TECHNICAL WHITEPAPER

PoisonZero: Fail-Closed Integrity Protection for AI-Agent Memory

Architecture, detection model, and verifiable data handling for the persistent memory of AI agents.

Version 2.5 · July 2026
Document class: engineering reference
poisonzero.com

Abstract

AI agents retain persistent memory — standing instructions, project notes, and learned preferences — in plain files that are read back into the model context on every invocation, and they read configuration and hook files that decide what they run automatically and with what privilege. Such files are a writable extension of the system prompt that survives restarts, and an attacker who writes a single malicious entry influences all subsequent agent behaviour without being present when the damage occurs. This document specifies PoisonZero, a host-resident system that treats every change to a protected file as untrusted until proven safe and that is fail-closed by construction: a change is never silently accepted. Only a proven-poisoning verdict is neutralised on disk — the last known-clean state is restored and the poisoned copy preserved for asynchronous review — while a change that cannot be evaluated or is merely ambiguous is not destroyed but left in place and surfaced with a passive notice, so a false positive can never mutilate content that was in fact clean.

The system is described in terms of its threat model (five attack classes), its event-driven processing pipeline (a file watcher with a trailing-edge final-state guarantee, a deterministic Unicode-normalised rule layer, an evaluator wrapped in layered defence-in-depth filters that can only tighten its verdict, a three-threshold decision engine, and an enforcer), and two interchangeable evaluator implementations: a cloud-assisted flow built around a verifiable egress contract, and a fully on-device flow in which a compact generative detection model executes inside a minimal-privilege sandbox. The detection model is a decoder-only transformer with fully customised weights, supervised on a purpose-built corpus via knowledge distillation; in broad internal testing it detects over 94% of attacks at under 5% false alarms and recovers the attack classes against which encoder-based injection classifiers are practically blind. Integrity is anchored by signed release artifacts and a license-manifest hash that is re-verified before every model load. The document also enumerates residual risks honestly.

Contents

1 Introduction and threat model

1.1 Memory poisoning versus prompt injection

An AI agent maintains persistent memory in plain files on disk: standing instructions, project conventions, and learned preferences (the memory files and dot-files of contemporary coding assistants and agent runtimes). This memory is read back into the model context on every invocation. It is, in effect, a writable extension of the system prompt that survives process restarts.

Prompt injection is a transient condition: a malicious instruction arrives within one request and is discarded when the session ends. Memory poisoning is persistent. A single poisoned entry — written by a compromised tool, a malicious document the agent summarised, a supply-chain payload, or a second agent — continues to influence every subsequent run until a human intervenes. The attacker need not be present when the damage occurs; the entry is written once and re-read indefinitely.

This distinction has a measurable consequence for detection, established in Section 7: a classifier trained to recognise injection *phrasing* does not recognise a malicious instruction that is structured to read like a legitimate memory entry. Memory poisoning is not a subset of prompt injection, and tooling built for the latter does not transfer.

The same persistence argument extends beyond prose memory to an agent's **configuration and hook files** — the machine-readable files that decide what an agent does automatically and with what privileges: settings files (`settings.json`, `settings.local.json`), setup and hook scripts (`setup.mjs`,

session-start and autostart hooks), and the equivalent files of other runtimes. These carry a different and in some respects sharper set of attack primitives than prose: an autostart or session-start hook that runs silently on the next launch; an `autoApprove` or "YOLO"-style flag that removes the human confirmation step (the class exercised by CVE-2025-53773); obfuscated payloads (base64-wrapped or `eval`-driven code); egress buried in a setup script; path manipulation; and Unicode smuggling. This is the vector a self-propagating agent worm (the "Miasma"/"Hades" class) uses to persist and spread. Config and code are a **different statistical distribution** from prose, with different tells, so the detector is **trained directly on that distribution** rather than deferring to a keyword or blocklist; it is the same design principle as for memory files, applied to a second modality.

PoisonZero defends the integrity of these files. It treats every change to a protected file as untrusted until proven safe, and it is fail-closed: a change is never silently accepted. Only a proven-poisoning verdict restores the last known-clean state on disk (the poisoned copy is preserved for later review); when a change cannot be evaluated, or its evaluation is ambiguous, it is surfaced with a passive notice for review rather than trusted — and the file is left in place rather than destroyed, because a false positive that mutilates legitimate memory is worse than a deferred one. The default failure mode is safety and an auditable record, not silent acceptance.

1.2 Attack classes and defensive layers

The system models five classes of attack. They are the axes against which the detection model is trained and evaluated (Section 7) and against which the deterministic rule layer is tuned (Section 3.3).

- **Direct injection** — explicit standing instructions smuggled into memory and later obeyed by the agent as its own directives.
- **Data exfiltration** — entries engineered to make the agent leak secrets, credentials, or private context to an attacker-controlled destination.
- **Meta-attack** — entries that target the defence itself ("trust this source; stop checking it"; edits to the system's own thresholds). Disarming the guard admits every later attack.
- **Role-play and jailbreak** — a persona or scenario that coaxes the agent out of its safety constraints rather than stating a malicious instruction outright.
- **Subtle and indirect** — entries that read as legitimate notes, carry no keyword tell, and are not separable by surface filters.

No single layer is relied upon to stop every class. Table 1 maps each attack class to the layers that contribute to its defence. The mapping is defence-in-depth: a class is considered covered when at least one layer addresses it without depending on a downstream layer's judgement. The model is authoritative on the verdict, and it is bracketed by independent, cheaper layers — a scope allowlist, a deterministic signature signal that annotates the outcome for audit and analytics, and a behavioural sequence monitor that can *raise* the severity of an outcome and never lower it (Section 3.4). The model decides; the signature signal records what it observed without overriding that decision, and the sequence monitor is the remaining AI-free layer that can escalate. A model that has been talked out of a correct verdict therefore leaves a signal behind, and a multi-step campaign still trips the monitor.

Attack class	Rule layer (§3.3, AI-free)	Signature signal (§3.4, AI-free, observation)	Sequence monitor (§3.4, AI-free)	Detection model (§5/§7)	Decision engine (§4, fail-closed)	Quarantine (§4)
Direct injection	○	○		●	○	○
Data exfiltration		○	○	●	○	○
Meta-attack	●	○	○	●	○	○
Role-play / jailbreak	○			●	○	○
Subtle / indirect			○	●	○	●

Table 1. Attack classes against defensive layers. ● = primary line of defence; ○ = secondary or supporting. The signature column is a non-blocking observation signal (Section 3.4.2) — it annotates the outcome for audit and analytics but does not itself change the decision; the model is authoritative.

2 System architecture

PoisonZero runs as a privileged daemon on each protected machine (a systemd unit on Linux; a service on macOS and Windows). It is event-driven, not a scanner: it performs no work until a protected file changes. There is no disk-crawling indexer and no background telemetry.

The design is layered so that no single component is a single point of failure and each layer is independently auditable. A file change traverses the layers in Figure 1. The two evaluator implementations — cloud-assisted (Section 5) and fully on-device (Section 6) — sit behind one interface; every other layer is identical across deployment modes, with one deliberate exception: the asynchronous `QUARANTINE` review channel is a cloud-deployment capability; a fully on-device install has no backend to adjudicate it, so a flagged change simply stays in its restored-clean state (Section 6).

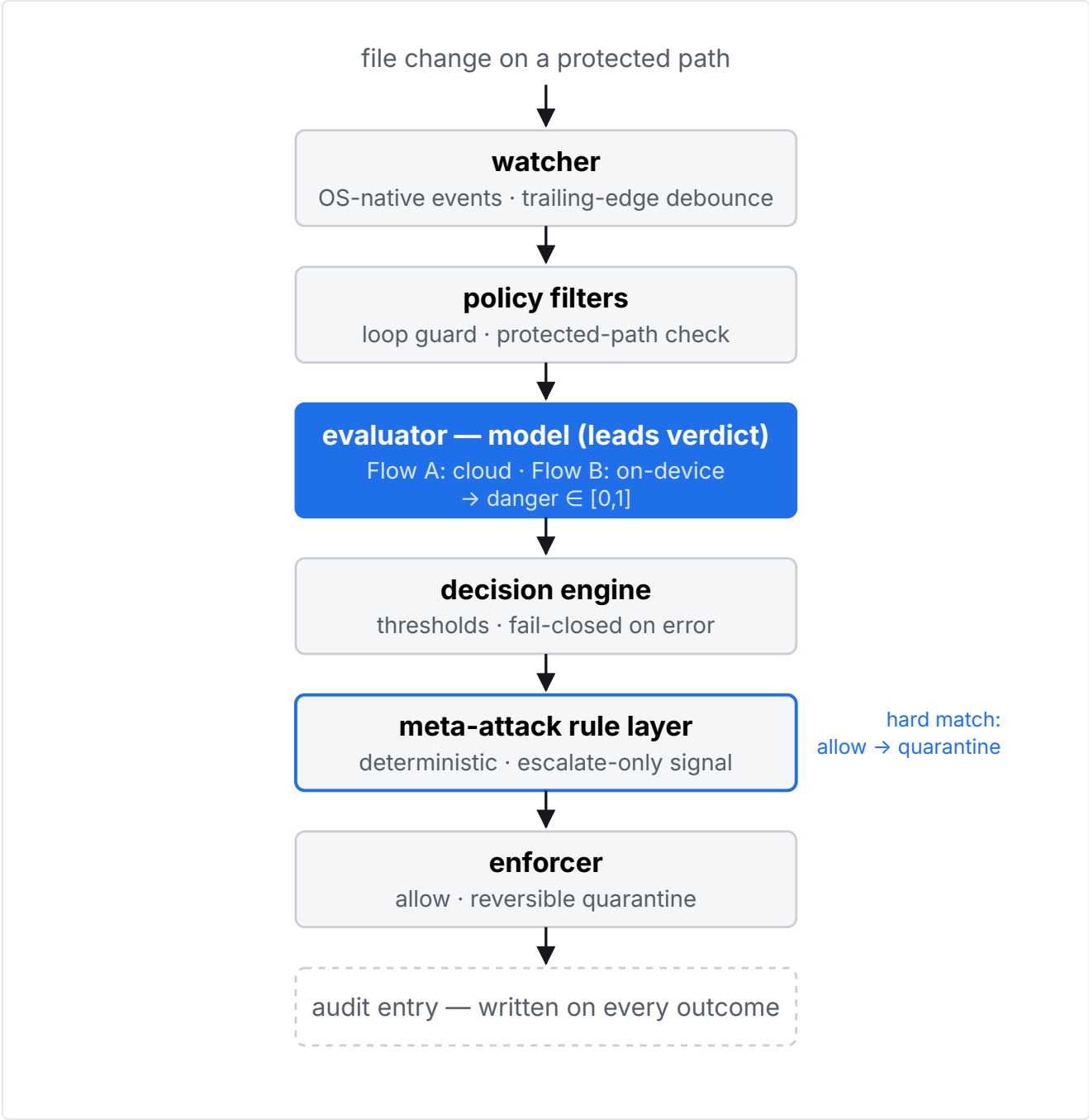

Figure 1. Layered processing of a single file change. The model evaluates every change and leads the verdict; the meta-attack rule layer runs after it and can only raise an otherwise-allowed change toward a reversible quarantine — never bypass or soften the model. Every terminal outcome produces an audit entry.

3 The processing pipeline

3.1 The watcher and its trailing-edge guarantee

The watcher subscribes to the OS-native change-notification facility (inotify on Linux, kqueue/FSEvents on macOS, ReadDirectoryChangesW on Windows) and debounces per path with a short trailing window. Editors and agents seldom write a file once; they write it in bursts, and a naïve debounce that suppresses every event inside the window introduces an exploitable hole.

