OMNAFY

The Omnafy platform architecture

An MCP gateway with governance in the request path

Audience	Architects and technical evaluators
Version	1.0
Published	August 31, 2026
Online	omnafy.com/whitepapers/platform-architecture/

Abstract

Most systems that govern AI access do it beside the request path: a proxy that logs, a dashboard that reports, a policy engine consulted when someone remembers to consult it. Anything beside the path can be routed around, and anything that can be routed around is not a control.

Omnafy puts governance in the path. There is exactly one aggregate MCP endpoint in a deployment, every tool interaction crosses it, and the two enforcement points inside it run before a model sees a tool list and again before a call is dispatched. Our own administrative API is published through the same endpoint under the same checks, which means administering the platform produces the same tiered, audited tool calls as using it. There is no privileged internal client.

This paper is the engineering description of that system: why MCP is the universal interface, how the five planes divide responsibility, what the gateway actually does between receiving a call and returning a result, how the scope algebra stays computable, how definitions move through a lifecycle that works like code review, how runs are bounded, and how the whole thing splits across two cloud accounts so that regulated data never reaches infrastructure we operate.

It is written for architects and technical evaluators. It assumes you know what an API and an identity provider are, and it explains MCP-specific machinery where that machinery carries the argument.

1. MCP is the universal interface

Model Context Protocol is the open protocol connecting AI clients to tool servers. We did not invent it. We govern it.

The decision that shapes everything downstream is that MCP is the *only* interface. Staff AI clients speak MCP to the gateway. The runtime executing an autonomous agent speaks MCP to the same gateway. Our administrative API is dual-exposed: over HTTP for the console, and as MCP tools reached through the gateway, each operation carrying its own manifest and risk tier.

That last part is the load-bearing half. If the administrative surface were a side channel, then every claim about total audit and universal policy checks would carry an asterisk naming the one caller class that skips them. Instead, an agent drafting a definition and a human approving one go through the same door as every other tool call, and both leave the same kind of record. Approving a definition is a T3 operation, which is why no agent can hold it regardless of scope.

Three properties follow from the single-protocol choice.

Aggregation is possible. Because every downstream system is wrapped as an MCP tool server, the gateway can present one catalog assembled from many servers, apply one namespacing scheme across it, and filter it per caller. A per-integration governance story would have as many enforcement models as integrations.

Governance metadata has a home. MCP tools carry a name, a description, and input and output schemas. We add a manifest alongside each tool declaring its risk tier, side-effect class, reversibility, evidence hints, cost hints, idempotency support, and dry-run support. That manifest is what makes a tier a property of the tool rather than an assertion by the caller.

The client is irrelevant to governance. Any conforming MCP client works. The principal is the person, resolved through the customer's identity provider, and the model behind the client changes nothing about the checks. Staff keep the AI clients they already like.

2. The five planes

A plane is a design concept first and a workspace package second. The boundary rule is that planes may depend on a shared contracts package and never on each other, which is enforced mechanically rather than by convention: a dependency-graph check in continuous integration rejects a cross-plane import, and a test proves the check rejects a known-bad one.

PLANE	RUNS IN	OWNS
Gateway	Customer boundary	The single aggregate MCP endpoint: downstream aggregation, namespace assignment and collision rejection, session-to-principal resolution, scope-filtered listings, call-time enforcement, audit emission
Policy plane	SaaS for authority; enforcement replicated to the gateway	Principals, scopes and the subset comparison, risk tiers and ceilings, approval queues with evidence, autonomy graduation and demotion
Control plane	Omnafy SaaS	The registry of versioned definitions and their lifecycle, the trigger framework, the dual-exposed administrative API, tenancy, per-run metering
Execution plane	Customer boundary	The automation module: run lifecycle, transcript capture, budgets and timeouts, spawn depth limits, the kill switch
Integration plane	Customer boundary	Tool servers wrapping real systems: registration and manifest conformance, health reporting, the adapter taxonomy

One structural decision separates this from most agent platforms: **the governance core never depends on the execution plane.** Gateway plus policy plane plus audit is a complete, deployable product with no automation in it at all, sold as governed workspace access for staff AI clients. Automation is a module

layered on top. The dependency edge points one way and is enforced in continuous integration. In a governance-only deployment, the registry sits empty and the trigger framework is dormant while tenancy and metering keep governing and billing staff traffic exactly as before.

That ordering is not a packaging trick. It means the controls were built first and the agents live inside them, rather than controls being retrofitted onto an agent platform.

3. The gateway

The gateway is the most trusted component in the platform. It sees full tool-call payloads, holds the replicated policy state it enforces against, and is the emitter of record for the customer-side trail. Everything in its design is aimed at keeping the thing being trusted small: it implements two built-in tools, performs no payload transformation beyond namespace qualification and principal-context injection, makes no model calls, and forwards neither sampling nor elicitation from downstream servers. Everything else in its catalog is aggregation.

It enforces policy and never authors it. Policy authority is SaaS-side; the gateway evaluates every listing and every call locally against replicated state.

3.1 Registration and the pinned catalog

Registration is the trust decision for a tool server. Everything after it is enforcement of what registration approved.

When an operator initiates registration through the administrative API, itself an audited mutation, the gateway fetches the server's registration bundle from its manifest endpoint. The fetch is pull-based: the gateway trusts only what it retrieved itself, never a bundle pushed to it. It canonicalizes and hashes each per-tool manifest, using RFC 8785 canonical form and SHA-256 per tool, and the gateway's hashes are the authoritative ones. It assigns the namespace and binds it to the server's authenticated identity, rejecting collisions before any manifest enters the catalog or the ratification queue. It forwards the bundle and hashes to the control plane for human tier ratification, which is the only hop where anything crosses the boundary, and bundles are contractually free of regulated data for exactly that reason. When the ratified set replicates back down, the gateway pins the manifests, at which point and not before the tools become listable and callable.

