HITACHI

Skills are all you need

Developer Guide

Designing, Packaging, and operating enterprise skills

Centre for Architecture & AI (CAAI) – Vitor Domingos

March 2026

Hitachi Digital Services

Table of contents

March 2026

Intended audience: platform engineers, AI/ML engineers, solution architects, security and SRE teams, DevOps

Background

Introduction

The CAAI – Skills are all you need Whitepaper argues that skills governed, callable capabilities with stable contracts, should be the unit of execution in enterprise agent ecosystems.

This guide addresses the engineering question: how do you actually design, package, deploy, and operate a skill.

The objective is portability and safety. A well-built skill can be invoked by any agent runtime that speaks MCP or function-calling conventions, governed by a platform that enforces policy regardless of the caller, and operated with the same discipline as any production service. This guide covers contract design, runtime patterns, gateway responsibilities, security and defensive design, observability, composition, testing, MCP integration, deployment, and lifecycle management.

> *The contract is the product. Everything else; runtime, policy, telemetry, lifecycle, exists to make the contract trustworthy.*

A note on the title: it is a deliberate nod to "Attention Is All You Need" (Vaswani et al.), the paper that established the transformer architecture and set the direction of travel for today's GPT-class models. The intent here is not to claim a similar scientific breakthrough, but to signal the same design instinct: pick the right primitive, and scale becomes manageable.

ENGINEERING FOUNDATIONS

Design Principles

Six principles underpin skill engineering. They shouldn't be aspirational; they are the minimum conditions we classify to operate a skill in a regulated enterprise environment.

- Every skill has explicit timeouts, retry budgets, and compensation paths.
- If a skill cannot complete within its budget, it fails with a stable error code rather than spinning indefinitely.
- Business rules belong in code and policy files rather than in prompts alone.
- A skill may use a language model internally for inference or summarisation, but any action that changes enterprise state should be governed by deterministic logic.
- Credentials are scoped and short-lived, issued by the gateway per invocation.
- Agents never hold operational secrets.

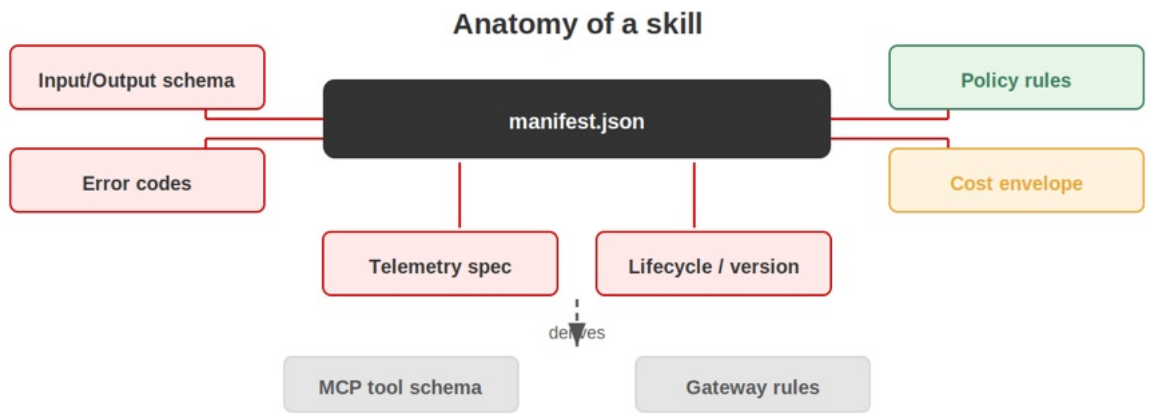
Every skill classifies its inputs and outputs by sensitivity and enforces retention and redaction rules accordingly. Every invocation emits traces, metrics, and structured logs with cost attribution tags; if a skill cannot be instrumented, it is not production ready. Contracts follow semantic versioning with automated compatibility tests: breaking changes require a new major version and a parallel support window.

CONTRACT DESIGN

The contract specification

The contract is the single most important artefact in skill engineering. It defines what the skill accepts, what it returns, how it fails, what policy it requires, and what telemetry it emits. A contract should be machine readable; so prefer JSON Schema or OpenAPI for HTTP-shaped skills, and an explicit manifest schema for non-HTTP runtimes.

The diagram below shows how the manifest sits at the centre of the skill's architecture. Every other artefact like MCP tool schemas, gateway validation rules, catalogue metadata, derives from it.



A manifest includes the skill name and semantic version, a human-readable description suitable for agent discovery, typed input and output schemas with defaults and enumerations, a catalogue of stable error codes mapped to HTTP status codes, policy requirements (data classification, required scopes, approval flags), telemetry expectations (which metrics and log fields the skill guarantees), and a cost envelope declaring expected p50, p95, and maximum cost per invocation.

