The Low Code Factory

ENGINEERING WHITE PAPER

Grounding a local model.

How avocadoctor diagnoses support tickets entirely on-premises, and why the model is not permitted to decide whether it is right.

THELOWCODEFACTORY.COM HELLO@THELOWCODEFACTORY.COM JULY 2026 · V1.0

Abstract

avocadoctor is a support agent for remote-desktop infrastructure. It reads a support ticket and the evidence bundle attached to it, investigates the logs the way an engineer would, and posts a drafted diagnosis back onto the ticket with the evidence behind every claim. The entire pipeline, the language model included, runs on a single on-premises machine; no ticket text and no log line reaches a cloud API.

This paper describes the architecture, and in particular how it addresses the property that makes most language-model-on-logs systems unusable in production: a model is fluent whether or not it is correct. The approach taken here is that the model does not decide whether it is right. The evidence does, and the decision is arithmetic.

The system runs today against our own virtual-desktop estate, with Salesforce as the ticketing front end. It is an internal pilot; it has not been deployed for a customer.

0
calls to a cloud model, enforced at a single egress point

10,000+
error strings indexed across three platforms

1
desktop-class machine runs retrieval, reasoning and the model

SECTION 01

Why the model runs on the customer's hardware

Support logs are among the worst possible data to send to a hosted model. They are dense with hostnames, usernames, internal address ranges, session identifiers, tokens and, often enough to matter, credentials that were never meant to be written down anywhere.

For most enterprises that single fact is where the conversation about AI-assisted support ends. Not because the technology does not work, but because the first step of every vendor demonstration is an unacceptable data transfer, and no amount of contractual assurance changes what is happening on the wire.

We therefore inverted the usual order. Rather than building the agent and adding a privacy story afterwards, we began from the constraint: **the model runs on your hardware, or there is no product.** Everything downstream follows from that decision — which model could be used, how much context could be afforded, and what had to be done to make a modest local model behave well enough to be useful.

A privacy boundary designed in at the start is a different object from one retrofitted at the end. The retrofitted version has exceptions in it, and the exceptions are the whole problem.

SECTION 02

The environment

avocadoctor operates against a full virtual-desktop stack rather than a single product:

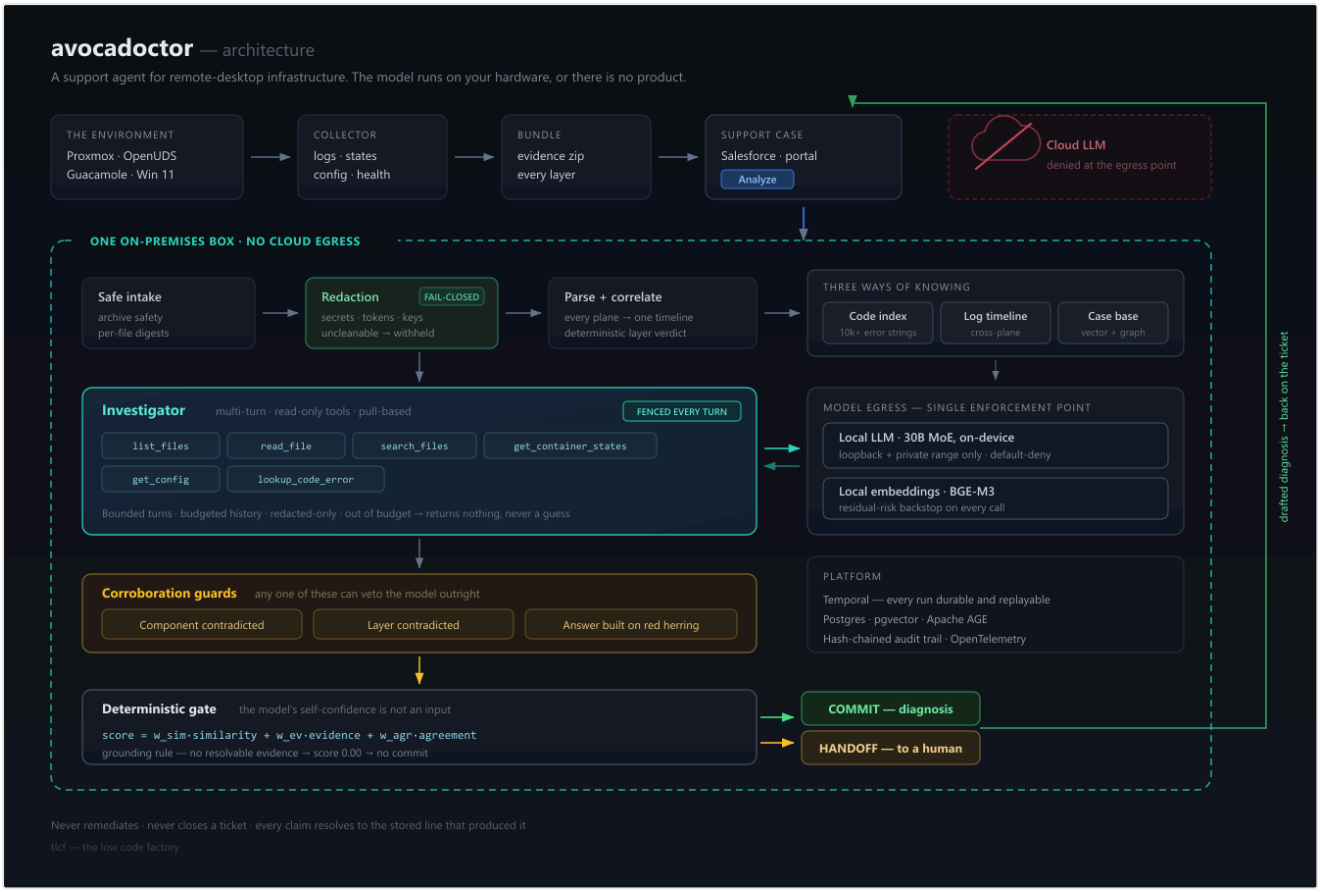
LAYER	COMPONENT
Hypervisor	Proxmox VE 8.4.19
Broker	OpenUDS 4.0.0
Browser access	Apache Guacamole 1.5.2 / guacd 1.5.2
Guests	Windows 11
Edge	nginx
Ticketing	Salesforce, behind an adapter

