
runveil: A Transparent Egress Firewall and Audit Layer for Developer–AI Interactions

Dang Phan
Independent Developer

Abstract

Developers increasingly delegate work to third-party AI tools — Claude Code, Cursor, GitHub Copilot, ChatGPT — that read local files and transmit source, secrets, and customer data to external model providers over TLS. Organizations have little visibility into, and no control over, this "Shadow AI" egress, and existing data-loss tooling does not see it. We present runveil, an open-source system that places a transparent, tool-agnostic enforcement and audit layer at the boundary where humans and their AI agents send data out. runveil consists of (i) a local forward-proxy agent that intercepts AI-bound HTTPS using a device-local certificate authority, inspects request bodies with content-aware detectors (credential scanning, path/DLP rules), and applies graduated actions — allow, redact, warn, block — inline before traffic leaves the machine; and (ii) a multi-tenant control plane that distributes versioned policy and aggregates a per-request audit trail with analytics, retention, and export. Because interception is transparent, runveil captures egress that opt-in AI gateways structurally miss, and it functions without cooperation from the AI tool — a property that matters as coding tools become autonomous agents. We describe the architecture and policy model, and verify the end-to-end system: device-token authentication, ETag/304 policy caching, and audit ingestion all behave to specification, the enforcement pipeline correctly blocks/redacts representative secret and sensitive-path payloads, and the implementation is covered by 417 agent and 36 control-plane automated tests. We argue runveil is a practical oversight substrate for human–AI data flows and a complement to model-side safety.

1. Introduction

AI coding assistants have moved from autocomplete to agents that act: they read a repository, decide which files are relevant, and send their contents to a remote model. The same is true of chat tools into which developers paste logs, stack traces, and configuration. Every one of these interactions is an egress event — proprietary or sensitive data crossing the organizational boundary to a third party — and it travels as ordinary TLS to a vendor API. Conventional data-loss prevention (DLP) was built for email and file uploads and does not inspect this channel; AI "gateways" only observe calls that an application is explicitly configured to route through them. The result is a visibility and control gap precisely where the highest-value data now flows.

We frame this as an AI oversight problem. As organizations adopt AI, a basic governance requirement is the ability to see and control what their people — and the autonomous tools acting on their behalf — send to external models, and to retain an auditable record of it. This is the human side of AI safety: not aligning the model, but governing the interface. A control point here is also a defense that does not depend on the AI tool's cooperation, which is important when the tool is an agent making its own decisions about what to read and transmit.

Threat model. A developer (or an agent running with their credentials) transmits secrets, credentials, source, or regulated data to a third-party model. The organization wants (a) inline prevention of the most dangerous cases, (b) graceful degradation (redaction) rather than blanket blocking, and (c) a tamper-resistant audit trail for incident response and compliance. We assume the organization controls its endpoints (can install software and a trust anchor) and that traffic uses HTTPS to known AI hosts.

Our main contributions are:

- 1. A transparent, tool-agnostic interception architecture for AI egress: a local CA + forward proxy that observes and enforces on AI-bound traffic without modifying the AI tool, capturing Shadow AI that opt-in gateways miss and functioning against autonomous agents.*
- 2. An inline content-aware policy engine at the egress boundary — credential and path detectors with graduated actions (allow/redact/warn/block) — driven by a small, auditable YAML policy grammar.*
- 3. A multi-tenant control plane that turns per-endpoint enforcement into organizational oversight: centrally edited, versioned policy with fast rollout, plus an append-only audit trail with analytics, retention, and export — and which, by design, stores findings and metadata, not message bodies.*
- 4. A working, open-source end-to-end system released under an open-core model, with functional verification of its core guarantees.*

2. Related Work

AI gateways / proxies (LiteLLM, Cloudflare AI Gateway, Portkey, Helicone) sit in front of model APIs to add caching, routing, rate-limiting, and logging. They are opt-in: an application must be configured to send traffic through them, so they observe only sanctioned calls and cannot see a developer's Claude Code session or a pasted ChatGPT prompt. runveil differs by intercepting transparently at the endpoint, capturing arbitrary AI egress without per-tool integration.

