

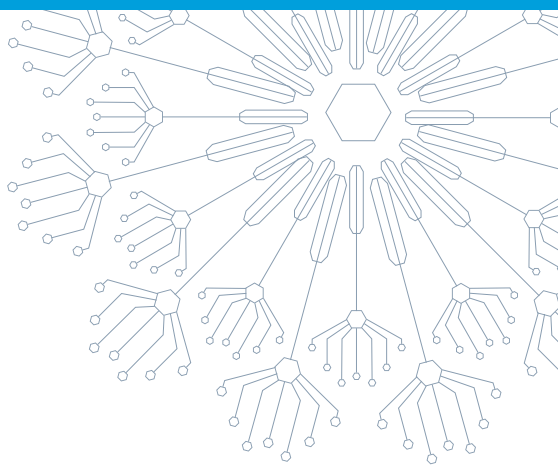
AHEAD

Running Agents in Production

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

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

This paper condenses a full-length AHEAD eBook of the same name, which adds the agentic patterns in depth, twelve worked examples, complete reference architectures for AWS, Microsoft Azure, Google Cloud, and NVIDIA on-premises, and the full operating model for building, governing, and operating an agent fleet.



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 is measurable. Industry analysts expect more than 40% of agentic AI projects to be canceled by the end of 2027, undone by escalating costs, unclear value, or inadequate risk controls. Further, Gartner now puts $234 billion of enterprise software spend at risk of agentic disruption by 2030. Adoption surveys tell the same story from the other side: roughly one third of enterprises report agents deployed in production, while barely 20% have mature governance around them.

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 to a fully autonomous harness. The second is which platform to standardize on. Both feel decisive, yet are mostly a distraction from the decisions that actually govern **risk** and **value**.

Three decisions determine whether an agent reaches production and stays there safely:

1. **Classify the agent by the risk it carries:** the class 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:** lifecycle, governance, runtime management, and the platform engineering beneath.

Platform selection follows from these decisions. It does not lead them. The sections below compress the full framework: the vocabulary of agent types and control, the path from ideas to a prioritized portfolio, the building blocks of agent development, the operating model that runs agents in production, and the first ninety days.

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. Four tests separate a use case that justifies an agent from one that does not: (1) there is a goal to reach, (2) the path varies with context, (3) human involvement is exception-based, and (4) the work spans systems. If every step is known in advance, the work is automation, and automation is the right tool. The most dependable production systems use both, with deterministic automation around the parts that must be guaranteed and agent reasoning where judgment is genuinely required.

Every agent fits one of six classes, and the class functions as a permission statement: what the agent is allowed to do and the oversight that requires. Risk follows blast radius, defined by the cost of a wrong action and whether it can be reversed. A **Suggestion** agent recommends and a person acts; an **Augmentation** agent produces artifacts a person still commits. Only from the **Action** class onward does the agent itself change something in a connected system.

