode signing, Linux detached signatures where available.

Workstation Acceptance Criteria

A workstation install is not complete until all acceptance checks pass. These are not optional — they are the definition of done. The packaging spec defines seven checks:

CHECK	EXPECTED RESULT	WHAT IT VALIDATES
All binaries `--help`	Each binary returns help or version output	Binaries are on PATH and executable
Safe prompt passes Guard	`orca-guard claude-hook` allows a benign prompt	Guard is installed and not blocking everything
Fake provider key blocked	A synthetic `sk-ant...ard-` key is blocked; the fake value is not echoed	Guard is catching secrets and not leaking them in error output
Env-var references allowed	`process.env.NAME` and `import.meta.env.NAME` references pass	Guard's allow-list is correct — code references are not false positives
Direct secret-file read blocked	`cat ~/.ssh/id_ed25519` in a `PreToolUse` hook is blocked	Guard catches direct secret-file reads in tool calls
Synthetic session reaches sink	A test event appears in `/v1/events` or `/v1/sessions`	Push transport, token, and sink are all wired correctly
Capture survives reboot	After reboot, the systemd service or LaunchAgent restarts capture	Persistence is configured correctly

GUARD SMOKE TESTS

The Guard smoke tests are concrete and runnable:

```
# Safe prompt — should pass (exit 0)
printf '{"hook_event_name":"UserPromptSubmit","prompt":"hello world"}' \
  | orca-guard claude-hook
echo $?  # 0

# Fake provider key — should block (exit 2)
CANARY="$(python3 -c 'import uuid; print("sk-ant...ard-" + uuid.uuid4().hex)')"
printf '{"hook_event_name":"UserPromptSubmit","prompt":"use %s"}' "$CANARY" \
  | orca-guard claude-hook
echo $?  # 2

# Direct secret-file read — should block (exit 2)
printf '%s' '{"hook_event_name":"PreToolUse","tool_name":"Bash","tool_input":
{"command":"cat ~/.ssh/id_ed25519"}}' \
  | orca-guard claude-hook
echo $?  # 2
```

The fake provider key test is critical: the key must be blocked, and the fake value must not be echoed in the output. If the Guard echoes the secret it was supposed to block, it has created a new exfiltration path — the error message itself would contain the secret. The test verifies both the block and the non-echo.

GUARD MODES

The installer can configure Guard in four modes:

MODE	USE	BLOCKS
`production`	Default for production, customer data, shared hosts	Hard-blocks secret-shaped values, direct secret-content reads, and secret-manager reads; asks for human approval on gray-area path-only `PreToolUse` references
`dev`	App builds and local development where agents need to discuss env vars and paths	Secret-shaped values and direct secret-content reads
`audit`	QA/report-only mode	Allows the action, but reports what Production Mode would hard-block or require approval for
`max`	Enterprise managed workstation	Strictest — all production blocks plus additional enterprise policy

Allow-list behavior is intentionally narrow. Mode is configurable, but broad local "allow this secret path" files are not supported because they would let a workstation bypass the control. Current built-in exceptions cover template files such as `.env.example`, source-code env identifiers such as `process.env.NAME` / `import.meta.env.NAME`, selected path-only references, and approved deploy-time env-file loaders that do not dump the environment. Direct secret-content reads and real secret-shaped values remain hard blocks.

WHY THESE TESTS MATTER

Each test catches a specific class of install failure:

- `--help` **failure**: `orca-guard` is not on the PATH. The binary was not installed or the PATH was not updated. Without this, no other test can run.
- **Safe prompt failure**: The Guard is misconfigured or in the wrong mode. If a safe prompt is blocked, every prompt will be blocked, and the agent is unusable.
- **Fake key not blocked**: The Guard is not in `production` or `max` mode, or the secret patterns are not loaded. This is a security gap — real secrets would pass through.
- **Fake key echoed**: The Guard has a bug in its error output. This is a security gap — the error message itself is an exfiltration path.
- **Env-var reference blocked**: The Guard's allow-list is too narrow. This is a usability gap — legitimate code references are blocked, and developers will disable the Guard to work around it.
- **Secret-file read not blocked**: The `PreToolUse` hook is not configured. This is a security gap — an agent can read secrets directly from the filesystem.
- **Synthetic session not reaching sink**: The push token, sink URL, or network path is wrong. This is a capture gap — events are lost.
- **Capture not surviving reboot**: The systemd service or LaunchAgent is not enabled. This is a persistence gap — capture stops on reboot.

Together they prove the full edge posture is operational. An install that passes all seven checks has capture, Guard, push, persistence, and health — the complete stack.

Customer Install at Scale: Three Models

The scale guide (`docs/ORCA_CUSTOMER_INSTALL_AT_SCALE.md`) defines three install models for customer deployments. Each model has different ownership boundaries and operational responsibilities.

MODEL A: CUSTOMER-MANAGED KUBERNETES

Use when the customer requires data residency, private network routing, or their own backup and security controls.

- **Customer owns**: the Kubernetes cluster, disk, backup policy, ingress, logging, and secrets manager.
- **ArmyknifeLabs provides**: images, manifests, operational guidance, and update procedure.
- **Tokens**: generated per tenant/customer and loaded into the customer's secrets manager.

Reference files:

```
deploy/gke/orca.yaml
deploy/gke/README.md
docs/ORCA_CUSTOMER_INSTALL_AT_SCALE.md
```

The customer controls the infrastructure, the secrets, and the network. ArmyknifeLabs provides the images and manifests but does not have access to the running system. This is the model for regulated customers, government agencies, and enterprises with strict data residency requirements.

MODEL B: ARMYKNIFELABS-MANAGED SAAS ORIGIN

Use when ArmyknifeLabs operates the production origin on behalf of customers.

- **ArmyknifeLabs owns**: GKE or equivalent stateful cloud runtime.
- **Cloudflare Edge** fronts the public API.
- **Each customer receives**: scoped read, ingest, admin, and optional origin tokens.
- **Tenant isolation**: enforced by token policy at the sink.
- **Backups, audit retention, and restore drills**: ArmyknifeLabs operational responsibilities.

In this model, the customer points their workstation clients at the Cloudflare Edge URL. The Edge Worker validates the customer's scoped tokens and forwards to the GKE origin. The customer never touches the origin infrastructure. This is the model for SMB customers and design partners who want the platform without the operational burden.

MODEL C: DOCKER COMPOSE PILOT

Use for single-tenant pilot, isolated lab, or customer proof of value.

- Single host with Docker Engine and Compose plugin.
- One writable sink, local Docker volumes.
- Customer-managed backup target, manual upgrade window.
- Not the preferred multi-customer SaaS runtime.

This is the fastest path to a working deployment. The customer runs `docker compose up -d` on a single host and gets the full stack. It is not horizontally scalable, but it is sufficient for a pilot or a small team. The Compose file is the same one that runs on the lab host — no special pilot build is needed.

SCALE GUIDE CONSTRAINTS

All three models share the same production rules:

1. Do not route production customers to the `.114` lab host, home LANs, or developer-only tunnels.
2. Do not expose the sink directly to browsers.
3. Do not give browser code an Orca bearer token.

4. Do not run more than one writable `agentchron-sink` against the same SQLite database.
5. Do not store `AGENTCHRON_DB_PATH` on tmpfs, ephemeral disk, or unreliable network storage.
6. Do not copy raw unsanitized customer logs into production storage unless the customer contract explicitly allows raw retention.
7. Do not use the legacy all-scope token for new customers.

These rules apply to all three models. Model C (Compose pilot) is the most likely to violate rule 4 (running multiple sinks) because a pilot environment is often treated casually. The scale guide is explicit: even in a pilot, there is one writable sink.

