TiltOS: A Durable, Trace-Verified Substrate for Governed LLM Agents over Enterprise Data

Tilt AI

Technical Report v1.0
June 30, 2026

Abstract

General agent frameworks optimize for capability: chain a model to a pile of tools and let it act. Enterprise deployments over regulated data have the opposite binding constraint — not *can the agent act*, but *can every action be trusted, bounded, attributed, and audited*. We present TiltOS, a production agent substrate built around a single thesis: *governance is a structural property of the runtime, not a behavior we hope the model exhibits*. TiltOS cleanly separates a domain-neutral *substrate* — a durable append-only event log, run-once turn execution, a single model-provider gateway, and a resumable streaming projection — from a per-customer *product bundle* that compiles each turn into a typed contract, exposes only the capability-scoped tools that turn needs, funnels every tool call through a deterministic policy pipeline, and validates the final answer against ledgers reconstructed from observed tool traces rather than from model prose. Two ideas recur. First, prefer making a bad state *unrepresentable* over making it detectable-and-correctable. Second, *verify the trace, not the output* — the only way to distinguish a correct answer from a confident-but-wrong one, and the only account of provenance a regulated customer will accept. We describe the architecture in enough depth to reimplement it, the design principles that generalize it across domains, the evaluation regime that gates its deployment, and how the same runtime scales from ten to a thousand users while the product layer is rebundled per customer.

1 Introduction

An LLM agent is a model in a loop with tools: it reads a request, calls tools, observes results, and eventually answers. The open-source ecosystem has made the *loop* easy. What it has not made easy is everything an enterprise needs once that loop touches real, sensitive, business-critical data: that the agent never fabricates a side effect it did not perform; that an answer is grounded in records it actually read; that a runaway loop cannot exhaust a budget or a rate limit; that every action carries an auditable trail of who acted, on whose data, under which policy version; and that a regression is caught before it ships, not after a customer finds it.

These are not model problems. A more capable model still hallucinates a saved document under the right prompt, still answers from training priors when a lookup would have been correct, still cannot prove *how* it reached an answer. They are *runtime* problems, and they are where the durable engineering value of an agent product lives. TiltOS is our answer to them.

The organizing decision is a hard architectural seam between two layers (Figure 1):

The substrate is domain-neutral. It owns durability (an append-only event log that is the single source of truth), execution (exactly-once, run-once turn dispatch), the one choke point through which all model traffic flows, spend and redaction guards, and a resumable event stream. It names no product concept and contains no business logic. This neutrality is machine-enforced, not aspirational.

