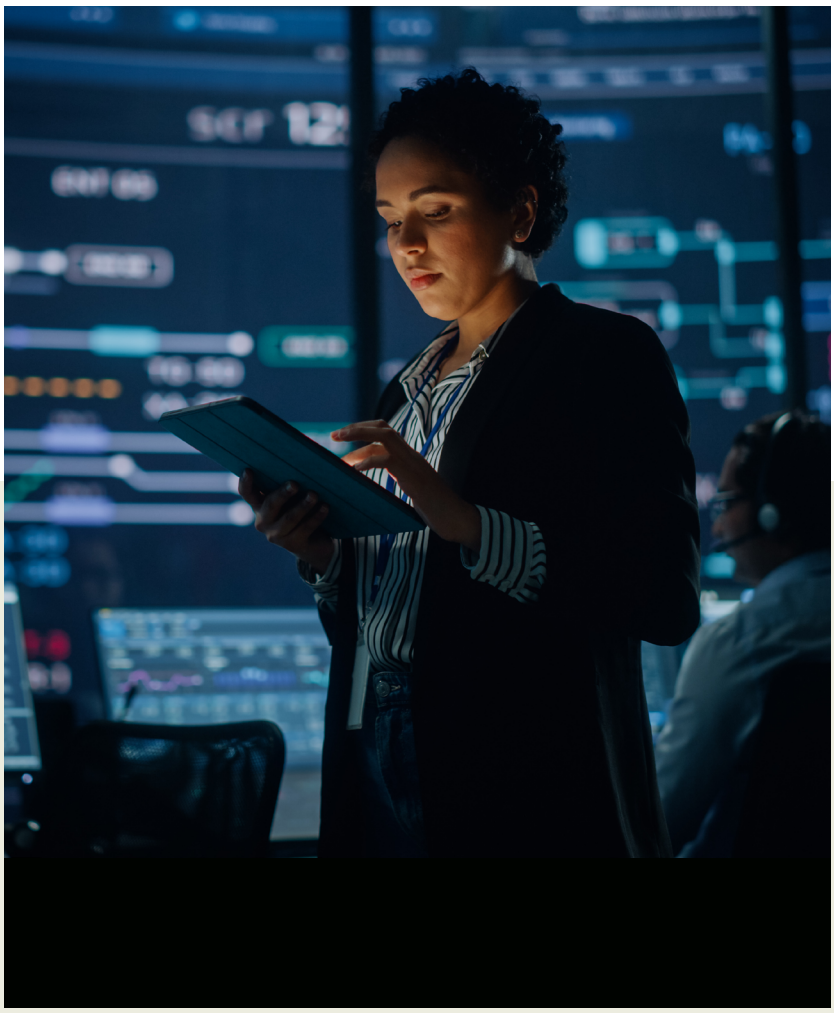
UST + AWS | AI-NATIVE PLATFORM ENGINEERING

An agentic control plane



A practitioner's guide to AI-native platform engineering

WHITEPAPER

Ravi Julapalli
Senior Director
AI-Native Platform Engineering

ust.com

Contents

The case for a control plane

Enterprise AI has moved through three distinct phases. Agents started as assistants: summarizing, drafting, and answering questions. They became builders: generating code, designing tests, and accelerating delivery. Now they are operators. They manage incidents, making reliability decisions, and executing operational responses with minimal human involvement. A well-implemented agentic control plane should allow AI agents to resolve the majority of operational incidents autonomously, while humans manage exceptions.

That shift creates an infrastructure problem most enterprises are not prepared for.

AI agents have important similarities to APIs. They still require versioning, routing, throttling, and controls to limit blast radius. But agents introduce something APIs never had: non-determinism. An API either works or it does not . An agent can degrade gradually, and make many subtly incorrect decisions before anyone notices. Research shows 91% of ML models experience performance degradation without proactive monitoring. Unlike APIs, there is no straightforward automated rollback.

The platforms enterprises built for cloud-native workloads were never designed for this. They assume deterministic behavior. They route on availability, not confidence. They govern at deployment, not runtime. They enforce policies on resources, not context.

> **The gap is structural. It also compounds over time.**

Three failure modes that surface in production

Drift without detection

An agent that performed well in staging loses accuracy in production as conditions change around it.