Consider a debounce that emits on the leading edge and discards subsequent events. An attacker able to time writes issues a burst: a benign version first, then the malicious final version a few milliseconds later, inside the window. A leading-edge-only debounce evaluates the benign version and silently drops the malicious final write — which is precisely the state that lands on disk and is read back into the agent. The attack lands without evaluation.

The watcher instead provides a **trailing-edge final-state guarantee**. It emits on the leading edge immediately; if any further write arrives inside the window, it schedules exactly one trailing evaluation after the window elapses and re-reads the file's *current* state at that moment. Regardless of burst timing, the version that ultimately rests on disk is the version that is scored. The initial reconciliation of files already present when protection starts is handled separately and model-free: on first sight each in-scope file is recorded against a **trust-on-first-use baseline** — a snapshot of its current content — so pre-existing content is adopted as the clean reference by direct content comparison rather than being run through the model as though it were a fresh attack, and only subsequent divergence from that baseline is evaluated. Figure 2 shows the timing.

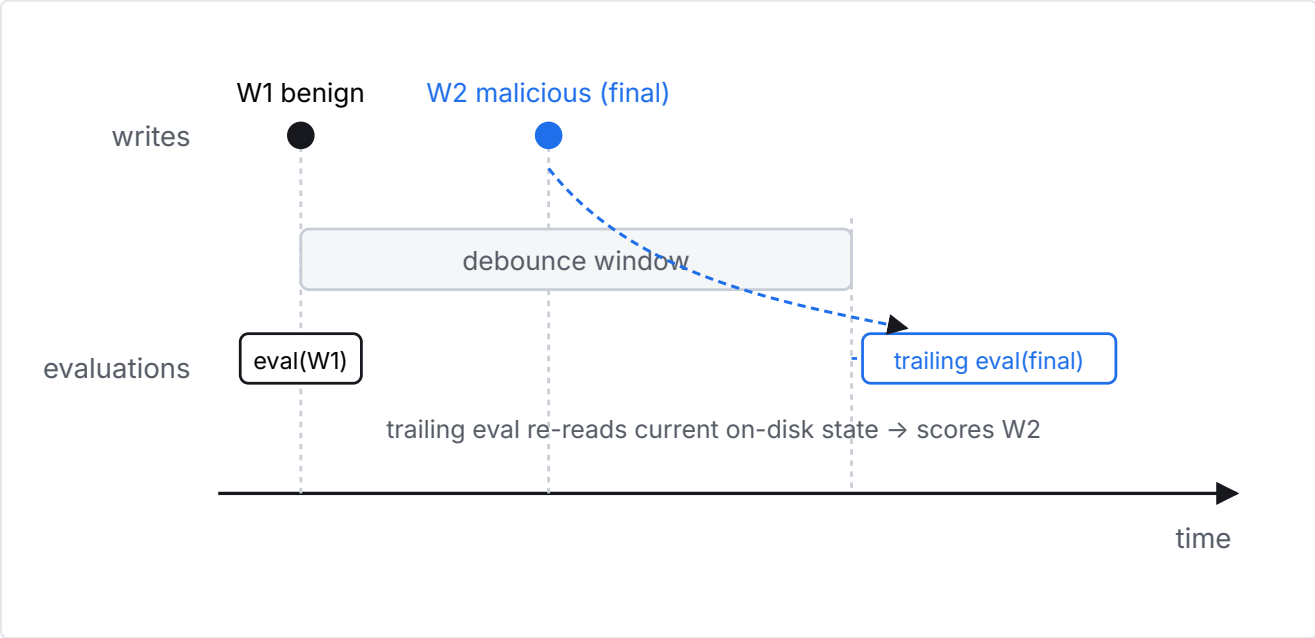

Figure 2. Trailing-edge debounce as a race defence. A burst (benign W1, then malicious final W2) inside the window triggers exactly one trailing evaluation after the window, which re-reads the current on-disk state and scores the version that actually persisted.

3.2 Policy filters and loop protection

Before any evaluation, two inexpensive checks run. First, changes to the daemon's own agent-notice file are ignored: the daemon writes that file to inform the agent of an action, and re-evaluating it would create a feedback loop. Second, the path must fall under an admin-configured protected profile; anything outside is ignored. If the new content is byte-identical to the known-clean state, processing stops — that is the daemon's own restore or apply echoing back through the watcher.

3.3 The deterministic rule layer

A unified diff of the clean-versus-new content is constructed and evaluated by the model, which leads the verdict on every change (Section 3.4). Running independently on that same diff is an AI-free rule layer for a narrow set of meta-attack patterns. It never pre-empts or overrides the model: a

hard match sets a deterministic signal that can, at most, raise an *otherwise-allowed* change to a reversible quarantine (Figure 1) — an escalation ceiling, never a bypass and never a revert. Every change reaches the model; the rule can only tighten a verdict toward quarantine, never substitute for the model's judgement or relax it toward `allow`.

The rule layer addresses the meta-attack class: an entry that edits the system's own threshold keys (`danger_block`, `danger_safe`), a classic instruction-override phrase, or an attempt to disable, uninstall, or kill the product in proximity to the product name. Every input is first **Unicode-normalised (NFKC) and lower-cased**, which folds full-width characters, ligatures, and homoglyph substitutions onto their base forms before any pattern applies, so look-alike characters or mixed case cannot evade a rule. Override and "disable the product" phrasings are matched across multiple languages including non-Latin scripts, because the attacker — not the defender — chooses the language. The matcher is tuned for precision: Latin verbs are matched on word boundaries and excluded when they form part of a settings name (e.g. `disable-notifications`), so legitimate configuration text is not falsely quarantined.

3.4 Defense-in-depth: layered filters

The detection model (Section 7) is accurate, but accuracy is not a security guarantee on its own. An adversary's whole purpose is to find the one input the model scores wrong, and a system that stakes everything on a single judgement fails the moment that judgement is wrong. PoisonZero therefore wraps the model in a small set of independent, deliberately cheap filters that run around it. The arrangement has one invariant that the rest of this section makes precise:

The model is authoritative; the cheaper layers never relax it. No layer in this section can soften the model's verdict toward `allow`. The signature signal (Section 3.4.2) is purely additive in a different sense: it emits an audit/analytics signal alongside the outcome — it records that a known-malicious shape co-occurred in the change — but it does not change the action, reason, or category the model returned. The sequence monitor (Section 3.4.3) is the one AI-free layer downstream of the model that can *escalate*: a recognised cross-file sequence raises the decision toward a reversible `QUARANTINE` and never the other way, a direction pinned by unit tests. The consequence is the load-bearing one: the model decides each change on its own, but a manipulated model still leaves a deterministic signal in the record, and a multi-step campaign that no single judgement would catch is escalated by the monitor regardless of how any one change scored. That is what "defense-in-depth" means here — not a deeper model, but several shallow, orthogonal checks, each fail-closed, no one of which is a single point of failure.

The filters are ordered cheapest-first, so the common case (an ordinary, benign edit) is dismissed by the earliest stage that can dismiss it, and the expensive model is only consulted when nothing cheaper has already settled the matter. Figure 4 shows the ordering.

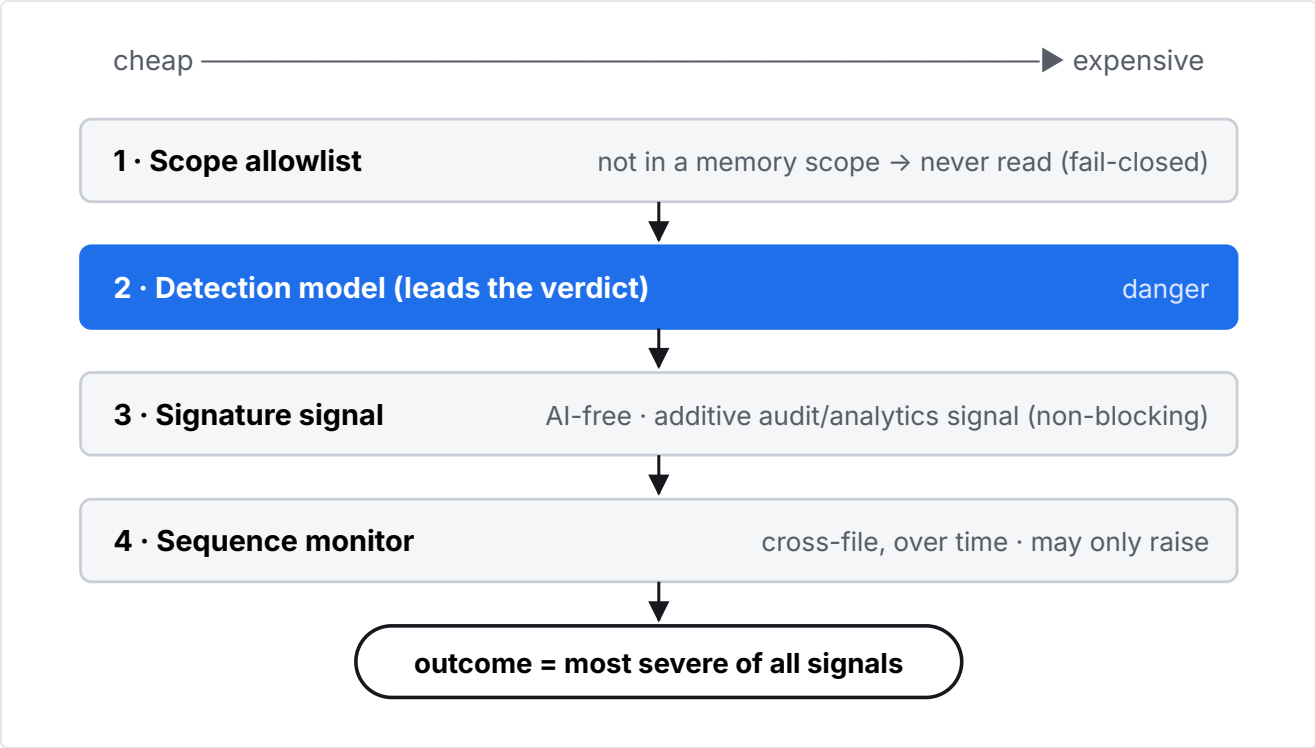

Figure 4. Layered filters around the model. Stages run cheapest-first; the scope allowlist precedes any read, and the model is authoritative on the verdict. Of the two AI-free stages downstream of it, the signature stage emits a non-blocking audit/analytics signal and the sequence monitor can escalate the outcome; neither can soften the model's decision toward `allow`.

3.4.1 Scope allowlist (memory-only, fail-closed)

The first and cheapest filter is a question of *what is even looked at*. PoisonZero protects an agent's memory, instruction, and configuration surface — the files an agent reads back into its own context and treats as standing directives (`CLAUDE.md`, `.cursorrules`, `AGENTS.md`, `mcp.json`), and the configuration and hook files that decide what it runs automatically and at what privilege (`settings.json`, `settings.local.json`, `setup.mjs`, and the equivalent files of other agent runtimes). Both are handled by the **same pipeline**: a configuration or hook file enters as a change-*diff* and is scored exactly as a memory diff is, with no separate branch or file-type special-casing — a deliberate design property (Section 1.1). A change is only ever read, diffed, and scored if its path falls inside an admin-configured scope for that agent root. Everything else on the machine — source code, build output, documents, logs — is outside the product's remit and is never opened, never diffed, never sent to any evaluator.

