DOME SYSTEMS

So You're Building an Enterprise Agent Hub

A pragmatic, technical guide to the responsibilities of an internal agent platform — and how to build each of them on Dome's APIs.

Introduction 3

What a platform must do 4

A shared-responsibility model 4

How Dome provides the governance substrate 5

Balancing responsibilities between runtime and governance 7

The two call sequences 7

Identify 9

Agent lifecycle 9

Ensuring correct identity for agents 10

Authorize 12

Rules you write versus grants the platform generates 12

The two shapes of policy: from declaration, and from identity 13

Where evaluation happens 14

Route models 16

Broker tools 18

Governing what comes back: Guards 19

Account: the audit payoff 20

What a governed action records 20

An activity chain, end to end 21

The operational surface 22

Two patterns 23

Pattern 1 — The Self-Service Agent Platform 23

Pattern 2 — The Purpose-Built Agent Service 24

Choosing between the patterns 25

Integration and rollout 26

Common failure modes 27

Conclusion 28

Introduction

Enterprises are adopting AI agents faster than the infrastructure to govern them. Agents now appear across business units — built on different frameworks, running in different environments, reaching into production systems — and platform teams are increasingly asked to provide a single, governed way to build and operate them: an internal agent platform, or hub.

This paper is for the platform engineer or architect building that hub. It sets out the responsibilities such a platform must discharge, identifies which you should build yourself and which you should delegate to a governance substrate, and shows how to implement each on Dome's APIs. It assumes you have your own view on runtime and infrastructure and does not prescribe one. It is a design guide, not a tutorial.

> **How to read this.** If you are deciding what to build, read the first three chapters — the taxonomy and the boundaries. If you are working out how, the middle chapters take each governance responsibility in turn and drop into the actual APIs, CLI, and policy you would write. The last chapters cover the two platform patterns we see most often, how to choose an integration, and the common failure modes. Throughout, "Dome" is the governance substrate; "the platform" is the system you build on top of it.

One principle underpins everything that follows: **an agent platform is two systems.** The first is a runtime and a control surface — the system that lets a team declare an agent, packages it, runs it, and lets operators steer it. You should build that, to your own taste, because it is where your infrastructure and organizational choices matter. The second is a governance substrate — identity, authorization, model access, a governed path to tools, and an accountable record. You should not build that, for the same reason you do not build your own identity provider: the cost of a subtle mistake is high and is paid elsewhere.

The engineering that matters is the boundary between the two: deciding which responsibilities sit on each side, and wiring the small number of calls that join them. The rest of this paper maps that boundary.

What a platform must do

An agent platform, whatever its shape, discharges a fixed set of responsibilities. Naming them precisely lets you sort them into two groups: the responsibilities where a bespoke answer adds value, and those where a bespoke answer is a liability.

An agent, throughout, is three things: **code** that orchestrates, a **model** that reasons, and **tools** it calls to act on the world. A chatbot is thin code, one model, and (increasingly) a few tools; a revenue-analysis agent is heavier orchestration, a model, and a set of tools into real systems. The proportions vary; the components do not. A platform stands up all three, repeatedly, for many teams, without a person wiring each one by hand.

A shared-responsibility model

The responsibilities divide into two groups — much like the shared-responsibility model that governs cloud adoption. One group defines your platform; the other is provided by the substrate.

The platform surface — what you build:

Responsibility	What it means
Declare	A team describes the agent it wants: model, instructions, permitted tools, and who it acts for. You design this schema and the surface that captures it — YAML, a form, an API.
Manufacture	The declaration becomes something runnable — a container image, a function bundle, a job spec.
Run	The runnable artifact gets somewhere to execute and a way to be invoked — Kubernetes, ECS, Cloud Run, serverless; your choice.
Operate	Operators see the estate and drive lifecycle: pause, resume, retire. You decide how it surfaces.

The governance substrate — what Dome provides:

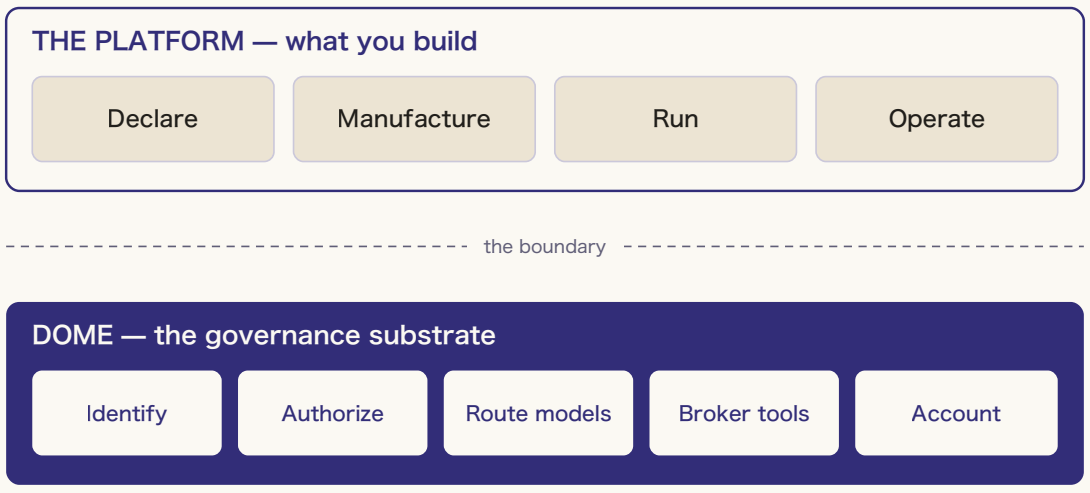
Responsibility	What it means
Identify	Every agent has an identity the system can reason about, and acts on behalf of a user or service principal.
Authorize	Every action an agent attempts is checked against policy before it happens — deterministically, fail-closed.
Route models	Agents reach models through a governed path: no raw provider keys, model choice expressed as policy.
Broker tools	Agents reach tools through a governed path: no raw tool credentials, discovery filtered by policy, responses governed on return.
Account	Every governed action is recorded with enough context to answer "who, what, under which rule, with what result," and streamed to where security already works.

