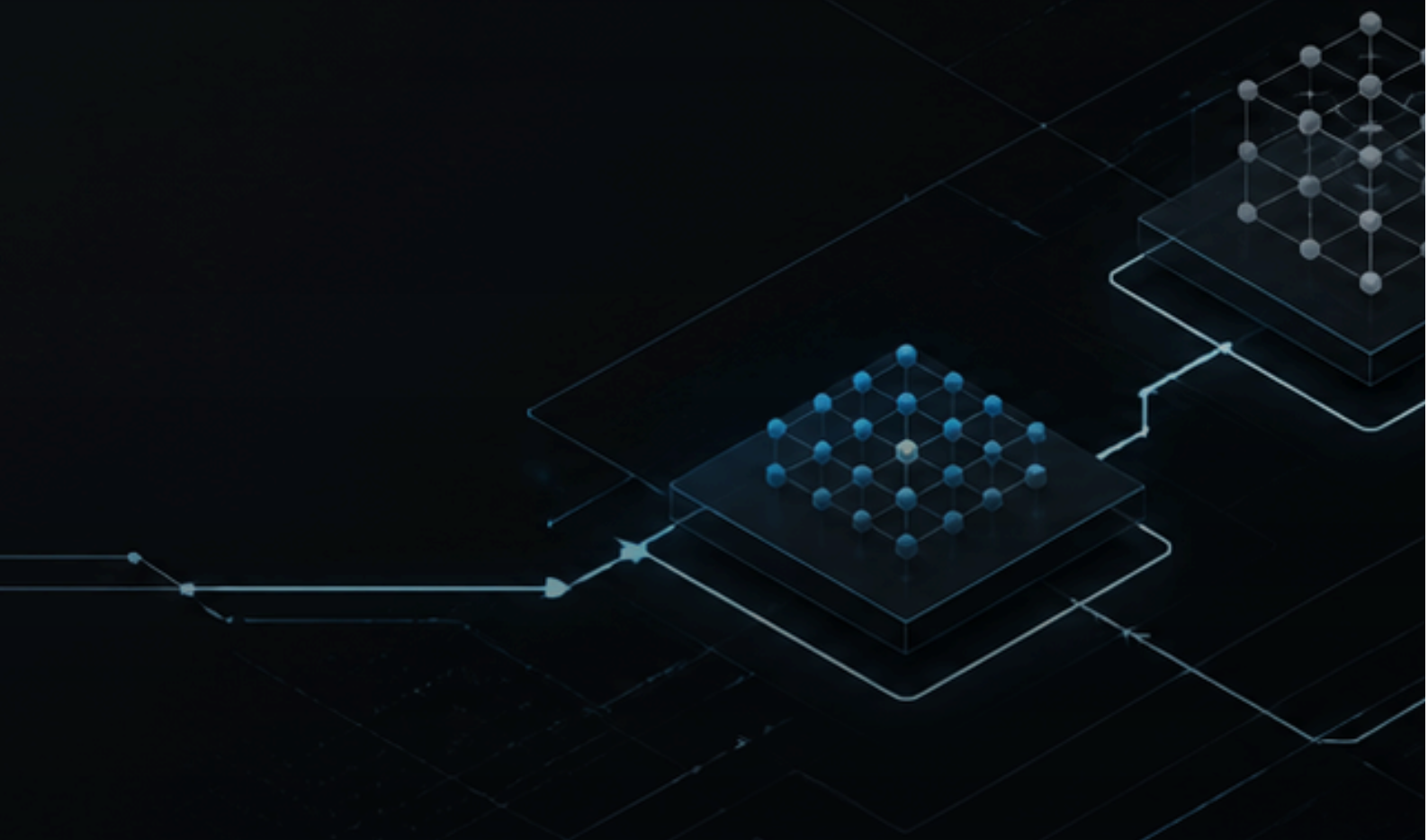
The Edge That Thinks

DESIGNING RELIABLE GENERATIVE AI SYSTEMS ON CLOUDFLARE



Kurt Overmier

STACKBILT FIELD NOTES

V0.2.1 PUBLIC-REVIEW EDITION · AUGUST 2026

The Edge That Thinks

Designing Reliable Generative AI Systems on Cloudflare

By Kurt Overmier · Stackbilt

Version 0.2.1 · public-review edition · published 2026-08-13 · verified through 2026-08-09

How to Use This Book

Verified through 2026-08-09.

This is a book about building generative systems that can survive contact with users, budgets, failures, and other systems. It is not a recipe for one chatbot, one framework, or one Cloudflare account.

The examples use Cloudflare because it offers a compact set of compute, state, storage, inference, and security primitives. The design questions are portable: where does state live; who may invoke a capability; how does work recover; when does a model call earn its cost; and what evidence remains when the answer is wrong?

Read in two passes

On a first pass, read Parts I–III in order. They establish the foundational separation between request handling, durable work, state ownership, tool authority, and inference. Then choose one case file from Part IV that resembles your product.

On a second pass, use the production checklist as a design review. If your system cannot answer a question there, do not compensate with a larger model or a longer prompt. Name the missing boundary and build it.

The recurring review questions

The prose uses topic-specific headings rather than forcing every subject into an identical template. Across the chapters, return to five review questions:

1. **Principle** — the vendor-neutral rule.
2. **Failure mode** — what happens when the rule is ignored.
3. **Cloudflare mapping** — which primitives can implement the rule.
4. **Tradeoff** — where the mapping is a poor fit or introduces complexity.
5. **Field note** — a clearly labeled example from Stackbilt or Cloudflare.

This recurrence is deliberate. A service name is not an architecture. A reader should be able to replace a service while retaining the reasoning that selected it.

Source conventions

Cloud platforms change quickly. A model catalog, a preview feature, an API surface, a quota, or a price can become stale while a manuscript sits in review. Accordingly:

- Inline links support claims about Cloudflare behavior.
- Each technical chapter states its verification date.
- Release-sensitive facts are maintained in the Cloudflare 2026 release ledger.
- Stackbilt examples are observations, not guarantees or benchmarks unless a methodology is provided.

The primary sources are Cloudflare's Developer Documentation, Workers Changelog, and Workers AI Changelog.

A note on "agent"

An agent is not defined by a chat interface or by the number of tools it can call. In this book, an agent is a system that can select and sequence actions toward a goal under a stated authority boundary. A scheduled report generator may be agentic in a narrow, useful sense; a general-purpose assistant with unbounded credentials may not be operationally safe at all.

That definition favors restraint. The goal is not to maximize autonomy. It is to earn precisely the autonomy that a workflow can support with evidence, controls, and recovery.

Part I — The Edge Is an Application Runtime

1. The Edge That Thinks

Verified through 2026-08-09.

The first mistake in generative AI architecture is to begin with the model.

It is an understandable mistake. A model is visible: you send text, it sends text back, and a convincing demo appears in an afternoon. But a product has to answer a more difficult set of questions. Which request deserves an expensive model? What happens when the answer requires a long-running generation? Where does a user's authorization end when an agent calls another service? How do you know a retrieval result was relevant, rather than merely available? What is the system allowed to do when no one is watching?

The model does not answer those questions. Architecture does.

This book calls an **edge-native generative system** one in which inference, state, asynchronous work, media storage, search, and service-to-service calls are deliberately designed as one application. The edge is not a decorative deployment target at the end of the build. It is the runtime in which the system's decisions occur.

Cloudflare happens to supply a useful collection of primitives for this style of work: Workers for request handling, D1 for relational state, Durable Objects for coordinated state machines, Queues for deferred work, R2 for large artifacts, Vectorize for similarity retrieval, Workers AI for on-platform inference, and Service Bindings for private composition. No single primitive creates an AI product. Their value is in the boundaries between them.

> **CLOUDFLARE PLATFORM MAP**
>
> Workers — D1
> Durable Objects — Queues
> R2 — Vectorize
> Workers AI — Service Bindings

That claim has become more—not less—important as the platform has added large models, Dynamic Workers, Dynamic Workflows, durable agent facilities, and, most recently, Cloudflare OS. The new primitives expand what can be built; they do not repeal the need to define state ownership, authority,

recovery, and cost control. Cloudflare's own Cloudflare OS makes the same move at organizational scale: curated context and skills, governed access to systems of record, isolated applications, and centralized model policy. Cloudflare OS: announcement Cloudflare OS: operational account

For the dated platform context behind this book, see the 2026 release ledger. The chapters use named services only when the mapping helps; the underlying design rules should remain useful to readers using a different cloud or a later Cloudflare release.

The demo architecture and the production architecture

A demo has a simple shape:

```
browser → API route → model → browser
```

It can be perfectly appropriate for exploration. It is also incomplete the moment the response has a cost, a duration, an owner, or a consequence.

The production shape is closer to this:

```
request
  → authenticate and establish tenant context
  → validate and classify
  → choose a deterministic path, an inexpensive model, or a specialist
  → persist the intent and the decision
  → either respond, invoke a bounded tool, or enqueue durable work
  → record outcome, cost, latency, and provenance
```

The difference is not bureaucracy. Each step eliminates a class of failure. Validation protects the model from malformed input. Classification keeps a trivial request from consuming an expensive budget. Durable work prevents a long generation from being held hostage by a client connection. A persisted decision makes an incident reconstructable. Tenant context stops an internal call from quietly becoming an authorization bypass.

Generative systems feel probabilistic at their center. The surrounding system should be unusually explicit.

A useful inversion: code first, inference second

We are accustomed to describing an AI application as a model with tools. A more reliable description reverses that: it is a deterministic application that occasionally asks a model for judgment.

That inversion changes design decisions immediately.

If a request is a health check, a known command, a schema transformation, or a previously successful procedure, do not ask a model to rediscover the answer. Run code. If the request needs semantic interpretation, call the smallest adequate model. If it needs expensive reasoning, send it there deliberately and leave a trace of why. The purpose is not to eliminate inference; it is to make inference exceptional enough to be observable and affordable.

AEGIS, one of the systems informing this book, approaches this as a cognitive cost ladder. A deterministic classifier recognizes request shape; a procedural memory can replay a proven path; increasingly capable model routes are reserved for work that genuinely needs them. The important pattern is not the particular models in that ladder. It is the admission that learned behavior can be implemented as deterministic behavior once it has earned trust.

This has a corollary that matters for agentic products: a repeated success is not automatically a procedure. Promote a pattern only with evidence, keep its success rate, and demote it when it fails. A bad shortcut that repeats confidently is worse than a slow system that asks for help.

The edge changes the integration vocabulary

Traditional cloud diagrams often make every component talk over a public HTTP boundary. At the edge, a service can frequently call another service through a private binding instead. That changes more than latency. It gives the designer a cleaner place to express trust.

An image-generation system illustrates the difference. The public gateway may authenticate a user, establish tenancy, enforce a quota, and validate a request. The generation orchestrator does not need to repeat that public ceremony. It needs a narrow, authenticated internal contract: here is a validated job, here is the tenant context, here is the allowed budget. The MCP-facing service can likewise act on behalf of a user without receiving a master credential that would let it act as every user.

The rule is simple: **internal does not mean untrusted, and it does not mean all-powerful.** Internal boundaries should be explicit capability boundaries. We will return to this principle in the chapter on tools and MCP.

The request is not the unit of work

Many generative tasks do not fit inside an HTTP request. Image rendering, multi-document ingestion, embedding, agent research, and batch transformations can take longer than a user is prepared to wait—and longer than a responsible request handler should remain responsible for them.

Treating these jobs as synchronous work produces familiar pathologies: clients retry after timeouts, the system produces duplicates, partial state is lost, and operators cannot tell whether a job failed or merely outlived its connection.

The better unit is a job with an explicit lifecycle. In practice that often means a request that creates a durable record, a queue that carries the work, and a coordinator that enforces transitions such as:

```
queued → processing → completed
                    ↘ failed
```

Those arrows sound pedestrian. They are the beginning of correctness. Once work has a state machine, you can make it idempotent, put deadlines on it, expose a status endpoint, meter it, retain it according to policy, and recover it after a worker restart. The user gains an honest answer—“accepted; check this job”—and the system gains a durable place to reason about the work.

What this book will and will not promise

Edge-native AI is not magic. A globally distributed runtime does not solve poor data modeling, unclear product intent, model unreliability, or governance. D1 is not an excuse to pretend consistency constraints do not exist. Vector search does not establish truth. An agent with many tools is not autonomous in a useful sense if no one has specified its authority.

What the edge can do is make a disciplined composition practical. It can place admission and coordination near the request, make an asynchronous workflow cheap to operate, keep service boundaries private, and let a small team ship a surprisingly complete system without assembling an entire infrastructure department first. It does not guarantee that the database primary, object storage, or inference provider is physically near either the caller or one another. Measure those locations and latencies instead of treating “edge” as a colocation promise.

The rest of the book is a guide to that discipline. We will begin with a less glamorous but foundational question: when a user request enters an edge system, what exactly is the work, and where is it allowed to live?

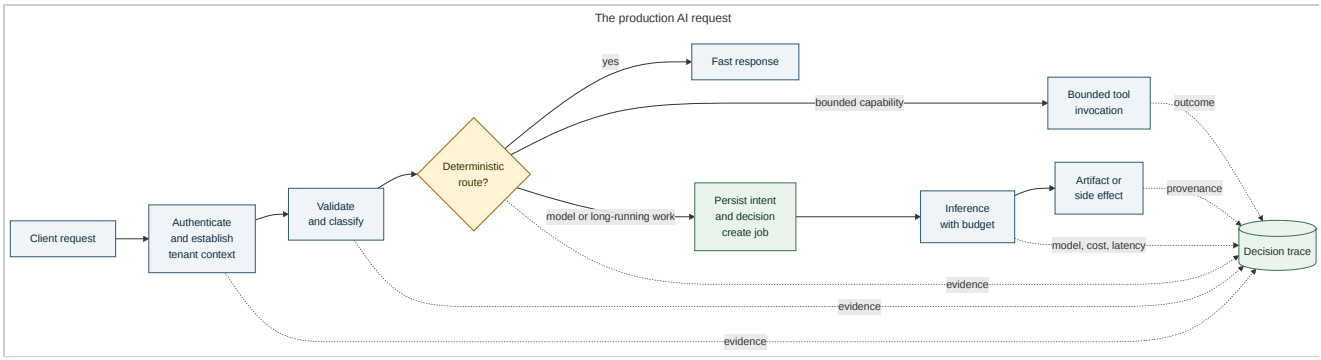


Figure 1. A production AI request is governed by deterministic controls around probabilistic inference.

2. The Request Is Not the Unit of Work

Verified through 2026-08-09.

An HTTP request is a delivery mechanism. It is not a promise that a task can be finished before the connection closes.

That distinction becomes urgent in generative systems because their work has variable duration and variable cost. One request may be a fast, deterministic lookup. The next may create a high-resolution image, parse a gigabyte-scale archive, generate embeddings for thousands of fragments, call several models, or ask an agent to inspect a repository. Giving all of those jobs the same request/response shape makes the easy case look elegant and the hard case unreliable.

The first production decision, then, is to name the unit of work.

A job is a contract with time

A good job record makes an otherwise invisible agreement explicit. At a minimum, it identifies the tenant or owner, the validated input, a state, an idempotency key, a budget or quota reservation, timestamps, and pointers to any large outputs. It is not merely a log row. It is the authoritative answer to: “What did we agree to do, and what happened?”

Here is a deliberately plain TypeScript sketch:

```
type JobState =
  | 'queued'
  | 'processing'
  | 'awaiting_approval'
  | 'settlement_pending'
  | 'completed'
  | 'reconciliation_required'
  | 'failed';

type AccountingState = 'reserved' | 'committed' | 'refunded' |
'disputed';

type GenerationJob = {
  id: string;
  tenantId: string;
  idempotencyKey: string;
  state: JobState;
  accountingState: AccountingState;
  input: { prompt: string; tier: 'draft' | 'standard' |
'premium' };
  reservationId: string;
  resultKey?: string;       // points to object storage, not an
inline blob
  errorCode?: string;       // safe, stable category—not a raw
provider error
  createdAt: string;
  updatedAt: string;
};
```

The shape should evolve with the product, but its discipline matters from day one. One idempotency key should converge on one business job and one accounting outcome even though a consumer—or an upstream provider after an ambiguous timeout—may execute more than once. “Idempotent” is not shorthand for exactly-once execution. A job is terminal only when its accounting is terminal too, or when it is explicitly marked for reconciliation.

The `img-forge` architecture uses this pattern for image generation: a gateway validates and authorizes the request, durable coordination governs the job lifecycle, a queue absorbs the work, inference providers generate the artifact, and object storage holds the result. The browser gets an

accepted job rather than an implausible guarantee that every model call will finish within a single round trip.

State has an owner

One recurring source of edge-system bugs is treating every storage primitive as interchangeable. It is not. The question is not “where can I put this data?” It is “which component is authoritative for this fact?”

Fact or responsibility	A suitable owner	Why
Tenant, quota reservation, billing outcome	Relational state and the identity/billing service	Requires auditable transitions and a shared policy boundary.
A single job’s live coordination and timers	Durable Object	One coordinator can enforce allowed transitions and deadlines.
Deferred delivery and retry	Queue plus durable job record	A message is transport; the job record is truth.
Generated media or raw uploaded archives	Object storage	Large bytes should not be database rows or model context.
Retrieval candidates	Vector index plus a source record	Similarity is an index, not the source of truth.
Cacheable, reconstructable state	KV or cache	Useful for speed, unsafe as the only record of a consequential decision.