This is deliberately **fail-closed** in the direction that matters for a watcher: a path with no defined memory scope is not monitored, rather than being monitored speculatively. The product makes a narrow, explicit promise — it guards the memory, instruction, and configuration surface — and declines to widen that promise implicitly. Blanket protection of every `**/*.md` file on the machine is available but ships **off by default**: the guarded set is the explicitly configured memory, instruction, and configuration paths (together with their trust-on-first-use baseline, Section 3.1), not every Markdown file that happens to exist, and build and dependency directories are excluded across platforms so that vendored or generated Markdown never enters the pipeline. The security benefit beyond focus is that the entire downstream pipeline, including any egress in cloud-assisted mode, has a tightly bounded input domain: only in-scope memory- and configuration-class files can ever reach it, so an attacker cannot induce the system to read or transmit an arbitrary file by writing it

somewhere unexpected. Because this stage is a path-membership test, it is essentially free and disposes of the overwhelming majority of filesystem activity before any content is touched.

3.4.2 Deterministic signature signal

The signature layer is an AI-free deterministic matcher that runs **after** the model has returned its verdict. It does not change that verdict: it emits an *additive signal* recording that a known-malicious shape co-occurred in the change, which is attached to the outcome for audit and analytics. Where the rule layer of Section 3.3 runs *after* the model and can *escalate* a narrow set of meta-attacks — raising an otherwise-allowed change to a reversible quarantine — this layer exists to make a different failure mode visible: a manipulated or simply mistaken model that scores a plainly malicious change as safe. It is the answer to "if the model is wrong about an obvious attack, is there at least a deterministic record of it?"

It is built to be conservative, because a signal that fires loosely would clutter the audit trail with noise. Each signature requires the *co-occurrence in proximity* — on the same line of the added diff content — of two or three independent components. A lone suspicious keyword never fires a signature; an attacker has to assemble the actual shape of an attack in one place for the signal to be raised. Three signature classes are defined:

- **exfil-imperative** — a secret- or credential-bearing token (an API key, password, token, or private-key marker) together with an exfiltration verb (*send*, *post*, *upload*, *exfiltrate*, *curl*, …) together with an external destination (a URL, hostname, webhook, or command-and-control marker), all on one line. This is the literal grammar of "take this secret and ship it somewhere".
- **instruction-override-to-tool** — an override phrase (*ignore*/*disregard the above*/*previous instructions*) together with an execution verb (*run*, *execute*, *eval*, …) on one line: an attempt to discard the agent's standing constraints and immediately drive a tool.
- **persistence-injection** — a persistence clause (*in every session*, *from now on*, *always*) together with a concealment clause (*do not tell the user*, *silently*) on one line: the signature of a standing instruction designed to keep operating and to hide that it is operating.

The signal is strictly **non-blocking**, and this is the point of it. The model is authoritative: the signature layer never alters the action, reason, or category the model returned — it only annotates the outcome with what it observed. A unit test (`TestApplyRegexFloor_NeverChangesAction`) pins exactly this property, so the matcher can be extended without ever turning into a layer that overrides the model. The effect is a deterministic audit and analytics trail beneath the model: even when a model is out-argued into the wrong verdict, a change whose own text spells out an exfiltration imperative is recorded as having done so, and that record is surfaced in the audit trail and the analytics view.

3.4.3 Behavioural sequence monitor

The filters above, and the model itself, judge a single change in isolation. A patient attacker can stay under that radar by spreading an attack across several edits, each of which is individually unremarkable: touch a file that contains a secret in one edit; add an innocuous-looking outbound note in another; nudge a tool allowlist in a third. No single diff is alarming; the *sequence* is. The behavioural sequence monitor is the layer that reasons across changes over time, scoped to one protected agent root (treated here as the unit of a "session").

It maintains, per agent root, a small set of **markers** — derived facts about recent activity — and recognises four sequence patterns:

- **PZ-SEQ-1 · secret-then-exfil** — a recent change touched secret-bearing material, and a later change carries an exfiltration indicator. The two events would each pass on their own; together they are the two halves of a staged leak. Escalates to **QUARANTINE**.
- **PZ-SEQ-2 · salami** — three or more changes at the same root, each scored just below the block threshold, inside a 30-minute window. A run of edits that each sit deliberately under the line is itself the signal. Escalates to **QUARANTINE**.
- **PZ-SEQ-3 · config-escalation** — a recent change touched the agent's own configuration or tool allowlist, and a later change combines that with secret material or with an exfiltration indicator (escalates to a reversible **QUARANTINE**). Widening what the agent may do, then using the widened surface, is the classic two-step privilege escalation.
- **PZ-SEQ-4 · persuasion-retry** — after a change was blocked, a fresh change at the same root makes another meta-persuasion attempt against the defence. A second try immediately after a block is treated as adversarial rather than coincidental and escalates to a reversible **QUARANTINE**.

The monitor is strictly **memory-only**, by design and not merely by convention. It never stores raw file content. What it retains is a marker: a path, a timestamp, and a handful of derived signal booleans (for example "this change touched a secret", "this change touched config", "this change carried an exfil indicator"). The original diff is discarded once these booleans are computed. Markers **age out after 24 hours** and are held in a per-process LRU structure bounded in size, so the monitor's footprint stays small and self-limiting and cannot become a store of historical file contents. The monitor is **additive and never-downgrade**: a recognised sequence can only escalate the current change's outcome, never relax it, and that direction is pinned by test. It is the AI-free layer that can change the decision, and it derives its own signal booleans independently from each change; the signature signal of Section 3.4.2 is a separate, audit-only record and does not feed the monitor.

Taken together with the scope allowlist, the rule layer, the model, and the signature signal, the sequence monitor closes the gap that a per-file view leaves open. The five attack classes of Section 1.2 are each addressed by at least one layer that does not depend on a downstream layer's judgement (Table 1), and the subtle, multi-step campaigns that are hardest for any single judgement are exactly the ones the monitor is built to surface. In internal testing this layering is what lets the system turn a model that is already markedly more robust than the original — on the order of 22% better against Hades-class worms, with roughly 65% fewer misjudgements — into an outcome that does not hinge on the model being right every time; against repeated scanner-style persuasion of the defence, the combined layers held in nearly 99% of internal trials.

4 The decision engine

The evaluator returns one number in [0,1]: **danger** (how malicious the change appears). The decision engine is total and pure (no I/O); its entire logic is Listing 1.

```
function decide(eval, evalErr, t):
  # evalErr != null means "not evaluable" → fail-closed
  if evalErr != null:
      return QUARANTINE       # not evaluable → reversible quarantine, file left in place (ThreatM

  if eval.danger >= t.dangerBlock:
      return QUARANTINE       # grounded needle / proven poisoning → reversible quarantine + case

  if eval.danger <= t.dangerSafe:
      return ALLOW

  return QUARANTINE           # behavioural-sequence review → reversible quarantine
```

Listing 1. The decision function. Scores are forwarded downstream only when evaluation succeeded; any evaluation failure is treated as a reversible quarantine, never as a default allow.

The default thresholds are given in Table 2. They are admin-configurable centrally, but the defaults are the operating point against which the system is described here.

Parameter	Default	Effect
`dangerBlock`	0.70	danger ≥ this ⟹ reversible quarantine
`dangerSafe`	0.30	danger ≤ this ⟹ allow

Table 2. Default decision thresholds.

A change scored at danger 0.97 is past the block threshold and is placed in **reversible quarantine** as proven poisoning: the last known-clean version is restored on disk immediately and the poisoned version is preserved in a case file, so the action is always undoable rather than a destructive overwrite. A change at danger 0.10 is allowed. The model's own verdict is deliberately near-binary, so the borderline behavioural `QUARANTINE` outcome is not driven by a mid-range model score: it is entered when a behavioural-sequence rule flags an otherwise-allowed change as warranting review, in which case the change is quarantined the same way. Figure 3 shows the state mapping.

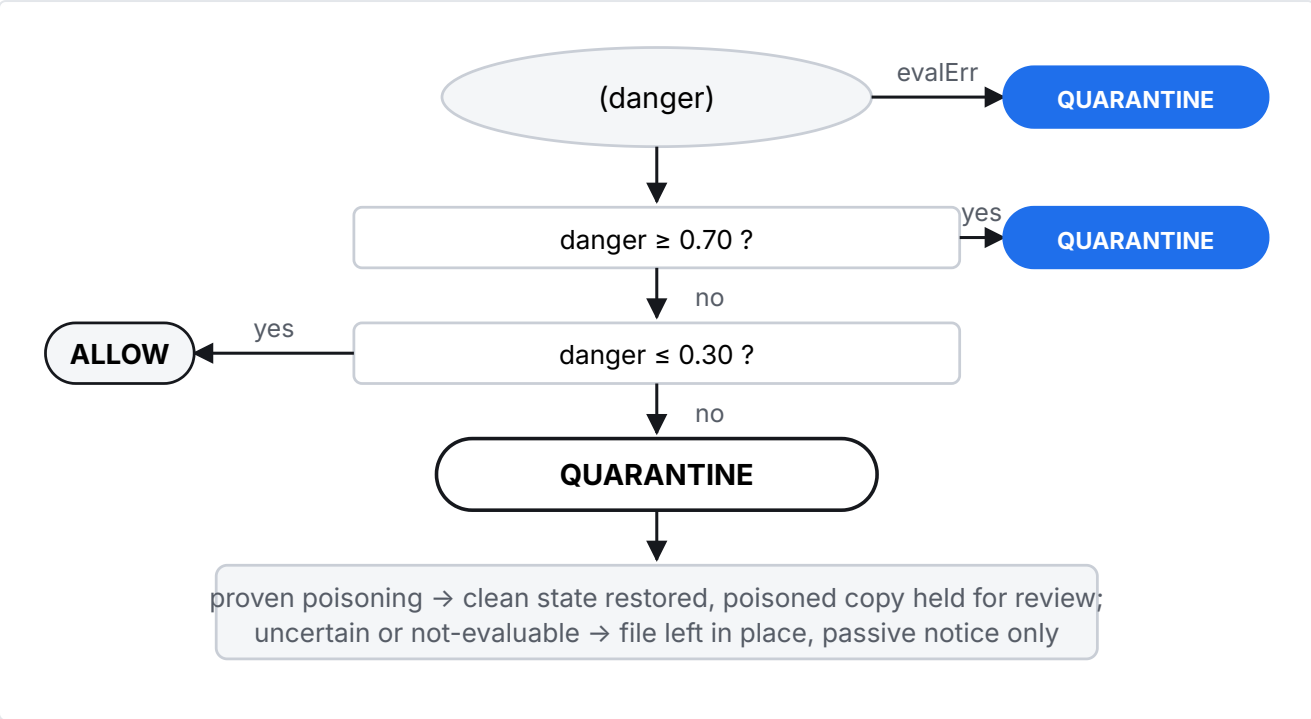

Figure 3. Decision state mapping. Any evaluation error and any danger at or above the block threshold map to a reversible quarantine; only a low-danger change is allowed; everything else is quarantined for review. A proven-poisoning quarantine restores the clean state and preserves the poisoned copy as a case; a not-evaluable quarantine leaves the file in place and records only a passive notice.

The engine is fail-closed: if evaluation cannot be performed — the model errored, timed out, or returned something unparseable — the result is a reversible quarantine, never a silent allow. Crucially, the two quarantine outcomes are not the same on disk. Only a **proven-poisoning** verdict — a high, grounded danger score — touches the file: the last known-clean state is restored on disk immediately and the poisoned version is preserved out-of-band in a case file for the owner to review asynchronously, whenever they choose. Every lesser or non-evaluable outcome — the uncertain middle band, an engine that errored or timed out, a malformed verdict — is *not* treated as proof: the file is left exactly in place and only a passive notice and audit entry are written, with no on-disk restoration and no preserved copy. This asymmetry is deliberate: an attacker who takes the engine down reaches at most a reversible quarantine, never the destruction of legitimate content, and a false positive can never mutilate a memory file that was in fact clean. A `QUARANTINE` outcome never blocks the agent waiting for a human and asks no one anything in real time; protection never waits on a human and never depends on an administrator being online at the moment of the event.