Tenant Scope Model and Scoped Tokens

Each customer or workspace receives scoped tokens. Token policies are comma-separated key/value strings:

```
tenant=customer-a,team=platform,workspace=orca,visibility=private,agent=neo
```

`tenant` aliases event `org`. Use either `tenant` or `org`; prefer `tenant` in customer install docs and `org` in event/query fields.

SUPPORTED SCOPE KEYS

```
actor
tenant
org
team
workspace
visibility
agent
```

TOKEN CLASSES

TOKEN	CAN READ	CAN INGEST	CAN MUTATE WORKFLOW/LIBRARY
`ORCA_READ_TOKEN`	Yes (within scope)	No	No
`ORCA_INGEST_TOKEN`	No	Yes (within scope)	No
`ORCA_ADMIN_TOKEN`	Yes (within scope)	No	Yes
`ORCA_ORIGIN_TOKEN`	Yes	Yes	Yes

SCOPE BEHAVIOR

- **Read tokens cannot ingest or mutate.** A compromised read token cannot inject fake events into the corpus.
- **Ingest tokens cannot read.** A compromised ingest token cannot exfiltrate the corpus.
- **Admin tokens can mutate workflow/library state.** A compromised admin token can modify workflows and Library artifacts, but cannot ingest events.

- **Scoped read tokens cannot widen query filters outside their policy.** A token scoped to `tenant=customer-a` cannot query `org=another-customer`.
- **Scoped ingest tokens stamp missing event scope and reject conflicting scope.** If an event arrives without scope, the token's scope is applied. If an event arrives with a conflicting scope, it is rejected.
- **Scoped direct reads are checked against stored row/session metadata.** Even if a scoped token tries to fetch an event by ID, the event's stored scope is checked against the token's scope.
- `/v1/audit` **and** `/v1/stats` **require unscoped admin/origin access.** Scoped tokens cannot see aggregate stats or audit logs, because those could reveal information about other tenants.
- **Cross-tenant reads return** `403`**.** This is the negative test that must pass before go-live.
- **Admin mutations write** `api_audit_event` **rows** with the token class, action, route, status, and details.

EXAMPLE CUSTOMER TOKEN ENVIRONMENT

````
ORCA_READ_TOKEN=<... scope to `tenant=customer-a`. The read token can query events,
sessions, graph context, and Library artifacts within that tenant's scope, but nothing
outside it.

### Token Lifecycle

Token policy strings may include `expires_at=<RFC3339>` or `expires_at=<YYYY-MM-DD
HH:MM:SS>`. Expired tokens are rejected before route-scope checks. Setting any non-empty
`revoked_at=...` value revokes that configured token immediately and records the lifecycle
value in `auth_token_metadata`.

Current limitation: explicit token expiry/revocation enforcement is still a productization
gap. Until that lands, offboarding must rotate deployment secrets and redeploy without the
customer's tokens. The offboarding procedure is:

1. Disable customer emitters at edge/gateway.
2. Remove customer scoped tokens from secrets manager.
3. Redeploy sink and edge with updated secrets.
4. Confirm old tokens return `401`.
5. Export or delete customer data per contract.
6. Record final audit extract.
7. Remove customer-specific raw archive copies per retention policy.
8. Confirm backups follow contractual deletion or retention requirements.

## Go-Live Acceptance

A deployment is not production-ready until the go-live acceptance checklist is complete.
This is the gate.

### Health and Scope Checks

1. **Health checks pass:**
   ```bash
   curl -fsS https://<orca-domain>/v1/health
   ```

2. **Scoped ingest succeeds:**
   ```bash
   curl -fsS \
     -H "Authorization: Bearer $ORCA_...EN" \
     -H "Content-Type: application/json" \
     -d '{"events":[{"type":"user","sessionId":"customer-a-
smoke","uuid":"smoke-1","message":{"role":"user","content":"smoke"},"source_path":"smoke://
install","host":"install-check","byte_offset":1}]}' \
     https://<orca-domain>/v1/events
   ```

3. **Scoped read succeeds:**
   ```bash
````
````

```
   curl -fsS \
     -H "Authorization: Bearer $ORCA_...EN" \
     'https://<orca-domain>/v1/search?limit=5&org=customer-a'
   ```

4. **Cross-tenant read fails with `403`:**
   ```bash
   curl -i \
     -H "Authorization: Bearer $ORCA_...EN" \
     'https://<orca-domain>/v1/search?org=another-customer'
   ```
   Expected: HTTP `403`.

5. **Admin mutation requires admin token:**
   ```bash
   curl -fsS \
     -H "Authorization: Bearer $ORCA_...EN" \
     'https://<orca-domain>/v1/library' -d '...'
   ```
   A read token attempting the same mutation returns `403`.

6. **Audit shows smoke tests:**
   ```bash
   curl -fsS \
     -H "Authorization: Bearer $ORCA_...EN" \
     'https://<orca-domain>/v1/audit?limit=20'
   ```

Backup and Recovery Checks

7. **SQLite PVC snapshot succeeds.**
8. **Restore drill succeeds** — restore into isolated namespace, verify `/v1/health`, `/v1/stats`, fetch a known session, run source coverage for one sampled source path. Document snapshot ID, command history, elapsed time, and result.

Operational Checks

9. **Disk/ingress alerting is active.**
10. **Incident contacts are recorded.**
11. **Customer has emitter configuration.**
12. **Customer has support path.**
13. **Rollback image and snapshot are documented.**

A deployment that has not completed this checklist is not production-ready. The checklist is not a suggestion — it is the acceptance gate. The scale guide states: "Do not declare the install production-ready until all are true."

Tenant Onboarding Checklist

Before a tenant goes live:
```
```

```
- customer id assigned
- tenant/org slug approved
- team/workspace values approved
- retention class selected
- raw archive policy selected
- read token generated
- ingest token generated
- admin token generated, if needed
- token scopes configured
- legacy token disabled for new customer
- customer emitter configured
- smoke event ingested
- scoped read succeeds
- cross-tenant read returns `403`
- audit row appears for read and ingest tests
- backup policy includes this tenant's data
- customer runbook delivered

### Security Acceptance Checklist

- TLS terminates at approved customer or Cloudflare edge
- sink, SQLite, Neo4j, and Qdrant are private
- browser never receives a bearer token
- legacy all-scope token disabled for new customer
- read token cannot ingest
- ingest token cannot read
- admin token required for mutations
- scoped token cannot query another tenant/workspace
- `/v1/audit` available only to unscoped admin/origin operators
- audit rows include MCP tool where applicable
- secrets are stored only in approved secret manager
- backups encrypted
- raw archive policy approved
- restore drill completed

## GTM SKUs

The first GTM product is `Orca Data Factory with A2A Mesh NATS`. Four SKUs cover the
deployment spectrum:

| SKU | Purpose | Buyer |
|---|---|---|
| `orca-data-factory-smb` | Single-tenant Docker Compose package for pilots, SMB, and
sovereign single-node installs | Technical founder, IT manager, small platform team |
| `orca-data-factory-enterprise` | Kubernetes/cloud package with customer-managed secrets,
storage, ingress, backups, and HA-ready NATS | Enterprise platform/security team |
| `orca-workstation-client` | Per-OS developer endpoint package with capture, Guard, RBAC
hooks, and health checks | Every developer machine |
| `orca-data-factory-airgap` | Offline package with signed tarballs, checksums, docs, and
```

```
no network package pulls | Regulated, defense, disconnected labs |

### Product Boundary

**Include in the GTM package:**

- Orca origin runtime: `agentchron-web`, `agentchron-sink`, GraphRAG API, Neo4j, Qdrant,
Foundry utilities, MCP bridge.
- A2A Mesh NATS base bus with customer-owned auth and subject naming.
- Orca Guard secret gate and approval/deny review data.
- Claude Code RBAC hooks and `orca-mcp-gateway` policy enforcement.
- Workstation capture clients for Linux, macOS, and Windows.
- Data Factory bronze/silver/gold tooling, quarantine, manifests, and smoke checks.
- GraphRAG ingest/query utilities and MCP search path.
- ContextOS/governance receipt emitters where present.
- Customer install docs, smoke scripts, checksums, package manifest, and offline/air-gap
handoff notes.

**Exclude from the default GTM package:**

- Command Center UI as a required component.
- Agent2600 product surfaces.
- Live lab `.env` files, tokens, signing keys, customer secrets, or local secret-store
exports.
- Raw archive, SQLite, Neo4j, Qdrant, or GraphRAG volumes.
- Local Git metadata and `target/` build artifacts.
- The internal gold corpus.

Command Center and Agent2600 are add-ons or design-partner tracks unless the SOW explicitly
includes them. No package ships with lab tokens, `.env` files, data volumes, or internal
gold corpus. The packaging spec is explicit: "No lab tokens, `.env` files, database
volumes, or local corp secrets ship in a customer package."

### Guard Mode Defaults by SKU

| Deployment | Default Guard mode |
|---|---|
| SMB/pilot | `production` |
| Enterprise managed workstation | `max` |
| UAT | `audit` (observe-mode only) |
| Developer troubleshooting | `dev` (by explicit local admin action) |

The installer ships Guard in `production` mode by default. This is Pattern 8 — Code-Owned,
Narrow Allow-Lists — applied at install time. Broad local "allow this secret path" files
are not supported because they would let a workstation bypass the main control.

### Data Factory Gates

The package must expose these commands and docs:
```

```
- raw archive verification
- bronze export
- silver clean/dedup/redact/license pass
- silver review
- gold promotion
- SFT/eval/index smoke
- GraphRAG ingest of approved internal subset

Hard gates:

- raw is immutable
- secret/PII filters fail closed
- restricted-license rows are excluded from training and query indexes
- `internal_retrieval_only` can feed GraphRAG but not model training
- two-reviewer gold approval for production gold
- manifests include hashes, record counts, schema versions, command, and git commit

### A2A Mesh NATS Package

The GTM package includes NATS as the internal event/receipt/control wire. SMB profile:
single NATS container in Docker Compose, bind internally by default. Enterprise profile:
NATS StatefulSet or managed NATS, TLS required, customer-managed credentials, JetStream
enabled only when durable streams are required.

Suggested subject families:

```text
a2a.team.<tenant>.room.<instance>.governance_receipt
a2a.team.<tenant>.room.<instance>.guard_decision
a2a.team.<tenant>.room.<instance>.session_event
a2a.team.<tenant>.room.<instance>.foundry_event
a2a.team.<tenant>.room.<instance>.health
```

DEFINITION OF DONE

The GTM package is done when:

- Linux, macOS, and Windows workstation packages install and smoke cleanly.
- SMB Compose starts on a fresh host and passes platform smoke.
- Enterprise cloud spec is complete enough for AWS, Azure, GCP, and on-prem implementers to deploy without lab assumptions.
- NATS is packaged as the base bus and does not ship lab credentials.
- Claude Code Guard/RBAC is installed and verified.
- Data Factory commands and gates are documented and executable.
- GraphRAG ingest instructions accept only approved internal data.
- Package manifest proves no live secrets, data volumes, or local git metadata are included.
- Command Center and Agent2600 are absent from the default SKU or clearly marked optional/future.
```

Orca Guard and Hooks in the Installer

The installer's job is not just to place binaries — it establishes the full capture + guard posture on a fresh workstation by configuring Claude Code hooks in `~/.claude/settings.json`.

HOOK CONFIGURATION

The installer configures multiple hook points:

HOOK EVENT	COMMAND	PURPOSE
`UserPromptSubmit`	`orca-guard claude-hook`	Block secrets in prompts before they reach the model
`PreToolUse`	`orca-guard claude-hook`	Block direct secret-file reads and risky tool calls
`SessionStart`	`agentchron-claude-hook`	Capture session start event
`PostToolUse`	`agentchron-claude-hook`	Capture tool use events
`Stop`	`agentchron-claude-hook`	Capture session end event
`SessionStart`	`orca-session-rules.py`	Inject GraphRAG context on session start

The Claude Code settings shape for Guard:

```
{
  "hooks": {
    "UserPromptSubmit": [
      {"hooks": [{"type": "command", "command": "orca-guard claude-hook"}]}
    ],
    "PreToolUse": [
      {"hooks": [{"type": "command", "command": "orca-guard claude-hook"}]}
    ]
  }
}
```

The `orca-session-rules.py` hook is the GraphRAG-first agent context hook. It runs on `SessionStart` and `UserPromptSubmit`, derives safe query terms from the hook payload, calls `/v1/graph/context`, and injects a compact evidence-cited context block when the backend answers quickly. The hook fails open, caps graph calls at 2 seconds by default, and negative-caches timeouts for 5 minutes. This means a slow GraphRAG backend does not block the agent's session — it just means the agent starts without the context block. See `docs/ORCA_GRAPHRAG_AGENT_HOOK_ROLLOUT.md` before rolling this hook beyond canary hosts.

GRAPHRAG CONTEXT HOOK

The context hook invocation:

```
orca-session-rules.py --include-graph-context --hook-payload-stdin --graph-only
```

This runs on `SessionStart`, derives safe query terms from the hook payload (session working directory, project name, workspace tags), calls `/v1/graph/context`, and injects a compact context block into the session. The hook is designed to be safe by default:

- **Fails open**: If the GraphRAG backend is unavailable, the session starts without context. The agent is not blocked.
- **Caps graph calls at 2 seconds**: A slow backend does not delay the session start.
- **Negative-caches timeouts for 5 minutes**: If the backend times out once, it is not retried for 5 minutes, preventing repeated slow calls.

INSTALLER GUARD MODE CONTROL

The installer can switch Guard mode without rewriting token config:

```
# Switch to dev mode for local development
./install.sh --skip-bin-install --no-env --configure-claude-guard --guard-mode dev

# Switch to audit mode for QA
./install.sh --skip-bin-install --no-env --configure-claude-guard --guard-mode audit

# Switch back to production
./install.sh --skip-bin-install --no-env --configure-claude-guard --guard-mode production
```

The `--skip-bin-install` and `--no-env` flags tell the installer to only update the Claude Code hooks, not touch binaries or the push environment. This lets operators change Guard posture without reinstalling the full package — useful for switching from `audit` (during UAT) to `production` (after UAT passes).

ORCA LLM GATEWAY

`orca-llm-gateway` is the egress defense-in-depth layer. Point Claude Code at it with `ANTHROPIC_BASE_URL` so requests are scanned before an Anthropic-compatible upstream receives them.

```
# Build
cargo build --release -p agentchron-agent --bin orca-llm-gateway
cp target/release/orca-llm-gateway ~/.local/bin/

# Dead-upstream smoke
ORCA_GATEWAY_TOKEN=*** \
ORCA_GATEWAY_UPSTREAM=http://127.0.0.1:9 \
orca-llm-gateway --bind 127.0.0.1:19741

# Test: blocked secret returns 403
CANARY="$(python3 -c 'import uuid; print("sk-ant...ard-" + uuid.uuid4().hex)')"
curl -sS -i \
  -H 'Authorization: Bearer *** application/json' \
  http://127.0.0.1:19741/v1/messages \
  --data "{\"messages\":[{\"role\":\"user\",\"content\":\"$CANARY\"}]}"

# Expected: HTTP 403 and orca_guard_blocked
# Safe requests with dead upstream should return 502, proving the gateway
# attempted to forward only after Guard allowed the body.
```

For real upstream forwarding:

```
export ORCA_GATEWAY_UPSTREAM=https://api.anthropic.com
export ORCA_GATEWAY_UPSTREAM_API_KEY=*** vault>
```

THE FULL EDGE POSTURE

After the installer runs, a fresh workstation has:

1. **Capture**: `agentchron-agent` or `agentchron-push` streaming sanitized JSONL to the sink.
2. **Guard**: `orca-guard` blocking secrets in prompts and tool calls before they reach the model.
3. **MCP gateway**: `orca-mcp-gateway` enforcing RBAC on MCP tool calls.
4. **GraphRAG context**: `orca-session-rules.py` injecting cited context blocks on session start.
5. **LLM gateway**: `orca-llm-gateway` scanning egress before the upstream receives it (optional, configured separately).
6. **Persistence**: systemd service (Linux) or LaunchAgent (macOS) surviving reboots.
7. **Health checks**: `agentchron-ingest-healthcheck.sh` verifying the pipeline is flowing.

