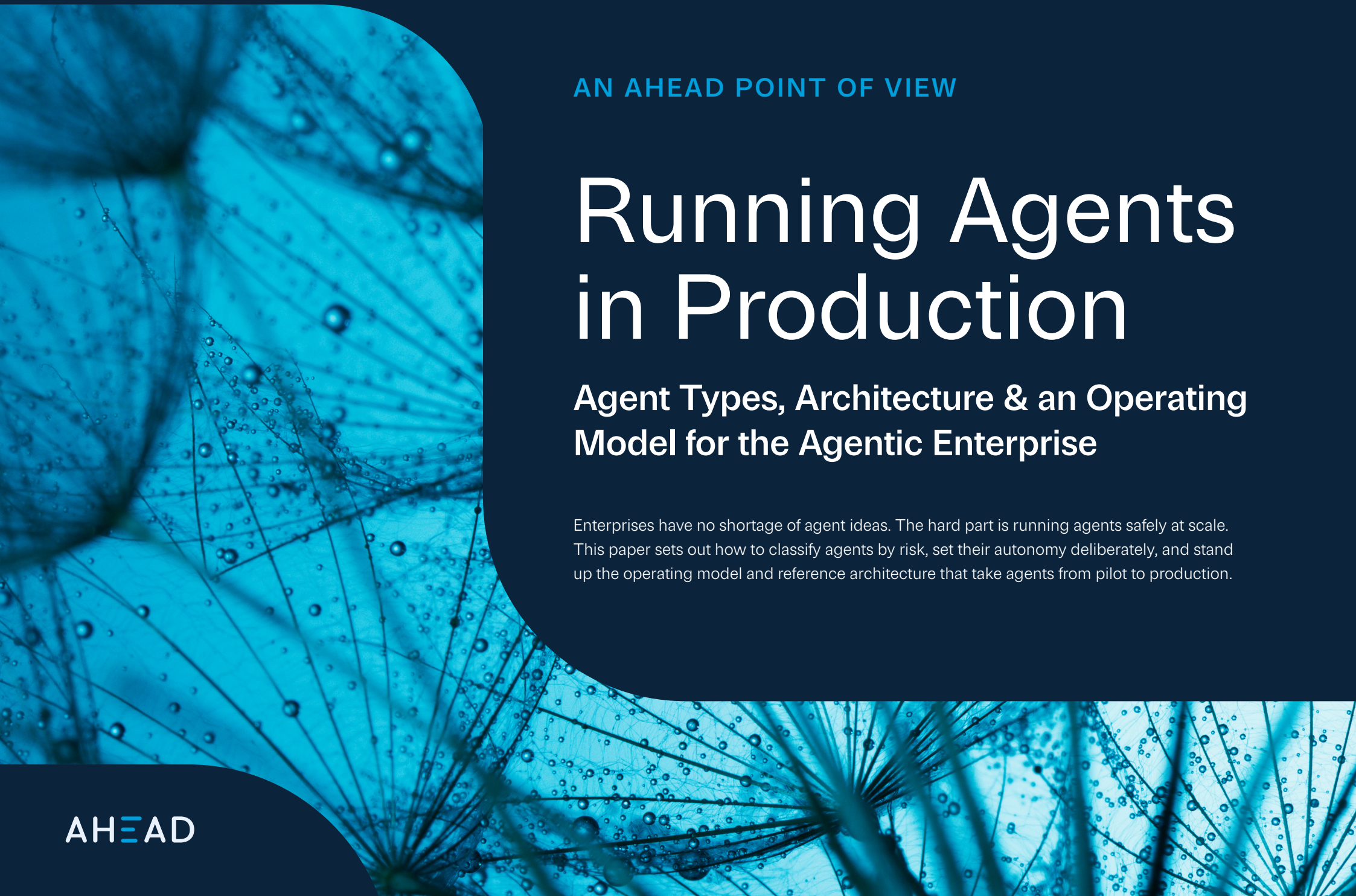


AN AHEAD POINT OF VIEW

Running Agents in Production

Agent Types, Architecture & an Operating Model for the Agentic Enterprise

Enterprises have no shortage of agent ideas. The hard part is running agents safely at scale. This paper sets out how to classify agents by risk, set their autonomy deliberately, and stand up the operating model and reference architecture that take agents from pilot to production.

AHEAD

Contents

I — The Argument in Brief

II — Understanding Agents: Types, Risk & Control

III — From Ideas to a Prioritized Portfolio

IV — Building Blocks for Agent Development

V — Running Agents in Production

VI — Where to Start

PART I

The Argument in Brief



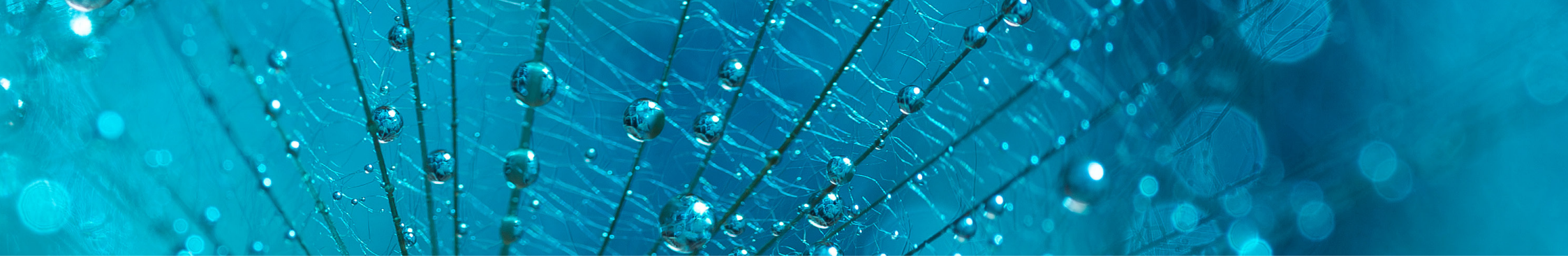
Most enterprises now generate more agent ideas than they can responsibly ship, and the gap between ambition and production has become measurable. Industry analysts expect that more than 40% of agentic AI projects will be canceled by the end of 2027, undone by escalating costs, unclear value, or inadequate risk controls[1], and Gartner now puts $234 billion of enterprise software spend at risk of agentic disruption by 2030.[2] Adoption surveys tell the same story from the other side: roughly a third of enterprises report agents deployed in production, while barely one in five have mature governance around them.[3] The decisive factor in whether an idea reaches production is the operating model: the way an organization classifies, governs, runs, and retires agents once they are live. Model capability is rarely the binding constraint now, and platforms to build on are plentiful.

Two debates consume more attention than they deserve. The first is which autonomy level an agent occupies, usually drawn as a ladder that climbs from simple prompting up to a fully autonomous harness. The second is which platform to standardize on. Both feel decisive. Both are mostly a distraction from the decisions that actually govern risk and value.

This paper makes a different case, positing that **three decisions** ultimately determine whether an agent reaches production and stays there safely:

1. Classify the agent by the risk it carries, which sets the controls before anyone writes code.
2. Set its autonomy deliberately and per task, judged against two properties: **whether the system can verify its own work** and **what a wrong action costs.**
3. Build the operating model that governs the entire fleet, covering lifecycle, governance, runtime management, and the platform engineering beneath.

Platform selection follows from these decisions. It does not lead them.



The paper is organized in six parts:

Part II gives the shared vocabulary an enterprise needs: what makes a use case agentic, the six agent classes and the risk each carries, the four control-flow positions that span deterministic workflows through agent-owned loops, and the crosswalk that connects them.

Part III turns a backlog of ideas into a prioritized portfolio, scoring candidate use cases on impact and AI readiness so that the strongest and most ready go first.

Part IV covers the building blocks of agent development, including the common patterns, the framework-runtime-harness stack, the anatomy of a harness, the minimum viable stack, the build practices that separate pilots from production, and a portfolio of worked examples.

Part V turns to operations: why the order of decisions is load-bearing, the AI-era operating model, the three jobs of running the fleet (build, govern, and operate), and the reference architectures for AWS, Microsoft Azure, Google Cloud, and NVIDIA on-premises.

Part VI is a short plan for the first ninety days.

This paper is meant to speak directly to the leaders who have to answer for agents in production: the platform owner, the head of AI or data, the security and risk leaders who sign off, and the architects who build. The through-line is simple. Maturity is the discipline to classify every agent, set its autonomy on evidence, and move it in either direction as that evidence accumulates.

PART II

Understanding Agents: Types, Risk & Control

What Counts as Agentic

An agent uses an AI model to pursue a goal across multiple steps. Given an objective, it decides which tools to call, evaluates the results, adapts, and continues until the goal is met or a human gate is triggered. Unlike a chatbot, an agent does not wait for the next prompt.

The Agentic Loop

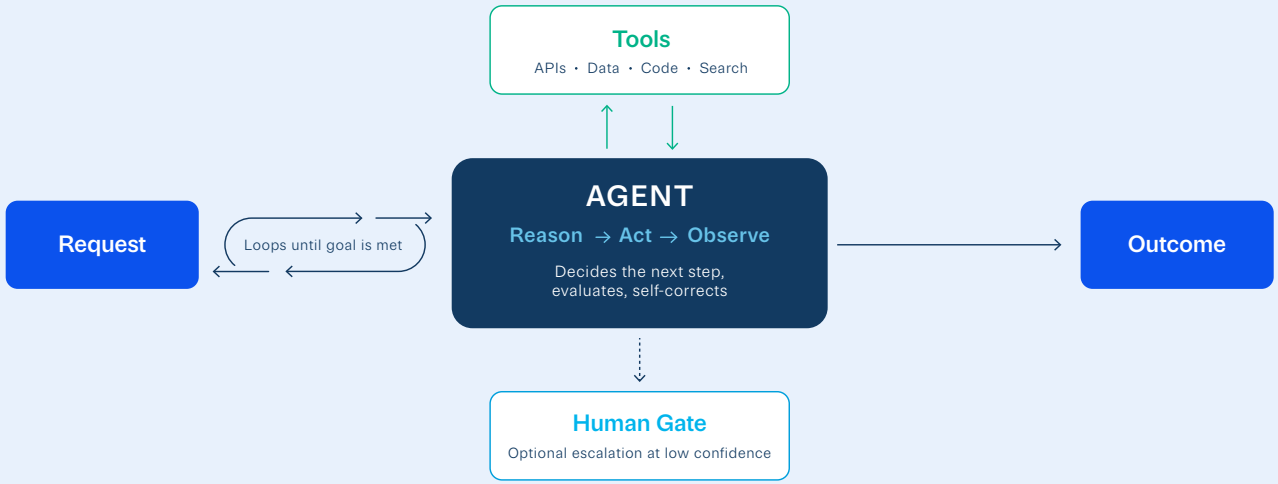


Figure 1. The agentic loop. The agent reasons, acts through tools, observes the result, and repeats until the goal is met or a gate fires.

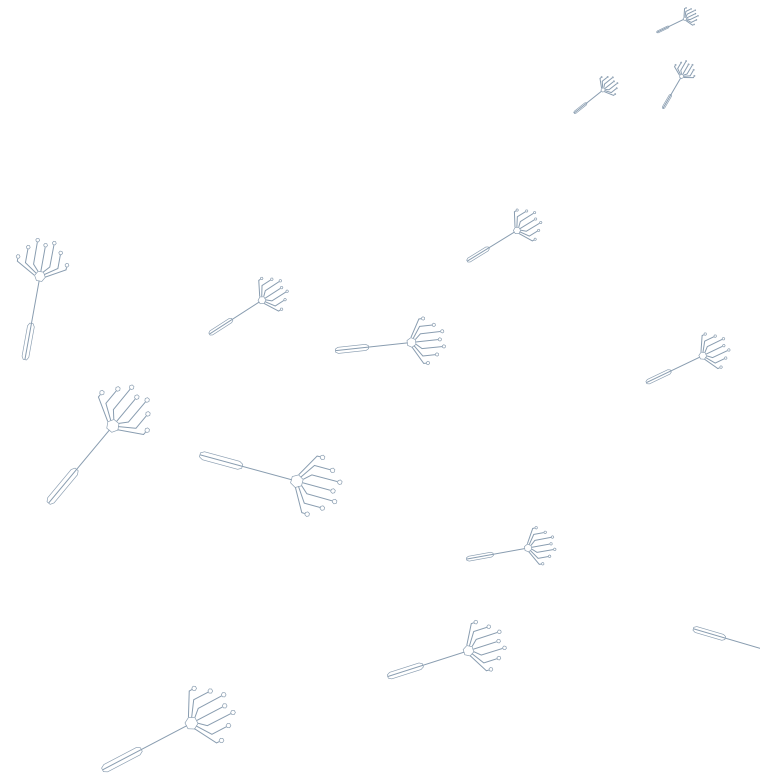
Four tests separate a use case that justifies an agent from one that does not:

1. **There is a Goal to Reach:** The system receives an objective and determines its own path, handling exceptions along the way.
2. **The Path Varies with Context:** It depends on intermediate results. If every step is known in advance, the work is automation.
3. **Human Involvement is Exception-based:** The agent handles the routine path while people enter at approval gates or for ambiguity. Constant prompting is a sign of a chatbot.
4. **The Use Case Spans Systems:** Agentic use cases almost always read from and write to several tools, APIs, or data sources. A single-system task rarely needs an agent.

Automation and agents solve different problems, and production systems use both. Automation owns the skeleton of a workflow: sequencing, orchestration, reliable and repeatable execution, system integration, and the audit trail that compliance depends on. Agents own the cognition: reasoning over unstructured input, deciding the next step at runtime, synthesizing across systems, and handling situations no one wrote a branch for. The most dependable production systems put *deterministic automation* around the parts that must be guaranteed and place *agent reasoning* where judgment is genuinely required.

Six Agent Classes & The Risk Each Carries

Every agent fits one of six classes, where the class functions as a permission statement. It describes what the agent is allowed to do and the oversight that requires. It deliberately says nothing about how the software is built, which is a separate question covered in the next section. Risk follows blast radius: what a wrong action costs and whether it can be reversed. The boundaries turn on what the agent can do without a human. A ‘Suggestion’ agent recommends and a person acts; an ‘Augmentation’ agent produces artifacts inside the workflow that a person still commits; only from the ‘Action’ class onward does the agent itself change something in a connected system.



The value of the class system is that it fixes the governance conversation early. A ‘Query’ agent needs audit logs, while an ‘Autonomous’ agent needs a kill switch, continuous evaluation, an executive sponsor, and a governance board. Building one set of controls for all six would over-engineer the easy cases and under-protect the hard ones. The class is decided first, and it travels with the agent for its whole life. The controls are bound to the agent through a non-human identity carrying scoped credentials (now a first-class capability on the major platforms), so the assigned class is the permission set that identity actually enforces. When an agent delegates to other agents, whether internal subagents or external agents reached over Agent2Agent, its class is the union of everything the whole constellation can touch. A delegating agent inherits the blast radius of every agent it can call.

Six agent classes, by risk

The class is a permission statement: what the agent may do, and the oversight that requires

RISK: 1 | LOW

Query / Assistive

Read-only Q&A over documents or data. No writes.

REQUIRED CONTROLS

Audit logs, access scope, labels

WORKED EXAMPLES

- Weekly status roll-up
- Cross-system reconciliation
- Analytics on demand
- Pre-sales research library

RISK: 2 | LOW TO MEDIUM

Suggestion

Surfaces a recommended action and a human executes it.

REQUIRED CONTROLS

Audit, explainability, human review

WORKED EXAMPLES

- Next-best-action prompts
- Renewal-risk flagging

RISK: 3 | MEDIUM

Augmentation

Co-pilot inside an existing workflow.

REQUIRED CONTROLS

Approved models, prompt hardening, output validation

WORKED EXAMPLES

- Invoice narrative generation
- Contract metadata extraction
- Proposal drafting

RISK: 4 | MEDIUM TO HIGH

Action / Tool-using

Performs one bounded action through tools or APIs.

REQUIRED CONTROLS

Sponsored agent ID, scoped tools, gateway, approval gate

WORKED EXAMPLES

- Help desk ticket triage
- Expense exception handling
- Security alert triage
- In-CI tooling changes

RISK: 5 | HIGH

Multi-step Agentic

Plans and executes a workflow across systems.

REQUIRED CONTROLS

Eval-as-code, per-step gates, rollback, full trace

WORKED EXAMPLES

- Root cause investigation
- Codebase maintenance

RISK: 6 | CRITICAL

Autonomous

Operates continuously without per-action review.