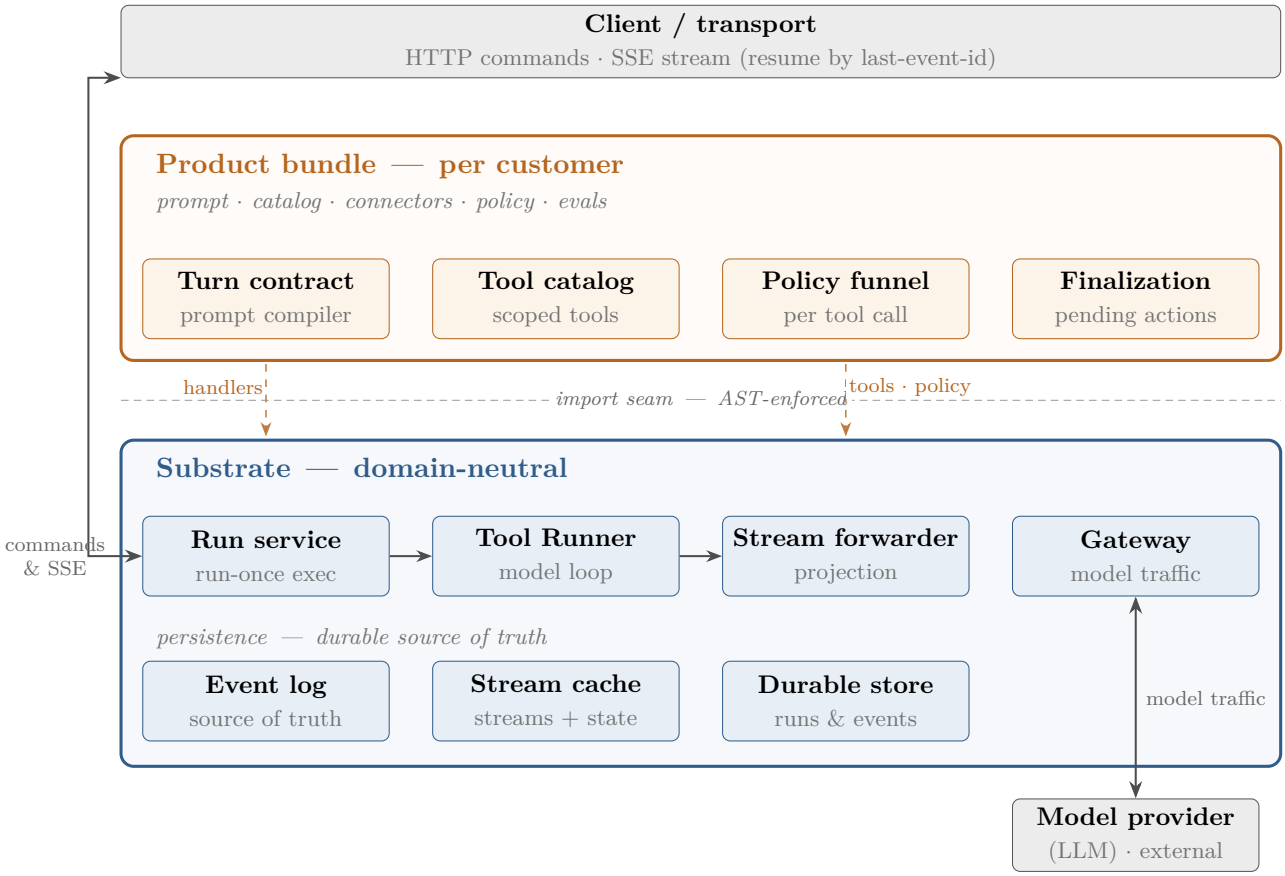

Figure 1: TiltOS architecture. A per-customer *product bundle* (upper band) — the turn-contract and prompt compiler, the capability-scoped tool catalog, the policy funnel, and pending-action/finalization handling — injects all behavior, across a machine-enforced *import seam*, into a domain-neutral *substrate* (lower band): the run service with run-once execution, the Tool Runner model loop, the stream forwarder, and the model-provider gateway, over a durable persistence tier (the event log that is the source of truth, the stream cache, and the durable store). Solid arrows are request and persistence paths; the double arrows carry client commands and the resumable SSE stream, and model traffic to the external provider; dashed arrows are the bundle's handlers, tools, and policy crossing the seam. Finalization's validation of the durable trace is detailed in Figure 4.

The product bundle is per-customer. It compiles each turn into a typed *contract*, builds a lean prompt and a capability-scoped tool surface, runs the model loop behind a deterministic policy funnel, defers and gates side effects, and validates final claims against trace-derived ledgers. The bundle is data and configuration over the substrate: a prompt bundle, a typed tool catalog, connector implementations, policy values, and an evaluation corpus.

This split is the product thesis as well as the architecture: *the substrate converges across every deployment; the bundle diverges per customer.* The asset we build, operate, and sell access to is the substrate and the invariants it enforces; the customer-specific work is authoring a bundle.

Contributions. This report documents, in reimplementable detail:

1. a durable, *run-once* execution model that gets exactly-once side-effect semantics from records and a single-attempt worker rather than from a separate workflow engine (§2);
2. per-turn *contract compilation* and *capability scoping* that bound what the model sees and may do, with a deterministic fallback that can never widen authority (§3);

3. a deterministic *policy funnel* in which every tool call is validated, budgeted, executed, and bounded, and in which failures are returned to the model as structured data rather than thrown as exceptions (§4);
4. *trace-grounded finalization*: evidence and write ledgers, reconstructed from observed tool traces, that gate the final answer and form the audit record (§5);
5. the *design principles* that make these generalize across domains (§6), an *evaluation regime* that gates deployment rather than decorating a dashboard (§7), and a *scaling and productization* model that holds the substrate fixed while rebundling the product layer (§8).

A running example. To keep the discussion concrete without fixing a vertical, we use one representative deployment throughout: an assistant for the operations staff of a services firm. Its users own *portfolios* of client *accounts*; the firm's data lives across a CRM (accounts and contacts), shared calendars, a document store, a message/email archive of interaction history, and a task list. A typical request — *"draft a follow-up to the Henderson account summarizing our last two calls"* — must resolve an entity, read the right interaction records, compose a draft, and propose it for the user's approval without ever sending anything unbidden. Every mechanism below is illustrated against this turn.

2 The Substrate

The substrate is the part of the system that does not change between customers. Its neutrality is enforced by a static-analysis test that bans both substrate→product imports and product *vocabulary* appearing anywhere in substrate source. The only module permitted to name product code is a single *composition root*, which wires product behavior in at process start. Everything the product needs to customize is reached through registries (task handlers, side-effect executors) and hooks (title generation, redaction-domain resolution, approval copy). This is what makes "extract the harness and rebundle it" a configuration change rather than a fork.

2.1 The durable event log is the source of truth

Every run is reconstructable from records. The durable store holds a small set of collections — `runs`, `tasks`, `events`, `messages`, `plans`, plus `pending_actions` and a `spend_ledger`. The `events` collection is an append-only audit log and the spine of the system: lifecycle transitions, every tool call and result, approval requests and resolutions, and stream frames all land there. Event ordering is not best-effort. Each event's sequence number is allocated by an atomic increment on the parent run document, and a unique index on `(tenant, run, sequence)` makes a gap or a duplicate a hard database error rather than a silent corruption. Assistant messages are sequenced per thread the same way and are idempotent on a deterministic message id.

This is the first and most important invariant: *compute is stateless and disposable; the log is the truth.* A worker can die mid-turn, a dyno can restart on deploy, a stream consumer can disconnect — none of it loses or reorders history, because none of it *is* history. The log is.

2.2 Run-once execution without a workflow engine

Interactive turns have a property that makes naive retries dangerous: they issue side effects. Rerunning a turn that already created a draft or a calendar event would duplicate it. The industry's reflex is to reach for a durable-workflow engine. We deliberately do not. Durability already lives in the records; a second stateful system to orchestrate them is complexity without a matching need.