This is the complete edge posture — not just a capture agent, but the full governance and context stack. The installer's job is to make all of it work on a fresh machine with one command, then prove it with smoke tests.

Patterns and Anti-Patterns

PATTERNS DEVELOPED IN THIS CHAPTER

Pattern 2: Local-Before-Transport. The installer ships sanitized config. The push token is passed only through the environment, never on the command line. The `push.env` file is written with `0600` permissions. The local sanitizer runs before any bytes leave the host. The installer configures the hooks that make local-before-transport

the default behavior on every workstation. This pattern runs from the capture agent (Chapter 2) through the sanitizer (Chapter 6) to the installer — every layer sanitizes before transport, and the installer ensures the hooks are configured to make it automatic.

Pattern 8: Code-Owned, Narrow Allow-Lists. The installer sets Guard in `production` mode by default. Broad local allow-list files are not supported. Enterprise policy bundles are planned but must not allow raw secret values. The Guard's allow-list is code-owned — it ships with the package, not user-editable at the workstation level. This pattern appeared first in Chapter 5 (Guard) and is enforced here at install time. The four Guard modes (`production`, `dev`, `audit`, `max`) are the only configuration surface; there is no "add your own allow-list" escape hatch.

Pattern 12: Reproducibility Envelope. Versioned tarballs with SHA-256 sidecars. Package manifests with build timestamp, git commit, channel, and component list. The air-gap SKU ships signed tarballs with checksums and no network package pulls. Every artifact is reproducible from manifest + code commit. This pattern connects to the Data Foundry's run manifests (Chapter 10) and the deployment manifests (Chapter 12) — the same principle of reproducible, versioned artifacts applies to the installer packages.

ANTI-PATTERNS ADDRESSED

Anti-Pattern 7: Premature Horizontal Scaling. The installer does not offer a "multi-sink" option. The workstation client points at one sink endpoint. The scale path is edge + web replicas, not multiple writable sinks. The installer enforces the single-writer constraint by design — there is no configuration that would cause a workstation to write to two sinks simultaneously. If the sink endpoint is behind a load balancer, the load balancer must route all writes to the single writable sink instance.

Anti-Pattern 1: Single Shared Token. The installer generates scoped tokens for customer deployments. The scope flags (`--org`, `--team`, `--workspace`, `--visibility`, `--purpose`) ensure every event is tagged correctly from the first push. The scale guide requires disabling the legacy all-scope token for new customers by setting `AGENTCHRON_LEGACY_TOKEN_ENABLED=false` in the customer environment, so that only scoped tokens issued by the installer are accepted by the sink.

The installer infrastructure is where Orca meets the customer's workstation. The Linux and macOS bootstrap scripts are deliberately small — they acquire a versioned tarball, verify its SHA-256, and delegate to the packaged `install.sh`. The packaged installer places the full edge posture: capture, Guard, MCP gateway, GraphRAG context hooks, persistence, and health checks. The acceptance criteria prove that all of it works before the install is declared complete.

The three customer install models — customer-managed Kubernetes, ArmyknifeLabs-managed SaaS, and Docker Compose pilot — share the same production rules, the same tenant scope model, and the same go-live acceptance checklist. The four GTM SKUs cover the deployment spectrum from SMB to air-gapped defense environments. The product boundary is explicit: the GTM package includes the Data Factory, the Guard, the capture clients, and the GraphRAG tooling. It does not include lab tokens, data volumes, or the internal gold corpus.

The installer is the last mile. A platform that captures, sanitizes, stores, refines, and serves governed knowledge is useless if it cannot be installed on a fresh workstation with one command and proven to work. The installer infrastructure is what makes Orca deployable at fleet scale.

Chapter 14: The Orca Roadmap

This is the closing chapter. The previous thirteen chapters built the platform piece by piece: the capture agent, the sink, the web console, the Guard, the sanitizer, the MCP gateway, the attestation layer, the GraphRAG ingestion, the Data Foundry, the ContextOS edge, the deployment infrastructure, and the installer fleet. This chapter looks forward — to the 2027 secure AI fabric, the three-step product ladder, the competitive moat, and the build order that gets there. It connects the themes that ran through every chapter and explains why the governed data engine, not the model, is the durable advantage.

Position: Trust Layer for Open-Weight AI Adoption

Orca goes to market as the trust layer for open-weight AI adoption. Not a model lab. Not a commodity inference host. Not a generic LLM router. The platform's job is to let an enterprise say yes to open-weight models without giving up auditability, policy, or control.

The buyer pain is not "where can I get the cheapest token." The pain is:

- Can I safely adopt open models without supply-chain compromise?
- Can I prove which model, prompt, tool, dataset, graph context, and agent acted?
- Can I keep sensitive data inside my environment?
- Can I let developers and agents use AI without leaking secrets or bypassing policy?
- Can I generate audit evidence without trusting a vendor dashboard?

Every chapter in this book addressed one or more of these pains. The capture pipeline (Chapters 2–4) proved which agent acted and what it did. The Guard (Chapter 5) prevented secrets from leaking. The sanitizer (Chapter 6) ensured that what left the host was clean. The MCP gateway (Chapter 7) enforced policy on tool calls. The attestation layer (Chapter 8) signed provenance. The GraphRAG index (Chapter 9) retrieved governed context. The Data Foundry (Chapter 10) refined exhaust into renewable assets. The ContextOS edge (Chapter 11) served cited context back to agents. The deployment infrastructure (Chapter 12) made it portable. The installers (Chapter 13) made it deployable at fleet scale.

The 2027 roadmap weaves these capabilities into a single product narrative: a secure AI fabric where every model, every prompt, every context block, and every tool call is governed, attested, and auditable.

Step 1: Certify and Harden (Revenue Now)

The fastest wedge maps directly to supply-chain fear. Ship a paid CI/CLI gate that scans, redteams, hardens, and signs open model artifacts before they enter customer environments. The roadmap document (`docs/ORCA_SECURE_AI_FABRIC_2027_ROADMAP.md`) is explicit: this is the fastest wedge because it maps directly to supply-chain fear.

DELIVERABLES

```
orca certify model <repo|path>
```

The certification gate produces five deliverables:

1. **Scanner and license policy report** — checks the model artifact for known vulnerabilities, pickle deserialization exploits, embedded prompts, and license restrictions. The license policy report flags restricted licenses that would prevent internal training or retrieval use, connecting to the Data Foundry's `license.policy` field (Chapter 10) — a model with a restricted license cannot feed the gold pipeline.
2. **Redteam report and remediation status** — runs an automated redteam assessment against the model, testing for prompt injection, tool poisoning, jailbreaks, and safety gaps. The remediation status shows which issues were found, which were fixed, and which remain open.
3. **Signed model attestation** — an Ed25519-signed attestation that the model passed the gate, using the same signing path as the ContextOS attestation layer (Chapter 8). The attestation is a DSSE receipt — the same format used for Data Foundry gold promotion (Chapter 10).
4. **Fail-closed policy for unsigned or failed artifacts** — if the artifact is unsigned or fails the gate, the policy is fail-closed: it does not enter the customer environment. This is the same fail-closed principle as the secret filter (Chapter 6) and the Data Foundry quarantine (Chapter 10).
5. **Public-safe hardened model leaderboard** — shows which open models have been certified, what issues were found, what was remediated, and what the attestation says. This is not a marketing page — it is an evidence pack that an auditor can verify offline.

