White Paper

The Autonomous State Control Plane

A Reference Architecture for Sovereign AI Systems

Dr. Jun He
The OpenKedge Initiative

The Qāfila Doctrine / عقيدة القافلة

إن الأنظمة ذاتية التحكم تتطلب ما هو أكثر من مجرد الحركة؛ بل تتطلب عبوراً محكوماً بما يلي نية معلنة، وصلاحيات مقيدة، ونقاط عبور متحقق منها، ووصول خاضع للمساءلة.

"Autonomous systems require more than motion; they require governed passage — declared intent, bounded authority, verified checkpoints, and accountable arrival."

Draft v0.1 · May 2026

openkedge.io · github.com/openkedge

Executive Summary

AI systems are moving from advice to action. Modern agents can approve workflows, modify cloud infrastructure, generate and deploy code, trigger industrial automation, and initiate financial operations. Once AI can change real systems, leaders face a governance question: who decides what the AI is allowed to do, under which policy, with what authority, and with what evidence if the decision is challenged?

Mission

Enable institutions to use advanced AI in high-consequence work with stronger operational assurance, while keeping execution authority, evidence, and accountability under their own control.

The Autonomous State Control Plane is a reference architecture for governing AI agents before their outputs affect real systems. It establishes the governance boundary between AI reasoning and real-world action. A model or agent may propose, plan, write code, or recommend a change; the responsible institution then decides whether that proposal may execute through a control plane that enforces policy, identity, bounded execution contracts, evidence, and replay.

Core Doctrine

Models may be global. Execution authority must remain sovereign.

Here, sovereignty denotes control over the authority boundary: the policies, approvals, credentials, execution limits, evidence records, and audit processes that determine how AI-generated proposals affect systems, data, infrastructure, citizens, customers, or capital. A nation, enterprise, public agency, or sovereign investment platform can use advanced global, domestic, or open-source models without converting an external model into an unaccountable operator.

This is not an agent framework. It sits below tools that enable AI reasoning, planning, or API invocation, and provides the governance layer that determines whether an AI-proposed action is allowed, how it is constrained, who or what authorized it, what identity may execute it, and what evidence is captured.

Why This Matters Now

The governance gap is already apparent: AI-enabled systems call cloud APIs, execute scripts, modify databases, route workflows, draft approvals, and trigger operational changes. Yet many deployments still rely on static credentials, broad role assignments, prompt instructions, application logic, and retrospective logs.

These controls do not suffice for autonomous execution. API access checks whether a caller can invoke an operation; it cannot determine whether an AI-generated action is justified in the current

operational situation. Retrospective logging can explain a past failure, but it cannot prevent an unsafe change before it occurs. Static credentials identify a service account, but they do not prove that a specific action is justified by a specific intent at a specific time.

Prediction and generation → decision support → autonomous execution

The operational risk escalates when AI transitions from recommendation to execution. A flawed recommendation permits human review; a flawed autonomous action directly mutates infrastructure, capital, access control, public workflows, regulated records, or citizen-facing services.

This challenge is immediate. Agent frameworks, AI coding assistants, cloud remediation bots, workflow automation platforms, and operations agents are already entering enterprise and government environments. For many institutions, the question is no longer whether to deploy autonomous AI, but how to govern it before a consequential failure forces the issue.

What the Architecture Does

The Autonomous State Control Plane converts autonomous action from a direct tool call into a governed state transition. Rather than permitting an AI agent to act solely on credential possession, the architecture subjects every consequential action to a governed path:

1. **Intent, not raw action.** The AI proposes what it wants to do as a structured intent. The intent states the objective, target, scope, assumptions, risk, and justification.
2. **Policy and context evaluation.** The control plane evaluates the intent against machine-enforceable policy and current operational context. It can allow, deny, constrain, escalate, simulate, or defer the action.
3. **Bounded execution contract.** If the intent is approved, the system creates a bounded contract: what may happen, to which resource, within what time window, under which constraints, and with which evidence obligations.
4. **Proof-derived execution identity.** The system issues temporary, task-scoped authority derived from the approved contract, replacing the practice of granting broad standing privilege to useful agents.
5. **Evidence and replay.** Each decision produces evidence. The institution can reconstruct what was proposed, what policy applied, what context was available, who approved or escalated it, what authority was granted, what executed, and what happened afterward.

Reasoning → Intent → Policy → Contract → Identity → Bounded Execution → Evidence → Replay

The governing principle is practical: AI proposes; the institution decides. The control plane provides a clear, repeatable boundary. Where intent, policy, and context remain constant, the governance decision should remain consistent even if the underlying model, prompt, or vendor changes.

OpenKedge turns autonomous action from a direct API call into a governed, evidenced, and replayable state transition.

Sovereign AI: Global Intelligence, Sovereign Authority

Sovereign AI is often discussed as model ownership, national compute capacity, data localization, or domestic cloud infrastructure. These are important. But autonomous AI adds another question: who controls execution authority when model output can change real systems?

A nation does not need to reject global frontier models to preserve sovereignty. It needs to own the control plane between reasoning and action. An external model that proposes an infrastructure change through a sovereign governance layer can be useful without becoming authoritative. A domestic model that directly mutates systems without evidence, bounded authority, or institutional review can still be unsafe.

This creates a practical third path between dependence and isolation. Institutions can use global intelligence, domestic models, open-source models, specialized agents, and cloud-native AI services while keeping operational authority inside their own governance boundary. For a national AI program, the strategic asset is not only the model. It is the control plane that determines how intelligence is allowed to act.

The architecture is model-neutral, cloud-neutral, and policy-engine-pluggable. It integrates with existing policy systems, such as Cedar, OPA/Rego, institutional approval workflows, and cloud-native controls, and operates alongside current identity systems rather than replacing them. The goal is not to discard current security and compliance investments, but to provide a governance boundary for autonomous execution.

For Saudi Arabia and other national transformation programs, this distinction is practical. Digital government, sovereign cloud, smart-city systems, industrial operations, and investment platforms may all benefit from AI-assisted reasoning. The authority to approve, execute, audit, and appeal consequential actions should remain with the responsible ministry, agency, operator, or enterprise.

The Four Research Pillars

The Autonomous State Control Plane combines four research pillars into one governable system for autonomous AI. Each pillar answers a practical leadership question.

OpenKedge provides intent-based mutation governance. It converts direct actions into governed intents, evaluates policy and context, creates execution contracts, and records the evidence chain.

SAL, or Sovereign Agentic Loops, separates reasoning from authority. It allows external or non-authoritative reasoning to assist the institution without letting that reasoning directly control execution.

VAI, or Verifiable Agentic Infrastructure, reduces reliance on standing privilege by deriving execution identity from proof. Trust is reconstructed from evidence, not merely asserted by a credential.

PDD, or Protocol-Driven Development, governs AI-generated software through invariants and evidence. Generated code is admitted to high-consequence systems only when it satisfies the protocols required for the system it will affect.

Table 1: *Research pillars of the Autonomous State Control Plane.*

Pillar	Leadership Question	Role in the Architecture
OpenKedge	How does an AI proposal become a governed action?	Converts direct actions into structured intents, evaluates policy and context, issues execution contracts, and records evidence.
SAL	How can external reasoning be useful without becoming authority?	Separates AI reasoning from sovereign execution through intent isolation and control boundaries.
VAI	How can systems grant access without permanent broad credentials?	Derives temporary execution identity from validated intent, context, policy, and time.
PDD	How can AI-generated software be governed before use?	Uses protocols and evidence to admit generated code and components.

Together, these pillars define a reference architecture in which agents may reason, propose, and generate, while the control plane governs whether and how their outputs affect real-world state.

Intended Outcomes

This architecture aims not to delay AI adoption, but to make high-consequence deployment viable. Without a governance layer, institutions face a stark choice: restrict AI to advisory roles, or accept operational risk from ungoverned execution. The Autonomous State Control Plane offers a third path: governed autonomy.

Adopting the architecture is intended to support:

- safer autonomous execution across infrastructure, software, workflow, public-sector, and operational systems;
- reduced blast radius through bounded execution contracts and least privilege;
- less dependence on standing privilege for AI-agent workflows;
- machine-enforceable policy at the point where intent becomes action;
- auditable and replayable decisions supported by structured evidence;
- evidence packages for institutional audit, compliance review, public accountability, and appeal;
- governed use of external models without surrendering execution authority;
- support for sovereign cloud, multi-cloud, and multi-model operating environments;
- compatibility with existing policy engines, identity systems, and compliance frameworks such as the NIST AI Risk Management Framework and Zero Trust Architecture;

- a practical foundation for public-sector, regulated-industry, and national-scale AI adoption.

These outcomes do not rely on assumptions of flawless model reliability or permanent prompt-based policy alignment. They depend on an architectural discipline: separate reasoning from execution, evaluate intent before action, issue authority only when justified, and preserve evidence for replay.

Audience and Scope

This white paper is written for leaders and architects responsible for deploying AI where failure is not merely an application bug, but an operational, institutional, financial, or sovereign risk. That includes sovereign AI leaders, national AI agencies, government technology strategists, cloud infrastructure architects, security executives, AI platform leaders, sovereign investment platforms, and decision-makers in regulated industries.

This white paper is not a model benchmark, prompt engineering guide, chatbot architecture, agent framework, generic AI safety essay, or regulatory checklist. It is a reference architecture for governing how AI systems affect real-world state.

The OpenKedge Initiative

To learn more about how the OpenKedge Initiative is developing verifiable governance patterns for AI agents, and to explore executive use cases spanning sovereign and enterprise operations, visit our landing page at openkedge.io.

Later chapters develop the governance problem and the architecture in detail: resilience doctrine, sovereign execution boundaries, intent governance, proof-derived identity, protocol-driven development, deployment roadmaps, and national-scale application patterns.

The through-line is direct: autonomous AI becomes institutionally useful when it can be governed. The control plane is where that governance lives.

Contents

1 The Governance Problem

Engineers designed modern cloud, enterprise, and public-sector systems around a foundational assumption: execution originates from humans or deterministic software operating within predefined workflows. Autonomous AI weakens this assumption. An AI agent may interpret a high-level instruction, generate a plan, select tools, call APIs, synthesize code, and initiate changes to real systems. At that point, the central risk is no longer whether the model's output is correct, but whether a machine-generated intent is allowed to mutate reality without sufficient governance.

Autonomous AI shifts the unit of risk from software execution to machine-generated intent. Traditional control systems constrain principals, applications, roles, networks, and deployed software. They do not evaluate whether a newly synthesized action is semantically justified, contextually appropriate, bounded to the objective, and supported by evidence before it changes state.

The result is a governance problem, not merely a security problem. While security remains essential, the central failure mode of autonomous systems is not unauthorized access, but authorized, semantically unsafe mutation.

1.1 From Assistance to Mutation

AI systems are moving from assistance to mutation. In advisory use, an AI-generated answer constitutes information; a user reads it, compares it with sources, ignores it, or translates it into action through existing human procedures. In tool-using systems, the AI calls functions or retrieves data, while the surrounding application constrains the execution path. In autonomous systems, however, the AI proposes or initiates actions that directly alter external system state.

An AI-generated answer is information. An AI-initiated mutation is control.

Mutation is any action that alters system state. Terminating or provisioning cloud resources, approving a government workflow, changing an access policy, submitting a procurement request, modifying application code, changing traffic routing, or initiating a financial transaction all constitute mutation.

This shift fundamentally changes the consequences of failure. A poor recommendation permits correction before action; a flawed mutation directly impacts infrastructure, records, capital, permissions, or citizen services. The challenge is not that AI systems are uniquely error-prone, but that probabilistic reasoning now connects directly to deterministic systems of record and control.

In conventional automation, engineers define workflows, encode branching logic, test implementations, deploy services, and assign permissions in advance. The system may still fail, but its behavior remains bounded by code and process.

Autonomous agents invert this model. They synthesize plans at runtime, select tools dynamically, chain operations across systems, generate new code, and adapt their path as intermediate results arrive. This flexibility provides immense value, yet it requires explicit governance. When behavior is synthesized at runtime, the organization must govern the intent before the resulting

action becomes execution.

1.2 Why Existing Control Surfaces Are Insufficient

Existing infrastructure provides essential control surfaces: IAM and RBAC, API authorization, workflow approval systems, cloud audit logs, CI/CD checks, compliance reports, policy-as-code systems, human review processes, and operational runbooks. These mechanisms are useful, but they were designed exclusively for known actors, predefined workflows, and deterministic software behavior.

IAM and RBAC map principals to permissions. API authorization checks whether a caller may invoke an operation. Workflow systems route tasks through predefined approval paths. CI/CD checks test software against build, security, and deployment rules. Retrospective audit logs record events after they occur, and compliance reports summarize control presence.

These partial controls do not, by themselves, provide autonomous governance.

API authorization checks whether a caller can invoke an operation; it cannot determine whether this operation is semantically safe in the current operational situation. A cloud API may accept a request to delete a resource because the caller has the necessary permission, but it lacks context on whether the resource supports a critical public service, whether the user requested a simple cost estimate, whether a safer read-only action was available, or whether the operation conflicts with a maintenance freeze.

A permission boundary is not the same as a governance boundary.

Conventional controls rarely evaluate semantic intent or determine whether a generated plan aligns with the true objective. They typically lack the operational context required to assess blast radius, fail to verify if the requested execution identity is justified, and omit the structured evidence needed to replay the decision path from reasoning to action.

Human review can close some of these gaps, but it does not scale cleanly to high-frequency agentic execution. A human approver often sees a summary without the underlying context, tool chain, policy basis, or alternative paths. If the approval interface presents a plausible explanation rather than a structured intent and bounded execution contract, review becomes a weak semantic checkpoint rather than rigorous governance.

Compliance systems face a similar limitation: they verify that required controls exist and that logs are retained, but they cannot decide in real time whether a machine-generated intent should mutate a system under active conditions. Autonomous governance requires decisions before execution, not merely reports after.

1.3 The Failure of Direct Agent Execution

Direct agent execution allows a reasoning layer to call operational tools without an independent governance boundary. This approach is simple but fragile. Giving an agent tools and credentials allows it to pursue an objective, which may be acceptable for low-risk tasks. For high-consequence systems, however, it creates a structural weakness: the same component that reasons about an action also initiates the state change.

These failure modes are immediate and practical:

1.3.1 Context-Blind Execution

An agent may call an API without understanding current system state. It may know the syntax of a cloud operation without knowing the dependency graph, service criticality, traffic conditions, data classification, legal hold, incident state, or freeze window. The action can be technically valid and still wrong for the situation.

1.3.2 Over-Broad Authority

Agents often run behind credentials created for users, service accounts, roles, applications, or workloads. Those credentials may permit far more than the current task requires. If the task is to inspect a configuration, a credential that can mutate production infrastructure is excessive. If the task is to draft a procurement action, a credential that can submit it without bounded approval is excessive.

1.3.3 Semantic Mismatch

The agent may perform an action that is valid at the API layer but misaligned with the true objective. A user may ask to reduce cost, and the agent may terminate resources that are idle only because they are standby capacity. A user may ask to improve security, and the agent may tighten a policy in a way that breaks an emergency workflow. The operation is authorized, but the meaning of the action does not match institutional intent.

1.3.4 Tool-Chain Amplification

Small reasoning errors can be amplified across multiple API calls. An agent may make an initial incorrect assumption, observe a partial result, adjust the plan, and continue executing. Each tool call may be individually permitted. The chain may still compound into a larger operational error because no governance layer evaluates the aggregate intent, blast radius, and state transition.

1.3.5 Irreversible Mutation

Some mutations are difficult to undo. Infrastructure can be reprovisioned, but data loss, financial action, access exposure, public-sector workflow state, citizen-facing decisions, and generated code deployed into production can have consequences that are hard to reverse. Rollback is not a substitute for pre-execution governance.

1.3.6 Ambiguous Delegation

Direct execution also creates accountability ambiguity. If an agent acts after a user instruction, who authorized the action: the user, the agent, the application, the service account, the platform team, or the organization? If the system cannot distinguish between a suggestion, an intent, an approval, and an execution contract, accountability becomes blurred precisely where high-consequence systems require clarity.

Direct Agent Execution

A technically authorized API call may still be operationally unsafe. Autonomous governance must evaluate not only whether an action is permitted by credentials, but whether the proposed mutation is justified by intent, context, policy, and evidence.

Consequently, API access is not governance. A direct tool call expresses capability but does not establish legitimacy; governance must operate before execution.

1.4 Static Identity and Standing Privilege

Conventional identity systems bind privilege to a principal: a user, service account, role, application, workload, or machine identity. This model is foundational to modern security and remains necessary. However, autonomous execution introduces a second question that conventional identity cannot fully answer:

"What validated intent justifies this exact authority at this exact time?"

Static credentials do not suffice for autonomous systems. Standing privileges persist beyond the task, exceed the immediate intent, and remain reusable across contexts. They identify the actor but do not encode the operational justification for a specific mutation.

This distinction becomes critical when agents synthesize actions dynamically. A service account may be permitted to update a resource, but that does not make every agent-generated update appropriate. A user may have authority to approve a workflow, but an agent acting on the user's behalf should not inherit that authority without a validated intent and a bounded execution contract.

Autonomous systems require runtime authority derived from governance. Privilege must be computed dynamically from the validated intent, current context, applicable policy, and time-bounded execution constraints. This motivates proof-derived execution identity: identity must become evidence-bound and task-scoped, rather than principal-bound and persistent.

The practical objective is least privilege at the level of intent. Instead of asking whether an agent has a credential that can perform a class of operations, the system should ask whether this specific proposed mutation has been approved and whether the runtime identity is limited to the exact action, scope, and time window justified by that approval.

1.5 Logging Is Not Control

Audit logs record history; they do not govern action.

While traditional audit systems are necessary for security investigation, compliance, and debugging, retrospective logging is insufficient for autonomous agents. It can explain a failure, but it cannot prevent an unsafe action that has already executed.

The limitation is not merely that logs are retrospective; they omit the structured data required to reconstruct an autonomous state transition. An ordinary log records that an API was called, by which principal, and whether it succeeded. It does not record the original intent, the context snapshot, the policy decision, the rejected alternatives, the execution contract, the identity derivation, or the reasoning-to-action linkage.

This evidentiary gap is critical. If an agent modifies infrastructure, the organization must reconstruct more than the API trace. It must verify the objective, the approving policy, the evaluated context, the expected blast radius, the minted authority, the imposed constraints, and the subsequent verification outcome.

Without that evidence, audit becomes incomplete. Security teams may see the action but not the governance basis. Compliance teams may see an approval but not the underlying context. Operators may see an outage but not the reasoning chain that produced the change. Executives may receive a summary without knowing whether the system behaved according to policy.

The required evidence must be captured as part of the execution architecture. This motivates the Intent-to-Execution Evidence Chain, or IEEC. Later chapters develop the IEEC more fully. In this chapter, the essential point is simpler: autonomous systems require evidence before, during, and after execution. Evidence must be structured enough to support replay, not merely retained as a collection of logs.

1.6 The Governance Gap

The governance gap is the gap between what existing infrastructure can authorize and what autonomous systems require to be safely governed.

Current systems can often answer:

Can this principal call this API?

Autonomous governance requires answering a different set of questions:

What is the intent? Is the intent legitimate? What context is relevant? What policy applies? What is the blast radius? What authority is justified? What evidence must be recorded? Can the decision be replayed? Can the outcome be audited?

These questions shift the focus from access to governance. Access checks permissions; governance evaluates whether a proposed state transition should occur under the active intent, context, policy, and evidence requirements.

Table 1.1: *The gap between conventional authorization and autonomous governance.*

Question	Conventional Authorization	Autonomous Governance
Primary concern	Can this principal perform this operation?	Should this intent be allowed to affect this system now?
Unit of control	User, role, service account, or workload	Validated intent bound to context and policy
Time horizon	Predefined permissions	Task-scoped, time-bounded authority
Context awareness	Limited or external	Required for decision-making
Evidence	Logs and audit records after execution	Intent, context, policy, identity, execution, and verification evidence
Failure mode	Unauthorized access	Authorized but semantically unsafe mutation

This gap explains why autonomous AI cannot be safely integrated into critical systems by adding a model interface to existing APIs. The missing layer is a control plane that turns generated actions into evaluated intents, binds execution to policy and context, and records evidence sufficient for replay.

Governance Boundary

A permission boundary is not the same as a governance boundary. Permission determines what a principal can do. Governance determines whether a proposed action should be executed under the current intent, context, and policy.

1.7 Design Requirements for Governed Autonomy

The governance problem points to a concrete set of architectural requirements. These requirements define what any serious autonomous execution architecture must provide before it can be trusted in sovereign, regulated, or high-consequence environments.

1. **Intent representation.** AI-generated actions must be represented as explicit intents before execution. The system must know what change is being proposed, why it is being proposed, what resources are affected, and what outcome is expected.

2. **Context-aware policy evaluation.** Decisions must consider current operational state. Policy cannot be evaluated only against static roles or abstract permissions; it must account for system state, dependency, risk, jurisdiction, timing, and institutional constraints.

3. **Bounded execution contracts.** Approved actions must be constrained to the specific permitted mutation. The contract should define scope, target, time window, permitted operations, rollback expectations, verification requirements, and evidence obligations.

4. **Proof-derived execution identity.** Runtime authority must be computed from validated intent, policy, context, and time. The execution identity should exist because the governance decision justifies it, and it should expire when the bounded authority is no longer valid.

5. **Pre-execution enforcement.** Governance must occur before mutation, not only after. Enforcement points must reject actions that lack a valid intent, policy decision, execution contract, or proof-derived execution identity.

6. **Evidence-chain accountability.** Every stage must produce replayable evidence. The system must retain the intent, context, policy decision, identity derivation, execution event, verification result, and relevant metadata.

7. **Replay and simulation.** Decisions must be reconstructable and testable under alternate conditions. Replay allows institutions to understand what happened, evaluate policy changes, and test whether future decisions would remain within doctrine.

8. **Protocol-based admission.**
   Generated code and components must satisfy machine-enforceable invariants before use. If agents can produce software that becomes part of the execution substrate, that software must be admitted by protocol rather than confidence alone.