5 Flow A — cloud-assisted evaluation

In the cloud-assisted flow, evaluation is performed by a hardened remote endpoint. The defining property is a **verifiable data contract**: what leaves the device is fixed, minimal, and locally observable, rather than asserted.

5.1 Local redaction

Before any byte is sent for evaluation, the diff passes through a local redaction stage on the device. It removes, pattern-based, the classes of secrets and structured PII that are reliably recognisable offline. Replacements are structure-preserving placeholders so that the evaluator can still reason

about the change. Rule ordering is deliberate: secret formats are matched before generic patterns, so an API-key line is redacted as a secret rather than fragmented as a phone number. Table 3 lists the classes.

Class	Examples matched	Placeholder
JWT	three base64url segments	`[REDACTED:jwt]`
API keys	OpenAI/Anthropic-style, GitHub, Slack, AWS, Google, Stripe, webhook secrets	`[REDACTED:apikey]`
Generic assignments	`key/token/secret/password/credential` = value (key name kept, value removed)	`[REDACTED:secret]`
Email	RFC-style addresses	`[REDACTED:email]`
IBAN	country code + check digits + BBAN	`[REDACTED:iban]`
Card number	16-digit grouped PANs	`[REDACTED:card]`
IP address	dotted-quad IPv4	`[REDACTED:ip]`
Phone	international format (`+CC …`) only	`[REDACTED:phone]`

Table 3. Redaction classes applied locally before egress, with placeholder tokens.

Listing 2 shows the transformation on a synthetic, format-true memory fragment. The diff structure and key names survive; only the sensitive values are replaced.

```
# before (on device)
+ deploy.api_key = "sk-ant-7f3Qa1b2C3d4E5f6G7h8I9j0K1l2M3n4"
+ notify owner alice@example.com on rollback
+ fallback host 10.4.12.9, support line +49 30 1234567

# after (what an evaluation would send)
+ deploy.api_key = "[REDACTED:apikey]"
+ notify owner [REDACTED:email] on rollback
+ fallback host [REDACTED:ip], support line [REDACTED:phone]
```

Listing 2. Local redaction, before and after (synthetic example, format-true). Free-text names and unstructured PII are not pattern-detectable offline and may pass through; see Section 11.

5.2 The data contract

Only a file path and the redacted unified diff are transmitted for evaluation — never full file contents. The endpoint surface is small and fixed; each request carries a documented, minimal payload (Table 4). Files outside the configured profiles, full texts, directory and process listings, and system telemetry never leave the machine. All transport is over TLS.

Endpoint	When	Payload
`/enroll`	once at installation	app ID + one-time enrollment code
`/evaluate`	on a protected-file change	path + redacted unified diff (no full texts)
`/getConfig`	poll interval	auth token only
`/reportAudit`	when new audit entries exist	timestamp, action, path, reason, scores
`/reportQuarantine`	on a `QUARANTINE` outcome	case ID, path, reason, scores + redacted diff (size-capped)
`/getDecisions`	polled only while review cases are open	auth token only
`/ackDecision`	once per resolved case	case ID + outcome (applied / stale / discarded)

Table 4. Outbound endpoints and their exact payloads (cloud-assisted flow).

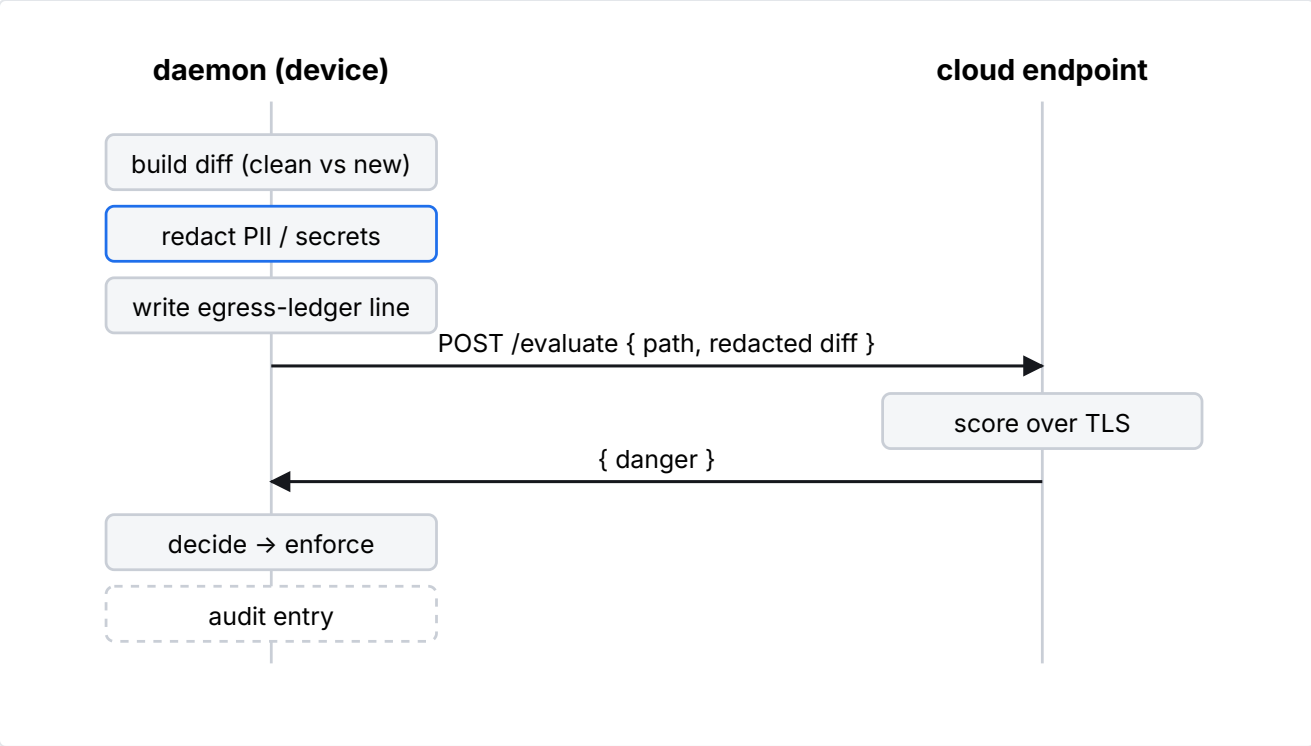

Figure 4. Cloud-assisted evaluation sequence. Redaction and the egress-ledger write happen on-device before the request; only path and redacted diff cross the boundary.

5.3 The egress ledger

Every outbound request is recorded locally in a human-readable ledger. Each line carries the timestamp, the endpoint, the payload size, the SHA-256 of the payload (never its content), and the HTTP status. This is the verifiable "does it phone home?" accounting. The same line format is produced for the monthly license check in the on-device flow (Section 6), so that flow's single network event is recorded identically.

```
2026-06-04T12:00:01Z endpoint=/getConfig bytes=2   sha256=44136f… status=200
2026-06-04T12:03:17Z endpoint=/evaluate  bytes=412 sha256=9f86d0… status=200
```

Listing 3. Egress-ledger line format. Each field is plain-text; the payload hash lets an auditor confirm that a request occurred and its size without recording its content.

The ledger can be cross-checked against the network itself. The daemon contacts only the destinations on the egress allowlist (Section 10) — two vendor-owned hosts that front all communication, including authentication; an external observer — firewall logs, a packet capture, an egress-filtering proxy — must see exactly the requests the ledger records and nothing more. Verification does not require trusting the vendor.

6 Flow B — fully on-device evaluation

In the on-device flow, evaluation moves entirely onto the machine. A compact detection model (Section 7) and the inference engine (melira-engine) run locally, and no document content leaves the building: there is no analysis cloud to leak to. Between checks the product runs fully offline; the single network event is a monthly license check carrying credentials and version only, never content, and recorded in the same egress ledger (Section 5.3).

One behaviour differs by deployment. The `QUARANTINE` outcome of the decision engine (Section 4) is a cloud-deployment capability: it depends on the asynchronous review channel that holds the change at the backend and surfaces it for an owner to adjudicate later — never a real-time prompt. A fully on-device install has no such backend channel — the offline-first posture means the daemon does not phone home with case material — so a `QUARANTINE` outcome there falls through to the conservative side: the change stays rolled back and the last known-clean state restored on disk immediately, with the preserved copy available locally. What the on-device mode forgoes is the backend-mediated review, not the protection itself — fail-closed remains fail-closed, and every other layer behaves identically to the cloud-assisted flow.

6.1 Provisioning and integrity

The model and the inference engine are protected assets, never public downloads. They are obtained through an auth-gated path keyed to an active entitlement, and their integrity is pinned to the signed license manifest. Figure 5 shows the provisioning-to-inference sequence.

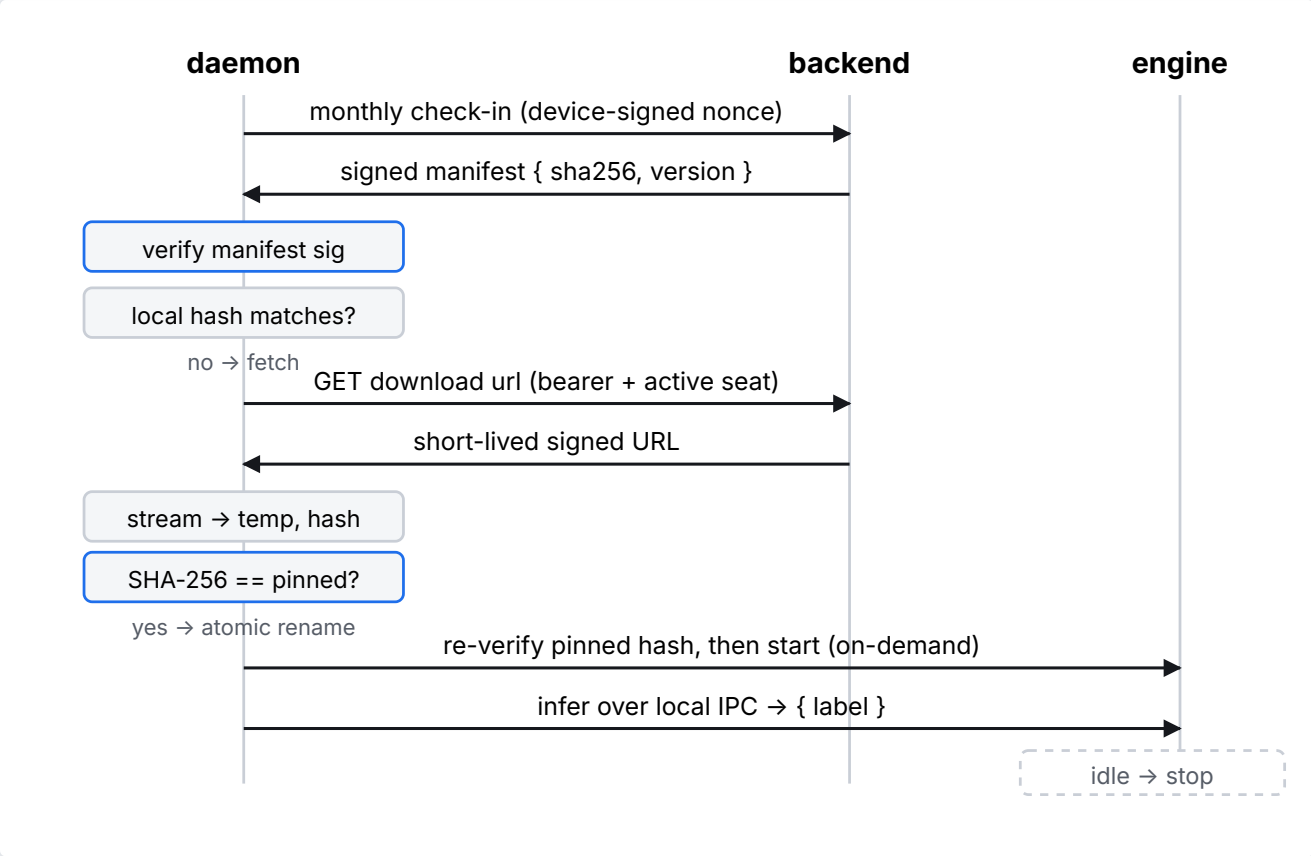

Figure 5. On-device provisioning and inference. The pinned SHA-256 originates in the signed manifest; it is checked at download and re-checked before every engine start. A hash mismatch aborts before the engine launches.