Enterprise DLP / CASB and AI-security vendors (Netskope, Zscaler, Nightfall, Harmonic, Prompt Security) increasingly advertise GenAI coverage, typically via network-level cloud proxies or browser agents. These are powerful but heavyweight, closed, and oriented to large enterprises; runveil targets the developer endpoint with an open-source, agent-first design and an explicit "no bodies stored" posture, and is specifically aimed at coding-agent traffic.

Secret scanners (gitleaks, trufflehog, detect-secrets) find credentials in repositories or commits — at rest, not in flight. runveil applies similar detection at egress time, on the actual outbound payload, where a block can still prevent the leak.

LLM guardrail frameworks (e.g., Meta's LlamaFirewall/CodeShield, NeMo Guardrails) filter model inputs/outputs inside an application. runveil is complementary: it operates below the application, on the wire, independent of the model and tool.

Someone would use runveil over these when they need visibility and enforcement over unmanaged AI tool usage on developer machines, transparently and from open source, with an organization-level audit trail — a combination none of the above provides.

3. Methods

System overview (Figure 1). runveil has two components communicating over two stable HTTP endpoints; everything behind them is independent.

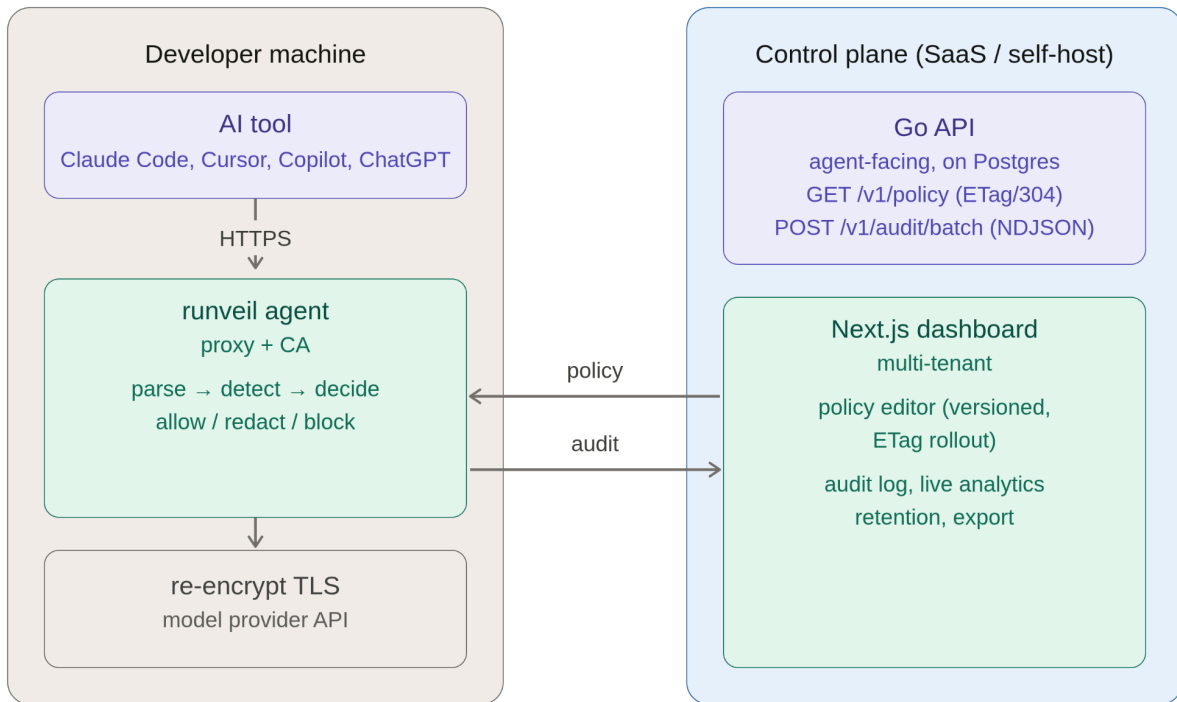


Figure 1. Transparent interception at the endpoint; centralized policy and audit at the control plane.