The catalog is the gateway's in-memory image of everything registered, built exclusively from pinned state. Each entry carries the qualified name, the pinned manifest and its hash, the ratified tier and declared side-effect class, the server binding with its authenticated identity and mTLS material, current health and quarantine state, and any deprecation marker.

Three rules govern it. **Only pinned tools are dispatchable**, so whatever a server lists on its own endpoint, the gateway dispatches only to tools it holds a pinned, ratified manifest for. A served-but-unpinned tool is invisible and uncallable. **Health suppresses, quarantine voids**: tools on an unhealthy server are suppressed from listings while retaining ratification, and calls that reach them fail as execution failures rather than policy denials, whereas quarantined tools have ratification voided and are delisted, denied, and alarmed. **The catalog is versioned**, with a monotonic version incrementing on any registration, ratification, quarantine, health change, or sunset delisting, which drives change notifications to clients and cache invalidation.

After registration the gateway re-verifies the served surface against the pins at every connection establishment and periodically thereafter. Any drift quarantines the affected tools.

3.2 Namespacing and collisions

Every tool a client sees is qualified as `namespace.local-name`.

Namespace assignment is gateway-owned. It happens at registration, is bound to the server's authenticated identity, and treats the bundle's requested namespace as advisory only. A server never chooses its own prefix and never sees qualified names: the gateway applies the prefix on the way to clients and strips it before dispatch. Two namespaces are reserved and unassignable, one for the gateway's built-in tools and one for the administrative API's tool surface, so no downstream server can impersonate the platform's own operations.

Scopes resolve against namespaces. A selector like `intake.*` is meaningful because exactly one server identity is bound to `intake`. That binding is what makes scope grants stable references rather than string matches against whatever happens to be listed.

Collisions are rejected, never repaired, and rejection happens before any part of the offending bundle enters the catalog. A registration whose namespace is already bound to a different server identity is rejected in full, and the gateway never silently assigns a variant name, because a registration proceeding under a namespace its operator did not review is how impersonation starts. A bundle with duplicate tool names is rejected in full. A namespace, once bound, is never reassigned while any scope selector references it and is not reused after retirement, because a reused namespace would silently re-resolve existing grants against a new server. A registration requesting a reserved namespace is rejected in full.

Every rejection emits a registration audit event naming the operator, the server identity, and the rule violated.

3.3 Visibility filtering

Visibility filtering is the first enforcement point. The tool list a model operates over is computed per principal, server-side, before the response is built.

The computation runs on every listing request with three local inputs: the resolved principal, re-resolved from the bearer token on this request rather than cached from the session; the principal's effective scope evaluated from replicated policy state; and the catalog at its current version.

A catalog entry survives into a listing only if every predicate holds. It is pinned and ratified, so unratified, voided, and quarantined tools are invisible to everyone. It is servable, meaning not health-suppressed and not past a deprecation sunset. Its qualified name matches at least one selector in the principal's effective scope. Its ratified tier does not exceed the principal's tier ceiling. Its resources intersect the principal's tenant slices. And for a T3 deciding form, the principal is a human whose scope grants it and whose ceiling is T3.

Surviving entries are annotated with tier and side-effect metadata, namespace-qualified, and only then paginated. Filtering strictly precedes pagination, so no cursor boundary can leak the existence of an out-of-scope tool.

A presentation-layer filter intersects with these predicates. A tenant administrator may hide a tool or a whole namespace to trim noisy listings without touching grants. Visibility is the scope filter intersected with the hide filter, so hiding can only narrow: hiding an out-of-scope tool changes nothing, unhiding restores only what scope already allows, and a hidden tool called by name answers exactly like a nonexistent one. Hiding is presentation and never a second policy mechanism. Revoking scope remains the way to take authority away.

Graduation state is deliberately not an input. Graduation changes the posture of a call, meaning immediate execution versus a queued proposal, never the visibility of a tool. A principal sees every tool it could lawfully propose against.

The filtered listing is a pure function of effective scope and catalog version, so the gateway memoizes it keyed on exactly that pair. Invalidation is by version, not by timer. Correctness never depends on the cache: a stale key can only miss, never serve a listing computed from superseded state.

Filtering is an enforcement verdict, so it is recorded. Each listing request emits one policy-decision event carrying the principal, the scope and catalog versions evaluated, and the counts of visible and filtered entries.

3.4 Listing cost

The listing is model context, and context costs tokens on every serve. A tenant that opts in receives a minimized listing built from three mechanisms, each chosen so the minimized listing stays callable and never becomes a second visibility mechanism.

Descriptions are summarized to the first sentence, bounded at 140 characters. Descriptions dominate listing bytes and summarize losslessly, because the full text stays retrievable. Schema annotations, meaning intra-schema descriptions and examples, are stripped while every structural keyword is preserved verbatim. We rejected deferring or stubbing whole schemas: a listing whose schemas cannot validate a call is a listing that breaks calling. And a built-in tool returns one visible tool's full metadata on demand, gated by the same predicates as the filter, so a tool outside the caller's visibility answers exactly like a nonexistent one.

Minimization is presentation over the pinned catalog. It runs after visibility filtering, adds and removes nothing, changes no manifest or hash or tier, and both enforcement points are unaffected byte for byte. Against our seed catalog, whose curated descriptions are already terse, minimization measures about 13% smaller. The reduction grows with description verbosity.

3.5 Sessions and principal resolution

Transport sessions are an availability mechanism, never an authority mechanism. The MCP session id groups requests for transport purposes. Authority derives from the bearer token, and the gateway re-resolves the token to a principal on every request.

The consequences are worth being precise about. Disabling a principal in the policy plane takes effect at that principal's next call, not at their next login. A live session confers nothing once its token's principal is gone. And there is no anonymous request of any kind, including the initialize handshake and the listing call.