Example: equipment diagnostics manifest

```
{
  "name": "equipment_diagnostics",
  "version": "1.2.0",
  "description": "Assess equipment health and propose actions",
  "inputs": {
    "equipment_id": {"type": "string", "required": true},
    "signals": {"type": "timeseries", "required": true},
    "time_window": {"type": "string", "default": "72h"}
  },
  "outputs": {
    "health_score": {"type": "number"},
    "risk_level": {"type": "string", "enum": ["low","medium","high"]},
```

```
    "recommendations": {"type": "array"},
    "evidence": {"type": "array"}
  },
  "errors": [
    {"code": "ASSET_NOT_FOUND", "http": 404},
    {"code": "POLICY_DENIED", "http": 403},
    {"code": "BACKEND_TIMEOUT", "http": 504}
  ],
  "policy": {
    "data_classification": ["internal","confidential"],
    "required_scopes": ["asset.read","signals.read"],
    "requires_approval": false
  },
  "telemetry": {
    "metrics": ["latency_ms","success","cost_units"],
    "logs": ["trace_id","invocation_id","equipment_id"]
  },
  "cost_envelope": {
    "p50_cost_units": 0.002,
    "p95_cost_units": 0.006,
    "max_cost_units": 0.02
  }
}
```

The cost envelope deserves particular attention. Attaching expected cost per invocation to the manifest makes cost a first-class property of the contract. If bounded cost cannot be guaranteed, because the skill calls a model with unbounded output for instance, then the skill should be flagged as experimental until stabilised.

Request and response

A concrete example helps ground the abstraction. A request to the equipment diagnostics skill sends a pump identifier and 72 hours of vibration and temperature signals. The response returns a health score, a risk classification, actionable recommendations, and evidence links that trace back to source data.

```
// Request
{ "equipment_id": "PUMP-17", "time_window": "72h",
  "signals": { "vibration_rms": [0.12, 0.11, 0.15],
               "temperature_c": [62.1, 62.3, 63.0] } }

// Response
{ "health_score": 0.71, "risk_level": "medium",
  "recommendations": [
    "Inspect bearing assembly within 7 days",
    "Verify lubrication schedule" ],
  "evidence": [
    "hist://plant01/pump17/vibration_rms?window=72h",
    "cmms://workorders?asset=PUMP-17&window=180d" ] }
```

PACKAGING

Packaging and Repository Structure

A skill should be treated as a deployable artefact with everything needed to build, test, and operate it in a single repository. The manifest sits at the root alongside source code, policy rules, tests, documentation, and CI pipeline definitions.

```
skill/
  manifest.json       # Contract, policy, cost envelope
  src/                # Implementation
  policy/             # Rego rules and test fixtures
  tests/              # Contract, integration, resilience
  docs/               # Runbook, architecture notes
  ci/                 # Pipeline definitions
  CHANGELOG.md
```

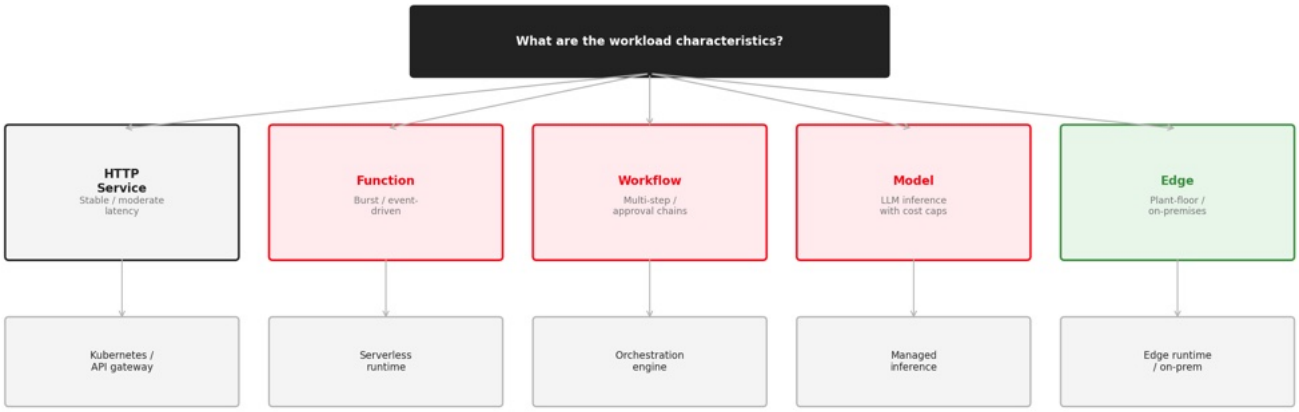
Keeping policy rules alongside the skill rather than in a central repository means the skill team owns its governance artefacts and can test them in CI. The platform provides policy templates and shared libraries; the skill team instantiates and extends them.