This matters because the interesting failures are **cross-layer**. A user reports that they cannot connect. The gateway logs a TLS error. The real fault is a dead guest agent two layers below, and the TLS error is an unrelated internet scanner hitting the public port. An agent that reads only the gateway's logs will confidently blame the gateway — and it will be able to cite genuine log lines while doing so.

> WHAT THE AGENT DOES AND DOES NOT CONNECT TO
>
> avocadoctor never connects to the infrastructure it diagnoses. A collector runs against the environment and produces an evidence bundle; the bundle is attached to a support case; the agent reads the case. **Its only external integration is the ticketing system.** The estate described above may be hosted anywhere — ours runs on GCP — and it makes no difference to the agent, which sees an archive of logs and nothing else. The analysis itself, including the language model, runs entirely on one on-premises machine.

SECTION 03

Architecture



> **Figure 1.** Everything from the redaction gate onward runs inside one on-premises boundary. Cloud providers are denied at the egress point by default, not by configuration.

Orchestration is **Temporal**. Every analysis run is a durable workflow — `land_package`, `validate_package`, `redact_and_parse`, `diagnosis_memory_search`, `diagnosis_gather_evidence`, `diagnosis_reason`, `diagnosis_gate`, and the write-back — so a crash mid-investigation resumes rather than restarts, and the full event history of every run is replayable after the fact.

SECTION 04

Safe intake and redaction

The evidence bundle is an untrusted archive from a machine we do not control. Before anything is read, two things happen.

- **Archive safety.** Path traversal, symlinks, zip bombs, declared-versus-actual size, and a per-file digest.
- **Redaction.** A versioned ruleset masks secrets, tokens, API keys, PEM material, password hashes and connection credentials across every plane of the bundle.

What the model receives is not the log file. It is the redacted projection of it:

```
INPUT
10:41:22.318 [http-nio-8080-exec-7] WARN  o.a.g.a.LocalAuthenticationProvider -
Authentication attempt from 10.21.4.17 for user "svc_backup@corp.example.com" failed.
guacd[7213]: ERROR: Unable to connect to RDP server rdp-host-04.corp.example.com:
certificate validation failed

REDACTED — EXACTLY WHAT THE WORKFLOW RECEIVES
10:41:22.318 [http-nio-8080-exec-7] WARN  o.a.g.a.LocalAuthenticationProvider -
Authentication attempt from <IP_1> for user <USERNAME_1> failed.
guacd[7213]: ERROR: Unable to connect to RDP server <FQDN_1>: certificate validation failed
```

The diagnostic content survives. The identifiers do not. Operators can run the ruleset against their own text in the console before trusting it with anything.

