DOME SYSTEMS

Agents Need to Talk to Models

A Field Guide to Dome's LLM Router

Introduction

Every agent needs a model, and the naive arrangement is an environment variable holding a provider key. It is one line of configuration and it works immediately, which is exactly why it spreads.

Across a hundred agents it produces three problems at once. The key is a shared secret in a hundred deployments, so rotating it is a coordinated release and an incident means revoking access for everything at once. There is no cost attribution, because the provider's invoice knows about one API key, not about which team or which customer generated the spend. And there is no reliable answer to which model an agent is actually using, because the model name is a string in application code that someone changed in a pull request nobody read closely.

The **Broker** makes model access a governed capability instead of a shared secret. The agent points a standard client at a Dome endpoint, Dome resolves which upstream should serve the request, fetches the real provider credential server-side, authorizes the call against the model that will actually run, dispatches it, inspects what streams back, and records the tokens. The agent never holds a provider key.

This guide covers the configuration surface rather than the internals. Its companion, Agents Need to Talk to Your Systems, does the same for tool traffic.

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. 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.

What a governed model call is

The integration is intentionally unremarkable. Model traffic targets a **Gateway** — the same named access surface that carries tool traffic — at its `/v1` suffix. In the agent's code it is a one-line change of base URL, from the provider's host to `https://<host>/gateways/<id>/v1`, with the agent's Dome token as the bearer credential.

Dome exposes provider-native wire shapes, so existing SDKs work unmodified:

Method	Path, relative to `/gateways/<id>`	Shape
`POST`	`/v1/messages`	Anthropic messages
`POST`	`/v1/chat/completions`	OpenAI chat
`POST`	`/v1/responses`	OpenAI responses
`POST`	`/v1/embeddings`	OpenAI embeddings
`POST`	`/v1/moderations`	OpenAI moderations
`POST`	`/v1/passthrough/:name`	Opaque per-provider passthrough
`GET`	`/v1/models`	Cedar-filtered model listing

Streaming and non-streaming both work; SSE chunks are inspected in flight rather than buffered to completion. The `/gateways/<id>` segment is required — a bare `/v1` fails closed with a `400` and a `"select a gateway"` message, so a client that loses its Gateway id cannot fall through to an ungoverned default.

`GET /v1/models` is worth noting as a design affordance. It returns the Cedar-filtered set, which means an agent can enumerate what it may use and a pool rename does not require a coordinated client change. Discovery is exempt from the Gateway admission grant but stays membership- and Cedar-filtered.

Each call then runs a fixed sequence: admission against the Gateway, pool resolution, authorization against the upstream that will actually be dispatched, request Guards, provider dispatch with the credential injected from Vault, response Guards over the stream, failover if a candidate fails, and audit.

Connecting a provider

A model connection names an upstream, how Dome authenticates to it, and any provider-specific configuration.

```
dome model add claude-sonnet \
  --provider anthropic \
  --model claude-sonnet-4-6 \
  --api-key "$ANTHROPIC_API_KEY" \
  --gateway prod
```

Dome speaks a set of distinct wire families and normalises the rest, so the Broker is provider-agnostic in practice: `anthropic`, `openai`, `google`, `bedrock`, `azure_openai`, and the hosted OpenAI-compatible vendors — `mistral`, `groq`, `together`, `fireworks`, `deepseek`, `xai`, `perplexity`, `cohere`, `openrouter`, `cerebras`, `nvidia`, `deepinfra`, `sambanova`, `ai21`, `databricks` — plus `openai_compatible` and `custom` for anything you host yourself.

The key is stored in Vault under the provider's managed auth header, which differs by vendor and which you do not need to configure: `Authorization: Bearer` for OpenAI, Bedrock, and the hosted compatible vendors; `x-api-key` for Anthropic; `api-key` for Azure OpenAI; `x-goog-api-key` for Google.