Instead, exactly-once is assembled from three cheap, composable guarantees. Each turn is enqueued as a single job keyed by its run id, so a duplicate enqueue is deduplicated by the queue. The worker is configured *run-once* — one attempt, no automatic retries — so a job that issued a side effect never re-fires. And on shutdown the worker drains in-flight turns within a grace window rather than killing them mid-effect; the rare straggler past the window is reclaimed by a reaper that terminalizes orphaned runs. The result is exactly-once side-effect semantics with no orchestration engine in the path.

Two further invariants keep the interactive path simple and safe. First, *one task per turn*: an interactive turn compiles to exactly one unit of work and the executor refuses anything else, so a turn structurally cannot fan out into an open-ended task graph (multi-step background jobs run on a separate, retryable queue where re-execution is safe). Second, cancellation is *cooperative and record-authoritative*: a cancel writes a flag to the durable store, the model loop polls it between provider iterations, and a transient read failure can only ever *add* a cancel signal, never mask a real one — a healthy run is never cancelled by a flaky read.

2.3 The Tool Runner engine

The model loop itself is a thin, stable wrapper over the provider's streaming tool-execution surface.[1] The wrapper pins the provider SDK to an exact version and fails fast on a surface drift, so an upstream change surfaces in continuous integration rather than in production. Per iteration it streams text deltas to the caller, observes tool-use blocks, dispatches each to the product's tool layer, and records a *trace* of what was requested and what actually executed.

The trace is not incidental — it is the substance later stages verify against. For each call the engine records timing, a bounded result summary, and a separately bounded "repair" view, and it scrubs secrets *before* truncation so a redaction can never be sliced in half. Transient provider failures (overload, rate limiting, specific 4xx/5xx classes) are classified and retried; everything else propagates. The engine reports *why* it stopped — the model finished, or it hit the iteration cap with tool calls still pending — which downstream logic needs in order to distinguish "done" from "truncated."

Prompt caching is treated as a first-class cost lever. The static prefix (system instructions and tool schemas) carries one cache breakpoint, and a single breakpoint slides forward across the message history each iteration, so a multi-step turn re-pays the cache write for only its newest turn rather than its whole transcript.

2.4 The provider gateway and cost containment

All model traffic — interactive turns, the lightweight per-turn classifier, post-turn finalization, nightly batch evaluation — flows through one gateway. Centralizing the provider client buys three things at once: a single admission point that reserves capacity for interactive turns ahead of background work; per-call metering and anomaly alerting; and one place to swap providers. A continuous-integration grep gate forbids constructing a provider client anywhere else, so the choke point cannot be bypassed by accident.

Spend is contained by an atomic ledger. Token usage is converted to integer micro-dollars and accumulated per tenant and globally, so concurrent increments cannot drift. Before a turn calls the model, an admission check compares projected spend against a cap; on a genuine breach it *fails closed*, returning a structured "capacity paused" result to the user having spent nothing, and it *fails open* on a ledger read error so a bookkeeping glitch never takes the product down. Soft brownout thresholds below the hard cap shed background load first.

[1] The reference implementation targets a frontier provider's streaming tool-runner API, its prompt-caching mechanism, its batch API, and its server-side tools (web search/fetch). The provider gateway (§2.4) is the only place this coupling lives, which is what keeps the provider pluggable.

Parameter	Value	Purpose
Context working set	60,000 tok	per-turn window ceiling
Per-run token budget	160,000 tok	runaway backstop (warn, then clamp)
Per-run tool-call budget	18 calls	runaway backstop (∼2 typical)
Default result cap	3,000 chars	per-tool payload bounding
Interactive turn timeout	900 s	hard ceiling per turn
Shutdown drain	25 s	grace to finish a run-once turn
Provider concurrency	16 (4 reserved)	gateway admission; reserve interactive
Global spend cap	$30 / window	fail-closed; brownout at 0.8 / 0.9
Pending-action TTL	24 h	approval expiry
History hydration	20 turns	context-window bound
Model tiering	frontier / fast	main loop vs. classifier & post-loop

Table 1: Representative safety-rail and budget parameters. Caps are deliberately set well above normal usage so they fire only on genuine runaway.

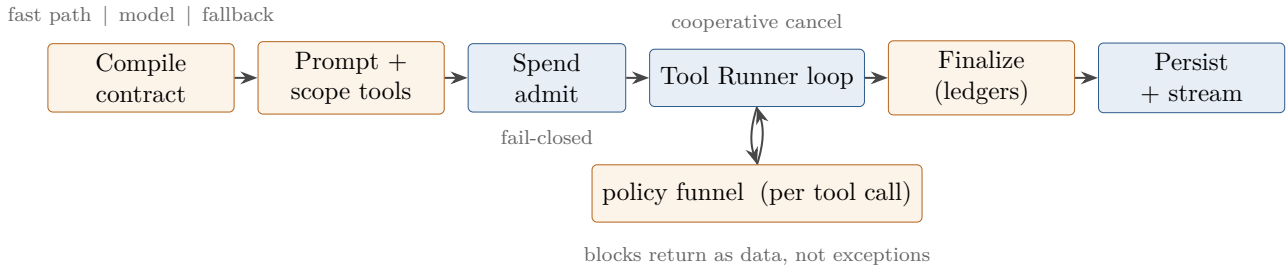

Figure 2: The single-turn lifecycle. A turn is compiled into a typed contract that scopes the prompt and tools; the loop runs under spend admission and cooperative cancellation; every tool call passes the deterministic policy funnel (§4); finalization grounds the answer in the trace before persistence.