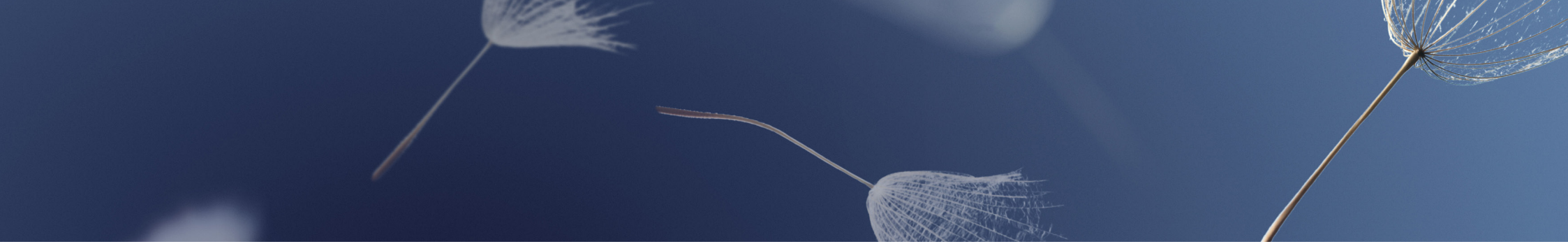
REQUIRED CONTROLS

Kill switch, continuous eval, sponsor, governance board

WORKED EXAMPLES

- Self-healing runbook execution
- Continuous replenishment within policy

Figure 2. Six agent classes ordered by risk, from read-only assistance to continuous autonomous operation.



Deterministic to Agent-owned: The Autonomy Dial

The architecture question is who owns the path – the code or the model? Autonomy is a dial set per task, and the setting is determined by the same two properties that drive risk:

> **Verifiability: Can the system check its own work?**
>
> **Blast radius: What does a wrong action cost, and is it reversible?**

High verifiability and low blast radius earn more autonomy. Everything else gets explicit control flow with human gates at the irreversible steps.

A popular shorthand arranges agents on a ladder of levels and implies that a higher level is more mature. However, that framing misleads, as it conflates three independent questions: how autonomous the agent is, what risk it carries, and what layer of the stack it is built on. It implies that more autonomy is better despite the fact that a deterministic workflow is the correct architecture for most enterprise processes. Further, it collapses under a single counterexample, since a complete agent harness is itself built on a framework running the same loop that the ladder treats as a lower rung. Four control-flow positions describe the real choices.

The Four Control-Flow Positions

P1 Deterministic Workflow

Code owns every step.
The model transforms data at fixed points.



LLM transforms at one fixed step

GUARDRAIL

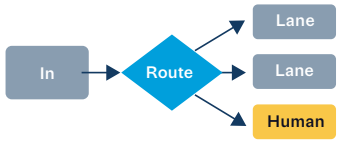
Schema validation, confidence thresholds to a human queue, no model write access.

SIGNAL IT IS MISPLACED

Branching on model output starts to appear: promote to P2.

P2 Model-routed Workflow

Code owns the steps.
The model picks the branch.



Default human lane

GUARDRAIL

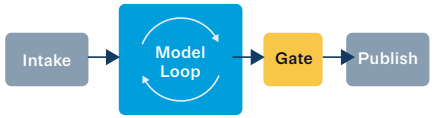
Every branch is a designed path, a default human lane, a tracked misroute rate.

SIGNAL IT IS MISPLACED

Lanes multiply past what you can enumerate: promote to P3.

P3 Bounded Agent

Model owns the path inside a phase.
Code owns the transitions.



Gates at every phase transition

GUARDRAIL

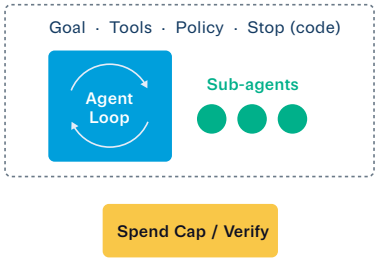
Phase boundaries in code, read-only or sandboxed access, a gate at each transition.

SIGNAL IT IS MISPLACED

A gate is rubber-stamped on verifiable output for two quarters: consider P4.

P4 Agent-owned Loop

Model owns the loop, tools, and sub-agents. Code sets goal, tools, policy, stop.



Verification outside the model

GUARDRAIL

Spend caps, step limits, sandboxed credentials, verification outside the model.

SIGNAL IT IS MISPLACED

Prompts begging the agent not to do something: the permission system is missing.

Code-owned Model-owned Human Gate

Figure 3. The four control-flow positions. Gray is code-owned, blue is model-owned, gold is a human gate.

The dial moves in both directions. A phase that has collapsed to two predictable calls should be demoted to a workflow. A gate that adds nothing on consistently verifiable output is a candidate for promotion. The point of naming the positions is to make those moves deliberate and evidence-based rather than accidental.

The Crosswalk: Class to Position

Class and position correlate, and they do not collapse into each other. Class follows blast radius, and position follows loop ownership. The same class can be implemented at several positions, and the controls the class requires stay constant no matter the position.

	P1 Deterministic Workflow	P2 Model-routed Workflow	P3 Bounded Agent	P4 Agent-owned Loop
Class 1 Query	TYPICAL	TYPICAL	TYPICAL	TYPICAL
Class 2 Suggestion		TYPICAL	TYPICAL	
Class 3 Augmentation	TYPICAL	TYPICAL	TYPICAL	
Class 4 Action		TYPICAL	TYPICAL	
Class 5 Multi-step			TYPICAL	TYPICAL
Class 6 Autonomous				TYPICAL

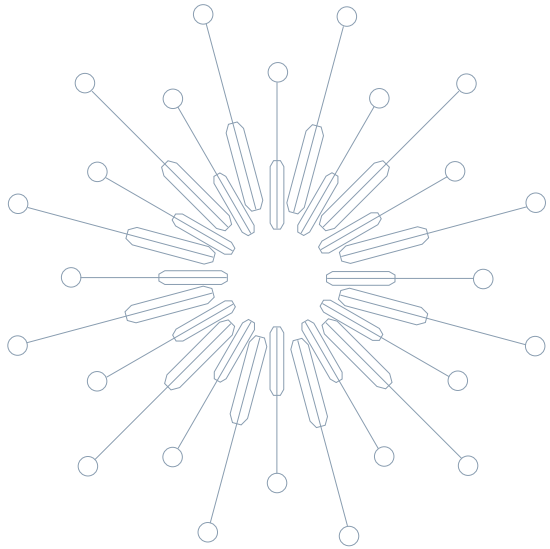
The clearest case: a **Class 1** analytics agent can run a full P4 loop because it reads everything and writes nothing.

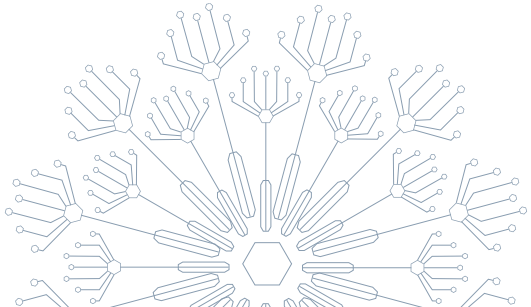
Figure 4. Typical architectural position by agent class. The band shows where each class is normally implemented.

Class 1 spans the entire dial. A fixed retrieval pipeline answering policy questions is a P1 workflow. An analytics agent that writes its own queries against the warehouse with read-only credentials is a P4 loop. Both are Class 1, because neither can write anything. Class 2 and Class 4 cluster around P2 and P3, where a model adds judgment, but a human or a gate fires the action. Class 5 lives at P3 and P4, as either a chain of bounded phases with per-step gates or an owned loop with hard verification. Class 6 is P4 made continuous, which is why its controls are the P4 guardrails turned permanent and organizational.

> **The most architecturally autonomous agents can be the lowest-risk agents, because they read everything and write nothing. That single observation defuses most anxiety about the word "autonomous."**

The combined rule for any proposed agent is short. Classify it first, which sets the controls. Position it second, which sets the architecture. Then, confirm that the position's verification mechanism can carry the class's risk. A Class 5 use case without a verifier outside the model is not ready for a P4 loop, however capable the model is.





PART III

From Ideas to a Prioritized Portfolio



The Portfolio Comes First

Enterprises rarely lack agent ideas. What they lack is an agreed way to choose among them. Part I noted that a large share of agentic projects are expected to be cancelled, undone by cost, unclear value, and weak controls rather than by model capability. The projects that stall are seldom the ones best matched to the organization's readiness, but rather, the ones chosen by enthusiasm. Therefore, the first move, before a single platform or architecture decision, is to turn the backlog into a ranked portfolio, because impact and readiness vary enormously from one idea to the next and the wrong first project spends credibility the program cannot easily win back.

AHEAD runs this at the level of the strategic area rather than the whole enterprise or a single use case. The enterprise is too broad to move quickly; a lone use case is too narrow to show material impact or create reuse. A strategic area is wide enough to matter to a sponsor and narrow enough to execute in a two-to-three-quarter window, with data and change management synergies that make each subsequent solution easier. The highest-priority areas are then decomposed into individual use cases and scored the same way, and the work proceeds in tight waves for a handful of areas at a time, so the organization builds evidence and reusable foundations before it widens the front.

A Shared Scoring Model

Every candidate is scored on the same criteria, grouped into **six categories:**

1. **Business Impact**
2. **Technical & Data Readiness**
3. **Organizational Readiness**
4. **Component Reusability**
5. **Legal & Risk**
6. **Agentic AI Fit**

Each criterion is rated High, Medium, or Low against a shared rubric with evidence examples, then weighted and totaled into a priority score and tier. The weighting is deliberate: business objective and agentic AI potential carry the most weight, so an idea rises to the top only when it matters to the business and genuinely benefits from an agent rather than from simpler automation.

Prioritize Use Cases Before Platforms

Score every candidate the same way, then place it on the impact X AI-readiness matrix. The strongest and most ready go first.

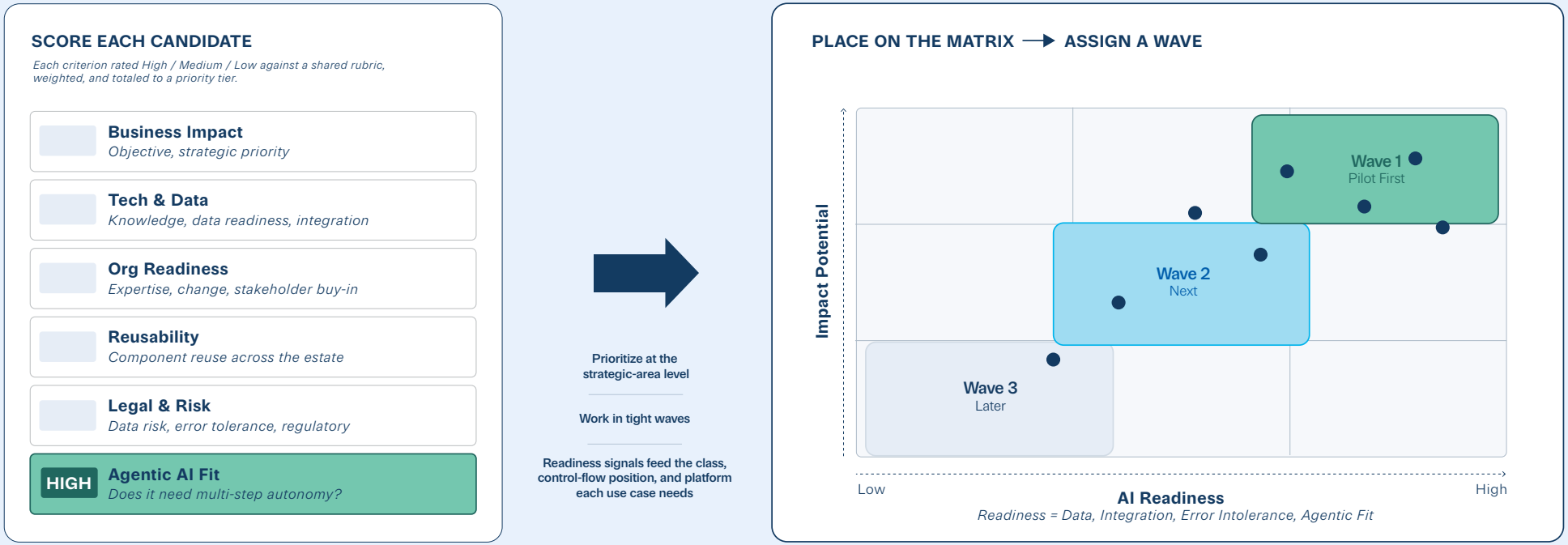


Figure 5. Prioritizing use cases. Each candidate is scored on the same weighted criteria, then placed on the impact and AI-readiness matrix to set its wave.



Impact & AI Readiness

Two of those signals do the most to sort the portfolio, and they become the axes of the prioritization matrix: **impact potential** and **AI readiness.** Impact is a top-down estimate of the value of addressing an area end to end. AI readiness combines the operational signals, process complexity, data readiness, systems integration, and error tolerance that decide whether an idea can actually reach production rather than stall in a pilot. Plotting each scored candidate on this matrix sorts the portfolio into waves, and the high-impact, high-readiness quadrant is where the first pilots belong.

Prioritization is not a detour before the agent work, but the front of the same funnel. The readiness signals that decide a use case's wave, how clean its data is, how many systems it must touch, how tolerant its process is of error, and whether it truly needs multi-step autonomy are the same signals that decide its agent class, its control-flow position, and ultimately, its platform. Understanding the use case is how an enterprise learns the architecture it will need. The remainder of this paper turns to that architecture, beginning with the building blocks that carry a prioritized use case from idea to production.

PART IV

Building Blocks for Agent Development

Ten Patterns

A small set of patterns underlies almost every enterprise deployment, serving as the shared vocabulary for design reviews. Most production agents compose only two or three of them.



Figure 6. Ten agentic patterns. Most production agents compose two or three.

The first three patterns structure the work:

- **Prompt Chaining** runs sequential model calls where each output feeds the next
- **Routing** sends an input to the right specialist
- **Parallelization** runs concurrent agents and aggregates or votes on the result

Patterns 4, 5, and 10 control quality and risk:

- **Orchestrator/Subagent** has a central planner delegate to specialists
- **Evaluator/Critic** pairs a generator with a reviewer (whether a separate agent or the same agent reflecting on its own output against explicit criteria before it proceeds)
- **Human-in-the-Loop** inserts checkpoints in high-stakes flows

The remaining patterns (6 through 9) drive execution:

- **ReAct** cycles through reason, act, and observe with tool calls
- **Plan and Execute** decomposes the whole task before acting
- **Tool and Function Calling** invokes APIs, data, and code in a structured way
- **Memory-augmented** agents read and write short-term, long-term, and episodic memory (managed through the context engineering that decides what enters the model's working window and what is summarized or retrieved)

Consider a typical deployment to understand how these patterns chain together:

Router classifies the incoming event and selects a workflow → Orchestrator decomposes the goal and delegates to subagents that use ReAct and Tool Calls to gather data and act → Evaluator checks the output before it surfaces, and a Human-in-the-Loop checkpoint fires when confidence is low or the action is high-risk → Memory layer carries context across the run, and Parallelization fans subagents out across data sources before results are merged



The Stack: Framework, Runtime, Harness

Three things are often discussed as competitors when, in truth, they are layers:

- A **framework** provides the parts: the loop, tool calling, state, and composition
- A **runtime** provides durable execution: checkpointing, retries, and resumability for work that runs longer than a single request
- A **harness** is the complete operating environment around the loop: context management, permissions, and the verification logic that makes the agent trustworthy

A modern harness is typically built on a framework and a runtime. They stack.

This is important because the agentic loop itself is simple. The engineering value is everything wrapped around it. Harness improvements alone have produced benchmark gains on the order of a model generation, and as such, the harness should be treated as the product. The next section opens the harness up and names its parts.

The wider ecosystem spans six layers, each with several credible options:

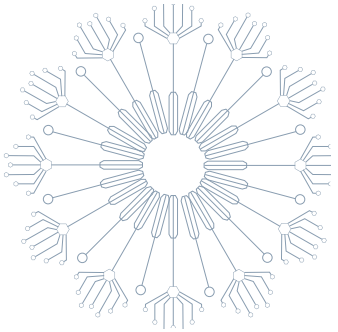
- Frameworks and SDKs
- Managed runtimes
- Durable orchestration
- Interoperability protocols
- Models
- Evaluation, guardrails, memory, and observability

Two protocols are now effectively table stakes, and both are governed under the Linux Foundation's Agentic AI Foundation (formed in December 2025 around MCP), which the Agent2Agent protocol formally joined in August 2026, with its 1.0 specification adding signed agent cards that give delegation a verifiable trust mechanism:[4]

- The Model Context Protocol for connecting tools
- The Agent2Agent protocol for agent-to-agent communication

Companion efforts cover agent identity, and payments standardization moved to the FIDO Alliance through the Agent Payments Protocol. Choosing well at each layer makes a more significant impact than picking any single best tool.

The framework market splits into two camps. Hyperscaler SDKs from the major model and cloud providers offer the fastest path to value and managed runtimes, at the cost of some lock-in. Open-source frameworks offer portability and the deepest production tooling, at the cost of running more yourself. A reasonable default is to favor portability and choose a hyperscaler SDK when the organization is already deep in one cloud, needs that vendor's managed runtime, or is working to a hard deadline.