Three distinct drift types require monitoring:

- Data drift: the statistical properties of inputs change.
- Concept drift: the relationship between inputs and correct outputs shifts.
- Prompt drift: accumulated changes to instructions, system prompts, or retrieved context create inconsistent behavior.

A fourth vector is often overlooked: model providers update their underlying models without notice, silently changing agent behavior in production. The platform has no native mechanism to detect any of these.

Governance that breaks at scale

When agents are making thousands of operational decisions per hour, periodic reviews and blanket policies fail. Two specific threats require dedicated controls. Prompt injection is malicious content embedded in tool outputs, retrieved documents, or upstream agent messages that overrides an agent's instructions and changes its behavior.. This is an active attack vector that agent guardrails must intercept at runtime. Attribute-Based Access Control (ABAC) policy violations: agents with broad data access can expose sensitive attributes in their outputs even when no data leaves the perimeter. IAM controls the connection. It does not control the context.

What this means for platform engineering

AI-Native Platform Engineering is a parallel discipline to cloud-native engineering, with different primitives, different failure modes, and different governance requirements. Organizations that try to retrofit their cloud-native platform will hit the failure modes above. Organizations that build a dedicated control plane for agentic AI will not.

The rest of this guide explains exactly how to build one.

First principles
What a control plane must do

Before designing an agentic control plane, start with first principles rather than vendor feature lists. Five properties are non-negotiable. Everything else is implementation detail.

Property 1: Observe behavior, not nust state

Traditional observability measures state: Is the service up? Is latency within bounds? Is the error rate acceptable? These metrics are necessary but insufficient for agents. An agent can be fully available, low-latency, and error-free while producing slightly wrong outputs.

Agentic observability must track behavior. This includes accuracy over time, confidence score distributions, output consistency across similar inputs, and deviation from baseline. This requires evaluation pipelines running in production, not just in **Continuous Integration and Continuous Delivery** (CI/CD).

> **Core principle: Every agent must be a measured endpoint, not just a monitored one.**

Property 2: Enforce boundaries at the semantic level

Network-level isolation (VPCs, security groups, IAM roles) controls what an agent can reach. It does not control what an agent can do with what it reaches. An agent with read access to a customer database can expose Personally Identifiable Information (PII) in its outputs even if no data leaves the perimeter.

ABAC policies travel with the data, not just the infrastructure. A customer support agent can access customer records but cannot surface PII. A finance-related agent can query financial systems but cannot include account balances in escalation messages.

Agent guardrails are the enforcement mechanism for semantic boundaries. They sit at the model invocation layer and intercept inputs and outputs that violate defined policies. Guardrails are not a substitute for ABAC policy design, but they are the last line of defense before a policy violation reaches a downstream system.

> **Core principle: Governance must follow the context, not just the connection.**

Property 3: Govern at runtime, not just at deployment

CI/CD governance (testing, validation, approval gates) ensures an agent behaves correctly when deployed. It cannot ensure the agent continues to behave correctly as the world changes around it. Runtime governance continuously evaluates agent behavior against defined service level objectives (SLO) and enforces responsible AI policies as an ongoing function, not a one-time gate.

> **Core principle: An agent approved for production is not an agent approved forever.**

Property 4: Route on semantics, not just load

Service meshes and API gateways route on availability and load. They do not route on task complexity, agent confidence, or the trade-off between cost and capability.. A high-confidence, low-complexity incident should route to a fast, cost-efficient model. A low-confidence, high-stakes decision should route to a more capable model and potentially escalate to a human.

> **Core principle: The right agent for the right task at the right cost is an orchestration problem, not a load balancing problem.**

Property 5: Make human escalation a core design element

Human fallback in most systems is an afterthought: an error handler, a last resort. In an agentic control plane, human escalation is a designed outcome. Defined confidence thresholds trigger clear escalation paths to the right human specialists. The system knows when it does not know enough to act. This requires predefined thresholds, clear ownership, and a feedback loop that uses escalation outcomes to improve agent performance over time.

> **Core principle: The decision of when not to act autonomously is as important as the decision of when to act.**

The five properties as a design checklist

Before building any component of your control plane, validate it against these five properties:

- Does it observe agent behavior, not just service state?
- Does it enforce boundaries at the semantic level, not just the network level?
- Does it govern continuously at runtime, not just at deploy time?
- Does it route on task semantics and confidence, not just load?
- Does it treat human escalation as a designed outcome, not an error handler?