Two structural rules keep session grouping from becoming an authority channel. A session is immutably bound at initialize to the tenant and principal that created it, and a request presenting a different principal's token on that session id is refused with the same answer an expired session gets, so the refusal reveals nothing about whose session it was. And the resolved principal is per request, never shared: each request carries its own resolved snapshot to the handlers that read it, so two requests in flight on one session cannot observe each other's authority.

CLIENT TYPE	CREDENTIAL PRESENTED	RESOLVES TO
Staff MCP client	Short-lived access token from the deployment's auth broker, which federates to the customer identity provider over OIDC	The staff member's own human principal, never a shared or workspace principal

CLIENT TYPE	CREDENTIAL PRESENTED	RESOLVES TO
Execution-plane runtime	Run-scoped token from the in-boundary token issuer, obtained under the runtime's cloud-account deployment identity	The run's agent principal, carrying run id and definition version into every event
Service integration	OAuth 2.1 client-credentials token for a pre-registered client	The registered service principal

Resolution produces a per-request principal snapshot: principal id and type, the seed of the principal chain, effective scope, tier ceiling, tenant slices, and, for agent principals, the run id and the declared task id. Every subsequent pipeline stage and every audit event consumes this snapshot. Nothing downstream re-derives identity.

On dispatch, the gateway injects the principal context into the call metadata. The receiving server treats this as the only source of attribution and selects downstream credentials scoped to that principal. That is the adapter-side half of the confused-deputy defense. The gateway-side half is that the context is injected from the gateway's own resolution and never copied from anything the caller supplied.

3.6 The call pipeline

Every call traverses the same six stages in the same order, whatever the caller class.

Authenticate. Resolve the bearer token to a principal snapshot. Failure is rejected at the HTTP level before any tool semantics apply.

Resolve the tool. Split the qualified name, find the server binding for the namespace, and load the pinned manifest. An unknown name, an unpinned tool, a quarantined tool, and a tool outside the caller's visible set all return the same unknown-tool error. That identical answer is the anti-enumeration rule.

Policy check, the second enforcement point. Against replicated state: is the tool within the principal's effective scope, is its ratified tier within the ceiling, is the tenant slice compatible, does an active kill switch cover the calling run or definition, did an agent caller declare a known task, and what autonomy posture applies. Any input the gateway cannot evaluate produces a fail-closed denial.

Dispatch, on the allow path. Strip the namespace prefix, inject the principal context, and forward over mTLS to the bound server with a per-call timeout bounded by deployment configuration. On transport failure the gateway retries only calls that are reads or that declare an idempotency key. A keyless write is never automatically retried.

Result handling. Map server errors to the structured error surface, check output conformance where an output schema is declared, and attach the audit reference the client will receive.

Audit and return. Write the full-fidelity event. The durable write gates the response. Then return the result.

The pending-approval branch holds the call, emits a policy-decision event with an evidence reference, creates the queue item, and returns a structured pending result. On release, the gateway re-runs the policy check before dispatching the held call, so a released call whose tool manifest changed, or whose principal's grants narrowed, in the interim is not dispatched. A release is a necessary input to execution, never a bypass of enforcement.

3.7 Failure behavior

CONDITION	BEHAVIOR	CLIENT-VISIBLE OUTCOME
Policy replica unavailable or stale beyond the bound	Fail closed, audited denial	`policy_state_unavailable`, retriable
Token valid but principal unknown or disabled	Fail closed, audited denial	`principal_disabled`
Tool unknown, unpinned, invisible, or quarantined	Unknown-tool error, audited denial, identical for all four	Unknown tool
Manifest hash changed after ratification	Delist pending re-ratification, deny calls	`ratification_voided`
Kill switch engaged for the calling run or definition	Deny before dispatch	`kill_switch_active`
Agent call without a declared task	Deny before dispatch	`task_undeclared`
Downstream server unreachable or timed out	Audited as an execution failure, not a policy denial	Tool error, `downstream_unavailable`
Customer-side audit store unavailable	Suspend execution, alarm	Fail-closed error, retriable
Upward envelope channel down	No effect on calls; envelopes buffer with sequence numbers intact	None, invisible by design

The asymmetry in the last two rows is the design speaking. The customer-side trail is canonical and gates work. The upward copy is a governed export and never does. Degraded governance never degrades into open governance: the gateway has no mode in which it stops enforcing and keeps serving.

4. The policy plane

The policy plane answers four questions for every action: who is calling, what are they allowed to touch, how dangerous is the thing they are touching, and does this particular caller get to do this particular thing without a human in the loop right now.

Authority lives in the SaaS; enforcement happens in the customer boundary. The replicated set is exactly what a decision needs and nothing more: principal records and their token-validation material, scope grants in canonical form, ratified manifest hashes with their tiers, per-pair graduation state, and outstanding approval decisions. One non-decision section rides alongside, the per-tenant listing presentation state, which the gateway applies to listing text and which no enforcement decision ever reads. All of it is metadata and none of it is regulated data, so replication does not strain the residency invariant.

4.1 The scope algebra

A scope has three components: selectors over namespaced tools and resources, a tier ceiling, and tenant slices. A principal's effective scope is the union of its grants normalized to a canonical form, with selectors deduplicated and absorbed, ceilings reduced per grant, and slices flattened.

Comparison is componentwise:

$S_1 \subseteq S_2$ holds when $\text{selector-set}(S_1) \subseteq \text{selector-set}(S_2)$ and $\text{ceiling}(S_1) \leq \text{ceiling}(S_2)$ and $\text{slices}(S_1) \subseteq \text{slices}(S_2)$.

Because selectors are exact names or trailing-wildcard prefixes, selector-set inclusion reduces to string and prefix containment over canonical forms: decidable, near-linear after sorting, and exact.