RUNTIME

Runtime Patterns

Not every skill runs the same way. The choice of execution substrate depends on latency requirements, governance constraints, and operational trade-offs. Five patterns cover the majority of enterprise scenarios.

- **HTTP service skills** suit stable integrations with moderate latency. They run as long-lived processes behind a Kubernetes service or API gateway with predictable scaling and conventional observability.
- **Function skills** suit burst workloads and event-driven processing, trading cold-start latency for elastic scaling. Execution time limits bound runaway cost.
- **Workflow skills** handle multi-step enterprise processes — change management, approval chains, remediation sequences — where long-running state and compensation justify a dedicated orchestrator.
- **Model skills** wrap inference endpoints with explicit constraints: token budgets, output schema validation, and cost caps. Output schema validation is mandatory; unbounded model output is a cost and safety risk.
- **Edge skills** run on plant-floor or on-premises infrastructure where cloud round-trips are not acceptable. They introduce specific concerns: local secrets handling, offline audit buffering, dependency caching, and remote update mechanisms.



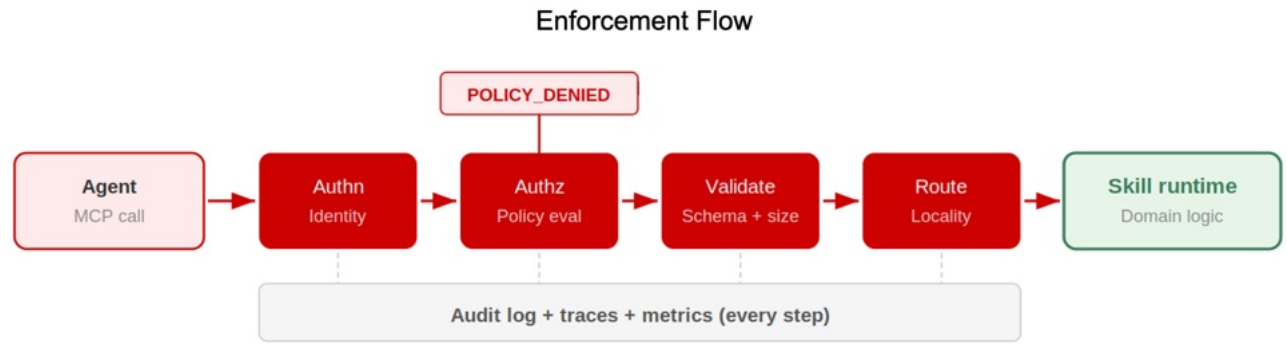
Pattern	Best for	Trade-offs	Typical substrate
HTTP service	Stable integrations, moderate latency	Ops overhead, scaling	Kubernetes / API gateway
Function	Burst workloads, events	Cold starts, execution limits	Serverless runtime
Workflow	Multi-step processes, approvals	Long-running state overhead	Orchestration engine
Model	Inference with explicit constraints	Cost control, schema validation required	Managed inference endpoint
Edge	Plant-floor latency	Deployment complexity	Edge runtime / on-prem

SECURITY

The Gateway as Enforcement Boundary

The gateway is the single enforcement point in a skill-centric architecture. It handles authentication, authorisation, policy evaluation, quota enforcement, request validation, routing, trace context propagation, and audit logging. These concerns must not be pushed into individual skills. If they are, consistency is lost and platform-level assertions about control become impossible.

The diagram below traces a request through the gateway. Each stage, with authentication, authorisation, validation and routing, is independently logged. A denied request returns a stable error code without reaching the skill runtime.



Skills should assume the gateway has already enforced baseline controls and focus exclusively on domain behaviour. The practical result is simpler skill implementations: no credential management, no access policy evaluation, no cross-cutting concerns.

> *Do not push gateway concerns into individual skills. Consistency requires a single enforcement boundary, not per-skill implementations of authentication and policy.*

GOVERNANCE

Policy as Code

Policy evaluation should be automated, testable, and versioned alongside the skill. Open Policy Agent with Rego is the most common choice, though any engine that supports declarative rules and test fixtures serves the purpose.

The example below grants write access to the CMMS only when the subject holds the correct scope and the risk level is not high. Note that the default is deny; skills must be explicitly permitted, not permitted by omission.

```
package skill.authz

default allow = false

allow {
  input.subject.scopes[_] == "cmms.write"
  input.request.action == "create_work_order"
  input.request.risk_level != "high"
}
```

Test policies in CI as part of the skill pipeline. Keep policies small and composable: build templates for common patterns - read-only access, write-with-approval, break-glass - and let skill teams extend them. Policy versioning should track skill versioning; a major version bump in the skill should trigger a review of its policy rules.

Policy rules belong in the skill repository, not in a central policy store. Centralising policy creates a coupling point: every skill change requires a separate policy review cycle in a repository the skill team does not own. Co-location means the skill team can test policy changes in the same CI run as the implementation change.