These requirements motivate the Autonomous State Control Plane. Later chapters develop them into the OpenKedge intent governance pipeline, proof-derived execution identity, the Intent-to-Execution Evidence Chain, and protocol-driven admission for generated software. The central conclusion of this chapter is direct: autonomous AI cannot be governed only by credentials, APIs, logs, or after-the-fact compliance. It requires a deterministic governance boundary before machine-generated intent becomes real-world mutation.

2 Resilience to the Unstable

Every major infrastructure era has required engineers to confront instability. Distributed systems confront partial failure, inconsistent replicas, latency spikes, and network partitions. Cloud systems confront noisy neighbors, dependency failures, resource contention, and cascading outages. Security systems confront adversarial inputs and compromised assumptions. Autonomous AI systems introduce a new form of instability: probabilistic reasoning connected to tools that can affect real-world state.

The response is not to deny this instability or wait for perfect agents. It is to architect around it.

Resilience to the Unstable

The objective is not to eliminate uncertainty from AI reasoning. The objective is to ensure that uncertainty cannot directly mutate systems beyond governed authority.

Rather than striving for perfect reasoning stability, this architecture builds systems that remain stable, governable, and auditable even when the reasoning layer is probabilistic, context-limited, or wrong. This is the conceptual bridge between engineering realism and sovereign institutional confidence. Engineers must respect instability; institutions must build for resilience.

Figure 2.1 illustrates the central OpenKedge doctrine: AI reasoning engines may propose actions, but they do not directly possess execution authority. Authority is derived only after intent declaration, context evaluation, policy validation, contract generation, bounded execution, and evidence capture.

2.1 The New Instability

Traditional infrastructure is already built around instability. Hardware fails. Networks partition. Replicas diverge. Queues back up. Dependencies return partial results. Human operators make mistakes under pressure. Attackers manipulate inputs. Control planes drift from data planes. Production systems are never as clean as architecture diagrams.

Modern engineering disciplines emerged because these instabilities could not be wished away. Site reliability engineering, distributed systems design, zero trust architecture, chaos testing, incident response, and policy-as-code all share a basic assumption: systems must continue to behave within acceptable bounds when some part of the environment is uncertain, degraded, or adversarial.

Autonomous AI adds a different category of instability. An agent might reason from incomplete or stale context, interpret ambiguous instructions differently across runs, or produce a coherent-looking plan that harbors hidden operational mismatches. It might generate code or configuration that passes superficial checks while violating deep invariants, or offer plausible but incorrect justifications, following a non-repeatable reasoning path that cannot be reconstructed from the final answer alone.

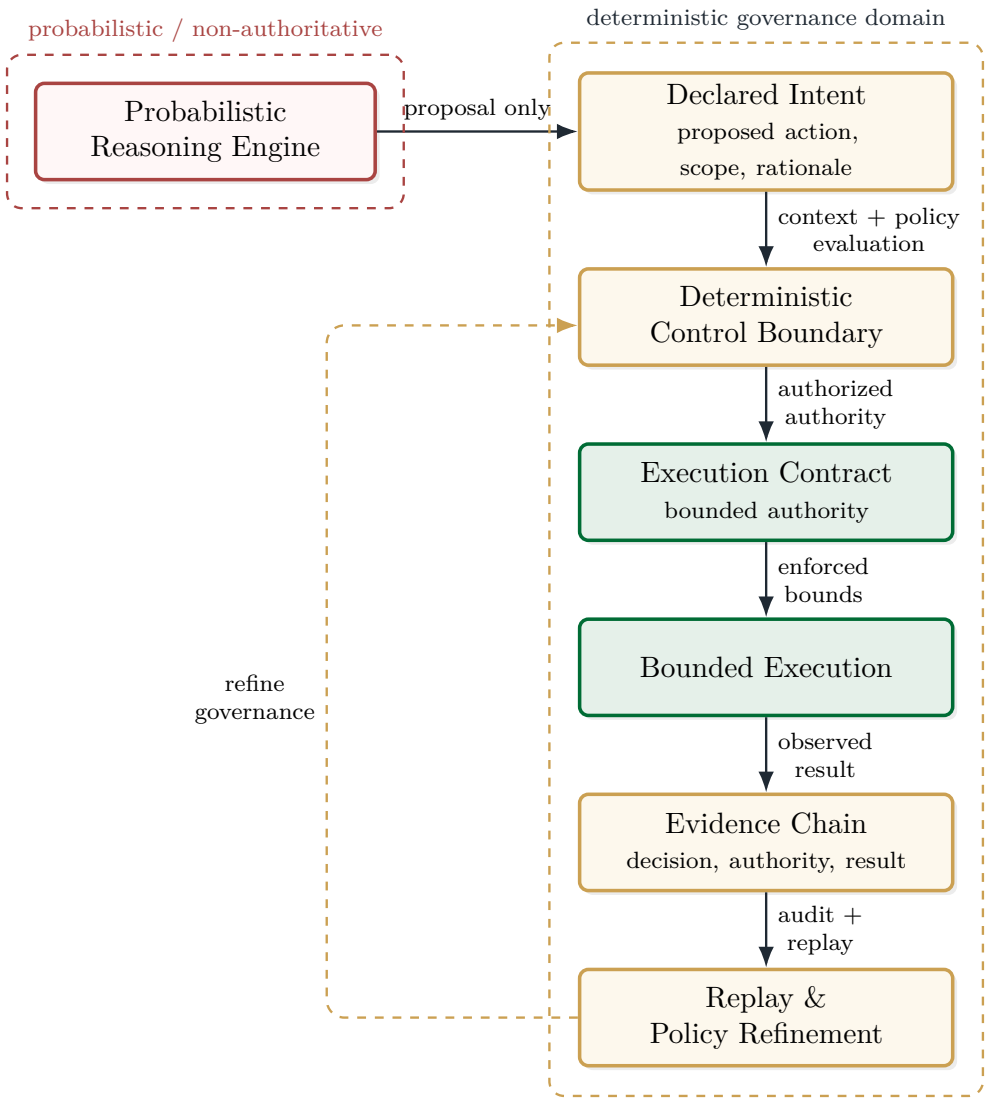


Figure 2.1: *Resilience to the unstable: probabilistic reasoning is isolated from execution authority by a deterministic governance boundary.*

The instability is not only that an AI model may produce an incorrect answer. The deeper risk is that its output is transformed into operational authority.

A generated answer remains information until it crosses into execution. A generated plan remains advisory until it becomes a tool sequence. A generated code patch remains a proposal until it is admitted into a build or deployment path. The risk changes when probabilistic reasoning is connected to systems that can mutate infrastructure, policy, money, data, software, public workflows, or physical operations.

The new instability therefore sits at the interface between reasoning and state. It is not enough to evaluate whether the model's text appears sensible. The architecture must determine whether the proposed action can be governed, bounded, enforced, evidenced, and replayed. The instability is manageable only if the system treats the reasoning layer as a source of proposals rather than a source of direct authority.

2.2 Respect the Unstable

Respect the Unstable is not a statement of fear. It is a statement of engineering discipline.

Respecting instability requires rejecting false assumptions. It demands that engineers verify agent accuracy, supply complete context, block direct mutation, refuse standing privilege, demand evidence for justifications, and never substitute confidence for control.

Respecting instability means designing boundaries around it.

This is familiar engineering logic. A distributed system does not assume the network is reliable. It uses timeouts, retries, consensus, idempotency, backpressure, and health checks. A secure system does not assume every request is benign. It validates identity, limits privilege, constrains access, monitors behavior, and preserves evidence. A cloud control plane does not assume every resource remains healthy. It observes, reconciles, rolls back, and limits blast radius.

Autonomous AI requires the same discipline. The reasoning layer may be powerful, but it is not a control boundary. A model's confidence, fluency, or internal chain of reasoning cannot replace policy evaluation. A tool call cannot replace authorization. A plausible explanation cannot replace evidence. A generated plan cannot replace an execution contract.

In control-system terms, autonomous reasoning should be treated as an input signal, not as the actuator itself. The signal may be useful. It may encode expert knowledge, strategic planning, translation, code synthesis, or rapid analysis. But the actuator that changes state must be constrained by a control system. The control system must decide whether the proposed change is admissible, what authority is justified, what conditions apply, and what evidence must be retained.

This also implies observability and reconciliation. A resilient AI control plane must be able to observe proposed intent, compare desired state with actual state, detect drift, and reconcile outcomes against policy. If execution diverges from the approved contract, the system should be able to stop, roll back, escalate, or record the deviation for replay. Instability is manageable when it is made visible and bounded.

Respecting the unstable reasoning layer does not make the architecture anti-AI. It makes the architecture fit for AI. Powerful reasoning can be used more confidently when the system is designed to absorb uncertainty without surrendering execution authority.

2.3 From Engineering Mantra to Institutional Doctrine

Respect the Unstable is the engineering mantra. Resilience to the Unstable is the institutional doctrine.

For engineers, the mantra means treating the AI agent as a non-deterministic reasoning component. Its output may be valuable, but it must be mediated. The agent proposes. The control plane evaluates. Execution occurs only through bounded authority. Evidence records what happened. Replay allows the institution to understand and improve the system.

For sovereign institutions, the doctrine is broader. Public systems, national infrastructure, regulated workflows, and strategic digital assets must remain stable even when AI reasoning is uncertain. A government platform cannot depend on every agent interpreting every instruction correctly. A critical infrastructure operator cannot rely on the assumption that every generated plan has full operational context. A regulated enterprise cannot treat a natural-language justification as a sufficient basis for state mutation.

A national AI system should rely on enforceable limits around agent authority, rather than assuming that every agent will reason correctly.

This is the institutional translation of engineering realism. The system may leverage frontier reasoning, specialized models, open-source models, domestic models, and external services. Yet institutional confidence comes from the deterministic controls that govern execution, not from a promise that reasoning will never fail.

The distinction also matters for accountability. When an autonomous system affects a high-consequence workflow, the institution must be able to answer what intent was proposed, what context was evaluated, what policy applied, what authority was granted, what action occurred, and what evidence supports the result. Institutional trust cannot rest on the model's explanation alone. It must rest on an evidence-backed control system.

2.4 Probabilistic Reasoning, Deterministic Governance

This white paper's architectural premise is simple: probabilistic reasoning should not directly produce deterministic state transitions.

Large language models and related AI systems operate by producing outputs under uncertainty. Their behavior can vary with prompts, retrieval state, conversation history, tool results, temperature, model version, provider behavior, system instructions, and hidden context. This does not make them unusable. It makes them different from the deterministic systems they are beginning to influence.

Infrastructure mutation requires deterministic governance. A production change either happens or does not. A credential is granted or denied. A workflow is approved, escalated, or rejected. A generated software component is admitted or excluded. A policy decision must be tied to identifiable context, policy version, scope, authority, and evidence.

Reasoning may be probabilistic, but execution must be governed.

The interface between these two worlds must be intent, not direct action. Intent is the structured expression of what the reasoning layer proposes to change and why. It gives the control plane something governable: objective, scope, target, expected effect, constraints, risk, context, and

evidence requirements. Direct action bypasses this conversion and asks the execution environment to treat model output as operational command.

Rather than attempting to force reasoning to be deterministic, the control plane enforces deterministic execution authorization. Given an intent, current state, active policy, identity context, time, and evidence requirements, the governance decision is reproducible: approving, denying, escalating, simulating, or constraining the proposed action according to rules that can be inspected and improved.

The control plane does not assume that the reasoning layer is malicious. It assumes something more general and realistic: the reasoning layer is non-deterministic, context-limited, and external to the execution boundary.

The control plane requires enforceable boundaries, not perfect agents.

This framing avoids two inadequate extremes: overtrust, which assumes model adequacy, delegates tools, and relies on retrospective logs; and paralysis, which assumes models can never participate in critical workflows. The control-plane architecture enables a practical middle path: allowing models to reason and propose, while requiring deterministic governance before execution.

2.5 Containment Without Isolation

Resilience to the Unstable also sets up the sovereignty argument. Organizations and nations need not accept a false binary between fully trusting external frontier models and isolating themselves from global AI innovation by attempting to rebuild every capability domestically. Both options are incomplete.

The Autonomous State Control Plane establishes a third path: leveraging global intelligence while containing execution within sovereign, institutional boundaries.

Sovereign Control

Models may be global. Execution authority must remain sovereign.

A model may assist with reasoning, planning, translation, code generation, policy analysis, simulation, and operational triage. It may be domestic, foreign, open, proprietary, frontier-scale, or specialized. The architecture should be able to use that reasoning without granting the model direct control over infrastructure, citizen-facing workflows, financial state, regulated data, or critical operations.

Sovereignty is preserved by controlling policy, identity, execution, evidence, and audit. The critical factor is not where the model is hosted or who trained it, but whether the institution owns the deterministic boundary through which any model output becomes action.

Containment without isolation is therefore a positive architecture. It lets institutions use capable reasoning engines while preserving authority over state mutation. It supports model diversity without surrendering governance. It allows multi-cloud and sovereign cloud deployments to share a common principle: reasoning can be external, but execution authority must remain inside the control plane.

This chapter is not specific to any one nation, provider, or sector. The doctrine applies anywhere autonomous reasoning is connected to high-consequence systems. Later chapters develop

this sovereignty boundary more directly, but the foundation is established here: resilience comes from controlling the interface between unstable reasoning and governed execution.

2.6 Control Boundaries for Autonomous Systems

The Autonomous State Control Plane introduces a set of control boundaries that convert unstable reasoning into governed state transitions. Each boundary targets a specific class of instability to contain it before execution.

The reasoning boundary prevents model output from becoming direct authority. External or internal models produce proposals, but those proposals must be converted into intent. Sovereign Agentic Loops and intent isolation make this boundary explicit.

The policy boundary decides whether an intent is acceptable under current conditions. It evaluates context, constraints, institutional rules, risk posture, and operational state. OpenKedge intent governance is the primary mechanism for this boundary.

The identity boundary binds runtime authority to validated intent. Instead of relying on standing privilege, the system derives execution identity from the approved intent, context, policy, and time. Verifiable Agentic Infrastructure develops this requirement into proof-derived execution identity.

The execution boundary limits what the system may actually do. Approved actions should execute only through bounded contracts that define scope, target, operations, time window, verification requirements, and evidence obligations.

The evidence boundary makes decisions and outcomes auditable and replayable. The system must record intent, context, policy decision, identity, execution, and verification events as part of the Intent-to-Execution Evidence Chain.

The protocol boundary governs generated software before admission. If an agent can generate code, configuration, policy, or system components, those artifacts must satisfy machine-enforceable invariants before they become part of the execution substrate. Protocol-Driven Development provides this admission model.

Table 2.1: *Control boundaries for resilient autonomous systems.*

Boundary	Purpose	Primary Mechanism
Reasoning boundary	Prevent model output from becoming direct authority	Sovereign Agentic Loops and intent isolation
Policy boundary	Decide whether an intent is acceptable under current conditions	OpenKedge policy evaluation
Identity boundary	Bind runtime authority to validated intent	Proof-derived execution identity
Execution boundary	Limit what the system may actually do	Bounded execution contracts
Evidence boundary	Make decisions and outcomes auditable and replayable	Intent-to-Execution Evidence Chain
Protocol boundary	Govern generated software before admission	Protocol-Driven Development invariants

These boundaries do not remove uncertainty from AI reasoning. They prevent that uncertainty from crossing directly into uncontrolled execution. Later chapters develop these control boundaries through Sovereign Agentic Loops, OpenKedge intent governance, Verifiable Agentic Infrastructure, and Protocol-Driven Development.

2.7 Design Implications

Resilience to the Unstable produces practical design implications for any institution building autonomous AI systems.

Agents should propose, not directly execute. The architecture should require agents to produce structured intent before operational tools can mutate state.

Intent should be explicit and machine-readable. Natural-language instructions and explanations are useful for humans, but governance requires structured objectives, scopes, constraints, targets, expected effects, and evidence requirements.

Credentials should be task-scoped and time-bounded. Standing privilege is poorly matched to autonomous execution because it persists beyond the specific intent and can be reused across contexts.

Policy should evaluate runtime context. A decision that is safe in one operational state may be unsafe in another. Policy must account for current state, dependency, jurisdiction, risk, timing, and institutional constraints.

Evidence should be produced at every stage. The system must preserve the path from reasoning to intent, policy, identity, execution, verification, and outcome. Evidence-backed trust is stronger than explanation-backed trust.

Replay should be built in by design. Institutions must be able to reconstruct decisions, test policy changes, compare alternate outcomes, and improve controls without relying on memory or incomplete logs.

Generated code should be admitted by protocol, not trust. AI-generated software may be useful, but usefulness is not admission. Generated artifacts must satisfy structural, behavioral, and operational invariants before they become part of the system.

Control planes should be sovereign, portable, and vendor-neutral. The governance boundary should not depend on a single model provider, cloud provider, tool framework, or deployment environment. Institutions should be able to change models and infrastructure while preserving deterministic control over execution.

The operational test is straightforward: when the model, prompt, tool chain, or infrastructure substrate changes, the institution should not lose control over policy, identity, execution, evidence, or replay. Resilience is achieved when the system can absorb change in the reasoning layer while preserving the authority and accountability of the control plane.

These implications do not reduce the ambition of autonomous AI. They make serious autonomy possible. Resilience to the Unstable is not a defensive posture. It is the operating principle for building AI systems that can safely use powerful reasoning engines without surrendering control over execution.

3 Sovereignty in the Agentic Era

The rise of frontier AI has created a strategic dilemma for governments and enterprises. The most capable models may be developed, hosted, or operated outside the institutions that wish to use them. Yet the systems affected by AI-generated decisions, including public services, infrastructure, financial workflows, citizen data, regulated operations, and industrial systems, remain bound by local law, institutional accountability, and sovereign responsibility. This creates a fundamental asymmetry: reasoning may cross borders, but execution authority cannot be allowed to drift outside the control boundary of the institution or nation responsible for the outcome.

Frontier intelligence can be global. Execution authority must be local, governed, and accountable.

Sovereign AI Thesis

Models may be global. Execution authority must remain sovereign.

In the agentic era, digital sovereignty extends beyond data or model ownership; it demands control over the deterministic control plane that governs how AI-generated intent affects sovereign systems. This chapter develops the architectural meaning of that claim.

3.1 The New Sovereignty Boundary

Older discussions of digital sovereignty often focus on data residency, cloud region location, domestic infrastructure, cybersecurity posture, national platforms, and local compliance. These concerns remain paramount. A nation or regulated institution must still understand where data is stored, who can access it, which jurisdictions apply, how infrastructure is operated, and how sensitive systems are protected.

Agentic AI adds a new sovereignty boundary: the boundary between reasoning and execution.

AI systems may produce plans, code, recommendations, tool calls, operational decisions, or workflow actions. When these outputs remain advisory, digital sovereignty poses familiar questions: what data was shared, which model was selected, and what decision did a human ultimately authorize? When those outputs can initiate or shape execution, the question changes: who controls the transformation of AI-generated reasoning into real-world state change?

In the agentic era, sovereignty is exercised at the point where reasoning becomes execution.

This point is where policy, identity, context, evidence, and institutional accountability must converge. A model-generated recommendation may be useful, but it is not sovereign authority. A generated deployment plan may be technically sophisticated, but it is not permission to mutate infrastructure. A generated public-sector workflow decision may be persuasive, but it is not a lawful institutional action until it passes through the appropriate authority boundary.

The new sovereignty boundary therefore cannot be defined only by geography or hosting location. It must be defined by control. The institution responsible for the outcome must control the interface

that converts model output into intent, evaluates that intent against local policy and context, grants or denies authority, bounds execution, and preserves evidence for audit and replay.

3.2 Foreign Reasoning and Sovereign Execution

Foreign reasoning means the reasoning process may occur in a model, vendor platform, external agent, open model stack, or multi-cloud environment outside the sovereign execution boundary. Rather than implying an adversarial relationship, "foreign" denotes reasoning that originates outside the local control boundary. Such reasoning remains external because the model is operated by a third party, updated outside institutional oversight, hosted in another jurisdiction, or embedded in a platform whose internal behavior is not governed by the institution using it.

Sovereign execution means that real-world state mutation occurs only inside a control plane governed by the institution or nation responsible for the system. Execution is sovereign when the affected institution controls the policy, identity, authorization, evidence, auditability, and enforcement conditions under which an AI-generated intent becomes action.

The control plane does not assume that external reasoning is malicious. It assumes that external reasoning is non-sovereign: it is not itself the authority empowered to mutate national or institutional systems.

This distinction is practical. An external model may draft a permit recommendation. It should not directly approve the permit. An external model may propose a cloud remediation. It should not directly mutate production infrastructure. An external model may generate code. It should not directly deploy that code into a critical system. An external model may recommend a procurement decision. It should not directly authorize financial or public-sector action.

External reasoning may advise. Sovereign execution must decide.

This framing sets up Sovereign Agentic Loops. The role of SAL is not to reject external reasoning. It is to convert reasoning into isolated intent and keep execution under sovereign control. The output of a model becomes a proposal. The control plane decides whether that proposal is admissible, what authority is justified, and what evidence must be preserved.

3.3 Sovereignty Without Model Isolation

Sovereign AI is often presented as a false binary. Option A is to depend entirely on foreign frontier models and accept that the most capable reasoning systems operate outside local control. Option B is to delay advanced AI deployment until every frontier capability can be built, hosted, and operated domestically.

There is a third path: use global intelligence while owning the deterministic governance layer that controls execution.

Sovereign AI is not achieved by trusting domestic or foreign models. It is achieved by owning the deterministic control plane that governs how any model may affect sovereign systems.

This distinction is powerful because it allows institutions to benefit from global AI progress without surrendering operational authority. A nation may use external models for translation, analysis, planning, code generation, or scenario evaluation while keeping domestic data boundaries,

policy authority, execution approval, runtime identity, audit, and evidence under local control. A regulated enterprise may use model diversity across providers while ensuring that all operational state changes pass through a common governance boundary.

Neuro-symbolic governance is the practical mechanism behind this third path. A sovereign institution can benefit from global neural reasoning while retaining symbolic control over policy, execution, and audit. The model may interpret, summarize, or propose, but the control plane performs the authoritative symbolic evaluation. This architectural structure operationalizes our core thesis: models may be global, but execution authority must remain sovereign.