CERTIFICATION FLOW

```
Model repo or artifact path
  → Scan (vulnerabilities, pickle exploits, embedded prompts)
  → License policy check (restricted vs. internal_training_allowed)
  → Redteam assessment (prompt injection, tool poisoning, jailbreaks)
  → Remediation (fix what can be fixed, document what cannot)
  → Sign attestation (Ed25519 via ContextOS/ATCS)
  → Publish to hardened model registry
  → Fail-closed if unsigned or failed
```

An enterprise that wants to adopt an open-weight model — Llama, Mistral, Qwen, DeepSeek, or any other — runs `orca certify model` against the model repo or artifact path. The gate scans for known vulnerabilities, checks the license policy, runs a redteam assessment, produces a remediation status, and signs the result as an attestation. If the artifact is unsigned or fails the gate, the policy is fail-closed: it does not enter the customer environment.

WHY THIS IS FIRST

Certification is the fastest wedge because it maps to a fear that every security team already has. Open-weight models are downloaded from Hugging Face, GitHub, or mirrors with varying levels of scrutiny. The supply chain is real: model files can contain pickle deserialization exploits, embedded prompts, license restrictions, and safety gaps. A CI gate that scans, hardens, and signs before the model enters the environment is a product that a CISO can buy without explaining what a GraphRAG is.

The hardened model leaderboard (EMPIRE-808) is the public proof surface. It shows which open models have been certified, what issues were found, what was remediated, and what the attestation says. This is not a marketing

page — it is an evidence pack that an auditor can verify offline. The leaderboard connects to the router (Step 2): the router checks the model registry attestation before routing a request to a model. A model that is not in the registry — or whose attestation has expired — is not routed to.

STANDARDS ALIGNMENT

EMPIRE-800 covers aligning model attestations to OMS Sigstore SLSA and in-toto. This means the attestation format follows established supply-chain security standards rather than inventing a new one. SLSA (Supply-chain Levels for Software Artifacts) provides a framework for verifying the integrity of build artifacts. In-toto provides a framework for verifying the integrity of the build process. By aligning with these standards, the certification gate produces attestations that security teams can verify with existing tooling, not just with Orca's own verifier.

The claim discipline is important here: the roadmap says to gate "public compliance certification claims" until smoked. Claiming SLSA Level 3 compliance before the build process has been audited to that level would be the kind of vendor-dashboard trust that the platform was built to replace. The certification gate ships what it can prove; broader compliance claims come after audit.

Step 2: Route with Proof (Recurring Control Plane)

The second step ships a GraphRAG-aware multi-LLM smart router that acts as a `BASE_URL` replacement. It is OpenAI-compatible — any client that can point at `https://api.openai.com/v1` can point at the Orca router instead.

REQUIRED ROUTER BEHAVIOR

1. Accept OpenAI-compatible requests.
2. Run Orca Guard preflight (secret/PII block, RBAC, MCP tools/call policy).
3. Classify prompt/data route class.
4. Fetch signed GraphRAG context blocks when allowed.
5. Check model registry attestation before routing.
6. Route to local, customer-hosted, or approved external providers.
7. Emit signed audit events and receipts.

ROUTER ARCHITECTURE

```
Developer / Agent / App
        |
        v
OpenAI-compatible Router API
        |
        +--> Orca Guard preflight
        |    - secret and PII block
        |    - discipline RBAC
        |    - MCP tools/list and tools/call policy
        |
        +--> Hermes / tenant / policy context
        |
        +--> GraphRAG context block API
        |    - signed context blocks
        |    - provenance receipts
        |    - visibility_scope enforcement
        |
        +--> Hardened model registry
        |    - scan report
        |    - redteam report
        |    - license policy
        |    - attestation status
        |
        +--> Route decision
             - local model
             - customer endpoint
             - approved external provider
             - deny / human approval
```

Every allow, deny, route, context injection, and model selection emits audit metadata and, where security-relevant, a signed governance receipt. This is the ContextOS attestation layer (Chapter 8) applied to routing decisions — the same Ed25519 signing path, the same RFC 8785 JCS canonicalization, the same provenance chain.

HOW IT CONNECTS TO PREVIOUS CHAPTERS

The router is not a new product. It is the composition of capabilities built in the previous chapters:

- **Guard preflight** is the Orca Guard from Chapter 5, running in the `orca-llm-gateway` mode that was already smoke-tested with a dead upstream.
- **GraphRAG context blocks** are the Context Block API from Chapter 11 — `compose_context_block(topic, scope)` returning signed, cited, supersession-aware context.
- **Model registry attestation** is the Certify and Harden output from Step 1 — scan reports, redteam reports, license policies, and signed attestations.
- **Audit events** are the `api_audit_event` rows from the sink (Chapter 3), extended with signed governance receipts.
- **Tenant/policy context** is the scoped token model from Chapter 13 — `tenant=customer-a,team=platform,workspace=orca,visibility=private`.

The router is the front door into the platform, not the platform itself. It competes on proof, policy, context, and data sovereignty — not on provider breadth or token price.

WHAT THE ROUTER DOES NOT DO

The roadmap is explicit about what to avoid:

- Do not compete on provider breadth. Compete on proof.
- Do not claim "cheapest router" — commodity routing is commoditized.
- Do not claim "we replace OpenRouter" — OpenRouter is a commodity router; Orca is a governance layer.
- Do not claim "we host frontier models" — the platform routes to models, it does not host them.

ROUTE DECISION MATRIX

The router's route decision is a matrix of model attestation, data classification, and policy:

CONDITION	ROUTE	RATIONALE
Model attested, data public, policy allows	External provider	Normal routing — attested model, no sensitive data
Model attested, data sensitive, policy allows	Customer-hosted	Sensitive data stays in customer environment
Model attested, data sensitive, no customer model	Local model (if available)	Fallback to local model to keep data in-house
Model not attested	Deny	Fail-closed — uncertified models are not routed to
Model attested but attestation expired	Deny	Stale attestation is treated as no attestation
Prompt contains secrets (Guard block)	Deny	Guard preflight blocks before routing
Prompt classified as high-risk	Human approval	Gray-area prompts require human sign-off

Every route decision emits audit metadata. Security-relevant decisions also emit signed governance receipts — the same Ed25519 signing path as the ContextOS attestation layer (Chapter 8). The receipt includes the route decision, the model attestation hash, the context block hash (if any), the Guard decision, and the tenant scope. This receipt is what an auditor uses to verify that a specific request was handled correctly.

Step 3: Own the Flywheel (Compounding Moat)

The third step uses Orca Data Factory output to create customer-local GraphRAG bundles and eventually vertical SLM/nano-LM artifacts. This is where the moat compounds.

THE DATA FLYWHEEL

```
Agent/developer exhaust
  → captured and sanitized (Ch 2-6)
    → Data Foundry raw → bronze → silver → gold (Ch 10)
      → GraphRAG ingest of approved internal subset (Ch 9)
        → context blocks served to agents (Ch 11)
          → agents produce better work
            → better work is captured and refined
              → the corpus compounds
```

The flywheel is not a metaphor. It is a literal feedback loop, and it has a safety gate: only `success`-outcome, reviewed records re-enter the graph. Without that gate, the loop amplifies errors — agents read the graph, their sessions get re-captured into silver, and the graph drifts toward whatever the agents did, whether it was right or wrong. The feedback-loop safety gate (Chapter 9) ensures that only reviewed success outcomes re-enter, and retrieval success-vs-failure becomes free DPO preference data. The assets improve themselves.

WHY THIS IS THE MOAT

A model is a perishable snapshot. It is replaced every base-model cycle — every few months, a new foundation model makes the previous one obsolete. A model lab that fine-tunes today's model has to start over when tomorrow's base model ships.