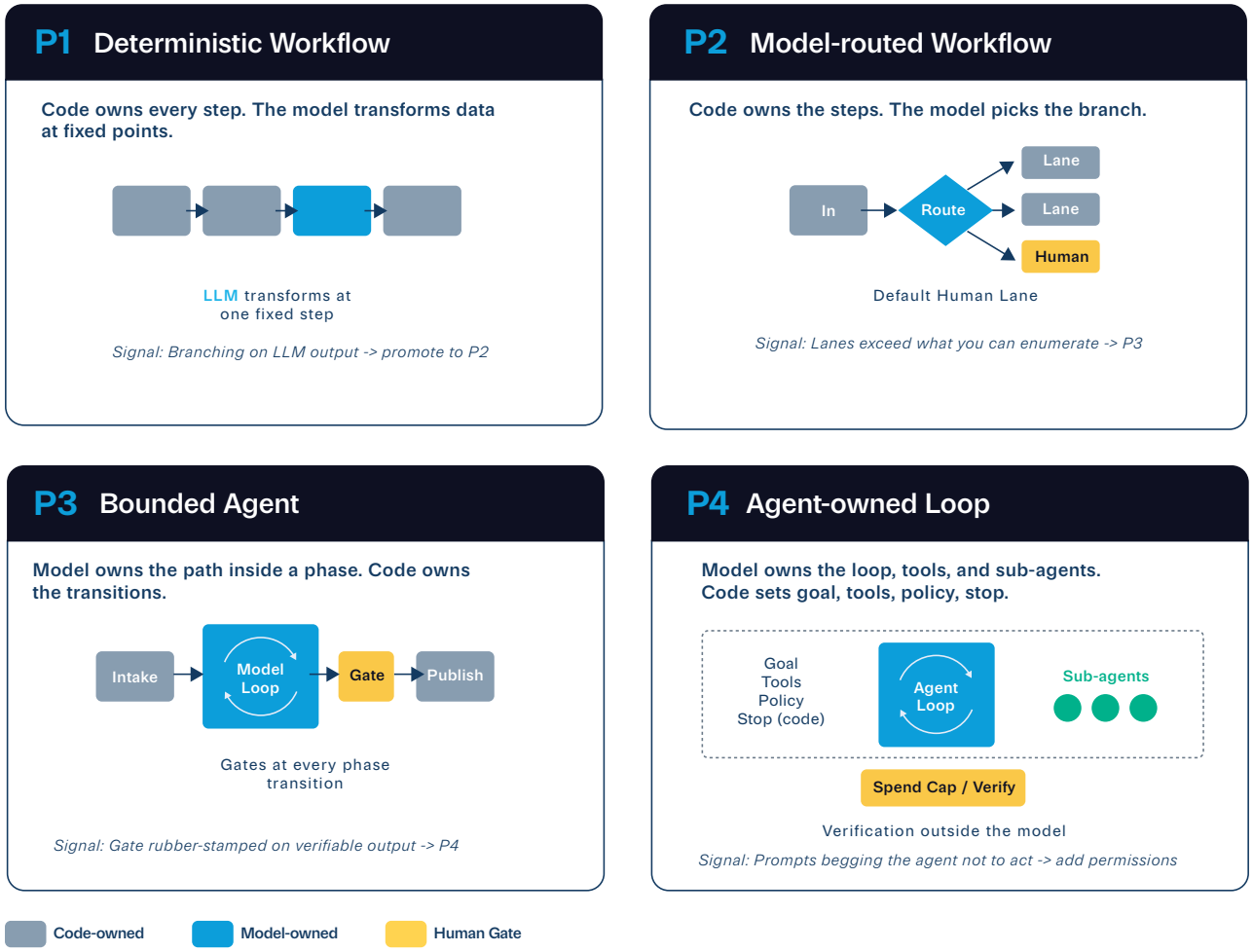
The value of the class system is that it fixes the governance conversation early. A **Query** agent needs audit logs; an **Autonomous** agent needs a kill switch, continuous evaluation, an executive sponsor, and a governance board. One set of controls for all six would over-engineer the easy cases and under-protect the hard ones. The class is decided first and travels with the agent for its whole life, bound to a non-human identity carrying scoped credentials. When an agent delegates to other agents, its class is the union of everything the whole constellation can touch.

SIX AGENT CLASSES & THE RISK EACH CARRIES

RISK	CLASS	REQUIRED CONTROLS	WORKED EXAMPLES
1 LOW	**Query / Assistive** Read-only Q&A over documents or data. No writes.	Audit logs, access scope, labels	• Weekly status roll-up • Cross-system reconciliation • Analytics on demand • Pre-sales research library
2 LOW-MED	**Suggestion** Surfaces a recommended action and a human executes it.	Audit, explainability, human review	• Next-best-action prompts • Renewal-risk flagging
3 MEDIUM	**Augmentation** Co-pilot inside an existing workflow.	Approved models, prompt hardening, output validation	• Invoice narrative generation • Contract metadata extraction • Proposal drafting
4 MED-HIGH	**Action / Tool-using** Performs one bounded action through tools or APIs.	Sponsored agent ID, scoped tools, gateway, approval gate	• Helpdesk ticket triage • Expense exception handling • Security alert triage • In-CI tooling changes
5 HIGH	**Multi-step Agentic** Plans and executes a workflow across systems.	Eval-as-code, per-step gates, rollback, full trace	• Root cause investigation • Codebase maintenance
6 CRITICAL	**Autonomous** Operates continuously without per-action review.	Kill switch, continuous eval, sponsor, governance board	• Self-healing runbook execution • Continuous replenishment within policy

Figure 1. *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) and blast radius (what a wrong action costs and whether it is reversible). High verifiability and low blast radius earn more autonomy. Everything else gets explicit control flow with human gates at the irreversible steps.

Four positions describe the real choices. In a deterministic workflow (P1), code owns every step and the model transforms data at fixed points. In a model-routed workflow (P2), code owns the steps and the model picks the branch. In a bounded agent (P3), the model owns the path inside a phase and code owns the transitions. In an agent-owned loop (P4), the model owns the loop, tools, and sub-agents while code sets the goal, tools, policy, and stop conditions. The dial moves in both directions: a phase that has collapsed to two predictable calls is demoted to a workflow, and a gate that adds nothing on consistently verifiable output is a candidate for promotion.

Class & Position Do Not Collapse into Each Other

Class follows blast radius; 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. A fixed retrieval pipeline answering policy questions and an analytics agent that writes its own read-only queries against the warehouse are both Class 1, because neither can write anything.

> **The most architecturally autonomous agents can be the lowest-risk agents, because they read everything and write nothing.**

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 high-risk use case without a verifier outside the model is not ready for an owned loop, however capable the model is.