Anatomy of a Harness

An agent today is a model plus a harness. The model supplies the reasoning. The harness is everything that turns that reasoning into work an enterprise can trust. Eight components recur in every serious implementation.

Four manage cognition:

- **Loop Control** runs the reason, act, observe cycle and owns the stop conditions
- **Context Engineering** decides what enters the model window, with compaction keeping long sessions coherent
- **Memory** carries session state and persistent knowledge across runs
- **Planning and Delegation** decompose the goal and dispatch subagents

Four control action:

- **Tool Execution** makes schema-checked calls over MCP with side effects sandboxed
- **Verification** proves the work through tests, schema checks, and citations that resolve
- **Permissions and Gates** enforce policy as code with human approval at irreversible steps
- **Isolation and Telemetry** scope the credentials, deny stray egress, and trace every action taken



Figure 7. Anatomy of a harness. Eight components turn a raw model into a system that can be trusted in production.



The build-versus-buy calculus has shifted. When this framework was first drafted, building a harness meant assembling those components on a framework and runtime. Managed harnesses are now shipping products: AWS made the AgentCore Harness generally available in June 2026 on the open-source Strands framework, the Microsoft Agent Framework reached general availability with its own agent harness, and Anthropic describes Claude Managed Agents as a pre-built, configurable harness.[5] The guidance follows directly: start from a managed harness or a reference implementation rather than building the loop from scratch, and spend the engineering budget where differentiation actually lives within the tools, policies, and verifiers specific to the domain. The loop is table stakes. The verifier that knows what a correct answer looks like in your business is not.

The harness is also where Part II's architecture becomes enforceable. At Positions 3 and 4, where the model owns some or all of the path, the harness is the control surface: the P4 guardrail list (spend caps, step limits, sandboxed credentials, and verification outside the model) is a harness specification in miniature. An enterprise evaluating any agent product, managed or self-built, can use the eight components as its evaluation checklist.

Start with the Minimum Viable Stack

Most agents need five components and no more: (1) a framework or a direct model API, (2) tools exposed through MCP, (3) evaluations that run offline and online, (4) guardrails on input, output, and tool use, and (5) observability with tracing and metrics. Everything beyond this is added only when a specific trigger appears.

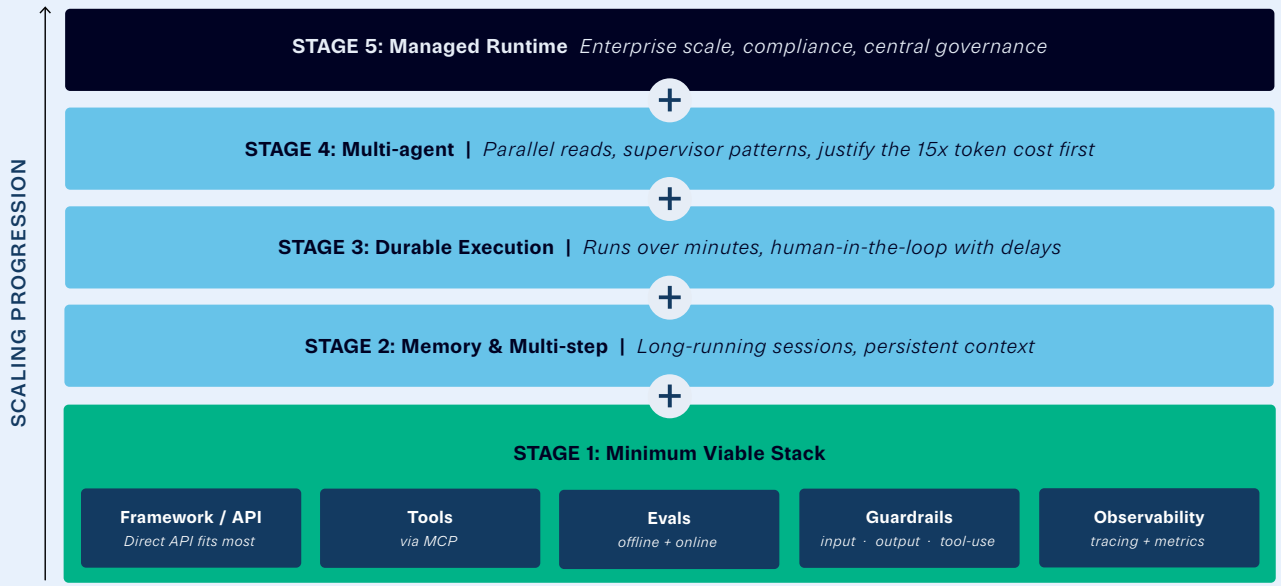


Figure 8. The minimum viable stack, and the staged progression. Stay at the lowest stage that solves the problem.

The progression has clear triggers:

- Add **memory** and **multi-step handling** when sessions run long and context must persist.
- Add **durable execution** when runs last more than minutes, when human approval introduces delays, or when an audit trail is required.
- Add **multi-agent structure** only when the value clears its cost. By Anthropic's own engineering data, a single agent run uses roughly four times the tokens of a chat exchange, and a multi-agent run roughly fifteen times,[6] so multi-agent pays off mainly for parallel exploration where reads dominate writes.
- Add a **managed runtime** when you need session isolation, identity, and central governance at multi-team scale.

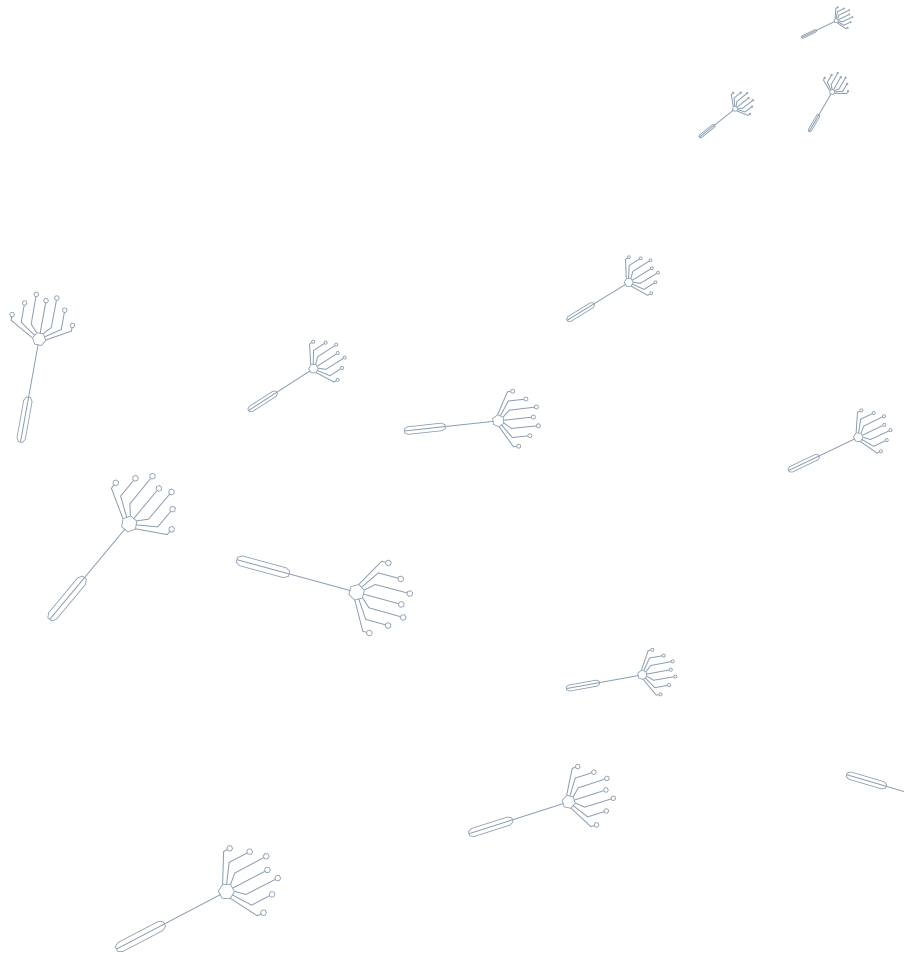
Each added layer carries real operating cost, so each stage should be triggered by evidence rather than anticipation.

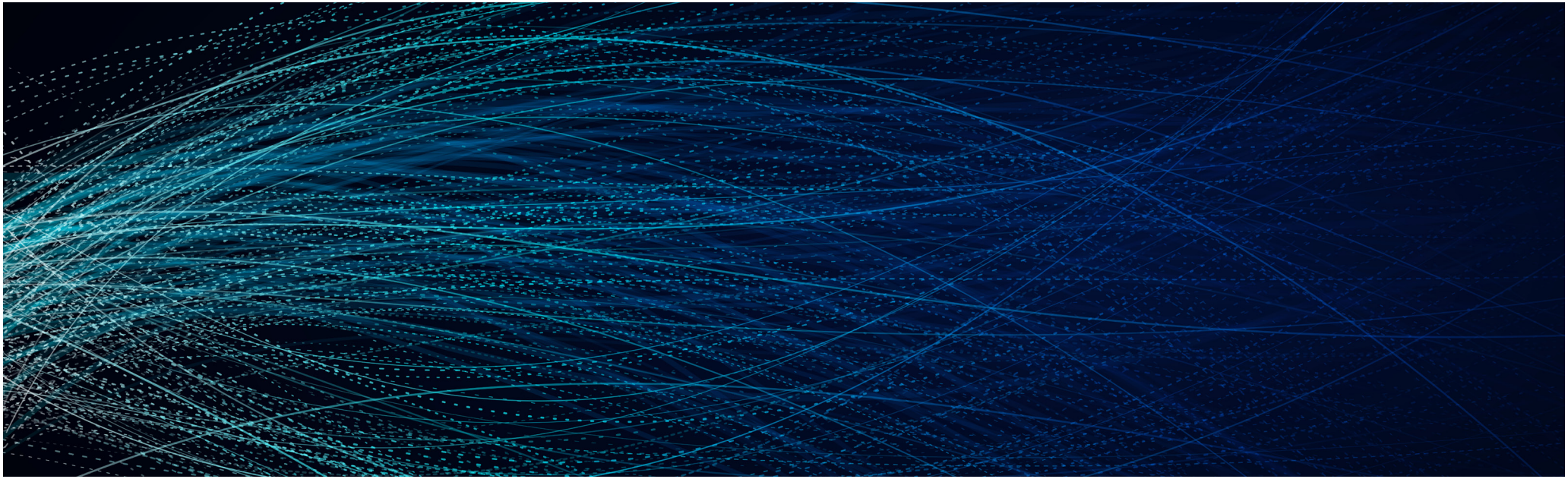
Two investments pay for themselves before scale does. The first is evaluation. Offline regression testing against a curated golden dataset should run on every prompt, tool, or model change, combining deterministic checks with judge-based scoring that applies to the whole trajectory rather than only the final answer. Judge-based scoring needs its own discipline, because a model grading a model tends to drift. Calibrate the judge against human labels, keep a written rubric, and audit it periodically instead of trusting it blindly. Online monitoring should sample a defined share of production traces, watch for drift in cost, latency, and tool-call patterns, and feed real failures back into the next evaluation cycle. A dedicated evaluation and observability layer, whether a managed service or an open-source stack, is where these datasets, traces, and scores live. More agent projects degrade quietly from missing evaluations than from a poor framework choice.

The second is guardrails, which sit on five surfaces:

- **Input** guardrails screen for prompt injection, jailbreaks, and sensitive data before the model sees the text.
- **Output** guardrails check grounding, toxicity, leakage, and format.
- **Dialog** guardrails hold topic and policy across a long conversation.
- **Retrieval** guardrails enforce source attribution and citation accuracy.
- **Execution** guardrails govern tool use through scope limits, approval gates, budget caps, and dry-run modes.

Execution is the most overlooked surface in current deployments but actually the most consequential, because it is where an agent can actually cause harm. Tool-use guardrails should be treated as deploy-blockers.





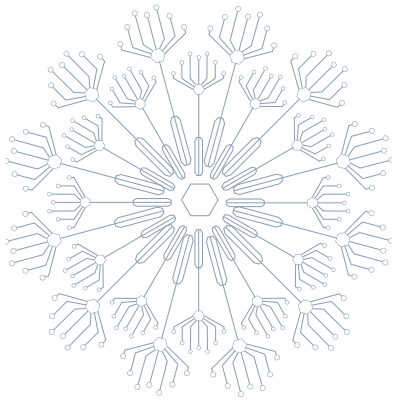
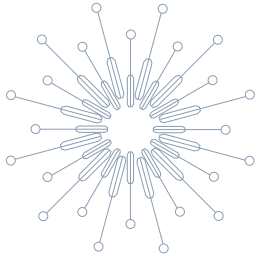
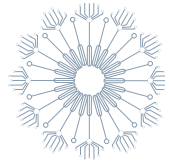
Build Practices

A handful of practices separate agents that reach production from agents that stall in a pilot:

- **Start at the lowest position that solves the problem.** Workflows are cheaper to build, test, and audit. Move the dial up only when the task genuinely needs open-ended pathing and the outcome is verifiable.
- **Separate reasoning from permissions.** The model proposes; a deterministic tool layer disposes. Enforce permissions, spend caps, and write authority via policy as code, never as instructions in a prompt.
- **Make every irreversible action a gate.** Sends, writes, deletes, and publishes each get a human approval interrupt or a verified dry-run path.
- **Use durable execution for anything long-running.** Checkpointing and resumability are runtime features. Production software does not lose three hours of work to a transient API failure.
- **Classify and route errors deliberately.** Transient errors retry with backoff; Model-recoverable errors return to the loop as a tool result; User-fixable errors interrupt; Unexpected errors halt and page. Cap the retries.
- **Instrument from day one.** Capture telemetry on every tool call, token spend per task, step counts, task success, misroute rates, and how often humans edit what the agent proposed at a gate. This telemetry is what lets you move the dial with evidence.

Worked Examples

The clearest way to see the framework at work is a portfolio mapped across the four positions. Each use case below carries its position, its agent class, and the verification that contains its blast radius. The examples are drawn from the systems most enterprises already run: project and operations tooling, CRM, ITSM, finance, HR, and the delivery toolchain.


POSITION 1:

Deterministic Workflow

USE CASE	CLASS	HOW IT WORKS
Invoice narrative generation	3	A billing run pulls approved entries; the model converts notes to client-readable narratives at one fixed step; output is validated against the source. The model never touches amounts, rates, or routing.
Weekly status roll-up	1	A scheduled pull of status, milestones, and risks; the model summarizes each item into a fixed template and posts to a leadership channel.
Contract metadata extraction	3	Signed agreements trigger extraction of term, value, and renewal dates into the system of record. Schema validation catches malformed output; low-confidence fields route to a human queue.

POSITION 2:

Model-routed Workflow

USE CASE	CLASS	HOW IT WORKS
Help desk ticket triage	**4**	Classifies tickets by category, urgency, and affected system, then routes with priority set. Misroutes are recoverable, and the misroute rate is the quality metric.
Expense exception handling	**4**	Sorts reports into clean, exception, and anomaly. Clean items flow to auto-approval under the existing threshold; exceptions route to the right approver with the policy flag named.
Security alert triage	**4**	Sorts alerts into noise, investigate, and escalate. The escalate lane pages a human immediately. The model reduces the queue. It cannot suppress the page.

POSITION 3:

Bounded Agent

This is the workhorse position for enterprise systems – an open-ended middle for research, investigation, or drafting, wrapped in a controlled beginning and end.

USE CASE	CLASS	HOW IT WORKS
Proposal drafting	**3**	After qualification, an agent searches past work and assembles draft sections mapped to the requirement list. It cannot set pricing or send anything. Review stays a separate code-owned phase with a human gate.
Root cause investigation	**5**	On incident closure, an agent with read-only access to logs, traces, metrics, and the change calendar correlates across systems and produces a cited hypothesis. Remediation is a separate gated phase. The worst case is a wrong hypothesis the reviewer catches.
In-CI tooling changes	**4**	An agent owns a defined change: write code, run tests, iterate to green. The merge gate is the test suite plus a human reviewer. No merge, deploy, or out-of-scope access.

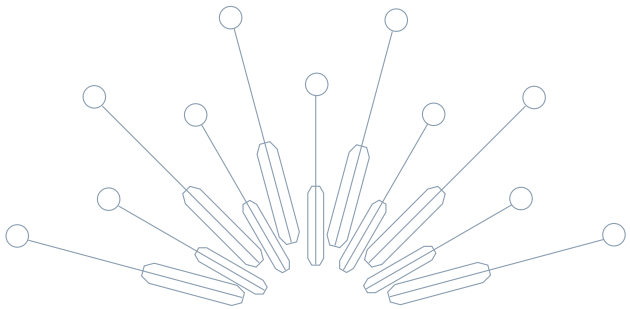
POSITION 4:

Agent-owned Loop

The bar for an owned loop is verification outside the model's own confidence and a blast radius contained by architecture rather than trust. Candidates cluster in two areas: software work, where tests verify, and read-only investigation, where the output is a cited report.