SECURITY

Threat model and defensive design

Skills must assume hostile inputs. The agent may be misled by prompt injection. The user may be adversarial. Upstream content, like the ones from knowledge bases, documents, or third-party APIs may be poisoned. Defensive design is the baseline posture for any skill operating in an agentic context

Injection can be mitigated at multiple layers, and the gateway enforces authorisation independently of the agent's reasoning. Within the skill, strict schema validation, maximum payload sizes, and allow-lists of permitted operations prevent unexpected inputs reaching backends. Data exfiltration is controlled through output schemas and redaction rules. Privilege abuse is bounded by quotas, rate limits, and anomaly detection. Supply-chain risk demands signed artefacts, provenance metadata, and certification tiers.

Threat	Example	Mitigation
Prompt injection	User content triggers unauthorised write	Gateway authz; read/write separation; approvals
Data exfiltration	Skill returns sensitive fields	Output schema + redaction + classification
Privilege abuse	High-frequency write calls	Quotas, rate limits, anomaly alerts
Denial of service	Oversized payload	Size limits; streaming; early rejection
Poisoned sources	Knowledge base contains malicious instructions	Approved sources; content scanning; citations
Supply chain	Compromised skill package	Signed artefacts; provenance; certification

OPERATIONS

Observability and Cost Attribution

Every skill invocation should produce a trace span, structured metrics, and a log envelope. Instrument with OpenTelemetry to ensure compatibility with standard backends and to preserve trace context across skill composition chains.

The minimum telemetry for each invocation includes a trace ID that links to the broader orchestration context, an invocation ID unique to this call, skill name and version, structured outcome, latency, and cost attribution tags. Cost attribution should record model token usage where applicable, compute time, and external backend calls.

Example log envelope

```
{ "timestamp": "2026-03-04T10:12:32Z",
  "invocation_id": "inv-8f7c",
  "trace_id": "4bf92f3577b34da6a3ce929d0e0e4736",
  "skill": "equipment_diagnostics",
  "version": "1.2.0",
  "outcome": "success",
  "latency_ms": 412,
  "cost_units": 0.0031 }
```

Each skill should have dashboards covering: reliability (success rate, error codes, retries); latency (p50/p95/p99); cost (units per invocation, token usage); policy (deny rates by rule and scope); and dependency health. Keep alerting conservative. For writing skills, add an approvals dashboard: approvals issued, time-to-approve, and rollback rate.

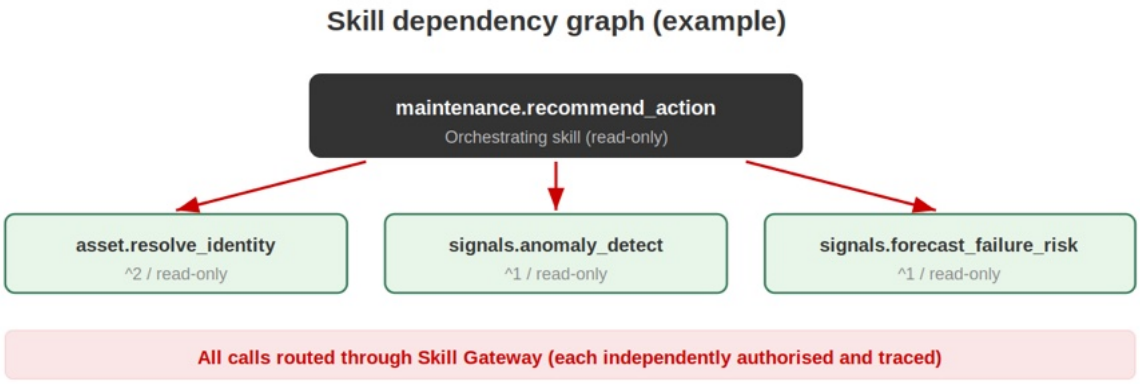
Cost attribution is the mechanism that connects skill execution to business unit budgets. Without it, agent systems appear to have zero marginal cost until a bill arrives. Tag every invocation with at minimum: skill name, version, caller identity, and business unit. Make cost variance visible in the same dashboard cycle as reliability.

ARCHITECTURE

Composition and Dependency Graphs

Skills will frequently invoke other skills. A predictive maintenance workflow might call asset identity resolution, anomaly detection, failure forecasting, and work-order creation in sequence. Composition is where the architecture proves itself operationally or collapses under hidden coupling and cascading failures.

All inter-skill calls route through the gateway so that composition remains observable and governed. Dependencies must be explicit in the manifest. At minimum, declare which upstream skills are invoked, what privilege boundaries are crossed, and what the failure semantics are.