That exactness is the entire design driver. The non-escalation invariant is only an invariant if the comparison is mechanical. The moment it requires human judgment or heuristic approximation, one of two failures follows. An under-approximation admits escalation holes. An over-approximation rejects legitimate drafts, which trains organizations to widen creator scopes until the check is vacuous. A grammar rich enough to express arbitrary conditions is a grammar in which subset inclusion is undecidable in general, and every practical policy language that went that way ended up shipping an approximate analyzer.

We inverted the trade. The grammar is restricted so the comparison is exact by construction: exact names and trailing-wildcard prefixes only, no negation, no mid-pattern wildcards, no predicates, and no negative grants at all so composition stays monotone. Approval routing may ride on a grant but sits

outside the algebra, because routing directs oversight rather than expanding authority; to close the soft-escalation hole, an agent-created definition inherits routing no laxer than its creator's, and only the human approver may relax it.

The standing rule for future extensions: a conditional layer may only further restrict an allowance after the subset check, so the unconditional scope remains a sound over-approximation. An extension that makes the comparison undecidable does not ship.

The canonicalizer's correctness is tested rather than formally verified. We list that as an accepted risk rather than leaving a reader to discover it.

4.2 Worked example

Two grants from a home health deployment shape, in illustrative rather than normative syntax:

```
{
  "scopes": [
    {
      "name": "intake-agent-standard",
      "tools": ["intake.*", "dbops.read_*"],
      "tierCeiling": "T2",
      "tenantSlices": ["branch-east"],
      "approvalRouting": {
        "T1": "intake-coordinator-queue",
        "T2": "intake-supervisor-queue"
      }
    },
    {
      "name": "clinician-review",
      "tools": [
        "intake.get_*",
        "intake.list_*",
        "clinical.read_*",
        "clinical.get_recommendation",
        "clinical.record_admission_decision"
      ],
      "tierCeiling": "T3",
      "tenantSlices": ["branch-east"]
    }
  ]
}
```

The intake agent can see and call every `intake.*` tool and the generic read-only database tools up to T2, so it can propose external records requests, but each proposal routes to the supervisor queue until the pair of identity and task graduates. It cannot see `clinical.record_admission_decision` at all: that

tool is outside its selectors, above its ceiling, and gated against non-human principals. Three independent denials deep.

The clinician holds broad reads, the recommendation review surface, and the deciding tool. Nothing in their scope lets them draft or approve definitions, which belongs to other roles.

4.3 Agent identity, run scope, and the cap

Each run receives a freshly minted agent principal, but every run of the same definition shares one durable agent identity in the policy plane's records. Scopes, ceilings, and graduation state attach to the durable identity. The run-scoped principal exists so attribution can name the exact execution.

The mint happens runtime-side at run admission under delegated authority, bound to exactly the effective scope the definition version was approved with, and it is reported into policy state as a recorded, audited identity mutation, so the policy plane remains the record of who exists. The bound scope persists immutably on the run record, and the gateway resolves every call of the run at the intersection of the durable identity's current scope with that cap. Narrowing the durable identity narrows every live run. The cap can never widen one. A disabled durable identity mints no new runs, checked fail-closed at both the runtime gate and the policy directory.

Agent and service principals are ceiling-capped at T2 by construction. The policy plane will not issue a non-human principal a T3 ceiling, because T3 tools do not admit non-human deciders.

5. The control plane

The control plane manages what may run: the registry and definition lifecycle, the trigger framework, the dual-exposed administrative API, tenancy, and metering. It holds no regulated data by construction rather than by filtering, because definitions, scopes, trigger bindings, tenancy configuration, and metering counters are governance metadata.

One relationship frames it: the control plane publishes and the execution plane consumes. Approved definition versions, trigger bindings, and SaaS-originated firings flow down. Redacted envelopes and metering counters flow back up.

5.1 Definitions are data

A definition is a declarative artifact stating what an automation is, what it may touch, and what it may spend. An agent definition declares goal and operating instructions, requested scope, budgets, timeouts, spawn depth limit, and model configuration. A workflow definition declares an explicit step graph whose steps invoke agents or tools.

The requested scope is embedded in the artifact itself, deliberately. Because authority is declared in the document rather than configured out of band, the non-escalation comparison can be evaluated against the artifact at rest, at draft time, at approval time, and by any later auditor of the registry, without reconstructing runtime context. A definition read out of the registry answers "what could this do" completely, on its own.

Composite chains are workflow definitions carrying a chain block: no new artifact kind, no new lifecycle, no new administrative operation. The block declares the composite's own exposure, meaning the local tool name and description and input schema the gateway lists it under once live, plus its step graph of ordered steps and single-level parallel groups. Each step names an existing namespaced tool with per-parameter wiring from the composite's input, an earlier step's output, or an authoring-time literal.

Admission is fail-closed twice over. Wiring rules are validated structurally at the registry gate, and every referenced tool must resolve to a registered, ratified tool identity, so a dangling reference is refused with a per-step error.

The governance algebra over a chain is computed rather than trusted. The composite's effective tier is the maximum of its constituents' ratified tiers, read from their pinned manifests: a floor the declared tier may sit above, never below, re-checked at approval because ratifications can change in between. Its effective scope is the union of constituent scopes, so the chain's tool identities join the non-escalation operand directly and the comparison covers what the chain actually calls even when the author under-declares, transitively through composed definitions. Incomparability fails closed, and the rejection returns a counterexample witness.

5.2 Versions and lifecycle

Every revision is a new version and the registry keeps all of them.

Drafts are the only mutable state. A version's content is writable while it is a draft and frozen at approval. Any later change, whether a prompt edit or a scope narrowing or a budget bump, is a new version that begins at draft and walks the full lifecycle. There is no in-place edit of anything approved.

Promotion deletes nothing. Approved, live, and retired versions remain with full content, which is what makes definitions diffable: a structural diff between any two versions calls out the requested-scope delta.