Endpoints are prefilled from the provider registry when you omit `--endpoint`. Five providers require it explicitly, because there is no sensible default: `azure_openai`, `bedrock`, `databricks`, `openai_compatible`, and `custom`.

```
dome model add gpt-4o \
  --provider azure_openai \
  --model gpt-4o \
  --endpoint https://my-resource.openai.azure.com \
  --api-key "$AZURE_OPENAI_KEY" \
  --provider-config '{"api_version":"2024-08-01-preview","deployment":"gpt-4o-prod"}' \
  --gateway prod
```

`--provider-config` carries anything vendor-specific and merges with `--model` and `--endpoint`. On `update` it replaces wholesale rather than merging, so pass the complete object.

Two more flags matter at connection level:

Flag	Purpose
`--attributes`	Cedar attributes as JSON, so rules can reason about properties of this connection — region, data classification, cost band
`--filter-window-bytes` / `--filter-window-tokens`	Per-connection streaming inspection window. `0` inherits the workspace floor.

`--attributes` is the hook that makes several of the later patterns possible. A connection tagged `{"region":"eu","residency":"gdpr"}` can be reasoned about by a rule without the rule naming the connection, which means adding a second EU model does not require editing policy.

Provider is immutable after create. Switching providers means adding a new connection and moving pool membership, which is deliberate — it keeps the audit record's notion of "which upstream served this" stable over time.

OAuth-authenticated providers

Some providers authenticate with something other than a static key: Anthropic OAuth, an Azure AAD/Entra service principal, Google Workload Identity for Vertex.

```
dome model add claude-oauth \
  --provider anthropic \
  --model claude-sonnet-4-6 \
  --auth-method oauth \
  --credential-type shared

dome model oauth-connect claude-oauth
```

`oauth-connect` prints a single-use URL valid for about ten minutes; tokens land in Vault on the callback. `oauth-disconnect` revokes the credentials and deletes the Vault bundle while preserving the client configuration, so a later reconnect reuses the same registered client. Per-user OAuth models skip `oauth-connect` — consent is gateway-triggered on each end user's first call.

Pools

A Pool is a named set of model connections with a routing strategy. It is the object agents address, and the indirection is what lets model choice change without an agent redeploy.

```
dome model pool create production --failover-max 2
dome model pool member add production claude-sonnet --priority 0 --weight 3
dome model pool member add production gpt-4o       --priority 0 --weight 1
dome model pool member add production claude-haiku --priority 1
dome model pool set-default production
```

Members

Each member carries three fields:

Field	Default	Meaning
`--priority`	`0`	Failover walks ascending. `0` is primary.
`--weight`	`1`	Distribution within a priority bucket
`--enabled`	`true`	Whether the member participates in routing

The two-level structure is the useful part. Members sharing a priority are peers and split traffic by weight; a higher priority number is a fallback tier reached only when the tier above it fails. In the example above, Sonnet and GPT-4o split primary traffic 3:1, and Haiku serves only when both are failing.

`--enabled false` takes a member out of routing without removing it, which is the right way to park a model during a provider incident — the membership and its weights survive for when you turn it back on.

Routing strategies

Strategy	Selects	Uses
`priority_weighted` (default)	By weight from the lowest priority tier	Priority and weight
`round_robin`	Rotates through all members in order	Neither
`least_loaded`	The member with the fewest in-flight requests, ties broken randomly	Neither

`--strategy-scope` controls whether the stateful strategies share their counters: `workspace` shares round-robin rotation and in-flight counts across all agents, `caller` tracks them per agent. `priority_weighted` ignores the setting because it holds no state.

The default is right most of the time. Reach for `round_robin` or `least_loaded` only when members are genuine peers — same capability, same cost band — because both strategies discard the notion that one model is preferred. `least_loaded` is the better choice of the two when member latency varies, since it responds to actual load rather than assuming uniform service time.