Dependency manifest extension

```
{ "name": "maintenance.recommend_action",
  "version": "1.0.0",
  "depends_on": [
    {"skill": "asset.resolve_identity", "version": "^2"},
    {"skill": "signals.anomaly_detect", "version": "^1"},
    {"skill": "signals.forecast_failure_risk", "version": "^1"}
  ],
  "privilege_boundary": "read_only" }
```

Transaction semantics in skill chains are almost always compensating rather than ACID. Design write skills with idempotency keys bound to the invocation, bounded retry budgets, and a named compensation action. Maintain audit coherence across the chain through trace propagation: every step links to the same orchestration trace ID.

Cascading failure is the primary operational risk in composed skill chains. Each skill must honour its timeout budget; a skill that hangs degrades all callers. Declare circuit breaker thresholds in the manifest and test them as part of the resilience suite. When a dependency degrades, the skill should degrade gracefully, returning partial results with evidence citations rather than failing completely

SAFETY PATTERNS

Writing-Skill Safety Patterns

Writing skills require stronger controls because they change enterprise state. The most effective default is to split every write operation into two phases: propose and commit. In the propose phase, the agent calls a read-only skill that generates a plan. The gateway then enforces an approval workflow for the commit phase. The commit skill performs bounded changes using idempotency keys, honours change windows, and writes immutable audit evidence.

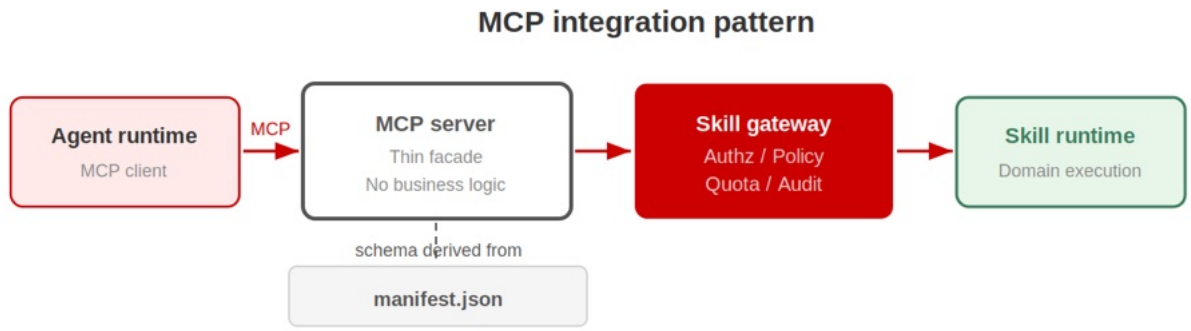
Control	Implementation	Why it matters
Idempotency	Keys bound to each invocation	Prevents duplicate writes on retry
Approvals	Ticket or policy approval gate	Governance for privileged actions
Rollback	Named compensation action	Resilience if downstream steps fail
Change windows	Time-based policy checks	Protects production environments
Evidence	Immutable audit entry with refs	Compliance, forensics, traceability

Idempotency keys must be bound to the invocation ID, not generated by the skill. The gateway issues invocation IDs; using them as idempotency keys means duplicate calls from retrying agents are automatically detected and deduplicated without the skill needing to maintain its own state.

INTEGRATION

MCP Integration

The goal for skills is to usable by any agent and MCP should be the default exposure interface. The MCP server acts as a thin facade with no business logic. The gateway remains the single enforcement point. The diagram below illustrates the integration pattern.



Derive MCP tool schemas from the skill manifest to prevent drift. When the manifest is updated, the MCP schema should regenerate automatically. Naming conventions matter for agent discovery: use the pattern **domain.family.operation** so that agents and catalogue tooling can organise skills by domain and detect duplication.

MCP facade sketch

```
// Pseudocode: MCP server as thin facade
server.registerTool("equipment_diagnostics", async (req) => {
  const decision = await gateway.authorise(
    req.subject, "equipment_diagnostics"
  );
  if (!decision.allow) return { error: "POLICY_DENIED" };
  return await gateway.invoke(
    "equipment_diagnostics", req.input, { trace: req.trace }
  );
});
```

The MCP server should have no awareness of the skill's implementation. It receives a request, forwards it to the gateway with the caller's identity and the originating trace context, and returns whatever the gateway returns. Business logic, data classification, and policy decisions are gateway and skill concerns, not MCP concerns.