Revert is promotion, not rollback. No edge leaves retired. Reinstating earlier behavior means drafting a new version with the prior content and taking it through review and approval again, because a revert that skipped review would be an unreviewed change wearing an old version number.

Five behavioral rules govern the lifecycle. Any authorized principal may draft, including an agent, and drafting is a propose-class operation. Only a human may approve, and the approval operation is ratified T3. At most one version is live per definition, and activation is atomic supersession in which the target becomes live and the previously live version becomes retired in one transition, each side emitting its own event. Triggers resolve only against live versions, so a definition with no live version can be fired by nothing and a firing landing between retirement and the next activation is refused at admission. And every transition is audited, carrying the version, the from and to states, and the recorded result of the scope check.

5.3 Review like code

The registry is designed so that changing agent behavior carries the same discipline as changing source code. The correspondence is exact enough to tabulate.

CODE REVIEW HAS	THE REGISTRY HAS
A diff	A structural version diff with the requested-scope delta called out
A reviewer	A human approver exercising a T3 operation no agent can hold
Checks that gate merge	The subset check at draft and again at approval, failing closed with a counterexample witness
Review comments	A reject operation with a structured reason code; the version stays a draft and the record shows why
History and blame	Every version retained; every transition an event carrying the full principal chain
Revert via a new commit	Revert via a new version through the full lifecycle

The analogy breaks in one deliberate place. In code review, a sufficiently trusted author can often self-merge. In the registry, no drafting history, no graduation, and no ceiling configuration ever lets the drafter approve their own definition. Graduation can remove the queue stop on the draft call. It never removes the lifecycle gate on the definition.

5.4 Triggers

A trigger binds an event source to a definition, and a firing causes a run. Four source kinds exist and the set is closed until a design change reopens it: cron schedules, system events on a message-bus topic with a filter, inbound documents and email on a watched ingestion channel, and manual invocation by an authorized principal.

Manual invocation is not a path around governance. It is an administrative operation governed like every other tool, and because starting a run expands the set of things that run, it carries expand-side tiering.

A binding names a definition, never a version, and each firing resolves the binding against the currently live version at admission time. Two properties fall out for free: activating a new version redirects every existing binding atomically with the supersession, and retiring the live version silences every binding until a successor goes live. Fail closed, never fail-frozen on stale behavior.

Beyond its source specification, a binding carries an input mapping and an overlap policy. For document and email sources the input mapping produces a reference to the customer-side document, never its content, because the payload has no reason to exist control-plane-side. The overlap policy for recurring sources is either skip a firing while a prior run is still in flight, or queue it behind that run. Skip is the default for cron, because a schedule that stacks runs behind a slow predecessor converts one incident into a backlog of them.

Budgets and timeouts are deliberately not on the binding. They bind at admission from the definition version, so an operator cannot quietly loosen a reviewed budget by editing a trigger.

A firing is evaluated on whichever side of the boundary its source lives, because moving the event would mean moving its payload. Cron and manual firings are SaaS-fired and carry no customer payload. Document, email, and system-event firings are evaluated customer-side.

6. The execution plane

The runtime turns an approved definition into a run. It is the automation module, and three consequences of that framing shape every mechanism.

The runtime is a caller, not an authority. It consumes the same published contracts as everyone else, authenticates as a principal, and reaches every tool through the gateway's two enforcement points. A run's tool call is policy-checked and audited exactly like a staff member's, and nothing in this plane can allow what the policy plane would deny.

The core never depends on the runtime. The governance core sees an agent run only as a principal making tool calls. Run identity, transcripts, budgets, and the kill switch are runtime concerns invisible to the core.

The runtime is not an audit emitter. The audit trail of a run is the gateway's record of it, carrying the run id and definition version. The transcript adds the model-side detail no audit event carries. The runtime writes records, not audit events.

6.1 Why it lives in the customer’s account

The runtime is the most payload-saturated component in the platform. A transcript contains every prompt, every completion, every retrieved record, and every tool argument and result at full fidelity. For a healthcare deployment that is regulated data end to end. The residency invariant therefore admits only one placement: inside the customer's own cloud account, delivered as versioned container images plus an infrastructure construct rather than source.

The same placement makes the model rule straightforward to honor. Inference runs against a managed model service in the customer's account, or against a model API under the customer's own agreement, using customer-held credentials the runtime reads from the customer stack's secret store. We hold no model credentials on this path and have no operator access to the running stack.

The inverse property also holds and a security review will check it: there is no SaaS-side execution path. If the customer stack is down, our side alarms on missing health telemetry and can do nothing to customer systems. The control plane holds definitions as data, and only the runtime, inside the customer boundary, ever interprets one.

6.2 The runtime over an agent SDK

The runtime does not implement an agentic loop from scratch. An agent SDK supplies the loop, meaning model invocation, tool-use plumbing, message management, and sub-agent spawning. The runtime wraps that SDK with the platform's obligations: run identity, MCP-only tool access through the gateway, transcript capture, budget and timeout accounting, the spawn depth limit, parking, and kill-switch responsiveness. The reference realization is one supervising scheduler service plus one container task per run.

The SDK is an implementation dependency, not a governance boundary, and the design assumes it is imperfect. Nothing the SDK does can widen a run's authority. The run task holds no downstream credentials, no SaaS credentials, and no model credentials beyond the configured path. Adapter security groups admit traffic only from the gateway, so a tool call that does not cross the gateway has nowhere to land. And egress from the customer stack is allow-listed at the network layer. A bug in the loop can waste budget, which the runtime bounds. It cannot mint authority.

6.3 Admission

A run is created queued when a trigger fires or an authorized principal invokes a live version manually. Admission is where everything the run will be governed by gets bound, in four steps.