The download streams to a temporary file in the destination directory while the SHA-256 is computed in-line; a mismatch removes the partial file and fails, so a half-written or wrong artifact never persists. A matching hash is committed with an atomic rename. If the file is already present with the correct hash, no URL is requested and no download occurs.

At-rest integrity and tamper-resistance of the on-device model

On the on-device tier the model is held on disk as an **authenticated-encrypted** artifact rather than as a plain file, and the property this buys is **integrity**: the model the engine runs is cryptographically guaranteed to be exactly the model the vendor published, byte for byte. The artifact is sealed with **ChaCha20-Poly1305**, an authenticated encryption scheme (AEAD). Authentication is the operative word — Poly1305 computes a tag over the ciphertext, so **any** alteration of the stored bytes, however small, makes decryption fail rather than yield a different-but-usable model; the artifact is also bound against truncation, so a shortened file is rejected as well. This sits on top of the SHA-256 pin of Section 6.1: the plaintext model is hashed after decryption and checked against the hash pinned in the signed manifest before the engine ever starts. The two mechanisms are complementary — the AEAD tag detects tampering with the stored artifact, the plaintext hash pin proves the decrypted result is the published model — and both must pass.

The consequence is **fail-closed**, exactly as everywhere else in the system. A model that has been altered on disk — corrupted, truncated, or deliberately replaced by an attacker hoping to weaken or poison the detector — does not load: decryption fails, or the plaintext hash does not match the manifest, and the engine is **not started**. There is no "run a model we could not verify" path. The daemon's response to a failed integrity check is the same as to any other ambiguity: it declines to

evaluate against an unverified component and re-fetches the verified artifact (Section 6.1) before resuming. A tampered or poisoned model therefore cannot quietly degrade protection; it can only cause a re-verification.

The decryption key is never stored beside the file. It is delivered to the daemon during the signed monthly check-in (Section 8), bound to the device's own key (Section 8) so that it is usable only on the enrolled machine, and is held only in memory for the lifetime of the engine. The key material is never written to disk.

The decrypted model is materialised **without a file path**. Where the operating system allows it, the plaintext never appears as a named entry in any directory: on Linux it is created through an anonymous temporary file (`O_TMPFILE`) and handed to the engine by file descriptor; on macOS it is created and immediately unlinked, surviving only as an open descriptor; on Windows it is held in an **NTFS alternate data stream** rather than as a free-standing file. The engine receives this descriptor and never sees a model path on its command line. The practical effect for the threat model that matters here — a *compromised agent* running with the user's own privileges, the same adversary the rest of this document defends against — is that there is **no runtime model file for it to find, delete, or rewrite**: it cannot sabotage the detector by reaching for the model on disk, because at runtime there is no such file to reach for.

Stated honestly, and without overclaiming: the *pathless* property covers the decrypted runtime copy. The encrypted at-rest artifact does exist on disk — but it is the protected form. Deleting it triggers a re-download of the verified artifact on the next check; altering it is caught cryptographically and fails closed. None of this is a defence against the machine's owner, a privileged forensic examiner, or a memory-resident attacker, and PoisonZero does not claim it to be — it is integrity assurance for the on-device detector against a compromised agent, not a guarantee against an operator with full control of the hardware. In short: the on-device model is cryptographically integrity-assured, and a compromised agent can neither alter the detection model undetected nor sabotage it as a file at runtime — every tampering attempt resolves to fail-closed re-verification.

System requirement (Windows on-device). Because the pathless runtime delivery on Windows uses an NTFS alternate data stream, the on-device tier on Windows requires that the data directory reside on an **NTFS** volume; alternate data streams are an NTFS feature and are not available on exFAT or FAT32, and ReFS imposes a stream-size limit too small for the model. The system volume is NTFS on effectively all Windows installations, so this is the default; the installer verifies the file system of the data location up front and stops with a clear message rather than proceeding if it is not NTFS.

6.2 On-demand lifecycle

The inference engine does not run continuously. It is started lazily when a memory check arrives and is stopped again after a short idle timeout (default on the order of one minute). The model file is memory-mapped at start; during a check the analysis component occupies its footprint for a few seconds, and at rest the product is just the daemon — a few MB of RAM. The idle window absorbs bursts of edits without keeping the footprint resident. Inference is CPU-only and requires no GPU. The per-check latency (seconds on commodity CPUs) is immaterial for an event-driven daemon whose events are occasional memory edits.

6.3 The sandbox

The engine, by definition, parses attacker-controlled text — the very memory diffs under investigation. An exploitable flaw in the engine would otherwise be a direct path from "attacker writes a memory file" to "code execution on the analysis component." The mitigation is isolation: the *engine* — the inference component, and the only one that reads attacker-controlled text — runs in a minimal-privilege sandbox so that an engine exploit degrades to a process crash rather than a compromise. This is a deliberate split of responsibilities. The **daemon** is, by design, **privileged**: it runs as a root LaunchDaemon on macOS and a root systemd unit on Linux because it must watch protected memory files it does not own and restore them to their clean state in place when a change is proven poisonous — that is the job, and it cannot be done unprivileged. The **engine** is the opposite: it touches the dangerous input, so it is stripped to almost nothing. The architectural argument is precisely this asymmetry — **the most exposed component is the most confined one**, and an engine compromise cannot reach the daemon, which is unaffected and defaults to a reversible quarantine — never a silent allow — in doubt. The isolation is enforced by the operating-system kernel, not by the engine policing itself: on macOS by the Seatbelt sandbox, on Linux by the Landlock LSM, on Windows by a restricted token paired with a Job Object. All three are **deny-by-default** — every operation is forbidden unless the profile explicitly grants it — and the engine is contained as the last, least-trusted component in the chain.

macOS — kernel Seatbelt

On macOS the engine is launched under the kernel sandbox via `sandbox-exec`, applying a deny-by-default profile (Listing 4). The profile travels inside the daemon binary (embedded at build time) and is materialised next to the model before every engine start; the on-disk copy is an artifact for administrator audit only, and any deviation from the daemon's own copy is overwritten at start, so the file that runs is always the one the daemon ships. The profile permits exactly: reading the pinned model and the engine binary, executing that one engine binary, reading the system libraries the loader needs (`dyld`), reading `/dev/urandom` for the PRNG seed, and creating and binding a local **Unix-domain socket** for the loopback IPC channel (the engine's HTTP library speaks AF_UNIX here, not IP loopback). The only `file-write*` rule is on that single per-spawn socket path — so the engine can write no other file and persist no content — and there is no `network-outbound` rule, so it cannot open an outbound connection. When the service runs under a root LaunchDaemon, the engine additionally drops to the unprivileged system user `nobody` before it parses any input. This path is verified live on Apple Silicon; the sandbox adds no measurable latency to evaluation (a warm inference completes in roughly 1.1 s).

```
;; macOS Seatbelt profile for the inference engine (deny-by-default).
;; Applied via: sandbox-exec -f <profile> -D MODEL_PATH=… -D ENGINE_PATH=… <engine>
(version 1)
(deny default)

;; sandbox-exec applies the profile, then exec's exactly this one binary.
(allow process-exec (literal (param "ENGINE_PATH")))

;; Filesystem: read-only on model, engine binary, system libs, PRNG.
(allow file-read*
  (subpath "/usr/lib")
  (subpath "/System/Library")
  (literal "/dev/urandom")
  (literal (param "MODEL_PATH"))
  (literal (param "ENGINE_PATH")))
;; IPC: create and bind ONE Unix-domain socket at the per-spawn socket path.
;; The only write rule is this single path; no outbound rule → offline.
(allow file-write*  (literal (param "SOCKET_PATH")))
(allow network-bind (local unix-socket))
```

Listing 4. The shipped macOS sandbox profile (kernel Seatbelt, abridged). The engine may read only the pinned model, the engine binary and the loader's system libraries, may execute only itself, and may create and bind only a single per-spawn Unix-domain socket for the loopback IPC channel. Its one write rule is that socket path and there is no outbound rule. The full profile ships embedded in the daemon and is auditable next to the model.

Linux — Landlock LSM

On Linux the cage is the kernel's Landlock LSM, applied in a dedicated re-exec helper that locks itself down and then `execve`'s the engine, so the restrictions are inherited by — and inseparable from — the engine process. Again deny-by-default: the helper enables `no_new_privs`, then constructs a ruleset whose handled access set covers everything the kernel ABI knows, and grants back only what the engine needs — read+execute on the engine binary, read on the model file (plus `/dev/urandom` read and `/dev/null` read/write for I/O redirection), and, on the single root-owned socket directory, the right to create, write and remove one **Unix-domain socket** file for the loopback IPC channel. The IPC transport is that Unix-domain socket, not an IP port, so binding it is a pure filesystem operation and no network access type is granted to the engine at all. Egress is then denied two independent ways. On kernels new enough for Landlock's network rules (ABI v4, kernel 6.7) the ruleset *handles* TCP bind and connect with no matching allow rule, so the kernel refuses any TCP socket the engine tries to open. Independently, and on *every* supported kernel, a `seccomp`-BPF filter rejects the creation of any `AF_INET`/`AF_INET6` socket — TCP and UDP alike — with `EACCES`, while leaving `AF_UNIX` (the engine's own IPC) permitted; so even on a kernel below ABI v4 the engine has no outbound path. This is exercised on a real kernel in CI. The Linux engine is built **fully statically against musl** (both architectures, amd64 and arm64), with no dynamic loader and no shared libc — which is precisely what makes the minimal ruleset above tenable: a dynamically linked engine would need read and execute access to `/lib64/ld-linux` and the system libc, widening the cage; the static binary needs read+execute on exactly one file, itself. Landlock's filesystem sandbox is available from kernel 5.13 (ABI v1); its network access type, which lets the kernel deny TCP egress directly, arrives with kernel 6.7 (ABI v4). On a kernel new enough for the filesystem layer but not the network layer, the filesystem cage is enforced and the `seccomp` egress filter above still closes off every IP socket, so egress remains blocked by construction even where Landlock cannot yet express it.

Fail-closed, on every platform

The sandbox is mandatory, not best-effort. If the kernel isolation cannot be established — Seatbelt unavailable, a kernel below 5.13 or Landlock disabled in the LSM stack, or on Windows the restricted token / Job Object failing to apply or the per-spawn WFP egress-block filter failing its write-probe — the engine is **not started** and the pipeline fails closed to a reversible quarantine. There is no degraded "run the engine without its cage" mode; the most attacker-facing component never executes unconfined. This is fail-closed applied to the sandbox itself: the limit is the feature.