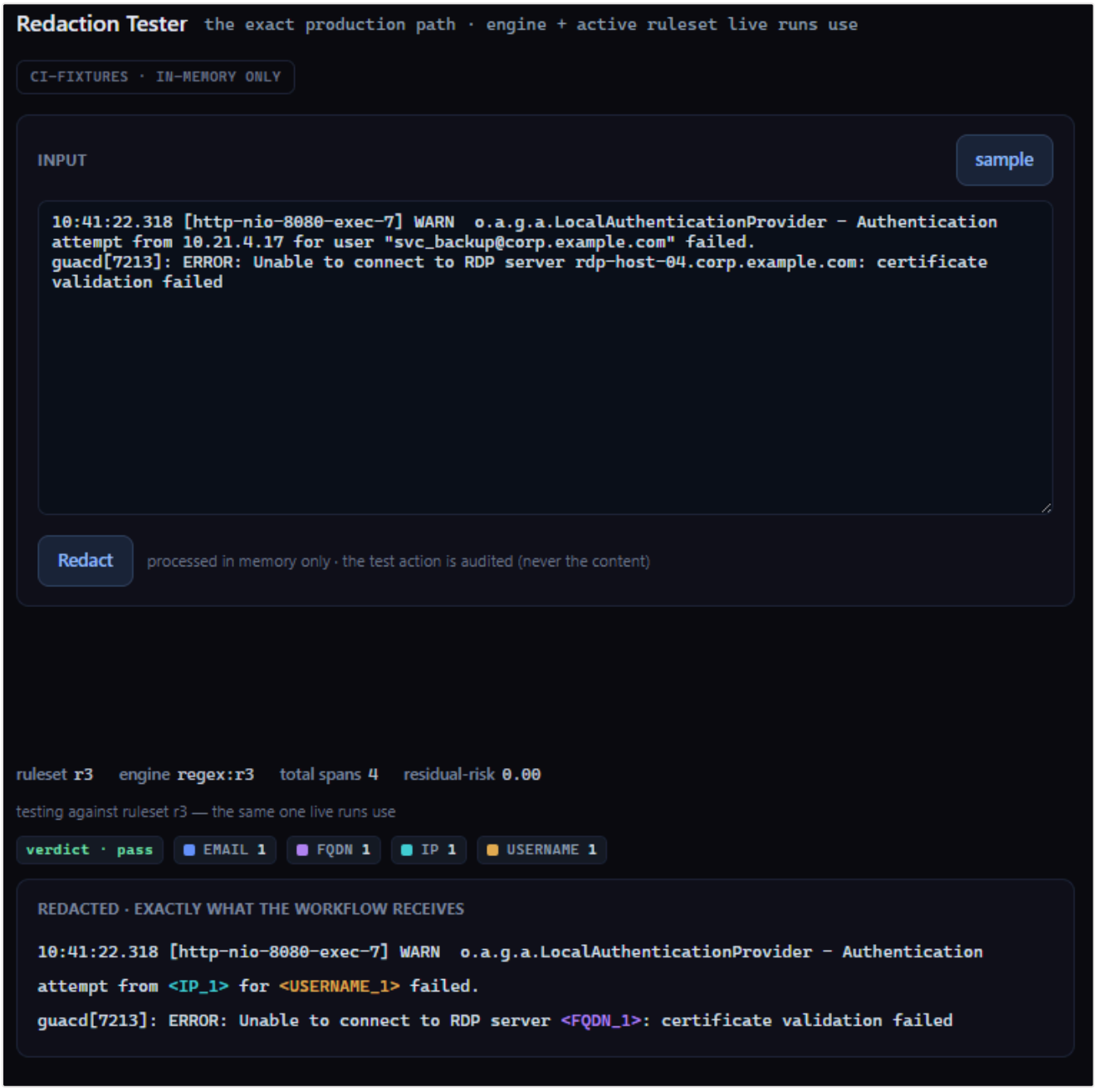


Figure 2. The redaction tester runs the same engine and the same active ruleset the live pipeline uses, with a residual-risk score and a span-by-span breakdown of what was masked.

The egress boundary

Every model call in the system passes through **one enforcement point**, which does two things.

1. **Endpoint fencing.** Default-deny. Only loopback and private-range addresses are permitted at all, and every cloud provider is refused *before any byte leaves the process*. If the local model is unreachable the pipeline stops rather than quietly reaching for an API key.
2. **A residual-risk backstop.** Everything outbound is re-scanned. If anything resembling a secret survived the ruleset, that content is withheld from the model, the run continues without it, and the withholding is recorded in the audit trail.

Critically, inside the investigator both checks run **on every turn**, not once at the start. A model that opens a new file on turn five is fenced on turn five.