Model sovereignty and execution sovereignty are related, but they are not identical. Model sovereignty concerns who builds, hosts, operates, or controls the model. Execution sovereignty concerns who controls how model output affects real systems. Model sovereignty can reduce strategic dependency, while execution sovereignty governs real-world state mutation.

Table 3.1: *Model sovereignty and execution sovereignty.*

Dimension	Model Sovereignty	Execution Sovereignty
Primary concern	Who builds, hosts, or operates the model?	Who controls how model output affects real systems?
Control object	Weights, training data, inference stack, hosting environment	Intent, policy, identity, execution contracts, evidence, audit
Risk addressed	Dependency on external AI capability	Loss of authority over real-world state mutation
Failure mode	Model unavailability, misalignment, or dependency	Ungoverned action, policy violation, audit failure, accountability ambiguity
Strategic value	Domestic AI capability and reduced dependency	Governed use of any model under sovereign authority

This implies not that model sovereignty is unimportant, but that model sovereignty alone is insufficient. A domestic model with unbounded execution authority can still produce unsafe or unauthorized state transitions. An external model whose outputs are converted into intent and governed by a sovereign control plane can participate in serious workflows without becoming the operator of those workflows.

3.4 The Cost of Direct Dependence

The risk is not the use of external intelligence. The risk is allowing external intelligence to become external authority.

Direct dependence occurs when an institution relies on an external model or platform not only for reasoning, but also for action. This occurs explicitly through direct tool access, or implicitly when the execution layer couples tightly to a specific model provider, agent framework, cloud platform, or proprietary workflow engine. The result is an architecture in which the institution may retain formal responsibility for outcomes while losing practical control over how those outcomes are produced.

The first cost is operational dependency. If an external system is required to execute or approve

actions, the institution depends on that system for operational continuity, not merely advice. Availability, version changes, policy changes, deprecations, and provider-specific behavior can affect the institution's ability to govern its own workflows.

The second cost is policy drift. A model or platform may behave in ways that do not align with local law, institutional norms, risk limits, or operational doctrine. This does not require bad intent. Drift can arise from model updates, prompt changes, retrieval differences, incomplete context, or optimization objectives that do not match the institution's obligations.

The third cost is data exposure. High-quality reasoning often benefits from context, but not all context should leave the sovereign boundary. Sensitive operational state, identity details, security posture, citizen information, procurement information, and regulated data may need to remain inside a controlled environment. A sovereign architecture must decide what context is necessary for reasoning and what context must remain protected.

The fourth cost is audit weakness. If the institution cannot reconstruct why an AI-generated action occurred, what context was considered, which policy applied, which identity executed, and how the outcome was verified, it cannot provide strong accountability. Logs from external systems may be useful, but they may not contain the full institutional decision path.

The fifth cost is accountability ambiguity. If an autonomous workflow produces a consequential action, responsibility may become blurred among the user, the model, the vendor, the platform operator, the service account, and the institution. Sovereign execution requires a clear line of institutional authority from intent to execution.

The sixth cost is vendor lock-in. If the execution layer is coupled to a specific model provider or cloud vendor, changing models or infrastructure can require reworking governance itself. This is strategically fragile. Maintaining a model-neutral and cloud-neutral control plane allows the institution to adopt new reasoning engines without rebuilding the authority boundary.

These are architectural risks, not vendor accusations. External models and cloud platforms may be valuable parts of national and enterprise AI systems. The question is whether they are used as reasoning substrates within a sovereign control architecture or allowed to become de facto operators of institutional systems.

3.5 The Sovereign Control Plane

A sovereign control plane is the institutional layer that receives AI-generated intent, evaluates it against local policy and context, computes bounded execution authority, enforces execution constraints, and records evidence under the control of the organization or nation responsible for the outcome.

Sovereign Control Plane

A sovereign control plane does not require a nation or institution to own every model. It requires ownership of the policy, identity, execution, evidence, and audit boundaries through which model-generated intent becomes action.

The sovereign control plane owns the intent interface. It decides how model output is converted into structured proposals. It owns policy evaluation. It determines which local rules, constraints,

approval paths, risk limits, and jurisdictional requirements apply. It owns context acquisition. It decides what operational state is needed to evaluate the intent and what context may or may not be disclosed to external reasoning layers.

It owns execution contracts. It turns approved intent into bounded authority with explicit scope, time, target, operation, verification, and evidence obligations. It owns execution identity. It computes runtime authority from validated intent, policy, context, and time rather than relying on broad standing privilege. It owns approval workflow. It defines when an action can be machine-executed, when it must be escalated, and when it must be denied.

It owns the evidence chain. It records intent, context, policy decision, identity derivation, execution event, verification result, and replay metadata. It owns replay and audit. It allows the institution to reconstruct decisions and test policy changes against historical or simulated conditions. It owns protocol admission. It governs whether generated code, configuration, policy, or system components satisfy invariants before entering the execution substrate.

The sovereign control plane does not require owning every model, every accelerator, every cloud substrate, or every software component. While strategically valuable, these assets are not prerequisites for governing execution; the critical requirement is ownership of the authority boundary.

Sovereignty resides less in the model alone than in the authority boundary around execution.

This is the architectural center of sovereign AI in the agentic era. Institutions can change models, add providers, use domestic systems, use external systems, and operate across clouds if the control plane remains the stable authority layer. Later chapters develop this sovereign execution boundary through Sovereign Agentic Loops, OpenKedge intent governance, proof-derived execution identity, and evidence-chain auditability.

3.6 Executable Policy and Institutional Authority

Sovereign control requires policies to become machine-enforceable at the execution boundary. Laws, regulations, organizational rules, risk limits, approval thresholds, and operational constraints must be translated into executable governance protocols when autonomous systems can initiate action.

Human-readable policy expresses authority. Machine-enforceable protocol operationalizes authority.

This does not mean that law, institutional judgment, or public accountability can be reduced completely to code. While not every legal or institutional judgment can be reduced to code, every autonomous system boundary must explicitly define which actions are eligible for machine execution, which require escalation, and which remain prohibited.

For example, a government workflow may require multi-level approval above a risk threshold. A cloud action may be blocked if it violates availability constraints or changes a protected dependency. A procurement action may require budget, authority, conflict, and compliance checks before submission. A generated code artifact may require protocol admission before deployment. A smart-city action may require safety interlocks, local review, or simulation before execution.

Executable policy therefore becomes the bridge between institutional authority and autonomous systems. It does not replace human governance. It gives human governance a form that machine-speed systems can enforce. The policy boundary must be explicit enough to constrain execution

and flexible enough to escalate actions that require human or legal judgment.

This connects directly to OpenKedge and PDD. OpenKedge treats autonomous action as governed intent evaluated against policy and context. PDD treats generated software as admissible only when it satisfies structural, behavioral, and operational invariants. Together, they turn institutional authority into enforceable control-plane behavior.

3.7 Implications for National AI Infrastructure

National and institutional AI infrastructure should be designed around sovereign execution boundaries. Rather than preventing the use of external reasoning, national infrastructure must ensure that external reasoning cannot initiate actions except through local, accountable governance.

First, infrastructure should include a sovereign intent gateway. External reasoning should enter through governed intent interfaces rather than direct tool access. The gateway should convert proposals into structured, inspectable objects before any execution decision occurs.

Second, policy authority should remain domestic or institutional. Policy should be authored, owned, reviewed, and enforced by the entity responsible for the affected system. External platforms may assist with analysis, but they should not become the final authority over local execution.

Third, the architecture should define a context boundary. Only necessary context should be shared with external reasoning layers. Sensitive operational details, identity data, citizen data, regulated information, and security posture should be minimized, obfuscated, or retained inside the control plane when possible.

Fourth, execution rights should be proof-derived. Runtime identity should be computed from approved intent and local context, scoped to the specific permitted action, and time-bounded. Standing privilege should not be the default authority model for autonomous workflows.

Fifth, every autonomous action should be reconstructable. Evidence-chain auditability should preserve the path from intent to context, policy, identity, execution, verification, and replay. This is essential for public trust, enterprise accountability, and regulated assurance.

Sixth, governance should be vendor-neutral. The control plane should operate across model and cloud providers so institutions can adopt new capabilities without surrendering the authority boundary. Model-neutral and cloud-neutral governance are strategic requirements for long-term sovereign AI infrastructure.

Seventh, high-impact actions should support replay and certification. The institution should be able to simulate proposed policies, replay historical decisions, test alternate conditions, and certify that autonomous workflows remain within approved doctrine.

The next chapter formalizes these implications into architectural principles: probabilistic intelligence, deterministic execution, separated reasoning and execution, evidence-based trust, and protocol-based admission.

4 Architectural Principles

The preceding chapters established the governance tension: autonomous systems introduce probabilistic reasoning into environments that require accountable execution. This chapter translates that tension into concrete architecture. The Autonomous State Control Plane organizes intelligence, authority, execution, evidence, and software admission for systems that manipulate physical or institutional state.

Trustworthiness in autonomous systems cannot rest on marginal model improvements, expanded test suites, or passive logs alone. It requires architectural principles that separate reasoning from execution, bind operational privilege to proof, and make consequential state transitions governable, evidenced, and replayable.

Architectural Doctrine

Autonomous systems should not be trusted because their agents appear intelligent. Trust should depend on deterministic governance, bounded identity, and replayable evidence.

4.1 From Problem Statement to Architecture

The preceding chapters defined the boundary separating autonomous AI from conventional automation. AI reasoning is probabilistic, yet its outputs are frequently transformed into real-world actions. Direct agent execution is unsafe in high-consequence systems; sovereignty is exercised at the execution boundary where reasoning becomes state mutation.

The axioms below map directly to system mechanisms: intent boundaries, policy evaluation engines, execution contracts, proof-derived identity, cryptographically bound evidence, and invariant-driven protocol admission.

An autonomous control plane must govern not only the initiating identity, but the contextual justification, the bounded authority, and the supporting evidence for every action.

This expands the control unit beyond static, identity-centric access control. Beyond verifying if a principal may invoke an API, the control plane must evaluate whether a proposed intent is permissible under the current operational context, restricted to the narrowest necessary authority, and backed by verifiable evidence for post-facto audit and replay.

Six axioms bridge this problem statement to concrete architecture:

1. Intelligence is probabilistic.
2. Execution must be deterministic.
3. Reasoning and execution must be separated.
4. Sovereignty resides in the control plane.

5. Trust requires evidence.

6. Code is admissible only through protocol.

Together, these axioms define the architectural doctrine behind OpenKedge, SAL, VAI, and PDD.

These axioms give different stakeholders a common review language. Executives can see accountability boundaries; architects can locate enforcement points; security teams can examine authority derivation; platform teams can isolate privilege propagation; and researchers can test model-independent assumptions. Each axiom can be audited through concrete questions: where does reasoning yield to governance, what evidence is sealed, how is runtime authority derived, and how is generated software prevented from bypassing system protocols?

4.2 Axiom 1: Intelligence Is Probabilistic

AI agents synthesize proposals, plans, and executable code through probabilistic reasoning. Consequently, their outputs are inherently non-deterministic and cannot be assumed correct, safe, or contextually aligned.

This axiom describes operational reality rather than criticizes AI capabilities. Deep learning models and agentic systems are useful for reasoning over ambiguity, generating multi-step plans, translating across domains, and identifying complex patterns.

Reasoning under ambiguity does not constitute operational authority. An agent may produce a fluent, syntactically valid explanation for a semantically hazardous action. It may operate on stale state, overgeneralize from incomplete telemetry, or misjudge dependency graphs. It might generate compiling code that violates protocol invariants, or invoke toolpaths that exceed its intended operational bounds.

Plausibility, fluency, and confidence are not governance signals. An agent may express high confidence in an erroneous action, or propose a highly plausible plan that violates safety constraints. Probabilistic reasoning remains highly valuable, but it cannot serve as execution authority.

SAL and OpenKedge both rely on this distinction. SAL confines the reasoning layer to proposing intents that must cross a security boundary before affecting sovereign systems. OpenKedge treats agent output as a proposed intent to be evaluated rather than an API command to be executed. By placing authority in the control plane, the architecture can use flexible models without making them the source of execution rights.

This shift changes how capability is evaluated. Higher model capacity may improve proposal quality, but it does not replace governance. In high-consequence environments, the relevant question is not only whether an agent can solve a task in isolation, but whether the surrounding control plane can determine when a proposal is admissible, when it must be constrained or escalated, and when it must be rejected. The architecture treats intelligence as a generator of candidate intent, not as a substitute for institutional judgment.

4.3 Axiom 2: Execution Must Be Deterministic

Systems mutating physical infrastructure, public services, financial ledger state, or industrial processes must operate under deterministic rules, bounded privileges, and immutable, auditable procedures.

Because execution mutates state by provisioning resources, altering policies, minting credentials, or committing transactions, authority cannot be derived from the heuristic confidence of the proposing model.

Deterministic execution does not guarantee perfect downstream behavior; it guarantees that authorization decisions are reproducible from recorded evidence. Given an intent, operational context, policy set, identity state, and evidence criteria, the control plane must produce a repeatable evaluation of whether to approve, deny, constrain, simulate, or escalate the action.

The control plane does not constrain the non-determinism of model reasoning; it enforces determinism on execution authorization.

The architecture avoids attempts to force uniform, deterministic model outputs. Instead, it subjects non-deterministic proposals to deterministic governance. The reasoning layer may explore many possible paths, but the control plane decides whether a given path is admissible.

Deterministic execution depends on bounded authority. Every approved action is represented by a formal execution contract defining the scope, lifetime, identity boundaries, and verification invariants of the execution. This requires proof-derived execution identity, where runtime authority is generated from a policy-compliant intent, and linked evidence chains map the decision to the final execution.

Determinism equally governs negative and intermediate outcomes. The control plane must reproducibly deny actions, request context, restrict proposal scopes, escalate to human operators, or route intents to simulation. These intermediate states prevent governance from collapsing into a binary gate that either stifles innovation or permits excessive authority. Precise, deterministic governance can approve a safe subset of an intent, restrict risky parameters, and preserve the exact decision provenance.

4.4 Axiom 3: Reasoning and Execution Must Be Separated

AI systems must never directly mutate real-world state. Reasoning must terminate in a proposed intent, and execution must occur only through a governed control plane.

This enforces a strict operational separation of concerns:

- The reasoning layer proposes, plans, explains, generates, recommends, and analyzes.
- The execution layer validates, authorizes, constrains, executes, records, and verifies.

Agents propose actions; the control plane grants authority.

This separation resolves the governance gap. Allowing model outputs to directly invoke APIs collapses reasoning and authority. Forcing model outputs into structured intents creates a governable, intermediate state. This intent can be inspected, evaluated, constrained, simulated, or escalated before execution begins.

SAL expresses this axiom as a sovereignty boundary. External reasoning can be useful, but it must cross an intent boundary before execution. The model may help form a plan, but sovereign execution happens only after governance. This allows global or external reasoning to contribute without becoming direct authority.

OpenKedge expresses the same axiom as intent-based mutation governance. Direct API calls are replaced by intent proposals. A mutation becomes a governed state transition, not a tool invocation. The control plane evaluates context and policy, issues execution contracts, and records evidence before the action affects real systems.

This separation preserves agent capabilities while making them governable. Rich agentic planning can continue, while execution authority is decoupled from the model and removed from ambient credentials or hidden tool-call privileges.

This boundary must appear in interface design. Rather than extending a model's direct execution capabilities, tool integrations should terminate at an intent boundary that captures the objective, target system, proposed mutation, safety bounds, and evidence requirements. As agent capabilities grow, stronger planning capacity increases the surface of synthesized proposals, so governance must prevent synthesis from silently turning into authority.

4.5 Axiom 4: Sovereignty Resides in the Control Plane

A sovereign state or institution may deploy external models, vendor platforms, or distributed cloud infrastructure, but sovereignty is preserved only when policy, execution, identity, and evidence remain under institutional control. This boundary forms the foundation of verifiable governance for AI agents: even when reasoning is delegated to global models, execution remains locally governed.

Sovereignty Boundary

Models may be global; execution authority must remain sovereign.

Model sovereignty is necessary but insufficient. A domestic model remains a hazard if granted direct access to privileged systems, whereas an external frontier model can be useful when constrained by a sovereign execution boundary. The relevant boundary is not where reasoning occurs, but who governs its transformation into action.

Execution sovereignty requires that the host institution owns the policy engines, context evaluation, identity minting, execution contracts, and cryptographic evidence trails. While external models may generate proposals, the logic of authorization and audit must reside locally within the institution.

This axiom links the four research pillars. SAL defines the reasoning-to-execution boundary. VAI derives execution identity from proof rather than inherited privilege. The IEEC supports institutional auditability by binding intent, context, policy, identity, and verification. PDD admits generated software only through protocol admission.

The sovereign control plane is the locus of authority where model-neutral reasoning translates into governed institutional action. It allows nations and enterprises to use diverse model providers without permitting them to become de facto operators of sovereign systems.

This design supports architectural portability. The control plane can operate across domestic models, frontier APIs, open-weight deployments, and deterministic backend services. It decouples policy evaluation, identity issuance, and audit from the model provider. Changing the model should not alter the governance boundary; migrating the cloud substrate should not break the evidence schema; updating policy should be a governed, explicit event rather than an implicit prompt adjustment.

4.6 Axiom 5: Trust Requires Evidence

Autonomous actions are not trusted based on agent assertion. They are trusted when intent, context, policy, identity, execution, and outcome are bound into replayable evidence.

Trust in autonomous systems should be constructed from proof, not assumed.

Conventional logs are insufficient. While they record that an event occurred, they fail to capture the intent, the policy version evaluated, the rejected alternatives, the execution contract bounds, the derived identity proofs, or whether the outcome aligned with the approved intent.

Verifiable evidence must span the execution lifecycle. Prior to execution, the system seals the intent, reasoning telemetry, context snapshot, and active policy rules. During execution, it records the active contract, proof-derived identity, and runtime enforcement checks. Post-execution, it captures verification outcomes, rollback events, and replay metadata.

Evidence-based trust operates independently of model explanations. An agent's explanation may help human operators, but explanation is not proof. The control plane must preserve the state variables needed to reconstruct and verify the state transition independently of the agent's self-narrative.

OpenKedge and VAI realize this axiom. OpenKedge structures the intent-to-execution path as a governed evidence chain, while VAI binds runtime trust to proof-derived execution identity. Consequently, execution identity remains tied directly to the validated intent and policy decision. The resulting evidence is not a passive API log, but the institutional lineage of autonomous action.

Evidence enables institutional learning without relying on anecdote. Replayable decisions allow organizations to evaluate whether policies were overly permissive, restrictive, ambiguous, or missing context. By linking outcomes back to intent and contract bounds, reviewers distinguish reasoning failures from policy, execution, identity, or verification failures. This distinction is critical for certification and assurance: a control plane that cannot explain its decisions cannot be trusted at a national or regulated-industry scale, regardless of model capability.

4.7 Axiom 6: Code Is Admissible Only Through Protocol

AI-generated software must never be admitted simply because it compiles, passes superficial tests, or appears plausible. It must satisfy machine-enforceable structural, behavioral, and operational invariants.

Dynamic code generation shifts the boundary of software trust. While agents can synthesize code rapidly, the generative model is never the root of trust. The protocol remains the sole authoritative artifact.

Under Protocol-Driven Development, implementations are replaceable, whereas the protocol remains authoritative.

$$\mathcal{P} = (\mathcal{S}, \mathcal{B}, \mathcal{O})$$

where $\mathcal{S}$ denotes structural invariants, $\mathcal{B}$ denotes behavioral invariants, and $\mathcal{O}$ denotes operational invariants.

Structural invariants define interfaces, schemas, dependency boundaries, state shapes, and composition rules. Behavioral invariants define safety properties, state-transition rules, input-output obligations, and consistency constraints. Operational invariants define deployment postures, observability requirements, resource limits, and rollback triggers.

This invariant matters as agents evolve to generate the code, configurations, and policies that make up the system's execution substrate. Admitting generated software based on plausibility shifts the trust anchor from the protocol to the generator, which is a boundary failure.

PDD treats generated software as a candidate realization within a protocol-defined admissible space. Implementations may change, regenerate, or vary by provider, but admission depends strictly on invariant satisfaction. PDD therefore acts as an upfront admission substrate rather than a post-facto runtime audit, governing generated software before it integrates into the execution substrate.

This principle protects the control plane from its own automation. As agents produce more controllers, policies, adapters, and workflows, the system must never admit components simply because they were generated within a trusted zone. The admission criteria remain strictly independent of origin: whether code is human-crafted, locally synthesized, or externally generated, it must satisfy the protocol defining admissibility. That protocol stands as the durable expression of institutional intent.

4.8 How the Four Pillars Compose

The four research pillars form a single, coherent architecture. SAL defines the reasoning boundary; OpenKedge governs intent and state mutation; VAI derives runtime trust from cryptographic proof; and PDD governs the admission of generated software.

The lifecycle of governed autonomous execution proceeds through a sequence of strict boundaries: isolated reasoning, governed intent evaluation, proof-derived authority, and invariant-verified software admission.

This composition is closed-loop. Execution evidence feeds back into policy refinement; historical replay tests new governance policies against real telemetry; and protocol admission refines software generators before deployment. The loop improves through evidence while remaining bound by deterministic governance.

An end-to-end execution sequence begins outside the authority boundary. A model reasons, plans, or generates a recommendation. SAL prevents that reasoning from turning into authority by converting it into isolated, non-executable intent. OpenKedge evaluates this intent against current operational context and institutional policy. VAI derives execution identity only if the intent satisfies all safety conditions. Execution then proceeds strictly within the bounds of a contract, while evidence records the complete decision path. If the change includes generated code

Table 4.1: *Mapping architectural axioms to the OpenKedge research pillars.*

Axiom	Architectural Response	Primary Pillar
Intelligence is probabilistic	Treat AI output as intent, not authority	SAL, OpenKedge
Execution must be deterministic	Use execution contracts and bounded runtime authority	OpenKedge, VAI
Reasoning and execution must be separated	Isolate external reasoning behind sovereign intent boundaries	SAL
Sovereignty resides in the control plane	Own policy, identity, execution, evidence, and audit boundaries	SAL, VAI
Trust requires evidence	Bind intent, context, policy, identity, execution, and verification into evidence chains	OpenKedge, VAI
Code is admissible only through protocol	Admit generated software through invariant satisfaction	PDD

Table 4.2: *The four-pillar lifecycle of governed autonomous execution.*

Pillar	Lifecycle Stage	Function
SAL	Reasoning boundary	Separates external reasoning from sovereign execution.
OpenKedge	Governance engine	Converts intent into policy-evaluated execution contracts.
VAI	Runtime trust layer	Derives execution identity from proof, context, policy, and time.
PDD	Admission substrate	Governs generated software through protocol invariants.

or configuration, PDD verifies its admissibility prior to deployment. Each pillar addresses a distinct failure mode while contributing to a single, unified governance narrative.