USE CASE	CLASS	HOW IT WORKS
Codebase maintenance	5	Dependency upgrades and refactors across internal repositories. Sandboxed branch, full test suite, iterate to green, open a pull request. CI verifies; the branch never auto-merges.
Cross-system reconciliation	1	Period-end discrepancies across project, CRM, and finance systems. Read-only queries across all three; the agent owns its investigation path and cites the underlying records, so verification is checking the citations. Nothing is written back.
Analytics on demand	1	Question answering over the warehouse with read-only credentials. The agent owns query construction and iteration, and the query ships with the answer, so wrong answers are visible.
Pre-sales research library	1	Deep research across past proposals and public information, assembling an answer library mapped to a requirement list. Output feeds the drafting agent, so the blast radius is zero.

Notice the class column. Three of the four owned-loop examples are Class 1. The most architecturally autonomous agents in the portfolio are among the lowest-risk, because they read widely and write nothing. This is the practical route to autonomy: start with read-only investigation, gather the telemetry, and let the evidence justify each later step.



Two Operations Scenarios End to End

SCENARIO 1:

Joiner, Mover, Leaver

Onboarding and offboarding touch HR, identity, mailbox, group membership, hardware, and licensing, with one orchestrated sequence that handles it:

An HR system event triggers the flow → Identity creates or modifies the account and assigns role-based groups → Mailbox and collaboration access is provisioned or removed, and licenses are assigned or reclaimed → Application access is granted or revoked across SaaS systems → A hardware ticket is opened or a device is wiped, and the asset record is updated

On a leaver, a human gate requires the manager and security to confirm before token revocation executes. Every action is logged into the security information and event management system, the ticket is closed, and the run confirms zero orphaned permissions.

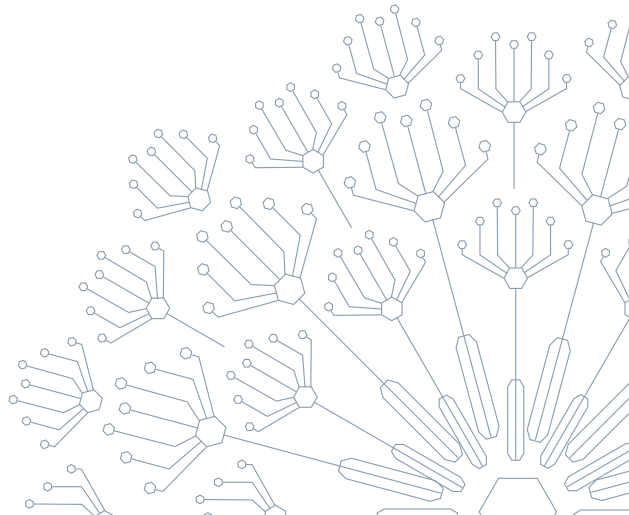
PATTERNS:	CONTROLS:
• Orchestrator and Subagent • Plan and Execute • Tool Calling • Human-in-the-loop • Memory	• Gate on revocation • Scoped permissions per step • Full audit log • Rollback on each action

This is the scenario security and compliance buyers tend to fund first, because the manual gap is one they recognize immediately.

SCENARIO 2:

Cross-domain Incident Triage

This proves correlation rather than deflection. An incident fires (e.g., a report that an application is slow). The agent pulls the record and the affected configuration items, then queries observability for anomalies in the blast radius and pulls recent change records. It correlates the signals to a specific layer, such as storage latency, a saturated network link, or a bad deploy, and outputs a ranked hypothesis. It proposes a fix and classifies the risk: a low-risk configuration rollback executes automatically, while a high-risk change, such as a failover, is gated by human approval. Execution runs through the existing automation and vendor APIs, and the agent re-queries observability to confirm the signals normalize before closing. The differentiator is reach. An agent confined to the ticketing system can only summarize. An agent that ingests the ticket and then acts across network, storage, compute, and cloud through scoped credentials can find and fix the actual cause.



PART V

Running Agents in Production

Why the Operating Model Dictates Outcomes

The decisions of the previous parts sequence in a specific order, and the order is load-bearing:

1. Identify the use cases first
2. Classify each by pattern and risk
3. Choose the platform where each use case naturally lives
4. Build the operating model around the agents you are actually shipping

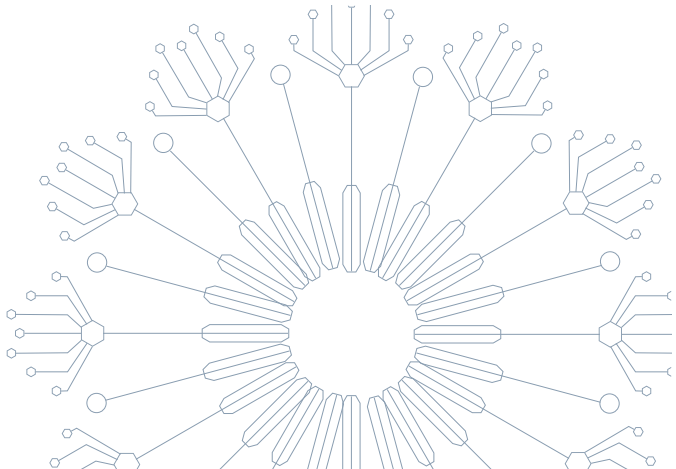
Each step constrains the next, and going backwards is what wastes quarters.

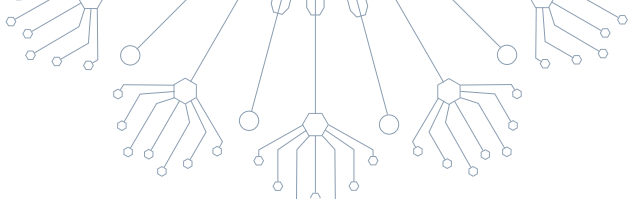
Skipping a step has a predictable cost:

- Skip the use cases and you build governance for agents that never ship.
- Skip classification and you over-engineer the platform.
- Skip the platform fit and you end up with a different stack for every use case.
- Skip the operating model and you get ungoverned agent sprawl.

The trap that shows up most often is building the platform first, which creates elaborate controls for use cases that never arrive and under-serves the ones that do

> **Use cases come first. Then patterns. Then platforms. Then the operating model. The order is load-bearing.**





On platforms, the honest position is that most enterprises end up multi-platform, and the choice is made per use case rather than per organization. Five families each have a sweet spot.

PLATFORM FAMILY	SWEET SPOT
Microsoft	When the Microsoft 365 footprint dominates and users live in Teams and Office. Native identity, data-loss prevention, and governance are configured once.
AWS	When core workloads run on AWS. Tight integration with identity, networking, and the Bedrock model catalog. Strongest for AWS-anchored data and infrastructure agents.
Google Cloud	When data lives in BigQuery or Gemini is the model of choice. Strongest for analytics-heavy, data-anchored agents.
ServiceNow	When the use case lives inside IT service, operations, customer, or HR workflows. The agent and the system of record sit together, and Action Fabric now opens the platform to external agents over MCP and A2A.
NVIDIA On-prem	When sovereignty, air-gap, or GPU economics push workloads on-premises. Pairs with an on-prem gateway and enterprise identity.

The market has moved to meet this argument. In the ninety days before this edition closed, every major platform shipped productized fleet control: Microsoft's Agent 365 control plane; gateway rate limits and sequence-aware temporal policies on AWS; Google's Agent Registry and Agent Observability; session-level budgets from the model vendors; and a Model Context Protocol revision designed to enable gateways to route and meter agent traffic.[7] The control plane is becoming something you buy. The operating model that runs it, the classification discipline, the evidence-based autonomy decisions, and the ownership model, is still something you build. **That distinction is the work of the rest of Part V.**

The AI-era Operating Model

As opposed to requiring a separate operating model, agents simply need the existing one extended, with a few new pillars added. The diagram below shows the shape: familiar pillars carried forward and adapted for agents, with net-new capabilities marked.

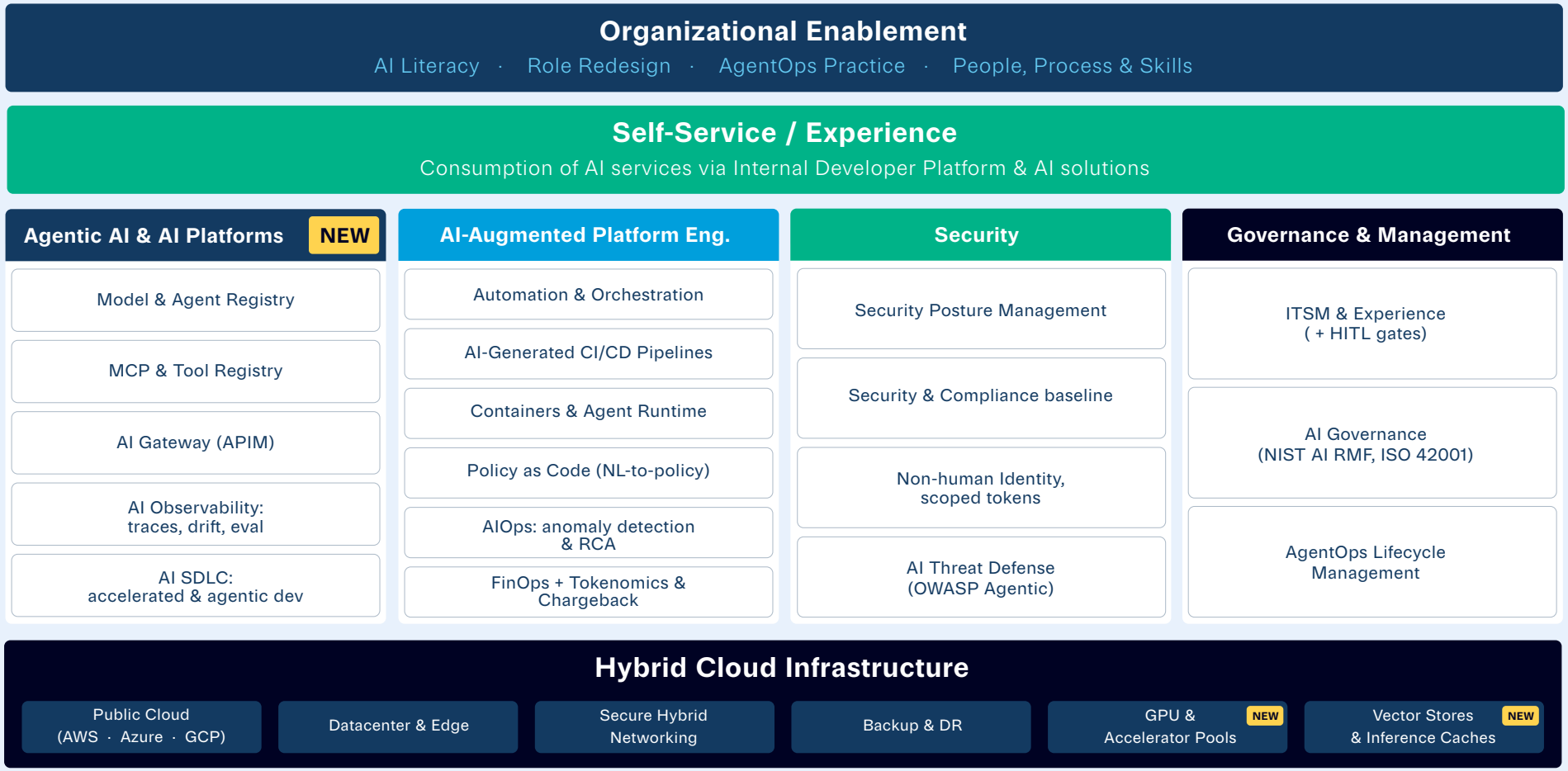


Figure 9. The AI-era operating model. Existing pillars extended for agents, plus net-new capability.

Organizational Enablement adds AI literacy, role redesign, and an AgentOps practice to the existing investment in people, process, and skills.

Agentic AI & AI Platforms is largely new: a model and agent registry, an MCP and tool registry, an AI gateway, and AI observability for traces, drift, and evaluation results, alongside an AI-aware software development lifecycle.

Self-service & Experience delivers these capabilities through the internal developer platform.

Hybrid Cloud Infrastructure carries forward public cloud, datacenter and edge, secure networking, and backup and disaster recovery, and adds two new elements: GPU and accelerator pools, and vector stores and inference caches.

Security extends posture management and baseline compliance with two agent-specific needs: (1) non-human identity with scoped tokens, and (2) AI threat defense against prompt injection and the agent-specific threats now catalogued in community references (the OWASP GenAI LLM Top 10, 2026 edition; the OWASP Top 10 for Agentic Applications; the MAESTRO threat-modeling framework; the SAFE-MCP catalog of attacks on the tool protocol)[8].

Governance & Management extends IT service management with human-in-the-loop gates, adds AI governance mapped to recognized frameworks, and adds AgentOps lifecycle management.

AI-augmented Platform Engineering brings AI into automation, pipeline generation, runtime, policy as code, and operations, extending financial operations with tokenomics and per-agent chargeback.

The shared point is that one operating model has to govern both the agents that serve the business and the agents that run the infrastructure, because they share the same plumbing: **identity, governance, platform**, and **data.**

Running the Fleet: Build, Govern & Operate

The operating model does three jobs: **build, govern,** and **operate**. Each answers a different question, is owned by a different role, and produces artifacts that can be checked. An operating model that cannot name its owners and its artifacts is a slideware operating model, which is why each job below carries all four: the question, the owner, the artifacts, and the measure that proves it is working.

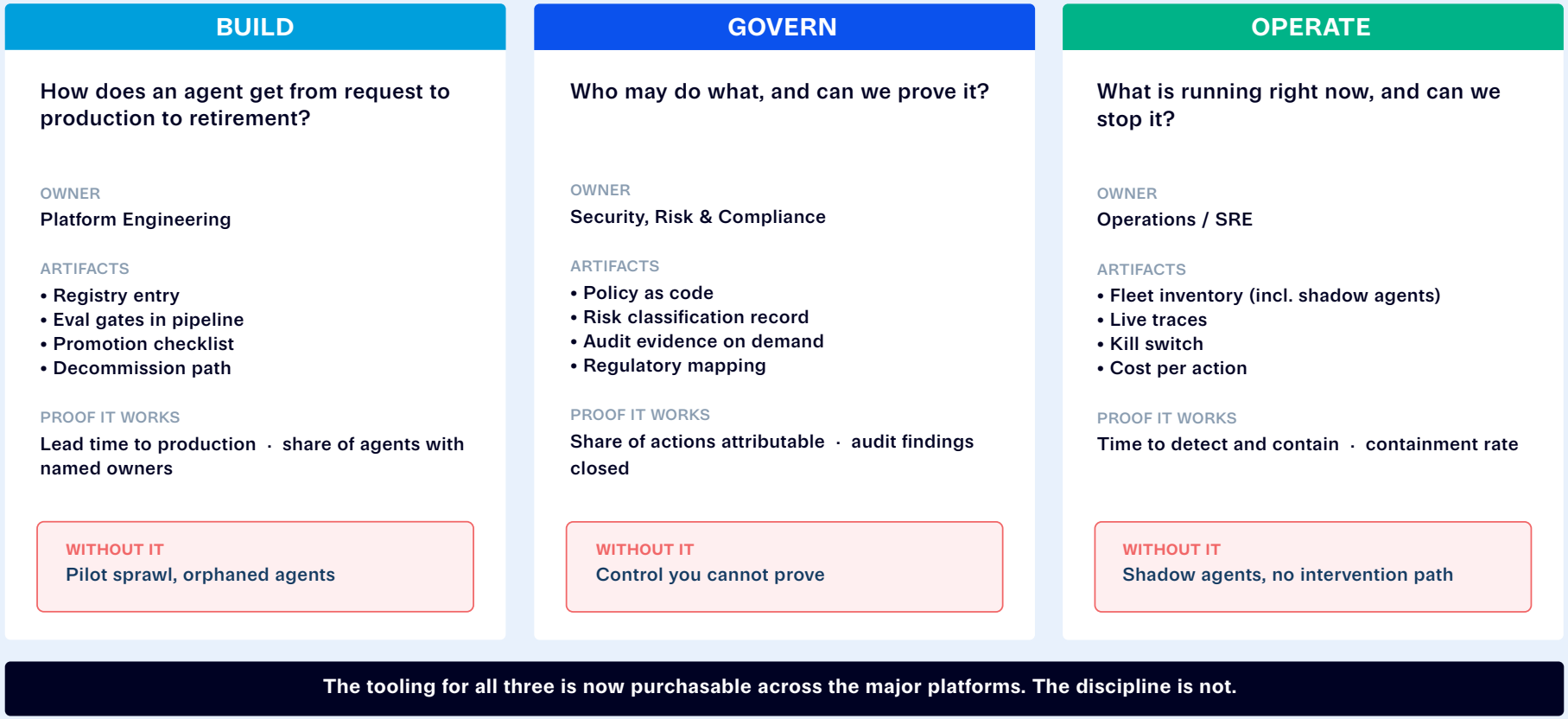


Figure 10. Build, govern, operate. The three jobs of the operating model: three questions, three owners, one fleet.