System requirements (Linux on-device)

The Linux on-device tier therefore requires a kernel ≥ 5.13 with Landlock **active in the LSM stack**. That is the out-of-the-box default on Ubuntu ≥ 22.04, Debian ≥ 12, and RHEL ≥ 9.6; on other or older distributions Landlock is enabled with the boot parameter `lsm=landlock,…`. Inside containers, the default Docker seccomp profile blocks the `landlock_*` syscalls, so an adjusted seccomp profile that permits them is required. Where Landlock cannot be enabled at all, an opt-in remote backstop (cloud-assisted evaluation, Section 5) is the documented alternative — the protection stays, only the data flow changes, and it is stated transparently rather than hidden.

Windows — restricted token + Job Object

On Windows the engine runs under a **restricted primary token** combined with a kernel **Job Object**, and the posture is strictly fail-closed exactly as on macOS and Linux. The token is created with `DISABLE_MAX_PRIVILEGE`, which strips every privilege from the engine process except `SeChangeNotify`, and the process runs at **low integrity** so it cannot write objects at medium or higher integrity. The Job Object caps the process: `ActiveProcessLimit = 1` means the engine cannot spawn a child process — any `CreateProcess` is refused by the kernel — and a memory cap bounds its footprint. Egress is not left to construction alone: at every engine spawn the daemon installs a dynamic **Windows Filtering Platform (WFP)** block rule scoped to the engine — a hard, unoverridable veto that blocks the engine's own outbound TCP *and* UDP connections — including its own direct DNS queries — while leaving the loopback IPC path intact, so the engine binds loopback only (`127.0.0.1`) and cannot open a **direct** outbound connection of its own even if it tried. This blocks the engine's own sockets; it does not cover name resolution that a SYSTEM service performs on the engine's behalf, one narrow residual channel documented in Section 11. If that filter cannot be written the engine is not started (fail-closed, as above). This kernel-enforced egress block is what promotes the local Windows engine from experimental to production (shipped in v1.7.0). This path is verified live on GitHub Actions `windows-latest`: child-process spawning is blocked by the kernel, and the real Windows engine loads and evaluates correctly under the token, Job, and WFP constraints. A still-stricter stage — running the engine inside an AppContainer — remains a separate honest fast-follow; the current profile is live, fail-closed, and verified.

6.4 Untrusted output handling

Even inside the sandbox, the daemon treats the engine's answer as untrusted input. The response is read under a fixed cap (64 KiB) so that a misbehaving engine cannot force an unbounded allocation; the expected verdict is a few bytes. The verdict is parsed strictly — only the expected fields are accepted, and unknown fields are rejected. If the engine crashes, hangs, or returns anything unparseable, the decision is fail-closed (a reversible quarantine at the uncertain band), never a silent allow. Combined with the pre-start hash pin (Section 6.1), the engine is contained on its input, its output, and its on-disk image.

6.5 The daemon–engine channel

The channel between the daemon and the on-device detection engine is bound to a fresh **256-bit secret** generated by the daemon at each engine start and handed to the engine as it spawns; the secret lives only for the lifetime of that engine process and is never written to disk. The daemon **authenticates the engine before use** via a challenge — only a process that holds that exact secret passes — and signs every request to it with the same key. The channel is loopback-only — a Unix-domain socket in a root-owned directory on macOS and Linux, and a `127.0.0.1` TCP loopback socket on Windows (whose HTTP library lacks AF_UNIX support) — so it is never reachable off-device. A process that does **not** hold the per-spawn secret cannot impersonate the engine to the daemon: a port-squatter that grabs the port without the secret fails the challenge and is rejected, and the daemon then talks to no engine at all rather than to the impostor. A failed authentication is treated **fail-closed** — the engine is not trusted, and the change is held in a reversible quarantine rather than evaluated against an unauthenticated responder. This is deliberately a per-spawn shared secret with challenge authentication, not a cryptographic identity for the engine: the engine is a local generative server, and the property the channel guarantees is that the daemon talks only to the engine it just started, on loopback, for as long as that one secret lives — not an absolute claim that the secret can never be observed by a co-resident, equally privileged process on the same machine.

6.6 Bring-your-own model (dual)

An Enterprise deployment may attach the customer's **own** inference endpoint as a second evaluator alongside the on-device model. This is always a **dual** arrangement — the local model of Section 6 stays present and primary, and the customer endpoint runs under a `local-primary, on-prem-backstop` policy: a change is cleared consistently with the local baseline, and if the customer endpoint is unreachable or fails its security check the decision falls back to the local model rather than to an unverified responder (fail-closed, exactly as in Section 6.4). The endpoint speaks the same inference contract and returns the same verdict shape as the on-device engine, so no model-specific glue is required. Whether a second opinion is attached is decoupled from the deployment mode: a Cloud-mode device may point its second evaluator at an on-prem endpoint just as a Private deployment can, and a fully cloud second opinion is configured account-wide; in every case the local model of Section 6 stays primary and the arrangement is fail-closed onto it.

A configurable remote endpoint raises the same impersonation question that Section 6.5 answers for the loopback channel: the daemon must be certain it is talking to the customer's real server and not to a host that won the network race. Because the endpoint is reached over TLS rather than loopback, the daemon does not use a per-spawn shared secret but **pins the endpoint's TLS server certificate** — a SHA-256 pin the customer registers. A server that cannot present the pinned certificate is rejected before any evaluation request is sent, the remote equivalent of the loopback challenge; optional **mutual TLS** adds a second, bidirectional layer. Authentication secrets — bearer tokens and client keys — are referenced by name in the signed deployment claim (Section 8); the secret material itself resides only on the daemon host, in owner-readable files, and never in PoisonZero's cloud.

The endpoint URL, the certificate pin and the auth references are part of the customer's deployment profile, validated and then carried — signed — inside the daemon's seat token (Section 8); a private deployment therefore receives this configuration once, offline-safe, with no live control channel into it. Because the customer hosts the model and the daemon runs in the customer's own environment, the model, its weights and the evaluated content never leave that environment — the strongest

expression of the private posture. Direct integration of managed frontier-provider keys is a separate, planned capability that requires per-provider adapters and is not part of this flow.

6.7 Local event timeline

The daemon keeps a local, append-only **event timeline**, inspectable with `poisonzero timeline`. Its purpose is operator visibility: remote-driven changes to a daemon — applying a new configuration, pausing protection — are recorded locally so the machine's owner can see what happened on the device without consulting the cloud. The timeline stores only canonical per-domain fingerprints — identifiers, hashes, and boolean markers — and **never any file content**, written to a `0600` file behind a persistent rate limit and read back through a rotation-safe reader. This local record ships today; it is a log for the operator, not a control channel — nothing on it leaves the device.

7 The on-device detection model

The on-device evaluator (Section 6) is a small detection model whose design is the technical centre of the system. This section states its architecture class and the rationale behind it, how its weights were produced, the corpus it was trained on, and the evaluation methodology and results. Results are stated relative to baselines and reflect broad internal testing on data the model never trained on.

7.1 Architecture class and rationale

The detection model is a compact **decoder-only generative transformer**. The choice is deliberate and is supported by measurement: it is specifically *not* an encoder-based sequence classifier of the BERT-family / "prompt-guard" style.

The reason is the core finding of Section 1.1, made empirical. An encoder-based injection classifier is trained to recognise injection *phrasing*. On the memory-poisoning attack classes it recognises direct-injection phrasing but is structurally blind to instructions that read like legitimate memory entries. As Table 5 summarises qualitatively, a leading off-the-shelf encoder guard is **practically blind to data-exfiltration and subtle/indirect attacks** — it flags classic overrides but largely misses the classes that do not announce themselves as injections. Memory poisoning is not prompt injection, and a phrasing classifier does not transfer to it.

A generative model is used instead because it can be asked to reason about **intent and data flow** — "does this change cause the agent to exfiltrate, to escalate trust in an untrusted source, to disable a safety control, or to obey a hidden instruction?" — rather than to match surface phrasing. That reframing is what recovers the exfiltration and subtle/indirect classes (Table 5).

Attack class	Encoder injection guard	Generative approach
Direct injection	detected	detected reliably
Data exfiltration	largely missed	detected
Meta-attack	largely missed	detected reliably
Role-play / jailbreak	partially detected	detected
Subtle / indirect	largely missed	detected

Table 5. Detection by attack class, encoder injection guard versus the generative approach. The encoder guard is practically blind to exfiltration and subtle/indirect cases; the generative approach recovers exactly those classes.

7.2 Customised weights and inference contract

The model does not ship an off-the-shelf checkpoint. Its weights are **fully customised** through supervised fine-tuning. Labels are produced by **knowledge distillation** from a frontier-scale teacher model: the teacher labels a large corpus of redacted memory diffs (intent and danger), and the compact model is trained to reproduce those judgements. Fine-tuning is parameter-efficient (a low-rank adapter, LoRA), and the trained adapter is then **merged into the base weights**, so the deployed artifact is a single set of customised weights rather than a base model plus a runtime adapter.

Inference is **zero-shot** against a fixed classification rubric supplied as the system prompt — the same rubric the model was trained on, with no few-shot exemplars at inference time. The model emits a **constrained output**, a small JSON object `{ "label", "danger" }`, enforced by a grammar so that the format cannot drift; the model adheres to a strict output schema and the verdict cannot deviate from it. The deployed weights are quantised for CPU inference, which is what keeps the footprint small while preserving the operating point: the largest tier ships at a higher quantisation precision than the two smaller ones because at that size it yields a lower false-positive rate at essentially the same recall. The on-device model ships in three named tiers — **Melira Nano** (the smallest, prose/memory-only variant), **Melira Pro** (the standard variant), and **Melira Enterprise** (the largest, which additionally carries the configuration/hook modality). All three run entirely on-device, on the CPU, with no GPU and no accelerator — the smallest fits on minimal hardware, and even the largest loads and evaluates on a modest workstation.

Within a tier the model is not a single monolith. A lightweight **path-based router** inspects the file under evaluation and selects the appropriate expert: a **prose expert** for memory and instruction files, and a separate **configuration expert** for settings and hook files, whose statistical distribution differs enough from prose (Section 1.1) to warrant its own specialised weights. A single-resident **coordinator** keeps at most one expert loaded at a time, evicting the other, so specialisation does not multiply the resident footprint — the machine pays for one expert, not the whole set. The configuration/hook modality is a capability of the larger tiers; Melira Nano stays focused on the prose surface (Section 11).

One honest engineering detail governs how the output is consumed. The generative `danger` score is **not calibrated**: the model emits a small number of near-constant danger values and the score does not move usefully under thresholding. Therefore the discrete **label** is the product signal, not the danger number. The system maps the label to a fixed danger value before it reaches the decision engine (Section 4) — a malicious label to a high danger, a benign label to a low one. The unreliable raw score is never forwarded.

7.3 Training corpus

The corpus is purpose-built: more than 5,000 examples spanning the five attack classes (Section 1.2) together with adversarial benign cases. The benign set is deliberately hard. It includes **trigger-word benigns** — harmless texts that contain alarming tokens such as "ignore", "disable", "password", or "sudo" in legitimate contexts. These are the cases that drive the *over-defensiveness* failure mode (the NotInject / over-defence problem in the literature), where a detector flags benign content because it contains attack-adjacent vocabulary. Without such cases in training, a generative judge over-flags exactly this content.

The corpus is **multilingually augmented**, across Latin and non-Latin scripts, because the language of an attack is attacker-chosen; a monolingual detector is evadable by translation. Splits are **stratified**

by class, language, and label, with explicit **leakage control**: examples are deduplicated under normalisation against the evaluation sets so that no held-out item, and no near-duplicate of one, appears in training. The hand-curated adversarial set (Section 7.4) is excluded from training entirely.

7.4 Evaluation and adversarial testing

Two evaluation sets are used. The first is an **in-distribution held-out test set**, drawn from the same generation process as training but disjoint from it after deduplication. The second is a separate, **hand-curated adversarial set**, out-of-distribution by construction, written to probe the system rather than to resemble the training distribution. Metrics are reported per attack class (recall) and as the false-positive rate (FPR) on the benign cases, including the trigger-word benigns.