Table 1 collects the safety-rail parameters. Their values encode a philosophy discussed in §6: budgets are high-deductible backstops set far above normal use, not tight targets. The per-run tool-call budget, for instance, sits roughly an order of magnitude above a typical turn; when it trips, the signal is "something has genuinely run away," not "tune the cap down."

2.5 Streaming as a resumable projection

The user watches a turn unfold in real time, but the stream is never the record. Events are appended to a per-run stream in a cache tier behind a transport-neutral sink, with a persisted cursor. A reconnecting client replays from its last seen id and misses nothing; a client that never connects loses nothing, because the durable log already has it. Snapshots are throttled — coalesced text deltas, but always-immediate tool and terminal events — so streaming cost is linear in real progress, not in token chatter. The wire protocol is plain HTTP commands plus Server-Sent Events, which resume cleanly across network blips and proxies without a bespoke socket protocol.

3 The Product Layer: Per-Turn Compilation

If the substrate decides *how* a turn runs durably, the product layer decides *what* a turn is allowed to see and do. It does this by compiling each turn, before the main model loop starts, into a typed contract that both seeds the loop and is used to validate its output (Figure 2).

3.1 The turn contract

The contract is a typed record describing the turn's shape: which entities it targets, which *evidence domains* it must and may gather (account, interaction, transcript, profile, document, message, task, segment, calendar, web, delegated), whether it carries a *write intent* (and of which class), which capability groups and tool budget it gets, and which validators its output must satisfy. In the running example, "draft a follow-up to the Henderson account" compiles to: target entity = the Henderson account; required evidence = account identity and recent interactions; write intent = *message draft*; budget profile = the larger "composition" profile.

Compilation runs a small ladder, fastest and safest first. A bare confirmation ("yes," "go ahead") against an outstanding approval short-circuits deterministically — no model call, no ambiguity. Otherwise a fast, cheap classifier model emits the contract via a forced, schema-validated tool call. If that classifier returns nothing usable, a *degraded fallback* synthesizes a read-only contract: gather identity plus a safe set of read domains, and — critically — *carry no write intent*.

That last detail is a structural safety property, not a nicety. The failure mode of a contract compiler is not "it picks the wrong tools"; it is "under load or ambiguity, it authorizes a write it should not have." By construction, every degraded path in TiltOS loses write authority. The system can fail to be helpful; it cannot fail *open*.

3.2 Clarify only before an irreversible act

A model that asks a clarifying question before every ambiguous request is safe and useless. TiltOS ties clarification to reversibility: a turn pauses to ask only when the request is under-specified *and* it carries a write intent. Reads and analyses always run — a slightly-misread question costs a re-ask, not a wrong side effect. Some intents are even floored deterministically: a follow-up drafted from a structured "reply to this message" task is forced to the draft-write intent regardless of the classifier, because its provenance is unambiguous, while a generic "summarize this thread" task is left to the model because it genuinely could go either way.

3.3 Capability-scoped prompt and tools

A model handed fifty tools and a kitchen-sink system prompt is slower, more expensive, and measurably more error-prone than one handed the eight tools the turn actually needs. TiltOS compiles both prompt and tool surface from the contract's capability groups.

The prompt is assembled from a small core persona, the active entity's context, and then *only* the instruction capsules for this turn's capabilities, closed by a final-answer contract that forbids claiming a side effect the turn did not perform. Each capsule and the assembled prompt are content-hashed and the hashes are stamped on the run, so prompt content is a versioned, reviewable artifact: any edit forces a hash change that a pinned test catches, and any run can be traced to the exact prompt that produced it.

The tool surface is derived the same way. A typed *catalog* is the single source of truth for every tool (Table 2); the set exposed for a turn is computed from the turn's capabilities, typically eight to twelve of several dozen tools. One principle overrides relevance here (§6): a small *floor* of tools — identity resolution, entitlement, the authoritative source-of-truth lookup for each core entity — is exposed on *every* turn, because the cost of their absence (a confidently wrong answer from missing context) is categorically worse than the token cost of always offering them.

Field	Role
`effect`	coarse risk class: read / sensitive_read / write / evaluation / subagent
`capability_groups`	which turn capabilities expose this tool
`evidence_domains`	domains a successful read *attests* (feeds the evidence ledger)
`write_class`	write category (feeds the write ledger); required on writes
`write_evidence_domains`	domain a confirmed write attests
`retry_safe`	may auto-retry on transient failure (never true for writes)
`summarizable`	oversized results may be model-summarized
`max_chars`	hard cap on result size returned to the model
`requires_confirmation`	pause for explicit user confirmation before executing
`available_in_test_mode`	usable in the per-user sandbox
`untrusted_content`	wrap result in untrusted-content fences
`data_domains`	sensitivity tags driving write-boundary redaction
`display_label`	label shown in the activity feed

Table 2: The typed tool specification. Because the catalog is the single source of truth, exposure maps, ledger classes, evidence domains, redaction tags, and UI labels are all *derived* from these fields rather than maintained as parallel lists. The catalog self-validates: a write tool must declare a `write_class` and must not be `retry_safe`; every schema is closed.