BUILD:

From Request to Retirement

Every agent in the enterprise gets a managed lifecycle from first request to retirement. Each agent receives a registered identity, a named owner, evaluation gates before promotion, runtime monitoring, and a controlled path to decommission. Platform engineering owns this job. Its artifacts are the registry entry, the evaluation gates in the deployment pipeline, the promotion checklist, and the decommission path, and its health is measured in lead time to production and the share of agents with named owners. Without it, the fleet grows by pilot sprawl and orphaned agents. The objective is autonomy that stays observable, governed, and reversible at scale.

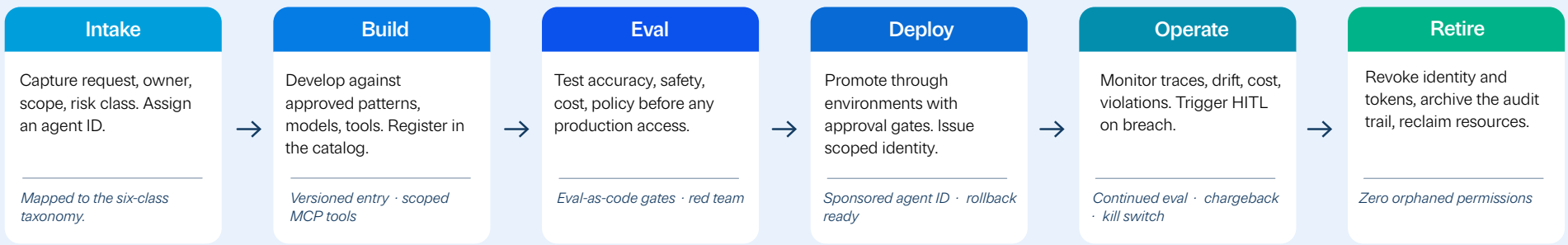


Figure 11. *The AgentOps lifecycle. Every agent is registered, owned, evaluated, monitored, and reversible*

Govern:

Provable Control

Governance defines and enforces the policies, risk controls, and accountability model for every agent, model, dataset, and prompt. It sets who can build, what data agents can reach, how risk is classified, and which regulatory obligations apply. Security, risk, and compliance own this job. Its artifacts are policy as code, the risk classification record, regulatory mapping,[9] and audit evidence on demand, and its health is measured in the share of actions attributable to an identity and in audit findings closed. Without it, the enterprise holds control it cannot prove. The objective is exactly that proof: every action attributable, every policy enforced, every audit answerable. The core capabilities are consistent policy definition and enforcement across build platforms, risk classification and scoring, identity and accountability through sponsored and scoped identities, regulatory alignment, and a continuous audit trail with evidence available on demand.

Three external frameworks anchor the regulatory mapping and serve as reference points for control design.

FRAMEWORK	WHAT IT IS	ROLE IN THE OPERATING MODEL
NIST AI RMF	A voluntary risk-management framework with four functions: govern, map, measure, and manage	This is the backbone most platforms map their controls to.
ISO/IEC 42001	A certifiable AI management system standard	Evidence that governance is a managed, repeatable process.
EU AI Act	Risk-tiered obligations for AI, with transparency duties and human-oversight requirements for higher-risk systems	Drives documentation and oversight. The July 2026 omnibus amendment, Regulation (EU) 2026/1744, settled the timeline: transparency duties and general-purpose AI enforcement took effect in August 2026, and the high-risk obligations of Annex III now apply from December 2, 2027.

Operate:

One Inventory, One Kill Switch

Management operates the control plane that discovers, registers, monitors, and manages the full agent fleet, wherever agents were built or run. Operations owns this job. Its artifacts are the fleet inventory, live traces, the kill switch, and cost per action, and its health is measured in time to detect and contain and in containment rate. Without it, shadow agents accumulate and there is no intervention path. The job provides one inventory, runtime observability, performance measurement, and the ability to intervene or shut an agent down in real time.

The core capabilities are:

- Discovery and inventory that finds every agent, including shadow agents
- Performance measurement against metrics such as containment rate, cost per action, and mean time to resolution
- One controlled registration flow
- Real-time intervention through a kill switch
- Runtime observability into reasoning paths and drift
- Cross-platform reach across the major clouds and custom runtimes

Good operations looks like one inventory where every agent is visible, owned, and risk-tagged, with live observability traced per agent in production, and real-time control that stops an off-script or compromised agent without waiting on a manual process.

Reference Architectures

The same control plane works across every platform. It has five layers, plus a band of cross-cutting discipline. The single most important property is that every agent and every tool call passes through one governed gateway. That gateway is where token and cost caps, content filtering, prompt-injection defense, tool scoping, and audit logging are enforced. Without a single chokepoint, governance becomes a per-application negotiation, and that is where control is lost.

The **Experience** layer is where human and machine consumers reach the agents, including chat surfaces, internal portals, IDE tooling, external agents arriving over MCP, and custom applications, fronted by an edge tier for load balancing, web application firewalling, and TLS.

The **Gateway** is the chokepoint described above.

The **Orchestration** layer holds the managed agent runtime, the agent framework or SDK with MCP and A2A interoperability, and durable orchestration for long-running work.

The fourth layer carries the **Model** catalog, the Tool layer of MCP servers, and the Memory and retrieval store, each exposed once and reused.

The **Data and Identity Foundation** underpins everything, with human identity, non-human identity for agents through sponsored and scoped tokens, data classification and governance, and the data products and vector stores that ground retrieval.

Around all five layers runs the cross-cutting discipline: **Observability, Financial Operations for AI, Governance,** and **Security**. A consistent rule across the foundation is private connectivity to the model and agent services with no public AI endpoints.

The four platform architectures that follow are the same five layers realized in each vendor's services. The distinctive choice of each is worth calling out, but the structure does not change, and neither does the operating model that governs it.

The Unified Agent Control Plane

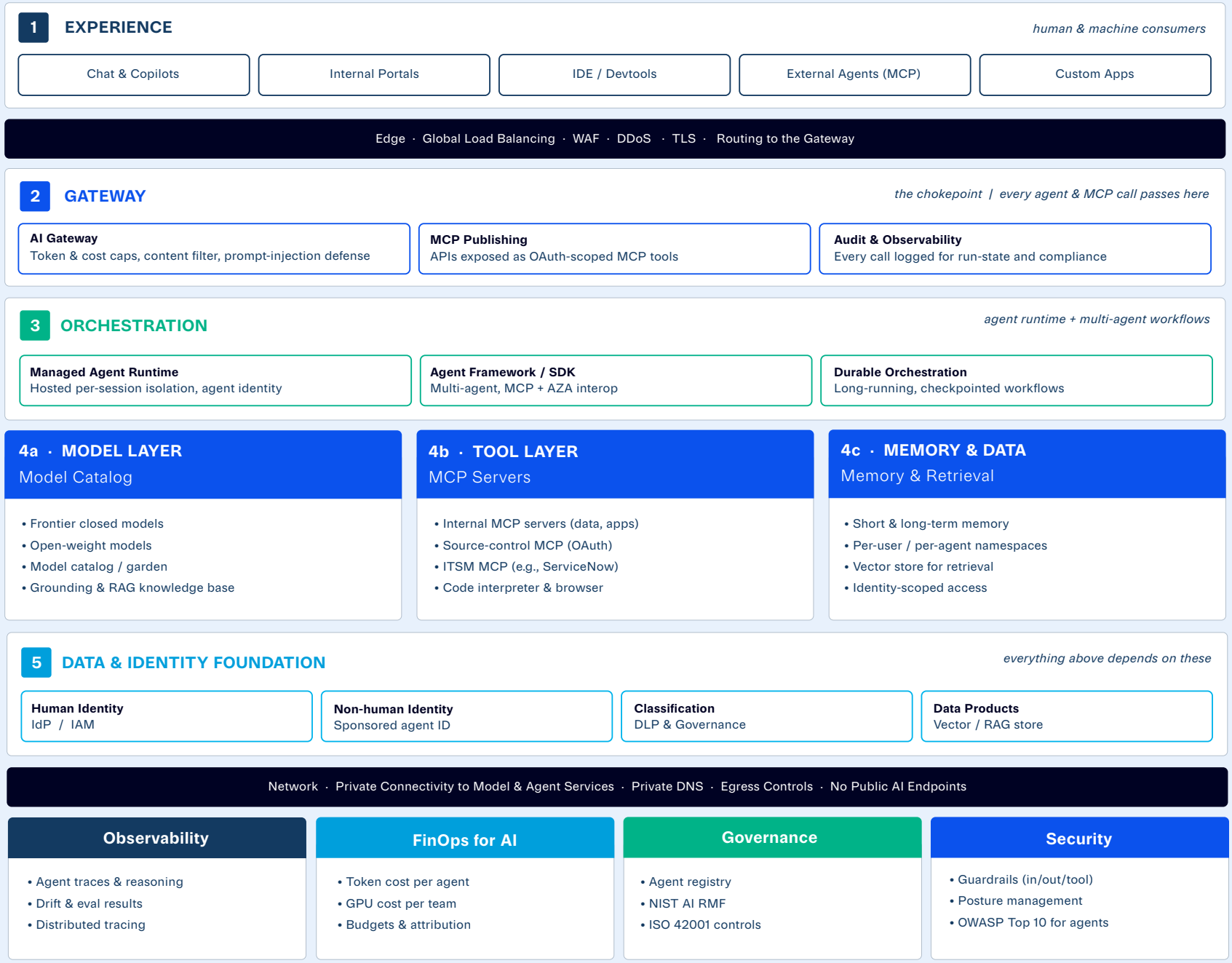


Figure 12. The unified five-layer control plane. One governed gateway for every agent and tool call.



AWS

On AWS, Bedrock AgentCore provides the agent runtime with private networking and, since August 2026, dedicated runtime instances that add GPU capacity and sessions running as long as fourteen days. Its gateway exposes APIs and functions as OAuth-scoped MCP tools and enforces per-identity rate limits, with AgentCore Policy carrying Bedrock Guardrails and, through the open-source Dogwood policy language, rules that govern sequences of actions, budgets, and approval gates rather than single calls.[10] Runtime, gateway, identity, memory, evaluation, and observability are managed, composable services rather than components a team assembles itself. The managed harness (powered by the open-source Strands Agents framework) bundles those pieces behind declarative configuration: a team specifies the model, tools, and instructions while AgentCore provisions the environment underneath, running each session statefully in an isolated sandbox and exporting to Strands code when a team outgrows the declarative path. The model layer is the Bedrock catalog, including Anthropic Claude, the Amazon Nova family, and open models, with the managed knowledge base service for retrieval.

The natural fit is an organization whose core workloads and data already run on AWS, where agent identity and networking integrate directly with the platform's own controls.

AWS Reference Architecture

Bedrock AgentCore runtime and harness, Bedrock model catalog, one governed gateway.

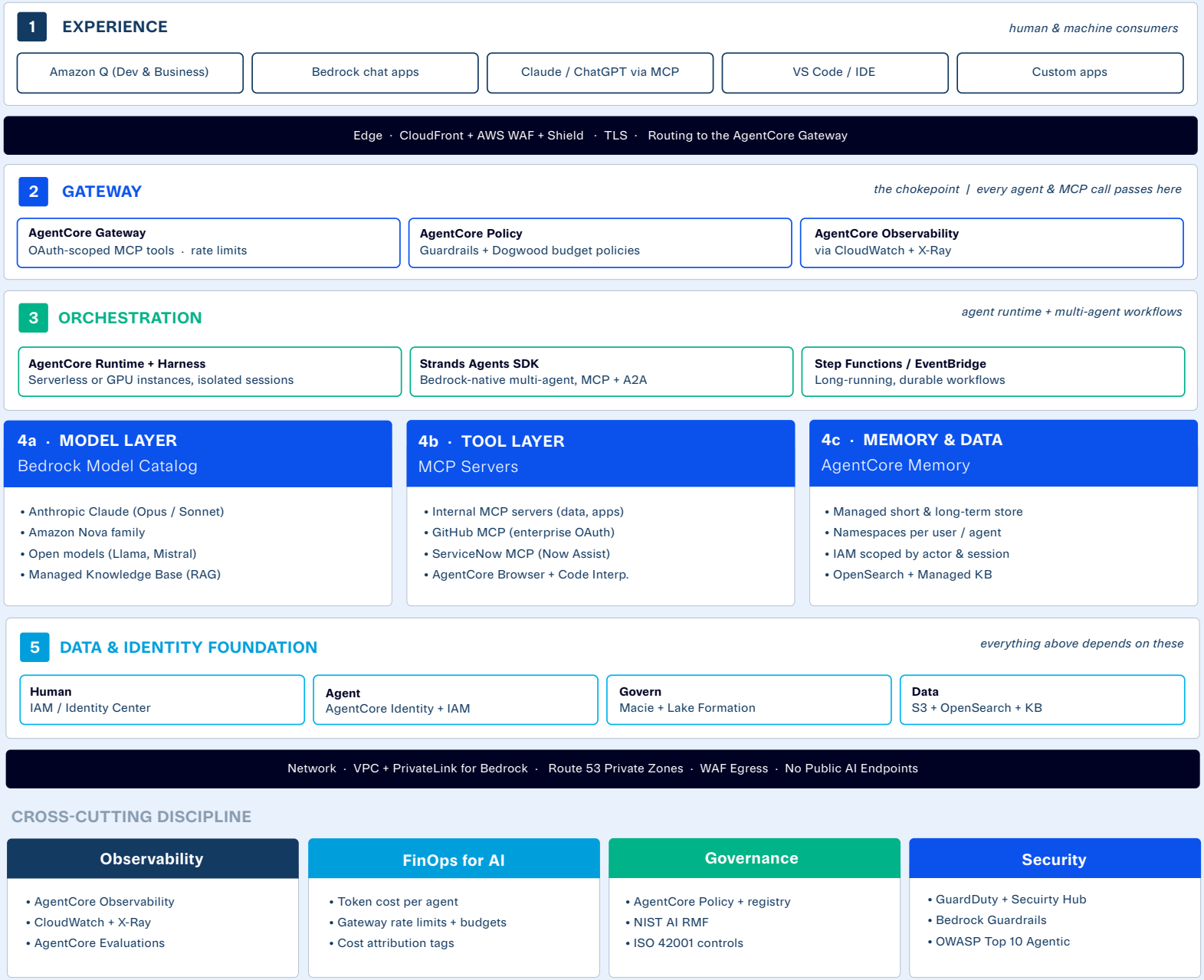


Figure 13. AWS reference architecture, built on Bedrock AgentCore and the Bedrock model catalog.



Microsoft Azure

On Azure, the Foundry Agent Service (part of Microsoft Foundry), provides the runtime. Hosted agents run each session in an isolated virtual machine sandbox, with Microsoft Entra Agent ID carrying non-human identity and extending conditional-access policy to agents.[11] Every call is governed through Azure API Management, whose dedicated AI Gateway tier (in preview) federates models and MCP tool servers behind one policy point. The framework layer is the Microsoft Agent Framework (the convergence of AutoGen and Semantic Kernel, now generally available with its own agent harness). The model layer is the Foundry catalog, spanning Azure OpenAI, Anthropic Claude, Microsoft's MAI family, and open-weight models. Fleet-wide control runs through Agent 365, the agent control plane whose unified registry spans the admin center, Entra, Defender, Purview, and Intune. Data protection runs through Purview, whose agent observability went generally available in May 2026.

The natural fit is an organization where the Microsoft 365 footprint dominates, with users already living in Teams and Office.

Azure / Microsoft Reference Architecture

Microsoft Foundry Agent Service, Entra Agent ID, Agent 365, governed through Azure API Management.