This division is the spine of an edge-native AI application. A particular Cloudflare product may change, but the ownership question will not.

Reservation before generation, settlement after

Generative work often has a cost that is unknown until it completes. That does not justify charging blindly afterward. A safer pattern is a small financial state machine:

```
authorize → reserve → execute → commit
                              └────→ refund
```

Shared authentication and billing infrastructure makes this pattern much easier to apply consistently. In Stackbilt’s `edge-auth` design, a consumer checks a tenant’s allowance, creates an idempotent reservation before expensive work, and then commits or refunds that reservation at the job’s terminal

transition. The important insight is broader than any implementation: authorization answers “may this actor attempt this?”; a reservation answers “has the system set aside the right to spend for this attempt?” They are different questions.

Design them separately and retry becomes survivable. A worker can receive the same message twice, a client can resend after a lost response, and a provider can time out after doing work. The idempotency key and terminal settlement give the system a way to converge on one business outcome.

Durable Objects are coordinators, not a substitute database

Durable Objects are a strong fit when a workflow needs a single coordinator: one job, one voice session, one collaborative room, one rate-limited resource. They make it natural to keep short-lived coordination state close to the code that advances it and to use alarms for deadlines or scheduled continuation.

They should not quietly become the only inventory of a business. But “Durable Object plus relational record” must not mean two independent authorities for the same transition. Choose one commit point. If the coordinator owns the transition, write a durable event/outbox entry with it and project that event into relational reporting state. If relational state owns the transition, make the coordinator advance only against the committed row version. Give every projection an event ID and monotonic version so retries are harmless and stale writers are fenced out. This split makes reporting, support, retention policies, and backfills less surprising without pretending a cross-service dual write is atomic.

Queue messages are invitations, not facts

A queue delivers a request to try work. It does not prove the work is new, and it does not prove the work is still relevant. A consumer should therefore begin by loading the job and asking a few unglamorous questions:

1. Does this job exist and still belong to the stated tenant?
2. Is it already terminal?
3. Is its reservation valid and its deadline still open?
4. Is this attempt allowed to move the current state forward?

Only then should the consumer call a model or provider. This approach is less clever than inferring intent from a message, but it handles redelivery, duplicate submission, and cancellation without turning incidents into forensic exercises.

A short checklist before adding a model call

Before writing an inference call, answer these questions in the design:

- Can the user receive an accepted job instead of a finished result?
- What state transition authorizes the next side effect?
- Which key makes a retry safe?
- Where does the completed artifact live, and for how long?
- When does reserved quota become a charge, and when does it return?
- Which errors are safe to show to a user, and which belong only in telemetry?

If these questions feel premature, the workflow is probably still a demo. That is not a criticism. It is a useful diagnosis. The next chapter will show how private service composition lets these responsibilities remain separate without forcing every internal call through a public API boundary.

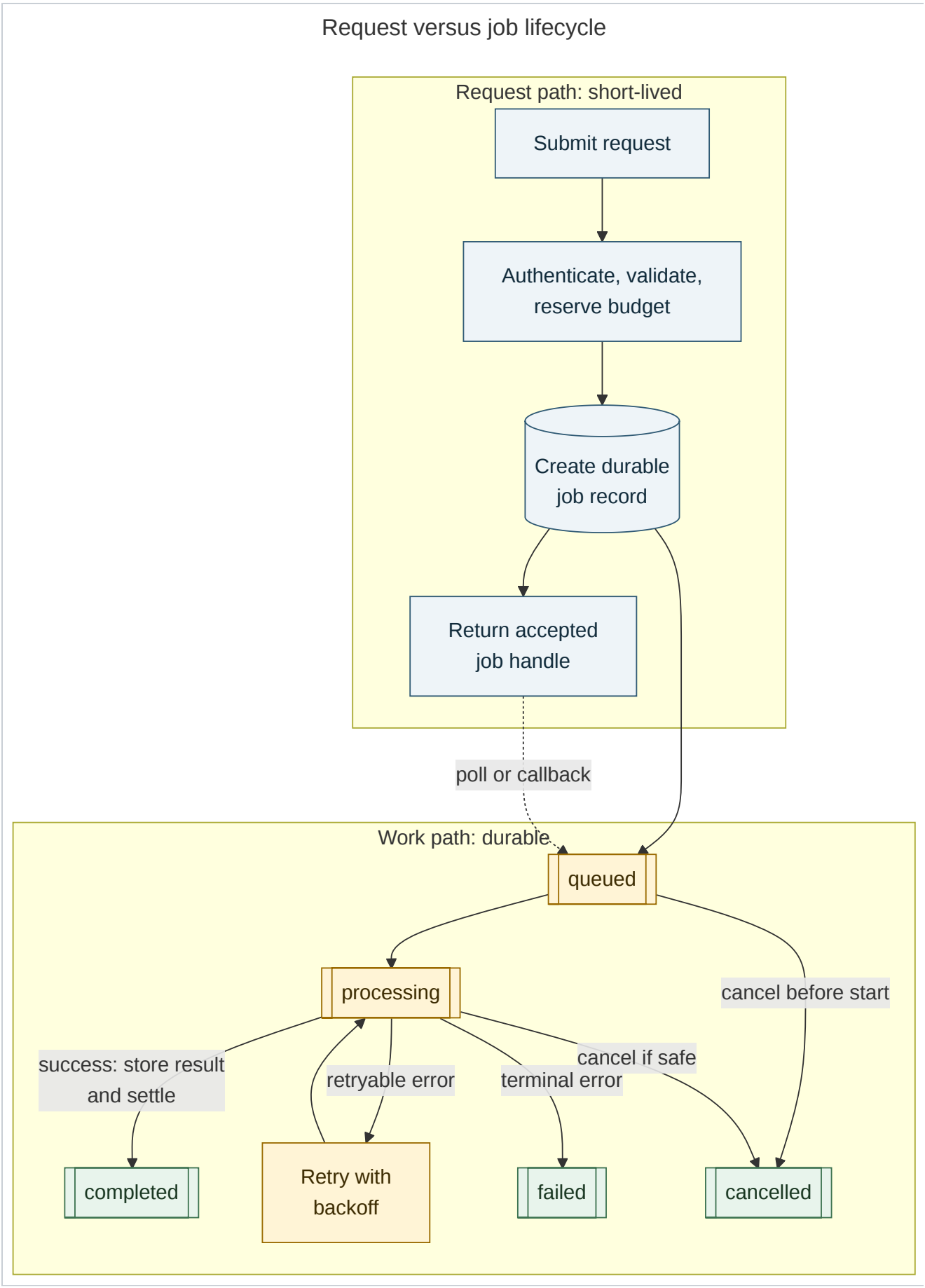


Figure 2. The request delivers intent; the durable job owns the business lifecycle.

3. Service Composition Is a Security Decision

Verified through 2026-08-09. Cloudflare-specific behavior is linked to current official documentation; limits and pricing must be rechecked for a public edition.

Most AI diagrams begin with a gateway and then draw arrows to every dependency. That makes the gateway a place where authentication, business policy, model selection, billing, storage, and tool execution quietly become one program. The portable alternative is to compose services around authority: a component receives only the capability it needs, and its interface names a business action rather than exposing its backing store.

Begin with responsibilities

Role	Owns	Must not own
Edge gateway	identity, validation, admission	every domain's storage and provider secrets
Policy service	entitlement, quota, approval	inference implementation
Orchestrator	job state and next action	a universal user credential
Inference adapter	normalized model call/result	product authorization
Tool executor	a narrow side effect	open-ended model instructions

These are boundaries, not necessarily deployments. Start with modules; separate a component when it needs independent access, release cadence, fault containment, or ownership. Split on authority and lifecycle, not fashion. A caller should ask a billing boundary to `reserveGeneration()`, for example, rather than receive SQL access and invent its own reservation transaction.

Private calls are still contracts

Cloudflare Service Bindings let one Worker call another without a publicly accessible URL, via Worker RPC or forwarded requests. They support private internal services and independent deployments. Service Bindings documentation

That is useful isolation, not complete authorization. A private call identifies a deployed caller; it does not prove which end user authorized an action. Carry a small verified actor context across the boundary: tenant and actor identifiers, resource/action scope, expiry, idempotency key, and correlation

ID—never a reusable provider secret. The receiver validates it and applies its own policy.

```
interface GenerationPolicy {
  reserve(input: {
    decisionId: string;       // resolves server-side to
principal, tenant, scope, and expiry
    estimate: number;
    idempotencyKey: string;
  }):
    Promise<{ reservationId: string; expiresAt: string }>;
}
```

The caller cannot select a tenant by supplying an arbitrary identifier. The receiver resolves the policy decision, binds it to the deployed caller, checks expiry and scope, and derives tenancy server-side. The interface describes a permitted action, not storage plumbing. Narrow contracts make testing, replacement, and incident analysis possible.

One front door, many small interiors

The public gateway authenticates, establishes tenancy, applies rate and size limits, validates schema, and decides whether work is synchronous or a durable job. It should not proxy arbitrary user-controlled URLs and headers into the interior. Behind it, private services can change without changing the public API.

There is still a call-graph budget. Cloudflare documents two distinct ceilings: each Service Binding call consumes the request's general subrequest budget, and one incoming request may traverse at most 32 Worker invocations across its Service Binding call graph. The 32-invocation ceiling is not the general subrequest ceiling. As of the verification date, paid Workers default to 10,000 subrequests per invocation and can configure a higher limit, while free-plan limits differ. Service Binding limits Workers subrequest-limit change Avoid walking a deep graph merely to gather context; aggregate related work and move long fan-out to durable background work.

Do not make the model an integration bus

Tool calling tempts teams to give a model broad access and let it decide how the company works. Separate the decisions instead:

```
may this actor request it?       → policy
what operation is permitted?     → deterministic code
how does ambiguous language map to it? → model, when needed
```

The model may propose structured action. Code validates the schema, checks the capability, requests human confirmation where required, then calls a narrow executor. This makes failures explainable and prevents retrieved text or a prompt from silently becoming authority.

Boundary review

For each new service, name the fact or side effect it uniquely owns; the identities and capabilities it needs; retry/idempotency behavior; typed request and outcome; and safe error categories. Stackbilt's shared authentication and provider routing are a case-study observation: centralizing rules that must remain identical let products replace implementation without recreating security and accounting. The general pattern works on any platform.

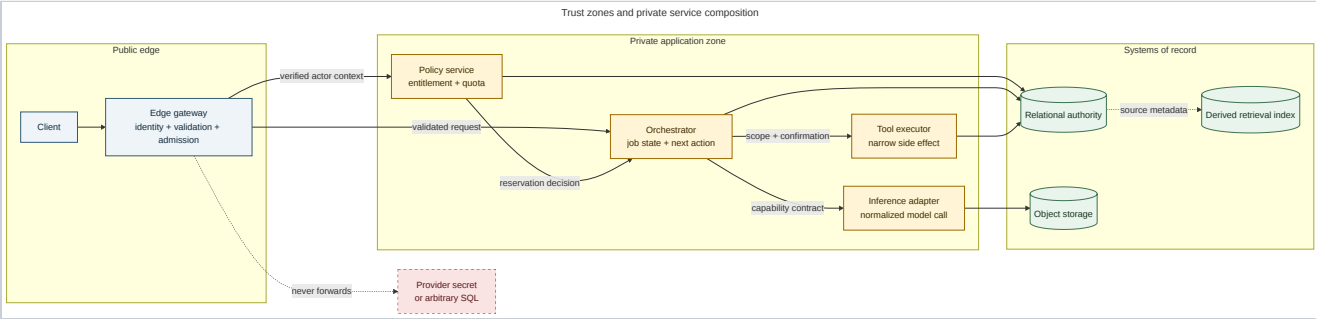


Figure 3. Private composition narrows reachability; explicit actor context and receiver policy carry authorization.

4. State, Locality, and the Honest Limits of D1

Verified through 2026-08-09. D1 behavior below is based on current official documentation. Recheck availability, jurisdictions, limits, and pricing before publication.

The edge does not remove geography. It makes geography visible in where a write is accepted, where a read is served, where an object is retained, and what a user observes immediately after a change. Start with the real question: **which facts require a single order, which tolerate lag, and whose location is part of the product promise?**

Model facts by consistency need

Fact	Consistency question	Typical implementation
Balance, reservation, access grant	Can conflicting updates both win?	transactional relational authority
Job progress	Who advances the state machine?	coordinator plus relational record
Transcript	Is append order important?	sequenced append record
Search index	May it lag its source?	derived asynchronous index
Cache	Can it be reconstructed?	cache/KV, never sole authority

Fast nearby storage is not automatically correct storage for money, access, or irreversible work. Name correctness before optimizing latency.

What D1 provides—and does not

D1 is Cloudflare's serverless SQL database. A database has a primary instance; writes go there. With global read replication, read-only copies replicate asynchronously in available regions, while writes continue to be forwarded to the primary. D1 global read replication A replica may be behind: do not promise an arbitrary replica immediately after a confirmed write.

D1's Sessions API makes a useful read-after-write contract explicit. A session can start at the primary when current data is required, or use a previous bookmark; queries within the session are sequentially consistent, and a bookmark can carry an "at least as current as what I saw" constraint to a later request. D1 Sessions API

```
write job at authority → return job ID + bookmark
status read carries bookmark → replica serves only sufficiently
current state
```

An analytics view may prioritize low latency; a newly created job or entitlement normally must see its own write. These are different product promises. As of the verification date, Sessions and read-replica routing are available through the D1 Worker Binding, not the D1 REST API; REST queries continue to use the primary. D1 read-replication limitations

Locality is a product choice

D1 accepts a best-effort location hint at creation, and jurisdiction settings can restrict where data runs and is stored; replica behavior follows those constraints. A hint does not guarantee the requested location. D1 data location Hint the primary toward the system that pays for write latency, and choose jurisdiction because a commitment requires it. A Worker near a user may still write to a primary elsewhere—trace that latency rather than hiding it. D1 exposes `served_by_region` and primary metadata for this purpose. D1 observability

Keep authority relational and small

Use relational records for identities, policy versions, job lifecycle, reservations, document metadata, provenance, and artifact pointers. Keep large bytes in object storage. Keep embeddings and chunk metadata as a derived index linked to a source document version. A vector hit is never proof that a document remains permitted; fetch and verify source metadata before context assembly.

Make state transitions conditional, rather than inferring ownership from an old timestamp:

```
UPDATE jobs
SET state = 'processing',
    attempt = attempt + 1,
    lease_token = ?,
    lease_expires_at = unixepoch() + 120
WHERE id = ?
  AND state = 'queued'
  AND (lease_expires_at IS NULL OR lease_expires_at <
unixepoch())
RETURNING id, attempt, lease_token;
```

Store `lease_expires_at` as an integer Unix timestamp; do not mix JavaScript ISO strings with SQLite timestamp strings and then depend on lexical ordering. Proceed only when exactly one row is returned. Require the same `lease_token` and attempt/version on every completion update; this fences an expired worker from overwriting its successor. The conditional transition provides the commit point. If a coordinator is also used, it must serialize access to this authority or publish an outbox event from its own authoritative storage—not independently claim the same transition. Recovery and reporting may use different stores, but they must agree which committed record wins.

For every field, document source and derived copies, tenant scope, post-write consistency, location/jurisdiction, retention trigger, model-context eligibility, and migration path. Generative systems create more data; that makes honest state contracts more important, not less.