Agent (Go). `runveil init` generates a device-local CA and installs it into the OS trust store; the agent then runs a forward HTTPS proxy on localhost and the developer points their tools' `HTTPS_PROXY` at it (one environment variable, or our background service sets it). For AI-bound connections the agent terminates TLS, parses provider-specific request bodies (OpenAI/Anthropic shapes), and runs a pipeline of stages: a secret scanner (pattern + entropy detectors for keys/tokens), and a path scanner that extracts file paths from agent tool_use blocks. Each finding is scored, the active policy is consulted, and an action is applied: forward unchanged (allow), forward with sensitive spans masked (redact), forward but flag (warn), or refuse with 403 (block). The agent writes a structured audit record — `request_id`, `time`, `host`, `method`, `path`, `status`, `decision`, `findings`, `bytes_in/out`, `duration_ms`, `vendor`, `endpoint`, `device_id` — to a local JSON-Lines log and, when configured, batches it to the control plane. Configuration is a single `config.json` (`control_plane_url`, `device_token`) written by `runveil enroll`; the agent runs as a background service (`systemd/launchd`).

Policy grammar. Policies are small, reviewable YAML: `version`, then `rules[]`, each with a `match` (one of `pattern`, `severity`, `all`, or a `path glob`) and an `action` (`allow|block|warn|redact`). This keeps enforcement legible and lets per-project differences be expressed as path-scoped rules within one policy rather than as separate configurations.

Control plane. A Go service exposes the two agent endpoints on Postgres: `GET /v1/policy` (device-token auth, `ETag/304` so idle agents poll cheaply) and `POST /v1/audit/batch` (NDJSON). A Next.js app provides multi-tenant management (GitHub OAuth, role-based access): a validated, versioned policy editor with ~30-second rollout via `ETag` change, an audit log with per-request detail, a live view, analytics (volume/error/latency, per-host and per-device), per-tenant retention, and CSV export. Device tokens (`dt_...`) are stored only as sha256 hashes; a stable `device_id` identifies a machine.

Key design choices. Transparent MITM (rather than opt-in routing) is what lets us see unmanaged tools and agents — the central requirement. Bodies are never stored centrally — only findings and metadata — making the audit trail useful without itself becoming a data-exfiltration risk. `ETag/304` + NDJSON keep the agent↔plane protocol cheap and were held byte-stable throughout development so deployed agents never break. Open-core (free agent, paid plane) lowers adoption friction for a security tool that must earn trust.

What is hard / did not work cleanly. TLS interception requires installing a trust anchor (a privileged, visible operation) and is bypassed by certificate-pinned clients; body inspection is provider-shape-specific, so a new vendor needs a parser; detection is heuristic (regex/entropy), with the usual false-positive/negative trade-offs.

Reproducibility. The system is open source and installs with one line (`curl ... | sh`); the agent runs entirely locally for evaluation, and the control plane runs against any Postgres. The full test suites reproduce the functional claims below.

4. Results

We report functional and contract correctness of the system rather than a detection-quality benchmark (which we scope as future work, Section 5).

Enforcement behaves correctly on representative payloads. Table 1 shows decisions produced by the agent's end-to-end pipeline on crafted requests, taken from the integration test suite.

#	Outbound payload (to a model API)	Detector	Decision	HTTP
1	Chat completion containing <code>aws_access_key_id</code> + <code>aws_secret_access_key</code>	secret-scan (2× high)	block	403
2	Message containing a single AWS key	secret-scan (high)	redact	200 (modified)
3	Agent <code>tool_use</code> reading <code>.../payments/charge.go</code>	path-scan, rule <code>block-payments</code>	block	403

4	Ordinary completion, no sensitive content	—	allow	200
---	---	---	-------	-----

Table 1. Representative inline decisions from the integration suite. Each row is an actual classified request; "modified" denotes redacted egress.

The agent↔control-plane contract is correct. In an end-to-end run against a freshly migrated Postgres, an unauthenticated GET /v1/policy returns 401; with a valid device token it returns 200 and an ETag; a conditional request with the matching If-None-Match returns 304; and a POST /v1/audit/batch of NDJSON records returns 2xx, after which the rows are present in the database, each stamped with the originating device_id. This confirms device-token authentication, policy caching, and audit ingestion end-to-end.