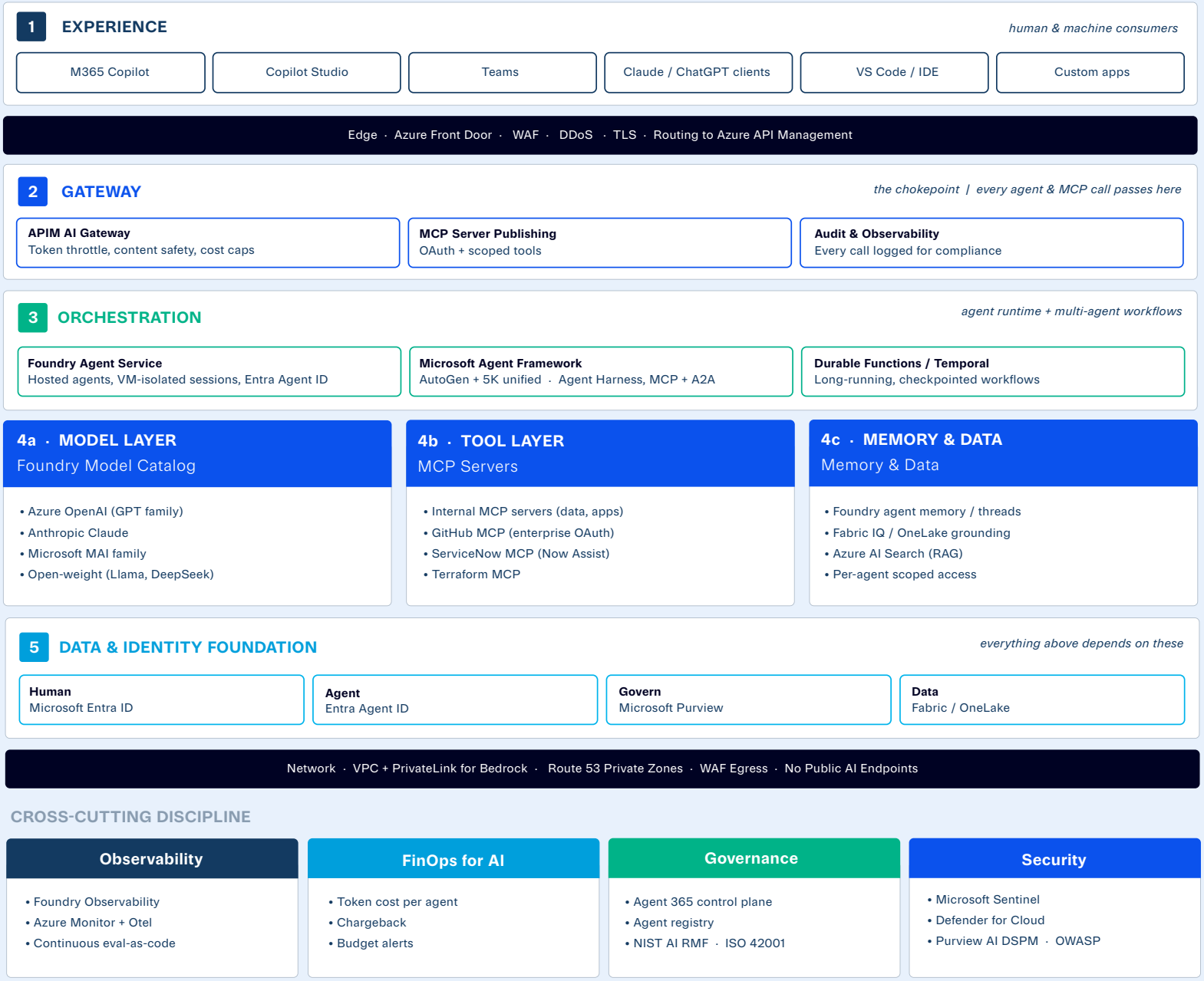


Figure 14. Azure and Microsoft reference architecture, built on Foundry Agent Service and Entra Agent ID.



Google Cloud

On Google Cloud, the Gemini Enterprise Agent Platform serves as the successor to Vertex AI and consolidates Agent Builder, Agent Engine, and Vertex AI Search, providing Agent Runtime and the Agent Development Kit (now in its second major version with a graph-based execution engine).[12] The chokepoint splits into two precise roles: Apigee publishes managed APIs as OAuth-scoped MCP servers, while the dedicated Agent Gateway governs ingress and egress MCP and A2A traffic and enforces Model Armor for prompt and response screening; agent identity is issued on the SPIFFE standard. The platform's governance surface incorporates an Agent Registry that catalogs agents and custom MCP servers, agent observability aligned to OpenTelemetry's emerging conventions, and consumption-based pricing. The model layer centers on Gemini, with Anthropic Claude and open models available, and retrieval is grounded on BigQuery and the platform's data stores.

The natural fit is an organization whose data lives in BigQuery or whose teams have standardized on Gemini, especially for analytics-heavy agents.

Google Cloud Reference Architecture

Gemini Enterprise Agent Platform, ADK, governed through Agent Gateway and Apigee

1 EXPERIENCE

human & machine consumers

- Gemini Enterprise
- Gemini in Workspace
- Conversational Agents
- Claude / ChatGPT clients
- VS Code / IDE
- Custom apps

Edge · Cloud Load Balancing + Cloud Armor · WAF · TLS · Routing to the Agent Gateway

2 GATEWAY

the chokepoint | every agent & MCP call passes here

- **Agent Gateway**
  Ingress + egress MCP & A2A, Model Armor
- **Apigee MCP Publishing**
  Managed APIs as OAuth-scoped MCP servers
- **Audit & Observability**
  Logged via Cloud Logging

3 ORCHESTRATION

agent runtime + multi-agent workflows

- **Agent Runtime**
  Hosted runtime, sessions, SPIFFE agent identity
- **Agent Development Kit (ADK)**
  Multi-language, multi-agent, MCP + A2A
- **Cloud Workflows / Tasks**
  Long-running, durable orchestration

4a · MODEL LAYER

Model Catalog

- Google Gemini 3 family
- Anthropic Claude
- Open (Llama, Gemma, Mistral)
- Data-store grounding (RAG)

4b · TOOL LAYER

MCP Servers

- Internal MCP servers (data, apps)
- GitHub MCP (enterprise OAuth)
- ServiceNow MCP (Now Assist)
- Apigee API hub as MCP tools

4c · MEMORY & DATA

Memory & Data

- Vertex Memory Bank
- BigQuery grounding
- Platform data stores (RAG)
- Per-agent scoped access

5 DATA & IDENTITY FOUNDATION

everything above depends on these

- **Human**
  Google Cloud IAM
- **Agent**
  SPIFFE Agent Identity
- **Govern**
  Sensitive Data Protection
- **Data**
  BigQuery + data stores

Network · VPC Service Controls · Private Service Connect · Private DNS · Cloud NGFW Egress · No Public AI Endpoints

CROSS-CUTTING DISCIPLINE

Observability

- Cloud Trace + Logging
- Agent Observability (OTel)
- Agent evaluations

FinOps for AI

- Token cost per agent
- TPU/GPU cost per team
- Budgets + attribution

Governance

- Agent Registry
- Gemini Enterprise plane
- NIST AI RMF · ISO 42001

Security

- Security Command Center
- Model Armor
- Sensitive Data Protection

Figure 15. Google Cloud reference architecture, built on the Gemini Enterprise Agent Platform (the successor to Vertex AI).



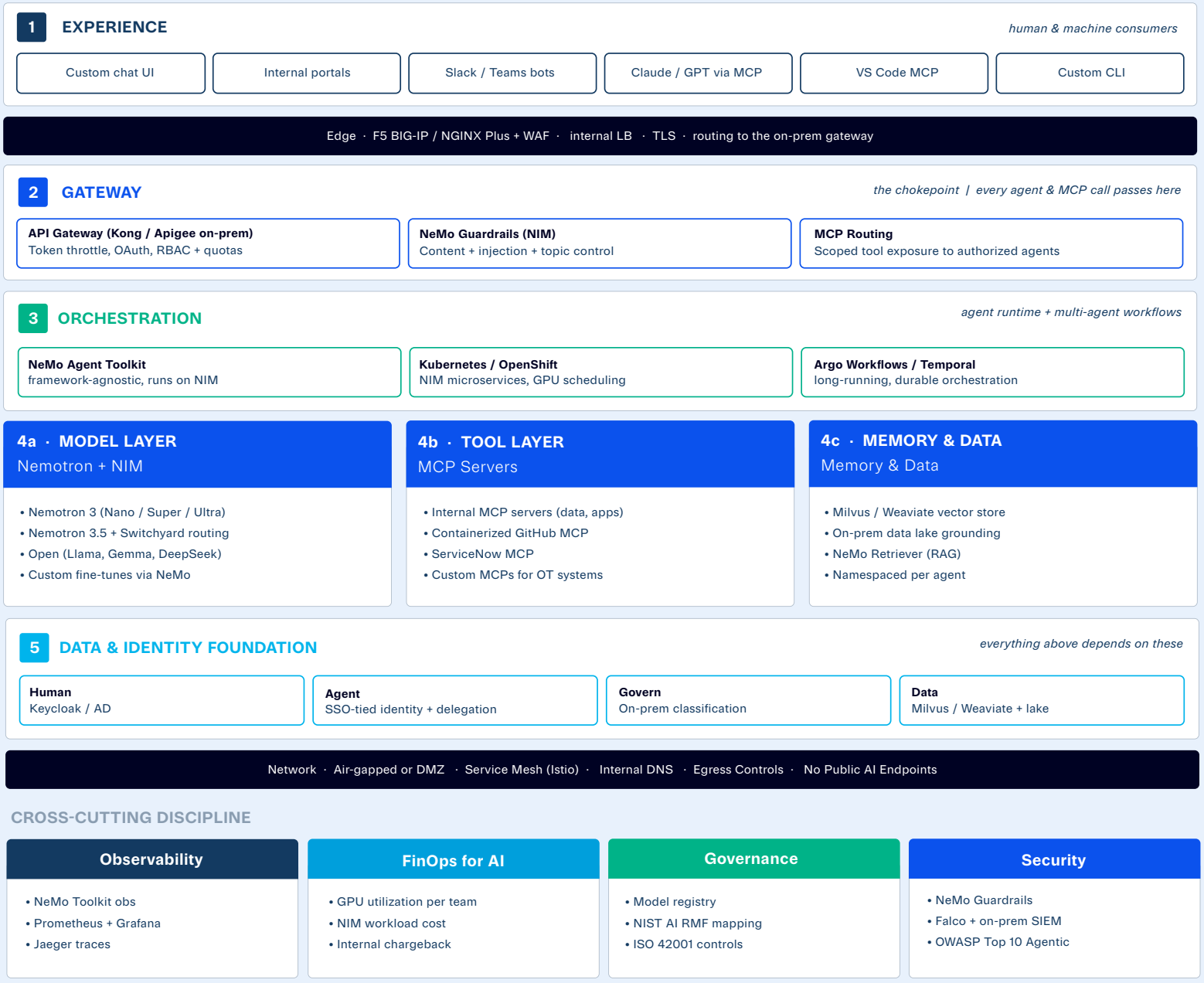
NVIDIA On-premises

On-premises, NVIDIA NeMo and NIM microservices run the model and agent layers on Kubernetes or OpenShift (following the five-tier Enterprise AI Factory validated design) with NeMo Guardrails enforced at an on-prem gateway. NVIDIA's current governance guidance ties each agent's identity to a human sponsor through single sign-on with delegation records, credential proxies, and gateway-level role controls and quotas – a richer model than plain service accounts. NeMo is positioned as an agent-first lifecycle suite, paired with the NeMo Agent Toolkit for building the harness. Models center on the Nemotron family: the third-generation Nano, Super, and Ultra sizes, joined in August 2026 by the smaller Lightning variant and the open-source Switchyard router that sends each task to the cheapest model able to carry it.[13] Retrieval is served by an on-prem vector store.

The natural fit is a workload where sovereignty, air-gap requirements, or GPU economics push it into the datacenter, where the same five-layer control plane runs entirely within the enterprise perimeter.

NVIDIA On-prem Reference Architecture

Sovereign and air-gapped: NeMo and NIM on Kubernetes / OpenShift.



Four platforms, one architecture, one operating model. The services differ. The five layers, the single governed gateway, the non-human identity model, and the cross-cutting discipline do not. That is what makes it possible to govern a multi-platform estate without a separate governance regime for each cloud.

Figure 16. NVIDIA on-premises reference architecture, built on NeMo and NIM on Kubernetes or OpenShift.

A Different Pattern: Building Directly on the Model Vendor

The four platforms above are mutually exclusive cloud choices that all implement the same five-layer control plane. A fifth path exists alongside them, and it is not a fifth cloud. Rather, it involves building directly on a model vendor's own managed agent runtime, bypassing a hyperscaler's agent platform entirely. Claude Managed Agents (Anthropic's hosted runtime for Claude agents) is the clearest example available today,[14] and is worth treating separately because it fills the control plane in a genuinely different shape as opposed to cloning it.

The shape is different in three ways:

1. **Model Layer:** The other four paths front a multi-model catalog, with Claude available as one option inside each. This pattern is single-vendor by design, trading catalog breadth for a harness tuned specifically around Claude's behavior.
2. **Gateway:** None of the other four ship without one, but Claude Managed Agents has no native AI gateway. Anthropic's safety filters operate inside the harness itself rather than as a separate policy chokepoint. While session budgets now cap spend per agent natively, a team wanting content filtering or fleet-wide policy still pairs it with its own metering layer, such as the gateway pattern described below in 'FinOps for Agents.'
3. **Identity:** AgentCore Identity, Entra Agent ID, and Google's SPIFFE-based Agent Identity are each a vendor-native non-human identity system. Claude Managed Agents ships permission policies, tool-confirmation events, and credential vaults instead, and a team wires those into whatever identity provider it already runs, be it Entra, Okta, or its own SPIFFE deployment.

The architecture that results is described internally as "decoupling the brain from the hands." Reasoning runs on Anthropic's infrastructure as a managed, sandboxed session with built-in checkpointing, memory, and observability. Execution (the tool calls and code that a session actually runs) can instead live inside the customer's own VPC, on-premises, or in a partner sandbox, reached through MCP tunnels rather than a shared execution environment (the tunnels themselves are still a research preview). A team can therefore keep code, files, and network egress inside its own perimeter even though the model call itself leaves it.

Building Directly on Claude Managed Agents

A different reference pattern: the brain runs on Anthropic's infrastructure; the gateway and identity are yours to bring.

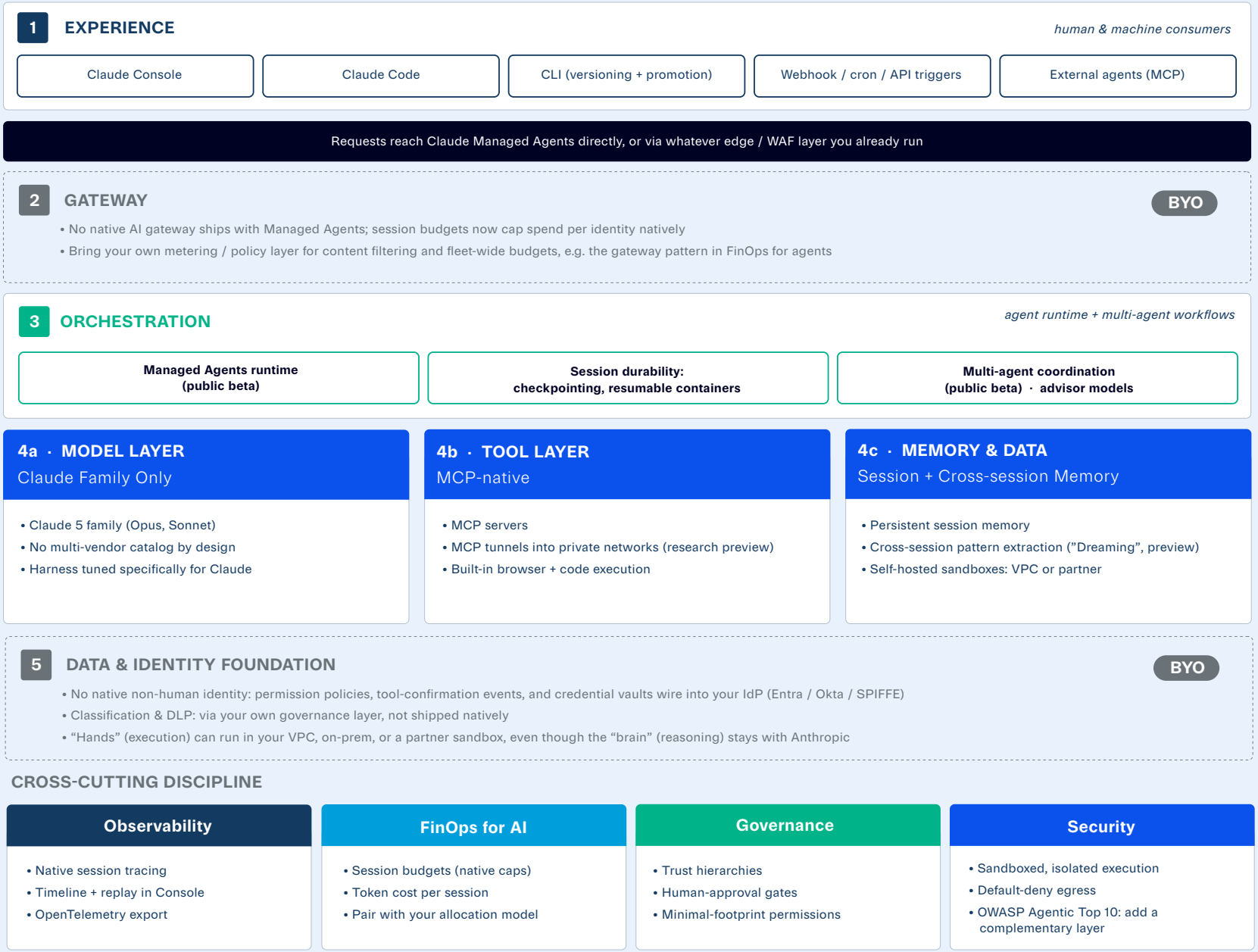


Figure 17. Building directly on Claude Managed Agents. The runtime, tool layer, and observability are native; the gateway and identity foundation are yours to bring.