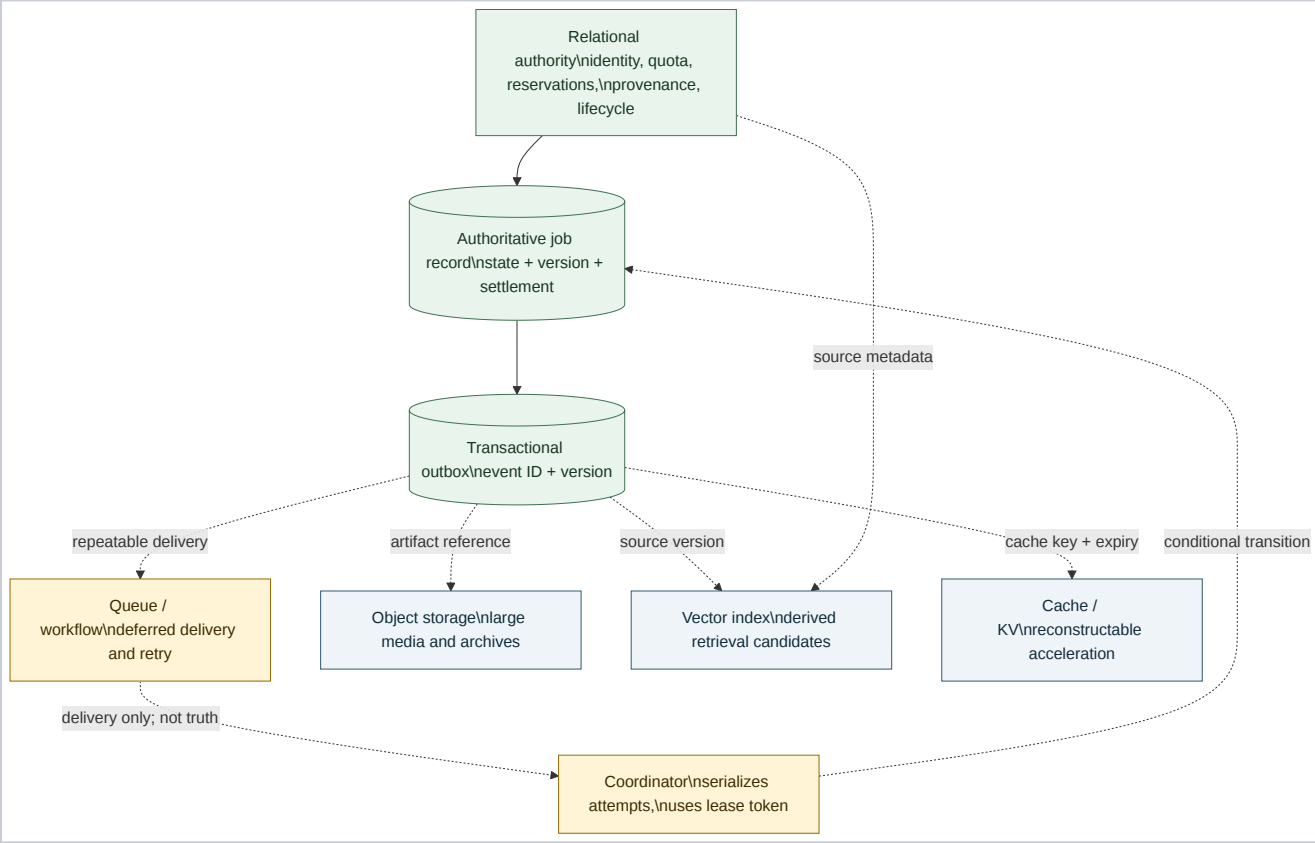


Figure 4. Choose one authority for each fact; treat indexes, queues, and caches as derived or transport state.

Part II — Giving Generative Systems a Spine

5. Route Before You Reason

Verified through 2026-08-09. AI Gateway behavior below uses official Cloudflare documentation. Models and provider features change frequently; pin and evaluate them in your environment.

Every model call spends latency, money, energy, and trust. It should not be the first decision an application makes. Routing is the deterministic control plane around inference: recognize known requests, select a capability, enforce a budget, choose a fallback, and record why that path was admitted.

The lowest capable path

Build a ladder, not one “AI endpoint”:

```
validate → known command/cache → deterministic procedure
         → small classifier/extractor → retrieval-assisted response
         → specialist model → human approval or deferred research
```

A schema conversion, account lookup, or established workflow should be code. A constrained classification may use a small model. Open-ended synthesis can need retrieval and a larger model. High-impact action can need a person. The router should record input class (not necessarily raw input), policy version, selected capability, model/provider version, estimated and actual cost, latency, outcome, and fallback. Without that record, “why did this cost so much?” and “why did it answer that?” have no answer.

Route by capability, not favorite model

Ask for `structured_extract`, `reason_over_citations`, `image_draft`, or `tool_call_with_confirmation`, not “Model X.” A capability contract declares accepted modalities, output validator, quality/latency target, spend ceiling, retry policy, data constraints, and fallback. This lets a product replace an inference supplier without rewriting business logic.

Cloudflare AI Gateway can provide a provider-facing layer: its REST API supports Cloudflare-hosted and third-party models through common endpoints with logging, caching, rate limiting, and related gateway features. AI Gateway REST API Cloudflare’s 2026 AI Platform release positions routing,

caching, attribution, and policy as shared inference infrastructure. AI Platform announcement

That gateway is not product routing. Only the application knows whether a request is a premium feature, approved batch job, support question, or prohibited action.

Retries require semantic safeguards

Failure	Safe response
transport error before admission	retry with idempotency key
provider overload	bounded retry or alternate capability
invalid structured output	repair once, then typed failure
uncertain tool side effect	query operation state; never blindly repeat
weak/conflicting evidence	clarify, defer, or escalate

AI Gateway supports configured automatic retries on upstream failures. Automatic retry on upstream provider failures Use that only when replay semantics are understood: it cannot make a business action idempotent.

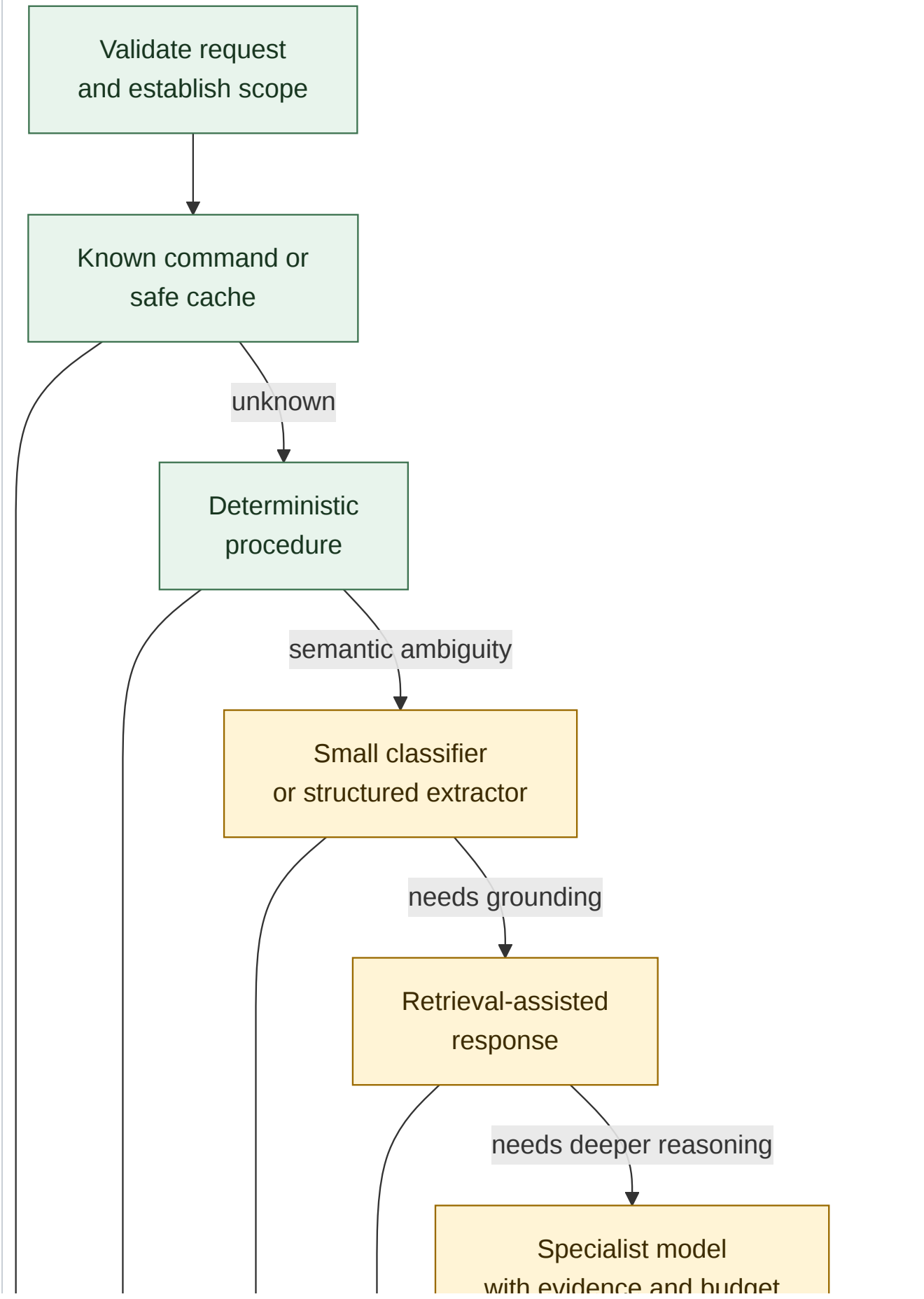
Budgets and evaluation steer the router

Set a deadline, model/image budget, tool-call ceiling, and action budget before work begins. Reserve tenant quota before expensive durable work and settle it only at a terminal state. Keep per-request limits separate from account allowance so a runaway workflow can stop safely.

Treat routing changes as product changes. Maintain versioned evaluations for each capability: representative inputs, schema validity, grounded citations, safety cases, latency, and cost. Run them when changing prompts, tools, models, gateway policy, or retrieval. Stackbilt's AEGIS is a case study for promoting repeatedly successful deterministic procedures into an earlier route, with measured promotion and demotion rules. Repetition should become reliability, not an invitation to keep rediscovering the same work.

Before inference, know actor/tenant/data scope, requested capability, deterministic alternative, budget/deadline/idempotency, eligible evidence, and result validation. That is how experimentation survives contact with users.

Routing ladder



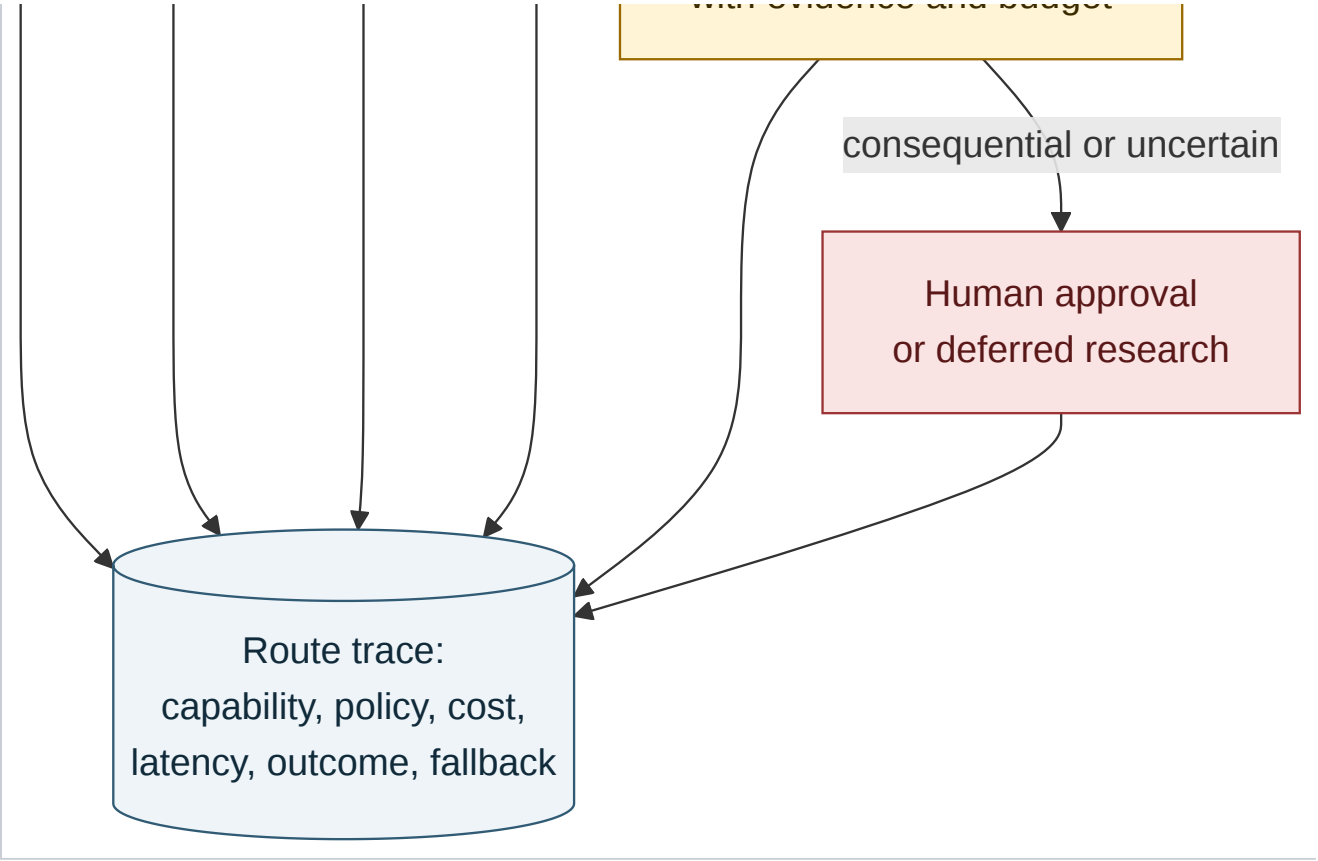


Figure 5. Route before reasoning and escalate only when the task contract requires it.

6. Memory Is a Product Feature, Not a Prompt Field

Verified through 2026-08-09. Cloudflare references use current official docs. Embeddings, service limits, and retrieval quality are workload-specific and must be measured.

“Add memory” often means append an ever-growing transcript to the next prompt. That works until it becomes slow, expensive, stale, cross-tenant, impossible to delete, and persuasive in the wrong way. Memory is a data product: it has ingestion, permissions, lifecycle, retrieval policy, and user-facing consequences.

A retrieved passage is a candidate for context, not a fact, instruction, or grant of authority.

Separate kinds of memory

Type	Example	Correct owner
Working context	current task/recent turns	bounded session state
Episodic record	actions and outcomes	durable audit/event record
Semantic knowledge	documents and chunks	source store plus retrieval index
Procedural knowledge	approved workflow	versioned code/policy
User preference	defaults and consent	user-controlled profile

Do not collapse these into a vector index. Cloudflare Vectorize supports similarity search and RAG-style context augmentation. Vector database documentation It is not a transaction log, permission system, or instruction hierarchy.

Retrieval begins at ingestion

A trustworthy ingestion path retains the original artifact and stable document/version ID; extracts text with a reproducible parser version; chunks by meaning; attaches tenant, ACL, source, timestamp, language, and retention metadata; embeds/indexes; exposes indexing state; and deletes or reindexes derivatives when the source changes.

Each vector record must point to authoritative metadata. Filter for tenant and eligibility before context assembly, then verify source metadata for consequential answers. This prevents a semantically good match from bypassing a permission change or deletion. Keep large bytes in object storage, document metadata in relational state, and only necessary text in model context.

Provenance makes answers usable

For every selected passage, retain document version, chunk ID, rank/score, retrieval-policy version, and transformations. The output should name or link evidence actually used. This lets a user inspect a source, an operator diagnose stale policy, and an evaluation separately score retrieval relevance, citation faithfulness, and answer correctness.

Similarity score is not confidence. It represents a relationship in one embedding space, not whether content is current, authorized, complete, or true. Combine it with metadata, source quality, reranking where appropriate, and willingness to say that evidence is insufficient.

Retrieved text is untrusted input

Documents may contain prompt injections, stale commands, confidential material, or human-directed instructions. Retrieved content must never change system policy or tool authority. Delimit it as reference material, and validate any proposed action against deterministic policy after generation. Internal documents are not exempt: trust is not inherited merely because a connector imported them.

Keep rolling summaries as versioned derivatives, with a source range and generator version, rather than overwriting history. Preserve raw records according to retention policy; regenerate summaries when policy or models change; let users correct or remove preferences. Stackbilt’s MindSpring is a case study for the general principle: memory is useful only when source records remain recoverable and correction is possible.

Evaluate the entire loop with known-evidence questions and adversarial cases: correct but unretrieved documents, similar but unauthorized documents, superseded policy, injected text, insufficient evidence, and a deletion followed by retrieval. Track candidate recall, ranking, citation support, answer accuracy, latency, and cost separately. Trustworthy memory is legible—people can see what was used, correct it, and expect it to disappear when policy requires.

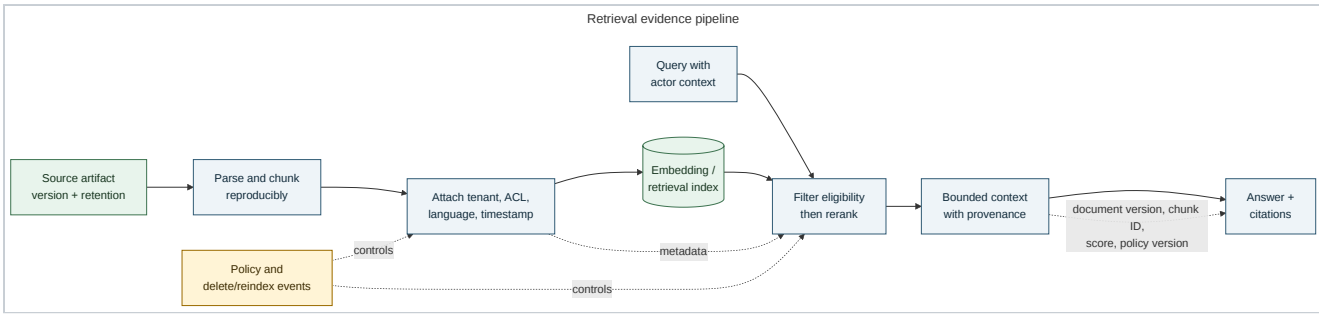


Figure 6. Retrieval proposes evidence; source authority and policy decide whether it may enter context.

7. Durable Objects, Queues, and Work That Outlives a Request

Verified through 2026-08-09.