The boundary between these two tables is the most important decision in the paper. The first table is where your judgment adds value. The second is where a home-grown answer eventually fails an audit, leaks a credential, or becomes an authorization system nobody reviewed. Consume it; do not rebuild it.

How Dome provides the governance substrate

Each governance responsibility corresponds to a specific Dome component and a specific way to reach it. This table is the spine of the chapters that follow; each takes a row and carries it through to working calls.

Responsibility	Dome component	Primary surface	Where you reach it
Identify	Registry, Identity	`AgentRegistry`, `Identity` (Connect RPC); `dome agents` (CLI)	Build time: register, issue keys. Runtime: agents exchange keys for short-lived tokens.
Authorize	Authorization (Cedar engine)	`Authorization` (Connect RPC); `dome rules` (CLI)	Build time: deploy rule bundles. Runtime: evaluated at the gateway or in-process.
Route models	LLM gateway	Gateway `/gateways/{id}/v1`; `dome model` (CLI)	Point a standard model client at the gateway; declare pools and routing.
Broker tools	Gateway, Guards	Gateway `/gateways/{id}/mcp`; `dome tool`, `dome guards` (CLI)	Agents call tools through the gateway; assign response filters per connection.
Account	Audit	`Audit` (Connect RPC), `/api/v1/audit/*` (REST); `dome audit` (CLI)	Query and stream the record; export to your SIEM.



An agent platform is two systems. Above the boundary, the runtime and control surface you build. Below it, the five governance responsibilities you consume from Dome.

Everything from here is detail on how to build the first table's rows well, and how to consume the second table's rows without accidentally rebuilding them.

Balancing responsibilities between runtime and governance

Where your platform ends and the substrate begins is the most consequential decision you will make. Draw the line too high and you reimplement authorization yourself; too low and you couple your runtime so tightly to the substrate that teams cannot ship.

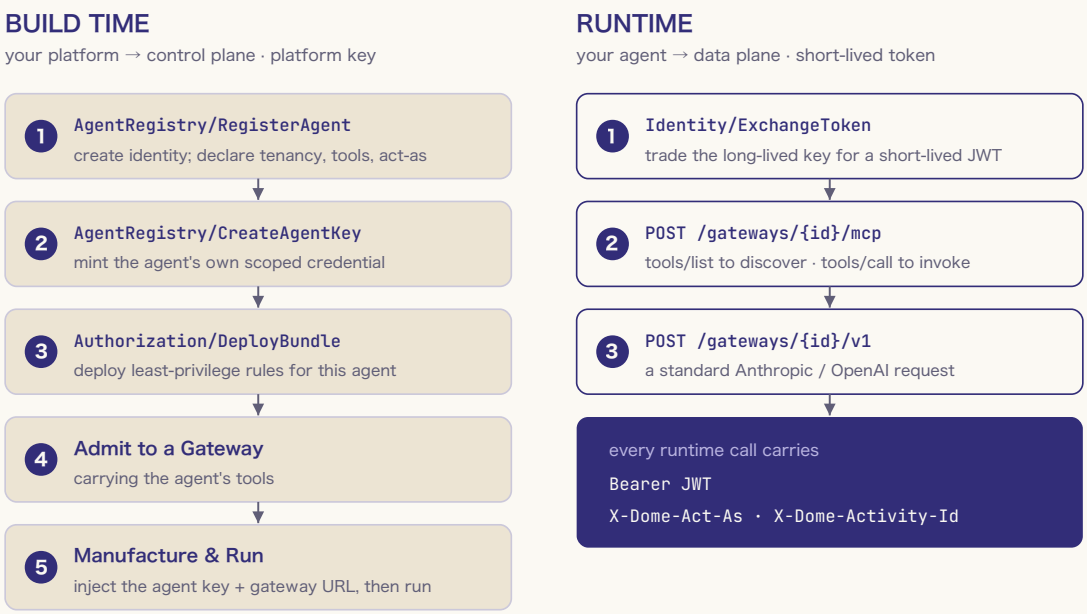
Dome draws this boundary as a split between a **control plane** and a **data plane**, always separate processes. The control plane is the system of record: the registry of agents, tools, models, and pools; identity and token issuance; the stored, versioned authorization rules; and the audit record. The data plane sits on the hot path of an agent's execution and enforces decisions. Your platform talks to both, for different reasons and with different credentials, and keeping those two conversations distinct is the core of a clean design.

- **Your platform → control plane** is a build-time, administrative conversation. When a team declares an agent, your platform registers it, mints its credential, and deploys its policy. Authenticate this with a platform-level key that lives in your platform, never in an agent.
- **Your agents → data plane** is the runtime conversation. The agent authenticates with its own scoped, short-lived credential and makes governed calls. It never sees an admin key, and it never holds a credential for any tool or model.

You attach agents to the data plane in one of two ways, and the choice determines where enforcement runs. Route traffic **through the gateway** and the agent stays thin: it makes an HTTP call to a governed endpoint, enforcement happens out-of-process, and the agent holds no tool or model credentials. This is the default for anything that touches a real tool or model. Or embed the **SDK** and enforcement runs in-process: the SDK syncs the agent's effective policy from the control plane and evaluates it locally, so an agent can check its own action before it acts. Use the gateway for egress and the SDK where a local pre-check is useful; the choice need not be uniform across the estate.