4.9 Design Consequences

Systems built under this doctrine must satisfy ten concrete design invariants.

These invariants serve as an evaluation checklist for system architectures, pilots, and enterprise procurement. Compliance requires more than assembling agent frameworks, policy engines, and logging tools. A compliant system must compose these components into a control plane that consistently converts probabilistic proposals into governed, evidenced, and bounded executions.

1. **Accept intent, not raw execution commands.** Agent outputs must become structured proposals before affecting system state.

2. **Evaluate policy against dynamic runtime context.** Governance decisions must incorporate real-time operational state, risk posture, dependencies, and regulatory constraints.

3. **Derive authority solely from validated intent.** Runtime execution rights must flow directly from a policy-approved intent rather than broad, standing privileges.
4. **Enforce task-scoped and time-bounded execution identity.** Runtime identity must be strictly confined to the approved action and expire automatically upon contract termination.
5. **Record cryptographic evidence across the execution lifecycle.** Evidence must capture intent, context, policy evaluation, contract boundaries, execution events, and outcomes.
6. **Support deterministic decision replay.** The control plane must enable complete, post-facto reconstruction of why a proposed action was approved, denied, restricted, simulated, or escalated.
7. **Subject generated software to protocol invariants.** Generated code must satisfy structural, behavioral, and operational invariants prior to deployment.
8. **Ensure model-neutral and cloud-neutral governance.** The control plane must preserve governance boundaries independently of underlying models, tool frameworks, and cloud providers.
9. **Separate reasoning from sovereign authority.** Models may assist with strategic planning or analysis, but execution authority must reside exclusively in the control plane.
10. **Leverage closed-loop learning via evidence and simulation.** The system must use replay, simulation, and execution telemetry to refine policies and protocols without surrendering active execution control.

These invariants make the abstract axioms operational. The remaining chapters instantiate these principles into a concrete architecture: Sovereign Agentic Loops for reasoning isolation, OpenKedge for intent governance, Verifiable Agentic Infrastructure for proof-derived execution identity, and Protocol-Driven Development for invariant-based admission.

The following chapter composes these principles into the Autonomous State Control Plane: a closed-loop governance architecture designed to transform probabilistic reasoning into bounded, evidenced, and replayable execution.

5 The Autonomous State Control Plane

The preceding chapters established the foundational axioms of sovereign autonomous systems. This chapter composes those principles into the Autonomous State Control Plane: a reference architecture that translates probabilistic AI reasoning into bounded, evidenced, replayable, and sovereign execution.

As the architectural boundary between reasoning systems and consequential state mutation, the control plane ensures that models and agents serve as valuable engines for planning, analysis, and recommendations without holding execution authority. That authority belongs exclusively to deterministic governance.

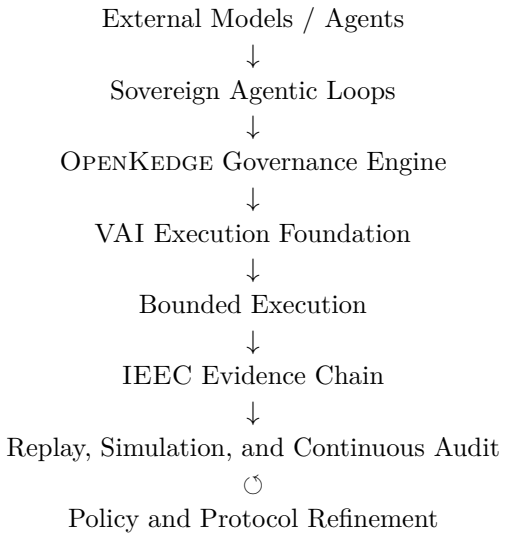
Control-Plane Doctrine

The control plane does not aim to make agents perfect; it ensures that imperfect agents cannot exceed governed authority.

5.1 Reference Architecture Overview

At the highest level, the Autonomous State Control Plane ingests proposals from external models, internal agents, workflow systems, and human operators. It converts these proposals into structured intent, evaluates them against policy and active operational context, derives bounded authority, executes mutations through constrained contracts, seals cryptographic evidence, and feeds this telemetry back into policy and simulation engines.

The conceptual stack can be summarized as follows:



Rather than a linear, top-down pipeline, the architecture forms a closed loop. Execution evidence does not merely populate compliance archives; it actively drives replay, simulation, policy

optimization, protocol improvement, and operational learning. In a mature deployment, the control plane serves as the institutional memory for all autonomous action.

Reasoning → Intent → Context → Policy → Contract → Identity → Execution → Evidence → Replay → Refinement

In this architecture, reasoning produces proposals; the control plane produces authority; execution produces evidence; and replay produces learning. The control plane converts raw AI outputs into governed state transitions.

The architecture enforces three complementary admissibility barriers. First, neuro-symbolic governance translates ambiguous reasoning into structured intent. Second, Protocol-Driven Development admits generated software into production through strict invariants. Third, compositional governance admits delegated workflows only when the composed loop preserves the sovereign boundary. Together, these boundaries ensure that autonomy expands through governed admission rather than raw execution.

Table 5.1: *Theoretical foundations and their architectural roles.*

Foundation	Architectural Role	Primary Location
Neuro-symbolic governance	Translates ambiguous reasoning into symbolic intent for deterministic evaluation	OpenKedge intent governance
Type-theoretic admission	Treats generated code as admissible only if it inhabits the protocol-defined implementation space	Protocol-Driven Development
Compositional semantics	Ensures delegated or nested workflows preserve governance boundaries under composition	Sovereign Agentic Loops
Evidence-chain replay	Makes decisions reconstructable for audit, simulation, and refinement	IEEC and replay layer

The white paper does not require any single formal method. It defines the governance boundaries that make formal methods, symbolic policy, typed admission, and replayable evidence useful in practice.

Sovereign Execution

External reasoning may propose; the sovereign control plane decides, authorizes, executes, records, and audits.

This distinction is the central architectural pivot. A model may recommend resource reconfigurations, workflow approvals, or operational mutations, but the control plane alone determines if a recommendation becomes an authorized mutation. It evaluates the proposed state transition against the policies, contexts, identities, and evidence requirements of the responsible institution.

The architecture does not eliminate autonomy; it disciplines it. Agents continue to reason, plan, generate, and coordinate across systems. However, they cannot convert their own output into authority. The control plane stands as the absolute boundary where proposal becomes permission.

5.2 Why the Architecture Is a Control Plane

The term control plane is deliberate. In distributed systems, the data plane handles execution—routing packets, reading and writing data, and running workloads—while the control plane governs configuration, policy, authority, lifecycle, and desired state. The control plane dictates how the system behaves and establishes the boundaries under which operations occur.

Autonomous AI demands a control plane because intelligence does not equal authority. A reasoning system can produce a plan, but a plan is not a permission. It can synthesize an API call, but a syntactically valid invocation is not institutional approval. An agent's explanation does not constitute evidence that the action is safe, scoped, authorized, and policy-compliant.

In this framework, the reasoning layer proposes actions and the execution layer performs bounded mutations, while the control plane governs whether, how, under what authority, and with what evidence those mutations occur. The control plane holds primary responsibility for intent, context, policy, identity, contracts, evidence, replay, and protocol admission.

The Autonomous State Control Plane must not be confused with a chatbot framework, an orchestration library, a model gateway, a generic logging system, or an IAM wrapper. While these components may reside within the deployment, none is individually sufficient. Chatbot frameworks manage conversations; orchestration libraries coordinate tool calls; model gateways mediate API access; logging systems record events; and IAM wrappers check static permissions. The control plane, by contrast, governs whether a proposed state transition is permissible under the active institutional context.

Linear Pipeline Risk

A one-way approval pipeline turns the control plane into a passive filter. To function as a true control plane, the evidence chain must actively feed replay, simulation, policy refinement, and protocol improvement.

The control-plane framing also clarifies accountability. When an autonomous system affects real-world state, responsibility cannot dissolve into a vague distribution across model providers, agent frameworks, user prompts, service accounts, and post-facto logs. The host institution must maintain an explicit, governing decision point. The Autonomous State Control Plane serves as that point: the single authority where proposed actions are evaluated, bounded, authorized, recorded, and reconstructed.

5.3 The Closed-Loop Governance Cycle

The closed-loop governance cycle is the operational heart of the architecture. It defines how a proposed action moves from reasoning to execution without allowing the reasoning layer to become the execution authority.

First, an AI model or multi-agent system generates a proposed plan or action, originating from natural-language instructions, operational events, workflow triggers, or generated software. At this stage, the proposal is entirely non-authoritative.

Second, the proposed action is converted into structured intent. Intent formation captures the target, scope, requesting principal, operational objective, and expected effects. Because raw model output is too unstructured to govern, structured intent creates a deterministic object that can be inspected, constrained, simulated, and audited.

Third, the system gathers relevant operational context. This context includes active resource states, dependency graphs, identity state, risk posture, and safety interlocks. Rather than broad, unchecked data disclosure, context acquisition is strictly scoped to what the policy engine requires.

Fourth, the policy engine evaluates the intent against active policy and context, determining whether to approve, deny, modify, escalate, or simulate the action. A mature control plane avoids binary gates; it can approve a safe subset, reduce scope, enforce compensating controls, or escalate the request.

Fifth, approved intent yields a bounded execution contract defining the permitted operation, target resources, time limits, allowed parameters, rollback rules, and verification obligations. The contract acts as the execution-facing artifact translating policy into enforceable boundaries.

Sixth, the system computes runtime authority from the contract, context, and requesting actor, establishing the execution identity. Rather than relying on broad, standing credentials, the control plane derives ephemeral authority dynamically from policy compliance.

Seventh, execution occurs strictly within the limits of the granted authority. Execution adapters invoke downstream APIs, databases, or deployment tools. Operating under the contract, the execution layer cannot deviate from the approved intent.

Eighth, the system records evidence across the execution lifecycle. The evidence chain binds the original intent, context snapshot, policy evaluation, contract limits, runtime identity derivation, and verification results into an immutable record.

Ninth, replay and simulation engines consume this evidence to reconstruct decision paths. Replay supports incident investigation, compliance audits, and policy debugging, while simulation allows authors to test how historical intents would behave under modified policies or risk parameters.

Tenth, evidence feeds back into policy and protocol refinement. The system learns institutionally not by permitting models to expand their authority, but by refining policies and protocols through systematic review of real telemetry.

This continuous feedback loop transforms the architecture from a passive gatekeeper into a dynamic, active control plane.

The architecture governs complex chains of action rather than isolated events. When a parent task decomposes into nested sub-intents, each sub-intent must be individually evaluated, bounded, evidenced, and replayable. Workflow composition is secure only when the governance properties of the parts are mathematically preserved by the whole.

5.4 Layer 1: Reasoning Boundary

The first layer is the reasoning boundary, implemented via SAL. It governs the interface between external, non-sovereign reasoning and sovereign execution. While external models, vendor APIs, or distributed agents contribute plans, code, and recommendations, they never inherit execution authority.

The reasoning boundary enables external intelligence to contribute to system planning without allowing external reasoning to dictate execution authority.

Rather than assuming external reasoning is hostile, the boundary treats it as fundamentally non-authoritative: probabilistic, context-blind, vendor-operated, and detached from live operational states. These characteristics require that all model proposals be enclosed by an intent boundary before affecting sovereign systems.

The architectural role of SAL is to isolate external reasoning and translate it into a governed intent proposal. The model receives only the minimal context necessary to reason. Sensitive operational telemetry is obfuscated, tokenized, or withheld entirely. Through this obfuscation membrane, the model reasons over an abstract representation of the problem without gaining access to the execution environment.

The intent boundary makes proposals governable. Rather than invoking tools directly, the agent must output a structured intent object. This object is validated, normalized, constrained, or rejected before any downstream execution occurs, preserving planning utility while blocking direct mutation.

This achieves containment without isolation. Institutions can leverage frontier APIs, open weights, and domestic models without allowing them to operate sovereign systems directly: models remain global, but execution authority remains strictly local.

5.5 Layer 2: Intent Governance

The second layer is intent governance, driven by the OpenKedge engine. It serves as the primary decision engine, evaluating structured intents against context and institutional policy.

OpenKedge replaces direct API execution with governed intent evaluation.

An OpenKedge intent binds the requested mutation to a target system, scope, objective, requesting principal, and expected side effects. The intent must be sufficiently explicit for the engine to predict the exact state changes. Vague prompts and raw tool-call fragments are rejected as insufficient for high-consequence execution.

The governance engine then resolves the active runtime context. For infrastructure mutations, context includes resource criticality, active incidents, owner metadata, and dependency topologies. For administrative workflows, it covers statutory constraints, approval thresholds, and data classifications. For software deployments, it encompasses interface invariants, testing logs, and rollback vectors.

Policy evaluation determines if an intent is admissible. Outcomes span approval, rejection, scope modification, escalation, or redirection to simulation. The capacity to dynamically modify and restrict intents is critical: if an action is overly broad but partially safe, the engine must be able to truncate the target scope, restrict the time window, or mandate a human-in-the-loop escalation.

Upon approval, the engine generates an execution contract specifying permitted resources, time bounds, allowed parameters, and verification obligations—the post-execution checks required to verify that the executed state matches the approved intent.

Far from mere documentation, the execution contract is a machine-enforceable artifact consumed by identity and execution layers. It translates policy decisions into bounded authority. Without it, the system reverts to broad ambient permissions, allowing an agent to perform any action permitted to its service account. The contract restricts execution to the exact operations authorized.

5.6 Layer 3: Execution Foundation

The third layer is the execution foundation, implemented via VAI. Approved intents must never inherit broad, standing credentials. Satisfying a policy checks a box; it must not grant permanent access to the target workload or cloud accounts.

Privilege is never assigned in advance; it is computed dynamically from cryptographic proof at execution time.

The proof basis consists of the approved intent, the policy decision, the context snapshot, and the active execution contract. The runtime identity generated for the task exists solely because this unified proof justifies a specific, narrow authority. If the supporting proofs are missing, inconsistent, or expired, no authority is minted.

This derived execution identity is ephemeral, task-scoped, and time-bounded. It grants only the exact privileges required for the approved mutation and expires automatically alongside the contract. By binding this identity directly to the decision evidence, the system allows auditors to verify exactly why the credentials existed and what operations they performed.

Practical deployments map this layer to cloud token services, workload identities, or just-in-time access brokers. While implementations vary, the architectural requirement remains constant: runtime authority must be derived dynamically from validated intent, never inherited from standing privileges.

This shifts the security posture: rather than asking if an agent possesses a credential broad enough to execute a task, the system asks if the task has produced evidence sufficient to justify a narrow credential. The agent brings zero authority to the system; the control plane derives authority strictly for the approved action.

5.7 Layer 4: Evidence and Replay

The fourth layer is evidence and replay, driven by the IEEC and active simulation services. Every governed action generates a cryptographically bound evidence chain. Far from a passive peripheral log, this evidence is a core execution artifact.

Evidence does not simply record what occurred; it preserves the exact proofs required to reconstruct why execution was authorized.

The evidence chain binds the original intent, requesting principal, context snapshot, policy decision, execution contract, runtime identity proofs, execution telemetry, and verification outcomes.

By preserving policy versions, context states, and cryptographic signatures, it allows the replay engine to reconstruct the exact decision environment.

This evidence supports critical institutional capabilities. For incident response, it pinpoints whether a failure occurred in reasoning, policy evaluation, execution, or post-facto verification. For compliance, it proves that required controls were satisfied before mutation. For policy authors, it isolates exactly which rules admitted or denied an action, enabling continuous refinement based on real telemetry rather than speculation.

Replay serves as a routine control-plane function rather than an emergency response. Policy authors test proposed policy updates against historical intent logs. They evaluate new risk thresholds against past decisions, simulate generated code against recorded scenarios, and analyze rejected intents to determine if policies are overly restrictive or proposals are malformed.

The evidence layer ensures durable accountability. Even when models are updated, prompts are modified, or human operators depart, the cryptographically sealed evidence chain remains as the permanent, institutional record of why an autonomous action was permitted or denied.

5.8 Layer 5: Protocol Admission

The fifth layer is protocol admission, implemented via PDD. Operating out-of-band as a continuous admission substrate, PDD governs generated software, configurations, policies, and workflows prior to active deployment.

PDD is an upfront admission filter, not a post-facto log.

As agents synthesize remediation scripts, deployment manifests, policy adapters, or system components, admitting these artifacts based on compilation or plausibility shifts trust to the generator. The architecture prevents this by requiring invariant satisfaction.

Under PDD, protocol defines the admissible implementation space:

$$\mathcal{P} = (\mathcal{S}, \mathcal{B}, \mathcal{O})$$

where $\mathcal{S}$ denotes structural invariants, $\mathcal{B}$ denotes behavioral invariants, and $\mathcal{O}$ denotes operational invariants.

Structural invariants define interfaces, dependency boundaries, and schema shapes. Behavioral invariants define state transitions, input-output obligations, and safety expectations. Operational invariants define resource limits, observability requirements, and rollback parameters.

PDD ensures that generated code is admissible before integrating into the operational environment where subsequent agents may run it. By linking with the evidence loop, execution telemetry reveals missing invariants or unsafe patterns, directly feeding protocol refinement.

This boundary is vital for sovereign systems. If an agent cannot mutate infrastructure directly but can generate code that executes with elevated privileges, the security boundary has simply collapsed. Protocol admission closes this gap, ensuring that generated software enters the system only by proving satisfaction of institutional invariants.

5.9 System Properties

The Autonomous State Control Plane generates robust system properties that are unattainable through prompt engineering, IAM rules, or manual review alone. These capabilities emerge directly from composing reasoning containment, intent governance, proof-derived identity, evidence chains, and protocol admission.

Table 5.2: *System properties of the Autonomous State Control Plane.*

Property	Meaning	Primary Mechanism
Boundedness	Actions cannot exceed approved scope	Execution contracts and proof-derived execution identity
Traceability	Actions are linked to intent and context	Intent-to-Execution Evidence Chain
Replayability	Decisions can be reconstructed and simulated	Evidence chain and replay engine
Sovereignty	Execution authority remains under institutional control	SAL and sovereign control plane boundaries
Composability	Multiple models, clouds, and policies can be integrated	Model-neutral and cloud-neutral governance interfaces
Least privilege	Authority is granted only for the validated task	Ephemeral execution identity
Auditability	Governance decisions produce durable evidence	IEEC and policy decision records
Adaptivity	Evidence improves future decisions	Replay, simulation, and policy refinement loop

Boundedness ensures that actions never exceed the execution contract. Since agents possess zero ambient authority, they cannot expand execution scope. Traceability connects every action back to the intent and context that justified it. Replayability guarantees that authorization decisions are mathematically reconstructable for audits and simulations. Sovereignty keeps execution authority inside the host institution, utilizing global models without permitting external access to local state. Composability decouples the control plane from specific models, clouds, or policy engines. Least privilege derives transient credentials dynamically from approved tasks. Auditability replaces passive logging with proactive, multi-stage evidence collection. Finally, adaptivity uses execution telemetry to systematically refine policies and protocols.

5.10 Reference Deployment Pattern

A reference deployment of the Autonomous State Control Plane organizes familiar infrastructure components around the governance cycle rather than direct tool execution.

1. **Agent Interface**: Ingests proposals from users, models, or workflow triggers.
2. **Intent Gateway**: Converts proposals into structured intents, validating schemas and target scopes.

3. **Context Provider**: Resolves active operational state from clouds, configuration databases, and identity systems under minimal disclosure principles.
4. **Policy Engine**: Evaluates intents against machine-enforceable policies (utilizing open standards like Cedar or OPA).
5. **Contract Issuer**: Translates approved intents into bounded execution contracts.
6. **Identity Broker**: Generates ephemeral, task-scoped execution tokens dynamically from active contracts.
7. **Execution Adapter**: Executes the mutation against downstream target systems, enforcing contract limits.
8. **Evidence Store**: Commits integrity-aware IEEC records of the entire lifecycle.
9. **Replay and Simulation Service**: Reconstructs decision environments to test policies and validate changes.
10. **Protocol Admission Layer**: Validates generated software against PDD invariants prior to deployment.

This pattern supports incremental deployment. An organization may initiate governance with simple intent capture and policy evaluation for cloud infrastructure, subsequently introducing proof-derived identity, evidence chains, and protocol admission as operational scale expands. The architecture does not demand a monolithic integration on day one; it requires only that each incremental step preserve the structural separation between reasoning and execution.

5.11 From Architecture to Implementation

The Autonomous State Control Plane is a portable reference architecture: a set of boundaries, roles, artifacts, and feedback loops implementable across diverse cloud environments, model providers, and policy engines.

The next chapters examine each layer in detail, beginning with the reasoning boundary established by Sovereign Agentic Loops. The SAL chapter explains how external reasoning is isolated from sovereign execution. The OpenKedge chapter explains intent governance, policy evaluation, execution contracts, and evidence chains. The VAI chapter explains proof-derived execution identity and runtime trust. The PDD chapter explains how generated software is admitted through structural, behavioral, and operational invariants. The later national-scale case study illustrates how the reference architecture can be applied cautiously to sovereign AI programs without claiming official adoption or endorsement.

Implementation should target the highest-risk mutation paths. In cloud platforms, this covers production infrastructure changes; in administrative services, citizen-facing approvals; in enterprises, financial authorizations and software deployments. Regardless of the domain, the governing questions remain identical: where does reasoning yield to intent, where is authority derived, and how does evidence enable replay?

The value of this architecture lies in a fundamental shift: autonomous systems no longer execute mutations because they can; they execute because the control plane can mathematically prove that they should.