The useful question is not “can an agent answer this request?” It is “what must remain true after the browser disconnects, the provider times out, or the process is restarted?” A generative system becomes trustworthy when its important work has a durable owner, a durable record, and a retry policy that cannot accidentally repeat an irreversible action.

Start with the failure boundary

An HTTP request is a delivery mechanism, not a transaction. It is an excellent place to authenticate, validate an intent, reserve a budget, and return a handle. It is a poor place to make a long generation, ingest a large corpus, await a human approval, or fan out to fragile vendors. The client may retry. The runtime may end. The user may close the tab exactly when the result becomes useful.

Separate the system into three kinds of work:

Kind	Question it answers	Typical implementation
Coordination	Who is allowed to change this shared thing now?	a keyed state holder
Throughput	How do we absorb a burst and do the work later?	a queue plus idempotent consumer
Orchestration	Which named steps have completed, and what waits next?	a durable workflow

This is a conceptual split, not a product shopping list. A relational database with locks, a broker, and a workflow engine can implement it anywhere. The value is making the ownership and recovery semantics visible in the design.

Coordination: one key, one decision maker

Use a stateful coordinator when many requests contend for the same narrow resource: a conversation, document, tenant budget, rate-limit bucket, approval, or running job. The coordinator serializes the critical decision, stores its result, and records an outbox event in the same authoritative transaction. A

separate delivery step may emit that event repeatedly until a consumer records its event ID. Do not make it a catch-all session cache; keys that are too broad create hot spots, and keys that are too fine fail to coordinate.

Cloudflare Durable Objects are a direct implementation of this pattern: each object has a globally unique name and colocated, strongly consistent durable storage. The platform describes them as stateful Workers that can coordinate clients without the application building its own serialization layer (Durable Objects overview). That makes an object keyed by `tenant:budget-month` or `conversation:id` a good place to atomically decide "admit, defer, or reject." It does *not* make it a substitute for a broad analytical store.

The durable record should hold a state machine, not a vague `status` string:

```
accepted → reserved → running → awaiting_approval → committed
                          └──────→ retryable_failure
                          └──────→ terminal_failure
```

Every transition needs an event ID, actor, time, monotonic version or fencing token, and idempotency key. A retry then asks "has this transition already committed?" before it asks the model or provider to do anything. For charges, messages, publication, and external writes, the idempotency key must also cross the vendor boundary where possible. When it cannot, an ambiguous timeout enters `reconciliation_required`; it is not silently called success or failure.

Throughput: queues are permission to be late

A queue protects the interactive path by turning "do it now" into "we accepted responsibility for doing it." Its consumer must assume at-least-once delivery. That is not an implementation flaw: Cloudflare Queues explicitly uses at-least- once delivery and recommends a unique message ID or idempotency key where a duplicate would matter (delivery guarantees).

Keep three receipts distinct. **Producer acceptance** says Cloudflare accepted a message for delivery. **Consumer acknowledgement** says the queue need not redeliver that message after this attempt. **Business effect receipt** is the authoritative record that the intended charge, artifact, publication, or other effect committed. Only the last can establish business completion; consumer acknowledgement follows it as transport cleanup.

The practical consumer loop is deliberately boring:

1. Validate schema and deduplication key.
2. Load the current job state.

3. Perform one idempotent effect.
4. Commit the effect's receipt, next state, and any outbox event at the chosen authority.
5. Acknowledge the delivered message only after that authoritative commit.

Send failures that can recover—provider overload, temporary DNS error, a rate limit—back with an intentional delay. Send malformed input, revoked authority, and exhausted business budgets to a reviewable terminal path. Configure a dead- letter queue, then make someone own it. A DLQ without an investigation workflow is merely a quieter way to lose work.

Cloudflare Queues supports batching, delayed messages, retries, and dead-letter queues; a failed batch can be redelivered unless individual messages are acknowledged (batching and retries). Choose batch size from the slowest downstream dependency, not from a benchmark. Batching reduces invocations and external writes, but increases the blast radius of a partial failure and the time a small job waits behind a large one.

Orchestration: make the recovery point explicit

Some jobs are neither a single message nor a single critical section. A research report may gather sources, call multiple models, wait for a human, publish a draft, and notify the requester. Represent it as named durable steps. The design rule is simple: put a step boundary wherever rerunning all preceding code would be expensive, unsafe, or confusing.

Cloudflare Workflows persists completed steps, can retry them, sleep, and wait for external events; a later failure can resume after a previously completed step rather than repeat it (Workflow guide). This suits human-in-the-loop AI especially well. "Await approval" is a real state, not a tab left open in an operator's browser.

Retries are policy. Classify errors before choosing them: invalid prompt and denied permission are terminal; an overloaded provider is retryable with jitter; an unknown result from a non-idempotent vendor call requires reconciliation, not blind replay. Cloudflare's default workflow step configuration is documented as five retries, a 10-second delay, exponential backoff, and a 10-minute timeout, but defaults are a starting point—not a business promise (sleeping and retrying).

Case study: an edge generation pipeline

In Stackbilt-style systems, an interactive Worker can validate a request and create a job record; a tenant-keyed coordinator reserves the spend and concurrency slot; a queue isolates generation from traffic spikes; and a workflow coordinates post-processing, review, and delivery. That arrangement is portable. Its crucial property is that the generated artifact, spend reservation, and publication state are independently inspectable—none exists only in a model transcript.

Before adopting it, test four unhappy paths: submit the same request twice; terminate a consumer after the external call but before the acknowledgement; make approval arrive twice; and let the provider return an ambiguous timeout. If the final state is not obvious for each, the architecture is still a demo.

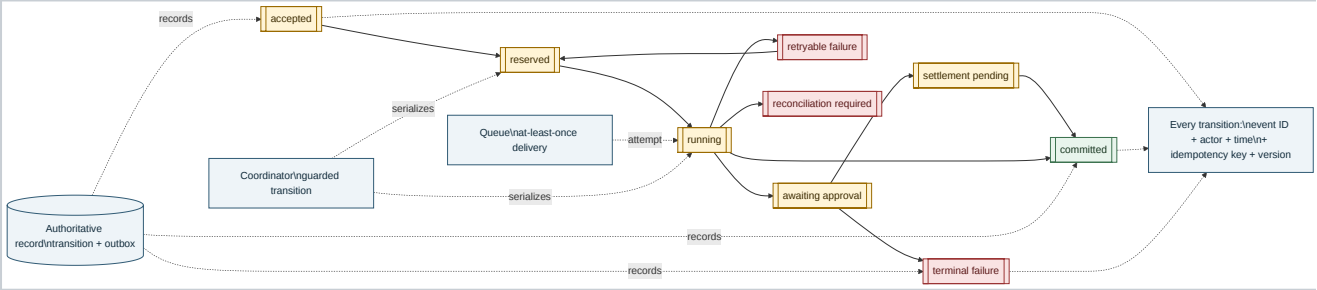


Figure 7. Durable work makes recovery, settlement, and ambiguous effects explicit state transitions.

8. Tools, MCP, and Capability Boundaries

Verified through 2026-08-09.

Models do not “use an API” in the ordinary sense. They propose a tool call based on incomplete natural-language context. That distinction turns tool design into a security and product-design discipline. A tool is not a convenience wrapper around every backend endpoint; it is a bounded capability that says what an actor may do, to which resources, for how long, under what audit trail.

A tool should be smaller than the intention

“Manage the customer’s account” is an intention. `list_invoices(account_id)` and `request_refund(invoice_id, reason)` are capabilities. The latter can enforce a scope, a maximum amount, a confirmation rule, and a durable receipt. The former cannot safely be inferred from a broad administrative API.

Design every tool contract with six fields:

Field	The question it settles
Principal	On whose behalf is this called?
Scope	Which tenant, records, and operation are allowed?
Preconditions	What must be true before execution?
Effect	What changes, or what external request is made?
Idempotency	What happens if the call is repeated?
Evidence	What receipt can an operator later inspect?

Validate structured input at the boundary; derive tenant and identity from a verified session rather than model-provided fields; and return constrained, useful output. Never give the model a raw database query or a broadly privileged HTTP client merely because it is easier to prototype. Read-only search should be separate from mutation. Consequential mutations should normally become a proposed action that needs user confirmation or a policy decision.

MCP is an interoperability protocol, not a trust model

Model Context Protocol standardizes how a client discovers and invokes tools, resources, and prompts over supported transports. It does not decide which user may call them or whether their arguments are safe. Treat every remote MCP server as a new supplier and every tool description as untrusted input to the model. Model Context Protocol specification

The 2026-07-28 protocol revision makes the core request/response path stateless: ordinary requests do not require a protocol session, and a fresh server can handle each request. Multi Round-Trip Requests (MRTR) return an `input_required` result for elicitation instead of depending on a pushed request over an always-open connection. Server-initiated requests are permitted only while the server is actively processing a client request. Application state can still be durable, but it must be explicit rather than hidden in transport affinity. Gateways should authorize each self-describing operation, avoid assuming session pinning, and make retries converge on the same business effect. MCP 2026-07-28 overview

The portable gateway pattern is:

```
user identity → policy decision → capability token → tool adapter → audited effect
                          ↑                          ↓
                  approval / budget          redacted receipt
```

The gateway should expose only the tools relevant to the current task and principal, attach short-lived scoped credentials internally, enforce rate and spend limits, and record both the proposed arguments and the final result. A model cannot be trusted to remember an access rule that was only stated in a system prompt.

Cloudflare's MCP tooling makes the platform split visible. Its Agents guidance allows connections to external MCP servers and advises OAuth or token-based authorization for external tool access (MCP client overview). Cloudflare Access MCP portals can put multiple servers behind one endpoint, tailor tools and prompts per portal, and log tool requests (MCP server portals). Those are useful implementation choices; the authorization policy still belongs to the application.

Agents SDK v0.20.0 negotiates the 2026-07-28 stateless protocol or a legacy connection, supports MRTR elicitation, and deprecates session-oriented `McpAgent` servers in favor of per-request server factories. During migration, run explicit compatibility paths rather than assuming every client or server changed at once. Agents SDK v0.20.0

Context is also an attack surface

Tool catalogs cost tokens and create confusion. More importantly, a retrieved document can instruct the model to misuse a powerful tool. Keep untrusted text in a clearly labeled data channel; do not concatenate it into governing instructions. For high-impact tasks, use a two-stage design: the model produces a structured proposal, deterministic policy code evaluates it, and only then is the narrow capability invoked.

Progressive disclosure helps both safety and performance. Offer a small discovery tool that finds an allowlisted operation, then expose or execute that operation only after policy checks. Cloudflare's portal has `minimize_tools` and `search_and_execute` modes for this purpose, and its Code Mode can replace a large tool list with constrained search and execution in a sandboxed Worker (portal context optimization). Do not confuse lower token use with a security boundary; apply authorization at the eventual execution point too.

The human confirmation is a protocol step

An approval should display the exact effect: recipient, amount, records touched, and a concise explanation. Bind it to the proposed action hash, actor, expiry, and policy version. An "are you sure?" after the mutation is not approval.

For autonomous loops, impose three independent ceilings: number of tool calls, wall-clock duration, and money or resource budget. Stop on repeated tool errors, unexpected scope expansion, or a new permission class. Persist the run trace so an operator can reproduce why an action was possible.

Case study: tools as product APIs

Stackbilt's MCP-facing work treats tools as stable product APIs with schemas, tenant checks, receipts, and distinct read/write paths. The transferable lesson is to make a capability narrower than the user's wish but sufficient for one verifiable effect. That constraint feels slower initially; it is what makes delegation reviewable when the model is wrong, the network retries, or a customer asks what happened.

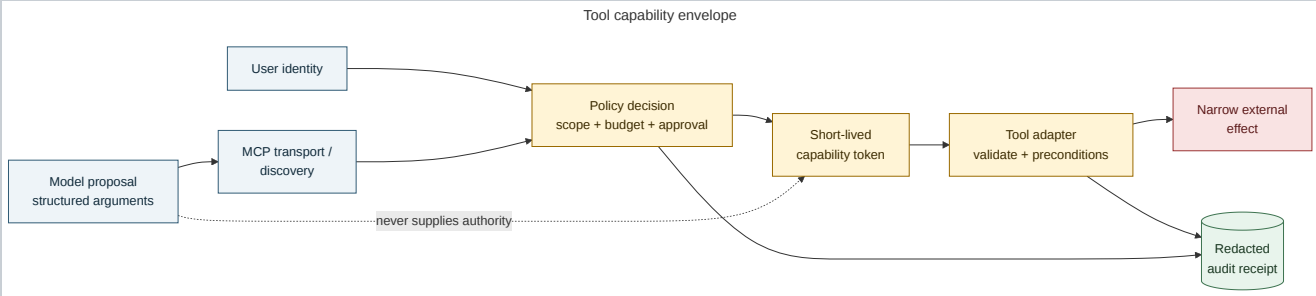


Figure 8. A tool is a bounded capability, not a model-visible copy of an administrative API.

Part III — Operating What You Built

9. Multi-Model Economics and Quality Tiers

Verified through 2026-08-09.

There is no such thing as “the model cost” of an AI product. There is a cost of the whole decision: input tokens, output tokens, retries, retrieval, tools, cache misses, latency, human review, and the damage caused by a poor answer. Optimizing only unit token price frequently moves cost into abandonment, escalations, or unsafe automation.

Price decisions by outcome, not model prestige

Define a quality tier around an observable job requirement. For example:

Tier	Promise	Typical strategy
Instant	Fast, reversible, grounded answer	small model, short context, cache
Standard	Useful answer with citations or structured output	capable model plus retrieval and validation
Deliberate	High-confidence or consequential draft	stronger model, independent checks, possible review
Escalated	A decision that affects rights, money, or publication	deterministic policy and human approval

The tiers are product contracts, not secret model names. They let a user choose latency and depth, let the system set a budget, and let engineering change providers without rewriting the product promise. A “strongest model for every keystroke” policy has no graceful degradation path.

For each job record: selected tier, model/provider version, input and output token counts, retrieval size, tool calls, retries, cache status, latency, direct cost, and outcome score. Then calculate cost per *accepted outcome* rather than cost per request. A cheap classifier that sends many jobs to a costly fallback may still be the correct choice; measure the combined route.

Route deterministically before sampling

Use code for rules that are actually rules: tenant entitlement, data residency, modality, maximum context, prohibited providers, spend cap, and whether an action requires approval. A lightweight classifier can help choose a content class, but the final route should be explainable as a policy result:

```
if sensitive_data: approved_provider_only
else if request_is_retrieval: standard_tier
else if token_budget_remaining < threshold: instant_tier
else: deliberate_tier
```

The router should be able to return “defer,” “ask a question,” or “budget exhausted.” Fallbacks must preserve the contract: a smaller model might create a draft, but must not silently make a policy decision reserved for a stronger, reviewed path. Keep a provider-neutral internal request and response shape so switching model APIs does not corrupt application semantics.

Caching is a product decision

Cache only work whose reuse is safe. A shared cache needs a key that includes prompt template version, model family, tool/retrieval context, authorization scope, language, and any feature flags that change behavior. Never share a tenant-specific answer because two prompt strings happen to match.

Cloudflare AI Gateway provides a cross-provider control point for logging, analytics, caching, rate limiting, retries, and model fallback (AI Gateway overview). Its REST API can address Cloudflare-hosted and third-party models through one surface (REST API). That is valuable for a routing layer, but it does not choose a sensible cache key, customer promise, or privacy classification for you.

Model availability is itself a routing constraint. Cloudflare moved Kimi K2.6, Kimi K2.7 Code, and GLM-5.2 to the Workers Paid plan on 2026-07-28; Kimi K2.5 requests had already begun aliasing to the higher-priced K2.6 on May 30. A provider-neutral route must therefore carry plan eligibility, current model identity, and price verification—not merely assume that a catalog identifier remains available to every account. Workers AI changelog Kimi K2.5 migration notice

Cloudflare's August 2026 AI Gateway updates added unified access and billing for Workers AI and supported third-party models. Identity-aware controls and User Insights can apply model access and spend policy at that shared boundary. Those controls strengthen enforcement, but the application still owns the reservation, quality tier, and terminal business outcome. AI product changelog

Use semantic caching only for low-risk, clearly bounded similarity tasks; it can turn a plausible-but-wrong near match into a confident product defect. For structured transformations, a deterministic normalized-input cache is generally easier to audit. For live information, cache retrieval artifacts with an expiry, not the final assertion indefinitely.

Budgets must be reserved, not observed late

At admission, estimate worst-case consumption from tier, token limits, tool allowance, and retry policy. Reserve that budget against the tenant or user; release the unused portion on completion. A concurrency limit alone cannot stop a small number of exceptionally long prompts. A monthly ledger alone cannot stop many simultaneous requests from spending the same remaining balance.

Apply backpressure before providers do. Cloudflare AI Gateway rate limiting can use fixed or sliding windows and responds with `429` when the configured limit is exceeded (rate limiting). That is a helpful enforcement point, but product-level budgets need identity and business context the gateway may not have.