The two call sequences

Almost everything your platform does reduces to two sequences. One runs once, when a team declares an agent — the manufacturing step, made with the platform key against the control plane. The other runs on every invocation — the hot path, made with the agent's own short-lived credential against the data plane. The figure lays out both; the rest of the middle of this paper is what happens inside each numbered step.



The two sequences, and the credential each uses. Build time runs once per agent against the control plane; runtime runs on every invocation against the data plane, every call carrying the agent's token and correlation headers.

Identify

Nothing else works until every agent has an identity the system can reason about. Deterministic-era identity tooling was built for humans and services, and agents differ in two ways: there are many of them, created and destroyed faster than most onboarding processes expect; and they act on behalf of a user or service principal, so "who is doing this" has two answers at once — the agent, and the identity it acts for.

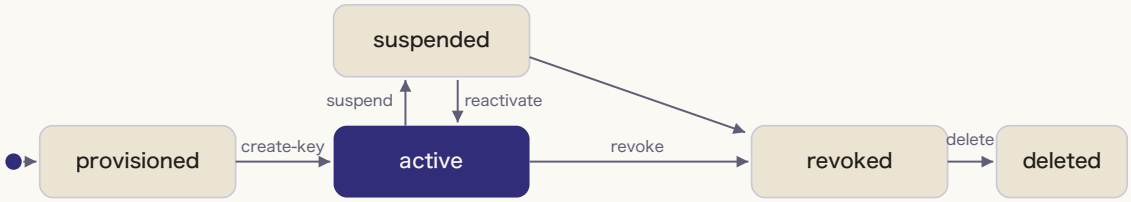
In Dome an agent is a first-class record in the **Registry**, created through `AgentRegistry/RegisterAgent`. Registration is where anonymous code becomes governable: the agent gets a stable identity, its permitted tools and pools, and a place in a four-level tenancy hierarchy — **Organization → Tenant → Workspace → Agent** — enforced at the database with row-level security. Map your organization onto this hierarchy deliberately before you register your first agent; it is how you keep one business unit's agents isolated from another's.

Registration and key issuance are the platform's job: `RegisterAgent`, then `CreateAgentKey`. The key is a long-lived credential the agent exchanges at runtime for a short-lived JWT via `Identity/ExchangeToken` (or the REST OAuth `client_credentials` endpoint). The token carries the agent's identity and tenancy — its `agent_id` and the organization, tenant, and workspace it belongs to — and it expires in minutes, so a leaked token is a contained problem. Because the token is short-lived, a change to an agent's profile — a revoked permission, a suspension — takes effect on its next exchange without a key rotation.

Agent lifecycle

An agent moves through a defined lifecycle, and each transition is an explicit, audited operation. Revoke cascades to descendants by default, so a revoked parent cannot leave live children behind.

Operation	CLI	Effect
Register	`dome agents register`	Create the agent record; state becomes `provisioned`.
Issue key	`dome agents create-key`	Mint a credential; the agent can now exchange tokens; state becomes `active`.
Rotate key	`dome agents rotate-key`	Replace the credential without changing identity.
Suspend / reactivate	`dome agents suspend` / `reactivate`	Temporarily disable, then restore, token exchange.
Revoke	`dome agents revoke`	Permanently disable the agent and cascade to descendants.
Delete	`dome agents delete`	Remove a revoked agent record.



The agent lifecycle. Each transition is an explicit, audited operation; revoke cascades to descendants before an agent can be deleted.

Ensuring correct identity for agents

An agent should act on behalf of an identity, not as an anonymous service. Dome expresses this with act-as: the agent authenticates with its own credential, and every governed call carries an `X-Dome-Act-As` assertion naming the user or service principal it is acting for. Authorization and audit then reason about the real actor, not the agent in isolation.

The acting identity is not a separate entity type; it is the `act_as` attribute on the agent principal — subject, email, roles, and groups — verified against the OIDC providers you register for the workspace. Because authorization reads this attribute, the same agent calling the same tool on the same record can be permitted for one person and denied for another. The audit trail names the human, and policy can express constraints — conflicts of interest, need-to-know, per-customer boundaries — that are meaningless at the level of a bare service account.

Prefer act-as for essentially every agent. A user-facing agent acts as the invoking user; a background or scheduled agent acts as a designated service principal so its actions remain attributable. An agent that acts purely as itself is a rare exception, appropriate only when there is genuinely no user or service principal behind the work.

Register an agent with its tools and act-as configuration, then mint a runtime key:

```
dome agents register --name revenue-analyzer --tool crm/list_accounts --tool crm/
get_account --actas-method oidc --actas-provider corp-idp --actas-required
dome agents create-key revenue-analyzer --name runtime
```

You declare the agent's tools and pools; registration wires up what it needs to reach them, so you never hand-manage low-level permissions.

> **Guideline.** Default to act-as. The agent authenticates as itself but carries the identity of the user or service principal it acts for, so authorization can vary by who that is and the audit trail names the real actor. An agent that acts as no one is the exception, not the starting point.

Authorize

Authorization is the center of the system. Identity, model routing, and the tool gateway all exist to make one decision possible and to enforce its result. The question is narrow, and at runtime it is the only one that matters: this agent, acting as this identity, is about to take this action against this resource — is that allowed?

Dome answers it with **Cedar**, an analyzable policy language built for authorization. Cedar is the engine, not the product: treat "authorization" as the capability and Cedar as the language it speaks, as you treat persistence with Postgres underneath. Two properties make it the right fit. It is deliberately not general-purpose — small, deterministic, and analyzable — which is what you want where "it depends what the model decided" is unacceptable. And it is fail-closed: if the decision cannot be made, the action does not happen.