Failover

`--failover-max`	Behaviour
`all` (default)	Try every eligible member until one succeeds
`0`	Try only the primary
`N`	Try the primary and up to `N` further members

Two properties of failover deserve attention because they are load-bearing for security and for correctness.

Every candidate is authorized on its own. Cedar evaluates against the upstream that will actually be dispatched: the primary up front, and each failover candidate immediately before its own attempt. So a `forbid` on a model attribute holds for every upstream the call could reach, not just the first one. Failover cannot route around a policy. A policy-refused candidate is skipped, with an `access.denied` event, rather than being treated as a successful route-around — and a deny on the primary of a non-streaming call is terminal.

Streaming failover has a hard boundary. Failover applies only before the first response byte reaches the caller. Once bytes are flowing, a later upstream failure terminates the stream

rather than restarting it on another model. Agents that stream need to handle a truncated stream; a low `--failover-max` will not save them from that.

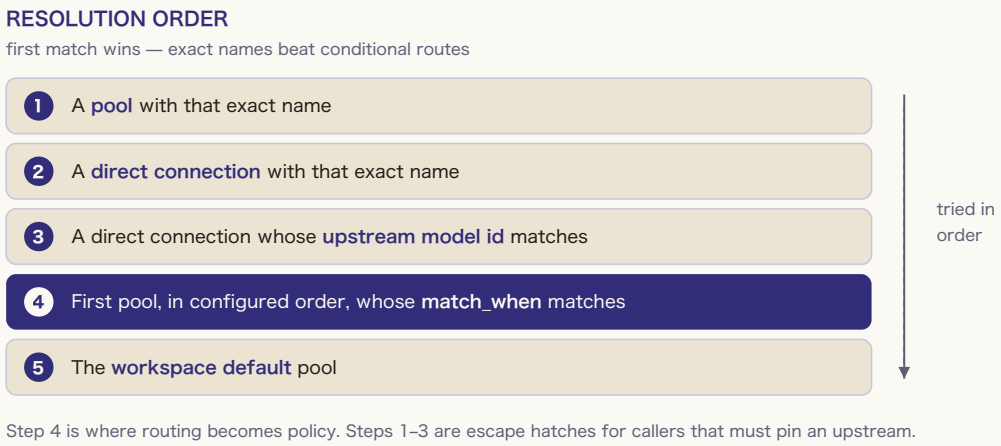
Response caching

Flag	Default	Meaning
`--cache-ttl-secs`	`0` (disabled)	Exact-match response cache lifetime
`--cache-scope`	`workspace`	`workspace` shares cache entries across agents; `caller` keeps them per agent

Exact-match caching helps where prompts genuinely repeat — classification, extraction, deterministic enrichment. Scope it to `caller` when a cached completion could leak context between agents or between the people they act for; that is the safer default whenever the prompt embeds anything tenant-specific.

Routing as policy

The capability that justifies the hop is that model selection becomes configuration rather than code. An agent sends a `model` value; Dome resolves it.



How a request's model value resolves. An agent that sends a pool name gets that pool; an agent that sends nothing meaningful falls through to conditional routing and then the workspace default. Because exact names win, a caller can always pin an upstream deliberately — which is also why an unreviewed model string in application code defeats your routing.

The practical consequence of exact-match precedence: if agents hard-code a specific model name, your `match_when` routing never runs for them. The pattern that works is for the application to send a pool name — or nothing at all and rely on the default — and to stamp dimensions onto the request rather than choosing a model.