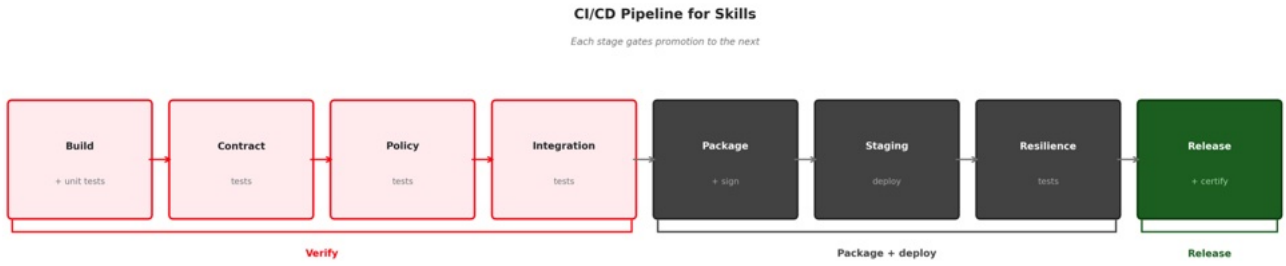
QUALITY

Testing, Certification, and CI/CD

The skill test suite covers four dimensions. Together they form the verification layer that gates every promotion in the CI/CD pipeline.

- **Contract tests** verify that the skill honours its manifest: correct input validation, output schema, and error codes. Run against every commit.
- **Policy tests** verify that access rules behave as expected under all defined scenarios, including edge cases (missing scopes, elevated risk levels, break-glass conditions).
- **Integration tests** verify backend interactions including failure modes: timeouts, 5xx responses, malformed payloads, and partial data returns.
- **Resilience tests** verify predictable failure: stable error codes under load, no thundering-herd effects, no runaway compute when dependencies degrade.

The CI/CD pipeline must be explicit and auditable, with each stage gating promotion to the next. No stage is optional for production skills.



Certification is a platform concern layered on top of CI. The platform defines risk-tiered gates: basic contract checks for read skills, full policy and threat-model review for write skills, and the tightest controls for break-glass operations. Promotion to each catalogue channel requires the appropriate certification level.

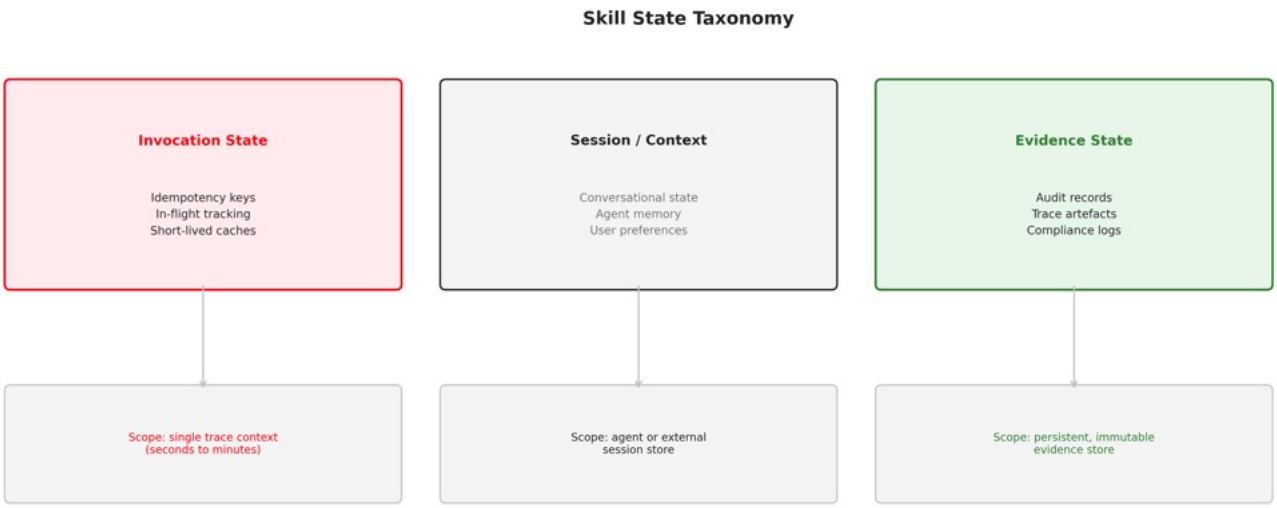
ARCHITECTURE

State and Memory

Current industry default posture is stateless execution. So, skills should not maintain long-lived memory between invocations. Persistent state belongs in dedicated systems: data stores, case management platforms, knowledge bases, and vector indexes.

That said, not all state can be eliminated. Three categories are legitimate within the skill boundary:

- **Invocation state:** idempotency keys, in-flight request tracking, short-lived caches, is scoped to a single trace context (seconds to minutes). It must not leak between concurrent invocations.
- **Session state and conversational context:** belong in the agent or an external session store, never in the skill. A skill that accumulates session context across callers becomes stateful in ways that are invisible to the gateway and untestable in CI.
- **Evidence state:** audit records, trace artefacts, must be persistent and immutable, always written to a dedicated evidence store before the invocation returns a success response.



DEPLOYMENT

Deployment and Edge

Deployment patterns follow from runtime choices. Shared platform deployments suit most cloud-hosted skills. Domain-cluster deployments suit ownership boundaries between business units. Edge deployments suit latency-sensitive workloads where cloud round-trips are not acceptable.