PART III

From Ideas to a Prioritized Portfolio

Enterprises rarely lack agent ideas; they lack an agreed way to choose among them. The projects that stall are seldom the ones best matched to the organization's readiness. Instead, they are often the ones chosen by enthusiasm. The first move, before any platform or architecture decision, is to turn the backlog into a ranked portfolio, because 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. 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 decomposed into use cases and scored the same way, and the work proceeds in tight waves so that the organization builds evidence and reusable foundations before it widens the front.

Every candidate is scored on the same weighted criteria: business impact, technical and data readiness, organizational readiness, component reusability, legal and risk, and agentic AI fit. **Impact potential** and **AI readiness** are the two signals that do the most work and become the axes of the prioritization matrix. Plotting each scored candidate sorts the portfolio into waves, and the high-impact, high-readiness quadrant is where the first pilots belong.

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.

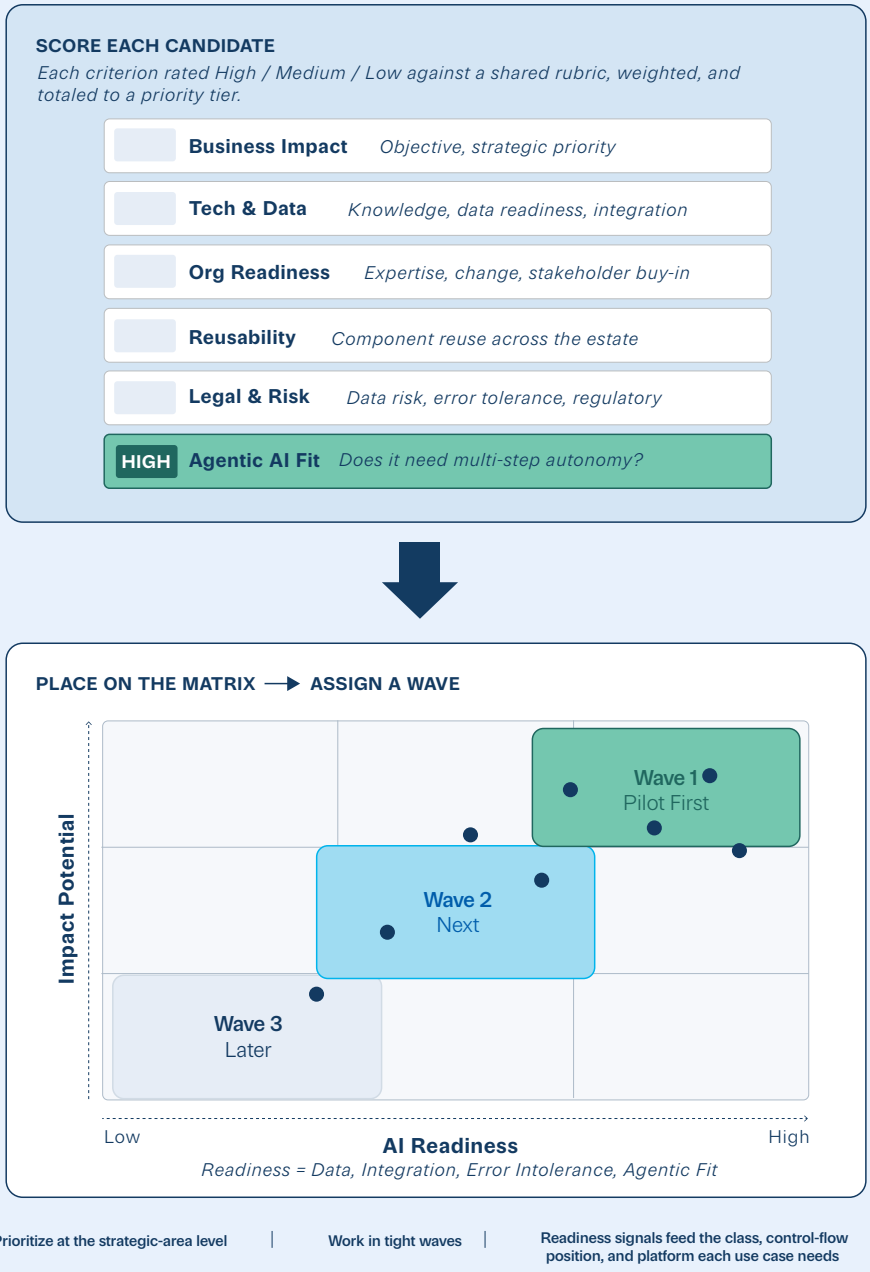


Figure 3. *Prioritizing use cases. Each candidate is scored on weighted criteria, then placed on the impact and AI-readiness matrix to set its wave.*

Prioritization is the front of the same funnel as the architecture. The signals that decide a use case's wave, how clean its data is, how many systems it must touch, and how tolerant its process is of error 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.