The `match_when` dialect

A pool's `--match-when` is a JSON predicate over the request. These inputs are available:

Input	Matching
`prompt_tokens`	Estimated prompt size (characters ÷ 4), with `gt`, `lt`, `eq`
`requested_max_tokens`, `tool_count`	Numeric `gt`, `lt`, `eq`
`endpoint`, `header.<name>`	Equality, `in`, `prefix`, `suffix`, RE2 `regex`
`principal.metadata.<key>`	String or numeric
`principal.act_as.sub`, `.email`, `.claims.<key>`	String or numeric
`principal.act_as.roles`, `.groups`	`contains`, `containsAny`, `containsAll`
Nested predicates	`any`, `all`, `not`

By default every condition must match; `any`, `all`, and `not` combine them differently. Three failure semantics are worth memorising because they all fail safe in the same direction: an unknown condition never matches, a missing value never matches, and an **unverified act-as attribute never matches**. A pool selected on `principal.act_as.roles` will not be reached by a caller whose act-as assertion was not verified — it falls through to the next pool rather than matching on an unverified claim.

Pools with non-empty predicates evaluate in ascending order and the first match wins, so ordering is part of the configuration:

```
dome model pool move premium --before standard
```

Order specific predicates ahead of general ones. A pool matching `prompt_tokens > 100000` placed after a pool matching `prompt_tokens > 20000` will never be reached.

Routing is not authorization

This is the one guardrail to hold onto. `match_when` decides which upstream serves a request. Cedar decides whether the caller may use it. They are evaluated separately, and routing runs first.

So a `match_when` predicate is not a control. If a model must never serve a particular caller, that is a `forbid` on `Dome::LLMModel` — which, because every failover candidate is authorized individually, holds across the whole chain. Rules can read `resource.name` for the caller-facing alias, `resource.resolved_model` for the upstream that will actually run, and `resource.pool` when the request came through a pool.

Patterns for routing

Pattern 1 — One default pool with cross-provider failover

The baseline, and the right first configuration for almost every workspace. One pool, two providers, a fallback tier.

```
dome model add claude-sonnet --provider anthropic --model claude-sonnet-4-6 --api-key
"$ANTHROPIC_API_KEY"
dome model add gpt-4o        --provider openai    --model gpt-4o            --api-key
"$OPENAI_API_KEY"
dome model add claude-haiku  --provider anthropic --model claude-haiku-4-5  --api-key
"$ANTHROPIC_API_KEY"

dome model pool create default --failover-max 2
dome model pool member add default claude-sonnet --priority 0 --weight 1
dome model pool member add default gpt-4o        --priority 1
dome model pool member add default claude-haiku  --priority 2
dome model pool set-default default
dome model pool gateways add default prod
```

Agents send no model, or send `default`. Provider diversity across priority tiers means a single vendor's outage degrades quality rather than stopping work.

Pattern 2 — Cost tiering by request shape

Route on what the request looks like, not on who sent it. Cheap models handle routine work; capable models handle hard work.

```
dome model pool create premium \
  --match-when '{"any":[{"prompt_tokens":{"gt":100000}},{"tool_count":{"gt":12}}]}' \
  --routing-strategy priority_weighted --failover-max 1
dome model pool member add premium claude-opus   --priority 0
dome model pool member add premium claude-sonnet --priority 1

dome model pool create triage --match-when '{"prompt_tokens":{"lt":4000}}'
dome model pool member add triage claude-haiku --priority 0

dome model pool move premium --before triage
```

`prompt_tokens` is estimated as characters ÷ 4, so treat thresholds as approximate and leave headroom. `tool_count` is a good proxy for orchestration complexity — an agent handing the model fifteen tools is doing something harder than one handing it two.

The appeal of this pattern is that it needs nothing from the application. No header, no metadata, no code change. It also survives a new agent joining the estate, because the routing describes requests rather than callers.

Pattern 3 — Cost tiering by identity

When entitlement rather than difficulty should decide the model — a premium customer tier, an internal team with a budget, a role that warrants the better model.

```
dome model pool create enterprise-tier \
  --match-when '{"principal.act_as.roles":{"contains":"tier_enterprise"}}'
dome model pool member add enterprise-tier claude-opus --priority 0

dome model pool create standard-tier \
  --match-when '{"principal.act_as.groups":{"containsAny":
["tier_standard","tier_trial"]}}'
dome model pool member add standard-tier claude-sonnet --priority 0

dome model pool move enterprise-tier --before standard-tier
```

The fail-safe property matters here: because an unverified act-as attribute never matches, a caller that cannot prove its tier falls through to the next pool and ultimately the workspace default. Entitlement cannot be claimed, only proven. Note that this is still routing, not entitlement enforcement — if a lower tier must be prevented from reaching Opus, pair it with a `forbid`.