The version check is fail-closed: the definition version must be live in the replicated registry state, and a non-live version, an unknown definition, or replicated state past its staleness bound refuses admission with a distinguishing reason.

The agent principal is minted under delegated authority, scoped to exactly the effective scope the definition version was approved with, with the cap described in section 4.3.

The run-scoped token is obtained by authenticating with the runtime's cloud-account deployment identity to the in-boundary token issuer. The gateway resolves the run task's session to that principal from this token and nothing else. Tokens renew for long runs and stop being renewable the moment the run reaches a terminal state, so a leaked run token dies with its run.

Budgets, timeouts, and the spawn depth limit are read from the definition version and fixed for the life of the run. They are not adjustable mid-run by anyone, including the agent. Raising a budget means a new definition version through the full lifecycle.

6.4 Budgets, timeouts, and spawn depth

Every run is bounded on three axes, all declared in the version and fixed at admission.

AXIS	BOUNDS	ON EXHAUSTION
Monetary budget	Model spend plus tool-call cost, using manifest cost hints as the estimate and metering as the actual	Failed, reason `budget-exhausted`
Tool-call budget	Total tool calls dispatched by the run, sub-agents included	Failed, reason `budget-exhausted`
Wall-clock timeout	Active execution time	Failed, reason `timeout`

Checks are pre-flight: a step whose estimated cost the remaining budget cannot cover is not started. The runtime never severs an in-flight tool call for budget reasons. A dispatched write completes and is recorded, and the run ends before the next step. A budget mechanism that truncates writes mid-flight converts a cost control into a data-integrity hazard.

The wall-clock timeout is an active-execution clock. Time spent parked awaiting approval does not count against it, because parked time is bounded by the approval item's own expiry rules. A slow queue must surface as a queue problem rather than silently consuming run budget and manufacturing pressure to rubber-stamp.

Sub-agent spawning is governed by three rules. A sub-agent is not a new principal: same run id, same agent principal, same effective scope, same ceiling, so delegation inside a run can never escalate. Sub-agent spend draws from the parent run's single budget pool, so fan-out is bounded by budget even

where depth is not the binding constraint. And depth is limited by a per-version maximum capped by a platform ceiling, where depth zero means no spawning and a spawn attempt at the limit fails as an in-run error the agent observes rather than a run terminal.

The depth limit exists because recursive delegation is the runtime's runaway mode. Unbounded spawn trees are how a single misbehaving run converts one budget into an unreviewable cascade. Depth bounds the structure, budget bounds the total, and the transcript records every spawn so a cascade is fully reconstructible.

6.5 Transcripts

The transcript is an append-only journal written to the customer-side store as the run progresses rather than assembled at the end, so a run that dies mid-flight still has a complete record up to its last step. It records every message, every model interaction with prompt and completion and token usage, every proposed and dispatched tool call with full arguments and results, every policy outcome the run observed, every sub-agent spawn, and every runtime intervention. At terminal state the journal is sealed: the transcript reference is fixed and no writer path to that object remains.

It never leaves the customer's account, in whole or in part. A reader who needs to know what the model saw pivots from an audit event's run id to the transcript inside the customer boundary. The audit trail answers what was done, by whom, under whose authority.

6.6 Parking and crash recovery

When a proposed call is held for approval, the runtime parks the run rather than holding compute open against a human's response time. The pending-approval outcome reaches the runtime as a structured result, the runtime appends the hold and its correlation id to the transcript, confirms the journal is durable, and signals the scheduler, which stops the task and marks the run awaiting approval. The transcript is the checkpoint, so parking adds no second persistence mechanism that could disagree with it.

When the decision replicates down, the scheduler relaunches the task, which replays the journal and resumes. On release the gateway dispatches the held call after re-checking policy at dispatch. On rejection or expiry the held call resolves as an audited denial, and the denial is data: the run resumes and may adapt, because a rejected action fails the action rather than necessarily the run. A rejection with a structured reason code is frequently the useful signal.

A run task that exits without reaching a terminal state is detected by the scheduler and relaunched a bounded number of times. Recovery replays the sealed prefix of the transcript and resumes from the last completed step. A tool call dispatched but unresolved at the crash is not blindly re-dispatched. The

reference implementation is deliberately conservative and surfaces every unresolved dispatch as an unknown-outcome error, because a wrongly classified re-drive is a duplicate action while a conservatively surfaced unknown costs the agent one verification read.

6.7 The kill switch

The kill switch halts one run, or every in-flight run of a definition. It is exercised by an authorized principal through the administrative API, which in MCP form is a governed tool call like any other, and lands as an administrative mutation event plus the correlated gateway record of its invocation.

Semantics are immediate cessation rather than graceful shutdown. A tool call already dispatched downstream may complete in the wrapped system: a sent fax is sent, and the kill switch stops the agent rather than unwinding the world. The unresolved call is recorded as such, and the reversibility metadata in the tool's manifest is what tells the operator whether cleanup is possible. The run moves to halted with the halting principal recorded, which is the one terminal state that names a human. A pending approval item belonging to a halted run is not edited, and if an approver later releases it the release dispatches nothing, because the gateway re-checks at dispatch. The transcript is sealed exactly as for any other terminal, which is usually why the run was halted.

The kill switch changes run state, never definition state. Stopping future runs is a lifecycle act: retire the version or disable its trigger, through the same administrative API. The operational response to a misbehaving definition is halt-all plus retire, two audited acts by an accountable principal.

6.8 Tool results are untrusted content

A tool result may contain text that reads as instructions. The runtime draws a hard line between the loop and itself: nothing in a tool result, however phrased, is interpreted as a directive to the runtime. Budgets, depth limits, parking, and halts are controlled exclusively by the definition version, the scheduler, and the administrative API.

A steered agent remains contained by the governance core, because tier gates, the propose-then-approve posture, and T3 permanence bound what any sequence of model decisions can do. The runtime deliberately claims no prompt-hygiene defense of its own. The containment is structural.