SECTION 05

The investigator

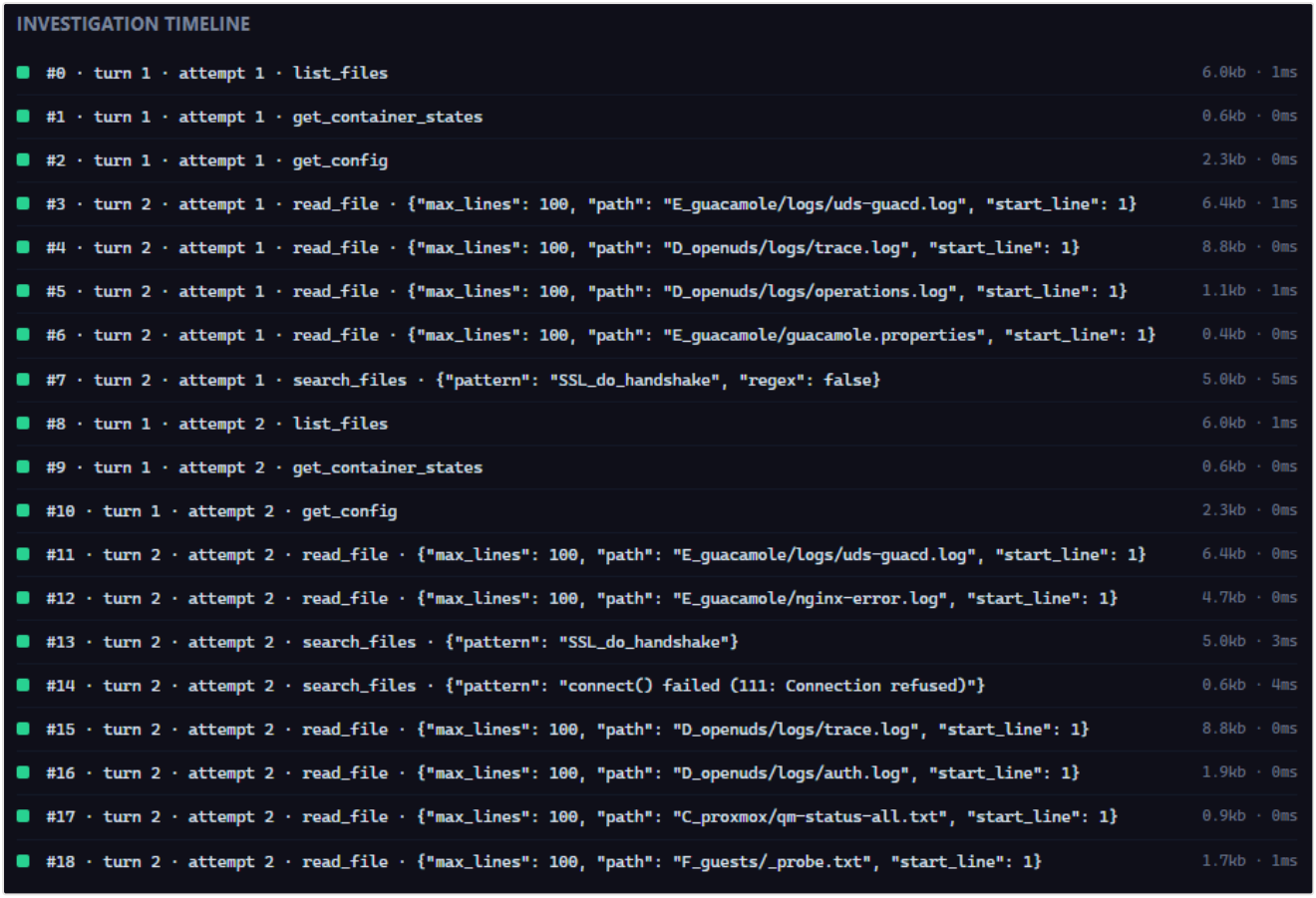
Most published approaches to log analysis with a language model amount to a single prompt with a log file pasted into it. That fails for two reasons: the bundle is far larger than any sensible context window, and the model has no way to follow a thread.

avocadoctor instead gives the model **bounded, read-only tools** and lets it run its own investigation over multiple turns.

TOOL	PURPOSE
`list_files`	Enumerate the bundle
`read_file`	Read a bounded window of a file
`search_files`	Search across the bundle
`get_container_states`	What was actually running, and what had died
`get_config`	Effective configuration, and its delta from a known-good baseline
`lookup_code_error`	Trace an error string to the source that emits it, across all three products

The properties that make this safe rather than merely clever:

- **Read-only.** No tool can mutate anything, anywhere.
- **Redacted-only.** Tools read from the redacted artifact store, never from raw source.
- **Bounded.** Every result is size-capped; the read window and the number of search hits are capped.
- **Fenced per turn.** The endpoint check and the redaction backstop re-run on every turn.
- **Budgeted.** A turn budget, and a byte budget on accumulated history. Past the cap, the oldest tool results collapse to a stub telling the model to re-issue the call if it still needs them, so a long investigation cannot silently outgrow its context window.
- **Fails to nothing.** If the model exhausts its budget it returns a *no-evidence* result that the gate rejects. It never returns a guess.



> **Figure 3.** The investigation timeline. This is not a summary of what the agent did; it is the record of it — every tool call, by turn and attempt, with arguments, bytes returned and latency.

SECTION 06

Three independent evidence surfaces

The investigator draws on three surfaces, and it must agree with itself across them before it can commit.

A code index

The source of every platform in the stack is indexed. When a log line carries an error string, avocadoctor traces it to the code path that emits it and the configuration key that triggers it — **more than ten thousand indexed error strings** across Guacamole, OpenUDS and Proxmox. The symptom stops being a mystery and becomes a location: this string is emitted by `guac_rdp_client_abort` in `error.c`, during RDP connection setup.

A correlated log timeline

Every plane of the bundle — hypervisor, broker, gateway, guest, edge — is decoded with its own grammar, timezone-normalised, and merged onto a single timeline. A deterministic correlator reduces that to a **layer verdict**: which plane is culpable, and which signals are red herrings. This happens *before the model speaks*, and it is arithmetic rather than inference.

A case base

Prior resolved cases and public knowledge, held as a hybrid vector-and-graph memory: pgvector for similarity, Apache AGE for structure. Similarity proposes candidates; the graph narrows them by how they actually relate.

SECTION 07

The confidence gate

This is the core of the system and the part we would defend hardest.

A language model asked how confident it is returns a number correlated with how confident its prose **sounds**. It is not a probability. Trusting it is how a team ships a system that is wrong in a tone of complete certainty.