`principal.metadata.<key>` is the equivalent for service-principal callers with no human behind them.

Pattern 4 — Provider-outage resilience

A dedicated shape rather than a side effect of Pattern 1: spread the primary tier across providers so that no single vendor is a single point of failure, and keep the fallback tier on a third.

```
dome model pool update production --routing-strategy least_loaded --strategy-scope
workspace
dome model pool member update production claude-sonnet --priority 0
dome model pool member update production gpt-4o        --priority 0
dome model pool member update production gemini-pro    --priority 1
```

`least_loaded` with workspace scope is the strategy that responds to a degrading provider rather than a failed one — a vendor that is slow but not erroring accumulates in-flight requests and stops attracting new ones. Weighted priority routing will keep sending traffic into the slow provider until it actually fails.

Set `--failover-max all` here. This is the pattern where walking the whole member list is what you want.

Pattern 5 — Data residency

Pin models by region, and make the constraint a policy rather than a routing preference.

```
dome model add claude-eu \
  --provider bedrock \
  --model anthropic.claude-sonnet-4-6 \
  --endpoint https://bedrock-runtime.eu-central-1.amazonaws.com \
  --attributes '{"region":"eu","residency":"gdpr"}'

dome model pool create eu-only --match-when '{"principal.act_as.claims.region":
{"eq":"eu"}}'
dome model pool member add eu-only claude-eu --priority 0
```

```
// EU-resident subjects can only reach EU-resident upstreams — including on failover.
forbid(
  principal is Dome::Agent,
  action == Dome::Action::"llm:invoke",
  resource is Dome::LLMModel
) when {
  principal has act_as &&
  principal.act_as has claims &&
  principal.act_as.claims.region == "eu" &&
  resource.region != "eu"
};
```

The pool routes; the `forbid` guarantees. Because authorization runs against each dispatched upstream individually, the guarantee holds when the pool falls back — the exact scenario where a routing-only implementation quietly leaks traffic to the wrong region. Tagging connections with `--attributes` rather than naming them in the rule means adding a second EU model requires no policy change.

Pattern 6 — Migration and canary

Move traffic between models by weight inside one priority bucket, with no agent redeploy.

```
dome model pool member add    production claude-sonnet-next --priority 0 --weight 1
dome model pool member update production claude-sonnet      --priority 0 --weight 19
# 5%
dome model pool member update production claude-sonnet-next --weight 4
# 20%
dome model pool member update production claude-sonnet      --enabled false
# cut over
```

For a canary you want to observe rather than serve, put the candidate in a separate pool on a separate Gateway — see the environment-split pattern in the companion tool-gateway guide. Weights inside a live pool are for progressive rollout; a separate Gateway is for evaluation you do not want mixed into production traffic by accident.

Pattern 7 — Endpoint-shaped routing

Different call shapes often want different models. Embeddings should never reach a chat model, and a moderation call should not consume premium capacity.

```
dome model pool create embeddings --match-when '{"endpoint":{"suffix":"/embeddings"}}'
dome model pool member add embeddings text-embed-3 --priority 0

dome model pool create by-header --match-when '{"header.x-workload":{"in":
["batch","backfill"]}}'
dome model pool member add by-header claude-haiku --priority 0
```

`endpoint` matching with `prefix`, `suffix`, `in`, or RE2 `regex` keys routing off the wire shape. `header.<name>` is the escape hatch for a dimension the request body cannot express — a batch flag, a workload class, a caller-asserted hint. Treat header-driven routing as a convenience rather than a control: an agent can set its own header, so pair anything security-relevant with a rule.