The data engine compounds. The capture pipeline, the renewable corpus, the eval suites, and the retrieval graph all grow with every session. The corpus does not become obsolete when a new base model ships — it becomes more valuable, because the new model can be trained on or retrieval-augmented with the same governed data.

This is the counter to Anti-Pattern 9 — Treating the Model as the Moat. The build spec's strongest strategic claim is: a model is a perishable snapshot that decays each base-model cycle; the data engine compounds. The durable IP is the capture pipeline, the renewable corpus, the eval suites, and the retrieval graph. Fine-tuning is an optional downstream, never the goal.

CUSTOMER-LOCAL MOATS

Each customer's moat is unique. The GraphRAG bundle, the gold data, the Library artifacts, and the workflow rules are all customer-local. Cloud federation is opt-in, summaries only. The customer owns their own moat — no competitor can copy it because it is built from the customer's own engineering exhaust.

This is Pattern 6 — Private-by-Default Scoping — at the strategic level. The platform's default is local-first, data-residency-by-default. The customer's data stays in the customer's environment. The moat is on-prem residency plus governed refinement plus the customer owns their own moat.

GraphRAG Strategy

The winning GraphRAG position is not "host a graph database." It is an automated, signed graph lifecycle.

CUSTOMER VALUE

- Data stays in the customer environment.
- Graphs can be local, ephemeral, or air-gapped.
- Graph bundles and patches are signed.
- Tampered nodes or edges fail verification.
- Router and agents consume bounded context blocks instead of raw database access.

DELIVERABLES

- Context block API (Chapter 11 — the `compose_context_block` function).
- Signed graph bundle format.
- Local/ephemeral graph runtime profile.
- Incremental graph patch format.
- Router integration for context injection.
- Evidence export tying graph context back to source receipts.

The signed graph bundle format is the GraphRAG equivalent of the Brain Bundle (Chapter 11). A graph bundle is a portable, signed export of a GraphRAG index — nodes, edges, and their provenance — that can be verified offline and loaded into an air-gapped environment. Tampered nodes or edges change the canonical hash and verification fails.

Real-Time Comprehension Layer

The router handles request-time policy. The real-time comprehension layer handles stream-time understanding: it turns silver-governed Orca exhaust into function and class atoms, deduplicates repeated patterns, analyzes canonical atoms once, and exposes the results to GraphRAG, Gold Data, and Command Center lenses.

This is the missing bridge between "we captured the work" (Chapters 2–6) and "the platform understands the work" (Chapters 9–11). The Data Foundry refines raw exhaust into silver and gold (Chapter 10). The comprehension layer extracts structured understanding from silver — not prose summaries, but function atoms, class atoms, and deduplicated patterns that can be stored in GraphRAG as structured analysis. This structured analysis is what makes the GraphRAG context blocks (Chapter 11) richer than keyword-matched snippets — the context block can cite a specific function atom that was extracted, analyzed, and stored, rather than a raw event that happens to contain a keyword.

CORE RULES

- **Tap silver, never bronze.** Bronze is normalized but not cleaned; silver has passed secret/PII/license/dedup gates. Tapping bronze would risk feeding unredacted content into the comprehension pipeline.
- **Decide the streaming substrate once before building parallel buses.** EMPIRE-811 is the ADR: NATS JetStream vs Kafka/Redpanda. This decision must be made before any stream is built, because the substrate choice affects durability guarantees, replay semantics, and operational complexity. Building parallel buses on different substrates would create a fragmentation problem.
- **Treat deduplication as the cost valve.** Repeated patterns are analyzed once, not every time they appear. If ten sessions all implement the same OAuth retry logic, the comprehension layer extracts the function atom once, stores it, and links future occurrences to the canonical atom. This is the same dedup principle as the compose algorithm (Chapter 11) — prefer canonical sources over copies.
- **Store structured analysis in GraphRAG, not only prose summaries.** A function atom is a structured record: name, signature, file, commit, behavior summary, and citation. This is more useful to an agent than a prose summary of "there's some OAuth retry logic somewhere."
- **Use Command Center lenses to show bounded comprehension instead of flooding the operator.** The comprehension canvas (EMPIRE-812) scopes what the operator sees — a lens for security atoms, a lens for build atoms, a lens for infrastructure atoms — rather than dumping every atom into one view.
- **Receipt worker actions and analysis lineage into ContextOS.** Every atom extraction, every dedup decision, and every analysis result is receipted. This is the same attestation principle as the presence attestation layer (Chapter 8) — every action is signed and traceable.

COMPREHENSION CONVEYOR (EMPIRE-809)

The conveyor is the streaming pipeline that moves silver events from the sink to the comprehension workers:

```
silver tap (sink)
  → streaming substrate (NATS JetStream or Kafka/Redpanda — ADR pending)
    → work-queue (durable, replayable)
      → two-tier workers
        → tier 1: fast extraction (function/class atoms, dedup check)
        → tier 2: deep analysis (canonical atom analysis, behavior summary)
          → durable result store (GraphRAG + content-addressed analysis store)
```

The two-tier worker design separates fast extraction (cheap, runs on every event) from deep analysis (expensive, runs only on canonical atoms that have not been seen before). This is the cost valve: most events are duplicates or noise, and only the canonical atoms warrant deep analysis. The deep analysis result is stored in a content-addressed analysis store — the same content-addressing principle as the context block cache (Chapter 11).

ATOM COMPREHENSION ENGINE (EMPIRE-810)

The atom engine extracts structured atoms from silver events:

- **Function atoms**: function name, signature, file path, commit, behavior summary, calling context, and citations.
- **Class atoms**: class name, methods, file path, commit, behavior summary, and citations.
- **Pattern atoms**: repeated code patterns (e.g., OAuth retry, error handling, logging) with a canonical representative and occurrence count.

Each atom is content-addressed — the same function in the same file at the same commit produces the same atom hash. This makes dedup automatic: if the same function appears in ten sessions, the atom engine extracts it once and links the other nine occurrences to the canonical atom.

This connects to the GraphRAG feedback-loop safety gate (Chapter 9). The comprehension layer feeds structured atoms into the graph. The graph serves context blocks to agents. Agents produce work. Work is captured and refined. Only success-outcome reviewed atoms persist. The loop closes — and the comprehension layer is what makes the loop produce structured knowledge, not just raw events.

Claim Discipline

The roadmap distinguishes between claims that can be made now, claims that must be gated until smoked, and claims to avoid.

USE NOW

- "GraphRAG-aware governance and routing layer for AI agents."
- "OpenAI-compatible router with Orca Guard policy enforcement."
- "Signed provenance and audit evidence for model, data, context, and agent actions."
- "Customer-local GraphRAG and data-sovereign deployment profiles."

These claims are backed by the capabilities built in the previous chapters. The Guard exists (Chapter 5). The signed provenance exists (Chapter 8). The GraphRAG context blocks are specified (Chapter 11). The deployment profiles exist (Chapters 12–13).

GATE UNTIL SMOKED

- Full hardware-root signing across every platform.

- Immutable durable-chain anchoring for every receipt type.
- Trust-score refusal if ATCS gating is not wired.
- Per-tenant BYOK if isolation is not proven.
- Public compliance certification claims.

These capabilities are on the roadmap but not yet proven. Claiming them before they are smoked would be the kind of vendor-dashboard trust that the platform was built to replace.

AVOID

- "We are the cheapest router." Commodity routing is commoditized.
- "We replace OpenRouter." OpenRouter is a commodity router; Orca is a governance layer.
- "We host frontier models." The platform routes to models, it does not host them.
- "AI Act compliant" without counsel-reviewed scope.
- "Cryptographically enforced" when enforcement is policy/gateway level.

Claim discipline is not just marketing hygiene. It is the same principle that governs the Data Foundry's fail-closed quarantine (Chapter 10): do not ship what you cannot prove. The competitive brief is explicit: treat every market statistic in the source reports as unverified until EMPIRE-807 validates it.