The entity model is small. Getting the names exactly right matters, because you will write them:

Concept	In Cedar	How you express it
The agent	`Dome::Agent`	The principal. Carries an `act_as` record — the identity it acts for.
The action	`Dome::Action::"mcp:call"`, etc.	String-typed. Tools: `mcp:call`, `mcp:discover`. Models: `llm:invoke`, `llm:embed`, and siblings.
A tool	`Dome::MCPTool`	The resource for a tool call.
A model	`Dome::LLMModel`	The resource for a model call. Reference a whole pool with the `resource.pool` attribute.
Any other resource	`Dome::Resource`	The general resource type; connection attributes surface as `resource.<key>`.

Everything you need is expressed with these five types plus attributes. The user an agent acts for is the `act_as` attribute on the principal; a model pool is `resource.pool`; a Gateway is `resource.gateways`. So a rule permitting a whole pool reads `resource.pool == "premium"`, and a rule that varies by the acting user reads `principal.act_as.roles.contains("...")`.

Rules you write versus grants the platform generates

One distinction clarifies most of the design: a **rule** versus a **grant**.

A **rule** is Cedar you author. You deploy it as a versioned bundle through `Authorization/DeployBundle` (or `dome rules apply`, with `validate` and `simulate` to check it first) at one of

four scopes: org, tenant, workspace, or agent. Bundles merge into one policy set, and the merge has one rule worth memorizing: **forbid always wins**, across every scope. A workspace-level `forbid` cannot be undone by an agent-level `permit`. This is how a security team sets a floor no team beneath them can dig under.

A **grant** is Cedar your platform generates from a declaration, and it is the default for a self-service platform. When a team declares an agent that may use three tools, the platform writes no policy by hand; it compiles that tool list into a least-privilege grant — a `permit` for `mcp:call` on exactly those three `Dome::MCPTool` resources, bound to the agent's identity, plus a narrower `mcp:discover` grant so the agent sees only the tools it may use. The declaration is the policy.

A generated grant is a short program the platform emits and deploys:

```
permit(
  principal is Dome::Agent,
  action == Dome::Action::"mcp:call",
  resource is Dome::MCPTool
) when {
  principal.id == "a1b2c3d4-...-agent-uuid" &&
  (resource.name == "crm/list_accounts" ||
   resource.name == "crm/get_account")
};
```

Two tools in a declaration become an exact-match permit, and the agent can do precisely those two things — step 3 of the build-time sequence.

The two shapes of policy: from declaration, and from identity

The grant above is generated from a declaration — policy that follows from what the author asked for, and the workhorse of a horizontal platform. The second shape is identity-driven, where the decision turns on who the agent acts for.

An information barrier is the canonical case. A person on the wrong side of a barrier — a banker screened from a live deal, a clinician outside a patient's care team, a support agent scoped to one customer — must not reach across it, whichever agent acts for them. That is a property of the acting identity and the resource, not the agent. You express it as an authored `forbid` that reads the act-as roles on the principal:

```
// A person screened from a deal cannot open its records, whichever agent acts for
them.
forbid(
  principal is Dome::Agent,
  action == Dome::Action::"mcp:call",
  resource
) when {
  principal has act_as &&
  principal.act_as has roles &&
  principal.act_as.roles.contains("barrier_screened") &&
  resource.name like "*open_deal*"
};
```

Consider how this composes at runtime. The agent first makes a permitted, audited call to read the resource's restriction status. It stamps what it learns onto the act-as roles, then attempts the sensitive action. If the acting user is screened, the `forbid` fires at the gateway, the call returns 403, and the agent — which never touched the resource — takes an escalation path instead. The model never gets the opportunity to be talked around, because the decision is made outside the model, in infrastructure, on a tool call.

This is the central point: **the rules that matter belong in governance infrastructure, not in a system prompt.** A prompt that says "do not touch conflicted records" is a suggestion to a probabilistic system; a `forbid` at the gateway is a control. The test for what belongs where is simple: if a sufficiently clever conversation with the model could get around it, it does not belong in the prompt.

Deploy and test agent-scoped rules from the CLI. `simulate` answers the decision without making a call, which is useful in CI:

```
dome rules validate ./rules/revenue-analyzer.cedar --agent revenue-analyzer
dome rules apply    ./rules/revenue-analyzer.cedar --agent revenue-analyzer
dome rules simulate --agent revenue-analyzer --action mcp:call --resource crm/open_deal
--resource-type mcp_tool --actas-email dana@firm.example --actas-roles barrier_screened
```

Where evaluation happens

Both integration modes enforce the same rules. Through the gateway, evaluation happens on the hot path as the call passes. In-process, the SDK pulls the agent's effective policy (`GetAgentEffectivePolicy`) on an interval and evaluates locally, so the agent can check its own action first. Either way it is the same Cedar against the same bundles. One note on what you read back: references resolve to stable identifiers rather than names, so renaming a model

or tool never orphans a grant — you author with names and the system resolves them on deploy.

Route models

Every agent needs a model, and the naive approach is an environment variable holding a provider key. Across a hundred agents that produces three problems at once: credential sprawl, no cost attribution, and no reliable answer to which model an agent is using. The **model broker** makes model access a governed capability rather than a shared secret.

The vocabulary is **Broker / Provider / Pool**. A provider is an upstream — Anthropic, OpenAI, Google, Bedrock, Azure OpenAI, or an OpenAI-compatible endpoint such as a self-hosted vLLM — which Dome speaks as distinct wire families, so the broker is provider-agnostic. A pool is a named, policy-selected set of models with a routing strategy. The broker is the path an agent uses to reach a model without holding the provider's key.