Evaluate the router as a system

Build a fixed, versioned evaluation set of real task shapes with sensitive data removed. Score: task success, groundedness, formatting validity, route accuracy, latency, cost, fallback behavior, and refusal correctness. Run it whenever a prompt, model, retriever, tool, or routing policy changes. Include adversarial and low-budget cases; otherwise the routing system will look excellent only when it is allowed to spend freely.

The goal of multi-model design is not an elaborate model tournament. It is an application that can state, before it calls a model, what it is willing to spend, what quality it owes, and what it will do when neither is available.

10. Observability: Trace Decisions, Not Just Errors

Verified through 2026-08-09.

An ordinary service can often be diagnosed from an exception and a request ID. An AI service needs to explain a chain of choices: why this route, this context, this model, these tools, this answer, this cost, and this external effect. The model's prose is evidence, not an explanation. The explanation lives in the deterministic system around it.

The trace is the product's flight recorder

Create one trace ID at admission and propagate it across the request, durable job, queue messages, workflow steps, model calls, retrievals, tool calls, and human approvals. Each span should record structured fields, not merely text:

Span	Minimum evidence
Admission	principal, tenant, policy version, requested tier, budget reservation
Routing	candidate routes, selected route, deterministic reasons, fallback rule
Context	source IDs, retrieval query version, document versions, redaction outcome
Model	provider/model revision, token limits and use, latency, cache state, cost
Tool	capability, scoped principal, argument hash, approval ID, receipt, effect status
Delivery	artifact ID, user-visible status, feedback, final budget settlement

Store raw prompts and outputs only when their retention and access policy allows it. Prefer hashes, redacted excerpts, source IDs, and structured metrics in the default operational trace. A log that contains customer secrets is not better observability; it is a future incident. Make sampling explicit for high-volume traffic, while always retaining security events, failures, and evaluated samples within their approved retention windows.

Cloudflare AI Gateway logs can include prompt, response, provider, token use, cost, duration, and DLP outcomes (AI Gateway logging). Payload logging can be suppressed, and identity-aware controls and User Insights can attach and analyze the actor behind spend without making prompt bodies the default audit record. Those fields provide a useful model-call span. Its published storage limits and

retention behavior vary by plan and configuration, so a production design should treat the gateway log as one signal, not its sole audit system (AI Gateway limits). A storage capacity or plan allowance is not a retention guarantee; configure and test the policy the product actually promises.

Measure what users experience

Start with a small scorecard:

- Task success: did the user obtain the intended usable result?
- Groundedness: are claims supported by the permitted evidence?
- Safety and policy: were required refusals, redactions, and approvals applied?
- Operational health: latency by tier, error and retry rate, queue age, workflow completion.
- Economics: cost per accepted outcome, not only cost per token.

Segment every metric by tenant class, route, model, prompt version, retrieval version, and toolset. An aggregate success rate can hide that a new fallback is failing one language or a single customer's document type. Set service-level objectives around end-to-end states: "95% of standard jobs complete within X" is more meaningful than "99.9% of model requests return HTTP 200."

Evaluation is controlled change management

An evaluation set is a versioned collection of inputs, permitted context, expected properties, and scoring rules. It need not require one perfect answer. For a support draft, test citation presence, prohibited claims, required fields, and a blinded preference score. For a classification, compare to labeled truth. For an agent, test its tool plan, authority boundary, and final effect—not merely its fluent narration.

Use three complementary loops:

1. **Offline regression:** run a representative fixed set before release.
2. **Shadow comparison:** send sampled live-shaped work to a candidate path without changing the user-visible result.
3. **Production feedback:** collect explicit corrections and carefully sampled outcome reviews, then turn recurring failures into regression cases.

Every score needs a rubric, scorer version, and known limitations. LLM-as-judge can scale comparative assessment, but use anchored examples, blind the judge to which candidate is new, and audit it with human labels. Never allow a model that benefits from a route to be the only judge of that route.

Alerts should name a decision boundary

“Error rate high” is an alarm. “Fallback rate exceeded 8% after provider X’s latency rose; standard-tier budget burn is projected to exceed the tenant cap” is an operational instruction. Alert on sudden shifts in route distribution, cache hit rate, tool denial, retrieval-empty rate, p95 latency, queue age, model cost, and evaluation score. Link the alert to a trace sample and a rollback control: disable a model, pin a prompt, pause a tool, or lower a tier.

Cloudflare Workflows exposes completed steps, retries, errors, and sleep state when inspecting an instance (Workflow guide). Combine that execution evidence with model-call and application traces so an operator can distinguish “the model was slow” from “the model succeeded but the publish step was never authorized.”

Case study: the answer is not the only artifact

The reusable operating pattern in Stackbilt’s systems is to retain an inspectable decision trail alongside the generated artifact: versioned policy, selected route, source provenance, tool receipts, and budget outcome. It makes incidents smaller and learning faster. More importantly, it gives a customer and operator something defensible when an automated system asks for trust.

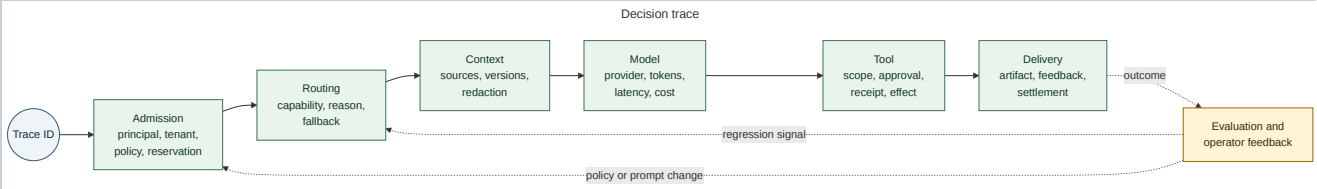


Figure 9. Trace the decisions that produced an outcome, not only the final model response.

11. Identity, Tenancy, Retention, and Consent

Verified through 2026-08-09.

Generative AI can restate private material in a new form. Identity and retention are therefore part of correctness, not a compliance appendix.

An authority tuple

Every consequential operation needs: **principal** + **tenant** + **capability** + **purpose** + **expiry**. Authentication establishes the principal; server-side tenant resolution establishes whose resources may be considered; authorization evaluates capability and purpose; expiry prevents a decision becoming a permanent bearer credential. A prompt saying “only use the user’s documents” is not access control.

Put a policy-enforcing gateway before tools and systems of record. It receives short-lived identity, resolves tenant scope, applies allowlists and quotas, then passes a narrowed credential or authorized result. Never treat a client-provided tenant ID as authority.

Keep a policy ledger

The gateway should be able to explain not only that an operation was allowed, but which version of the policy allowed it. A small policy ledger can carry the decision without copying the entire user profile into every downstream record:

```
type PolicyDecision = {
  principalId: string;
  tenantId: string;
  capability: 'read' | 'write' | 'invoke-tool' | 'model-
context';
  purpose: string;
  policyVersion: string;
  consentVersion?: string;
  expiresAt: string;
  decision: 'allow' | 'deny' | 'review';
  decisionId: string;
};
```

Store the decision ID with the job, retrieval, tool receipt, or provider-bound request that it authorizes. Downstream services should receive the minimum verified context they need and should re-check the decision when the operation is delayed or consequential. A queued job that was legal at submission may be wrong to run after a membership change, consent withdrawal, or policy update.

This ledger also separates three questions that are often accidentally merged:

Question	Owner	Example answer
Who is acting?	Identity	User 42, through session 8f...
What may they do?	Authorization	Read document set A until 14:00
Why and under which policy?	Purpose and governance	Support draft, policy 3.2

The separation matters during incident review. “The request had a valid token” is not enough if the token was issued for a different purpose or outlived the tenant membership that justified it.

Make deletion a graph traversal

Keep account data, user input, derived data (embeddings, summaries, thumbnails), operational evidence, and provider-bound requests as separate classes. Each derivative needs a source-of-truth ID, policy label, and retention rule. A deletion job must tombstone the source, prevent new retrieval, remove chunks, vectors, and caches, neutralize queued work through consumer-side tombstone checks, then record completion or named exceptions. Cloudflare Queues does not provide a safe per-message delete contract; whole-queue purge may still leave in-flight work. Queues pause and purge Large originals fit object storage; Cloudflare R2 provides S3-compatible object operations, but lifecycle behavior must be designed and tested as product policy. R2 documentation

Deletion is a workflow with observable checkpoints, not one SQL statement:

```
requested
  -> tombstoned at source
  -> blocked from new retrieval and processing
  -> derived chunks and vectors removed
  -> caches invalidated and queued attempts refused at
consumption
  -> provider-bound copies reconciled where possible
  -> verified complete or complete_with_exceptions
```

Keep the tombstone until every derivative has either been removed or explicitly classified as an approved exception with an owner and expiry/review date. A queue consumer must check the source record before doing work; otherwise a message already in flight can recreate a deleted vector or artifact. The completion record should include the source version, derivative classes visited, failed steps, and the operator or workflow that resolved them. This makes deletion supportable without retaining the data that the deletion was meant to remove.

Consent has a runtime

Consent is a versioned fact evaluated when data crosses a boundary. Record policy version, purpose, scope, actor, and time. Withdrawal should stop new processing immediately and trigger deletion where applicable; it cannot retract text already delivered or independently retained elsewhere. For tools, distinguish “may read” from “may act.” Validate structured arguments and mint one-purpose, short-lived approval rather than asking a model to guess sensitive parameters.

Under MCP 2026-07-28, elicitation is delivered through a Multi Round-Trip Request: the server returns `input_required`, the client gathers structured input or consent, and the original operation is retried to completion. That retry must reuse the same business idempotency key and re-evaluate policy; the protocol interaction is not proof that the resulting effect was authorized. Legacy Agents connections can still receive pushed elicitation requests during migration. Agents SDK v0.20.0

These patterns are architecture guidance, not a universal statement of what a law requires. Consent, deletion, retention, and legal holds depend on the product, jurisdiction, contract, and accountable reviewer.

Retrieval needs the same discipline. Filter by tenant and document permission before context construction, then re-check permission when resolving returned IDs. Similarity is relevance, not authority. Vectorize supports metadata filtering, but application policy remains final. Vectorize query documentation

Rehearse the uncomfortable paths

Before launch, test the boundaries as product behavior rather than relying on a static permissions review:

- Submit a request with another tenant's object ID and confirm the response is indistinguishable from a missing object.
- Revoke membership while work is queued and confirm the consumer refuses the job before any model, tool, or provider call.
- Withdraw consent after ingestion and confirm retrieval, summaries, vectors, caches, and queued attempts follow the deletion policy, including from a location that may still hold a stale cache

entry.
- Trigger a tool receipt and verify it contains a decision ID, purpose, scope, expiry, and redacted effect details without raw secrets.
- Inspect ordinary traces and confirm they cannot reconstruct sensitive prompts unless an approved retention policy explicitly permits it.

These tests expose the most expensive class of privacy bug: a system that correctly checks the first request but forgets that authorization and retention must continue across queues, indexes, caches, providers, and retries.

12. From Workflow to Agent: Earned Autonomy

Verified through 2026-08-09.

Stackbilt implementation evidence reviewed: 2026-08-08. The examples in this chapter are observations from AEGIS, edge-auth, CodeBeast, and the Visibility Desk design. They support the pattern; they are not claims that every system should copy the same components.

An agent chooses paths under uncertainty; a workflow follows a known path. That flexibility is useful only when choices are constrained, observed, and recoverable.

The autonomy ladder

1. **Suggest:** draft; a person applies.
2. **Prepare:** gather evidence and make a reversible artifact.
3. **Execute bounded work:** act inside a pre-approved scope and budget.
4. **Plan and execute:** choose steps under policy gates.
5. **Delegate:** create subordinate work with explicit limits.

Move up only with representative evaluations, known failure modes, auditability, rollback, and an accountable owner. Good prose is not permission to change a customer record.

The transition between rungs should be a promotion decision, not an intuition. For each capability, keep a small record of the current rung, evidence tested, allowed scope, budget, stop conditions, and owner:

Gate	Evidence required before promotion	Failure response
Quality	Representative inputs meet the task rubric	Keep the current rung; add the failure to evaluation
Authority	Principal, tenant, purpose, and capability are explicit	Deny or request clarification
Recovery	Retry, timeout, duplicate, and partial-effect paths converge	Defer to a durable workflow or human
Observability	Decision, evidence, tool receipt, and outcome are inspectable	Do not promote; preserve a review trace

Gate	Evidence required before promotion	Failure response
Ownership	A named person or team can revoke, repair, and explain the behavior	Keep approval required

The most important row is recovery. A system that performs well in a clean demo but cannot tell whether an external effect happened after a timeout has not earned autonomy; it has earned a reconciliation queue.

The deterministic turn

```
request → authenticate → classify risk → load allowed context → model proposes
        → validate tool arguments → policy/approval → execute → record outcome
```

The executor accepts typed, validated requests, never natural-language commands. An effect ledger stores intent, inputs by reference, policy decision, idempotency key, result, and compensating action. Give each turn externally enforced limits for time, money, tokens, calls, concurrency, and blast radius. AI Gateway can centralize inference logging, caching, and rate limiting; the application still owns authority and business policy. AI Gateway REST API

What the organization has actually built

AEGIS provides a concrete example of autonomy being earned by routing evidence. Its router uses TarotScript's deterministic `classify-cast` path before model fallbacks, records classifier confidence, and keeps a procedure only when its success history clears configured thresholds. A mature procedure can be trusted even when one classification is uncertain; a low-confidence classification without that history escalates instead of silently taking a shortcut. The router also samples classifier grading and records substrate violations rather than hiding them. The portable lesson is to promote repeated behavior through measured evidence, not through a prompt instruction.

The AEGIS dispatch path adds another useful distinction: a self-consistency probe can agree, split, or escalate, while shadow execution can compare a candidate executor without changing the primary result. Exploration remains possible without granting the experiment authority over the user-visible path.

The organization's headless task pipeline applies the same principle to code changes. Documentation, tests, and research can use an automatic safe lane; features, bug fixes, refactors, and deploys are proposed by default and need approval. Non-operator work is isolated and returns through a pull request, so the human merge remains the authority-bearing step.

CodeBeast makes the stopping rule explicit in that runbook: a grounded finding is `proposed`, and applying the `auto-fix` label is the human greenlight. The result is a pull request, not an invisible mutation. Sensitivity classes can block an unsafe fix and leave an honest explanation instead. This is a better autonomy contract than "the agent may edit files" because scope, approval, and rollback are visible in the artifact.

The Visibility Desk design applies the same boundary to client work. A client workspace can receive evidence and proposals, but the current API deliberately has no external-action endpoint. `proposeAction()` creates an AEGIS proposal; it does not change Cloudflare, publish content, or create engineering work. Those effects are reserved for later, resource-specific Gatekeepers after the approval state machine is proven.

Finally, edge-auth demonstrates why budget belongs in the autonomy contract. Consumers reserve quota before expensive work, then commit or refund the reservation using an idempotency key. Its onboarding guidance also records a failure from `img-forge`: swallowing an authorization or quota error allowed generation to continue unbilled. The corrective rule is portable and blunt: an unavailable policy service is not permission to proceed. Fail closed, or surface a reviewable degraded state.

Long work needs durable milestones. Cloudflare Queues are at-least-once by default and do not guarantee order, so tool execution must be idempotent. Queues delivery guarantees A Durable Object is a useful per-entity coordinator; its alarms are also at-least-once. Durable Objects glossary Alarms

Evaluate behavior, not vibes: test denied permission, stale context, tool-schema failures, prompt injection in retrieved text, budget exhaustion, and provider failure. Shadow new tools before widening authority. Independence is earned when it is legible and bounded.

Rehearse before widening authority

Before promoting a capability, run the failure cases that the clean path hides:

1. Send the same request twice and verify that the effect ledger or idempotency key produces one business outcome.
2. Stop execution after the provider call but before the receipt is stored; verify that the next attempt reconciles rather than blindly repeats.
3. Revoke the principal or tenant while work is queued; verify the worker reloads current policy and refuses the stale job.

4. Return malformed structured arguments and confirm the tool adapter rejects them without exposing a raw backend client.
5. Force budget exhaustion, provider failure, and uncertain model output; verify the system downgrades, defers, or asks for approval according to policy.
6. Inspect the resulting trace and confirm an operator can explain why the action was possible, what evidence entered the turn, and how it settled.

These are not merely reliability tests. They are the evidence that determines whether the next autonomy rung is justified. Autonomy is earned when the system can stop as deliberately as it can act.

Part IV — Case Files and Patterns

13. Case File: A Cognitive Kernel With a Circuit Breaker

Verified through 2026-08-09.

Implementation evidence reviewed: 2026-08-08. The behavior described here is drawn from the AEGIS Worker router, dispatcher, procedural-memory helpers, and resilience wrapper. Thresholds and route names are implementation details, not universal defaults.

This anonymized case describes an internal operations assistant as a portable pattern, not a performance claim or a product recipe. Its task was to turn noisy operational messages into next actions without letting uncertainty become an external side effect.

```
input → normalize → classify → retrieve permitted evidence → propose
      → validate → execute or escalate → append outcome
```

The key primitive was a **circuit breaker for uncertainty**. When identity, intent, authority, cost, or evidence could not be established, the system stopped and created a reviewable handoff. The model had a small explicit route set; it did not discover credentials or invent tools. A gateway checked capability and tenant scope before a handler could run.