PART IV

Building Blocks for Agent Development

Ten Patterns, Three Layers

A small set of patterns underlies almost every enterprise deployment:

- **Prompt Chaining, Routing,** and **Parallelization** structure the work.
- **Orchestrator and Subagent, Evaluator and Critic,** and **Human-in-the-Loop** control quality and risk.
- **ReAct, Plan and Execute, Tool Calling,** and **Memory-augmented Agents** drive execution.

Most production agents compose only two or three of them, and the patterns serve as the shared vocabulary for design reviews.

The components those patterns run on are layers, often mistaken for competitors.

1. A **framework** provides the parts: the loop, tool calling, state, and composition.
2. A **runtime** provides durable execution: checkpointing, retries, and resumability.
3. A **harness** is the complete operating environment around the loop: context management, permissions, and the verification logic that makes the agent trustworthy.

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, so treat the harness as the product and put the differentiation into the tools, policies, and verification specific to the domain.

The Minimum Viable Stack

START WITH THE MINIMUM VIABLE STACK; ADD LAYERS ON TRIGGER

Each added layer brings real operating cost. Stay at the lowest stage that works.

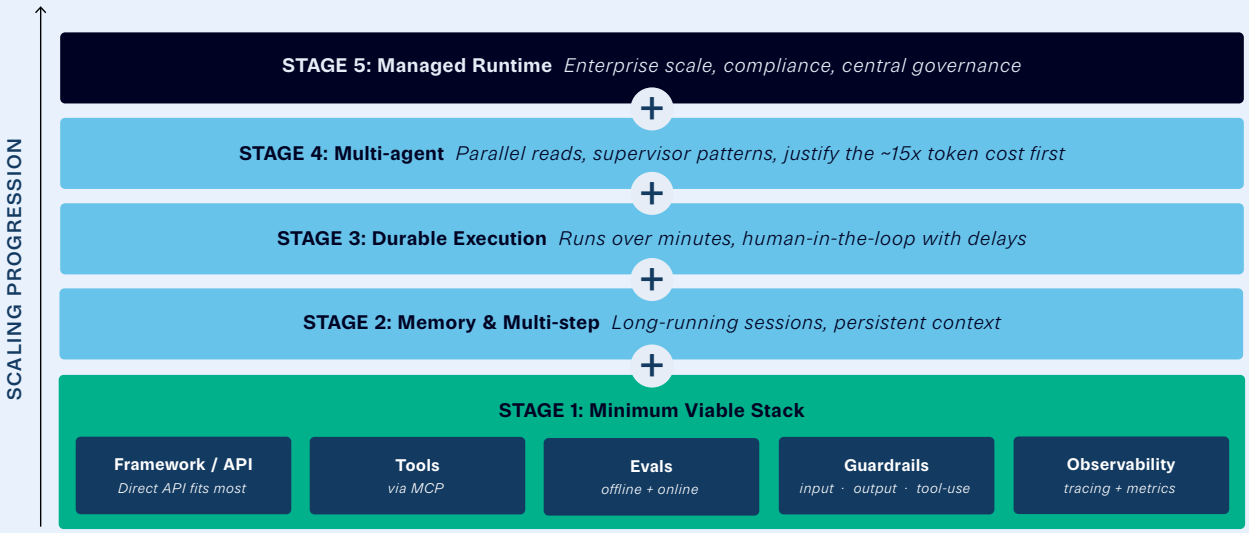


Figure 4. *The minimum viable stack and the staged progression.*

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:

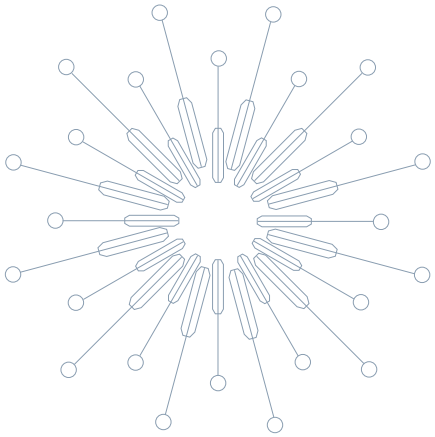
- Memory when sessions run long
- Durable execution when runs last more than minutes or wait on human approval
- Multi-agent structure only when the value clears its cost
- Managed runtime at multi-team scale

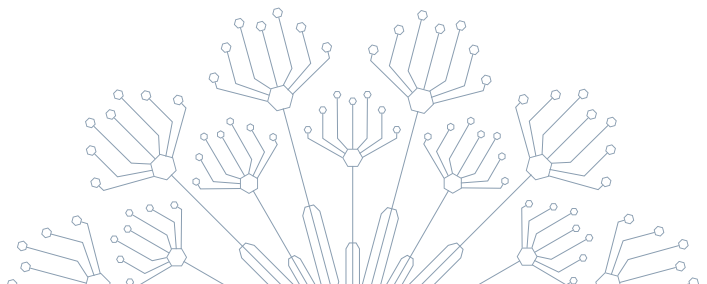
Each added layer carries real operating cost.