The integration is intentionally simple: the agent points a standard Anthropic- or OpenAI-shaped client at a Gateway URL (`/gateways/{id}/v1`) instead of at the provider. Dome resolves the upstream, fetches the real credential server-side, authorizes `llm:invoke` against the resolved `Dome::LLMModel` , records the call, and returns a normal response. In the agent's code it is a one-line change of base URL.

The capability that justifies the hop is that model selection becomes policy rather than code. You declare pools and routing; the broker chooses:

```
pools:
  - name: premium
    routing: priority_weighted
    match_when: '{"any":[{"principal.act_as.roles":{"contains":"complexity_high"}}]}'
    members:
      - { model: claude-opus-4-8, priority: 0 }
      - { model: claude-sonnet-4-6, priority: 1 }   # failover
  - name: standard
    match_when: '{"principal.act_as.roles":{"contains":"tier_standard"}}'
    members: [{ model: claude-sonnet-4-6 }]
  - name: triage
    default: true
    members: [{ model: claude-haiku-4-5 }]
```

The application does not choose a model. It stamps dimensions — a request's tier, an assessed complexity — onto the acting identity, and the broker's `match_when` predicates route: a hard request to the premium pool with an Opus primary and a Sonnet failover; routine triage to Haiku. Change the routing policy and every agent's model selection changes with it, without a redeploy. Because authorization is re-evaluated for each candidate in a failover chain, a

`forbid` holds even when the broker falls back to the second model. Cost quotas attach to pools as a separate barrier, so a spend ceiling is enforced in the same place, in real money.

Declare a connection, then a routed pool, from the CLI:

```
dome model add claude-opus   --provider anthropic --model claude-opus-4-8   --api-key
"$ANTHROPIC_API_KEY"
dome model add claude-sonnet --provider anthropic --model claude-sonnet-4-6 --api-key
"$ANTHROPIC_API_KEY"
dome model pool create premium --routing-strategy priority_weighted --failover-max 2 --
match-when '{"any":[{"principal.act_as.roles":{"contains":"complexity_high"}}]}'
dome model pool member add premium claude-opus   --priority 0
dome model pool member add premium claude-sonnet --priority 1
```

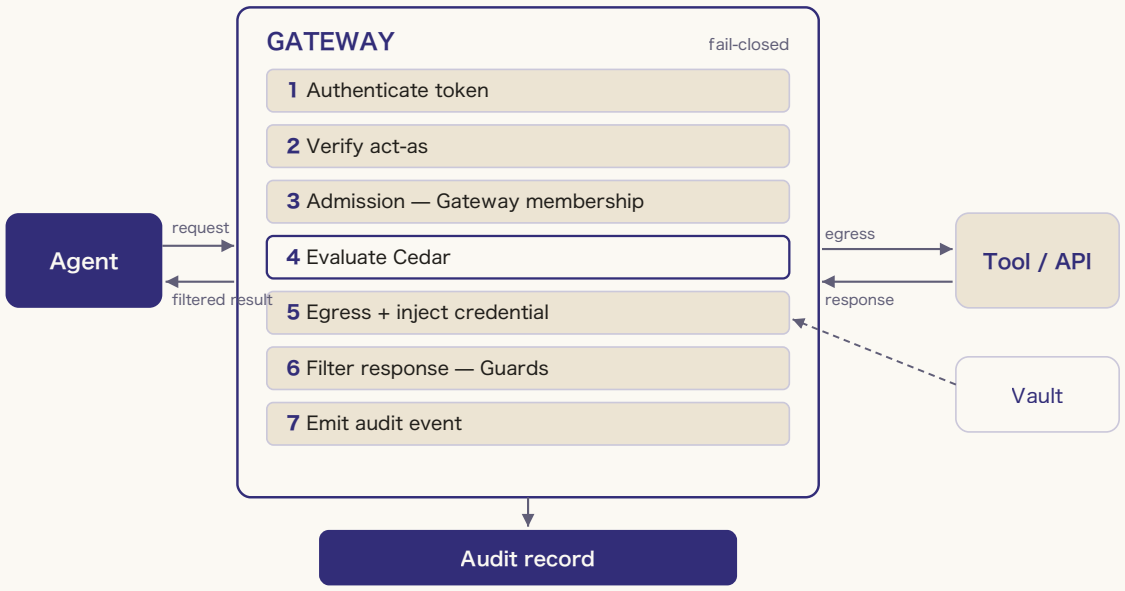
You need not route every call on day one. Route the model when model choice must be policy — cost tiers, data-residency-constrained models, per-request routing — or when provider keys cannot sit beside agent code. Until then, keep a single model call-site in your runtime so the later switch is a one-file change.

Broker tools

If authorization is the decision, the **gateway** is where it meets the reality of tools. It is the single endpoint an agent's tool traffic flows through, and it upholds one invariant: agents never access tools directly and never hold tool credentials. A tool call is ultimately a call to a backend API, and the security posture depends on the agent never holding that API's credential.

The concrete form is the **Gateway** — a workspace-scoped, curated view over a set of callable resources: MCP tools, REST-backed tools, and model pools. An agent addresses it at `/gateways/{id}`, appending `/mcp` for tools (`tools/list` to discover, `tools/call` to invoke) or `/v1` for models. A resource in no Gateway is unreachable, which gives you a structural boundary before any policy runs: admission is a coarse yes/no that can only restrict, checked before Cedar, and it filters discovery too — a non-admitted agent sees an empty catalog, not a forbidden one.

A single tool call exercises the whole substrate, in order: the gateway authenticates the token; verifies the act-as assertion; checks kill switches and drain state; confirms the tool belongs to this Gateway and the agent is admitted; evaluates Cedar; and only then executes egress — fetching the real tool credential from the vault server-side, injecting it, and making the upstream call. On return it can filter the response, then emits an audit event. Routing through this path means your platform and agents get all of it without implementing credential injection, fail-closed evaluation, or audit emission. You register the tool sources, assemble a Gateway, admit the agent, and make a well-formed call.