Edge skills introduce specific concerns absent from cloud patterns: device identity with rotation; local audit buffering with periodic sync; signed artefacts with staged rollouts and rollback; local caching for critical reference data; and store-and-forward telemetry for connectivity interruptions. Treat edge deployments as a controlled fleet, not as ad-hoc installations.

For edge skills, the gateway cannot intercept calls in real time if connectivity is absent. The solution is a local policy cache: the edge runtime holds a signed, time-bounded copy of the relevant policy rules and enforces them locally. When connectivity is restored, the local audit buffer is flushed to the central evidence store and the policy cache is refreshed.

LIFECYCLE

Versioning, Deprecation, and Catalogue Governance

Semantic versioning is the baseline. Breaking changes require a new major version and a parallel support window. Deprecated skills carry migration notes and end-of-support dates. Monthly catalogue reviews focus on duplicates, unused skills, and approaching end-of-support.

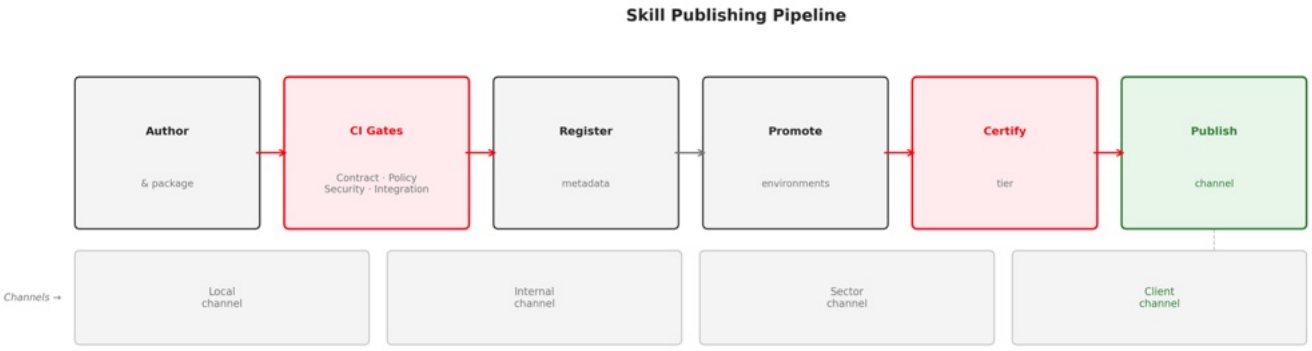
Catalogue hygiene requires ongoing discipline. Four signals warrant action:

- Skills with zero consumers should be deprecated. Zero consumers mean zero visibility into degradation, as runbooks go stale and owners move on.
- Low success rates warrant investigation before the skill is used in a new composition. A skill that fails 15% of the time silently degrades every workflow that depends on it.
- High policy-deny rates suggest a confusing access model. Either the required scopes are not documented clearly, or the skill's contract does not match the data it actually touches.
- High-cost variance indicates unbounded execution that needs caps or refactoring. Duplicate tags signal overlap that should be consolidated into a shared primitive.

DISTRIBUTION

Marketplace Publishing

The publishing workflow: author and package; CI gates for contract, policy, integration, and security; register with metadata; promote through environments; certify at the appropriate tier; publish to the relevant catalogue channel.



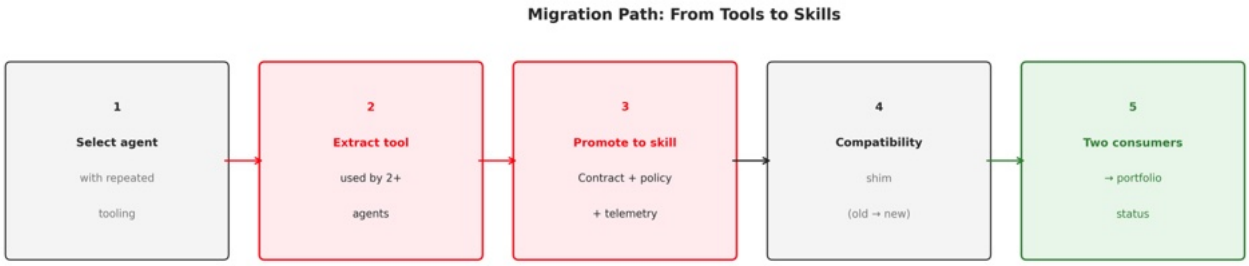
Channels control visibility and trust. A skill published to a client channel carries contractual controls and audit requirements that an internal channel skill does not. The tier determines which gates must be passed before the skill is visible in that channel.