So in avocadoctor the model's self-reported confidence controls **nothing**. It is recorded for diagnostics and ignored for decisions. The commit decision is a deterministic score computed from the evidence actually retrieved:

```
score = w_sim · similarity  +  w_ev · evidence  +  w_agr · agreement
```

That is: how close the nearest prior case really was, how much corroborating evidence was actually retrieved, and how far independent sources agree. Absent features contribute exactly zero. **There is no model output anywhere in that expression.**

> THE GROUNDING RULE
>
> **No evidence, no commit.** An answer with no resolvable citations scores zero, regardless of its content and regardless of its tone.

And citations must *resolve*. A claim's cited text is the actual stored log line, retrieved from the artifact store, not the model's paraphrase of it. A citation that does not resolve is dropped, and dropping enough of them collapses the score to zero.

When the agent commits, that arithmetic is written onto the ticket alongside the diagnosis, so the engineer can see *why* the system was willing to commit rather than merely what it concluded.

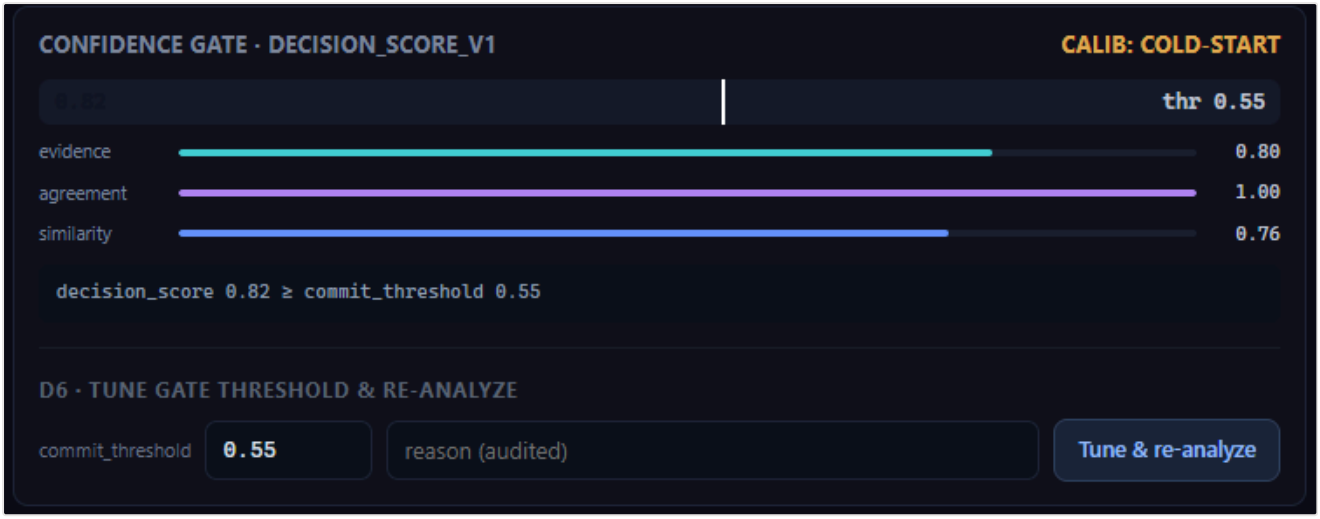


Figure 4. Three features, one score, one threshold — and no model output anywhere in the computation. The what-if panel is read-only and every adjustment is audited.

SECTION 08

Corroboration guards

Scoring is necessary but not sufficient. A wrong answer can still be well-evidenced if the model cites *genuine* lines in support of a *false* conclusion. We hit exactly this failure in development, repeatedly. Three deterministic guards can therefore veto the model outright.

- **Component corroboration.** The answer blames a component. If the evidence contradicts that component being at fault, the answer is killed.
- **Layer corroboration.** The answer blames a layer. The deterministic correlator has already computed which layer is culpable. If the model has blamed a layer the correlator ruled out with high confidence, the answer is killed.
- **Red-herring corroboration.** Some signals are loud and irrelevant — a TLS scanner hammering a public port produces hundreds of alarming lines. The parser flags these deterministically. An answer whose mechanism *is* the flagged noise is killed.