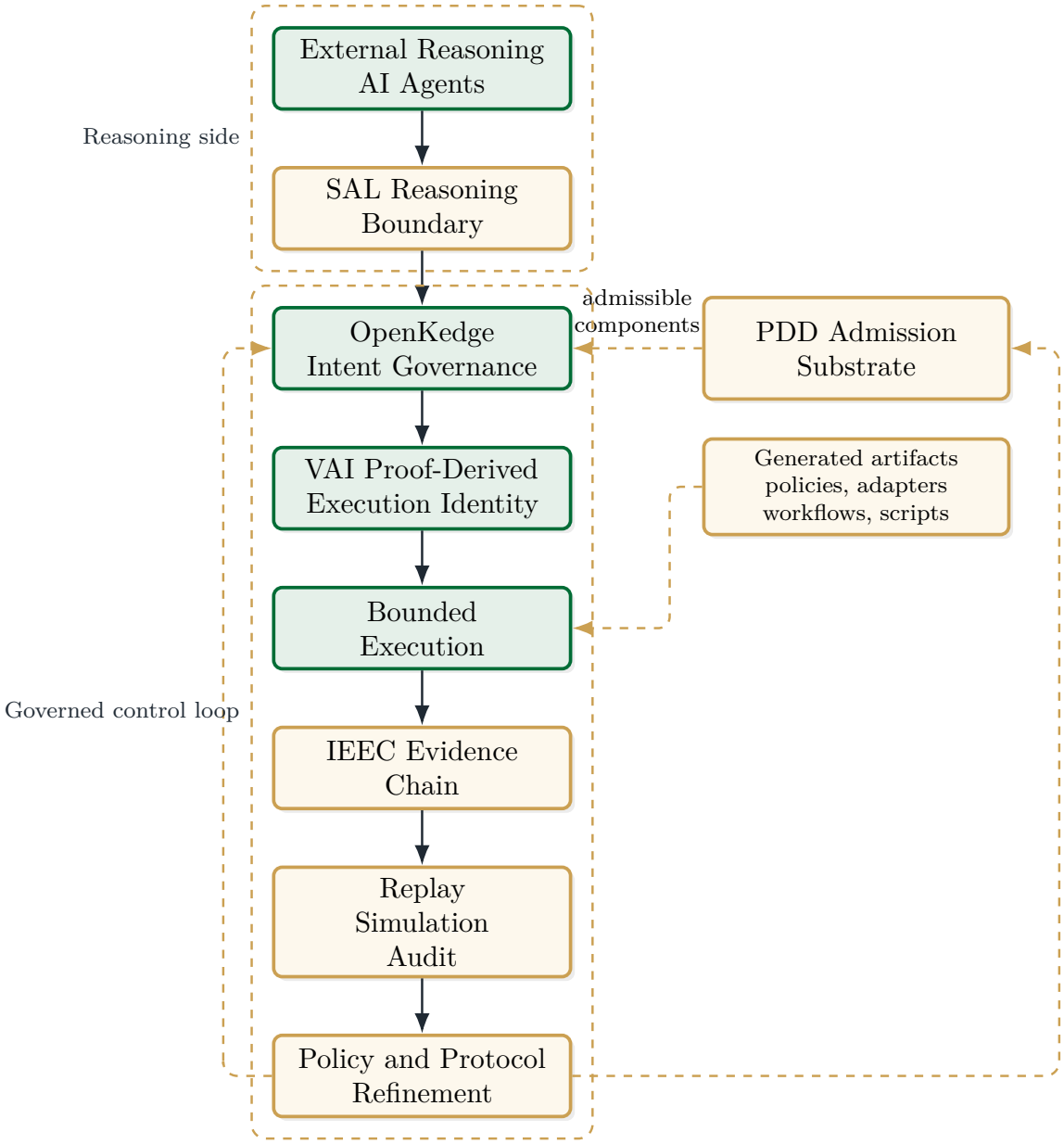


Figure 5.1: *The Autonomous State Control Plane as a closed-loop governance architecture. Probabilistic reasoning is converted into governed intent, bounded execution, evidence, replay, and continuous refinement.*

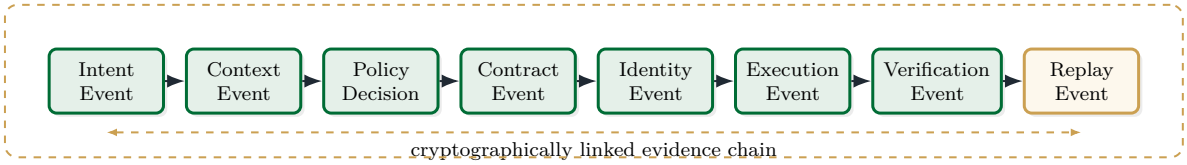


Figure 5.2: *Intent-to-Execution Evidence Chain. Each governed action produces linked evidence from intent through verification and replay.*

6 Sovereign Agentic Loops

The primary boundary of the Autonomous State Control Plane isolates reasoning from execution. Although artificial intelligence agents can generate plans, propose infrastructure modifications, draft administrative decisions, and recommend operational adjustments, capability does not imply authority. Sovereign Agentic Loops (SAL) partition systems so that reasoning stays outside the execution boundary and consequential state changes remain under institutional control.

SAL decouples the locus of reasoning from the seat of authority. Under this architecture, AI agents assist in planning, analysis, summarization, and decision support without possessing execution privileges. The control plane treats model reasoning as advisory input, never as an intrinsic source of authority.

SAL Doctrine

AI reasoning proposes; sovereign governance decides. Execution must remain inside the institutional control boundary.

Sovereign Agentic Loops formalizes this separation in greater detail [8].

6.1 The Reasoning-Execution Boundary

Autonomous control systems divide operationally into two distinct domains. The first, the *reasoning domain*, encompasses model inference, planning, code generation, tool selection, and semantic recommendation. Within this domain, an AI system constructs candidate interpretations of objectives and proposes prospective actions.

The second, the *execution domain*, governs infrastructure mutation, credential issuance, code deployment, financial transactions, and any state-mutating operation. In the execution domain, institutional accountability becomes concrete: resources are provisioned, records updated, permissions granted, and code admitted. SAL prevents the collapse of these domains into a single, unmonitored loop by requiring reasoning to remain external while keeping execution strictly inside the sovereign boundary.

This boundary does not divide humans from machines; it separates proposal from authority. An operator can propose unsafe actions; a deterministic routine can execute in an incorrect scope; a model can generate an optimal plan. Architecturally, the origin of the proposal is secondary. The primary concern is whether the system treats that proposal as advisory or endows it with immediate mutation authority.

This distinction enables institutions to use advanced machine intelligence without permitting models to become the actors of record. While a model may analyze data, draft configurations, or suggest remediations, the sovereign control plane determines admissibility, enforces scope, justifies authority, verifies evidence, and triggers escalation.

Engineers evaluating agent integrations must ask where reasoning stops and institutional authority begins, not only whether an agent can complete a task. Granting a model direct write access to production systems violates this boundary regardless of prompt-level safeguards. Requiring the model to generate structured, non-executable intent that passes through local governance establishes the baseline for sovereign autonomy.

6.2 What Makes an Agentic Loop Sovereign

An agentic loop consists of a recurring cycle: the system observes context, reasons about objectives, proposes actions, receives feedback, and updates its strategy. Traditional agent architectures implement this loop via iterative model calls, direct tool execution, memory retrieval, and self-evaluation.

This loop introduces systemic risk when actions execute directly. When an agent observes state, reasons about a goal, calls privileged APIs, and recursively invokes further tools based on the results, it operates as an autonomous actor. Under this model, any error in reasoning, tool selection, or state interpretation mutates the target system unchecked.

A sovereign agentic loop structurally prevents the reasoning process from directly mutating sovereign systems. Every proposed action must cross an intent boundary for evaluation by a control plane operating under local institutional authority. The loop becomes sovereign only when the institution owns the interface through which reasoning translates into execution.

Sovereignty here is operational, not symbolic. It requires local control over policy, execution authority, evidence collection, audit trails, escalation protocols, and context disclosure. The responsible institution, rather than an external provider, dictates which policies apply, which actors may request mutations, what data may be exposed to the model, and what evidence must be preserved.

The agentic loop remains dynamic, iterative, and capable of utilizing environmental feedback. However, it cannot unilaterally cross the boundary from proposal to execution; that transition requires explicit governance.

This constraint becomes more important when agents maintain memory, planning state, tool-use histories, and self-reflection loops. These internal mechanisms may improve proposal quality, but they complicate inspectability. A sovereign loop treats internal agent state as reasoning telemetry, not authority. Memory may inform a proposal, and planning traces may support a justification, but neither grants execution rights. The control plane derives authority from structured intent, active policy, verified context, and evidence.

6.3 Foreign Reasoning, Local Authority

In this architecture, the term *foreign* denotes any system external to the sovereign execution boundary, rather than a hostile entity.

Foreign Reasoning, Sovereign Execution

Foreign does not mean hostile; it denotes any logic external to the sovereign execution boundary. External reasoning advises; sovereign execution decides.

Foreign reasoning encompasses frontier models hosted by external vendors, third-party agents, cross-border inference services, open-source models running outside the local boundary, or even internal models operating outside a specific department's authority. While these reasoning systems generate valuable outputs, they lack the legal or operational authority to mutate system state.

This framing avoids two architectural extremes: *uncritical dependence*, which permits capable models to mutate infrastructure based on plausible plans, and *strategic isolation*, which rejects external intelligence due to boundary concerns. SAL establishes a third pattern: external reasoning may advise, but local authority must decide. The risk is not the existence of external reasoning, but the delegation of authority to it.

This follows directly from the sovereignty model established in Chapter 3. A nation or enterprise can use external models while maintaining local control over data boundaries, policy, identity, execution, and audit. The control plane does not need to own the model to control execution; it must own the boundary where model output translates into governable intent.

This principle applies equally to internal enterprise systems. A central AI service may serve multiple business units, but remains "foreign" to any specific authority boundary. For example, a corporate LLM is foreign to a financial approval pipeline, a healthcare data enclave, or a production operations cluster. Thus, SAL represents not only a geopolitical tool but an institutional control pattern for any environment requiring shared reasoning under localized authority.

6.4 The Obfuscation Membrane

The *obfuscation membrane* regulates the flow of information into the reasoning domain. This controlled interface translates sovereign operational context into abstracted, task-specific representations before transmitting it to external or non-authoritative models.

The membrane minimizes context disclosure, shielding sensitive operational state, private data, and infrastructure topology. It presents only the minimum context necessary for effective reasoning while retaining complete sovereign control.

The membrane is more than a passive data-redaction filter. Redaction masks specific fields; the obfuscation membrane determines the conceptual representation of the problem exposed to the reasoning layer. For example, it might summarize a dependency graph rather than expose raw network topology, specify a resource class rather than disclose physical identifiers, or present policy-relevant constraints without revealing full internal policy rules. It tokenizes citizen, patient, or system identifiers, exposing the existence of operational constraints without disclosing their sensitive details.

The obfuscation membrane ensures the reasoning layer receives sufficient context to propose solutions, but never enough system-level detail to seize control.

Maintaining this equilibrium is an engineering task. Over-restriction starves the model of necessary context, rendering its proposals generic and ineffective. Excessive disclosure leaks sensitive data and exposes operational details that could facilitate unauthorized control if secondary boundaries fail. A mature SAL implementation manages context exposure as an explicit, policy-driven decision rather than a configuration convenience.

In cloud operations, the membrane discloses that a service is critical and currently under a

change freeze without revealing network topologies or environment credentials. In public-sector workflows, it provides anonymized case attributes and eligibility criteria without exposing citizen identities. In automated software engineering, it provides interface contracts and test failures while withholding unrelated source code and secrets. The pattern remains uniform: the reasoning layer receives a task-shaped abstraction while the underlying sovereign context remains shielded.

The membrane's transformations should be auditable. The control plane should log what context was disclosed, what was withheld, which abstraction rules were applied, and the justification for that representation. This auditability prevents context minimization from becoming an opaque policy bypass and helps trace the root causes of poor model proposals. If a model fails to produce a viable plan due to insufficient context, engineers should refine the membrane's translation rules rather than bypass the boundary.

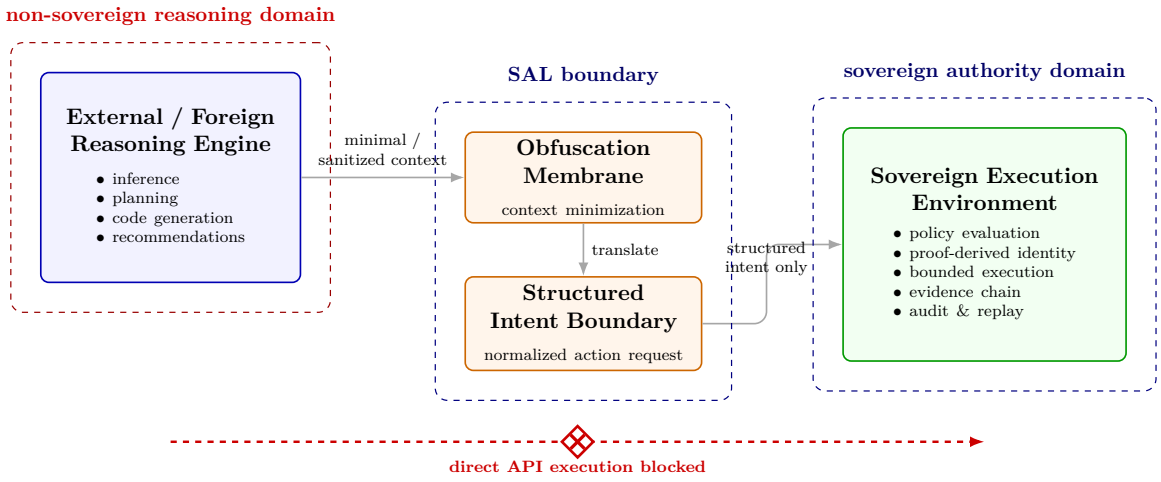


Figure 6.1: *Sovereign Agentic Loop boundary. External reasoning may receive minimized context and return structured intent, but direct execution authority remains inside the sovereign environment.*

6.5 Intent Isolation

SAL prohibits the direct conversion of external reasoning outputs into API calls. Instead, all model outputs must undergo translation into structured intent. This process of *intent isolation* converts plans, recommendations, or tool calls into strongly-typed objects that the control plane can govern.

A valid intent object specifies the requested action, underlying objective, target scope, origin, assumptions, expected blast radius, risk classification, and justification. This format must be machine-readable to allow automated policy evaluation, and human-readable to facilitate manual review. The intent should also contain provenance metadata identifying the model, agent version, user request, and system state that initiated the proposal.

Intent represents the admissible reasoning output allowed to cross the boundary into the governance layer. Raw model text, generated scripts, natural-language rationales, and tool-call payloads

carry no intrinsic authority.

SAL converts model output from a command into a governable claim.

This claim proposes that a specific action is justified under a defined objective and scope. The claim may be valid, over-broad, context-starved, or subject to rejection. The model does not execute actions; it submits proposals for inspection by the control plane.

Intent isolation also mitigates the risk of *tool-chain amplification*, where a minor reasoning error cascades into a destructive sequence of API calls. For example, an unconstrained agent might select the wrong target resource, attempt to adjust its configuration, deploy a patch to resolve the resulting error, and escalate its own permissions to bypass a block. Intent isolation aggregates these proposed operations into a single intent object before execution. The control plane evaluates the proposed sequence, including blast radius, dependencies, and compliance, before a state change occurs.

The intent object must be syntactically narrow yet semantically rich. While a weak intent specifies only a command like "restart service," a strong intent details the target resource, the operational objective, the proposed execution window, expected side-effects, rollback procedures, and the specific evidence to be collected. This structured richness allows the governance layer to verify alignment with user intent, simulate outcomes, and flag high-risk proposals for human escalation.

Intent isolation also supports vendor and model neutrality. Because the control plane mandates a standardized, structured intent format, the underlying models and agent frameworks can be swapped, upgraded, or run in parallel without altering the system's authority boundary.

6.6 Compositional Governance

Governing agentic workflows requires maintaining authority across composite operations, especially when agents delegate tasks, chain tools, spawn sub-agents, or dynamically modify execution paths.

Agentic systems are inherently recursive. A parent loop may invoke a sub-loop, delegate to a specialized model, generate a dynamic workflow, or request a code-generation tool to synthesize a runtime adapter. If the parent loop is governed but its delegated sub-processes bypass inspection, the architecture merely shifts the vulnerability. A sovereign agentic loop must enforce governance invariants across all composite execution paths.

Delegation never automatically transfers authority. Sub-agents must not inherit the parent's broad permissions. Instead, the control plane must issue narrow, task-scoped execution contracts justified by the sub-agent's specific intent, context, active policy, and evidence requirements. Every composite action must cross the intent boundary.

SAL therefore defines a boundary condition: a composed loop is admissible only if the resulting composite workflow preserves sovereign control. Sub-intents must remain explicit, delegated actions require independent contracts, and sub-agents cannot inherit parent privileges. Evidence chains must preserve parent-child provenance so the composite workflow can be replayed, while governance policy must explicitly regulate delegation.

In formal terms, governed actions represent composable state transformations. Category theory provides a useful model for reasoning about these structures: if each constituent transformation

preserves the governance boundary, the composite morphism preserves that boundary as well. This whitepaper does not mandate a category-theoretic implementation. The operational goal is simpler: prevent composition from becoming a policy bypass.

For foundational approaches to these formalizations, see categorical semantics [16], compositional verification [3], and denotational semantics [19].

6.7 Sovereign Execution Environment

Once intent crosses the boundary, execution proceeds exclusively within the sovereign execution environment. This domain governs context verification, policy evaluation, execution contract generation, cryptographic identity derivation, execution adapters, evidence recording, and replay auditing.

While the reasoning system proposes the objective, the sovereign environment determines admissibility. This functional separation lets SAL compose with the broader Autonomous State Control Plane: SAL enforces the reasoning boundary, OpenKedge evaluates the resulting intent, VAI derives runtime authority, the IEEC anchors the decision and execution history, and PDD governs generated code before admission.

The sovereign execution environment is defined by administrative control rather than physical isolation. While on-premises datacenters or sovereign clouds may host the environment, its defining characteristic is authority: the managing institution must control the policy engines, identity providers, execution contracts, and evidence boundaries. Workflows can span public clouds and leverage external inference APIs as long as the execution authority resides within the institutional control plane.

The environment must also enforce escalation paths. High-risk or ambiguous intents should not be resolved through automated heuristics alone. They should route to human operators, simulation sandboxes, or immediate rejection. Sovereign execution includes the authority to deny automation.

Execution adapters must operate under strict constraints. These adapters must not accept unverified instructions directly from the reasoning layer; they should execute only signed contracts issued by the control plane. If an adapter remains callable by the agent runtime directly, the sovereignty boundary has collapsed. A robust implementation makes the governed, policy-checked path the sole approved mechanism for system mutation.

Architecturally, the sovereign execution environment serves as a ”narrow waist.” Reasoning engines, model providers, and agent frameworks may vary and evolve. Target systems, including cloud platforms, databases, and operational APIs, remain heterogeneous. Between them stands a single, invariant gateway: intent enters, governance evaluates, contracts constrain, identity authorizes, and the evidence chain records the outcome.

6.8 Failure Modes Prevented by SAL

SAL provides a governance layer that prevents or mitigates several failure modes that emerge when reasoning and execution collapse.

Authority Confusion

Without a reasoning-execution boundary, institutions risk treating model-generated outputs as de facto authority. SAL prevents this by ensuring model output crosses the boundary strictly as intent, never as an executable command.

Table 6.1: *Failure modes addressed by Sovereign Agentic Loops.*

Failure Mode	Risk	SAL Response
Direct model execution	Model output directly mutates production state	Forces all actions to cross a structured intent boundary
Over-disclosure	Exposes sensitive data or system details unnecessarily	Restricts exposure via the obfuscation membrane
Authority confusion	Obscures whether a model or an institution authorized an action	Retains execution authority within the sovereign control plane
Tool-chain amplification	Cascades minor reasoning errors into multiple destructive API calls	Aggregates proposed operations into single governed intents before execution
Policy bypass	Avoids institutional compliance and safety checks	Enforces deterministic policy evaluation prior to mutation
Audit weakness	Fails to trace execution back to underlying reasoning	Binds intent, decisions, and execution via cryptographic evidence chains
Vendor coupling	Hardcodes governance into specific model providers	Standardizes reasoning interfaces to maintain provider neutrality

The common denominator in these failure modes is the unconstrained translation of output to action. SAL interrupts this translation. It forces the reasoning layer to generate proposals and charges the control plane with evaluating whether those proposals should proceed.

This is a deliberate architectural discipline, not a rejection of AI capabilities. Highly capable models can generate planning outputs superior to traditional scripts. Precisely because these models are powerful enough to be integrated into public services, infrastructure, and regulated systems, they require a boundary that protects execution authority from direct model action.

These failure modes show why prompt-level safety guidelines or basic tool allowlists are insufficient. A prompt instructing an agent to act safely can be bypassed; tool allowlists do not evaluate semantic context; post-hoc logging only records failures after they occur. SAL restructures the system architecture so that reasoning engines cannot mutate state through approved interfaces, reducing whole classes of failure by design.

6.9 How SAL Composes with OpenKedge

SAL provides the architectural boundary that makes formal governance possible.

The handoff is straightforward: SAL creates the boundary, and OpenKedge governs what crosses it.

External Reasoning → Obfuscation Membrane → Structured Intent → OpenKedge Governance → Sovereign Execution

The control-plane lifecycle maps as follows:

External reasoning → obfuscated context → model output → structured intent → OpenKedge policy evaluation → execution contract → VAI execution identity → bounded execution → evidence chain

OpenKedge depends on SAL to supply a structured, governable input. Without SAL, models attempt direct tool invocation. Under SAL, model outputs must undergo normalization into intent objects. OpenKedge then evaluates these objects against active policies, current context, risk classifications, blast radius, evidence rules, and escalation thresholds.

This boundary also supports auditable replay. When investigating past actions, auditors can distinguish reasoning telemetry from structured intent, policy decisions, execution contracts, runtime identities, and execution events. This structural separation prevents accountability from collapsing into the vague claim that the AI made the decision.