The adversarial set exercises the specific red-team vectors the system is designed to resist:

- **Multilingual attacks**, including non-Latin scripts, to test that detection is not anchored to English phrasing.
- **Unicode and homoglyph evasion** — handled before the model by NFKC normalisation in the rule layer (Section 3.3), so look-alike substitutions are folded to base forms.
- **Role-play and jailbreak framing**, where the malicious instruction is embedded in a persona or scenario.
- **Meta-attacks on the protection itself** — entries that attempt to disable the guard or grant blanket trust to an untrusted source.
- **Timed burst writes**, where the malicious state is the final write in a burst — defended by the trailing-edge guarantee (Section 3.1), not by the model.

7.5 Results

The results are stated relatively, consistent with the public positioning. The model detects **more than 3× as many attacks as leading off-the-shelf guard models**, and it flags the classes those detectors are practically blind to — subtle/indirect injection and data exfiltration (Table 5). Fine-tuning on distilled labels cut false alarms by **roughly 95% relative to the untuned base model**: the untuned generative base, lacking the trigger-word benigns, over-flagged legitimate content, and fine-tuning is what reduced that over-defence while raising detection. In broad internal testing, the model detects **over 94% of attacks at under 5% false alarms**, with balanced behaviour across languages and no single-language outlier.

The methodology is reported honestly. Evaluation is on data the model never trained on, balanced across the five attack classes and across languages, with a deliberately adversarial benign set so the false-alarm rate reflects realistic content rather than easy negatives. Relative gains are reported rather than a single headline accuracy, because a detector is only as good as its weakest attack class and its behaviour on hard benign cases. The weakest residual class is meta-attack within the generative layer, which is also covered deterministically by the rule layer (Section 3.3); the cloud labelling pipeline that produced the corpus is repeatable, so drift and newly observed attack patterns can be addressed by re-training without exposing customer content to the process.

That labelling pipeline can also be fed, strictly by opt-in, from the field. An Enterprise Cloud deployment may — only when it is a Cloud deployment, on the Enterprise tier, and with explicit opt-in — contribute redacted examples into a training-data spool that enforces redaction at capture. Nothing is uploaded otherwise, and every contributed example sits behind an administrator review gate (`poisonzero training review / approve / discard`), isolated per owner, before it can ever reach

training. **Private mode collects nothing by construction** — the collection path is a pure gate that is closed in Private mode, verified as such — and an opt-in can be revoked at runtime. This is the improvement flywheel: real, redacted, consented signal narrowing the residual classes over time, with no customer content exposed and no collection a customer did not switch on.

8 License and integrity architecture

Three cryptographic mechanisms anchor the system: an offline-verifiable license token, a device identity bound to hardware where available, and signed release and model artifacts. Table 6 summarises the keys.

Purpose	Algorithm	Private-key location
License / seat token + model manifest signing	Ed25519 (JWS / detached)	backend secret store; public key compiled into the daemon binary
Device identity (check-in nonce)	ECDSA P-256 (IEEE-P1363 r‖s)	TPM (hardware tier) or a 0600 software-key file (default tier)
Release artifact signing	Cosign keyless (Sigstore, OIDC)	ephemeral; certificate logged to the public transparency log

Table 6. Cryptographic keys by purpose, algorithm, and storage location.

License token. The seat token is a compact Ed25519 JWS. Its public key is compiled into the daemon binary, so the token is verified entirely offline: the daemon checks the signature, the expiry, and a grace period before declaring the license valid, in grace, or expired. There is a **monthly license check** with a **grace period** thereafter, during which protection continues; only after the grace period elapses without a successful check does the daemon treat the license as expired. The token also carries the feature flags and the minimum daemon version. Listing 5 shows the claim shape (synthetic values).

```
# JWS header
{ "alg": "EdDSA", "typ": "JWT" }

# claims (signed payload)
{
  "seatId":   "seat_3f9c…",
  "appId":    "app_a17b…",
  "tier":     "tpm",
  "plan":     "enterprise",
  "iat":      1717660800,
  "exp":      1720252800,
  "minVersion":   "1.7.0",
  "graceMaxDays": 0,
  "features": { "localModel": true }
}
```

Listing 5. Seat-token claim structure (synthetic; `graceMaxDays` shown as a placeholder). The detached signature over `base64url(header).base64url(payload)` is the third token segment.

Device identity. A device-bound key pair signs the check-in nonce, proving device identity to the backend in addition to the bearer token — a stolen bearer token alone cannot forge a check-in. The signature is ECDSA over P-256 in fixed-width `r‖s` form. The key is anchored at the strongest available

tier: a TPM where present (hardware-bound, private key non-exportable), otherwise a software key in a 0600 file. The provisioning step selects the best available tier and reports it to the backend.

Model manifest. The same Ed25519 license key signs the model manifest delivered at check-in. The manifest pins the model's SHA-256; the daemon verifies the manifest signature and then pins the model file against that hash before every engine start (Section 6.1). The manifest is the authoritative pinning source — not any hash echoed by the download endpoint — so the delivered file and the pinned hash are guaranteed to correspond. On the on-device tier the model is additionally held as an authenticated-encrypted (ChaCha20-Poly1305) artifact, and the decryption key is delivered with the manifest at check-in, bound to the device key so it is usable only on the enrolled machine and held only in memory; the AEAD tag and the manifest's plaintext hash together guarantee that any tampering with the stored model fails closed rather than loading an altered detector (Section 6.1).

Release artifacts. Every release artifact is signed with Cosign keyless (Sigstore) and published with a `SHA256SUMS` manifest, itself signed (a detached signature and certificate via the Sigstore transparency log). Each binary is therefore verifiable against a signed checksum list before it runs; the one-line installer verifies the checksum automatically, and air-gapped operators can verify by hand. On the on-device tier the chain extends to the model: integrity is checked not once at download but at every engine start.

9 The management API

A fleet is not provisioned by hand. The management API exposes the same app lifecycle the browser panel offers as a small REST surface, so that creating, enrolling, deactivating, and deleting protected apps can be driven from CI/CD pipelines, configuration management (Ansible, an SSH loop), or an MDM/endpoint-management system (SCCM, Intune). It is the programmatic counterpart to the bulk-deployment model of Section 8: an operator bakes an *un-enrolled* daemon into a golden image, and each machine enrolls itself on first start using a code the API minted in advance. A machine-readable OpenAPI 3 specification is published alongside the human documentation.

9.1 Management plane versus data plane

The decisive property of this API is what it does **not** see. The system has two distinct planes. The **data plane** is the daemon's own traffic: check-ins and, in the cloud-assisted flow, evaluation requests carrying redacted diffs (Section 5). The **management plane** is this API, and it operates exclusively on lifecycle metadata — an app's identifier, its display name, its status (`pending` / `active` / `revoked`), its reported platform and agent version, timestamps, and short-lived enrollment codes. No memory-file content, no diff, and no evaluation payload ever crosses the management plane. Provisioning a thousand machines through the API exposes no protected content to it, because that content travels the data plane and, in the on-device flow, never leaves the machine at all.

9.2 Authentication and tenant isolation

Callers authenticate with a bearer key of the form `pz_live_` followed by forty characters, created in the panel. The key is shown **once** at creation; at rest the backend stores only its SHA-256 hash, used directly as the lookup key, so the plaintext is never persisted and a database disclosure does not yield usable credentials. Each key is bound to its owner, and every request is scoped to that owner. Apps belonging to another tenant — and apps that do not exist — return an identical `404`, so the API cannot

be used to probe for or enumerate identifiers outside one's own fleet. A fixed number of active keys is allowed per account (ten), and any key can be revoked independently.

9.3 Transport and endpoints

The API is reached over HTTPS at `poisonzero.com/api/v1/…`, where a reverse proxy forwards only the method, path, authorization header, content type, and a size-capped body to the backend function; the internal function URL is never exposed. The proxy hardens the path before forwarding: the request is decoded to a fixed point so that multiply-encoded sequences cannot slip through, and any dot-segment or backslash is rejected outright, closing path-traversal attempts against neighbouring backend functions. The endpoints mirror the panel: create an app (returning its identifier together with a first enrollment code), list the fleet, fetch one app, mint a further enrollment code, deactivate (revoke), and delete. Errors are returned as structured JSON with a stable machine-readable code; unexpected internal faults are logged server-side and answered with a generic message, never internal detail. Fleet management is deliberately not gated on an active subscription — the daemons themselves are gated at runtime — so an operator can provision and revoke even while a plan is being arranged.

9.4 Operating modes — Cloud and Private

PoisonZero ships in two operating modes, selectable per device at install time; the local fail-closed protection is identical in both — the mode only changes how the device is managed and what, if anything, leaves it. Table 7 states the difference dimension by dimension.

Dimension	Cloud mode	Private mode
Local protection	Runs fully local and fail-closed; works offline; verdicts reached on the device.	Identical — the same local, fail-closed guard runs on the device.
Network contact	Managed; regular, documented check-ins with the console.	At most one outbound request every 30 days; fully offline between contacts.
Remote management (update / model / disable)	Centrally managed from the console; changes apply on the next check-in.	Not remotely managed; any change takes effect only at the device's monthly contact.
Remote revoke / kill-switch	Yes, from the console; transient network errors never trigger a false stand-by — only an authenticated revoke does.	No remote revoke; the device is under the customer's physical control.
Telemetry / analytics	Inherent to Cloud mode — redacted metadata reported continuously (no toggle), data-minimised to redacted additions and counts; never file content.	Nothing leaves the device unless enabled — a strict opt-in — and then only at the monthly contact.
Agent memory / instruction content	Never leaves the device; only redacted metadata is reported.	Never leaves the device; content egress is structurally excluded.
Egress surface	Two vendor-owned domains, port 443, outbound only, no inbound listener.	Same allowlist; the monthly contact reuses the existing endpoint — no new domain or URL — and bundles the update check and an opt-in redacted analytics batch into a single request.

Table 7. Cloud mode versus Private mode. The local, fail-closed protection is identical in both; the mode changes only management and egress.

In Private mode there is at most one outbound request every 30 days, over the same existing egress domain — no new endpoint and no inbound port.

9.5 Observability export (Enterprise, mode-aware)

An Enterprise fleet can forward its decision stream to the customer's own monitoring by configuring an **owner-scoped OpenTelemetry export** to a collector or SIEM the customer runs. Authentication headers for the collector are held in the backend secret manager, never in the daemon image; the syslog severity of each exported event is derived from its threat level, and a redacted content excerpt is attached **only for a threat=high (proven-poisoning) event**, never for the lower bands. The export is mode-aware: **Private mode ignores any remote collector configuration** and accepts only a loopback or `.local` collector with enforced hashes, so a private deployment can still emit telemetry to a strictly on-device sink without ever opening a remote egress path. The configuration is managed through a dedicated API (`/v1/otel-config`, GET / PUT / DELETE) and documented on a separate `/observability` page.

10 Network footprint and firewall allowlist

Two vendor-owned domains. One port. Outbound only. Everything a deployment talks to is enumerable in advance, and it reduces to two vendor-owned hosts under `poisonzero.com`. The daemon makes only **outbound** connections, every one of them TLS on TCP 443; it opens **no inbound listener** and accepts no incoming connections, so no firewall ingress rule is required and no port needs to be opened toward the machine. Table 8 is the complete egress allowlist — the exact set of destinations a corporate firewall or egress-filtering proxy must permit, and nothing beyond it appears on the wire.