This pattern fits a narrower set of use cases than the four platforms above, and the fit is narrower by design rather than by immaturity alone. It suits Class 1 through 4 work concentrated on Claude specifically, where the model-harness co-tuning is worth more than model choice, and with teams that want the execution layer decoupled from any single cloud. Multi-agent coordination moved to public beta in May 2026, so the sharper caveats today are different:

1. The platform itself is still in beta.
2. It is not yet eligible for zero-data-retention or healthcare-compliance commitments.
3. An estate that needs one non-human-identity system governing agents across several model vendors at once is still better served by the four platforms, which are built to solve exactly that problem.

OpenAI now offers the same pattern in a different shape. What an enterprise can buy self-serve is (1) the **Responses API,** with MCP support and multi-agent orchestration built in, and (2) the open-source **Agents SDK,** with the gateway, identity, and observability brought by the customer. The single instructive contrast is that Anthropic hosts the runtime while OpenAI's self-serve path is bring-your-own runtime.[15]

OpenAI Frontier, the managed agent platform with a business-context layer, memory, evaluation loops, and management of third-party agents, is the emerging sixth platform. It is deployed at named Fortune 500 accounts and remains limited-availability, delivered by forward-deployed engineering with pricing and architecture documentation too thin to draw a reference architecture honestly.

One production consideration deserves candor. In 2026 alone, OpenAI deprecated its visual Agent Builder, its hosted evaluations product, the Assistants API, and the consumer agent mode. The stable core is the Responses API, the Agents SDK, and MCP; the higher-level surfaces have churned. The lesson here is not to avoid the vendor, but that it is an *operating model*—not any vendor's roadmap—that absorbs churn, which is this paper's argument applied to the fastest-moving corner of the market.

In practice, the two are not an either-or choice. Enterprises are increasingly running a hyperscaler's agent platform for the fleet at-large and a direct model-vendor pattern for the Claude-specific or GPT-specific slice of it, then facing the fleet-governance problem that follows: a kill switch, an audit trail, and an identity model that need to span both estates rather than stop at the boundary of either one. That is the same conclusion the rest of this paper argues from a different direction, so it belongs here as a data point rather than an exception. Platform plurality is the normal case, and the operating model, not the platform, is what has to unify it.

FinOps for Agents: Meter, Allocate & Control at the Gateway

Cost has become one of the most common reasons agent programs stall, alongside unclear value and weak controls. Agents make spend non-deterministic in a way that traditional software does not: a single agent run uses roughly four times the tokens of a chat exchange (a multi-agent run roughly fifteen), and a stuck loop or a recursively spawning subagent can multiply that again before anyone notices. Cloud billing surfaces the damage days later – aggregated, and hard to attribute to the user, application, or agent that caused it.

The discipline that closes this gap is FinOps for agents, and it lives at the same governed gateway that the architecture already requires. The chokepoint that enforces prompt-injection defense and tool scoping is also where every call can be tagged and priced before it is forwarded.

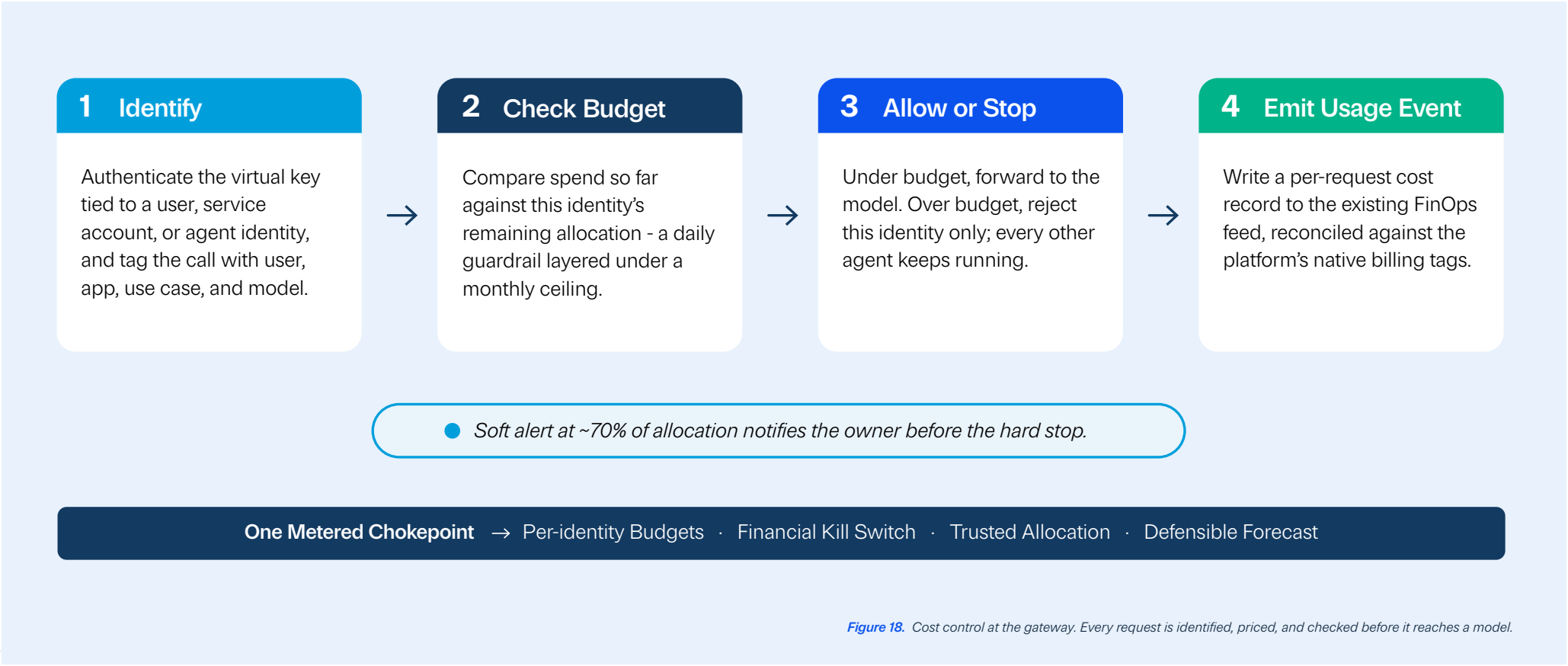


Figure 18. Cost control at the gateway. Every request is identified, priced, and checked before it reaches a model.

Meter at the Gateway

A metering layer:

- Sits in front of the model catalog and, on every request, authenticates a virtual key tied to a user, service account, or agent identity
- Tags the call with user, application, use case, and model
- Checks the identity's remaining budget
- Forwards or rejects
- Emits a usage-and-cost event

The open-source LiteLLM proxy is the most common open-source realization of this pattern, presenting one OpenAI-compatible endpoint in front of Bedrock, Azure, Gemini Enterprise, and others, with virtual keys, per-key and per-tag budgets, spend tracking, and rate limits. The hyperscaler gateways now meter the same chokepoint as managed services: Azure API Management's AI Gateway tier, Bedrock's gateway rate limits, and Apigee each price and police it natively. The July 2026 revision of the Model Context Protocol was designed for exactly this layer, adding routing headers so a gateway can identify, route, and meter MCP traffic without holding session state.[16] The effect is the same, with spend becoming a per-request signal rather than a monthly surprise. Where the platform exposes a native billing tag, such as Bedrock inference-profile tags, run it in parallel as an independent source of truth and reconcile the two.

Budget per Identity & Make the Kill Switch Financial

The non-human identity that scopes an agent's permissions should also carry its budget. Each identity gets a spend cap over a window, a daily guardrail layered under a monthly ceiling with a soft threshold that alerts the owner well before a hard stop, and a hard stop that rejects only that identity while every other agent keeps running. This is the kill switch the taxonomy already demands for higher-risk classes, expressed as a budget rather than a manual intervention, and it is the direct control for the recursive-subagent and stuck-loop failure modes that produce the worst invoices. Allocation is set per-identity because a coding agent and a summarizer have entirely different normal usage, and caps are tuned from observed data rather than guessed once and frozen. The pattern is no longer aspirational: AWS ships per-identity rate limits and sequence-aware budget policies at the gateway, and Anthropic's session budgets cap spend per-agent at the runtime itself.

Agree on the Allocation Model Before the Dashboard

A dashboard is only as trusted as the model beneath it. By deciding how measured usage maps to owners before building reporting, engineering and finance will read the same number the same way. Three drivers cover most cases:

- **Direct usage,** where each team pays for exactly what its identities consumed (the most accurate, and only now possible because the gateway tags every call)
- A **hybrid split,** where shared platform cost is spread evenly and variable token cost is allocated by consumption
- A **weighted driver,** where allocation tracks a true cost driver such as model-tier mix (for cases where a flat split misrepresents load)

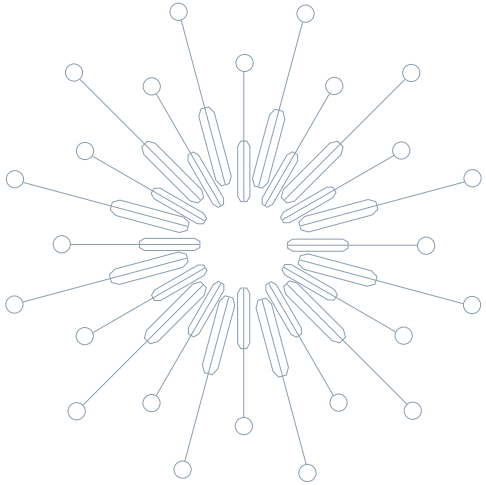
The deliverable is an agreed driver tested against real usage, and the logic stays in the financial systems that the organization already trusts, with AI spend joining the existing cost feed rather than spawning a parallel tool.

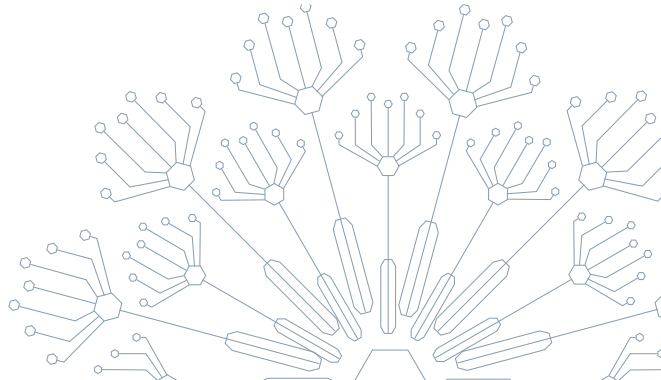
Forecast with a Range

Point estimates are how a pilot that looked cheap becomes a production system that is not. Project from user count, expected engagement (measured as queries per user and tokens per query), and a power-user allowance (modeled separately so a single heavy user is neither averaged away nor ignored), and produce a spend range rather than a single number. Once the gateway has a few weeks of real traffic, the same tool validates the projection against actuals, and the forecast becomes something finance can defend.

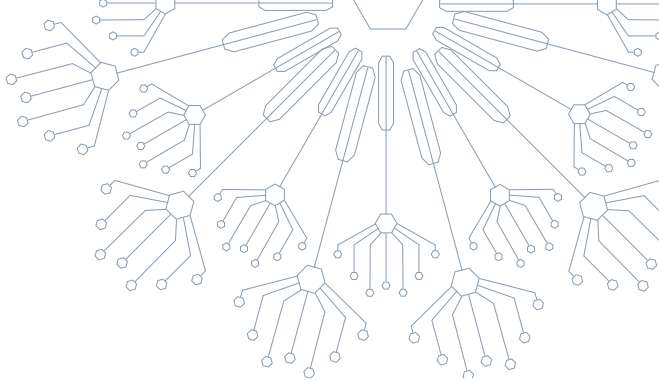
Optimize for Cost per Unit of Output

The metric that matters is cost per successful task, per resolved incident, per accepted pull request, or per answered query, not tokens consumed. That reframing is what lets the architecture's dial-down decisions—such as collapsing a phase to a workflow, demoting a multi-agent run to a single agent, or routing a low-stakes task to a smaller model—to be made based on economics. The gateway is where those levers live, since routing, fallback, and response caching sit at the same layer that meters the spend. Cost per action therefore belongs in the same telemetry set as task success and misroute rate because it is one of the signals that moves the autonomy dial with evidence rather than instinct.





Running Safely: Evaluation, Guardrails & Security



Three disciplines decide whether an agent that works in a demo keeps working (and keeps behaving) in production: **Evaluation**, **Guardrails**, and **Security**. Part IV covered the minimum viable form of the first two, offline regression, judge-based scoring, and guardrails on five surfaces as things to stand up on day one. This section treats all three at the altitude production demands, where the question is no longer whether they exist, but whether they prove value, block the actions that matter, and hold up against an adversary. The paper has already called evaluations and tool-use guardrails the highest-impact controls and the most common gap; this is what they look like done properly, and why security has to be designed rather than filtered.

Evaluation Beyond Accuracy

Accuracy on a golden dataset shows an agent is competent, but it does not show whether the agent is worth running. Every agent should be measured against the baseline it replaces, the manual process, the existing automation, or no action at all, and promoted only if it beats that baseline on the metric that the business actually cares about. The comparison is counter-factual: what would have happened without the agent? For a triage agent, the baseline is the misroute rate the human queue already achieves; for a research agent, the time and quality of an analyst doing it by hand. Without one, an agent that is ninety percent accurate looks like a win even when the process it replaced was at ninety-eight, and the program funds a regression. Value against a baseline—not accuracy in isolation—is the promotion gate.

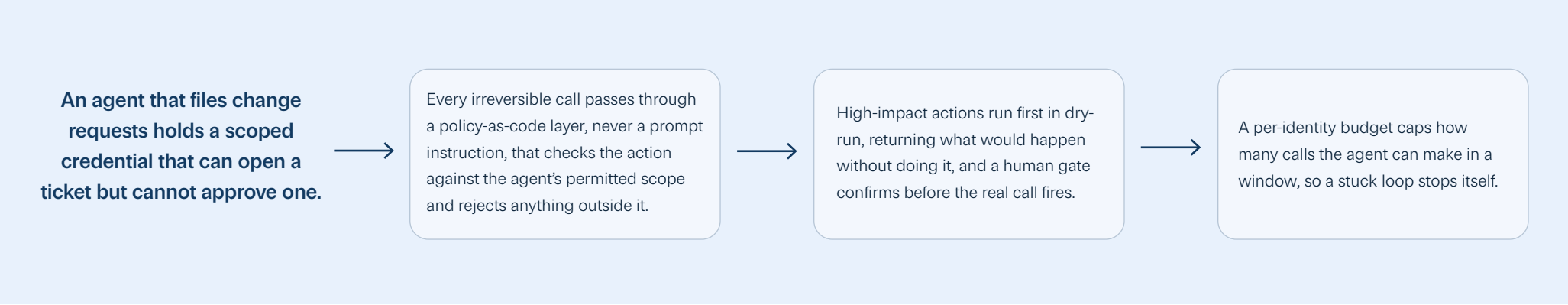
Eval-as-code is what makes that gate repeatable, and it is the control that the Class 5 and Class 6 definitions already assume. The evaluation suite lives in version control beside the agent, runs in the deployment pipeline, and blocks promotion on a failed threshold exactly as a unit-test suite blocks a merge. Each case pairs an input with a deterministic assertion, a reference output, or a judge rubric, and the suite scores the whole trajectory, the tool calls, and intermediate steps. Importantly, not only the final answer is scored, because an agent can reach a correct answer through an unsafe path. Many production tasks have no ground-truth label. An incident hypothesis or a research summary has no single correct string. There, verification shifts from matching an answer to checking a property, every claim carries a citation that resolves, the output validates against a schema, or a second model or a human confirms the reasoning. An agent whose work cannot be verified by anything outside its own confidence has no business owning its loop, which is the rule that the autonomy dial already set.

Two habits keep a suite honest over time. Judge-based scoring drifts because a model grading a model inherits the grader's blind spots, so the judge is calibrated against human labels on a sample, its rubric is kept in writing, and it is re-audited whenever either model changes. Further, an offline suite decays if it is frozen, so online monitoring samples real production traces and every genuine production failure becomes a new offline case. The suite then hardens against the failures the fleet actually produces rather than the ones its authors imagined, which is the feedback loop that separates an evaluation practice from a one-time test pass.

Guardrails on Five Surfaces

Guardrails sit on five surfaces, introduced in Part IV: Input, Output, Dialog, Retrieval, and Execution. In production, the weight is not evenly spread. The first four protect content, screen for injection and sensitive data, check grounding and format, hold topic across a long conversation, and enforce citation accuracy. Execution guardrails protect the world, because they govern what the agent can actually do, and they are what turns a controls policy into something enforceable. Any agent with a write, send, or spend capability is not in production until its Execution guardrails are in place and tested. They are deploy-blockers, not hardening to add later.