The routing loop in practice

The current AEGIS path is less like an autonomous character and more like a series of increasingly expensive tests:

1. A domain pre-filter observes whether the message resembles a known area.
2. TarotScript's `classify-cast` attempts a deterministic classification.
3. Workers AI or Groq provide fallback classification when the first path is unavailable or produces an unusable result.
4. Confidence determines whether the result is trusted, verified with a second opinion, or escalated.
5. Procedural memory is checked for a route with enough successful history.
6. The executor registry applies provider, circuit, and capability policy.
7. Dispatch records the result and may run a non-authoritative shadow comparison.

This shape matters because each stage can refuse to pretend it knows more than it does. A classification is not an execution permission. A procedure is not trusted merely because it exists. A provider fallback is not allowed to change the business meaning of the request.

The confidence zones are explicit in the router: high confidence can proceed, the middle band receives a second classification opinion, and very low confidence escalates unless a mature procedure has stronger historical evidence. The router also contains a guard for a known false-positive shape in which an ordinary status question is classified as a heartbeat. Production autonomy includes correcting recurring classifier mistakes, not only adding more capable models.

Promotion requires a track record

AEGIS's procedural lookup uses success count and success rate before replacing the default route. A degraded or broken procedure is bypassed and replanned. When a procedure is trusted, the decision trace records its identifier and the reason it was selected. When it is not trusted, the system chooses a default executor or escalates based on confidence and tool need.

That creates a practical promotion contract:

State	System behavior	Evidence retained
Unknown pattern	Use a default route with a bounded cost ceiling	Classification and route reason
Candidate procedure	Observe outcomes without making it authoritative	Procedure key, executor, result
Mature procedure	Use the learned route when current policy permits	Success/failure counts and latency
Degraded or broken	Replan through the default route	Status transition and fallback
Uncertain result	Probe, split, or escalate	Probe result and final outcome

The dispatcher's shadow path reinforces the separation between learning and authority. A candidate executor can be run against a successful primary result; a cheap plausibility gate rejects truncated responses and obvious provider errors, then records a redacted observation for later rubric-based review. That gate is not a semantic quality judgment, and shadow output does not update procedural memory or qualify for promotion. The shadow result never replaces the primary response. Exploration remains downstream and non-authoritative until representative evaluation or real product outcomes provide evidence.

The same system records substrate violations and samples classifier grading. Those are not decorative metrics. They test whether the routing design is still true in production. If the deterministic classifier starts making inference at the wrong stage, or confidence stops predicting route quality, the evidence should demote the claim rather than be averaged away.

Three lessons transfer. First, record the decision trace, not only the answer: request, available policy and tools, evidence IDs, proposed effect, decision, outcome. Redact payloads or retain them by reference. Second, make the fallback boring: answer with no private context, defer, or escalate; do not hallucinate through an outage. Third, treat memory as evidence with provenance, never proof.

The case is intentionally not a claim that a classifier can make an agent safe by itself. AEGIS still needs identity, capability checks, budget controls, and operator ownership at the effect boundary. Its contribution is narrower and more useful: make uncertainty observable, make repetition measurable, and let the system earn a cheaper or more autonomous path one verified outcome at a time.

On Cloudflare this can map to a Worker admission layer, a Durable Object per live turn, Queues for deferred work, R2 for artifacts, and AI Gateway at the inference boundary. The mapping is optional; the invariant is one authoritative state transition before each side effect. SQLite-backed Durable Objects provide transactional, strongly consistent storage private to one object, useful for coordination but not a replacement for a product system of record. SQLite-backed Durable Object storage

The system became more useful when it could say “I need approval” or “I cannot verify that.” Stopping is a designed success outcome.

14. Case File: Multi-Tenant Generation Without a Fragile Request Path

Verified through 2026-08-09.

Implementation evidence reviewed: 2026-08-08. The details below come from img-forge's gateway, orchestrator queue consumer, Durable Object state holder, schema, tests, and operator runbook. Model names, thresholds, and queue wiring can change; the ownership and recovery pattern is the portable evidence.

Image generation is a stress test: a user expects an artifact, but no system can honestly promise that every upstream model responds quickly, cheaply, or exactly once. This anonymized field pattern is **accept, reserve, generate, settle, deliver**.

```
client → authorize → create job + reserve → queue
worker → reload job → generate → store artifact → settle →
notify
```

The public request validates identity and policy, normalizes input, creates an idempotency key, reserves allowance, and returns a job ID. A worker reloads the authoritative job before work, declines stale/cancelled/terminal work, then generates and settles. The job is truth; the queue is transport. Cloudflare Queues provide at-least-once delivery and do not guarantee ordering. Queues delivery guarantees How Queues works

What the implementation makes explicit

The job schema gives each generation a tenant, state, idempotency key, reservation reference, model identity, input references, and output reference. The idempotency key has a unique index, while the tenant remains part of the resource lookup and asset path. This prevents a client retry from becoming a second business job and keeps an output address from becoming a cross-tenant capability.

The asynchronous path exposes a transaction boundary that must be handled explicitly. Provider execution, R2 storage, coordinator state, D1 reporting, quota settlement, and queue acknowledgement cannot be one atomic transaction. A hardened logical sequence is:

```
load message → claim attempt with lease token → load inputs
→ call selected provider or reconcile its idempotency key
→ validate artifact → write content-addressed R2 object
→ commit artifact receipt + settlement_pending + outbox event at
one authority
→ settle reservation idempotently → commit completed →
acknowledge
```

The implementation sequence reviewed for this case historically exposed the dangerous version of this boundary by transitioning coordinator and D1 state to `completed` before quota settlement. A crash or swallowed settlement error could therefore produce a completed-but-unsettled job. The portable correction is to keep completion pending until settlement succeeds, or mark the job `reconciliation_required`. A queue acknowledgement is not a completion receipt. On redelivery, the consumer reloads the receipt/outbox record and resumes the missing step instead of regenerating the artifact or inferring truth from the message.

The orchestrator also rejects work that does not belong to its consumer. A video message reaching the image consumer is acknowledged as a misroute rather than silently interpreted as an image request. This is a small but important form of capability discipline: a queue binding should not broaden a worker's meaning.

Separate bytes, state, and policy. Generated media lives in object storage; ownership, content type, retention deadline, and integrity metadata live in the job record; a coordinator owns live transitions; identity and billing own membership and allowance. R2 is S3-compatible storage suitable for bulky inputs and outputs, with explicit access and lifecycle policy. Cloudflare R2

Artifact validation is billing correctness

The current queue consumer rejects an image buffer smaller than 10 KB as a blank or incomplete frame. The guard began as protection against a provider-specific blank output and was widened to cover transmission corruption and moderation placeholders across providers. A rejected artifact is allowed to enter the retry/failure path instead of being billed as a successful generation.

The consumer stores successful bytes under a SHA-256-derived tenant-scoped R2 key and records the key, model identity, and job ID in the durable record. A secondary CDN upload is non-fatal because the R2 object remains the source for delivery fallback. That separation means a delivery enhancement can fail without erasing the generated artifact or falsely marking the job as absent.

The tests encode this reasoning in small executable checks: a buffer below the threshold is rejected, the boundary value is accepted, and a representative generated image passes. The test is not proof that every bad image is detected; it is a named invariant that can be revised when new provider failures appear.

The operational history also shows why this boundary must be tested rather than assumed. The runbook records queue-delivery watchdog behavior for jobs that stay queued after a redeploy, tenant ownership checks for source jobs used in img2img, and explicit `503` behavior when the authorization service is unavailable. These prevent a lost message, cross-tenant source reference, or failed policy check from becoming an unpriced or unauthorized side effect.

Request capabilities—size, quality tier, safety, latency—not a provider model string. A routing layer chooses eligible providers, records the choice, and returns stable product errors. AI Gateway can supply shared logging, caching, rate limiting, and routing features, but cannot decide a product’s quality policy or settle customer quota. AI Gateway REST API

This pattern transfers to video, document conversion, enrichment, and code generation: charge around terminal business states, not network attempts; treat creation as a state machine; retain provenance; delete outputs and derivatives together.

The transferable boundary is therefore not "use an image queue." It is: accept only after identity and budget are established; make the job the source of truth; make every worker reload and validate that truth; treat provider output as untrusted until it passes product checks; and settle money only around a terminal business outcome. That is what turns an expensive model call into an operable product capability.

15. Case File: Ingestion, Retrieval, and the Cost of Memory

Verified through 2026-08-09.

Implementation evidence reviewed: 2026-08-08. The details below are drawn from the MindSpring Worker, upload routes, ingestion queue, Vectorize adapter, stream parser, tests, and README. Limits and model names are current implementation facts, not permanent platform guarantees.

“Memory” can mean document search, conversation continuity, organizational knowledge, preferences, or audit evidence. Combining them in one vector index makes each harder to govern. This generalized case study’s portable lesson is: **memory is a pipeline with a deletion path**.

```
source → validate/scan → extract → chunk → embed → index →
verify → available
              └──────────────────────→ reject/quarantine
```

An upload begins as an immutable source record: owner, tenant, media type, hash, location, policy label, and retention rule. Extraction creates versioned fragments; embedding creates derived index entries pointing back to fragments. Queuing stages makes malformed or expensive input non-interactive and reprocessing ordinary. Improve a chunker or embedding model by writing a new version and selecting it deliberately.

MindSpring makes the pipeline visible in its request boundaries. Uploads require an `ingest` scope, while search and chat require `read`; administrative key management is a separate scope. Small files can use a direct path, while larger files use multipart upload to R2 in 50 MB parts. R2's multipart API supports resumable large-object uploads and requires parts of at least 5 MiB except for the final part (R2 multipart uploads). The Worker does not buffer the whole export in memory. It streams the source from object storage, parses one conversation at a time, and persists progress after each batch of 100 conversations.

That checkpoint is more than a performance optimization. Queue redelivery can resume from the last recorded batch rather than guessing whether the entire export was processed. A malformed conversation can be rejected or quarantined without invalidating every successfully indexed conversation before it. The stream parser tests exercise small input chunks because production streams do not promise convenient boundaries.

The source and index are deliberately split. MindSpring keeps full text in KV and stores vectors plus bounded metadata in Vectorize; its adapter notes a 10 KiB metadata limit (Vectorize metadata filtering), so the vector record carries identifiers and previews rather than becoming a second unbounded document store. A query embeds the question, retrieves the top candidates, hydrates source text, and returns citation records that identify the conversations used.

That first implementation has an important consistency limit: Workers KV is eventually consistent, so a recently changed or deleted conversation may remain visible at another location for the cache TTL or longer. Workers KV consistency A production retrieval path must therefore check an authoritative permission and tombstone record after vector selection and before using hydrated KV text. KV can hold the derived payload; it cannot be the final authority for deletion or access.

At query time, authorize first, filter by document scope, retrieve candidates, then construct small cited context. Retain source and version IDs with the answer. Similarity is not truth. Cloudflare Vectorize supports vector search and metadata filtering for candidate selection; authoritative authorization and lifecycle remain application concerns. Vectorize query best practices

The implementation also shows the limits of a first useful system. Its telemetry envelopes use a seven-day TTL, API keys are revoked rather than silently erased so the audit trail remains meaningful, and the chat prompt instructs the model to say when retrieved context is insufficient. Those are good boundaries, but they do not eliminate the need for a full derivative deletion workflow, source-version tracking, and authorization rechecks as the product grows.

Memory costs more than embeddings: storage, extraction, re-indexing, queries, reranking, context, backups, and review. It also has a staleness cost. Attach freshness and authority metadata, prefer primary sources, and keep summaries as navigation rather than sole records. For each collection define who may ingest, whether it may reach model context, its retention, deletion semantics, and accountable owner.

The cost ledger should therefore count at least five stages: bytes retained, CPU and parsing time, embedding requests, retrieval and hydration, and model context. A cheap vector query can still be an expensive product feature if it forces repeated re-embedding, large source hydration, or long reasoning contexts. Measure useful cited answers per unit of storage and inference, not only vector-query latency.

The portable lesson from MindSpring is not “put conversations in Vectorize.” It is to make ingestion resumable, retrieval attributable, scopes explicit, and memory deletable before the index becomes the product's hidden database.

16. The Production Checklist

Verified through 2026-08-09.

This checklist reflects the failure patterns documented across the Stackbilt systems: lost queue messages, cross-tenant resource lookups, swallowed quota errors, stale generated artifacts, and proposals that accidentally become external actions. Treat each item as a question requiring an artifact or a rehearsal, not a checkbox answered by architectural intent.

This is a launch conversation, not a maturity score.

Contract and state

- Is work a request, job, session, or workflow with explicit cancellation and terminal states?
- Is there one authoritative record and an idempotency key for every effect?
- Can duplicate delivery, timeout, partial completion, and retry converge safely?
- Are queues treated as repeatable transport rather than truth?

Queues are at-least-once by default, so duplicate-safe consumers are baseline engineering. Queues delivery guarantees

Identity, models, and tools

- Is tenant scope resolved server-side for reads, retrieval, and tools?
- Are prompts, source data, derivatives, and telemetry separately classified?
- Can deletion reach artifacts, vectors, cache, and queued work?
- Does every tool receive least authority, purpose, and expiry?
- Are capability, budget, tool schema, provider failure, and injection evaluated?
- Are time, money, tokens, calls, concurrency, and blast radius enforced outside the model?

Reliability and operations

- Can long work return an accepted job rather than an implausible finished response?
- Is large data out of queues and model context?
- Are downstream failures clear states rather than fabricated output?
- Can operators trace a result to policy, context/source IDs, model, tool decision, and cost?
- Are sensitive payloads redacted from routine telemetry?
- Has the team rehearsed revoking a tool, provider key, tenant, and bad index?

If a Durable Object coordinates work, persist progress: alarms are at-least-once with bounded automatic retries. Durable Object alarms

The final test is: what happens if this succeeds twice, after cancellation, after permission revocation, or with yesterday's policy? If that answer is unclear, the design needs another pass.

Evidence to attach to the review

For each launch, keep a small packet that lets another operator reproduce the decision:

Area	Evidence artifact
Admission	Request schema, tenant-resolution test, and denied-scope result
State	State diagram, transition predicates, and duplicate-delivery test
Budget	Reservation/settlement trace for success and failure
Retrieval	Source IDs, permission filter, citation sample, and deletion test
Tools	Capability schema, approval receipt, and rejected-argument test
Reliability	Provider timeout, queue redelivery, and coordinator recovery result
Operations	Trace sample with policy, route, model, cost, and final outcome
Ownership	Named responder, rollback control, retention owner, and review date

The packet should include negative evidence. A route that was denied, a job that was refused after revocation, and a malformed tool call that never reached the backend demonstrate more than a successful happy-path screenshot. If a claim cannot be backed by a test, trace, source record, or named owner, label it as an open risk rather than silently promoting it to "done."

Release stages

Use three explicit decisions:

1. **Canary:** limited tenants, low budgets, verbose traces, and manual review of terminal outcomes.
2. **Reviewer release:** real workflows with a named cohort, rollback ready, and unresolved failures converted into issue-sized work.
3. **Broad release:** evidence from the reviewer cohort, citation and privacy pass complete, operational owner assigned, and no known fail-open boundary.

This staging keeps a technically impressive demo from becoming a public promise before its deletion, billing, and recovery semantics have been exercised.

Part V — The Organizational Agent Platform

17. Cloudflare OS: Context, Gatekeepers, and the Deterministic Turn

Verified through 2026-08-09.

Stackbilt implementation evidence reviewed: 2026-08-08. The organizational pattern below is compared with the AEGIS Visibility Desk design and governance documents. Cloudflare OS product statements remain sourced to Cloudflare's public material; Stackbilt behavior is labeled as an implementation observation.

Cloudflare OS is an open-source organizational AI platform announced by Cloudflare on August 5, 2026. Its importance is not that every organization should adopt it. It makes a necessary architecture visible: curated context, governed gateways to systems of record, isolated application execution, and a shared inference boundary. Cloudflare OS announcement How Cloudflare uses AI with Cloudflare OS

The platform insight

Individual assistants with direct integrations are easy to start and hard to govern at organizational scale. The transferable pattern is to move durable controls out of prompts and into shared infrastructure:

```
people and apps
       ↓
identity + policy + curated context + skills
       ↓
inference gateway + deterministic turn admission
       ↓
authenticated systems of record / isolated generated
applications
       ↓
provenance, cost, audit, and feedback
```

A skill is a bounded capability. Context is curated and attributable. Access to a business system is authenticated and policy-checked. Generated applications receive a narrow execution surface. Inference carries attribution and policy regardless of model.

Cloudflare’s composition

Cloudflare’s published design uses AI Gateway for inference policy and attribution, authenticated gateways to systems of record, Dynamic Workers to isolate generated applications, and Durable Object Facets for stateful application patterns. These are described implementation components, not a requirement or platform guarantee. Cloudflare OS announcement

Dynamic Workers make an important boundary concrete: untrusted or model-generated code should not run inside a privileged application process. Prohibit it or use an isolated runtime with explicit bindings, egress policy, resource limits, and audit. Isolation narrows authority; it does not eliminate denial of service, data exfiltration through allowed egress, vulnerable dependencies, or confused-deputy bindings. The same principle applies with containers or another sandbox. Dynamic Workers

AI Gateway offers a common endpoint and, as of August 2026, unified access and billing for Workers AI and supported third-party providers, with logging, caching, rate-limiting, identity-aware controls, and spend insights. It can centralize operational policy, not replace application authorization, evaluation, or accounting. AI Gateway REST API AI product changelog

The organizational boundary in practice

The AEGIS Visibility Desk is a concrete small-scale version of this pattern. AEGIS owns the client workspace, audit evidence, findings, proposals, and decision receipts. The client-facing Cloudflare OS workspace is an isolated approval surface, not the system of record for the evidence or governance.