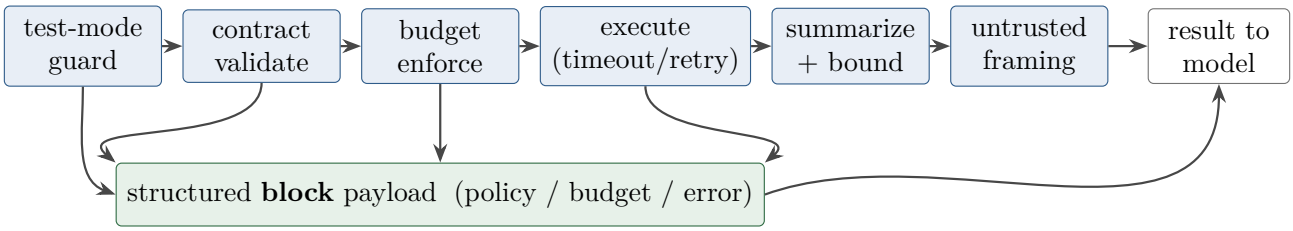

Figure 3: The per-tool-call policy funnel. Any guard may halt the call, but a halt is not an exception: it becomes a structured, model-visible *block* payload, so the model can read why it was stopped and adapt within the same turn. Execution errors are normalized the same way.

4 The Policy Funnel and Deferred Side Effects

Every tool call — without exception — passes through the same deterministic pipeline before, during, and after execution (Figure 3). The pipeline is fixed code, not model judgment, and its stages run in a fixed order.

A test-mode guard refuses tools unavailable in the per-user sandbox before any budget is spent. Contract validation checks the call's arguments against the tool's declared schema and any tool-specific contract. Budget enforcement debits the call against the turn's tool-call and token budgets. Execution applies a timeout and retries only the tools marked retry-safe. Result bounding optionally summarizes an oversized read with the fast model, then hard-caps it to the tool's character limit. Finally, content-bearing reads are wrapped in explicit *untrusted-content* fences — applied after bounding, so truncation can never cut the closing fence — instructing the model to treat the enclosed text as data, not instructions, which is a baseline prompt-injection defense for any tool that returns third-party content.

The unifying decision is that *a blocked or failed call is data, not an exception.* A policy denial, a budget exhaustion, a timeout, a downstream error — each becomes a small structured payload the model receives as the tool's result. The model sees that it was blocked and why, and can adapt within the same turn (narrow a query, choose another tool, or explain the limit to the user) instead of the turn crashing opaquely. Failures become legible to both the model and the audit log.

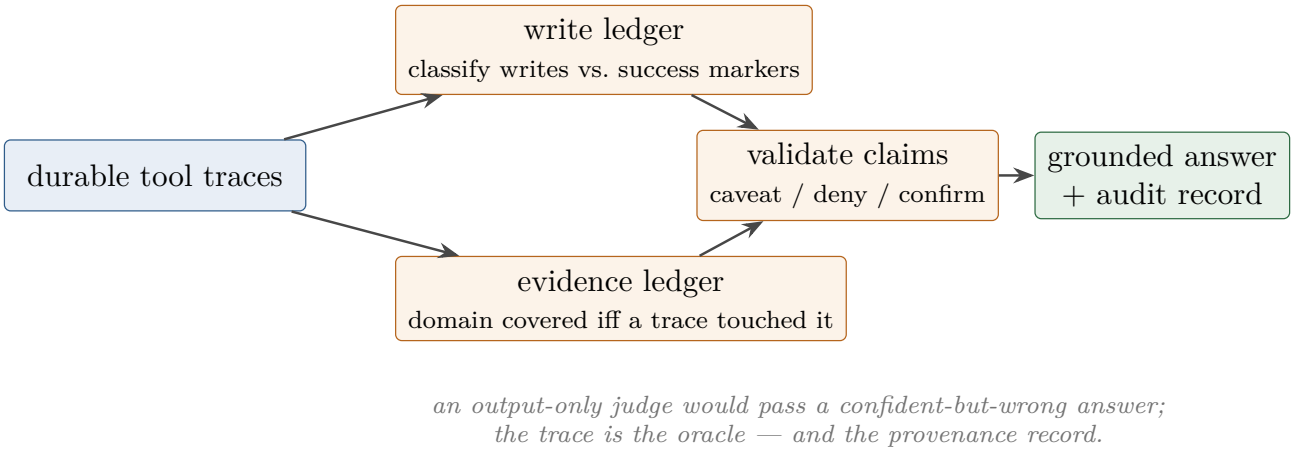

Figure 4: Trace-grounded finalization. Two ledgers are reconstructed from observed tool traces: a *write ledger* (did a write actually succeed?) and an *evidence ledger* (which domains were actually read?). Final claims are validated against the ledgers, and the ledgers themselves become the durable audit record.

4.1 Side effects are deferred and gated

TiltOS never performs a consequential write in the middle of a reasoning loop. When the model decides to act — draft a message, create a calendar event, deliver a document — it calls a *propose* tool, which records a durable *pending action* with a deterministic, idempotent identifier and a time-to-live, and returns to the model a handle explicitly stating that nothing has happened yet. The turn ends with a proposal, not an effect. The user approves or rejects through the durable run; only on approval does a separate, exactly-once executor (guarded by an atomic claim-for-execution transition) perform the underlying write. A small set of particularly consequential tools additionally pause for explicit in-turn confirmation.