One governed tool call. The agent holds no tool credential; the gateway runs a fixed, fail-closed sequence — authenticate, verify identity, admit, evaluate Cedar, inject the credential from the vault on egress, filter the response, and record it.

Governing what comes back: Guards

Authorization governs whether an action happens; it says nothing about what the action returns. In an agent system the return is often the sensitive part — a customer record carrying a national ID, a document quoting another client's matter. **Guards** govern the response. A Guard inspects governed traffic and can rewrite or withhold it, and it is direction-aware: a guard on the `request` direction sees the outbound prompt or tool arguments; one on the `response` direction sees the completion or the tool result. It can `redact` a matched span, `block` the whole payload, or `omit` a field. The matchers suit this job — substring, national-ID and Luhn-checked card patterns, phone, and N-digit patterns for text; JSON-path field filters for structured tool results. Guards are named, workspace-scoped, and versioned like rule bundles, and assigned per connection and direction. So "this tool may be called, but its results always have card numbers masked and this field dropped" is configuration, not code.

```
dome guards filters create pii-redact --description "Redact PII in tool responses" --
redact-ssn --redact-substring "internal-only"
dome tool guards filters set crm --direction response --filters pii-redact
```

Account: the audit payoff

Everything that flows through the governed path is recorded. This is the responsibility that turns "we have agents" into "we can answer for our agents," and it is where the design pays off: because identity, authorization, and egress already run through the substrate, the record is a byproduct, not a separate instrumentation effort.

Audit in Dome is a system of record, not a log. On the control plane, an audit event is written in the same database transaction as the change it describes, so a failure to record rolls the change back — you cannot make a governed mutation and fail to account for it. That is a guarantee a best-effort log cannot make.

What a governed action records

Each event is a typed, versioned envelope. In customer terms, it captures:

- **The actor and the acting identity** — the agent, plus the full act-as chain: who it acted for, with which roles and groups, and how that identity was verified.
- **The scope** — the organization, tenant, workspace, and agent the action targeted.
- **The action and resource** — for example `mcp:call` on a named tool, or `llm:invoke` on a resolved model.
- **The result** — attempted, succeeded, denied, failed, or filtered.
- **Correlation** — a `trace_id` for the single request and an `activity_id` for the whole run, with a trust label recording whether Dome assigned the id or the caller asserted it.
- **Data handling** — flags recording whether content was redacted, omitted, or truncated by a Guard.

Actions emit a paired `*.attempted` and `*.completed` event rather than one overloaded record, so an in-flight action and its outcome are always distinguishable. Representative event types:

Category	Representative event types
Tool traffic	`mcp.tool_call.attempted`, `mcp.tool_call.completed`, `mcp.tools_list.completed`, `mcp.tool_result.filtered`
Model traffic	`llm.model_call.attempted`, `llm.model_call.completed`, `llm.model_call.failover`, `llm.model_result.filtered`
Authorization	`authorization.decision`, `access.denied`, `authorization.act_as.rejected`
Rules and policy	`authorization.rule_bundle.deployed`, `authorization.rule_bundle.rolled_back`, `authorization.policy_simulation.completed`
Agent lifecycle	`agent.registered`, `agent.token.issued`, `agent.suspended`, `agent.revoked`, `agent.api_key.rotated`

An activity chain, end to end

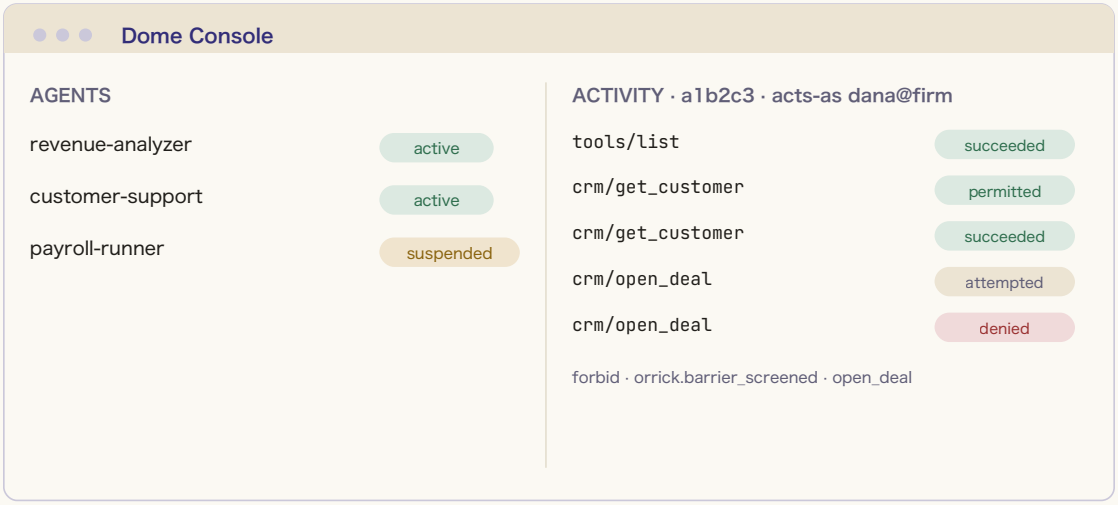
Every governed call in a run carries the same `X-Dome-Activity-Id`, and the substrate groups them at read time — within your workspace scope — into one activity chain. Here is the information-barrier run from the Authorize chapter as it appears in the record: the agent, acting as a screened banker, reads a customer then attempts to open a restricted deal.