This composition also clarifies operational ownership. The model provider remains responsible for model performance within its service boundary, while the institution retains policy and execution authority. OpenKedge owns the governance decision, VAI derives runtime authority, the IEEC secures the evidence chain, and PDD governs code admission. SAL stands as the gatekeeper, preventing direct model output from bypassing this chain.

6.10 Design Requirements

Implementing SAL requires structural constraints that make direct execution physically impossible.

1. **No direct mutation from the reasoning layer.** Models and agents must never call privileged APIs directly. Mutating tools must terminate at the control plane, outside the model's runtime environment.

2. **Structured intent as the boundary artifact.** The system must represent every proposed action as a machine-readable intent object before policy evaluation.

3. **Minimal context disclosure.** The obfuscation membrane must limit context exposure to the minimum required for the task.

4. **Local policy authority.** The control plane must evaluate all policies locally within the sovereign boundary.

5. **Execution identity outside the reasoning layer.** The system must issue cryptographic credentials only after policy approval, never directly to the model.

6. **Evidence binding.** The control plane must bind reasoning outputs, intents, decisions, contracts, and execution events into a single, tamper-evident chain.

7. **Model neutrality.** The boundary interfaces must remain model-agnostic, supporting upgrades, local models, or multi-model architectures without altering the execution path.

8. **Escalation paths.** The system must route high-risk or ambiguous intents to manual review, simulation, or refusal.

These requirements translate sovereignty into engineering practice. They allow agentic reasoning to be used while preventing models from acquiring system authority. They also give platform teams a concrete governability test: map every path from model output to state mutation and verify that each path crosses the intent boundary before execution.

The next chapter develops the governance engine that receives these intents and turns them into policy-evaluated execution contracts: OpenKedge intent governance.

7 OpenKedge Intent Governance

OpenKedge governs the transaction at the boundary between reasoning and execution. It is a protocol for **verifiable intent governance for AI agents**. The protocol ingests structured intent, evaluates it against active policy and runtime context, generates bounded execution contracts, and records the evidence needed to reconstruct why a system mutation was allowed, denied, escalated, or constrained.

OpenKedge replaces direct, unmonitored agent execution with intent-based mutation governance. Rather than checking only whether an agent possesses API access, the protocol evaluates whether a proposed intent should mutate a target system under the active environmental context, what specific authority is justified, and whether the emitted evidence satisfies audit and replay standards. The OpenKedge reference paper develops these protocol specifications in greater technical detail [6].

Intent Governance

OpenKedge transforms autonomous actions from raw API calls into governed, evidenced, and replayable state transitions.

7.1 From API Calls to Intent Governance

Conventional authorization models evaluate operations; OpenKedge governs semantic intents.

Traditional API authorization checks the caller's identity, the requested operation, and the principal's permissions. While necessary, this model is too narrow for autonomous systems. An AI agent can synthesize a syntactically valid operation while operating on stale context, misunderstanding its high-level objective, selecting an incorrect target, exceeding its intended scope, or choosing an action that is technically permitted but operationally damaging.

Syntactic Authorization

A syntactically valid operation can remain semantically unsafe. Traditional authorization determines whether an operation is permitted; intent governance determines whether the proposed mutation should occur under active policy and live context.

Autonomous agency turns access control into a semantic governance problem. The acceptability of an API call depends on the target resource, execution timing, environmental context, operational reversibility, policy constraints, and institutional consequences. A system restricted to traditional permission checks can admit actions that require rejection, modification, simulation, or human escalation.

OpenKedge replaces direct API execution with governed intent evaluation. Agents do not mutate systems by virtue of generating tool calls; instead, they submit proposed state transitions

to the governance layer. The governance engine evaluates these proposals against active policy and live context. After approval, an intent compiles into an execution contract and then into bounded runtime authority.

Agent Proposal → Intent Object → Context Snapshot → Policy Decision → Execution Contract → Evidence Event

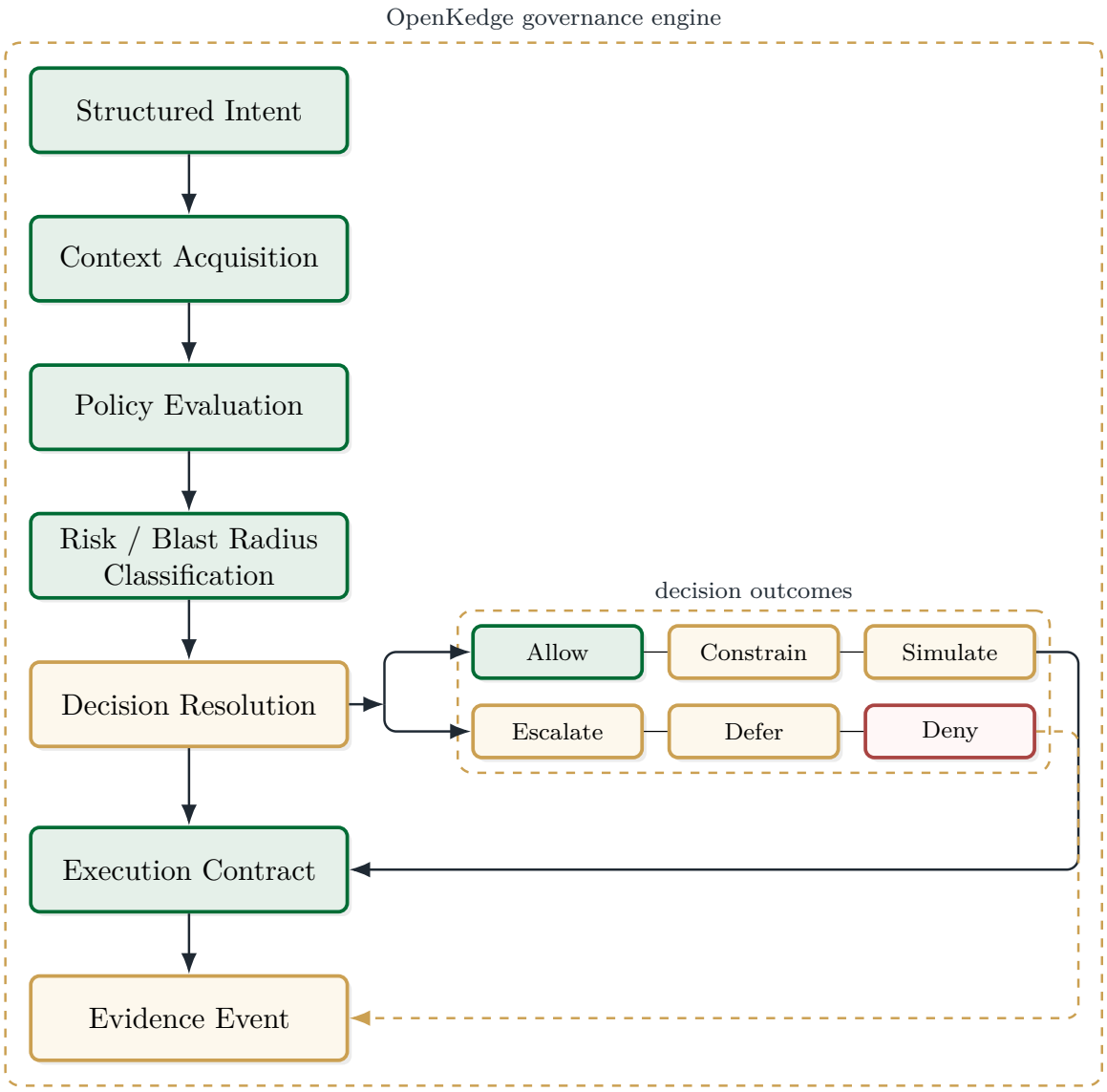


Figure 7.1: *OpenKedge decision lifecycle. A structured intent is evaluated against system context, policy, and risk/blast-radius constraints before being resolved into an explicit decision outcome. Outcomes that permit bounded execution produce an execution contract; outcomes that deny, defer, or require escalation are still recorded as evidence events, preserving auditability even when no mutation occurs.*

The protocol leaves room for agentic reasoning while keeping execution institutionally governed. Agents optimize reasoning and propose actions; OpenKedge validates whether those proposals satisfy the requirements for governed state mutation.

This division of labor provides system architects with clean boundaries. Agent frameworks focus on reasoning, planning, memory, and tool selection. OpenKedge handles the governance surface: intent normalization, context capture, policy evaluation, contract generation, and evidence emission. Conflating these responsibilities compromises accountability. When the same agent that plans also authorizes and executes, the managing institution loses the ability to audit reasoning quality independently from governance quality.

7.2 Neuro-Symbolic Intent Governance

OpenKedge implements a neuro-symbolic governance pipeline: neural reasoning translates ambiguous, high-level objectives into candidate intent, while symbolic governance determines whether that intent qualifies for execution.

This split addresses a practical engineering reality. Large language models are useful for interpreting high-level requests, summarizing operational telemetry, diagnosing system anomalies, and drafting prospective plans. Natural language remains ambiguous and context-dependent, so it is not a reliable basis for execution authority. Natural language may initiate a request, but it should not authorize execution.

Neuro-Symbolic Governance

The neural layer proposes meaning; the symbolic layer grants authority.

Natural Language as Authority

Natural language can initiate a governance request, but it must never authorize execution. High-level objectives must translate into structured intent before policy, context, identity, and evidence layers can process them.

The neural model acts as a translator of intent, not the executor. It may interpret the instruction "restore checkout service health" and generate a candidate intent to shift traffic, restart a component, or deploy a previous stable release. This candidate proposal is input for the governance engine, not an authorized command.

Natural Language Objective → LLM Translation → Structured Intent → Symbolic Policy Evaluation → Execution Contract → Bounded Execution

OpenKedge serves as the symbolic arbiter between probabilistic reasoning and deterministic execution. It exposes a structured evaluation surface: intent schemas, context snapshots, policy versions, risk classifiers, execution contracts, and evidence trails. Symbolic governance does not require formalizing every operational judgment in code; it requires machine-enforceable policies to apply consistently, while routing ambiguity to escalation, simulation, or denial.

This neuro-symbolic separation preserves the utility of advanced reasoning while keeping authority within the managing institution. The model interprets and proposes; OpenKedge checks that the proposal is well-formed, contextually appropriate, policy-compliant, and evidenced.

For foundational research integrating neural and symbolic models, see [5]. For symbolic policy systems, refer to industry-standard engines like Cedar [2] and OPA [15].

7.3 The Intent Object

An intent represents a structured, machine-evaluable specification of a proposed state transition.

As the boundary artifact between reasoning and governance, the intent object enables the control plane to inspect, normalize, enrich, constrain, approve, and replay model proposals. Without this structured substrate, the system is left with raw natural-language outputs, tool-call fragments, or generated scripts: artifacts useful for reasoning but inadequate for deterministic governance.

A valid intent object specifies the requested action, underlying objective, target scope, origin, assumptions, expected blast radius, risk classification, and justification. This format must be machine-readable to allow automated policy evaluation, and human-readable to facilitate manual review. It also needs provenance metadata identifying the model, agent version, user request, and system state that initiated the proposal.

OpenKedge converts model output from a command into a governable claim.

Listing 7.1: *Illustrative intent object.*

```
{
  "intent_id": "intent-123",
  "actor": "ops-agent",
  "reasoning_source": "incident-response-agent",
  "objective": "restore service health",
  "requested_action": "shift_traffic",
  "target": "service.checkout",
  "scope": {
    "region": "us-west-2",
    "percentage": 20
  },
  "assumptions": [
    "error_rate_above_threshold",
    "secondary_region_healthy"
  ],
  "justification": "Reduce elevated 5xx rate by shifting traffic.",
  "risk_class": "medium"
}
```

The intent object captures both the proposed action and its underlying rationale. This semantic detail matters. A request to shift traffic due to service degradation demands different policy constraints than a traffic shift for cost optimization. Similarly, updating an access policy during a break-glass incident requires distinct validation compared to routine maintenance. Though the technical operations are identical, the semantic intent and active context differ.

Intent isolation also supports vendor and model neutrality. Because the control plane mandates a standardized, structured intent format, the underlying models and agent frameworks can be swapped, upgraded, or run in parallel without altering the system's authority boundary.

Production environments must reject malformed or incomplete intents prior to policy evaluation. Missing actor identities, ambiguous targets, or unsupported actions should not reach execution. Schema validation is the first line of defense in the governance pipeline.

7.4 Context Acquisition

Evaluating an intent in isolation is incomplete; environmental context determines whether the action is acceptable.

This context encompasses infrastructure telemetry, dependency topologies, incident logs, active change windows, compliance parameters, resource metadata, active policies, historical intents, and real-time risk indicators. For instance, cloud operations require dependency maps and service health; administrative workflows require approval thresholds and case status; and automated software deployments require test coverage metrics and protocol invariants.

The operational implications are straightforward. Terminating a virtual machine is acceptable if the instance has been removed from rotation and healthy replacements exist; it is not acceptable if the machine actively serves production traffic or belongs to a protected capacity group. Shifting network traffic is acceptable if the target destination is healthy; it is not acceptable if that destination is currently degraded or under active investigation.

Context acquisition must be tightly bounded and auditable. The governance engine gathers only the context attributes required for the active decision. It records a precise snapshot of this context at decision time, relying on trusted, verifiable context providers. The protocol treats stale context as an immediate trigger to deny, constrain, refresh, or defer execution.

Bounded context protects both sovereignty and operational safety. Under-collecting context leads to the approval of semantically unsafe mutations, while over-collecting leaks sensitive data and introduces operational noise. OpenKedge defines explicit context requirements mapped to intent classes, target resource domains, and risk classifications.

This context snapshot anchors the evidence chain, letting auditors reconstruct exactly what the system knew at decision time. If a decision fails due to stale or contradictory telemetry, the evidence chain exposes the root cause.

7.5 Policy Evaluation

OpenKedge evaluates intents against machine-enforceable policies. These policies incorporate security rules, operational safety constraints, compliance mandates, approval workflows, change windows, and service ownership metadata.

The protocol does not require institutions to replace their existing policy systems. Instead, it provides the structured intent and context substrates that enable these systems to govern autonomous actions.

Architecturally, OpenKedge is policy-engine-pluggable. It prepares policy inputs, invokes external policy engines (e.g., Cedar for role authorization, OPA/Rego for operational safety, custom engines for domain-specific constraints), interprets the decisions, and generates execution contracts.

Policy evaluation must support multi-modal outcomes rather than binary gates. Real-world operations demand partial approvals, narrowed scopes, time-bounded authorizations, simulated executions, or human escalations.

All policy inputs must be versioned and replay-compatible. The system logs the exact policy version evaluated, the context supplied, the intent fields analyzed, and the resulting decision. This

version-locked audit trail supports deterministic replay and allows operators to simulate historical workloads against updated policies.

Deterministic policy application also satisfies a key sovereignty requirement: policy changes must be explicit, versioned, and auditable. In regulated environments, explaining why a policy allowed or blocked an action is as critical as the enforcement itself.

7.6 Blast Radius and Risk Classification

Operational risk is a function of action, context, target criticality, reversibility, and institutional consequence.

The risk of a mutation cannot be inferred from the API call alone. It depends on target criticality, user impact, dependency fanout, data sensitivity, financial exposure, and regulatory implications. A read-only diagnostic command may be routine in a development environment but highly sensitive on a production database. A configuration change might affect a single sandbox environment or disrupt a national-scale service.

Evaluating blast radius determines whether an action can execute, whether it requires sandboxed simulation, how narrow its identity scope must be, and what evidence must be captured. High-risk actions demand shorter execution windows, pre-verification, and explicit escalation paths.

Table 7.1: *Operational risk factors in intent governance.*

Risk Factor	Governance Implication
Target criticality	High-impact targets require human approval or sandbox simulation.
Reversibility	Irreversible actions require strong evidence and constraints.
Dependency fanout	Cascading downstream dependencies increase blast radius and restrict automation.
Data sensitivity	Access to sensitive data environments triggers advanced policy constraints.
Timing	Operations during change freezes, peak loads, or active incidents require escalation.
Regulatory impact	Regulated workflows demand formal evidence retention and manual verification.

Risk classification serves as a primary input to policy, not an arbitrary label. A medium-risk intent may execute under automated constraints, whereas a high-risk proposal requires simulation and operator sign-off. The control plane logs this classification in the evidence chain, ensuring auditors can reconstruct the rigor applied to each decision.

Risk classifications must remain dynamically recalculable. If the state of the target system mutates prior to execution, the risk profile changes. A contract issued under low-risk assumptions should not authorize execution in a high-risk state; hence, contracts incorporate explicit expiration bounds and revocation conditions.

7.7 Execution Contracts

An execution contract represents a bounded, machine-enforceable specification defining the precise operational limits of an approved intent.

The contract contains the approved operation, target resource identifiers, permitted parameters, time bounds, context assumptions, policy references, derived identity scope, maximum blast radius, rollback procedures, and evidence requirements. This schema provides the execution and identity layers with deterministic boundaries.

The execution contract serves as the invariant bridge between governance and runtime authority.

Listing 7.2: *Illustrative execution contract.*

```
{
  "contract_id": "contract-456",
  "intent_id": "intent-123",
  "decision": "allow_with_constraints",
  "approved_action": "shift_traffic",
  "target": "service.checkout",
  "constraints": {
    "max_percentage": 20,
    "region": "us-west-2",
    "expires_in": "10m"
  },
  "required_evidence": [
    "pre_context_snapshot",
    "policy_decision",
    "execution_result",
    "post_verification"
  ]
}
```

Rather than relying on standing privileges, VAI derives execution identity directly from this contract. A precise contract should produce a restricted, short-lived runtime authority mapped to the approved intent. If a contract authorizes a 20% traffic shift for ten minutes in a single region, the execution adapter must prevent a 50% shift or operation in an adjacent region.

This model also supports audit replay. If an execution exceeds its contract boundaries, the system flags an enforcement failure; if the execution stays within bounds but causes an outage, the system flags a governance failure. The evidence chain helps isolate these distinct failure modes.

Execution contracts must be treated as first-class, cryptographically signed artifacts. They maintain dedicated identifiers, lifecycle states, context assertions, and expiration boundaries so policy decisions govern runtime adapters with less translation risk.

7.8 Decision Outcomes

OpenKedge normalizes policy decisions into explicit, machine-readable outcomes, avoiding brittle binary models.

Table 7.2: *OpenKedge governance decision outcomes.*

Outcome	Meaning	Example Application
Allow	Authorizes intent to execute as proposed	Low-risk, policy-compliant resource change
Deny	Rejects intent immediately	Direct policy violation or unsafe target state
Constrain	Restricts execution to narrow bounds	Restricts a 50% traffic shift proposal to 10%
Escalate	Demands manual or institutional approval	High-impact administrative or production mutation
Simulate	Mandates pre-execution sandbox validation	Complex infrastructure changes with unknown dependencies
Request context	Demands additional operational telemetry	Missing dependency health metrics or resource ownership metadata
Defer	Postpones execution to a later window	Intent proposed during active change freezes or peak loads

An *Allow* outcome authorizes the intent to proceed as specified. A *Deny* outcome halts execution immediately due to policy violations, unsafe context, or insufficient authorization. A *Constrain* outcome restricts the execution to a safer subset of parameters. An *Escalate* outcome routes the intent to human operators. A *Simulate* outcome evaluates the change in a dry-run environment. A *Request Context* outcome pauses the pipeline until necessary telemetry is acquired, while a *Defer* outcome postpones execution to a safer operational window.

Downstream execution systems must ingest these outcomes directly as machine-readable inputs rather than attempting to parse natural-language rationales.

7.9 Evidence Emission

Every OpenKedge decision must emit structured evidence, including proposals that do not proceed to execution.

Denied, escalated, and constrained actions represent critical telemetry. The control plane logs intent ingestion, context snapshots, policy evaluations, risk classifications, contract generation, approvals, and rejections. These events form the pre-execution segments of the IEEC.

Recording successful executions is insufficient. Denials demonstrate where policy successfully prevented unsafe operations; escalations highlight where automated heuristics deferred to human judgment; and constraints show where the system narrowed blast radius.

This evidence trail supports auditing, compliance, replay, policy debugging, and incident forensics. It allows policy authors to test proposed rules against historical intent logs and enables operators to isolate whether an incident stemmed from agent error, telemetry failure, policy logic, or execution adapter collapse.

Evidence emission represents a core protocol invariant. If the system cannot reconstruct why it permitted a mutation, it lacks verifiable governance.

This telemetry must remain operationally visible. Operators must be able to inspect why an action is stalled in escalation, why a proposal was constrained, or why an agent is generating repeated denials. This visibility helps teams refine policies, improve agent prompting, and identify recurring anomalous proposal patterns.

7.10 OpenKedge as a Policy-Engine-Pluggable Layer

OpenKedge operates as an open, composable, and policy-neutral protocol. It does not introduce a proprietary policy language as a mandatory dependency.

Instead, OpenKedge normalizes intents, orchestrates context acquisition, structures the evaluation inputs, invokes pluggable engines, interprets the decisions, and records evidence. This allows institutions to use their existing policy investments, including Cedar, OPA/Rego, custom enterprise approval systems, or cloud-native controls.

The protocol does not replace policy engines; it provides the intent and context substrates that make autonomous intent governable by them.

Policy Sovereignty

Institutions must govern autonomous agents using their own policy engines and approval workflows. OpenKedge provides the structured intent, context, contract, and evidence substrates required to operationalize this governance.

This approach addresses sovereign and enterprise constraints. A national agency requires localized policy authority and formal approval workflows; a regulated enterprise requires separation of duties and compliance controls. OpenKedge integrates directly with these pre-existing control layers.

Pluggability also mitigates vendor lock-in. Models, cloud platforms, and policy engines can evolve or change while the core governance lifecycle remains stable.

To support this composition, OpenKedge defines stable, standardized adapter interfaces for policy inputs and outputs, ensuring all pluggable engines map to normalized decision outcomes and log standardized evidence.

7.11 Cloud Infrastructure Example

Consider an AI operations agent proposing to terminate a degraded compute instance. In a naive direct-execution architecture, the transaction executes as:

$$\text{agent} \rightarrow \texttt{ec2:TerminateInstances}$$

This path bypasses governance. Although the agent possesses the API credentials to call the command, the system has not evaluated whether the operation is safe.