- Local channels serve a single team. For experimental skills and proof-of-concept work.
- Internal channels serve delivery teams with certified primitives. Requires basic contract and policy certification.
- Sector channels serve practice teams with curated domain libraries. Requires full certification including threat model review.
- Client channels serve individual clients with contractual controls and audit requirements. Highest certification tier.

MIGRATION

Migrating from Tools to Skills

The migration should start with one agent that has repeated tooling. Extract a tool used by at least two agents and promote it into a skill with a contract, policy, and telemetry. Maintain compatibility shims: the old tool calls the new skill behind the scenes until consuming agents migrate. Once a skill has two independent consumers, treat it as portfolio: adding ownership, a runbook, and versioning discipline.



Compatibility shims have a defined end of life. Set the shim deprecation date at the point the skill reaches its first stable major version. Track shim usage in telemetry so the migration can be declared complete when consumer count drops to zero.

COMMON MISTAKES

Anti-Patterns

These anti-patterns recur in early implementations of skill-centric architectures. Each reflects a different failure to treat the skill as a production artefact.

- **Embedding business logic in prompts** without version control or tests creates untraceable rules that break silently when model behaviour changes. Business logic belongs in code and policy files, not in system prompts.
- **Passing credentials through agent context** exposes secrets to injection and logging. Credentials are a gateway concern; skills receive short-lived, scoped tokens per invocation.
- **Accepting unbounded payloads** without size limits or redaction lets oversized or sensitive inputs reach backends unfiltered. Every input must be validated against the manifest schema before reaching the skill implementation.
- **Publishing skills with no registered owner or telemetry** creates orphaned capabilities that degrade without anyone noticing. Every skill in the catalogue must have a named owner and emit the minimum telemetry envelope on every invocation.

- **Chaining skills deeply without a workflow boundary** creates fragile composition. When orchestration needs state, implement a workflow skill instead of chaining read skills in a long sequence that has no compensation path.

SUMMARY

Closing Observations

Skills provide stability where models are probabilistic: explicit interfaces, policy enforcement, and operational evidence. When skills are the unit of capability, agent frameworks become replaceable and the platform becomes durable.

The engineering work, with contract design, schema validation, policy-as-code, instrumentation, and lifecycle management, is the same discipline that made SOA and microservices operationally trustworthy. What is new is the context: these capabilities are invoked by probabilistic reasoning engines in non-deterministic sequences. That makes the discipline more important, not less.

> *Build the contract first. Make the gateway the enforcement boundary. Instrument everything. Keep write skills behind propose/commit. Treat the catalogue as a product.*

APPENDIX A

Developer Checklist

Use this checklist before promoting any skill to production or publishing above local channel.

- ☐ Manifest complete and validated against the platform schema.
- ☐ Input validation and size limits enforced at the skill boundary.
- ☐ Policy rules written, tested in CI, and linked to skill version.
- ☐ Telemetry emits trace ID, invocation ID, latency, outcome, and cost tags for every invocation.
- ☐ Errors mapped to stable codes with HTTP status equivalents.
- ☐ Runbook written with dashboards, alert thresholds, rollback procedure, and on-call ownership.
- ☐ Integration tests cover backend failure modes (timeout, 5xx, malformed response).
- ☐ Contract tests enforce backwards compatibility against the published manifest.
- ☐ For write skills: propose/commit tested, approval gate verified, compensation action tested.
- ☐ For model skills: cost envelope declared and monitored; output quality tests in place.
- ☐ Owner and support contact registered in the catalogue.
- ☐ Deprecation policy documented with migration notes.

APPENDIX B

Error Code Catalogue Template

Code	HTTP	Meaning	Retry?
POLICY_DENIED	403	Authorisation or policy blocked	No
VALIDATION_FAILED	400	Input schema validation failed	No
BACKEND_TIMEOUT	504	Backend exceeded timeout budget	Yes
BACKEND_UNAVAILABLE	503	Backend is unreachable	Yes
DEPENDENCY_ERROR	502	Dependency returned error	Maybe
QUOTA_EXCEEDED	429	Quota reached	After backoff

APPENDIX C

Resilience Testing Scenarios

Scenario	Expected behaviour	Checks
Backend 503	Retry with backoff; BACKEND_UNAVAILABLE	No thundering herd; stable latency
Backend slow	Timeout; BACKEND_TIMEOUT	No runaway compute; budget honoured
Quota exhausted	Early reject; QUOTA_EXCEEDED	Audit log present; no backend call
Oversized payload	Reject; VALIDATION_FAILED	Payload not forwarded to backend
Bad dependency	Validate; DEPENDENCY_ERROR	Error logged; no partial writes

HITACHI

Hitachi Digital Services

Hitachi Digital Services, a wholly owned subsidiary of Hitachi, Ltd., powers mission-critical platforms with cloud, data, IoT, and ERP solutions, underpinned by advanced AI. With over 110 years of expertise, we drive innovation and growth for a more sustainable future.