Build Order

The roadmap defines a fourteen-step build order that sequences the work to minimize dependencies and maximize early value.

STEP	TICKET	WORK	DEPENDENCY
1	EMPIRE-794	Define Orca Secure AI Fabric packaging and naming	None — lock naming so packaging and GTM stop drifting
2	EMPIRE-801	Complete Gold Data promotion path for SFT/eval/GraphRAG outputs	Powers GraphRAG and demos
3	EMPIRE-811	ADR: decide streaming substrate once (NATS JetStream vs Kafka/Redpanda)	Before any stream is built
4	EMPIRE-798	Define GraphRAG context block API for router and agents	Before router code hardens (Chapter 11)
5	EMPIRE-809	Real-time comprehension conveyor: silver tap → stream → workers → durable replay	Depends on substrate ADR
6	EMPIRE-810	Atom comprehension engine: function/class atoms, dedup, content-addressed analysis store	Depends on conveyor
7	EMPIRE-812	Comprehension-canvas wedge: CanvasChangeController and lens scoping	Depends on atom store
8	EMPIRE-796	Design open-core GraphRAG-aware multi-LLM smart router	Depends on context block API
9	EMPIRE-797	Prototype router MVP as OpenAI-compatible BASE_URL replacement	One mock/local + one external provider
10	EMPIRE-803	Wire Orca Guard into router and MCP governance gateway	Guard preflight in router
11	EMPIRE-795	Build Certify & Harden CI gate for open models	Step 1 product
12	EMPIRE-799	Create hardened model registry and attestation catalog	Router checks attestation before routing
13	EMPIRE-802	Build ephemeral edge GraphRAG runtime profile	Air-gapped/local graph
14	EMPIRE-804	Define A2AMesh deployment profiles for secure model and GraphRAG routing	Hybrid/air-gapped distribution

After the core router and registry are in place:

STEP	TICKET	WORK
15	EMPIRE-805	Generate compliance evidence packs for model routing and Data Factory
16	EMPIRE-808	Publish hardened open-model leaderboard and demo evidence path
17	EMPIRE-806	Package Orca Secure AI Fabric for SMB, Enterprise, and cloud deployment

The build order is not arbitrary. It front-loads the decisions that other work depends on (naming, substrate ADR, context block API) and defers the packaging work until the components are smoked. The comprehension layer (steps 5–7) is built before the router (steps 8–10) because the router needs context blocks, and context blocks are more valuable when they include structured atoms from the comprehension layer.

Competitive Moat Analysis

The competitive discovery document maps the landscape across three tiers. The analysis is not about naming enemies — it is about understanding which pieces of the Orca stack are copyable and which are not.

TIER 0: DIRECT STRATEGIC THREATS

COMPETITOR	WHAT THEY SELL	THREAT TO ORCA
OpenBox AI	Runtime governance, trust scoring, cryptographic audit trails	Highest direct governance/provenance threat — can copy "receipts + audit" language fast
Obot AI	Enterprise MCP Gateway, MCP catalog ($35M seed)	Strongest funded MCP infrastructure threat
Palma.ai AgentGateway	Enterprise MCP gateway with policy, approvals, on-prem, NATS references	Direct on MCP/RBAC/approval and pub-sub story
Kosli	SDLC governance for AI-assisted delivery, cryptographic evidence ($10M Series A)	Strong in regulated SDLC evidence; could move upstream
Backplanes/ Spotlight	Claude Code session reports, local redaction	Direct capture/reporting competitor; if they add governance + Foundry, they become serious
XHawk	Software factory, audited agent work, compounding knowledge layer	Closest to "agent work becomes organizational memory"
LineageLens	Provenance layer, prompt/model/context capture, hash chain, signed AI BOM	Strong message-market fit around "proof, not observability"

TIER 1: COMPONENT COMPETITORS

Claudoscope, ClawMetry, ClawSecure, Pylar, Context Overflow, DebugBase, Weavable, Graphiti/Zep, and Marmot each attack one piece of the stack — local viewing, observability, security, data access, agent memory, or GraphRAG primitives. None matches the complete Orca bundle.

TIER 2: PLATFORM GRAVITY

Anthropic, OpenAI, Google, Cursor, and Microsoft/GitHub own client-side gravity and cloud-side governance. They can absorb session sharing, local governance, and enterprise policy into their native stacks. Orca must win local-dev and customer-owned data sovereignty — the pieces the hyperscalers cannot or will not own.

WHAT IS COPYABLE AND WHAT IS NOT

A competitor can copy a router UI. A competitor can copy "capture Claude sessions." A competitor can copy a local redaction step. A competitor can copy a signed audit export. These are individual features.

What a competitor cannot easily copy is the full stack:

```
local sanitization (Ch 6)
  + governed foundry raw → bronze → silver → gold (Ch 10)
    + GraphRAG with signed context blocks (Ch 9, 11)
      + attestation chain (Ch 8)
        + deployment portability (Ch 12, 13)
          + customer-local residency (Ch 12, 13)
            + governed data flywheel (Ch 9, 10, 14)
```

This is the full medallion pipeline plus the GraphRAG plus the attestation chain plus the deployment portability. Each piece is non-trivial. The combination is the moat. And because each customer's moat is built from their own engineering exhaust, the moat is unique per customer — a competitor cannot copy it because they do not have the customer's data.

DEFENSIBILITY PLAN

1. **Ship proof before broad claim.** One live golden demo: fake secret blocked → denial receipt → row excluded from silver/gold → GraphRAG ingest only approved internal subset. One auditor demo: pick a gold row → show lineage → verify hashes/signatures offline. One developer demo: install workstation client → Claude session captured → query GraphRAG for prior fix.
2. **Make the package hard to copy.** Document exact OS packages and smoke tests. Ship Data Factory gates as runnable commands, not slides. Include a sample evidence pack and offline verifier. Keep A2A Mesh NATS as the visible platform substrate.
3. **Own the category words.** Use consistently: "governed AI-engineering exhaust," "Data Factory for AI-assisted software delivery," "customer-owned agent memory," "signed provenance for generated knowledge," "Guarded capture, governed promotion, trusted retrieval," "audit-grade internal knowledge assets." Avoid leading with "MCP gateway," "observability dashboard," "Claude session viewer," or "AI security scanner" — those are crowded boxes.

GTM Strategy

The roadmap defines a three-tier GTM approach that matches the three-step product ladder.

SMB ENTRY

- Self-serve secure model scanner and hardening gate (Step 1 — Certify).
- Open-core router (Step 2 — Route).
- Local GraphRAG starter.
- Paid remediation, attestation, and evidence exports.

The SMB entry is designed for self-service. A technical founder or IT manager runs `orca certify model`, gets a report, and decides whether to adopt the model. The open-core router is free; the paid tier adds attestation, evidence exports, and the hardened model registry.

ENTERPRISE ENTRY

- Design partner assessment.
- Top-5-model Certify & Harden pilot.
- One governed workflow routed through Orca Guard and GraphRAG.
- Evidence pack that security/compliance can verify offline.

The enterprise entry is a design-partner engagement. ArmyknifeLabs works with the customer's platform and security teams to certify their top 5 open models, route one governed workflow through the platform, and produce an evidence pack that the customer's auditors can verify independently. This is not a demo — it is a real deployment with real models, real workflows, and real evidence.

ENTERPRISE EXPANSION

- Private hardened model registry.
- Router platform subscription.
- A2AMesh on-prem/hybrid deployment.
- Gold Data Factory and local SLM/GraphRAG lifecycle.

The enterprise expansion is where the flywheel (Step 3) kicks in. The customer's private model registry grows with every certified model. The router becomes the default `BASE_URL` for all AI traffic. A2AMesh extends the platform to hybrid and on-prem environments. The Gold Data Factory produces customer-local GraphRAG bundles and, eventually, vertical SLM artifacts. The customer's moat compounds.