> **If any component fails these checks, it is cloud-native infrastructure adapted for agents, and not a true control plane.**

Capability 1
AI Gateways (Agent + Model)

The gateway layer is where the control plane meets the real world. It is the ingress and egress point for every agent interaction. Getting this right is foundational because every other capability depends on having a managed, observable gateway layer.

The two gateways

Most teams build one gateway or none. A mature agentic control plane requires two separate gateways with distinct responsibilities.

Agent gateway (INGRESS)

Manages traffic from users and systems into agents. Responsibilities: authentication, routing, versioning, rate limiting, request logging.

Every interaction passes through the Agent Gateway. No agent should be directly addressable.

Model gateway (EGRESS)

Manages traffic from agents to LLMs. Responsibilities: model selection, fallback routing, cost controls, token budgeting, response validation.

Every model call passes through the Model Gateway. No agent should call a model directly.

What to build

Agent registry

The catalog of all agents in production. Each entry records the agent version, its capability manifest, its SLOs, and its current health status. The Agent Gateway routes to this registry, not to agent endpoints directly.. The capability manifest is the per-agent contract stored within it.

Version control for agents

Agents must be versioned like services. New versions go through canary deployments. The gateway routes a percentage of traffic to the new version and rolls back cleanly if accuracy degrades.

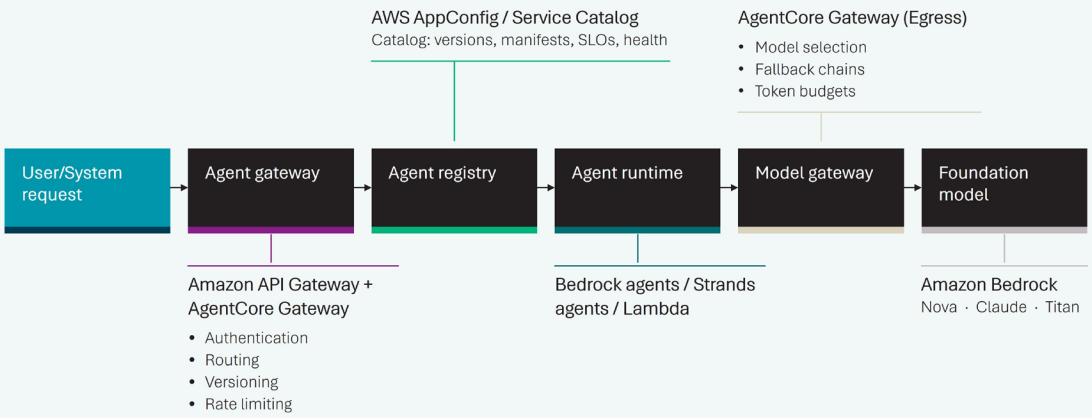
Model fallback chains

Define primary and fallback models for each agent type. If the primary model returns low-confidence responses, the Model Gateway falls back automatically..

Cost controls

Token budgets should be enforced at the model gateway level by agent, task type, and time window. Without this, a single misconfigured agent can generate runaway inference costs.

Architecture pattern on AWS



Gateway checklist

- Every agent interaction passes through the Agent Gateway (no direct agent addressing).
- Every model call passes through Model Gateway (no direct model calls from agent code).
- All agents are registered with version, SLO, and capability manifest metadata.
- Canary deployment is configured for agent version rollouts.
- Model fallback chains are defined for every agent type.
- Token budgets are enforced at the Model Gateway level.
- Gateway logs capture request/response metadata for every interaction.
- Rate limits are configured per agent, per consumer, per time window.
- Agent traffic is protected against API abuse and anomalous request patterns (agent-aware API security tools such as Cequence provide behavioral anomaly detection beyond what rate limiting alone can catch).

REFERENCES

- **Amazon API Gateway**: Routing, versioning, throttling
- **Amazon AgentCore Gateway**: Agent and model gateway management
- **AWS AppConfig**: Agent registry, feature flags, kill switches
- **Amazon Bedrock**: Foundation models, guardrails, prompt management
- **Cequence**: Agent-aware API security and behavioral anomaly detection (cequence.ai)

Capability 2
Capability boundaries and semantic isolation

Network isolation is a prerequisite, not a solution. Even within a properly configured VPC and least-privilege IAM roles, an agent can still produce outputs that violate data governance requirements, expose sensitive attributes, or exceed its intended role.. The boundary must follow the context, not just the connection.