Two investments pay for themselves before scale does. The first is **evaluation:** offline regression against a golden dataset on every prompt, tool, or model change, and online monitoring that samples production traces and feeds real failures back into the suite. More agent projects degrade quietly from missing evaluations than from a poor framework choice. The second is **guardrails,** which sit on five surfaces: input, output, dialog, retrieval, and execution. Execution guardrails, the scope limits, approval gates, budget caps, and dry-run modes that govern what an agent can actually do are the most overlooked surface – and the most consequential. Treat them as deploy-blockers.

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
- ✓ Separate reasoning from permissions so that the model proposes and a policy-as-code layer disposes
- ✓ Make every irreversible action a gate
- ✓ Use durable execution for anything long-running
- ✓ Instrument from day one so the telemetry exists to move the autonomy dial with evidence







PART V

Running Agents in Production

The Order of Decisions is Load-bearing

The decisions above sequence in a specific order: **identify the use cases → classify each by pattern and risk → choose the platform where each use case naturally lives → and build the operating model around the agents actually shipping.**

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 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.

The AI-era Operating Model

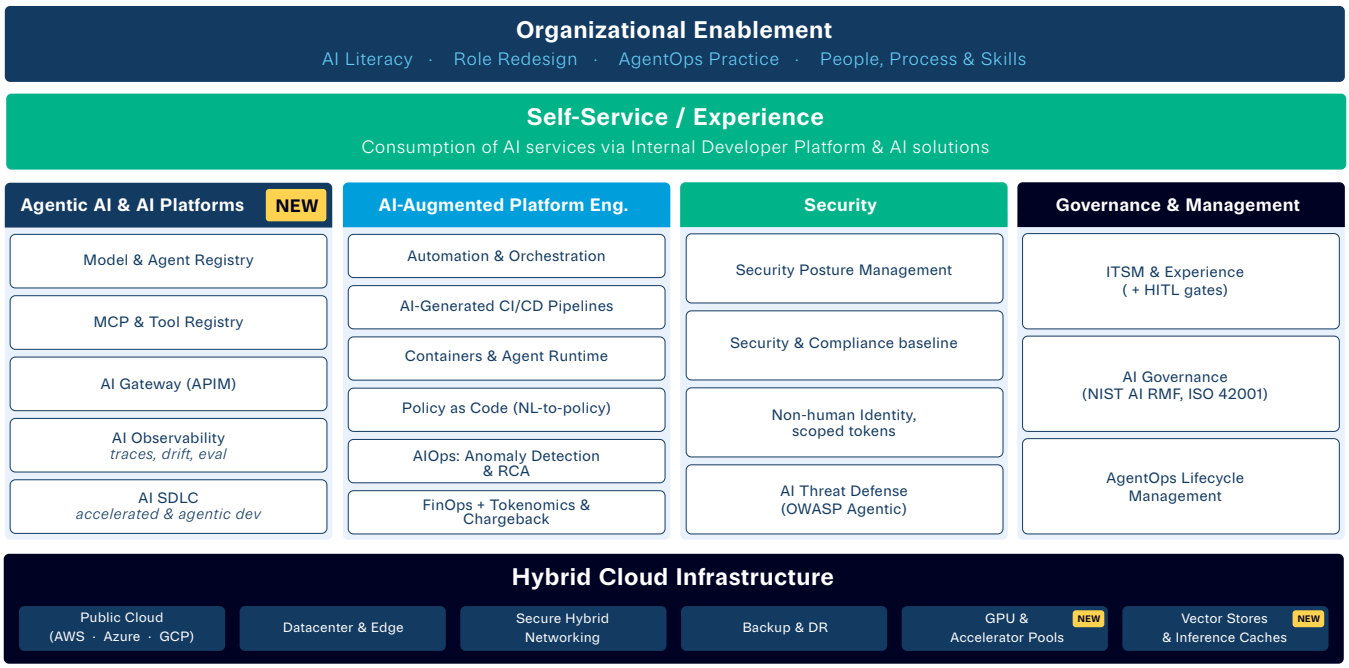


Figure 5. *The AI-era operating model.*

Agents do not require a separate operating model. They need the existing one extended, with a few new pillars.

- **Organizational Enablement** adds AI literacy, role redesign, and an AgentOps practice.
- **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.
- **Hybrid Cloud Infrastructure** adds GPU and accelerator pools, vector stores, and inference caches.
- **Security** adds non-human identity with scoped tokens and defense against prompt injection and the agent-specific threats catalogued in community references.
- **Governance & Management** extends IT service management with human-in-the-loop gates and adds AgentOps lifecycle management.
- **Platform Engineering** extends financial operations with tokenomics and per-agent chargeback.

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 identity, governance, platform, and data.

Build, Govern & Operate

The operating model does three jobs, and each has a named owner.

Build answers how an agent gets from request to production to retirement. Platform engineering runs a managed lifecycle of registered identity, named owner, evaluation gates before promotion, runtime monitoring, and a controlled decommission path.