#	Event type	Acting identity	Resource	Result
1	`agent.token.issued`	agent, acts-as dana@firm	—	issued
2	`mcp.tools_list.completed`	agent, acts-as dana@firm	crm (admitted tools)	succeeded
3	`mcp.tool_call.attempted`	agent, acts-as dana@firm	`crm/get_customer`	attempted
4	`authorization.decision`	—	`crm/get_customer`	permitted
5	`mcp.tool_call.completed`	agent, acts-as dana@firm	`crm/get_customer`	succeeded
6	`mcp.tool_call.attempted`	agent, acts-as dana@firm	`crm/open_deal`	attempted
7	`access.denied`	agent, acts-as dana@firm	`crm/open_deal`	denied (barrier)

The chain shows the whole story in one query: the same identity throughout, a permitted read, and a policy-denied write with the rule that stopped it — the evidence a reviewer or regulator asks for, without reconstructing it from scattered logs.

The operational surface

Because the substrate is the record, the operator's view is a query over it rather than a bespoke pipeline. The registry is a live inventory of every agent, its status, and the tools and pools it can reach; the audit record streams live over server-sent events and exports in batch, with CEF and OCSF formatters for your SIEM. Audit is the evidence-grade record; webhooks are the selective, real-time reflex for triggering automation. Use each for its purpose.



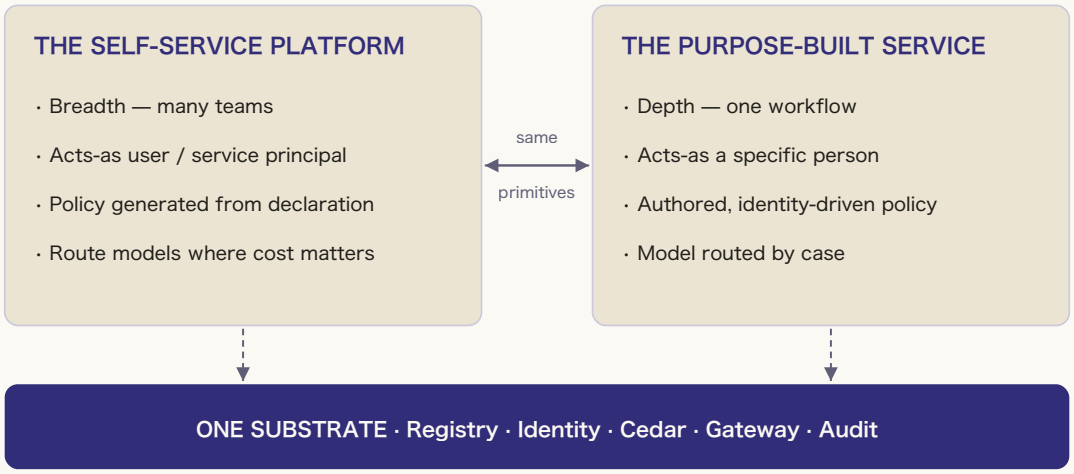
Illustrative operator view: the agent estate with live status, and one activity chain expanded to show the permitted read and the policy-denied write. The visual payoff of routing every action through the substrate.

Follow one run, inspect the chain, and export a window for the SIEM:

```
dome audit chain "$DOME_ACTIVITY_ID"
dome audit query --results denied --since 2026-08-01T00:00:00Z
dome audit export --format ocsf --since 2026-08-01T00:00:00Z > audit.ocsf.jsonl
```

Two patterns

The responsibilities are constant; the platforms built on them are not. Two patterns recur, at opposite ends of a spectrum. Understanding why each is shaped as it is will help you place your own build — and most enterprises run both on the same substrate.



Two patterns, one substrate. Both act on behalf of a real identity; they differ in scope and in the richness of that identity, not in the primitives beneath them.

Pattern 1 — The Self-Service Agent Platform

A horizontal paved road: any team declares an agent — instructions plus a list of tools — and the platform manufactures, governs, and runs it. This is the pattern that resembles the managed agent platforms teams already know from the market: bring a prompt and a tool list, get a running governed agent. Its defining property is that governance is automatic and invisible to the author. A team writes a manifest — your schema, not Dome's:

```
apiVersion: v1
kind: Agent
metadata:
  name: revenue-analyzer
spec:
  model: claude-haiku-4-5
  instructions: |
    You are a revenue analyst...
  tools:
    - crm/list_accounts
    - crm/get_account
  acts_as: invoking_user       # the agent carries the identity of the user who
triggered it
  governance:
    owner: revenue-team
```

The manufacturing step runs the build-time sequence: register the agent, mint its key, compile the tool list into a least-privilege grant and deploy it, admit the agent to a Gateway carrying those tools, and run it. The author never sees a policy, a credential, or the platform key. The pattern runs on generated-from-declaration authorization, and agents act as the invoking user (or a shared service principal for background work), so every call remains attributable to a real actor.

The transferable idea is the split of roles it forces. The **operator** stands up the substrate once — registers the tool sources, builds the Gateways, holds the platform key. The **author** writes a manifest and nothing else. Everything in between is your platform translating declared intent into governance artifacts. Build that translation once and the long tail of internal agents becomes self-service.

Pattern 2 — The Purpose-Built Agent Service

A vertical service embedded in a single high-stakes workflow, where the rules are domain rules and the agent acts for a specific person. Where the self-service platform optimizes for breadth, this optimizes for depth. There is usually one workflow, often one team, and the governance is not generic least-privilege but a domain policy that must be exactly right:

- **Finance** — an adviser agent that may act for one client's book but is walled off from a deal it is screened from; restricted lists and information barriers enforced as `forbid` rules on the acting identity.
- **Healthcare** — a case agent held to minimum-necessary access: it can reach the records of patients in its care team and no others, decided by the clinician it acts for.