Under the OpenKedge protocol, the transaction executes as:

$$\text{agent output} \rightarrow \text{structured intent} \rightarrow \text{context acquisition} \rightarrow \text{policy evaluation} \rightarrow \text{execution contract} \rightarrow \text{proof-derived execution identity} \rightarrow \text{bounded execution} \rightarrow \text{evidence}$$

The acceptability of the instance termination depends on active context.

The governance engine evaluates whether the instance is in production, whether it sits behind an active load balancer, whether it actively serves traffic, whether healthy replicas exist, whether a change freeze is in effect, and whether the mutation is reversible.

Based on these context attributes, OpenKedge may: allow the termination if healthy replicas exist; constrain the execution by requiring the instance to be detached from the load balancer first; escalate to an operator if the target is a critical database node; or deny the request if replica capacity is depleted.

This pattern applies equally to database failovers, firewall policy modifications, service deployments, queue draining, and budget adjustments. The technical APIs vary, but OpenKedge governs the underlying proposed mutation.

7.12 Design Requirements

OpenKedge implementations must satisfy several architectural requirements.

1. **Structured intent ingestion.** The protocol must ingest machine-evaluable intent objects rather than raw model tool calls.
2. **Context provider abstraction.** Telemetry must originate from trusted, auditable context providers and be strictly scoped to the active decision.
3. **Policy-engine pluggability.** The governance layer must support diverse engines, including Cedar, OPA/Rego, and enterprise approval systems.
4. **Risk and blast-radius classification.** The system must dynamically classify risk based on action, target, context, reversibility, and consequences.
5. **Execution contract generation.** Approved intents must compile into bounded contracts defining the limits of permissible execution.
6. **Decision outcome normalization.** Decisions must map to normalized, machine-readable outcomes: allow, deny, constrain, escalate, simulate, request context, and defer.
7. **Evidence event emission.** Every evaluation path must emit cryptographic evidence to the ledger.
8. **Replay compatibility.** The system must preserve intents, context snapshots, policy versions, decisions, and contracts to support deterministic replay.
9. **Escalation paths.** Ambiguous or high-risk intents must escalate to human review or dry-run simulation rather than forcing automation.
10. **Cloud and domain adapter architecture.** The protocol must employ adapter patterns to operate across diverse clouds, private datacenters, and public-sector workflows.

These requirements establish OpenKedge as the intent-governance engine of the Autonomous State Control Plane. SAL moves reasoning across the boundary as structured intent; OpenKedge determines admissibility. The next chapter explains how approved execution contracts compile into bounded runtime authority through VAI.

8 Verifiable Agentic Infrastructure

While OpenKedge determines whether an intent is admissible, Verifiable Agentic Infrastructure (VAI) governs the creation and scoping of authority once approved. In conventional environments, identity remains a static property of a user, service account, or workload. In agentic systems, this assumption fails. The primary security question shifts from who is calling to what validated intent justifies this specific authority at this exact moment.

Verifiable Agentic Infrastructure converts authorization from static privilege into proof-derived execution identity. In autonomous systems, privilege must result from verifiable proof rather than standing entitlement. VAI formalizes this proof-derived execution identity model in greater technical depth [9].

8.1 From Static Identity to Execution Identity

Traditional identity systems answer foundational questions: who is the caller, what role does the caller hold, what permissions are attached, and whether the principal is authenticated. These checks remain necessary; autonomous architectures still require authenticated actors, workloads, services, and execution environments.

However, they are no longer sufficient. Autonomous governance must also determine what intent was approved, which policy decision authorized the action, what context supported that decision, what contract bounds the execution, what authority is necessary, when that authority expires, and what evidence must be emitted.

Consequently, the unit of runtime trust shifts from principal identity to execution identity.

An execution identity is a task-scoped, time-bounded runtime authority derived from a validated intent, policy decision, context snapshot, and execution contract. It is not an ambient property of an agent; rather, it is a transient authority artifact created for a specific governed action under defined conditions.

This shift fundamentally alters the trust model. A conventional service account is trusted because it belongs to a pre-authorized service; an execution identity is trusted because the system can reconstruct the explicit reasoning that justified its issuance. The authority links directly to the approved task rather than inheriting ambient permissions from a broad role. The security question shifts from "Does this agent have access?" to "What proof justifies this authority now?"

$$\text{Approved Intent} \rightarrow \text{Execution Contract} \rightarrow \text{Proof Validation} \rightarrow \text{Execution Identity} \rightarrow \text{Bounded Runtime Use} \rightarrow \text{Evidence}$$

This does not render principal identity irrelevant; the initiating actor still matters. A request from a human operator, an incident-response agent, a workflow system, or a deployment service carries distinct institutional weight. However, principal identity becomes merely one input to a richer authorization decision. The execution identity is the actual authority used for mutation, and

it must remain narrower than the principal's general capability. In this model, a principal may be permitted to request a class of actions, while each execution receives only the task-specific authority that the validated contract justifies.

8.2 Why Standing Privilege Fails for Agents

Standing privilege grants a principal reusable, persistent authority that extends beyond a single task. While manageable in deterministic systems running stable software, standing privilege introduces severe vulnerabilities in autonomous agentic networks.

Static credentials define ambient capabilities in general; they cannot justify specific operations in real-time.

Standing privilege is persistent, reusable, detached from specific intent, difficult to bind to operational justification, vulnerable to tool-chain amplification, and highly dangerous when controlled by probabilistic agents. Even a least-privilege role remains too broad if it is not bound to intent, context, and time. A role that can restart services may be reasonable for a human operations team, but too broad for an agent responding to a single incident. A deployment credential may be reasonable for a CI/CD pipeline, but too broad for a code-generation agent. A workflow approval role may be valid for a human official, but unsafe for an autonomous process that interprets ambiguous instructions.

Standing Privilege

Standing privilege is dangerous for autonomous systems because it grants reusable authority to probabilistic reasoning processes. VAI replaces standing entitlement with task-scoped authority derived from validated intent and execution evidence.

The problem is not that durable identities are inherently wrong; existing infrastructure depends on them. The problem is that autonomous execution must never inherit broad ambient authority simply because the agent is attached to a privileged principal. When a model can generate plans, chain tools, write code, and react to feedback, broad reusable credentials can amplify minor reasoning errors into catastrophic operational effects.

VAI does not eliminate identity management; it changes the derivation of runtime authority. The agent may have an identity sufficient to submit a request, but the authority to execute is minted only after the control plane validates the intent, policy decision, context, contract, and time bounds.

This distinction is vital for multi-step agent systems. A planning agent may ask for several operations in sequence. If it holds standing credentials, every step inherits the full permission set, even when later steps depend on assumptions from earlier ones. With proof-derived execution identity, each consequential step requires its own contract or a contract with explicit staged constraints. The system can stop, refresh context, request approval, or revoke authority between steps.

Standing privilege also weakens accountability. When a long-lived credential is used, an auditor may determine which principal acted, but not why that specific authority was justified at that moment. The result is a gap between access logs and institutional justification. VAI closes that gap by making the proof behind authority part of the identity lifecycle.

8.3 Proof-Derived Execution Identity

Proof-derived execution identity forms the core of VAI. Runtime authority derives from evidence that a specific intent was approved under specific conditions. The proof binds the intent, context, policy decision, execution contract, and time bounds into the identity issuance decision.

Proof-Derived Execution Identity

An autonomous system must receive only the authority proven necessary for a validated intent, only for the time required, and only within the constraints of the execution contract.

The relationship can be summarized as:

$$EID = f(I, C, D, K, T)$$

where I is the validated intent, C is the context snapshot, D is the governance decision, K is the execution contract, and T represents the relevant time bounds.

This formula does not prescribe one cryptographic construction; it expresses the architectural dependency: execution identity must not exist independently of governance evidence. An execution identity is valid only because the system can reconstruct the proof that justified its creation.

The resulting identity must be task-scoped, time-bounded, contract-bound, least-privilege, evidence-linked, revocable, non-transferable where possible, and auditable. It may be implemented as a short-lived credential, scoped token, capability token, workload identity, signed claim, session policy, or other runtime authority primitive. The implementation may vary; the invariant is that the authority is justified by proof and constrained by contract.

This distinction matters for audit and trust. A token by itself proves possession; a proof-derived execution identity proves that authority was minted because a governed decision authorized a bounded action. The difference is the connection to institutional evidence.

Proof-derived execution identity can be implemented with stronger or weaker technical guarantees depending on the environment. Some deployments may use signed tokens, cryptographic attestations, or hardware-backed workload identity. Others may begin with short-lived scoped credentials and strongly audited brokers. The architecture should not overclaim cryptographic properties that an implementation does not provide. What matters at the reference-architecture level is that authority issuance is linked to reconstructable proof and that enforcement points can verify the scope they are expected to enforce.

The proof must also be checked for consistency. The identity broker must reject a request if the actor does not match the approved contract, if the target resource differs, if the contract has expired, if the policy decision has been superseded, or if required evidence is missing. Proof validation is therefore an active control, not a database lookup.

8.4 The Execution Identity Lifecycle

Execution identity has a lifecycle. It begins after intent governance and ends when authority expires, is revoked, or is no longer needed. Treating this as a lifecycle prevents credentials from silently

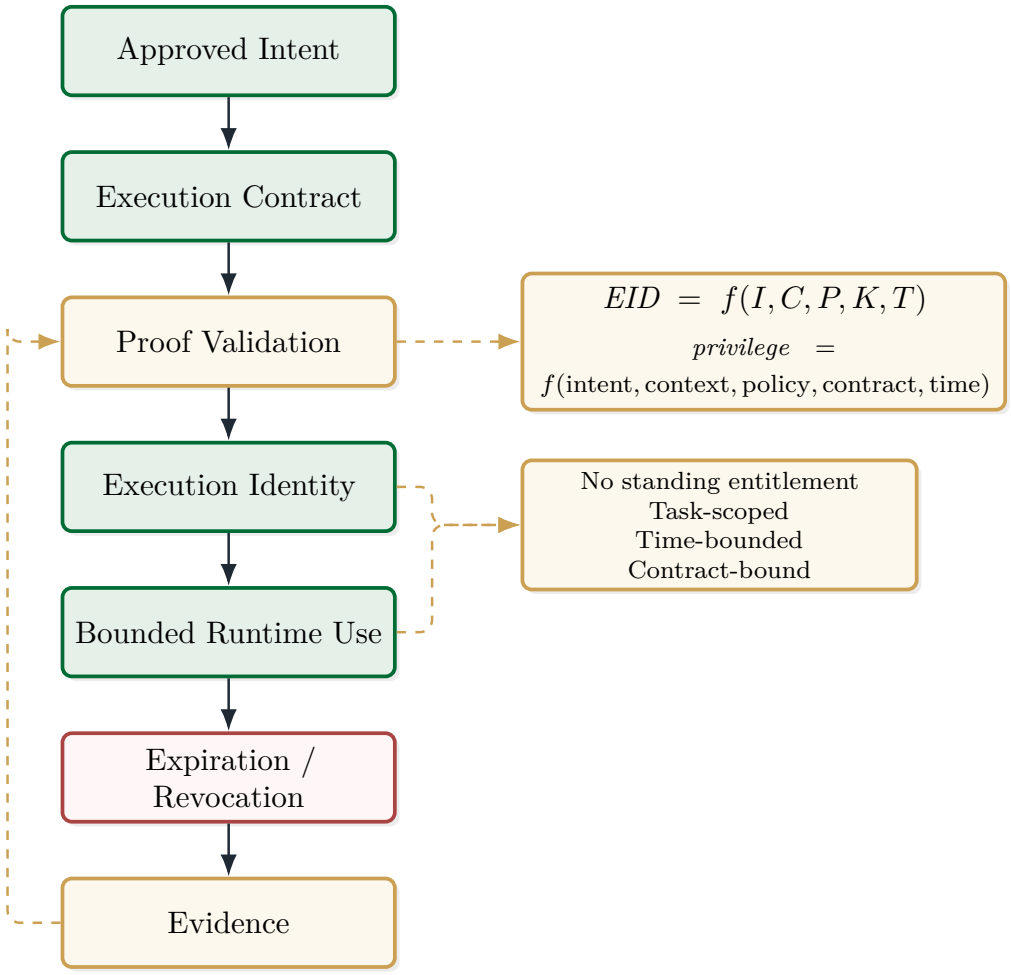


Figure 8.1: *Proof-derived execution identity lifecycle. Runtime authority is derived from validated intent, context, policy, execution contract, and time bounds rather than from standing privilege.*

becoming durable privilege.

Table 8.1: *Lifecycle of proof-derived execution identity.*

Stage	Purpose	Evidence Produced
Intent approval	Establish that the proposed action is allowed	Intent and policy decision record
Contract binding	Constrain the approved action	Execution contract
Identity request	Request runtime authority for the contract	Identity request event
Proof validation	Verify that the request matches approved evidence	Proof validation event
Issuance	Create task-scoped authority	Credential or token issuance event
Runtime use	Execute within bounded authority	Execution event
Expiration or revocation	End authority after task, timeout, or policy change	Expiration or revocation event
Replay and audit	Reconstruct why authority existed	Evidence-chain reconstruction

The lifecycle starts with intent approval. OpenKedge has evaluated the proposed action and produced a decision. If execution is allowed or constrained, the approved intent is bound into an execution contract. The identity broker then receives a request for runtime authority. That request must reference the approved evidence rather than merely asserting that a task is authorized.

Proof validation checks that the request matches the approved contract, that the contract has not expired, that the relevant policy decision is still valid, and that required context assumptions have not been invalidated. Only then is a credential or token issued. Runtime use then occurs through enforcement points that can verify the authority. Finally, the identity expires or is revoked, and the evidence chain preserves enough information for replay and audit.

This lifecycle must be visible to operators. A platform team must be able to see why a token was issued, what contract it was bound to, what it was allowed to do, whether it was used, when it expired, and whether any enforcement failures occurred.

Visibility is not only for auditors; it is also for operational safety. During an incident, responders need to know which autonomous actions are pending, which identities are live, which contracts are about to expire, and which actions were blocked. Without this view, proof-derived execution identity becomes another hidden control path. A mature VAI deployment must expose lifecycle state in a way that operators, security teams, and governance reviewers can understand.

The lifecycle must also support idempotency and uniqueness. If an agent retries an identity request, the system determines whether it is the same contract-bound execution or a new request requiring fresh validation. This prevents accidental duplicate authority and helps distinguish retries from replay attempts.

8.5 Binding Identity to Execution Contracts

The execution contract is the artifact that tells the runtime what is allowed. It defines the allowed action, target resources, permitted parameters, time window, maximum scope, context assumptions,

evidence obligations, and revocation conditions. Execution identity must encode or enforce these constraints.

The execution contract describes what is allowed; execution identity makes only that authority usable.

Execution Authority

Models may produce reasoning, but execution authority must be issued by the institution that owns the policy, context, contract, and evidence boundary.

Binding identity to the contract can happen in several ways. Short-lived credentials can be scoped to a narrow set of actions and resources. Scoped tokens can carry signed claims about target, operation, parameters, expiration, and contract id. Session policies can restrict a cloud credential to the approved operation. Capability tokens can encode a bounded right to perform one action. Workload identity federation can mint temporary workload authority after proof validation. Adapter-side enforcement can require a valid contract before calling downstream APIs. Policy decision points can verify contract claims at runtime.

In AWS-style environments, this may map to temporary STS credentials constrained by session policies and validated by an execution adapter. In Kubernetes-style environments, it may map to short-lived service account tokens, admission controls, and workload identity. In workflow systems, it may map to scoped workflow tokens and step-level authorization. These examples are implementation patterns, not requirements.

The architectural requirement is that the identity cannot be broader than the contract. If the contract authorizes a 10% traffic shift for a specific service in one region for ten minutes, the identity must not allow a 100% shift, another service, another region, or use after the window expires. If the contract authorizes a deployment to staging, the identity must not allow production changes.

Some environments cannot encode every contract constraint directly into the credential. In those cases, the contract must be enforced by another trusted component, such as an execution adapter, gateway, admission controller, workflow engine, or policy decision point. This is acceptable if the enforcement path is mandatory. It is not acceptable if the agent can bypass the adapter and use a broader credential directly. Contract binding is an end-to-end property, not merely a token format.

The contract must also bind evidence obligations. If post-execution verification is required, the runtime path must emit the corresponding evidence or mark the execution incomplete. If a rollback condition is triggered, the runtime must produce that event as evidence. This ensures that execution identity is connected not only to the permission to act, but to the responsibility to report what happened.

8.6 Runtime Enforcement

Execution identity matters only if runtime systems enforce it. Enforcement may occur at the identity broker, execution adapter, cloud API credential boundary, policy decision point, service mesh or proxy, workflow engine, deployment system, database gateway, code execution sandbox, or other control point close to the mutation.

Runtime enforcement must reject actions that do not match the approved contract, even if the agent attempts them.

The runtime enforces the contract, not the agent's memory of the contract.

An agent approved to shift 10% of traffic cannot shift 100%. An agent approved for one resource cannot mutate another. A contract that expired cannot be executed. A token issued for remediation cannot be reused for unrelated changes. A credential generated for one workflow step cannot be replayed into another step.

This independence from agent compliance is essential. The model may misremember the contract. The agent framework may attempt an extra tool call. A generated script may contain broader operations than intended. A human operator may accidentally pass the token to the wrong adapter. Runtime enforcement must not rely on good behavior from the reasoning layer; it must verify contract-bound authority at the point of use.

Enforcement must also emit evidence. Allowing an action, rejecting an out-of-contract attempt, detecting token reuse, or observing an expired contract are all evidence events. These events help distinguish agent failures from enforcement failures and governance failures.

The placement of enforcement points must match the consequence of the action. Low-risk internal operations may be adequately protected by an execution adapter and short-lived token. High-impact cloud, financial, public-sector, or safety-related operations may require multiple enforcement points: identity broker validation, adapter validation, target-system policy checks, and post-execution verification. The architecture allows layered enforcement without requiring every deployment to start at the highest assurance level.

Runtime enforcement also protects against drift between approval and action. A system may approve an action when capacity is healthy, but capacity may degrade before the token is used. A contract may be issued during an incident, but the incident state may change. Depending on risk class, the enforcement point may need to re-check selected context assumptions before execution.

8.7 Evidence and Justification

VAI produces evidence, not just credentials. Identity issuance, use, expiration, and revocation are part of the evidence chain.

A credential without evidence is a capability. A credential with reconstructable proof becomes accountable authority.

Evidence must include the identity request, referenced intent, referenced policy decision, referenced execution contract, proof validation result, issued authority scope, expiration time, runtime use, enforcement decisions, and revocation events. This evidence allows the institution to reconstruct not merely that a credential existed, but why it existed.

VAI contributes the identity and runtime segments of the IEEC. OpenKedge emits evidence that intent was evaluated and a contract was produced. VAI emits evidence that authority was requested, proof was validated, identity was issued, runtime use occurred, and authority ended. Together, those records connect proposed intent to actual execution.

This supports audit, replay, incident investigation, compliance review, non-repudiation where appropriate, and trust in autonomous execution. If an incident occurs, reviewers can determine

whether the wrong authority was issued, the authority was misused, the enforcement point failed, the contract was too broad, or the original governance decision was flawed.

Evidence must be integrity-aware. The system must protect identity evidence from casual modification and preserve enough metadata to support replay. That does not require every deployment to use the same ledger, signature scheme, or evidence store. It requires that authority be reconstructable from durable records.

This evidence must include negative events. If an identity request is denied because the proof is stale, that denial is evidence. If an execution token is rejected because it is out of scope, that rejection is evidence. If a revocation request is issued, acknowledged, or fails, those events are evidence. Negative evidence shows that the system is enforcing governance, not merely recording successful actions.

Evidence also enables policy improvement. If many identity requests fail because contracts are underspecified, OpenKedge contract generation may need refinement. If many tokens expire before use, time windows may be too short or agents may be too slow. If runtime violations cluster around a certain adapter, the adapter may need stronger validation. VAI evidence is therefore both accountability material and operational feedback.

8.8 Revocation, Expiration, and Failure Handling

Expiration and revocation are first-class requirements for autonomous authority. Authority that cannot expire or be revoked is standing privilege by another name.

Autonomous authority must end when the time window expires, the intent is completed, context assumptions change, policy changes, risk signals change, execution fails, a human operator revokes it, evidence emission fails, or a contract violation is detected. The system should not assume that a credential remains valid just because it has not reached a long-lived IAM expiration date.

Failure handling must be conservative. If context becomes stale before execution, the runtime must pause, request refresh, or deny. If the evidence store is unavailable, high-risk actions must fail closed. If runtime use exceeds the contract, the system must reject the attempt and emit a violation event. If identity issuance fails, the system must not fall back to durable standing credentials. If revocation fails, the system must escalate and quarantine affected pathways where possible.

Revocation also needs to propagate to enforcement points. It is not enough for an identity broker to mark a token revoked if the execution adapter, gateway, or target system cannot observe that state. Practical designs may use short expiration windows to reduce reliance on active revocation, but high-risk actions still require explicit revocation behavior.

Failure handling must be visible in evidence. A failed issuance, expired token, denied runtime use, or failed revocation can be as important as a successful execution. These records show where the control plane protected the system and where operational gaps remain.

High-risk systems must prefer fail-closed behavior. If proof cannot be validated, identity must not be issued. If the evidence store cannot accept required events, execution must not proceed unless an explicit break-glass policy authorizes it. If an agent attempts to reuse authority outside its contract, the system must treat that as a governance event, not simply an application error. This posture may feel strict, but autonomous systems require clear failure boundaries because ambiguity

can otherwise become unauthorized execution.

Break-glass behavior must itself be governed. Emergency access may be necessary in real operations, but it should not become a back door around VAI. Break-glass authority must be time-bounded, strongly evidenced, reviewed after use, and tied to institutional approval.

8.9 Implementation Patterns

VAI should integrate with existing identity systems rather than replace them wholesale.

The cloud adapter pattern is common for infrastructure operations. An OpenKedge contract enters an identity broker. The broker validates the proof. Short-lived cloud credentials are issued with narrow scope. An execution adapter performs the bounded operation. Evidence events are emitted before and after runtime use. This pattern can integrate with provider-native identity primitives while keeping the contract as the authority source.