Govern answers who may do what and whether it can be proven. Security, risk, and compliance define policy as code, classify risk, and keep audit evidence on demand, anchored to NIST AI RMF as the control backbone, ISO/IEC 42001 as certifiable evidence, and the EU AI Act, whose high-risk obligations now phase in through December 2027 under the July 2026 omnibus amendment.

Operate answers what is running right now and whether it can be stopped. Operations owns one inventory that finds every agent, including shadow agents, live observability into reasoning paths and drift, performance measured on containment rate and cost per action, and a kill switch that works in real time. The tooling for all three is now purchasable across the major platforms. The discipline is not.

AGENTOPS LIFECYCLE MANAGEMENT

A managed path for every agent: registered, owned, evaluated, monitored, reversible.

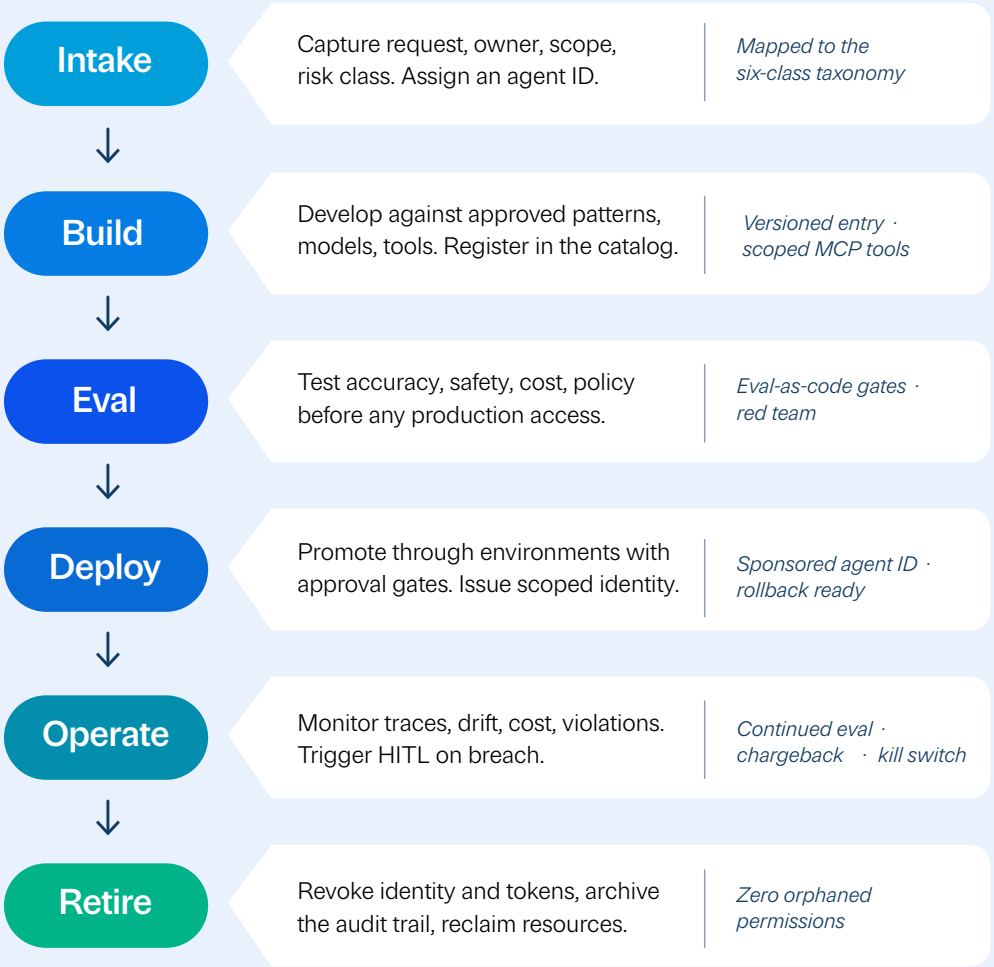


Figure 6. *The AgentOps lifecycle.*

One Architecture, Any Platform

The same control plane works across every platform. Five layers: (1) an experience layer where human and machine consumers reach the agents, (2) a governed gateway, (3) an orchestration layer holding the managed runtime, the framework with MCP and Agent2Agent interoperability, and durable orchestration, (4) a shared layer carrying the model catalog, the MCP tool servers, and memory, and (5) a data and identity foundation beneath. Around all five runs the cross-cutting discipline of observability, financial operations, governance, and security.

The single most important property is that every agent and every tool call passes through one governed gateway. That chokepoint 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 UNIFIED AGENT CONTROL PLANE

Five layers plus cross-cutting discipline. One governed gateway for every agent and tool call.