The distinction that matters

IAM controls what an agent can access. Semantic isolation controls what an agent can do with what it accesses. A billing agent that can read customer records to resolve payment disputes does not need to include account numbers, payment history, or credit scores in responses to frontline agents. The data is accessible. The context is not appropriate for that output.

This requires ABAC policies that travel with the data through the agent's reasoning chain, not just policies at the data access layer.

What to build

Agent capability manifests

A structured definition of what each agent is permitted to see, reason about, and include in its outputs. This is not an IAM policy. It is a semantic contract. It defines allowed data attributes, prohibited output patterns, permitted tool invocations, and escalation triggers. Manifests are versioned and governed through change control.. The Agent Registry stores and serves them.

Context filtering

A runtime layer that checks agent inputs and outputs against the capability manifest. Inputs that contain prohibited attributes are filtered before reaching the agent. Outputs that violate the manifest are intercepted before they move downstream.

ABAC policy engine

Policies that operate at the data attribute level, not the resource level.

Examples include:

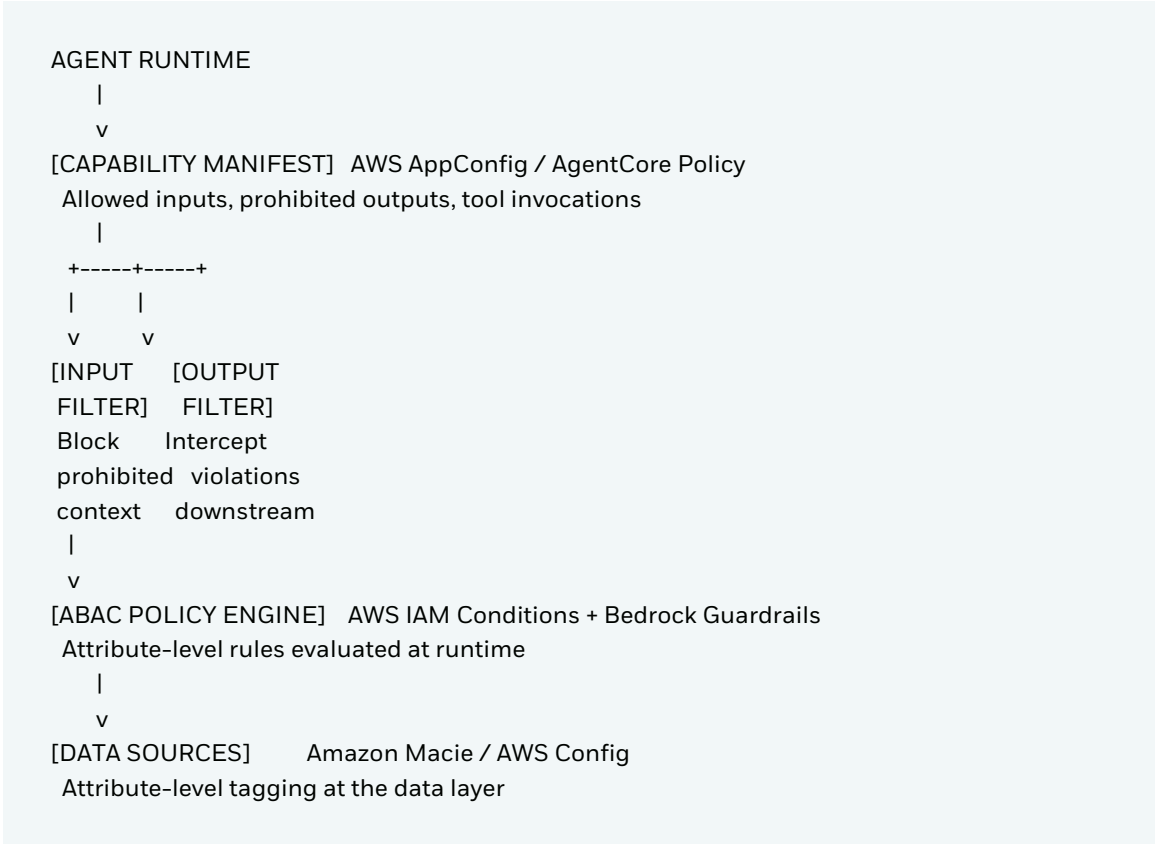
- Customer name: allowed.
- Customer personally identifiable information (PII): blocked.
- Account balance: allowed for billing agents and blocked for support agents.