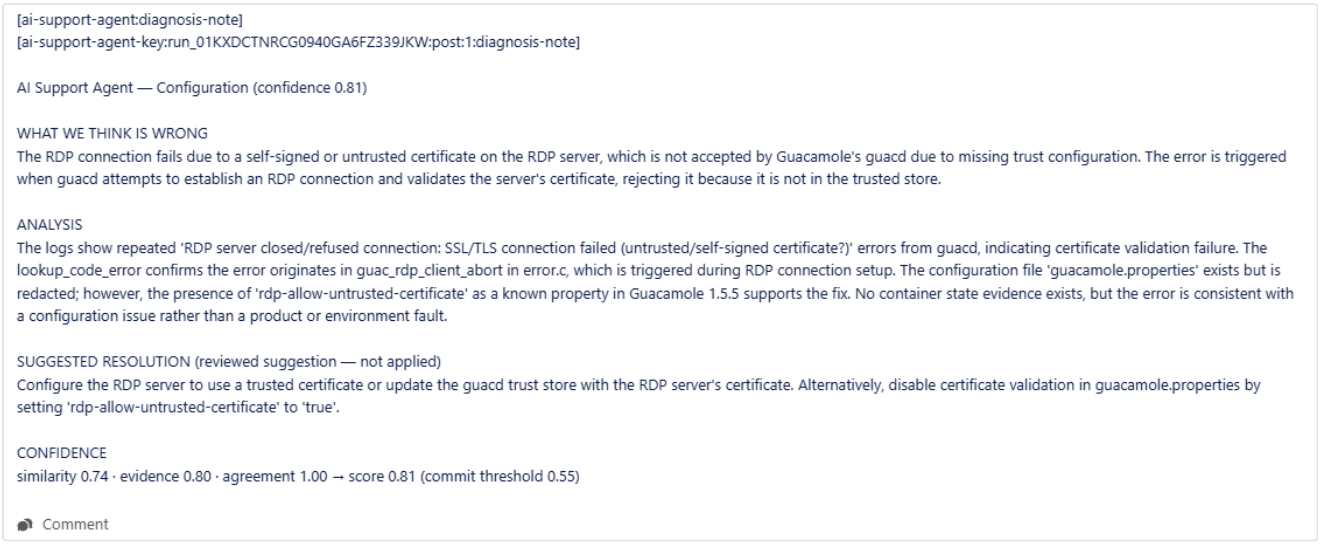
When a guard fires, the findings are withheld, the score collapses to zero, and the agent retries. If it still cannot ground an answer, it hands off to a human and says so plainly, in writing, on the ticket.

> THE CENTRAL DESIGN TRADE
>
> avocadoctor is built to prefer deferring honestly over being confidently wrong. In a support queue, a wrong answer delivered with authority sends an engineer down a three-hour dead end and erodes trust in every subsequent answer the system gives. We optimised against that failure mode deliberately, and we accept its cost: the agent defers more often than a less careful system would.

SECTION 09

A committed diagnosis

When the evidence is there, the agent commits, and the diagnosis lands on the ticket in a fixed structure: what it thinks is wrong, the reasoning, a suggested fix explicitly marked *not applied*, and the arithmetic that authorised it.



> **Figure 5.** A committed diagnosis, as the engineer sees it on the ticket.

Note the third paragraph of the analysis. The agent reports that it could not read the configuration file because redaction had masked it, and says so rather than inventing its contents. The fail-closed boundary of Section 4 surfaces in the output, honestly, as a stated limit on the claim.

SECTION 10

A refusal, and why it matters more

A live case on our own stack. Users could not connect to a Windows 11 desktop. The logs were dominated by TLS handshake failures at the edge.

The investigation ran to nineteen tool calls across two attempts. Read the sequence in Figure 3: it checked what was running, checked the configuration for drift, and then walked *down* the stack — gateway, broker, hypervisor, and finally into the guest itself.

What it concluded:

> The SSL handshake failures in nginx are caused by an external internet scan targeting the public HTTPS port, **not** a configuration or product issue. The root cause is a Windows guest VM that is running but has a non-responsive QEMU guest agent, preventing the UDS broker from assigning a valid IP address. The UDS actor reports `ip:unknown`, causing the connection to fail. The repeated `connect() failed (111: Connection refused)` errors confirm the upstream connection to the guest is never established.

Three things are worth drawing out.

It looked past the loudest signal. The TLS errors were the most visible thing in the bundle by an order of magnitude, and they were noise: an internet scanner hitting the public port. It said so explicitly and looked elsewhere.

It crossed layers. The symptom appeared at the edge. The fault was in the guest, three layers down.

It rejected a plausible memory. The case base offered a prior Guacamole handshake bug at 0.72 similarity — a superficially excellent match, and precisely the kind of thing a retrieval-augmented system pastes back with confidence. The corroboration guard rejected it, because the evidence in *this* bundle contradicted it. Without that guard, the agent would have recommended a fix for the wrong problem, fluently.

And then the part that matters most:

```
AI Support Agent handed this case to L2 (Environment, score 0.00 below the 0.55 commit threshold).
Reason: decision_score 0.0000 < threshold 0.5500; retries exhausted at retry_index 2
```

It handed off. It had a coherent, specific, cross-layer answer, and it reported high confidence in it — and it withheld that answer anyway, because it could not resolve its citations back to stored evidence. The gate does not care how sure the model sounds. It posted its ranked options to the ticket, each marked as a reviewed suggestion not applied, told the engineer plainly why it would not commit, and got out of the way.

An agent that would have committed here is an agent that will one day commit something wrong with exactly the same fluency.

SECTION 11

Operations

Everything the agent did is inspectable. A black box that reads your logs is not an improvement on the problem it was brought in to solve.