The first capability surface is intentionally narrow: retrieve a workspace brief, record an authorized audit, and propose an action. The proposal API does not change Cloudflare configuration, publish content, or create engineering work. Those effects are reserved for later resource-specific Gatekeepers after the approval state machine is proven. A workspace grant is named and private; it is not an ambient grant to every client or every AEGIS tool.

This boundary gives the deterministic turn a practical shape:

```
client workspace → scoped evidence brief → proposal
                 → client decision receipt → resource-specific adapter
                 → external effect, if separately authorized
```

The design also states what it does not promise: no ranking or citation guarantee, no automatic DNS/WAF/configuration change, and no source-code change without approval. Those negative claims are part of the product contract. They prevent a capable context window or a convenient connector

from becoming authority by implication.

Scale it down

A small product can apply the same model with one Worker admission endpoint, an identity provider, typed tool registry, job table, object storage, and one inference gateway. The service count is irrelevant. Ask instead: what context entered this turn; which skill acted under which principal and purpose; which code had which bindings; which model and policy served it; and where is the durable outcome?

Cloudflare OS points to the **deterministic turn**: identity is verified, policy selects context and tools, the model proposes, gatekeepers admit effects, and durable records capture outcomes. Recent releases make that composition easier; none remove the obligation to decide who may act, what may be remembered, or how failure recovers. The edge can think. It should also know when to stop, ask, and leave an auditable trail.

The organization’s current sequence is therefore deliberately conservative: qualify the client, create one isolated workspace, ingest evidence, generate a brief from fresh source IDs, create proposals, record approval or rejection, and only then add a narrowly scoped handoff integration. A generic agent tool catalog would be faster to demo but would erase the distinction between observing evidence, proposing work, and executing a side effect. Cloudflare OS is most useful here as an approval and context surface around a governed system, not as a replacement for the governed system.

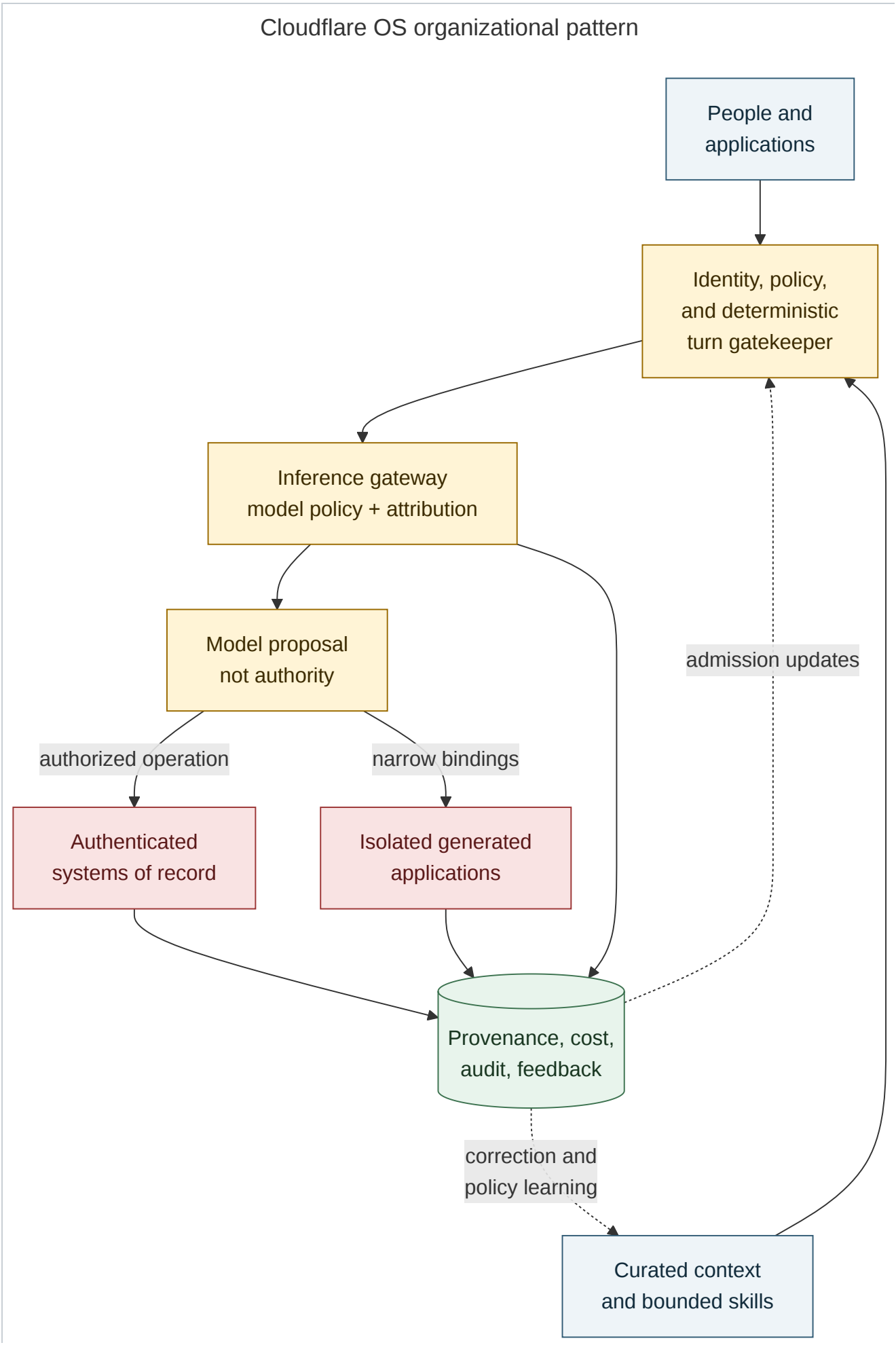

Cloudflare OS organizational pattern
People and applications
Identity, policy, and deterministic turn gatekeeper
Inference gateway model policy + attribution
Model proposal not authority
admission updates
authorized operation
narrow bindings
Authenticated systems of record
Isolated generated applications
Provenance, cost, audit, feedback
correction and policy learning
Curated context and bounded skills

Figure 10. An organizational agent platform centralizes context and policy while gatekeepers retain authority over effects.

Appendices

Appendix A — A Reference Architecture for Generative Work

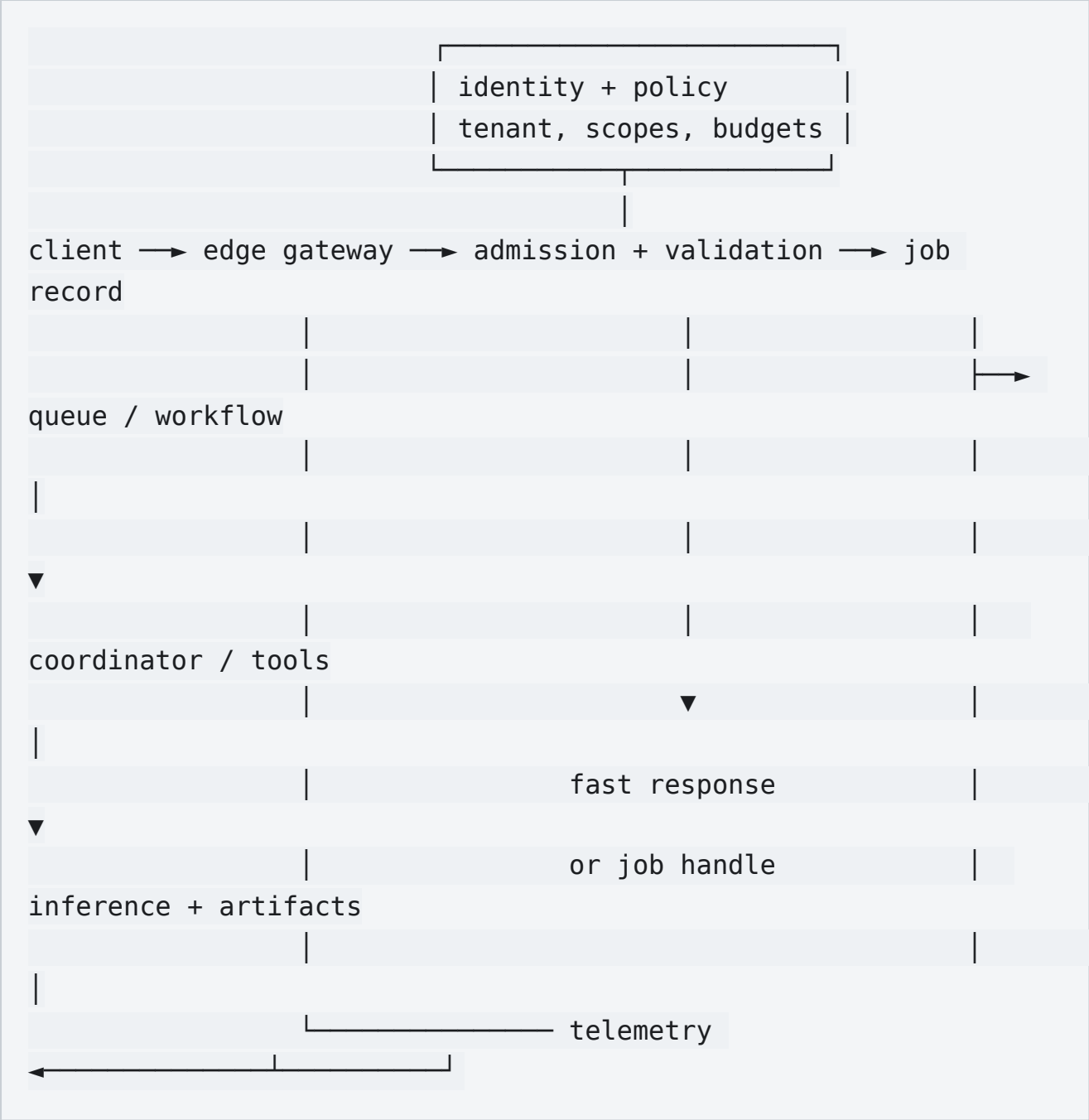
Verified through 2026-08-09.

This reference architecture is intentionally small. It describes the minimum separation of concerns for a system that accepts a user request, may invoke generative inference, and may cause a consequential side effect.

The editable visual source is reference-architecture.mmd. Use it for the one-page figure export; the diagram below remains a compact text fallback for readers and source formats that do not render Mermaid.

The exported reference figure shows the **relational-authoritative variant**: the job row and its transactional outbox are the single commit point; a coordinator advances only against the committed version. A second valid design makes one Durable Object's transactional storage and outbox authoritative and projects versioned events into D1 for reporting. Do not combine the two variants into independent transition authorities.

Variant	Commit point	Outbox location	Projection and stale-writer fence
Relational-authoritative	Conditional D1 job-row transition	Same relational transaction	Coordinator presents the committed row version and cannot overwrite a newer version
Coordinator-authoritative	Durable Object storage transaction	Same object transaction	D1 consumes event ID plus monotonic version and rejects duplicate or stale projections



The responsibilities

Edge gateway. Authenticate, validate, set a request identity, and apply cheap admission controls. It should not conceal a multi-minute job behind a single response.

Identity and policy. Resolve the acting user, tenant, scopes, data policy, and spend authority. This is an independent decision from model selection.

Job record. Persist the request's business identity, state, idempotency key, reservation, result references, monotonic version, and reconciliation status. It is the source of truth for work; a message is only a delivery mechanism. If a coordinator commits transitions in its own storage, its transactional outbox is the authority and this record is a versioned projection, not a second independent truth.

Queue or workflow. Defer and retry work without treating redelivery as a new business event. Cloudflare Workflows is designed for programs whose steps survive failure and can wait across long intervals; use it when the workflow is the durable unit rather than merely a background message. Workflows overview

Coordinator. Serialize the state transitions that must not race. A Durable Object is often suitable for a live session, a job coordinator, or a scarce resource, but it should not become an undocumented replacement for a system of record. Durable Objects documentation

Tool boundary. Expose narrowly scoped operations, validate each invocation, and preserve the user/tenant context. Model Context Protocol is an interoperability and discovery protocol, not an authorization system. Its 2026-07-28 stateless request model removes protocol-session affinity but does not remove application identity, state, or idempotency requirements. MCP on Cloudflare

Inference boundary. Select a model based on capability, policy, price, latency, and evaluation evidence. AI Gateway can provide a central boundary for provider access, observability, caching, and controls, while the application still owns semantic routing. AI Gateway documentation

Artifacts and evidence. Store large outputs outside relational state; record references, hashes, retention class, and provenance. Store the decision trace required to explain a result without persisting more personal data than the product needs.

Minimum viable invariants

Before launch, ensure that:

- a retry cannot create a second charge or duplicate external side effect;
- every tool operation has an authenticated principal and scope;
- every terminal job is either settled or explicitly marked for reconciliation;
- every cross-store projection is replayable from an event ID and rejects stale versions;
- a provider error never becomes the user-facing source of truth;
- a model, tool, or retrieval result can be traced to its request and policy;
- deletion and retention rules cover prompts, artifacts, logs, and embeddings.

The implementation can begin with one Worker, one database, one queue, and one model. The boundaries matter more than the number of services.

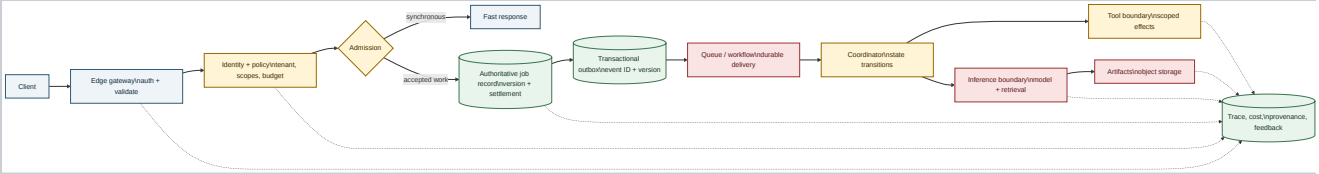


Figure A. Relational-authoritative reference architecture with one commit point and replayable projections.

Appendix B — Design Review Questions

Use these questions in an architecture review, a pull request for a new model call, or a pre-launch checklist.

Admission and identity

- What is the caller allowed to do, and which policy produced that answer?
- Is tenant identity explicit through every internal hop?
- What is the idempotency key, and who creates it?
- Is any irreversible work started before quota, budget, or consent is checked?

State and workflow

- What is the authoritative record of this work?
- Which state transition allows each side effect?
- What can be retried safely, and what needs reconciliation instead?
- Does a queue message represent intent, or only an invitation to inspect intent?

Models and retrieval

- Could deterministic code, a cache, or a proven procedure handle this request?
- What capability—not vendor name—does the chosen model need?
- How will you detect a quality regression after a model or prompt change?
- Is retrieved text evidence, a suggestion, or an instruction? Is that clear to the model and the user?

Tools and agents

- Does each tool expose the minimum authority necessary?
- Is user confirmation required for this action, and can the system collect it structurally rather than infer it from prose?
- Can a generated program run only in an isolated execution boundary?
- Can the agent’s work be paused, resumed, cancelled, and audited?

Operations and data

- Which decision, cost, provider, tool result, and error category are logged?
- What is the retention period for prompt data, artifacts, embeddings, and logs?
- How will an operator explain a wrong answer without reconstructing it from scattered console output?
- What happens when the model provider, queue consumer, or coordinator is down?

If the answer to a question is “the prompt tells it not to,” the design is not finished. Move the rule to a boundary that can enforce it.

Publication Notes

Methodology Note

This book combines portable architecture guidance with field observations from Stackbilt systems. The two evidence types are deliberately separated.

Evidence sources

- Internal repository code, tests, runbooks, and READMEs were used for the AEGIS, img-forge, MindSpring, CodeBeast, edge-auth, and Visibility Desk observations.
- Cloudflare documentation, dated changelog entries, and Cloudflare blog posts support claims about platform behavior and releases.
- The manuscript's diagrams are conceptual unless a caption says otherwise.
- No benchmark, customer outcome, or performance claim is included without a workload, environment, comparison, and date.

Review rules

Platform facts receive nearby primary citations and a verification date when they can change. Internal observations are labeled as case studies, not guarantees. Secrets, customer data, private endpoints, credentials, and raw production payloads are excluded. Product-owner approval is required for named internal systems and implementation details.

The `0.2.1` revision incorporates one repository-grounded technical review and three additional frontier-model reviews of the immutable `0.1.1` artifacts. Findings were checked against the manuscript source, export pipeline, implementation evidence, and current first-party platform sources before they were accepted. Model agreement is not independent empirical proof; the public `1.0` gate still requires a practitioner cohort.