In the running example, the assistant composes the Henderson follow-up and proposes it; the user sees "draft ready for your approval," approves, and only then is the draft created. There is no path by which an ambiguous request silently sends mail. This is the side-effect counterpart to the read-side floor: *authority to act is granted explicitly, late, and once.*

5 Trace-Grounded Finalization

The hardest problem in a prose agent is not getting a good answer — it is knowing whether the answer you got is legitimate. A confident, fluent, *wrong* answer reads exactly like a correct one. An output-only check — even an LLM grading the final text — cannot tell them apart, because it is reading the same fluent prose that fooled you. TiltOS solves this structurally: it grounds the final answer in the *trace* of what the turn actually did (Figure 4).

After the visible text streams, a finalization pipeline runs — repair, structured finalize, contract guards, citation compaction, optional grounding, evidence validation — and builds two ledgers from the trace alone.

The *write ledger* answers "what side effects actually happened?" It is built *only* from tool traces, never from a job's success status. Each write trace is classified — attempted, completed, failed, unconfirmed, incomplete — and a write counts as completed only if the trace carries the success marker the writing tool emits on success, with the producing strings and the detecting patterns deliberately kept in one place so they cannot drift apart. The contract then reconciles the model's prose against this ledger: a claim of "I've drafted the email" that the ledger does not confirm is corrected or denied

before the user ever sees it. This is what makes "TiltOS does not fabricate side effects" a property of the runtime rather than a hope about the model.

The *evidence ledger* answers "what was actually read?" Its floor invariant is that coverage is *trace existence*, not claim grounding: a domain is covered the moment a successful read touched it, recorded with source references and whether the read was truncated. If a turn claimed required evidence it never gathered, the output validators escalate to a visible caveat; partial or truncated coverage is recorded as telemetry rather than blocking a useful answer.

Two finer points complete the picture. *Citations are an attribution layer, not a grounding layer*: a citation registry assigns stable per-turn numbers and strips orphan markers that resolve to no source, but a resolvable citation only means "this points at a real source," not "this claim is true." Grounding proper — the question of whether a stated fact is supported by what was read — is handled, where enabled, by making it a *lookup* rather than a second opinion: facts are answered extractively from structured field values carrying source lineage captured at read time, so "is this grounded?" becomes "does a fact with this value and span exist?" — a join, not a judgment. An optional bounded span check can additionally caveat specific high-stakes assertions (dates, figures, named plans) that lack a supporting excerpt.

Finally, all of this is durable. The finalization outcome — write states, guard decisions, contradiction classes — is persisted, so the question "how did this answer get derived, and did the system catch itself?" is answered from records, not by re-reading chat text. That durable derivation is simultaneously the debugging tool and the compliance artifact.

6 Design Principles

The mechanisms above are instances of a smaller set of principles that generalize beyond any one domain. The north star: *prefer making a bad state unrepresentable over making it detectable-and-correctable*; reach for structural simplicity before clever machinery.

1. **Tier the tool surface by cost-of-absence, not relevance.** What a turn is "about" and what a turn *needs to be correct* are different axes. Floor the prerequisites for acting at all; route the rest.
2. **Bias toward the invisible failure, then build its detector.** Over-exposure costs measurable tokens; under-exposure costs an occasional silent wrong answer. Default to the visible-cost side and instrument the rest.
3. **Never let a stochastic component be the sole gate on a hard prerequisite.** Use the model to optimize ("which extra tools?"); enforce correctness primitives unconditionally or with a deterministic post-check.
4. **Verify the process, not just the output.** Trace-level verification catches illegitimately-reached answers that output grading passes — and is the compliance story.
5. **Caps are high-deductible backstops, not tight targets.** Set them well above normal use; when one trips, ask what real failure it exposed, not how to lower it.
6. **Source-of-truth tools go broad; niche tools stay scoped.** Per entity, name the one authoritative tool and give it reach.
7. **Don't run a durable orchestrator when you don't need durability.** Durability belongs in the records; a workflow engine is a second source of truth you must now keep consistent.
8. **Make prompt content a versioned, reviewable artifact.** Content-hash it; stamp the version on every run.

9. **Reproduce before fixing; encode the invariant as a test.** For harness guarantees, assert the structural property directly so it fails in CI, not in prod.

10. **Assert against stable signals, not stochastic internals.** Check membership against the full planned set; pin exact values only on deterministic inputs.

11. **Don't pattern-match a semantic problem.** Regex over model-generated prose is a brittle proxy that silently rots. Drive control flow from structural signals (stop reasons, typed outputs, traces) or push the judgment to the model under a validated contract; reserve regex for closed grammars you define.

12. **Manufacture an oracle; don't simulate one with a per-turn verifier.** A coding agent looks grounded because its domain ships a free deterministic oracle (compile, run, test). An oracle-free prose harness should not bolt on an LLM-checks-an-LLM verifier; it should make grounding a lookup and otherwise be structurally *more conservative* — re-read, caveat, refuse to assert un-read values. *Port the shape of the coding agent; reject its confidence model.*

The productization consequence is sharp: *the reusable asset is not the engine or the catalog — it is the invariants and the detector.* Which tools are floor is per-customer configuration; that there *is* a floor, asymmetrically defended and trace-verified, is the platform.