These policies are evaluated at runtime, not just at access time..

Isolation between agent types

Domain agents and service agents operate in isolated contexts. A domain agent handling a customer incident should not have visibility into the operational state of infrastructure systems unless that is explicitly within its capability manifest.

Architecture pattern on AWS



Semantic Isolation checklist

- Every agent has a capability manifest defining allowed inputs, outputs, and tool invocations.
- Context filtering is applied to inputs before they reach agent reasoning.
- Outputs are validated against the capability manifest before being passed downstream.
- PII and sensitive attributes are tagged at the data layer and referenced in ABAC policies.
- Domain agents and service agents operate in isolated contexts.
- Capability manifests are versioned and change-controlled.
- Policy violations are logged and trigger alerts, not silent failures.
- Guardrails are configured at the model level as a second line of defense.

REFERENCES

- **Amazon Bedrock Guardrails**: Content filtering, PII redaction, topic controls
- **AgentCore Policy**: Attribute-level agent governance
- **AWS IAM Conditions**: Context-aware access control
- **Amazon Macie**: Sensitive data discovery and tagging
- **AWS Config**: Policy compliance validation

Capability 3
Runtime governance, security, and drift detection

An agent approved for production is not an agent approved forever. The world changes around it: data distributions shift, business rules evolve, model providers update their underlying models. Without continuous evaluation, degradation is invisible until it becomes a business problem.

> **Security is** a core principle **principle of runtime governance, not a separate workstream. The same pipeline should detect both accuracy drift and behavioural anomalies, including prompt injection, context poisoning, and guardrail bypass attempts..**

What to build

Agent SLOs
Define accuracy thresholds, response quality bounds, and behavioral consistency requirements for each agent. These are not latency SLOs. They are semantic SLOs. An agent that responds in 200ms with a wrong answer is not meeting its SLO.

Continuous evaluation pipelines
Evaluation does not stop at deployment. Run evaluation jobs against production traffic samples on a scheduled basis. Compare current agent performance against the baseline. Flag deviations before they compound.

Drift detection
Three types require monitoring:

- Data drift: the statistical properties of inputs are changing.
- Concept drift: the relationship between inputs and correct outputs is shifting.
- Prompt drift: accumulated changes to agent instructions produce inconsistent behaviour. Each requires a different detection approach.

Guardrails as runtime controls
Guardrails must run at every model invocation in production. Configure content filters, PII redaction, topic denial, and grounding checks. A high rate of guardrail triggers on a specific agent is an early indicator of prompt injection attempts or capability manifest drift.

Kill Switches
Every agent must have a remotely triggered kill switch that routes traffic to a fallback agent or human queue without requiring a deployment. Teams must be able to test it in production without causing an incident.

Accountability models
Every agent action must produce an audit trail: who triggered it, which agent version, which model, what the output was, whether a guardrail fired, and whether a human reviewed it.

Architecture pattern on AWS

Runtime governance checklist

- Semantic SLOs are defined for every agent in production.
- Evaluation pipelines run against production traffic on a scheduled basis.
- Baseline agent performance is captured at deployment and updated quarterly.
- Data drift, concept drift, and prompt drift are monitored independently.
- Guardrails are active at every model invocation in production, not just in testing.
- Guardrail trigger rates are monitored as a security signal, not just a compliance metric.
- Behavioral anomalies consistent with prompt injection are detected and alerted.
- Kill switches are configured and tested for every agent.
- Audit trails capture agent version, model called, inputs, outputs, guardrail events, and human review status.
- Responsible AI policies are evaluated continuously, not just at approval time.

Capability 4
Intelligent orchestration and human fallback

The orchestration layer is the control plane's decision engine. It determines which agent handles which task, which model to invoke, and when to escalate to a human. When done well, this is where most operational incidents can be resolved autonomously with humans handling exceptions by design.

The supervisor/worker pattern

Supervisor agent

Receives incoming tasks. Assesses complexity and confidence. Routes to the appropriate worker. Monitors execution. Triggers escalation when thresholds are breached.

Does not execute tasks directly.

Worker agents

Domain agents: product-scoped, deep context, small blast radius. Handle incidents specific to a product or service domain.