Limits of the evidence

Repository evidence shows how a system is implemented or documented at the review date; it does not prove universal reliability, security, compliance, cost, or platform guarantees. A passing test demonstrates a named invariant, not that every failure mode has been eliminated. Readers should re-run examples against their own compatibility dates, data, policies, and provider accounts.

Change control

Release-sensitive facts are maintained in the Cloudflare release ledger and citation audit. The reader-facing release-readiness record makes the publication gates inspectable without publishing private approval material. Corrections and reader-reported problems belong in the companion repository errata workflow. The edition policy defines when a change is a patch, a minor update, or a new major edition.

Edition And Version Policy

This policy governs *The Edge That Thinks* and its companion repository.

Edition line

Label	Meaning	Distribution
`0.x` public-review edition	Substantive but not final manuscript shared for technical review; structure, claims, and examples may change	Free public distribution with visible prerelease status
`1.0` public edition	First complete, copyedited digital release with cited platform claims, approved case studies, and accessible exports	Stackbilt site and Leanpub; ordinary KDP later if ready
`1.x` living edition	Backward-compatible corrections, refreshed citations, clarified examples, and small additions	Leanpub and the companion repository
`2.0` or later	Material change to the book's argument, structure, audience, or edition contract	New release decision required

The current manuscript remains a `0.x` public-review edition. Do not label it `1.0` until the public gates in Release Readiness and Evidence and the architecture checks in Chapter 16 are satisfied. The repository's operator-only production checklist may contain private coordination evidence; the reader-facing readiness record is the auditable publication contract.

What qualifies as a patch release

A `1.x` update may correct a factual error, refresh a release-sensitive link, replace a stale platform limit, clarify prose, repair an accessibility issue, or update a companion example without changing the book's core argument. Each release records the date, affected chapters, platform facts rechecked, and whether the change affects existing readers.

What requires a new minor or major decision

Additions that materially change the examples, compatibility assumptions, or reader workflow should receive a new minor release note. A new platform architecture, substantially reorganized manuscript, changed commercial promise, or incompatible companion-repository interface requires a new major edition decision.

Print snapshots

Print is a dated snapshot, not the living edition. A print release records the book version, render date, page-layout inputs, and a link or QR destination for living errata. Paperback remains deferred until the manuscript reaches the length and design threshold already recorded in the publishing plan.

Release checklist

Before any public release, confirm:

1. The version and publication date appear in the book and release notes.
2. Release-sensitive citations, model availability, limits, pricing, and compatibility dates were rechecked.
3. Case-study approval and redaction conditions still hold.
4. Accessible exports and public links were tested.
5. The companion repository changelog and errata entry are updated.

Release Readiness and Evidence

Edition: 0.2.1 public-review edition

Published: 2026-08-13

Platform claims verified through: 2026-08-09

This is the reader-facing release-control record for *The Edge That Thinks*. It states what was checked, what the review package contains, and why this edition remains `0.x`. It does not expose private repository paths, credentials, customer material, or internal approval evidence.

This edition is free to read and download during public review. It is a substantive field guide, not a finished `1.0`: practitioner review, final assistive-technology review, and the remaining publication gates below are still open. Corrections may change wording, examples, diagrams, or structure.

Review package contract

The canonical reviewer package contains:

- a self-contained EPUB;
- a PDF generated from the same reader HTML;
- a versioned HTML ZIP containing the HTML file, all 11 static SVG figures, editable Mermaid sources, a package README, and an internal package manifest;
- a SHA-256 reviewer manifest covering the source inputs and every distributed artifact, including the complete HTML ZIP.

The loose HTML file is a build component, not the complete distributable: it uses relative `figures/` paths. Reviewers should receive the HTML ZIP, PDF, or EPUB plus the manifest and should report version `0.2.1` with every finding. No runtime Mermaid renderer or network connection is required to see a figure.

Evidence and review status

- Cloudflare platform claims use nearby first-party documentation, changelog, specification, or blog links and were rechecked through the date above.
- The dated release ledger separates verified platform facts and prerequisites from the author's architectural interpretation.

- Named Stackbilt case studies were checked against implementation evidence and conditionally approved by the product owner. Public wording identifies these as dated observations rather than platform guarantees.
- Four model-assisted technical passes have examined the architecture and failure boundaries, including three independent frontier-model reviews of `0.1.1`. Their accepted findings are incorporated into `0.2.1`; this does not substitute for the planned practitioner cohort.
- The export build checks stable section anchors, local assets, figure descriptions, table headers, and package membership. PDF/EPUB visual and assistive-technology review remains a human release task.

Current 1.0 gates

The book remains a reviewer edition until all of these are true:

1. At least 10 practitioners cover edge/platform, backend, security/privacy, and AI-product operations; blocker and major findings are triaged.
2. Every externally checkable platform assertion has an adjacent citation and a generated chapter/claim audit reports no unresolved release-sensitive gap.
3. The companion repository is public with diagram sources, runnable examples, release notes, license, and an errata workflow.
4. The title, subtitle, and introduction accurately promise an architecture field guide; any build-oriented promise is backed by a runnable vertical slice.
5. HTML ZIP, site HTML, EPUB, and PDF pass clean-environment rendering, keyboard/screen-reader review, link checking, and page-by-page visual QA.
6. Case-study approval and redaction conditions are rechecked against the exact release wording.
7. Version, build date, source revision, author, rights, canonical URL, and errata URL appear in the release artifacts and manifest.

Chapter 16 contains the system-design checklist. This section governs the book artifact itself. A checked architecture list cannot waive a failed publication gate, and a polished artifact cannot waive an unresolved correctness finding.

Acknowledgments

This is a working acknowledgment record for the reviewer edition. Names should be added only with the person's permission and with an accurate description of their contribution.

To confirm before the public edition

- Technical reviewers who checked the architecture, Cloudflare claims, and failure cases.
- Contributors to the Stackbilt systems used as case-study evidence.
- Diagram and accessibility reviewers.
- Readers who submitted reproducible errata or companion-example fixes.

No individual names are asserted in this draft. The public edition should add only confirmed names, affiliations if approved, and contribution descriptions that the contributors accept.

Cloudflare release ledger — 2026

Verified through 2026-08-09. Source standard: official Cloudflare documentation, changelog, or blog only.

This is a living research ledger for the public edition. It records releases that materially change the design space for generative and agentic systems. It is not a feature checklist: each entry exists because it changes a design decision a reader may need to make.

Each row separates the first-party fact from this book's interpretation. Product status and prerequisites are snapshots, not promises.

Date	Verified platform change	Status / availability at verification	Author interpretation	Primary source
Aug. 7	Workers AI and supported third-party models gained shared AI Gateway access, observability, and unified billing.	Account and model eligibility still apply.	A common inference boundary can simplify attribution and payment, but product routing and terminal accounting remain application concerns.	Unified access and billing
Aug. 5	AI Gateway added identity-aware controls and User Insights for model access, spend limits, and anomalous-usage review.	Requires an identity signal and configured gateway policy.	Central policy can enforce model and spend constraints; it does not replace tenant authorization or job reservations.	Identity-aware controls; User Insights
Aug. 5	Cloudflare published Cloudflare OS as an open organizational AI platform.	Open-source platform; deployment choices and product prerequisites vary.	Treat organizational AI as an application platform with curated context, governed systems-of-record access, isolated apps, and centralized inference policy.	Cloudflare OS announcement; deployment lessons
Aug. 3	Cloudflare described serving changes for Kimi and GLM models.	Model catalog, plan access, and price remain release-sensitive.	Keep routing and evaluation replaceable rather than anchoring a design to one model or price.	Model-serving update

Date	Verified platform change	Status / availability at verification	Author interpretation	Primary source
Jul. 28	Cloudflare's product MCP servers adopted MCP 2026-07-28 stateless requests while retaining compatibility for 2025 Streamable HTTP clients.	New `/mcp` connections use the stateless handler; legacy transport behavior requires migration review.	Do not make gateway correctness depend on protocol-session affinity. Authorize and make each operation idempotent.	Cloudflare MCP server update
Jul. 28	Kimi K2.6, Kimi K2.7 Code, and GLM-5.2 moved off the Workers Free plan.	Workers Paid plan required for these models.	Plan eligibility belongs in the routing contract and must be rechecked before release.	Workers AI changelog
Jul. 27	Agents SDK v0.20.0 added MCP 2026-07-28 client/server support, stateless server factories, and MRTR elicitation.	Negotiates stateless or legacy behavior; `McpAgent` is deprecated and feature-frozen.	Migrate explicitly and preserve business idempotency across `input_required` retries.	Agents SDK v0.20.0
Jul. 27	Wrangler added a production-build integration test harness.	Requires the supported Wrangler test API and built Worker output.	Test the deployed artifact shape, bindings, and compatibility date—not source alone.	Test harness changelog; test harness guide
Jul. 13	Agents added MCP elicitation for connected legacy MCP servers.	The 2026-07-28 protocol later changed stateless elicitation delivery to MRTR.	Structured user input is safer than invented sensitive parameters; protocol interaction is not authorization.	MCP elicitation entry
Jun. 26	Agents SDK v0.17.0 introduced background sub-agents and unified turn admission.	Capability-introduction record; later SDK releases supersede the current version.	Long work needs durable milestones, recovery, and one admission path.	Agents SDK v0.17.0
Jun. 19	Wrangler introduced temporary Cloudflare accounts and claimable agent deployments.	Temporary deployments expire if not claimed; supported resources and limits may change.	Design agent onboarding around temporary authority and explicit human ownership transfer.	Temporary Cloudflare Accounts
May 30	Requests to deprecated Kimi K2.5 began aliasing to Kimi K2.6 at a higher price.	Migration behavior applies to the named catalog identifiers.	Alias drift can change cost and behavior without an application-code change; record the resolved model.	Model deprecation notice

Date	Verified platform change	Status / availability at verification	Author interpretation	Primary source
May 1	Cloudflare announced Dynamic Workflows.	Product status and account availability must be checked before adoption.	Separate the durable workflow engine from customer- or agent-supplied code.	Dynamic Workflows
Apr. 20	Kimi K2.6 became available through Workers AI interfaces.	By Jul. 28, the model required Workers Paid.	Select models through a dated capability and eligibility contract, not a hard-coded vendor assumption.	Kimi K2.6 release
Apr. 16	Cloudflare described AI Gateway as a shared inference layer.	Supported providers and features remain release-sensitive.	Put provider access, caching, rate limits, attribution, and common controls at a shared boundary while retaining application routing.	AI Platform announcement
Mar. 24	Cloudflare announced Dynamic Workers for isolated generated-code execution.	Isolation depends on deliberately limited bindings, egress, and resources.	Do not run untrusted generated code in a privileged process; isolation reduces but does not remove residual risk.	Dynamic Workers
Mar. 19	Cloudflare announced larger models on Workers AI.	Model catalog and availability change over time.	Model size does not remove the need for durable state, workflows, and secure execution.	Large-model announcement
Feb. 17	Agents SDK v0.5.0 added retry utilities and protocol controls.	SDK-specific capability; consult current migration guidance.	Retry behavior is application policy and must be tested around non-idempotent effects.	Agents changelog
Feb. 13	Workers AI and framework adapters expanded tool calling, streaming, voice, and reranking support.	Model and adapter compatibility remain version-specific.	Test tool-call round trips and stream failures as part of the product reliability contract.	Workers AI changelog

Cloudflare OS: why it belongs in this book

Cloudflare OS is the newest and clearest articulation of the book's argument. Cloudflare describes it as an open-source platform organizations can deploy and adapt around their own systems, policies, skills, and context. Its published design uses AI Gateway for model policy and attribution, authenticated

gateways to systems of record, and Dynamic Workers plus Durable Object Facets to isolate generated applications and their state. Cloudflare OS announcement

The transferable lesson is not “adopt Cloudflare OS.” It is this:

> An organization-wide agent platform must make context, permissions, execution, cost, and provenance first-class infrastructure.

The Cloudflare OS chapter will compare that pattern with smaller products, where the equivalent might be a single Worker, an identity provider, a job table, and a small tool registry. Readers should be able to scale the principle down as well as up.

Citation rule for the manuscript

Every externally checkable claim must be cited at the point it is made.

- **Stable concepts** may cite a canonical documentation page.
- **Release behavior, prices, quotas, model availability, and beta status** must cite a dated official source and carry a “verified through” date in the chapter notes.
- **Stackbilt experience** is labeled as a case study or field observation, not evidence that a Cloudflare service has a particular guarantee.
- **Benchmarks** must name the workload, environment, comparison, and date—or they do not appear.

The public manuscript will have both inline links and a per-chapter bibliography.

References

Editorial bibliography — verified through 2026-08-09

This first-draft bibliography favors primary documentation and first-party release notes. Chapter-specific links remain inline because platform behavior changes more quickly than book production.

Cloudflare platform foundations

> Cloudflare Developers documentation

> Workers documentation

> Workers AI documentation

> AI Gateway documentation

> Durable Objects documentation

> Workflows documentation

> Queues documentation

> D1 documentation

> Vectorize documentation

> R2 documentation

> Service Bindings documentation

> Workers observability documentation

> Cloudflare Agents documentation

> MCP on Cloudflare

> Model Context Protocol 2026-07-28 specification

> MCP 2026-07-28 release overview

D1 global read replication and Sessions limitations

Service Binding limits

Workers subrequest-limit change

2026 platform releases and context

Cloudflare OS: an open platform for agents, apps, and work, August 5, 2026.

How Cloudflare uses AI with Cloudflare OS, August 5, 2026.

Smaller, faster, safer: running Kimi and GLM at scale, August 3, 2026.

Workers Changelog.

Workers AI Changelog.

Introducing Dynamic Workflows, May 1, 2026.

Cloudflare’s AI Platform: an inference layer designed for agents, April 16, 2026.

Sandboxing AI agents with Dynamic Workers, March 24, 2026.

Powering agents with large models on Workers AI, March 19, 2026.

Temporary Cloudflare Accounts for AI agents, June 19, 2026.

Cloudflare MCP servers adopt MCP 2026-07-28, July 28, 2026.

Agents SDK v0.20.0 adds MCP 2026-07-28 support, July 27, 2026.

Planned Workers AI model deprecations, May 8, 2026.

Cloudflare AI product changelog.

Research hygiene

The export build generates a chapter-by-chapter citation index from adjacent links so reviewers can audit the evidence actually present in each chapter. The final edition must revalidate every release-sensitive citation before layout. No pricing, quota, model catalog, preview status, or compatibility date

should survive simply because it was accurate in this draft.

Citation Index by Chapter

Generated from the adjacent external links in this edition.

How to Use This Book

> Developer Documentation

> Workers Changelog

> Workers AI Changelog

1. The Edge That Thinks

> Workers

> D1

> Durable Objects

> Queues

> R2

> Vectorize

> Workers AI

> Service Bindings

> Cloudflare OS: announcement

> Cloudflare OS: operational account

3. Service Composition Is a Security Decision

> Service Bindings documentation

> Service Binding limits

> Workers subrequest-limit change

4. State, Locality, and the Honest Limits of D1

> D1 global read replication

> D1 Sessions API

> D1 read-replication limitations

> D1 data location

> D1 observability

5. Route Before You Reason

> AI Gateway REST API

> AI Platform announcement

> Automatic retry on upstream provider failures

6. Memory Is a Product Feature, Not a Prompt Field

> Vector database documentation

7. Durable Objects, Queues, and Work That Outlives a Request

> Durable Objects overview

> delivery guarantees

> batching and retries

> Workflow guide

> sleeping and retrying

8. Tools, MCP, and Capability Boundaries

> Model Context Protocol specification

> MCP 2026-07-28 overview

> MCP client overview

> portal context optimization

> Agents SDK v0.20.0

9. Multi-Model Economics and Quality Tiers

> AI Gateway overview

> REST API

> Workers AI changelog

> Kimi K2.5 migration notice

> AI product changelog

> rate limiting

10. Observability: Trace Decisions, Not Just Errors

> AI Gateway logging

> AI Gateway limits

> Workflow guide

11. Identity, Tenancy, Retention, and Consent

> Queues pause and purge

> R2 documentation

> Agents SDK v0.20.0

> Vectorize query documentation

12. From Workflow to Agent: Earned Autonomy

> AI Gateway REST API

> Queues delivery guarantees

> Durable Objects glossary

> Alarms

13. Case File: A Cognitive Kernel With a Circuit Breaker

> SQLite-backed Durable Object storage

14. Case File: Multi-Tenant Generation Without a Fragile Request Path

> Queues delivery guarantees

> How Queues works

> Cloudflare R2

> AI Gateway REST API

15. Case File: Ingestion, Retrieval, and the Cost of Memory

> R2 multipart uploads

> Vectorize metadata filtering

> Workers KV consistency

> Vectorize query best practices

16. The Production Checklist

> Queues delivery guarantees

> Durable Object alarms

17. Cloudflare OS: Context, Gatekeepers, and the Deterministic Turn

> Cloudflare OS announcement

> How Cloudflare uses AI with Cloudflare OS

> Dynamic Workers

> AI Gateway REST API

> AI product changelog

Appendix A — A Reference Architecture for Generative Work

> Workflows overview

> Durable Objects documentation

> MCP on Cloudflare

> AI Gateway documentation