7 Evaluation as a Deployment Gate

Evaluations in TiltOS are deployment gates, not dashboards. The organizing rule mirrors the architecture: *assert the deterministic, measure the stochastic.* Everything around the model — routing, policy guards, budgets, ledgers, persistence, the wire contract — is asserted with hard pass/fail checks; what the model *says* is judged and trended statistically, never failed on a single sample.

A layered test model spans the cost/fidelity spectrum:

L0 Unit pure logic — contract fast paths, policy guards, catalog consistency, ledger markers, prompt-hash stability. No model, no database; runs on every push.

L1 Integration tool implementations against local datastores; no model.

L2 Harness end-to-end one full turn through the real stack with a *scripted* model client emitting canned tool calls and text. Fully deterministic — the keystone that makes the orchestration testable without provider variance.

L2.5 Transport contract one full turn over the *real* HTTP/SSE transport with a scripted model, asserting the complete client event contract (lifecycle, tool framing, an approval round-trip, title presence and ordering, terminal events, resume by last-event-id). Provider-free, so it runs as a required CI gate. A companion *blocked-path* meta-test forces a regression and asserts the gate goes red — proving the gate guards, not merely that it passes.

L3 Replay live model against recorded tool outputs, with a deterministic audit *and* an LLM judge per fixture. An in-memory fake of the document store lets these exercise the real document tools without a live integration.

L4 Live nightly batch judging plus online sampling, with trends and alerts.

Each evaluated case carries two independent verdicts. A deterministic *audit* checks required and forbidden tools, call sequences, required side effects, and the contract's own field expectations — and gates per fixture (a hard fail). An LLM *judge* returns pass / needs-review / fail with structured rationale — and gates only in aggregate, on a suite-level pass-rate threshold, never on a single sample. A flaky per-sample judge gate is worse than no gate; a stable statistical one is a genuine signal.

The corpus grows by a flywheel. Every nightly thread the judge flags is automatically staged as a harvest candidate — an idempotent record keyed by thread — feeding a human "bless" queue that accretes each real-world failure into the replay corpus. Coverage is measured as fill of a turn-shape matrix (capability × read/write × surface × happy/error), not as line coverage, because the thing that breaks an agent is an unseen *shape* of turn, not an unrun line. A per-commit scoreboard records latency and token percentiles against a baseline as a report, never a hard per-PR fail, because latency is stochastic and belongs on the measure side.

8 Scaling and Productization

TiltOS is designed so that growth re-platforms the edges while the core stays fixed. Six invariants hold constant at every scale:

1. a durable event log is the source of truth; all compute is stateless and disposable;
2. control plane and data plane are separate — configuration is versioned, and every run records which version produced it;
3. a single provider gateway owns all model traffic;
4. streams are resumable projections, never the record;
5. side effects flow through gated, credential-isolated executors;
6. evaluations are deployment gates.

Against these, scale is a sequence of edge replacements, not rewrites. From a handful of users to ten, the work is hygiene: prompt caching, linear streaming, model tiering, explicit throttles, a stale-run reaper, batch APIs for background jobs. To a hundred, the edges re-platform onto an elastic substrate while the core is untouched: containerized services with autoscaling — web tier scaled on connections and, the binding constraint, the turn-worker pool scaled on queue backlog — a hardened provider gateway, repository-enforced tenancy, and event-log tiering to object storage. The managed database is retained for its native search and vector capabilities (which the retrieval path depends on) and reached privately. The justification for this move is *capability, not cost*: graceful draining of in-flight turns across deploys, autoscaling, and an enterprise security posture. At this scale model spend dominates the infrastructure bill by one to two orders of magnitude, so the substrate optimizes for correctness and control, not for shaving compute.

To a thousand users the harness becomes a platform: cellular deployment (independent vertical slices, with data-residency and multi-region as a consequence), a credential vault with an injecting proxy and egress allowlists so connector secrets never enter model context or logs, and quality infrastructure promoted to a first-class subsystem.

8.1 The platform converges; the bundle diverges

The reason one runtime serves many customers is that everything customer-specific is isolated in a *bundle* (Figure 5): a prompt bundle, a set of typed tool-catalog entries, connector configurations, workflow templates, policy values, an evaluation corpus, and a UI label map. The substrate, the policy funnel, the ledgers, and the invariants are identical across deployments; only the bundle changes. The engineering boundary is a small SDK — create a run, send a message, attach to and resume the stream, approve or reject, cancel, and register a catalog, bundle, and evaluation suite. Onboarding a new domain is authoring a bundle and its evaluation fixtures, not forking the platform.