Controlling spend

A Quota caps USD spend over a window. The gateway prices completed calls against the workspace pricebook, updates the window's spend, and enforces every applicable Quota.

Six subjects are available, and choosing the right one is most of the work:

Subject	Covers
`workspace`	All governed LLM spend in the workspace
`agent`	One agent's spend
`act-as`	One verified end-user subject's spend
`pool`	All spend through one pool
`model`	One connection, optionally scoped to one pool, optionally per-caller
`gateway`	All spend through one Gateway's member pools and direct connections

The distinction that matters

Exhaustion behavior differs by subject, and it is the most useful lever in the whole quota surface.

A per-model budget scoped to a pool spills. When that model's budget is exhausted, the pool routes to the next member instead. Work continues on a cheaper or different upstream.

A total cap rejects. When a workspace, agent, act-as, pool, or Gateway cap is exhausted, calls return HTTP `429`. A direct call to an exhausted model with no pool spillover path also returns `429`, with `subject_type: model`.

That gives you two distinct instruments. Use a spilling budget to shape cost — "spend at most $1,000/month on Opus inside this pool, then fall back to Sonnet" degrades quality gracefully and never stops work. Use a rejecting cap as a genuine ceiling — "this workspace does not spend more than $5,000/month" is a hard stop you want to hit loudly.

```
# Hard ceiling for the workspace.
dome model quota set --subject workspace --limit 5000 --window monthly --name ws-
monthly

# Spilling budget: cap Opus inside the production pool, then fall back.
dome model quota set --subject model --model claude-opus --pool production \
  --limit 1000 --window monthly --name opus-in-production

# Per-caller daily allowance on an expensive model.
dome model quota set --subject model --model claude-opus --per-caller \
  --limit 25 --window daily

# One agent that should not be able to run away.
dome model quota set --subject agent --agent <agent-id> --limit 200 --window monthly
```

`--per-caller` applies the limit independently to each agent and each verified end user rather than pooling their spend, which is the difference between "this model costs us at most $25/day" and "no single user can spend more than $25/day on it." It is available on model subjects.

Operational semantics

A few properties will otherwise surprise you:

- **Subject identity is fixed at create time.** Create is insert-only per subject identity and window pair. Retargeting a Quota means creating a new one.
- **Windows are `daily` (UTC midnight) or `monthly` (UTC month start, default).** There is no rolling window.
- **`--disabled` creates a Quota that is stored but not enforced.** Disabling is also how you pause enforcement without losing the definition and its history — prefer it to `rm`.
- **Limit changes reach the gateway on the next config sync**, while spend accrues continuously. Raising a limit mid-incident is not instantaneous.
- **The API works in micro-USD.** `limit_micros: 1000000000` is $1,000.00. The CLI and MCP surfaces take dollars.
- Cost Quotas are a **Pro**-plan capability; Free workspaces have a create limit of zero.

Governing content

Guards inspect what crosses the model boundary in both directions.

Direction	Sees
`request`	The outbound prompt, before dispatch to the provider
`response`	The streamed completion, before it reaches the agent

Model connections take `text`-kind Filters, and the kind is immutable at create time. Text Filters match on substrings, US SSN patterns, Luhn-checked card numbers, phone numbers, and N-digit patterns, with two actions — `redact` rewrites the matched span, `block` withholds the whole message and short-circuits the chain. (`omit` is JSON-only and therefore a tool-connection action.)

```
dome guards filters create pii-redact \
  --description "Redact SSNs and internal markers in completions" \
  --redact-ssn \
  --redact-substring "internal-only" \
  --block-substring "TOP-SECRET"

dome model guards filters set claude-sonnet --direction response --filters pii-redact
```

Each `(connection, direction)` slot holds one ordered chain, and `set` replaces it entirely. Order narrow transforms ahead of broad blocking matchers so audit attribution stays predictable when several could fire. Guards fail closed: a Filter whose stored config cannot be decoded blocks that connection and direction rather than relaying uninspected traffic. Filters are versioned, and `dome guards filters rollback <name> --to-version <n>` copies an earlier config forward.