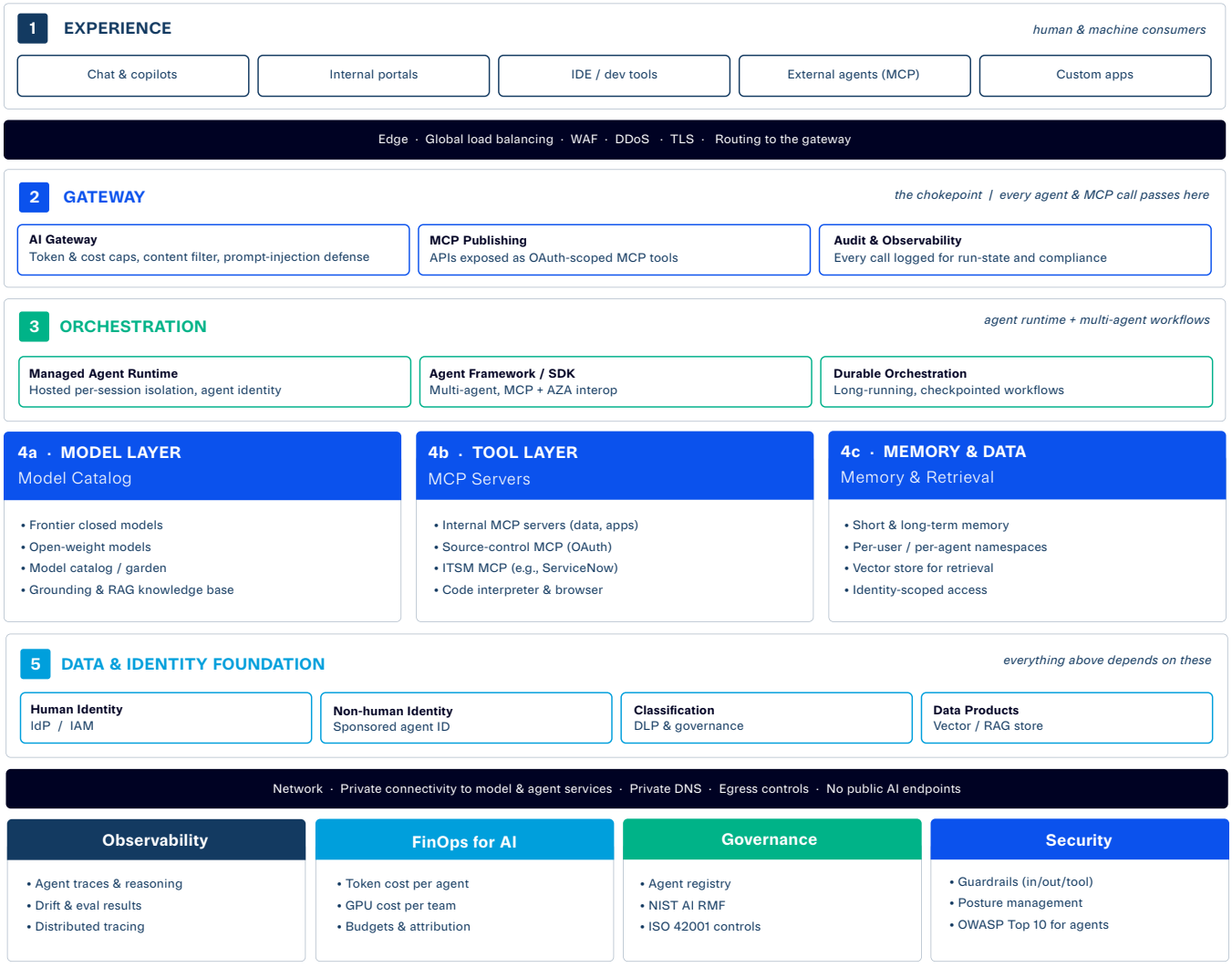


Figure 7. The unified five-layer control plane.

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.

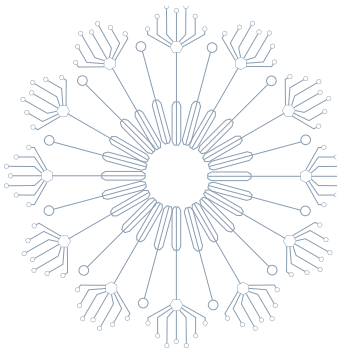
PLATFORM FAMILY	SWEET SPOT
Microsoft	The Microsoft 365 footprint dominates, with users living in Teams and Office. Identity, data-loss prevention, and governance are configured once.
AWS	Core workloads and data run on AWS. Strongest for AWS-anchored data and infrastructure agents.
Google Cloud	Data lives in BigQuery, or Gemini is the model of choice. Strongest for analytics-heavy agents.
ServiceNow	The use case lives inside IT service, operations, customer, or HR workflows. The agent and the system of record sit together.
NVIDIA on-prem	Sovereignty, air-gap, or GPU economics push workloads into the datacenter.