7. The integration plane

The integration plane is the tool servers that wrap real systems. It is the only plane whose components are routinely written by parties other than us, which is why it is governed by a published contract rather than by shared code.

7.1 The tool-server contract

A tool server implements three things: registration, meaning manifest submission; health reporting; and the MCP tool surface itself. Tool servers sit behind the gateway and are never called directly by clients, enforced at the network layer by security groups that admit traffic only from the gateway.

The per-tool manifest is the governance surface. It declares name, description, and input and output schemas; the risk tier; the side-effect class, which is one of read, internal-write, or external-action; a reversibility flag; evidence hints saying what an approver should be shown for calls to this tool; cost hints; idempotency support; and optional dry-run support.

Side-effect class and risk tier are declared separately on purpose. The class is a mechanical description of what the tool does. The tier is a ratified risk judgment. They usually align, with read mapping to T0, internal-write to T1, and external-action to T2, but ratification can rate a tool above its class's usual tier and never below it, and T3 exists only as a tier, because judgment risk is not a mechanical side effect.

The conformance rules an adapter engineer needs are stated as rules rather than discovered by trial: manifests are hash-pinned at registration, any schema or description change forces re-approval, namespaces are gateway-owned, and registration-time collision rejection means one engineer's adapter cannot be shadowed by someone else's or blamed for drift it did not sign. Tier ratification is a human platform-side act, so the adapter team is not the last line of defense. The adapter declares; the gateway and policy plane enforce.

7.2 Adapter taxonomy

CLASS	WRAPS	TYPICAL SIDE-EFFECT CLASSES	DEFAULT TIER CEILING	STATUS
Data-read	Databases, reporting surfaces, record lookups	read	T0	Current
System-write	Operational systems accepting reversible writes	read plus internal-write	T1; irreversible writes ratified T2	Current
Communications	Email, SMS, fax	external-action	T2	Designed, not shipped
Browser automation	Systems with no API, through their web interfaces	external-action	T2 plus a required verification step at registration	Designed, not shipped

Defaults guide declaration by authors and review by ratifiers. They never replace per-tool ratification: every tool's tier is ratified individually, and ratification may raise a declared tier, never lower it.

There is no T3 adapter class, because T3 is a judgment property rather than a side-effect property.

7.3 The ownership boundary

Generic adapters are ours and live in our repositories. Customer-proprietary adapters are the customer's intellectual property, live in the customer's own repositories, are built into the customer's own images, and never enter our organization. Customer engineers build them against our published SDK and contracts packages, and add them to the deployment construct's adapter set alongside the ones we deliver.

That line is a decision of record, not a preference. It is also the answer to a real objection from customer engineering teams, which is pressure to contribute proprietary system knowledge into a vendor's codebase.

8. Split deployment

One sentence carries the topology: everything that sees a raw payload deploys into the customer's cloud account, everything we operate sees only redacted metadata, and the seam between them is small, enumerable, and enforced on the customer side.

8.1 Two stacks

The **customer stack** is the gateway, the runtime, every adapter, the stores holding transcripts and full-fidelity audit events and approval evidence, the redaction pipeline, and the model path. It deploys into the customer's own account, in a region the customer chooses.

The **SaaS stack** is the multi-tenant governance core: the control plane, the policy plane's authoritative state, the console, and the redacted audit store.

Every boundary connection is initiated from the customer side. Downward state is pulled by the customer stack from our replication endpoint over mutually authenticated, tenant-scoped HTTPS with long polling. Upward events are pushed by the customer stack to our ingestion endpoint, at least once, with idempotent event ids and customer-side buffering. We hold no credentials, no network path, and no cross-account role reaching into the customer account.

Replicated state is versioned and integrity-checked, and the gateway enforces a staleness validity bound: state older than the bound is treated as unavailable and the gateway fails closed. The replication protocol is versioned so a SaaS deploy never strands an older customer stack, and we serve the current protocol version and its predecessor.

8.2 Artifact delivery

The customer stack is delivered as versioned artifacts, not source.

Every customer-side service is published as an immutable, version-tagged container image in our private registry. Customer accounts are granted cross-account read-only pull on that registry. This is the single cross-account grant in the whole topology, it points at our account rather than into the customer's, and every pull is initiated by the customer side.

The deployable unit is a published, versioned infrastructure construct that the customer's engineers instantiate inside their own infrastructure application. The construct pins every image by digest, so a release is a construct version plus its digest set and what was reviewed is what deploys. It takes the account-specific configuration as parameters: identity provider integration, model path, region, network placement, and the set of adapters to deploy.

Deploys, upgrades, and rollbacks run in the customer's own pipeline. An upgrade is a construct version bump and a redeploy; a rollback is the previous version of the same construct. We have no deploy access and no operator access to the running stack. Operability comes from health telemetry, and a customer can sever even that channel at the cost of our ability to support the deployment.

The SaaS side is delivered differently on purpose. It is ours to operate and is continuously deployed, with images built and pushed by the deploying workflow immediately before the deploy that consumes them. Each side therefore has exactly one producing workflow, which makes "the tag a deploy consumes is built by a workflow" a property a test can check rather than a convention. A customer release manifest carries no SaaS digest, because a SaaS digest is unusable by a customer stack and would imply the SaaS ships on the customer's upgrade cadence.

8.3 Reference architecture

The specific service choices below are the current implementation direction. The normative obligations are the invariants, the crossing rules, and the published contracts.

The **SaaS stack** runs the console as a static application behind a CDN, the administrative API and control-plane and policy-plane services as container services behind a load balancer, registry and policy and queue and metering state in a key-value store with the tenant id in every partition key under per-tenant encryption contexts, the redacted audit store as object storage with compliance-mode object lock plus an index for customer-facing trail queries, and an ingestion endpoint terminating the customer event channel into a queue. The audit write path is owned by a dedicated ingestion role that no read-path service assumes.