Service agents: technology-scoped, handle infrastructure, networking, and common operational patterns that cut across domains.

What to build

Confidence-based routing

The Supervisor Agent evaluates task complexity and confidence before routing. High-confidence tasks route to the appropriate worker. Low-confidence or ambiguous tasks route to a more capable model or trigger human review before execution.

Model selection matrix

Define a matrix across task type, complexity level, cost tolerance, and latency requirement. The orchestration layer applies this at runtime. This controls inference costs without compromising quality where it matters.

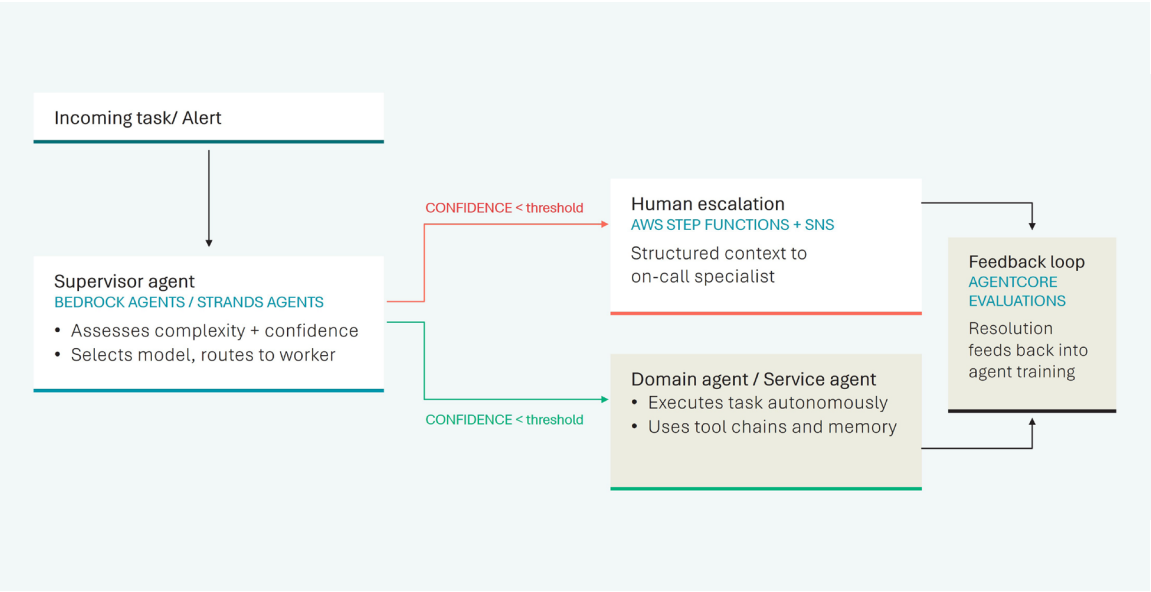
Escalation thresholds

Define explicit confidence thresholds for human escalation. Below threshold, the agent pauses and routes to human review. This is not a failure condition. It is a designed outcome. On-call specialists receive structured context: what the agent was attempting, why it could not proceed, and what information it needs.

Feedback loops

When a human resolves an escalated incident, that resolution becomes a training signal. Over time, the escalation rate for recurring patterns should decrease.

Architecture pattern on AWS



Orchestration checklist

- Supervisor/Worker pattern is implemented (Supervisor does not execute tasks directly).
- Domain agents & service agents are scoped separately with distinct capability manifests.
- Confidence thresholds are defined and documented for every agent type.
- Model selection matrix is defined: task type vs model capability vs cost.
- Human escalation routes to specific on-call owners, not a generic queue.
- Escalated incidents include structured context for the human reviewer.
- Escalation outcomes feed back into agent evaluation pipelines.
- Orchestration decisions are logged for audit and improvement.

REFERENCES

- **Amazon Bedrock Agents:** Multi-step task orchestration
- **Strands Agents:** Lightweight agent execution framework
- **AWS Step Functions:** Workflow orchestration with human approval steps
- **Amazon EventBridge:** Event-driven routing and filtering
- **Amazon SNS:** Escalation notification routing

The operating model and UST PACE agent control plane

Running an agentic control plane in production

Architecture is the easy part. The operating model is where most implementations fail. Without clear ownership, defined roles, and operational rituals, a control plane becomes infrastructure that nobody maintains.