The implementation is well-tested. The agent is covered by 417 automated tests (run with the race detector), spanning the detectors, the proxy/TLS path, policy loading, and integration; the control plane is covered by 36 tests using ephemeral Postgres containers. All pass in CI across Linux, macOS, and Windows.

Overhead. Inline processing adds modest per-request latency; against a local mock upstream in the integration tests we observe single-digit-millisecond proxy overhead. We do not claim a real-world latency figure — wide-area network time dominates, and a rigorous measurement against live provider APIs is [TODO: team to measure].

Robustness of claims. Our claims are functional (the system does what it says) and are backed by reproducible test suites and a live contract check, not by a single run. We deliberately do not assert detection precision/recall, because we have not yet evaluated against a labeled corpus; making that claim without that evidence would be unsound.

5. Discussion and Limitations

Implications for AI safety. runveil is infrastructure for human oversight of AI usage. Model-side safety (alignment, refusal training, guardrails) governs how a model behaves; runveil governs the interface — what leaves the organization toward the model and what record remains. As coding tools become autonomous agents that choose what to read and send, an enforcement point that does not rely on the agent's cooperation becomes more valuable, not less: it is a control that holds even if the agent is buggy, jailbroken, or adversarial about its own logging. The "store findings, not bodies" design also illustrates a pattern for monitoring that minimizes the new attack surface it creates.

Limitations. (1) Trust and bypass: interception requires installing a CA (privileged, and a target for tampering) and is defeated by certificate pinning; an insider can disable a local agent absent MDM-level enforcement. (2) Coverage: body inspection is provider-specific; unparsed providers are passed through opaquely. (3) Detection quality: regex/entropy detection has false positives and negatives, and we present no quantitative precision/recall — this is the most important open

evaluation. (4) Trust in the audit: records are append-only but not yet cryptographically tamper-evident. (5) Scope: we assume org-controlled endpoints; BYOD and pinned apps are out of scope. If the assumption that the organization can install software and a trust anchor does not hold, the transparent-interception approach does not apply and the contribution reduces to the policy/audit model behind an explicit proxy.

Future work. A labeled benchmark for detection precision/recall; additional provider parsers; semantic/PII detection using small models; cryptographic hash-chaining of the audit log for tamper-evidence; continuous SIEM forwarding; policies over agent actions (not just content); and time-partitioning for audit at scale.

6. Conclusion

We built runveil, a transparent, open-source egress firewall and audit layer for the traffic developers and their AI agents send to third-party models. By intercepting at the endpoint rather than asking tools to opt in, it provides visibility and inline control — allow, redact, warn, block — over Shadow AI that current gateways and DLP miss, and it pairs per-machine enforcement with an organization-level, body-free audit trail. We verified the system's core guarantees end-to-end and ship it as a working artifact. We see this as a small but concrete piece of AI-oversight infrastructure: governing the human–AI interface is a necessary complement to governing the model, and it becomes more pressing as the tools on the other side of that interface start to act on their own.

Code and Data

- **Code (agent):** <https://github.com/dawnn07/runveil> — open source; one-line install `curl -fsSL https://raw.githubusercontent.com/dawnn07/runveil/main/install.sh | sh`.
- **Data:** none collected; the system stores per-request metadata and findings, not message bodies. No external dataset was used.
- **Demo:** <https://runveil-cloud.vercel.app/>

References

1. LiteLLM. LiteLLM: Call 100+ LLM APIs. <https://github.com/BerriAI/litellm>
2. Cloudflare. AI Gateway. <https://developers.cloudflare.com/ai-gateway/>
3. Gitleaks / TruffleHog — secret scanning. <https://github.com/gitleaks/gitleaks> · <https://github.com/trufflesecurity/trufflehog>
4. Meta. LlamaFirewall / CodeShield.

Appendix

We used Claude (via Claude Code) extensively to design and implement the system — architecture, agent pipeline, control plane, and dashboard — and to draft this report. All functional claims were independently verified by running the automated test suites (417 agent + 36 control-plane tests) and an end-to-end contract test (auth, ETag/304, audit ingestion) whose outputs are reported in Section 4; no quantitative result is asserted that we did not observe. The final manuscript was reviewed and edited by the authors.