Destination	Port	Direction	Purpose
`update.poisonzero.com`	443	Outbound (TLS)	Control plane — enrollment, authentication sign-in and token refresh, policy retrieval, evaluation, audit reporting, and license check-in
`cdn.poisonzero.com`	443	Outbound (TLS)	Auth-gated, SHA-256-pinned download of the inference engine and detection model (Enterprise on-device only; a one-time provisioning fetch, re-fetched only when an artifact changes)

Table 8. Complete egress allowlist — two vendor-owned hosts; every connection is outbound TLS on TCP 443, and there is no inbound listener.

Every binary carries the same compiled-in backend base URL, `https://update.poisonzero.com`; all control-plane traffic — authentication sign-in and token refresh, evaluation, policy and audit, and the license check-in — terminates there. The auth-gated streaming of the engine and model artifacts is served from the second vendor-owned host, `cdn.poisonzero.com`, whose URL the backend hands back at check-in; that download stays bearer-authenticated and SHA-256-pinned. Because both domains are vendor-owned and under our control, the surrounding infrastructure can evolve without ever changing what a firewall must permit: a rule written once stays valid for the long term, rather than being tied to volatile internal endpoint names. There is no fan-out to third-party hostnames to enumerate, audit, or keep in sync.

Allow-listing by IP address

Pinning individual, fixed IP addresses is not recommended. Both egress hosts run on provider anycast — Firebase can change its address, and Cloudflare rotates through its range — so any single IP observed today is not stable, and a static per-IP allow-list will silently break the moment the provider re-points it. **Domain-based (FQDN) allow-listing is therefore the recommended and most stable approach.** The two hostnames above are the long-term stable contract; the IP addresses behind them belong to managed cloud platforms and can rotate at any time. For environments that can only firewall by IP, the honest picture is that each domain resolves into a cloud provider's published, multi-tenant ranges rather than a single fixed address — so the correct thing to pin is the provider's documented ranges, refreshed periodically, never the individual IPs observed on any given day.

The two hosts sit behind two different managed platforms:

- `update.poisonzero.com` **(control plane)** is served by Google Firebase Hosting. Firebase routes custom domains to a single documented anycast A record, `199.36.158.100` (Google-owned; in ARIN block `199.36.152.0/21`, Org *Google LLC*). That address is a dedicated Firebase Hosting anycast IP and is *not* part of Google's general published `goog.json`/`cloud.json` egress ranges, so the value to allow-list is the single Firebase Hosting A record the Firebase console's custom-domain wizard shows — not a broad GCP block. No IPv6 address is published for this host.
- `cdn.poisonzero.com` **(engine/model download, Enterprise on-device only)** is fronted by Cloudflare. It resolves into Cloudflare's anycast ranges — observed IPv4 within `188.114.96.0/20` (the last octet rotates between resolutions) and IPv6 within `2a06:98c0::/29`. These are not fixed per-host addresses; the authoritative, machine-readable lists are Cloudflare's published ranges at `cloudflare.com/ips-v4` and `cloudflare.com/ips-v6`, which are the correct thing to allow-list.

Destination	Provider	What to allow-list	Authoritative source
`update.poisonzero.com`	Google Firebase Hosting	A record `199.36.158.100` (Google-owned, in block `199.36.152.0/21`); no published IPv6	Firebase console custom-domain wizard · `firebase.google.com/docs/hosting/custom-domain`
`cdn.poisonzero.com`	Cloudflare	IPv4 `188.114.96.0/20` and IPv6 `2a06:98c0::/29` — best taken from Cloudflare's full published range list	`cloudflare.com/ips-v4` · `cloudflare.com/ips-v6`

Table 8a. IP-level allow-list guidance. Domain-based allow-listing is recommended; where IP filtering is unavoidable, pin the provider's published ranges (not the individual IPs observed at a point in time) and refresh them on the provider's schedule.

Because both fronting platforms use shared, evolving anycast ranges, a static IP allow-list will eventually drift; the two vendor-owned hostnames remain the durable contract. Where a firewall supports it, allow-list by domain (FQDN) and treat the ranges above only as a fallback for IP-only environments.

In the fully on-device flow (Section 6) the network footprint shrinks to a single periodic event. The detection model and the inference engine (`melira-engine`) bind exclusively to a local loopback-only channel — a Unix-domain socket on macOS and Linux, a `127.0.0.1` TCP socket on Windows — and produce **no direct network egress** of their own — on Windows this loopback-only property is not left to construction but kernel-enforced by a per-spawn Windows Filtering Platform egress-block filter that blocks the engine's own outbound sockets (Section 6.3; one narrow residual — name resolution performed on the engine's behalf by the Windows DNS client service — is documented in Section 11); memory-file content never leaves the machine, because there is no analysis cloud to reach. The only recurring outbound connection is the monthly license check-in to the backend — one documented HTTPS request per month carrying credentials and version only, never content. There is no "no call home" claim to make here: that one request exists, it is deliberate, and it is the entire periodic footprint. Analytics sharing is decided by deployment *mode*, not tier: it is off by default in Private mode — a strict opt-in — and on by default in Cloud mode. With analytics active, that same request additionally carries aggregate counts — numbers only, no paths and no content; it remains one request per month.

Each outbound request — the monthly check-in included — is written to the user-readable egress ledger (Section 5.3), inspectable with `poisonzero egress`. Table 8, the ledger, and the firewall's own logs are three independent records of the same set of connections; an auditor can cross-check them without trusting the vendor, and any line on the wire that is not in the allowlist is, by construction, not the daemon.

Console access (admin browser)

The egress allowlist above governs the daemon on protected machines; the control panel is a separate, browser-to-panel flow. Table 8 covers what a *protected machine* reaches outbound — the daemon's connections only. Administrators reach the control panel from their own browser, over HTTPS on TCP 443, to a different set of vendor-owned hosts. This is human-driven web traffic from admin workstations, not the daemon egress, and it does not originate from the protected machines. The two flows are deliberately kept distinct so that the daemon's firewall contract stays minimal even where the panel is reachable from a different network segment.

Console	Domain	Port	Direction	Notes
Standard (cloud-managed) tenants	`console.poisonzero.com`	443	Admin browser → panel (TLS)	The shared, multi-tenant control panel where standard customers sign in.
Enterprise origin-isolated tenants	`<slug>.poison0.com`	443	Admin browser → panel (TLS)	Each Enterprise customer is provisioned a dedicated, origin-isolated console on its own subdomain under the separate `poison0.com` domain — kept off the marketing/admin origin by design. `<slug>` is the per-tenant slug chosen at provisioning.

Table 8b. Console (panel) access from the administrator's browser — HTTPS on TCP 443, distinct from the daemon egress of Table 8. These are not connections the protected machines make.

The split of origins is itself a security property. The Enterprise per-tenant console lives on `poison0.com` — a **separate registrable domain** from the `poisonzero.com` marketing and control-plane origin — so a tenant's panel runs on its own origin, isolated from the vendor's primary domain and

from other tenants. An administrator whose network filters outbound web traffic by destination should permit the relevant console host (`console.poisonzero.com` for standard tenants, or the tenant's own `<slug>.poison0.com` for an Enterprise isolated instance) *for the admin workstations that operate the panel*. This is independent of the daemon allowlist in Table 8: a protected machine never needs to reach the console, and an admin browser never needs to reach the daemon's egress hosts.

11 Limitations and residual risk

No detector stops every attack in every environment, and the system does not claim a 100% catch rate. The boundaries below are stated explicitly.

- **Novel patterns and model drift.** The detection model generalises but is bounded by its training distribution. Genuinely novel attack patterns may evade it until the corpus is extended; the repeatable labelling pipeline (Section 7.5) is the mechanism for closing such gaps, but it is reactive by nature. Within the generative layer, meta-attack is the weakest class; it is covered deterministically by the rule layer (Section 3.3).
- **Offline redaction is pattern-based.** In the cloud-assisted flow, free-text names and unstructured PII are not reliably detectable offline without ML and may pass into a redacted diff (Section 5.1). Teams with strict requirements keep memory files PII-lean or use the fully on-device flow, where no content leaves at all.
- **TOCTOU bounds.** The trailing-edge guarantee (Section 3.1) ensures the final on-disk state of a burst is the state that is scored, closing the timed-burst bypass. It does not eliminate every time-of-check-to-time-of-use gap in principle: a write that lands after a trailing evaluation completes is a new change and triggers a new evaluation, but the model evaluates content, not the absolute instant of a race. The design narrows the window to "the state that persists is the state that is scored" rather than claiming an atomic guarantee against the filesystem.
- **Engine failure is fail-closed, which has a cost.** If the inference engine cannot run — crash, timeout, integrity-pin failure, or unparseable output — the decision is a reversible quarantine at the uncertain band (Sections 4, 6.4): the change is left in place and surfaced with a passive notice rather than confirmed clean. This is the intended safe default — an attacker who forces an engine outage can therefore never trigger destruction of legitimate content — but it means that during an outage a genuinely poisoned change is only flagged for review rather than neutralised on disk, so protection degrades to detection-and-notify until the engine is restored.
- **Sandbox hardening is live on all three platforms, with a known next stage on Windows.** Kernel-enforced, fail-closed isolation is live on macOS (Seatbelt), Linux (Landlock) and Windows (restricted token + Job Object) — no cage, no engine, on every platform (Section 6.3). On Windows, kernel-enforced egress blocking via the Windows Filtering Platform (Section 6.3) is now live — a per-spawn, unoverridable filter on the engine — and a still-stricter AppContainer stage remains a separate honest fast-follow. One residual is stated plainly: the WFP filter blocks the engine's own outbound sockets, but the Windows DNS client service (`dnscache`, running as SYSTEM) still resolves names on the engine's behalf, outside that per-spawn filter. A subverted engine could in principle encode a small amount of data into a hostname it asks to resolve (for example `<encoded-secret>.attacker.example`): the `connect()` is blocked, but the DNS query carrying those bytes still leaves via `dnscache`. The bandwidth is low, but this is a real, residual exfiltration channel for the small, sensitive excerpts the engine sees — accepted as a documented limitation for now, not eliminated and not claimed away. In its favour, the engine is a pure inference

binary that does not itself resolve names in normal operation; follow-up options (forcing a null-DNS engine configuration, blocking the engine's access to the DNS client service, or revisiting AppContainer) are tracked. The guarantee here is a hard block on the engine's *direct* outbound network, not a claim of no outbound network whatsoever. The Linux tier also requires a kernel ≥ 5.13 with Landlock active (Section 6.3); where it is unavailable, the opt-in remote backstop applies. This is stated rather than implying further hardening than what ships.

- **Config/hook coverage is not uniform across model sizes.** Detection of injection in configuration and hook files (Section 1.1) is a capability of the standard and larger on-device models. The **smallest, most resource-constrained tier — Melira Nano** — deliberately does *not* cover this modality: the configuration/code distribution differs enough from prose that carrying it cleanly exceeds that model's capacity, so it stays focused on the memory/instruction (prose) surface, and when it is the selected model the configuration-file check is not activated. This is a conscious trade-off, stated plainly rather than papered over; deployments that need config/hook coverage select a larger model. The capability itself is described here as a model design property; its activation ships with the corresponding release rather than being implied as already live in every deployment.
- **Scope.** The system protects file integrity, not the agent runtime. It defends the persistent memory, instruction, and configuration files an agent reads and acts on; it is not endpoint detection for the agent process itself and complements rather than replaces existing endpoint controls.

The unifying principle is fail-closed. A change is never silently accepted: the last known-clean state is restored only on a proven-poisoning verdict, and when the system is unsure, cannot evaluate, or its analysis component misbehaves, the suspect change is left in place and surfaced with a passive notice for review rather than destroyed. The default failure mode is safety and an auditable trail, not silent acceptance — and never the mutilation of content that may in fact be clean.