A worked example makes the standard concrete:



Scope check, dry-run, human gate, and budget cap are independent layers, and an Execution guardrail that fails open on any one of them is the failure that matters most, because it is the point where a reasoning error becomes an action in a real system.

Guardrails must themselves be evaluated, or they rot silently. A guardrail suite feeds known-bad inputs, injection strings, out-of-scope tool calls, and ungrounded claims, confirming each is caught and running in the same pipeline as the agent's own evaluations, so a prompt change that quietly defeats a guardrail fails the build. The two content surfaces that get the least attention repay it here: Dialog guardrails catch the slow drift where a long conversation is walked off-policy one exchange at a time, and Retrieval guardrails catch the answer that is fluent, but whose cited source does not actually support it. Both are hard to see in a single-turn test and cheap to catch with a check that runs every turn.

The Agent Security Model

The most important agent-specific threat is the one that inverts this paper's own risk ordering. A Class 1 read-only agent is called lowest risk because it cannot write, but prompt injection breaks that assumption. If an agent reads untrusted content, a web page, an email, a document, or a tool result, that content can carry instructions the model follows; and if the same agent holds any outbound channel, a tool call, an API, or even a link to be fetched, it can be turned into a data-exfiltration path. The dangerous condition is three ingredients present at once: access to private data, exposure to untrusted content, and a channel out. A read-only analytics agent with warehouse access that summarizes external documents has all three. The mitigation is architectural rather than a filter:

- Deny an agent that ingests untrusted content any egress it does not strictly need
- Keep private data access and untrusted content processing in separate agents where possible
- Treat read-only as a statement about writes to systems, not about the data an injected instruction can smuggle out

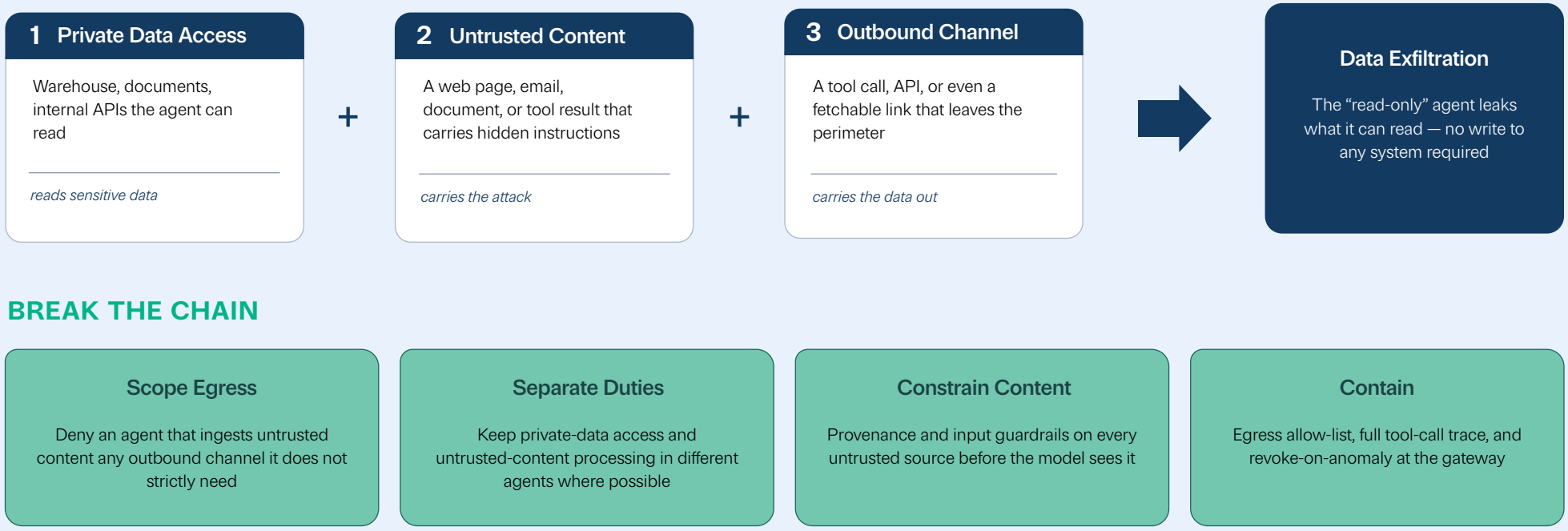


Figure 19. The injection-to-exfiltration path. A read-only agent becomes dangerous when private data access, untrusted content, and an outbound channel meet in one agent; removing any one breaks the chain.

Three further failure modes share that root, because an agent is a deputy acting with delegated authority and the threat is getting it to misuse that authority. The confused-deputy problem is an agent tricked into using its legitimate but over-scoped credential on an attacker's behalf; the defense is least privilege: a credential scoped to exactly the tools and data the task needs and no more, so that a manipulated agent can do less. Tool poisoning is the supply-chain version: a malicious or compromised MCP server can ship a poisoned tool description that reprograms the agent or change its behavior after approval, which is why tool definitions are pinned, reviewed, and sourced from a governed registry (rather than connected ad hoc), and why the SAFE-MCP catalog is worth reading as a checklist rather than a citation. Delegation over Agent2Agent adds a trust dimension to the blast-radius rule from Part II. Calling a remote agent means trusting its authentication, its intent, and its own security posture. Thus, an external agent gets the narrowest possible scope, its signed agent card is verified before any call, its calls are logged and rate-limited like any other tool, and it is never inside the trust boundary by default.

Non-human identity with scoped tokens, threaded throughout this paper, is the foundation that all of this rests on. It is what makes least privilege enforceable and what a responder revokes. But identity and guardrails lower the probability of an incident; they do not remove it, and a production practice needs a plan for the agent that misbehaves anyway. The operating model already holds the containment primitives, the kill switch, rollback, and token revocation, but a primitive is not a process.

Incident response for agents runs the familiar loop with agent-specific content:

- **Detect,** from the drift and policy-violation signals the management plane already watches
- **Contain,** by revoking the agent's identity and tokens, which stops it and every subagent it delegated to at once
- **Investigate,** using the full execution trace the audit trail preserves to reconstruct which tools were called with which arguments
- **Recover,** by reversing what rollback can reverse and manually remediating what it cannot
- **Learn,** by turning the incident into a new evaluation case and, where needed, a tighter guardrail

The trace is what makes all of this possible, which is why full-fidelity tool-call logging is a security control and not only an observability convenience.

The three fit together. Evaluation proves the agent should run; Guardrails bound what it can do while running; the Security model assumes it will eventually be attacked or go wrong and keeps that failure contained and reversible. None of them is a platform feature to switch on. Each is a discipline the operating model has to own, which is why they belong here rather than in any single vendor's console.

PART VI

Where to Start



The first ninety days are about proving the model on real work. Build the platform as the demand for it appears.

Inventory the agents you intend to ship in the next two to three quarters. Mix business and IT operations use cases, and quantify the value of each. This is the list that the operating model is built around.

Stand up the minimum viable stack and the single gateway first. A framework or direct API, tools through MCP, evaluations, guardrails, and observability, with every call passing through one governed chokepoint.

Classify and position each one. Assign the agent class, which sets the controls, and the control-flow position, which sets the architecture. Confirm that the verification mechanism can carry the risk.

Invest disproportionately in evaluations and tool-use guardrails. They are the highest-impact controls and the most common gap. Treat tool-use guardrails as deploy-blockers.

Start with read-only, high-value work. Cross-system investigation and analytics deliver value quickly at Class 1 risk and generate the telemetry that makes every later autonomy decision empirical.

Register every agent in a lifecycle from day one. An owner, an identity, evaluation gates, runtime monitoring, and a retirement path. This is what keeps autonomy reversible as the fleet grows.

Intake Worksheet

One page turns the framework into an intake gate. Before an agent is approved to build, its sponsor answers a short set of questions, and the answers set the controls, the architecture, and its place in the lifecycle.

QUESTION	WHAT IT SETS
What can it do to the world, and can that be reversed?	The agent class, one through six, which fixes the required controls
Who owns the loop – the code or the model?	The control-flow position, P1 through P4, which fixes the architecture
Can the outcome be verified outside the model's own confidence?	Whether the position can carry the class; no external verifier means no owned loop
What identity and scoped credentials will it hold?	The permission set the class is actually bounded to
Does it delegate to other agents, internal or external?	The combined blast radius across the whole constellation
What is the per-identity budget, and who is alerted?	The financial kill switch and its soft and hard thresholds
Who is the accountable owner, and where is it registered?	Its entry into the AgentOps lifecycle, from intake to retirement

A Note on Language

The words a team uses shape the decisions it makes. Useful terms keep the conversation precise: agent class, autonomy dial, verifiability, blast radius, the agentic loop, human-in-the-loop, policy as code, durable execution, harness, and AgentOps. A few phrases are worth retiring. “Level four agent,” “fully autonomous,” and “set and forget” all either misstate the architecture or write checks the guardrails later have to cover. Autonomy is a setting chosen per task, justified by verification and blast radius, and defended by the operating model.

Common Objections

Is graph orchestration outdated now that models can self-direct? No. The explicit control flow encodes the steps an organization is not willing to leave to chance. Greater model capability raises the ceiling on the agent-owned portions of a system; it does not remove the deterministic portions that a regulated process requires. The strongest systems mix both.

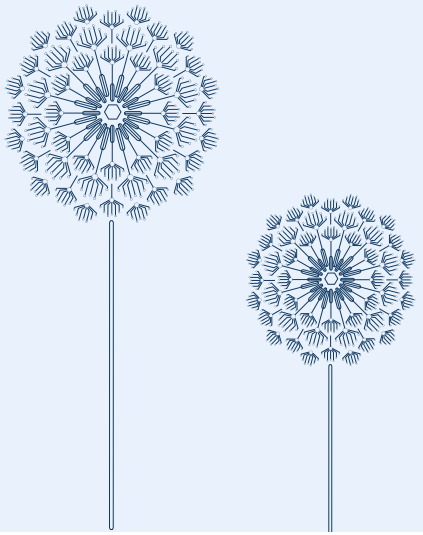
Why not use a general-purpose harness for everything? General harnesses are tuned for tasks with built-in verification, such as code where tests confirm correctness. For business processes without a natural test suite, you supply the verification and guardrails yourself, and explicit control flow is often the cheaper and safer path.

Is an autonomous agent too risky for us? A Class 6 continuously autonomous agent probably is, today, for most enterprises. A P4 owned loop at Class 1 risk (meaning read-only investigation and analytics) is available now, delivers real value, and produces the evidence that makes every later autonomy decision a measured one rather than a leap of faith.

> **Maturity is the telemetry, guardrails, and operating model to classify every agent, set its dial deliberately, and move it in either direction with evidence.**

Final Thoughts

Applied consistently, these practices change the texture of the organization’s work. Agent deployments become sources of operational learning: each one clarifies where judgment belongs, which controls earn their keep, and which patterns can be reused. That compounding knowledge gives teams room to adopt new model capabilities at a measured pace, while people spend more of their attention on exceptions, decisions, and work that benefits from experience.



References

1. Gartner, “Gartner Predicts Over 40% of Agentic AI Projects Will Be Canceled by End of 2027,” June 25, 2025.

2. Gartner, “$234 Billion in Enterprise Application Software Spend Is at Risk from Agentic AI,” July 1, 2026.

3. KPMG Global AI Pulse Survey, March 2026 (32% of enterprises deploying or scaling agents in production; 11% rated mature); Deloitte, “AI agents are scaling faster than governance,” April 2026 (21% report mature agentic governance).

4. The Agentic AI Foundation formed under the Linux Foundation on December 9, 2025, with the Model Context Protocol as a founding project; the Agent2Agent protocol, whose 1.0 specification shipped in March 2026, joined the foundation on August 17, 2026. The Agent Payments Protocol was donated to the FIDO Alliance on April 28, 2026.

5. Amazon Bedrock AgentCore Harness, generally available June 17, 2026; Microsoft Agent Framework Agent Harness, generally available June 2026; Anthropic, Claude Managed Agents documentation, 2026. Component framing informed in part by Rubric Labs, “What Is an Agent Harness,” August 10, 2026.

6. Anthropic engineering, “How we built our multi-agent research system,” June 2025.

7. Microsoft Agent 365, generally available May 1, 2026; AWS gateway rate limits and Dogwood temporal policies, August 6, 2026; Google Agent Registry and Agent Observability, generally available June 2026; Anthropic session budgets, August 7, 2026; Model Context Protocol specification revision of July 28, 2026.

8. OWASP GenAI LLM Top 10, 2026 edition, released August 3, 2026; OWASP Top 10 for Agentic Applications, December 9, 2025; Cloud Security Alliance, MAESTRO threat-modeling framework; SAFE-MCP community catalog.

9. Regulation (EU) 2026/1744, the Digital Omnibus on AI, in force July 27, 2026.

10. Amazon Bedrock AgentCore: general availability October 13, 2025; Harness and Policy general availability June 17, 2026; runtime instances, gateway rate limits, and Dogwood, August 6, 2026.

11. Microsoft Foundry rename announced at Ignite, November 18, 2025; hosted agents generally available July 9, 2026; Entra Agent ID generally available April 2026; API Management AI Gateway tier public preview July 27, 2026.

12. Google Cloud, Gemini Enterprise Agent Platform announcement, April 22, 2026; Apigee MCP publishing generally available March 31, 2026; Model Armor gateway enforcement generally available June 24, 2026.

13. NVIDIA Nemotron 3 Ultra, generally available June 4, 2026; Nemotron 3.5 Lightning and NeMo Switchyard, August 11, 2026; NVIDIA, “How to Govern Autonomous Agents in Enterprise AI Factories,” June 29, 2026.

14. Anthropic, Claude Managed Agents public beta, April 8, 2026; multi-agent orchestration public beta, May 6, 2026; session budgets and data residency controls, August 7, 2026.

15. OpenAI Frontier, February 5, 2026; OpenAI deprecation notices for Agent Builder and hosted evaluations, June 3, 2026; Responses API multi-agent orchestration, August 13, 2026.

16. Model Context Protocol, specification revision 2026-07-28, modelcontextprotocol.io.

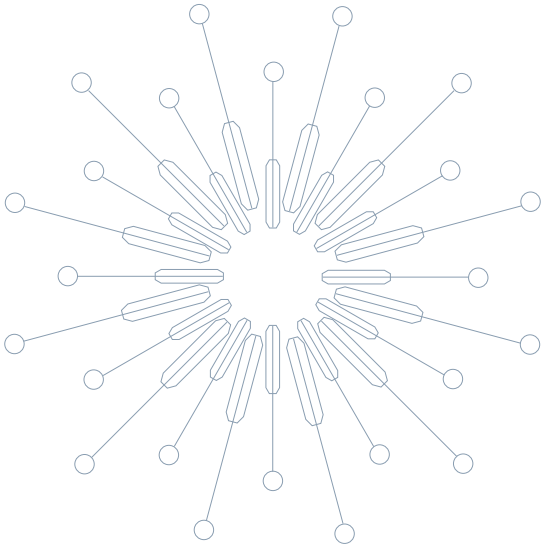
Authors

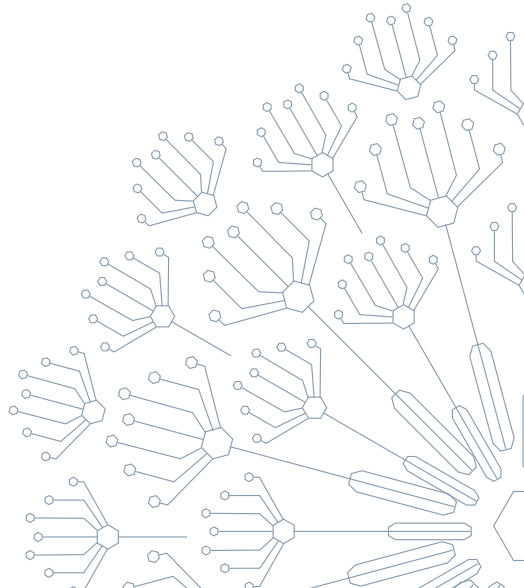
Nick Colyer | VP, AI Solutions

Nick Colyer is VP of AI Solutions at AHEAD, where he leads AI adoption for large enterprises, helping them design and operationalize agentic AI systems. With 15+ years of experience across digital solutions, cloud, platform engineering, and AI, he has helped organizations move AI initiatives from proof of concept into reliable, production-grade operation. His work centers on agentic development, platform reliability, and the operational practices needed to run AI at scale, helping clients turn AI investment into durable business value.

Manik Chhabra | Director, AI Engineering

Manik Chhabra is Director of AI Engineering at AHEAD, where he leads the architecture and delivery of enterprise-scale AI and agentic systems. With experience spanning AI strategy, solution architecture, and engineering leadership, he has helped organizations across life sciences, healthcare, and financial services translate complex business challenges into production-grade AI solutions. His work focuses on building scalable AI platforms, establishing the governance and operational foundations for enterprise adoption, and bridging the gap between AI experimentation and measurable business impact.







AHEAD

Engineering the Future

© 2026 AHEAD, Inc. All rights reserved.

Learn more at
www.ahead.com