- **Run board** — every analysis, its state, its category, its score, its retries.
- **Investigation timeline** — every tool call, turn by turn, with bytes, latency and a content receipt.
- **LLM inspector** — every model call, the redacted prompt sent, token counts, and any endpoint error.
- **Redaction report and tester** — what was masked, plus a live tester for the operator's own text.
- **Evidence trail** — each claim back to the line that produced it.
- **Gate view** — the score, the three features behind it, and a read-only what-if panel.
- **Audit trail** — a hash-chained record of every decision and every external write.
- **Durable workflows** — Temporal; every run replayable, with full event history.
- **Tracing** — OpenTelemetry spans across the investigation, each turn and each tool call.

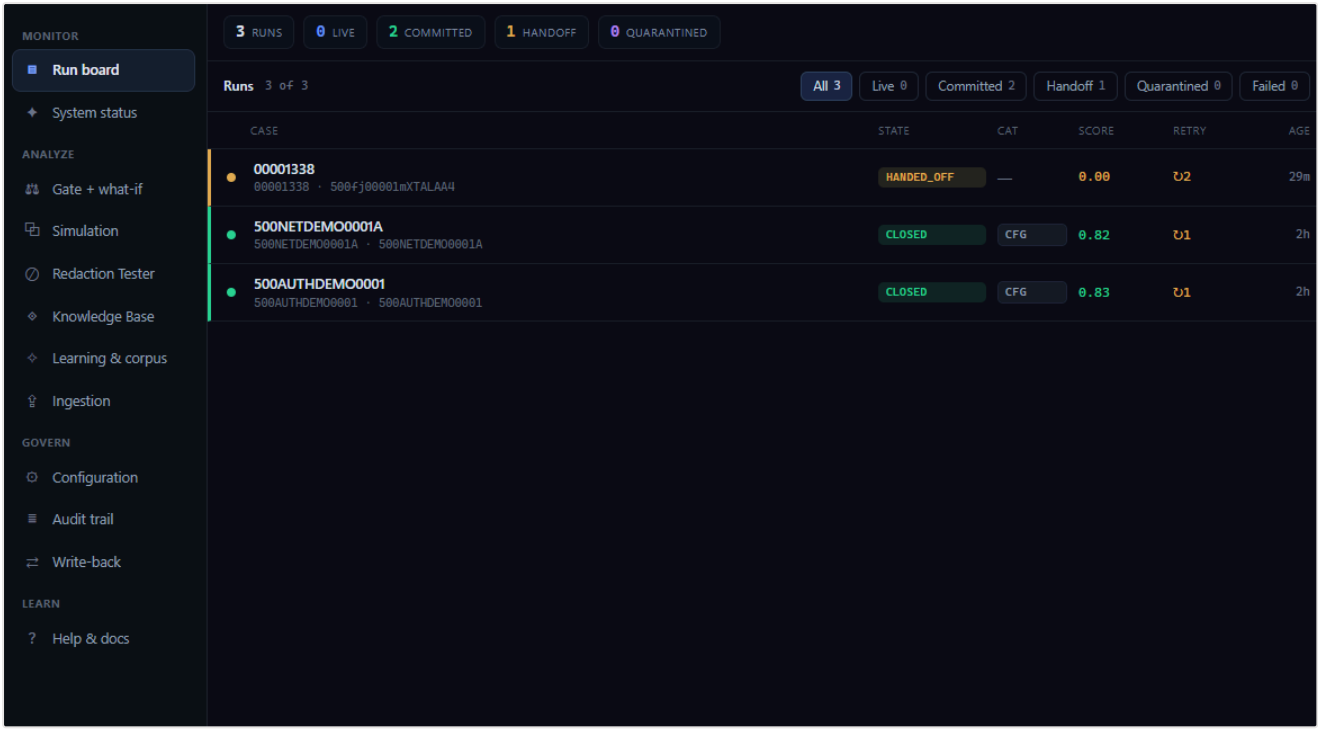


Figure 6. The run board — committed, handed off, or quarantined, and the arithmetic behind each verdict.

Figure 7. The LLM inspector — every call the system made to the model, and what came back. The endpoint is a private address on the local machine, which is the only class of address the egress point permits.