RISK MANAGEMENT

The roadmap identifies four risks that must be actively managed:

1. **Generic routing is commoditized.** Keep the router tied to GraphRAG, Guard, and attestation. A router without governance is a commodity; a router with governance is a control plane.
2. **Scanning alone is table stakes.** The differentiator is remediation, signing, and governed runtime use — not just detecting issues but fixing them, signing the fix, and enforcing the signed state at runtime.
3. **Auto-patching model behavior creates liability.** Preserve before/after evals, license metadata, and rollback. If a model is patched to fix a safety issue, the patched version must be eval-tested and the original must be rollbackable.
4. **Claims must stay evidence-backed.** Treat every market statistic in the source reports as unverified until EMPIRE-807 validates it. The competitive discovery document includes a source index with URLs for every claim — the platform must hold itself to the same standard it holds others to.

The Cypher Coordination Dependency

The Orca roadmap depends on Substrate (Cypher) for eight blockers that must be resolved in sequence. These are external dependencies — Orca cannot ship the full secure AI fabric without them.

EIGHT SUBSTRATE BLOCKERS

1. `cm-receiptd` SSE stream — the event stream that Orca consumes for governance receipts.
2. Canonical `AnomalyExtra` — the standardized anomaly metadata format.
3. `AnomalyDetected` receipt extension — the receipt type for anomaly detection events.
4. Hermes identity propagation — agent identity flowing through the Substrate to Orca.
5. ATCS registry YAML — the attestation registry configuration format.
6. AgentShield hot rule import — the ability to push new enforcement rules at runtime.
7. `RuleImported` receipt — the receipt type confirming a rule was imported.
8. AgentShield MCP server smoke — the MCP interface for AgentShield enforcement.

These blockers map to the four-plane trust model from Chapter 8: AgentShield enforces, Hermes identifies, ContextOS attests, Orca observes. The Substrate blockers are the plumbing that connects the planes.

ORCA'S SEVEN-STEP SHIP ORDER

Orca's side of the coordination:

1. Hermes identity — agent identity propagation through the Substrate.
2. AgentShield SSE bridge — connecting enforcement events to Orca's event stream.
3. Looking Glass live — real-time dashboard rendering of guardrail cards.
4. Alert engine — anomaly detection and alerting.
5. Dashboards — operational visibility.
6. Intelligence tiers — structured analysis and comprehension.

7. Feedback loop — captured knowledge becoming runtime enforcement rules.

The priority rule from Chapter 8 is explicit: identity plumbing comes before UI polish. Looking Glass cannot render trustworthy guardrail cards unless every event already carries Hermes identity and ContextOS attestation. Step 1 (identity) must precede step 3 (dashboard).

Connecting the Themes

This book built the platform across thirteen chapters. The 2027 roadmap is where the themes converge.

THEME 1: EDGE-TO-ORIGIN GOVERNED DATA PIPELINE

The capture → raw → bronze → silver → gold → GraphRAG → agent context pipeline is the product. The router (Step 2) is the front door that makes the pipeline accessible to any OpenAI-compatible client. The comprehension layer adds stream-time understanding between silver and GraphRAG. The flywheel (Step 3) closes the loop: agents read the graph, produce better work, and the work is captured and refined back into the graph.

THEME 2: FOUR-PLANE TRUST SEPARATION

AgentShield enforces, Hermes identifies, ContextOS attests, Orca observes. The router composes all four planes: Guard preflight (enforce), tenant/policy context (identify), signed governance receipts (attest), audit events (observe). The Cypher coordination dependency is the plumbing that connects the planes across the Substrate.

THEME 3: THE GOVERNED DATA ENGINE IS THE MOAT

A model is perishable; the data engine compounds. The Certify and Harden gate (Step 1) ensures that models entering the environment are safe. The router (Step 2) ensures that every request is governed. The flywheel (Step 3) ensures that the corpus grows with every session. The competitive moat is the full stack — local sanitization + governed foundry + GraphRAG + attestation chain + deployment portability — not any single feature.

THEME 4: LOCAL-FIRST, DATA-RESIDENCY-BY-DEFAULT

Customer Orca starts blank. The moat is on-prem residency plus governed refinement plus the customer owns their own moat. The edge GraphRAG runtime profile (EMPIRE-802) and the A2AMesh deployment profiles (EMPIRE-804) extend this to hybrid and air-gapped environments. Cloud federation is opt-in, summaries only.

THEME 5: COMPOSABLE DEPLOYMENT FROM LAPTOP TO FLEET

The same Docker Compose file runs on a laptop and in production. The GKE manifest is Kubernetes-shaped and portable to EKS/AKS. The Cloudflare edge is replaceable with customer ingress. The installers (Chapter 13) handle Linux, macOS, and Windows with per-OS persistent capture. The 2027 roadmap extends this composability to the router (OpenAI-compatible, deployable anywhere) and the GraphRAG (local, ephemeral, or air-gapped).

Patterns and Anti-Patterns

PATTERNS THAT ENDURE

Pattern 6: Private-by-Default Scoping. The customer owns their own moat. Cloud federation is opt-in. The GraphRAG bundle is customer-local. The flywheel compounds inside the customer's trust boundary. This pattern runs from the installer (Chapter 13) through the deployment (Chapter 12) to the 2027 roadmap.

Pattern 10: Hash-Only Provenance. The signed provenance chain is the competitive advantage. Every model attestation, every context block, every governance receipt, and every Data Factory manifest carries a hash that can be

verified offline. The competitive brief identifies this as the line Orca must not let remain hidden in engineering docs — ship the offline verifier and a sample evidence pack with the package.

Pattern 3: Byte-Offset Checkpointing / Replay Safety. The trust layer must be durable across the flywheel. If the graph is rebuilt, the context blocks must be reproducible. If a receipt is challenged, the evidence must be retrievable. The reproducibility envelope (SHA-256, record count, schema version, filter versions, generation command, git commit) ensures that every artifact can be regenerated from manifest + code commit.

ANTI-PATTERNS TO RESIST

Anti-Pattern 9: Treating the Model as the Moat. The data engine compounds, models are perishable snapshots. The roadmap is explicit: fine-tuning is an optional downstream, never the goal. The specialist model strategy — if scoped — builds a family of 13B LoRA specialists, not one generic model. But even the specialists are proof, not product. The product is the governed data flywheel.

Anti-Pattern 7: Premature Horizontal Scaling. The router scales via edge + web replicas, not sink replicas. The sink remains SQLite single-writer until the Postgres adapter ships. The GraphRAG scales via signed bundles and edge runtimes, not by replicating the central graph. The 2027 roadmap preserves the single-writer constraint that was established in Chapter 3 and reinforced in Chapters 12 and 13.

The Bottom Line

The product is not "cheap open-source AI." The product is controlled, attested, data-sovereign open AI.

The platform that this book described — chapter by chapter, from the local watcher on a developer laptop to the GraphRAG context block served back to an agent — exists to make that product real. The 2027 roadmap is the path from the current capabilities to the full secure AI fabric. The three-step ladder — Certify, Route, Own the Flywheel — sequences the work so that revenue comes first, the recurring control plane comes second, and the compounding moat comes third.

The competitive moat is not a feature. It is the full stack: local sanitization before transport, governed foundry promotion from raw to gold, GraphRAG with signed context blocks, attestation chains with Ed25519 signatures, deployment portability from Docker Compose to GKE to air-gap, and a data flywheel that compounds inside the customer's trust boundary. A competitor can copy any one piece. They cannot copy the combination, and they cannot copy the customer's own data.

Orca is the platform that lets an enterprise say yes to open-weight models without giving up auditability, policy, or control. Every chapter in this book built one piece of that promise. The 2027 roadmap is where the pieces become a fabric.