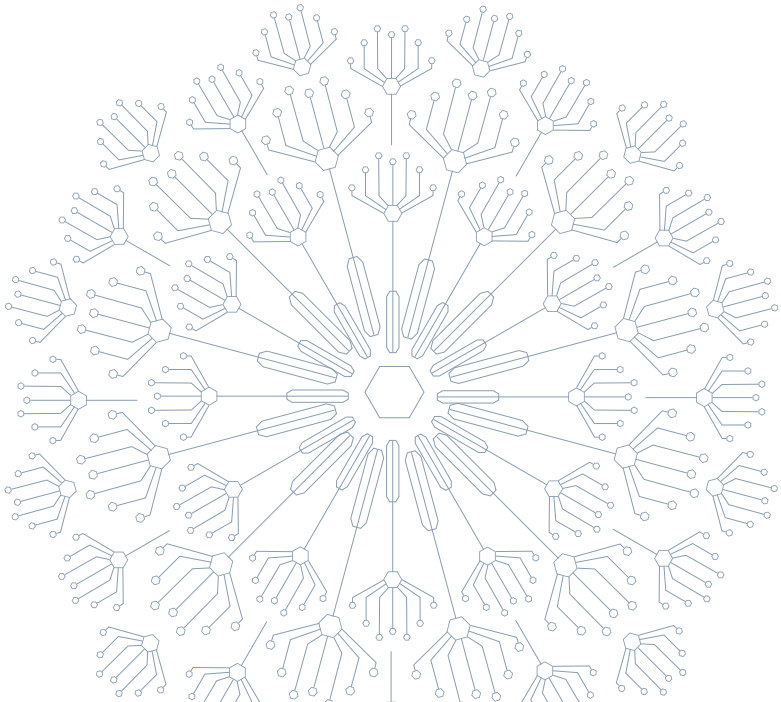
Note: *The full eBook develops a complete reference architecture for each, plus a fifth pattern for building directly on a model vendor's managed runtime.*

Cost, Safety & Security at the Same Chokepoint

Cost is one of the most common reasons that agent programs stall. A single agent run uses roughly four times the tokens of a chat exchange (a multi-agent run roughly fifteen), and cloud billing surfaces the damage days later. The gateway closes that gap. Every request authenticates a virtual key tied to a user, service, or agent identity. It is then (1) tagged with user, application, use case, and model, (2) priced, and (3) checked against budget before it is forwarded. Each identity carries a spend cap with a soft alert under a hard stop, so the kill switch that the higher-risk classes already require is expressed financially, and a stuck loop or a recursively spawning subagent stops itself. The metric to optimize is cost per successful task, not raw tokens consumed.

Safety runs through the same discipline. Every agent is measured against the baseline it replaces (the manual process or the existing automation) and promoted only if it beats that baseline on the metric the business cares about, with eval-as-code blocking promotion in the same way that a failing test suite blocks a merge. One security finding inverts the risk ordering and deserves an executive audience. A read-only agent stops being low-risk when three ingredients meet in one place: access to private data, exposure to untrusted content, and a channel out. Prompt injection can turn that agent into a data-exfiltration path. The mitigation is architectural rather than a filter. Deny egress the agent does not strictly need, separate private-data access from untrusted-content processing, and treat read-only as a statement about writes to systems, not about what an injected instruction can smuggle out.







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.

Classify and position each one. The class sets the controls, the position sets the architecture, and the verification mechanism has to carry the risk.

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

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.

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

Register every agent in a lifecycle from day one. An owner, an identity, evaluation gates, runtime monitoring, and a retirement path.

The Intake Worksheet

One page turns the framework into an intake gate. Before an agent is approved to build, its sponsor answers seven 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

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

Going Deeper

This brief condenses the full-length AHEAD eBook, *Running Agents in Production,* which adds the ten agentic patterns in depth, the anatomy of an agent harness, twelve worked examples mapped across the four control-flow positions, two end-to-end operations scenarios, complete reference architectures for AWS, Microsoft Azure, Google Cloud, and NVIDIA on-premises, FinOps allocation models, and the full evaluation, guardrails, and security discipline.

To request the eBook or a working session on your agent portfolio, contact your AHEAD team.



AHEAD

Combining cloud-native capabilities in software and data engineering with an unparalleled track record of modernizing infrastructure, we're uniquely positioned to help accelerate the promise of digital transformation.

Visit us at ahead.com.

Global Hub Offices

CHICAGO

444 W. Lake Street
Suite 3000
Chicago, IL 60606
United States

NEW YORK

500 5th Avenue
17th Floor
New York, NY 10010
United States

UNITED KINGDOM

310 Wharfedale Rd
Winnersh, Wokingham
RG41 5TP
United Kingdom

NETHERLANDS

Kerkenbos 10-123
6546 BJ Nijmegen
The Netherlands

INDIA

L-21, One Horizon Center
Golf Course Road, DLF Phase V
Sector 43, Gurugram 122002
India