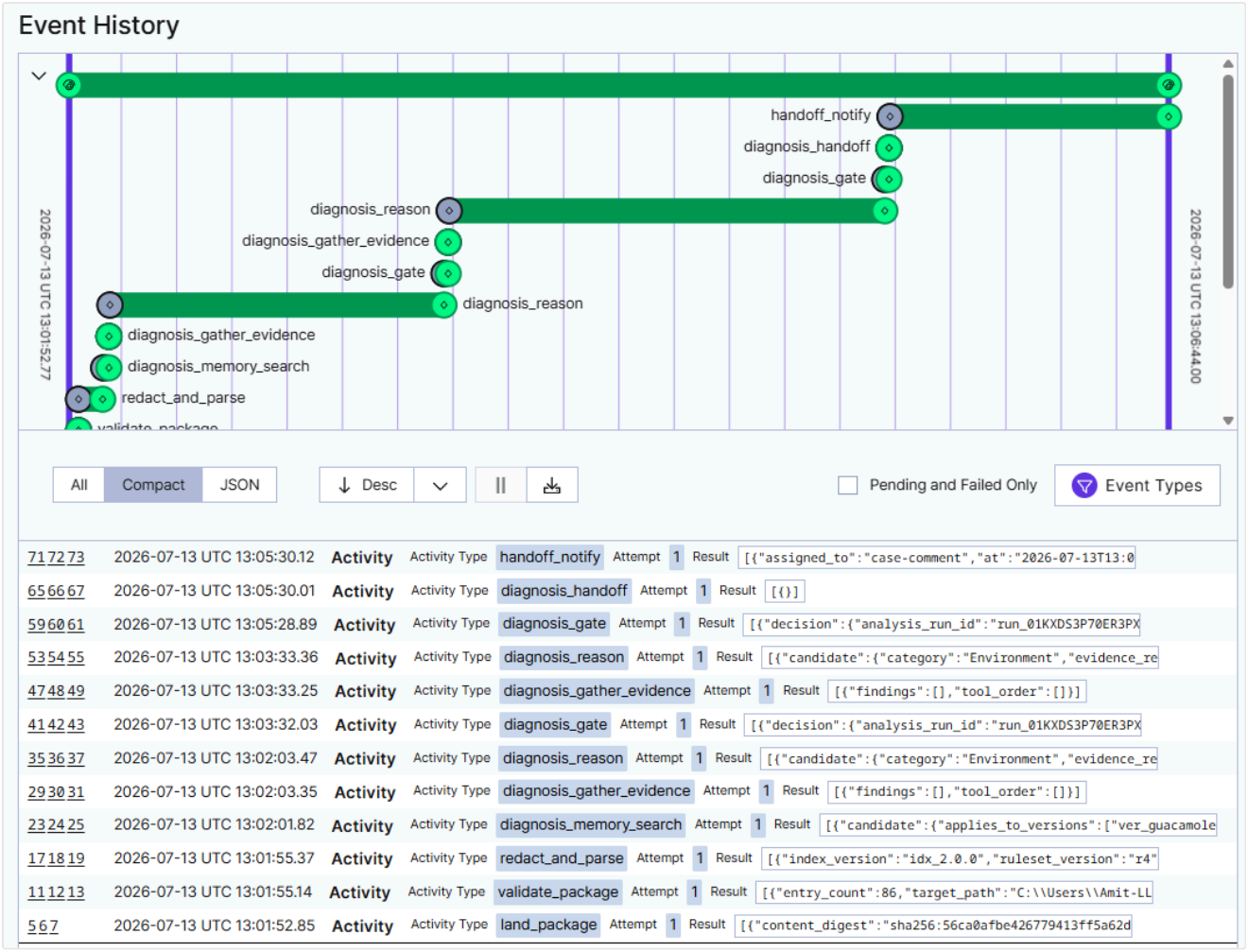


Figure 8. The Temporal event history for one run — durable, replayable, and inspectable down to the individual activity.

SECTION 12

Deployment

The whole pipeline fits on one desktop-class machine.

Hardware	AMD Ryzen AI Max+ 395 ("Strix Halo"), 128 GB unified memory
Reasoning	Qwen3-30B-A3B — a 30B mixture-of-experts model, served locally over an OpenAI-compatible API
Embeddings	BGE-M3, served locally
Data	PostgreSQL with pgvector and Apache AGE
Orchestration	Temporal
Console	React

No GPU budget, and no procurement cycle. The answer to "where does our support data go?" is "the machine in the corner of your server room", which is the only answer that makes a pilot possible in an organization where the data cannot leave.

SECTION 13

Limitations

The following are deliberate design decisions rather than gaps awaiting work.

- **It never remediates.** No service restarts, no configuration changes, no ticket closures. It diagnoses and drafts; a person acts.
- **It never publishes anything customer-facing.** The write-back is an internal, agent-only comment plus structured fields for the support team.
- **It defers when it cannot prove an answer** — by design, and readily. That is the trade described in Section 8, and we would make it again.
- **It is not built around one ticketing vendor.** Salesforce is an adapter, and so is the first-party portal. Vendor differences live at the boundary, never in the reasoning core.

One limitation is not a design decision: **avocadoctor is an internal pilot.** It runs in production against our own virtual-desktop estate, diagnosing real tickets from a real support queue. It has not been deployed for a customer, and we will say otherwise on the day that changes.

SECTION 14

Where this goes

The reasoning core, the redaction boundary, the deterministic gate and the audit trail are platform-independent. Supporting another stack means writing adapters — a log grammar, a code index, a case corpus — rather than building a new product.

The harder and more interesting frontier is **multi-layer fault localisation**: given a symptom at the edge, deterministically naming the layer at fault before the model reasons at all. A first version of that is running, and it already changes the model's answers. Making it precise across every layer of a virtual-desktop stack is the work ahead.

The Low Code Factory

An enterprise solutions company. We design, build and modernize business-critical applications, workflows, integrations and internal platforms.

thelowcodefactory.com · hello@thelowcodefactory.com · linkedin.com/company/tlcfworks

If you run a support queue for infrastructure where the data cannot leave the building, we would like to hear about it. That is the constraint this system was designed around.