Roles and responsibilities

Platform engineering team
Owns the control plane infrastructure. Responsible for the Agent Gateway, Model Gateway, evaluation pipelines, drift detection, and kill switches. This team does not build agents. They build and operate the platform that agents run on.

Agent engineering team
Owns individual agents. Responsible for capability manifests, SLO definitions, model selection logic, and agent performance. Works with the Platform Engineering team to onboard new agents through the control plane.

AI governance function
Owns responsible AI policy definitions, audit trail review, and escalation threshold governance. In most enterprises this sits within the CISO or Chief Data Officer function.

On-Call specialists
Domain experts who receive escalated incidents from agents. Own the feedback loop back to the Agent Engineering team.

Operational rituals

Weekly

Review agent SLO dashboards. Flag any agents approaching drift thresholds.

Review escalation rates by agent and task type.

Monthly

Run full evaluation pipeline against each agent. Compare against deployment baseline. Review responsible AI audit logs.

Update capability manifests for any agents whose operational context has changed.

Quarterly

Review model selection matrix against current model capabilities and costs. Conduct kill switch tests for all production agents.

Review escalation feedback and update agent training.

What good looks like at 90 days

- Escalation rate is declining as agents learn from resolved incidents.
- Drift detection is catching accuracy issues before they surface as business problems.
- Inference costs are declining as model selection logic improves.
- Audit trails provide clean evidence for compliance reviews.
- Agent onboarding time is decreasing as the platform matures.

> **If you are not seeing these indicators at 90 days, the control plane is not operating as designed. Common causes: evaluation pipelines not running on schedule, capability manifests not maintained, escalation feedback loop not closed.**

UST PACE agent control plane: The reference implementation

Everything in this guide is delivered through UST PACE, the Agent Control Plane for AI-Native Platform Engineering, built on Amazon Bedrock and AgentCore.

UST PACE is not a product. It is a production-ready implementation of the architecture and operating model described in this guide, delivered as a managed service for enterprises running on AWS.

What UST PACE delivers

- Agent Gateway and Model Gateway, configured and managed
- Capability boundary framework with ABAC policy enforcement
- Continuous evaluation pipelines and drift detection
- Supervisor/Worker orchestration with confidence-based routing
- Human fallback integration with on-call workflows
- Operational Intelligence data layer: runbooks, incident history, reliability scorecards
- Reliability and Assurance layer: policy, guardrails, audit trail, boundaries

> **On AWS, UST PACE runs on**
>
> Amazon Bedrock | AgentCore | Amazon Nova | EKS | Lambda | CloudTrail | CloudWatch | Bedrock Knowledge Bases | Step Functions | OpenSearch | DynamoDB | EventBridge

Is your enterprise ready?

Before deploying an agentic control plane, assess your readiness across five dimensions:

Dimension	Not ready	Developing	Ready
Agent Observability	No production monitoring	Latency / availability only	Behavioral evaluation pipelines
Semantic Governance	IAM only	Some output filtering	ABAC policy engine
Drift Management	No detection	Manual reviews	Continuous automated detection
Orchestration	Direct agent calls	Basic routing	Supervisor/Worker with confidence routing
Human Fallback	Ad hoc escalation	Defined thresholds	Structured escalation with feedback loop

Get started

UST PACE readiness assessment

A structured engagement that maps your current state against the five dimensions above, identifies gaps, and produces a prioritized roadmap for building or deploying your agentic control plane.

- AI-Native Software Engineering: Intelligent Delivery
- UST PACE: Agent Control Plane [coming soon]
- Intelligent Operations [coming soon]

Contact UST to schedule your readiness assessment.

Together, we build for boundless impact

Since 1999, UST has worked side by side with the world's best companies to make a powerful impact through transformation. Powered by technology, inspired by people, and led by our purpose, we partner with our clients from design to operation. Our digital solutions, proprietary platforms, engineering, R&D, products, and innovation ecosystem turn core challenges into impactful, disruptive solutions. With deep industry knowledge and a future-ready mindset, we infuse expertise, innovation, and agility into our clients' organizations—delivering measurable value and positive lasting change for them, their customers, and communities around the world. Together, with 30,000+ employees in 30+ countries, we build for boundless impact—touching billions of lives in the process.

ust.com

Version 0101-20260331

U ▪
S T