A request-direction Filter is the underused half. It sees the prompt before it leaves your perimeter, which makes it the place to stop a customer identifier or an internal marker reaching a third-party provider at all — a different guarantee from redacting the completion.

The streaming window

Completions arrive as fragmented SSE chunks, so a pattern can straddle a chunk boundary. The gateway buffers each connection's response into a sliding window and runs Filters over the decoded text, and the effective window is the **maximum** of three layers, with any layer set to `0` dropping out:

Layer	Set via	Bounds
Workspace floor	Dashboard **Settings → Config**, or `UpdateWorkspaceLLMFilterWindow`	Bytes ≤ 1 MiB, tokens ≤ 4096
Per-connection	`--filter-window-bytes` / `--filter-window-tokens` on `dome model add` or `update`	`0` inherits the floor
Per-request	`_dome.filter_window_bytes` / `_dome.filter_window_tokens` in the request body	The `_dome` key is stripped before the upstream sees it

Because it is a maximum rather than an override, a connection or a request can only widen the window, never narrow it below the workspace floor. Set the floor to the smallest window that reliably catches your patterns; a wider window costs latency on every streamed response.

What the record shows

Model traffic emits paired attempt and completion events, so an in-flight call and its outcome are always distinguishable:

Event	Records
`llm.model_call.attempted`	A call beginning — agent, acting identity, pool, intended upstream
`llm.model_call.completed`	Its outcome, with token counts, matched rule, pool, and the final upstream
`llm.model_call.failover`	A candidate failed and another was tried
`llm.model_result.filtered`	A Guard redacted or blocked streamed content
`access.denied`	A call, or a failover candidate, refused by policy

Two details make this record more useful than a provider invoice. Events carry the **resolved** upstream, not just the alias the caller asked for, so "which model actually served this" is answerable after a failover. And cost is derived server-side from tokens and the workspace pricebook rather than transmitted by the caller, so attribution cannot be misreported by an agent.

Because every call carries the act-as chain and the activity id, the same query answers cost-per-customer, cost-per-team, and cost-per-workflow without a separate metering pipeline.

Where to start

Routing every call through the Broker on day one is not the goal. The sequence that pays off fastest:

- **Register connections and one default pool.** Two providers, a fallback tier, `set-default`, attached to a Gateway. Agents change one base URL and stop holding provider keys — which is most of the security benefit, before any routing exists.
- **Keep one model call-site in your runtime.** Whatever your agents look like, funnel model calls through a single helper. That is the file that changes when you adopt the Broker, and the one that changes again when routing gets more sophisticated.
- **Add a hard workspace cap early.** It costs one command and it is the control you will wish you had during an incident.
- **Route when model choice must be policy.** Cost tiers, data residency, per-request selection, or provider keys that cannot sit beside agent code. Until then, a default pool is enough.
- **Stamp dimensions, don't choose models.** Have the application assert tier, complexity, or workload class and let `match_when` decide. An agent that hard-codes a model name bypasses your routing entirely, because exact names win resolution.
- **Pair every routing constraint that matters with a rule.** `match_when` selects; `forbid` guarantees. Residency, tier exclusions, and model restrictions belong in Cedar, where they hold across failover.
- **Use spilling budgets to shape cost and rejecting caps as ceilings.** They are different instruments; most estates need both.

Getting provider keys out of your agents?

Dome's Broker makes model access a governed capability — one base URL, no provider credentials in agent code, model choice expressed as routing policy, spend capped where the budget lives, and every token accounted for. Talk to us about the routing model for your estate at **domesystems.ai/contact**. Its companion guide, Agents Need to Talk to Your Systems, covers the tool gateway. More at **domesystems.ai/perspectives**.