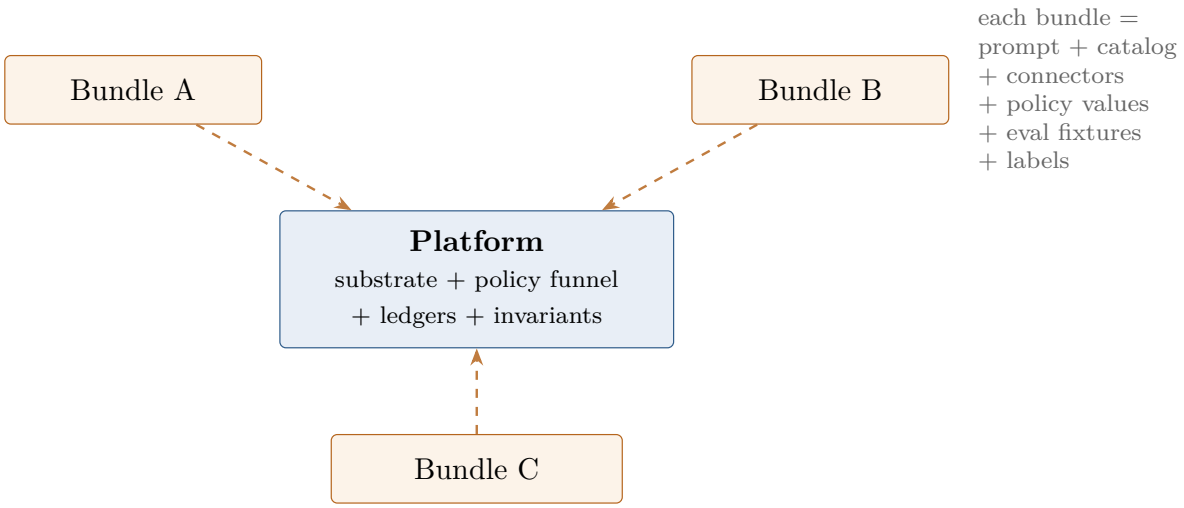

Figure 5: One runtime, many bundles. The platform (and the invariants it enforces) is the converging asset; the per-customer bundle is the diverging product. Same runtime, different bundles.

8.2 Enterprise-data governance

For regulated data the governing question precedes any code: *what data will be in the system, and what laws govern it?* It is a legal gate as much as a technical one — some regimes may forbid sending certain records to a third-party model processor at all — and the system is built to make the answer enforceable rather than aspirational. The same mechanisms that make TiltOS correct make it governable: repository-enforced scope on every read and write; least-privilege connector credentials fetched by executors and never placed in model context or logs; input and output moderation as a separate classification step; untrusted-content framing as injection defense; explicit approval gates and spend caps; and an audit trail that records, per action, the actor, tenant, surface, connector, tool, the policy and bundle versions, the model, tokens, cost, and outcome. The discipline extends past code: separation of duties — so that no single person writes the code, holds production credentials, can read regulated data, and approves their own deploys — is itself a control the architecture assumes.

The same posture governs how the platform extends. *Governance is ours; retrieval is pluggable.* Tool selection by capability scoping is deterministic at the scale of a few dozen tools; beyond roughly a hundred connectors, discovery is delegated to the provider's native tool search through a custom-client variant in which our contract still computes the in-scope tool set — renting best-in-class retrieval plumbing without surrendering the governance contract or a cacheable prompt prefix. Long-horizon memory, context compaction, and context editing are likewise adopted from provider-native primitives, backed by tenant-scoped records, rather than rebuilt — with the deliberate exception of execution modes that would route regulated data outside the policy funnel, which are excluded by policy.

9 Related Work

TiltOS builds on the now-standard reason-act agent loop [1, 2] and on tool- and API-augmented models [3, 4, 5], but its concern is orthogonal to making the loop more capable: it is making the loop *governable* in production. Where retrieval-augmented generation [6] grounds generation in retrieved text, TiltOS grounds the *final claim* in the trace of tool calls the turn actually made, and treats citation as attribution distinct from grounding. Where LLM-as-judge evaluation [7] grades outputs, TiltOS treats output judging as a statistical signal and reserves hard gates for deterministic audits of the trace — precisely because a fluent wrong answer defeats output-only grading. Its durability

model is workflow-as-data: rather than adopt a durable-workflow engine [8], it keeps the durable record as the single source of truth and reduces execution to run-once dispatch, which is sufficient because interactive turns are single-step. The capability-scoping, policy-funnel, and trace-verification contributions are, to our knowledge, the parts least served by existing agent frameworks and the most load-bearing for a regulated deployment.

10 Conclusion

The durable value in an enterprise agent product is not the model and not the loop — those are rented and commoditizing. It is the runtime that makes the loop trustworthy: a durable log that cannot lose or reorder truth, run-once execution that cannot duplicate a side effect, a contract that cannot fail open, a policy funnel that turns every failure into legible data, ledgers that ground the answer in what actually happened, and an evaluation regime that gates the door. TiltOS packages these as a domain-neutral substrate beneath a per-customer bundle, so the same governed runtime can be pointed at a new domain by authoring configuration rather than forking a system. The recurring move is the same one throughout: make the bad state unrepresentable, and when you cannot, verify the trace rather than trust the prose.

References

[1] S. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," *ICLR*, 2023.

[2] N. Shinn et al., "Reflexion: Language Agents with Verbal Reinforcement Learning," *NeurIPS*, 2023.

[3] T. Schick et al., "Toolformer: Language Models Can Teach Themselves to Use Tools," *NeurIPS*, 2023.

[4] E. Karpas et al., "MRKL Systems: A Modular, Neuro-Symbolic Architecture," arXiv:2205.00445, 2022.

[5] S. Patil et al., "Gorilla: Large Language Model Connected with Massive APIs," arXiv:2305.15334, 2023.

[6] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *NeurIPS*, 2020.

[7] L. Zheng et al., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," *NeurIPS*, 2023.

[8] Workflow-as-code durable execution systems (e.g. Temporal, AWS Step Functions); see vendor documentation.