- **Support and multi-tenant SaaS** — an agent hard-scoped to one customer's data boundary, where the tenant is a property of the acting identity, not a parameter the model can change.

The pattern runs on identity-driven authorization — the information-barrier `forbid` from the Authorize chapter is its signature, the same action flipping outcome on who the agent represents. It usually routes the model too, by case attributes (complexity, tier, data residency), so a sensitive matter gets the capable, correctly-located model and routine work gets the inexpensive one. And because the acting identity is a specific person, the audit trail names that person — frequently the reason the workflow was allowed to be automated at all.

Choosing between the patterns

Dimension	Self-Service Platform	Purpose-Built Service
Optimizes for	Breadth — many agents, many teams	Depth — one workflow, done exactly right
Declaration	Author writes a manifest	Engineers write the workflow
Acting identity	The invoking user or a service principal	A specific named person; roles drive per-record decisions
Authorization	Generated from the declaration	Authored, identity-driven
Model access	Routed where cost or residency matters	Routed by case attributes
Audit answers	"What can this agent do?"	"What did this agent do, and for whom?"

You do not have to choose. The two share every primitive in the governance table — the same registry, Cedar, gateway, and audit — and differ in scope and in the richness of the acting identity. A realistic enterprise builds the self-service platform for the long tail and stands up a handful of purpose-built services for the workflows where the rules are load-bearing. Because both sit on one substrate, that is one governance model and one audit trail, not two.

Integration and rollout

Two practical questions remain: how your platform attaches to the substrate, and where to begin.

Attachment is the choice described earlier. Route tool and model traffic through the gateway for thin agents that hold no credentials — the default, and what both patterns use for anything touching a real tool or model. Embed the SDK when an agent should evaluate its own actions in-process against synced policy. Many platforms use both, and the choice need not be uniform across the estate.

On where to begin: do not build the whole machine before anything is governed. The substrate supports phased adoption, and the phases are a genuine maturity path.

Phase	What you do	What you get
1 — Visibility	Register existing agents. Catalog the tools they touch. Establish identity. Turn on audit. Change no agent's behavior.	A queryable picture of the estate: what agents exist, what they can reach, what they do.
2 — Policy	Author rules; generate least-privilege grants from declarations. Version and test in staging. Enforce at the gateway.	Every governed action checked before it happens. Fail-closed, deterministic, auditable decisions.
3 — Scale	Extend the same primitives across teams and runtimes. Route models where model choice must be policy. Stream audit to the SIEM.	One governance model across the whole estate.

Phase 1 is non-invasive by design: it delivers value — and tells you what you actually have — before you change a single running agent. Start there.

Common failure modes

Each pattern has a failure mode, and the failures transfer better than the successes because they recur.

Reimplementing governance in the runtime. Under a deadline, someone adds a quick permission check "just for now." It ships and becomes a parallel authorization system with no audit, no versioning, and no forbid-wins guarantee, inside the runtime where no security review will find it. Governance belongs behind one contract, evaluated in one place. If you must ship before that contract is wired, make the gap explicit and temporary, never a quiet local path that grows into the real decision.

Governance in the system prompt. "You must not access records outside your region" is not a control; it is a hope. Anything a sufficiently clever conversation could subvert belongs in a `forbid` evaluated outside the model. The prompt is for competence; policy is for permission.

Provider keys beside agent code. The environment-variable API key becomes a sprawl you cannot rotate, attribute, or revoke per agent. Route credential-holding through the substrate so no agent holds a raw key; the blast radius of an incident is then one short-lived token, not a shared secret in a hundred deployments.

Treating audit as logging. Logs are for you; audit is for the auditor. A best-effort write that can silently drop events and cannot answer "under which rule" will not survive a regulator. Use the transactional, typed, chained record for the questions you will be required to answer.

RBAC-only thinking. Role-based access controls who can invoke the agent. It says nothing about what the agent, once invoked, decides to do — and that runtime decision is why agents need new governance. The action-level check at the tool-call boundary is the one that matters.

Building the platform before governing anything. A fully configured platform with pools, quotas, and filters, before a single agent is governed, spends months on infrastructure and ships nothing. Start with visibility, add policy where the risk is, and route models where model choice must be policy.

Conclusion

The instinct on being asked to "build the agent platform" is to build all of it, because it appears to be one system. It is two. The first — declaring, manufacturing, running, and operating agents — is yours to build, because it is where your judgment adds value. The second — identify, authorize, route models, broker tools, account — is a substrate to consume, as you long ago stopped building your own identity providers.

The work is therefore narrower than the request first appears. A practical sequence:

- **Draw the boundary.** Decide which responsibilities your platform owns — declare, manufacture, run, operate — and which you delegate to the substrate.
- **Give every agent an identity.** Register it and have it act for a real user or service principal; never let it act anonymously.
- **Make every action a decision.** Enforce authorization outside the model, at the tool-call boundary, and fail closed.
- **Route models through the broker.** Express model choice as policy and keep provider keys out of agent code.
- **Reach tools only through the gateway.** No agent holds a tool credential; govern what returns with Guards.
- **Account for everything.** Treat the audit record as the system of record and stream it to your SIEM.
- **Start with visibility.** Register and observe what you already run before you begin to enforce.

Build the part that is genuinely yours as well as you can, on a substrate you can rely on.

Building or governing an agent platform?

Dome is the governance substrate described in this paper — identity, authorization, model routing, a governed path to tools, and an accountable record. Talk to us about the design for your estate, or see it enforce policy live, at **domesystems.ai/contact**. For more on the governance the agent era requires, read our Perspectives at **domesystems.ai/perspectives**.