The **customer stack** runs in private subnets. The gateway is a container service behind the only load balancer staff clients can reach. The runtime is per-run container tasks supervised by a small scheduler. Adapters are one container service each, with security groups admitting traffic only from the gateway. Transcripts, audit events, and evidence live in object storage with compliance-mode object lock under customer-managed keys. Secrets live in a managed secret store. The redaction pipeline is a dedicated service and the sole holder of ingestion credentials. And egress leaves through a single controlled path.

8.4 Egress control

The load balancer is the single ingress and the egress allow-list is the single way out. The allow-list contains our SaaS endpoints, the registered adapters' vendor API hosts, and, on that model path only, the model API host. Because the list is derived from the set of registered adapters, adding an adapter is what widens egress: an audited registration act, not a network change made on the side.

How the derived list is enforced is a deployment parameter with two elections that are not equivalent.

The default is a network firewall enforcing the list by hostname over TLS SNI and HTTP host, with drop-established as the default action. An adapter reaching an undeclared host is dropped at the network layer whatever its container believes. This is the posture assumed everywhere else in this paper and required for any deployment holding regulated data.

The alternative is a security-group election, a cost measure for development and quality-assurance deployments only. A managed network firewall costs roughly $300 a month to run continuously, which is indefensible for a deployment whose purpose is to exercise the platform. Under this election, egress is narrowed to HTTPS on port 443 and nothing else, and the derived hosts are recorded in a stack output, but a security group matches on address and port and cannot match on hostname, so the allow-list is not enforced. The network-layer backstop against a mutated tool server is absent and only the in-boundary controls remain.

The consequence is stated in two places an operator cannot miss: a warning at synthesis time, and a stack output reading `security-group (NOT COMPLIANT, hosts unenforced)` that outlives it. A development profile that merely runs quieter than production is a development profile somebody eventually ships.

The load balancer's network exposure is also a customer choice, because staff clients and approver browsers must reach it from wherever the customer's people work. Internet-facing, with TLS-only listeners, a web application firewall, and optional source-address restriction, is the posture for staff on unmanaged networks. Internal, reachable only over the customer's private connectivity, is the posture for customers whose access paths are already private. In either posture the gateway serves no unauthenticated request beyond the challenge that bootstraps the OAuth handshake.

8.5 Failure and skew at the seam

When the SaaS is unreachable from the customer stack, tool traffic continues against replicated policy state until the staleness bound expires, after which the gateway fails closed. Upward events buffer and redeliver. Pending approvals cannot be decided while the queue is unreachable, and expiry semantics are evaluated when connectivity returns, where expiry is a denial and never an approval.

When the customer stack is down, we notice missing health telemetry and alarm. Nothing in the SaaS can or does act on customer systems in the stack's absence, because there is no SaaS-side execution path.

Version skew follows the protocol's support window: the SaaS may run ahead of a customer stack by one protocol version, and a customer stack older than that stops replicating, hits its staleness bound, and fails closed rather than misinterpreting newer state. That is a security property as much as a compatibility rule. It is the structural backstop for the vulnerability-management split in the shared-responsibility matrix, because a customer stack that is never upgraded does not run stale software ungoverned indefinitely. It eventually falls out of the supported range and stops serving.

9. Engineering conventions

A few conventions are load-bearing enough that an evaluator should know them.

One language and one infrastructure toolchain across the platform. Both stacks are TypeScript on the same infrastructure-as-code framework, with one compiler lineage across every workspace package, one lint configuration, one formatter, and one test runner, all configured once at the repository root. Packages never carry their own copies.

The interface-only dependency rule is mechanically enforced. Workspace packages interact only through published entry points, never by reaching into each other's source, and the dependency direction is fixed: planes may depend on the shared contracts package, adapters on the SDK and contracts, and contracts on nothing. A dependency-graph check runs in continuous integration, and a test proves the check rejects a known-bad import.

Merges are performed by humans. All implementation on the platform is agent-delivered: every change arrives as a pull request and a human reviews and merges it. Agents never merge and never push to the main branch. That discipline is the same one the definition registry applies to agent behavior, applied to the platform's own source.

Decisions live in decision records. Architecture documents describe the resulting design and link the record. They do not restate its argument and they never quietly contradict it.

10. Implementation status

The governance core is built: the gateway with aggregation, namespacing, pinning, visibility filtering, and the call pipeline; the policy plane with scopes, tiers, the subset comparison, approval queues, and graduation state; the control plane with the registry, lifecycle, triggers, the dual-exposed administrative API, tenancy, and metering; the runtime with run lifecycle, transcripts, budgets, spawn depth limits, and the kill switch; the redaction pipeline; and the audit pipeline across both stores. Composite chains are implemented as definitions with generated, pinned exposures.

The automation module has an end-to-end path under test that drives a scheduled firing across the boundary channel into a real run whose tool calls reach a real adapter through the real gateway, parks it on a hold, releases it through an approver's decision, and asserts the resulting record set on both sides of the boundary.

Designed and not shipped: communications adapters, the browser-automation tier, cross-tenant isolation tests in continuous integration, automated caller-behavior anomaly detection, a per-tenant rate and spend governor, and a multi-region posture for the SaaS stack. Nothing in a later phase is permitted to weaken an invariant established in an earlier one, and the threat model is re-reviewed at each phase boundary.

Further reading

Governing AI access to systems of record is the security and governance companion to this paper: the five invariants, the threat model, the audit record, and our published accepted-risk list. *Earned autonomy* covers risk tiers, approval queues, and graduation from the operations side. *HIPAA and the AI governance layer* covers the compliance posture.

Customers and prospective customers can request the internal contract set: the tool-server contract, the gateway API, the administrative API, the policy schema, and the audit events contract, along with the decision log.