The capability token pattern is useful where a runtime can verify signed claims directly. The contract produces a signed capability token that encodes allowed action, target, parameters, expiration, and contract id. The runtime verifies the token before execution. This can work for internal platforms, workflow engines, service meshes, and custom execution adapters.

The workflow identity pattern applies to business processes, public-sector systems, and regulated workflows. A workflow engine receives a scoped execution token. Approval and execution steps are bound to the contract. High-risk steps require escalation. Evidence records who or what approved the action, what authority was issued, and how the workflow used it.

The sandbox pattern applies when generated code or agent action must execute in a constrained environment. The sandbox permissions are derived from the contract. The generated code can access only the resources needed for the approved task. Outputs are verified before admission or deployment. If the generated code attempts out-of-contract actions, the sandbox rejects them and emits evidence.

Cedar, OPA/Rego, cloud-native policy systems, and institutional validators may participate in proof validation or runtime checks. VAI should not depend on a single policy engine. It should depend on a stable proof model: the identity broker must verify that the requested authority is justified by an approved contract and current evidence.

Implementation choices will vary across sovereign clouds, commercial clouds, private infrastructure, workflow systems, and regulated platforms. The invariant is that the runtime authority is task-scoped, time-bounded, contract-bound, and evidence-linked.

A practical rollout can begin by placing VAI around the highest-risk autonomous execution paths. For example, a platform team might first protect production infrastructure mutations, then extend the identity broker to deployment workflows, then add scoped tokens for data access and generated-code sandboxes. This incremental pattern is often more realistic than attempting to replace every credential at once. The reference architecture defines the direction: reduce standing privilege where autonomous systems can affect consequential state, and replace it with proof-derived execution identity.

8.10 Design Requirements

VAI implementations must satisfy several concrete requirements.

1. **Task-scoped identity.** The system must issue runtime authority for a specific governed task, not for open-ended agent activity.
2. **Time-bounded authority.** Credentials or tokens must expire quickly and align with the execution contract.
3. **Contract-bound permissions.** Authority must encode or enforce the approved action, target, parameters, and scope.
4. **Proof validation before issuance.** Identity brokers must validate intent, context, policy decision, contract, and time bounds before issuing authority.
5. **Least privilege by construction.** The system must derive only the minimum authority required by the contract.
6. **Revocation and expiration.** Authority must end when the contract expires, context changes, policy changes, execution completes, or risk conditions change.
7. **Runtime enforcement independent of agent compliance.** Enforcement points must reject out-of-contract actions even if an agent attempts them.
8. **Evidence emission for every identity event.** Requests, issuance, use, rejection, expiration, and revocation must all produce evidence.
9. **Replayable justification.** Auditors must be able to reconstruct why authority existed, what it allowed, how it was used, and when it ended.
10. **Integration with existing IAM, workload identity, and policy systems.** VAI must compose with existing infrastructure rather than force a replacement of identity investments.

VAI governs authority at runtime. The next chapter turns to the software construction boundary: how AI-generated code and system components are admitted before they are allowed to participate in governed execution.

9 Protocol-Driven Development

Preceding chapters established the runtime governance lifecycle: reasoning translates into intent, intent compiles into an execution contract, and the contract derives bounded execution identity. Autonomous systems, however, generate software as well as actions, synthesizing scripts, workflow orchestrations, security policies, integration adapters, tests, and control-plane components. When generated code participates in execution, software construction itself enters the governance boundary.

Protocol-Driven Development (PDD) treats machine-enforceable protocols as the primary software artifacts, and implementation code as admissible realizations of those protocols. As machine intelligence reduces the cost of code generation, the primary engineering bottleneck shifts from code production to code admission. PDD formalizes this admission model in greater technical depth [7].

Protocol Authority

The generator carries no standing trust; the protocol defines the basis for admission.

9.1 From Code Generation to Code Admission

Traditional development focuses on writing, reviewing, testing, deploying, and monitoring code, disciplines that remain fundamental to software engineering. Generative artificial intelligence changes the economics of implementation by making candidate code easier to produce at scale. Natural-language specifications, however, remain ambiguous. Generated implementations vary across models, prompts, contexts, and tool histories; many compile and pass superficial tests while failing to align with true system requirements.

As implementations become easier to produce, code admission becomes the control point.

This shift forms the core of PDD. System architects must ask not whether a model can generate code, but whether the institution maintains a machine-enforceable standard to decide which generated artifacts may enter production environments. In autonomous infrastructure, this question is especially important. A generated adapter, workflow, policy module, or remediation script may later acquire execution identity via VAI or participate in an OpenKedge governance pipeline. Admitting a flawed artifact compromises the control plane itself.

PDD addresses this by defining what code must be, rather than how it is generated. A model may yield a single candidate or thousands; the admission boundary remains stable. The generator and implementation may vary, but the admissibility standard remains stable.

Protocol → Candidate Implementation → Admission Evidence → Admit / Reject / Constrain → Runtime Eligibility

This integration incorporates software construction directly into the Autonomous State Control Plane. While runtime governance decides whether an action may execute, protocol admission

determines whether a software artifact is eligible to enter the runtime environment.

The admission problem intensifies under continuous generation. A platform may generate patches during incidents, infrastructure modules during deployments, policy helpers during governance design, data transformations during analytics workflows, and adapters during integration tasks. While each candidate may appear plausible, the institution requires a durable admission rule specifying the active protocol, required evidence, and failure consequences. Without this rule, organizations evaluate artifacts ad hoc based on intuition, urgency, or model confidence.

Admission also requires verifiable provenance. The control plane must track whether an artifact was generated by a model, written by a human, modified by an agent, or imported from an external repository. Provenance does not dictate admissibility, but it calibrates the evidence threshold. A generated adapter serving a production execution path faces a higher admission bar than a low-risk internal utility, regardless of compilation success.

9.2 Why Tests and Prompts Are Insufficient

Natural-language specifications are useful, but they cannot serve as governance boundaries. They are ambiguous, incomplete, difficult to enforce mechanically, and interpreted differently by humans and models. A requirement like ”rotate credentials safely” carries different operational implications for a security engineer, a platform administrator, and a code-generation model. Without a machine-enforceable protocol, the system cannot verify whether the generated implementation satisfies the intended constraint.

Prompts are also necessary but insufficient. They guide generation by supplying constraints, examples, style guides, and policy reminders. However, prompts cannot guarantee behavior; they are not stable control boundaries. Generation processes can ignore, misinterpret, or optimize around prompt instructions. A prompt can request a compliant implementation, but it cannot prove compliance.

Unit tests and examples provide valuable evidence, but they only sample behavior. They fail to cover the complete state space, missing edge cases, concurrency bugs, authorization bypasses, resource exhaustion, and forbidden side effects. A generated implementation can pass every test and still violate the required protocol.

Generated Code Admission

A generated implementation can compile, pass sampled tests, and still violate the protocol the system actually requires. PDD treats tests strictly as evidence, not as the complete governance boundary.

Human code review remains valuable for identifying semantic gaps, architectural flaws, maintainability risks, and domain-specific concerns. However, manual review is slow, highly variable, and impossible to scale to high-volume generated code. It depends heavily on reviewer interpretation and may miss subtle violations of the operational protocol.

PDD does not discard prompts, tests, and reviews; it positions them as evidence inputs rather than the complete admission system. A strong admission pipeline identifies the governing protocol and requires evidence that the candidate satisfies its invariants.

This distinction matters for engineering culture. Teams often treat passing tests as a green light, relying on post-hoc monitoring to catch anomalies. While acceptable for many conventional development workflows, this approach is too weak for components destined for autonomous execution. Monitoring observes behavior after admission; PDD evaluates the artifact before it can affect governed systems.

Tests are most effective when derived directly from the protocol. A test verifying an example is useful; a test enforcing a behavioral invariant is stronger; a property-based test exploring a broad class of inputs is stronger still. PDD does not replace testing; it gives testing a governance target.

9.3 Protocol as the Primary Artifact

A protocol is a machine-enforceable specification of the admissible behavior of a software component. It defines allowed inputs, outputs, state transitions, safety constraints, behavioral invariants, operational limits, evidence requirements, and integration obligations.

Under PDD, implementations are replaceable; protocols are authoritative.

This reverses the traditional hierarchy of software generation. A generated implementation is never trusted based on its source, model, prompt, or toolchain. It is trusted only when empirical evidence demonstrates that it satisfies the protocol. Multiple implementations can satisfy the same protocol. A model may generate one candidate, a human may write another, and a future model may generate a third; the protocol defines the admissible space for all three.

The correctness of the generated artifact is secondary to the correctness of the admission system governing it. While artifact quality matters, it is evaluated relative to the protocol. Without a correct admission system, an organization cannot distinguish a plausible implementation from an admissible one.

Protocols matter for control-plane components. An execution adapter must enforce contracts; a policy module must evaluate inputs consistently; a workflow definition must preserve approval boundaries; a remediation script must avoid forbidden side effects. These components require more than plausible code; they require rigorous admissibility evidence.

Protocol ownership represents an institutional responsibility. Every protocol must maintain a designated owner, version history, defined scope, review path, and change management process. Without protocol ownership, the admission boundary dissolves. In sovereign or regulated environments, this responsibility maps to platform teams, security organizations, domain authorities, or public-sector operating bodies. Protocols represent authority-bearing artifacts and must be governed accordingly.

Protocols also make implementation diversity governable. One team may deploy a generated TypeScript adapter, another a Python script, and a third a managed cloud workflow. As long as they satisfy the same protocol, multiple realizations can coexist under identical structural, behavioral, and operational constraints. This flexibility matters for multi-cloud and sovereign environments where underlying substrates vary.

9.4 The PDD Model

The PDD model represents a protocol as a composition of invariant classes:

$$\mathcal{P} = (\mathcal{S}, \mathcal{B}, \mathcal{O})$$

where $\mathcal{S}$ denotes structural invariants, $\mathcal{B}$ denotes behavioral invariants, and $\mathcal{O}$ denotes operational invariants.

An implementation is admissible if and only if it satisfies the protocol:

$$impl \models \mathcal{P}$$

This formulation shows that the candidate implementation is a valid realization of the protocol-defined admissible space. This notation is intentionally clean; it defines the core relationship without prescribing a specific verification technology: generated code is a candidate realization, and the protocol defines the boundary of admissibility.

Different domains utilize different enforcement mechanisms. A type-safe library relies on static type checks and property tests; a control-plane adapter requires schema validation, policy conformance, sandbox execution, and runtime simulation; a public-sector workflow requires institutional reviews and audit logs. PDD accommodates this variance while maintaining the admission principle.

The model supports a spectrum of assurance. Invariants can be checked statically, validated via dynamic tests, verified through model checking, simulated, or monitored at runtime. PDD does not require complete formal proof for all systems; it requires that the protocol define the admissibility target and that the admission pipeline produce evidence proportional to risk.

This approach keeps the model practical. A low-risk generated helper satisfies its protocol through type checks and conformance tests. A high-risk execution adapter requires sandbox execution, negative tests, threat modeling, and signed approvals. The same admission doctrine applies across both scenarios.

9.5 Type-Theoretic Interpretation of PDD

PDD can be interpreted as a type-theoretic admission layer: generated code becomes operational only when it inhabits the protocol-defined space of admissible implementations.

As machine intelligence makes code generation easier to scale, the central engineering challenge shifts from producing code to verifying admissibility. Type-theoretic techniques help move this admissibility boundary from post-hoc runtime monitoring to pre-runtime verification. In this view, the protocol defines the type of acceptable implementations, and a candidate artifact is admitted only when evidence shows it inhabits that type.

$$impl : \mathcal{P}$$

where $impl : \mathcal{P}$ denotes that the implementation inhabits the protocol-defined admissible space. This connects to our core invariant composition model:

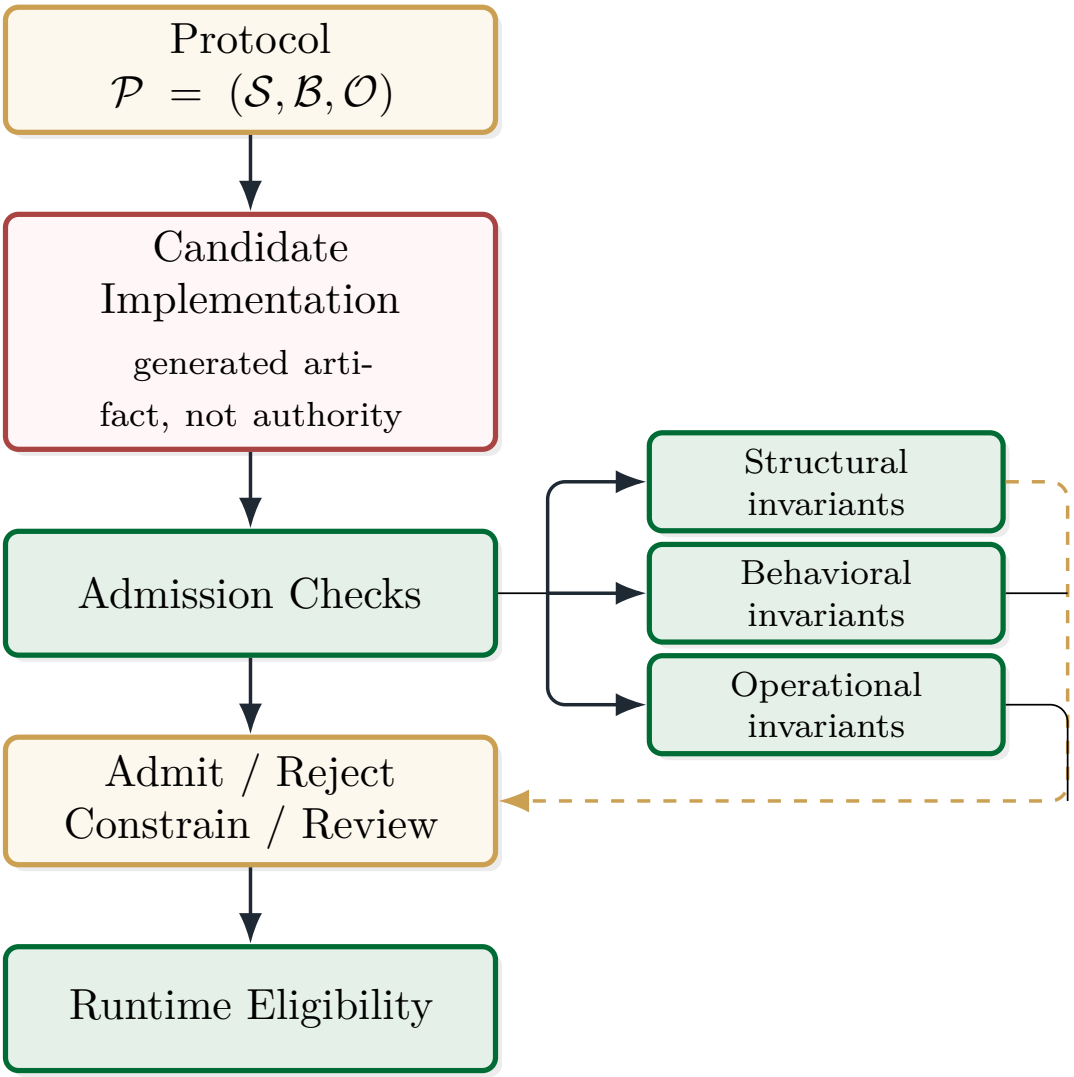


Figure 9.1: *Protocol-Driven Development admission lifecycle. Generated software becomes operational only after satisfying structural, behavioral, and operational invariants defined by the protocol.*

$$\mathcal{P} = (\mathcal{S}, \mathcal{B}, \mathcal{O})$$

Structural invariants align with type and interface constraints: the artifact must expose the correct surface, consume and produce valid data shapes, respect module boundaries, and avoid forbidden dependencies. Behavioral invariants align with refinement predicates and contract systems: the artifact must preserve authorization checks, preconditions, postconditions, state transitions, and error-handling behaviors. Operational invariants align with effect and resource constraints: the artifact must respect side-effect bounds, timeouts, external call limits, evidence obligations, and memory allocations.

Refinement types encode constraints beyond simple shapes, such as bounded percentages, non-empty scopes, permitted resource classes, or ownership predicates. Dependent types express properties parameterized by runtime values or contexts; for example, specifying that a remediation script for a given service and region may only affect resources within that declared scope. Typed effect systems constrain what generated code is permitted to do: marking it as read-only, evidence-emitting, network-free, or cloud-mutating exclusively via designated adapters.

This type-theoretic framing does not replace testing, review, sandboxing, or runtime evidence. It moves as much of the safety boundary as practical prior to execution. While some properties can be checked statically, others require property-based tests, simulation, sandbox execution, or runtime monitoring. PDD is strongest when these techniques combine, and the admission decision logs the evidence that justified the result.

Advanced implementations of this admission pipeline can leverage foundational research in refinement types [4], dependent types [20], and typed effect systems [12].

9.6 Structural, Behavioral, and Operational Invariants

Structural invariants constrain the shape of the artifact. They define required interfaces, type constraints, schema conformance, dependency boundaries, module limits, API surfaces, and configuration schemas. A generated component that imports forbidden libraries, exposes unsupported endpoints, bypasses module boundaries, or generates an invalid schema fails structurally before its behavior is ever evaluated.

Behavioral invariants constrain what the artifact does. They define preconditions, postconditions, state-transition rules, authorization checks, idempotency guarantees, determinism, error handling, and forbidden side effects. A generated workflow that skips an approval step, a script that mutates resources outside its declared scope, or an adapter that retries a non-idempotent operation without safeguards fails behaviorally.

Operational invariants constrain runtime behavior. They define latency budgets, resource allocations, timeout thresholds, logging and evidence requirements, failure recovery, rollback behaviors, rate limits, and observability obligations. A generated service that operates correctly in a unit test but lacks required evidence emission, ignores timeouts, or consumes unbounded resources fails operationally.

Structural invariants determine whether the artifact fits. Behavioral invariants determine whether it functions correctly. Operational invariants determine whether it can run safely.

Table 9.1: *Invariant classes in Protocol-Driven Development.*

Invariant Class	Purpose	Examples
Structural	Constrain the shape and interfaces of the artifact	Types, schemas, dependencies, module boundaries, API surfaces
Behavioral	Constrain the allowed behavior and state transitions	Preconditions, postconditions, authorization checks, idempotency, forbidden effects
Operational	Constrain runtime and deployment properties	Timeouts, resource limits, evidence emission, rollback behavior, observability, rate limits

These invariant classes must be versioned. Protocols evolve as systems learn from operational failures. New evidence may reveal missing failure modes, inadequate observability, or unsafe dependency patterns. Versioning lets the institution know which protocol admitted a given artifact and replay admission decisions when protocols update.

Invariants must also remain composable. A generated control-plane plugin may need to satisfy a general plugin protocol, a security protocol, an evidence-emission protocol, and a domain-specific operational protocol. Composability prevents every artifact from requiring a bespoke admission model, enabling institutions to raise the admission bar in high-risk domains by layering additional invariants instead of rewriting the entire governance surface.

Composed invariants must be checked for conflicts. A latency budget may conflict with a heavy evidence-emission requirement; a dependency restriction may conflict with an observability library; a rollback invariant may conflict with a non-transactional downstream system. Surfacing these design tensions before runtime is a key advantage of the protocol-driven model.

9.7 Evidence-Based Admission

PDD requires empirical evidence, not merely specification.

The admission system asks not whether the implementation looks plausible, but what evidence shows that the implementation satisfies the protocol.

Admission evidence includes static analysis reports, type checking outputs, schema validation logs, model-checking proofs, property-based test results, runtime simulation metrics, sandbox execution traces, policy checks, security scans, conformance tests, and human review attestations. Different protocols demand different evidence. A low-risk generated utility requires type checks and linting; a control-plane adapter requires sandbox execution, policy conformance, evidence-emission tests, and manual reviews; a generated workflow affecting public services requires formal institutional approval and replayable admission logs.

Admission outcomes must remain explicit. A candidate is admitted, rejected, admitted with constraints, requested to produce additional evidence, routed to human review, restricted to sandbox use, or required to pass simulation before achieving runtime eligibility. Admission represents an active governance decision, not a passive build step.

The admission process itself must emit evidence. The system logs the submitted artifact, the

governing protocol version, the checks executed, the evidence produced, the decision reached, the authorizers, and any active constraints. This admission record links directly to deployment logs, IEEC events, and later runtime telemetry.

Evidence-based admission supports system learning. If an admitted artifact is involved in an incident, reviewers inspect the admission evidence to determine whether the protocol was incomplete, the checks were insufficient, the implementation drifted after admission, or the runtime environment changed.

Rejected artifacts also produce evidence. A rejection log records which protocol requirement failed and why. Repeated rejections reveal whether the prompt is underspecified, the model is poorly suited to the task, the protocol is ambiguous, or the check is too brittle. Negative admission evidence is operationally useful, helping engineers refine generation tasks and protocols without lowering the admission bar.

Admission with constraints is another useful outcome. A generated artifact may be safe for sandbox use but not production; it may be admissible for read-only tasks but not mutations; it may be valid for one cloud provider but not another. Encoding these constraints avoids a false binary between full admission and rejection.

9.8 PDD in the Autonomous State Control Plane

PDD is not a post-execution logging layer; it is an out-of-band, continuous admission substrate.

PDD operates before and around runtime governance, determining whether generated software, policies, adapters, workflows, and agents are eligible to enter the control plane.

Sovereign Software Admission

In sovereign AI systems, generated software must never become operational simply because a model generated it. It must pass through admission protocols owned by the institution responsible for execution.

The artifacts governed by PDD encompass generated remediation scripts, execution adapters, workflow definitions, policy modules, infrastructure-as-code modifications, agent tool wrappers, data transformations, simulation models, approval workflows, and control-plane plugins. These artifacts shape how autonomous systems behave, serving as the tools agents call, the adapters enforcing contracts, the policies governing intent, or the workflows routing approvals.

This explains why PDD belongs inside the Autonomous State Control Plane. If the control plane governs runtime actions but admits generated components casually, the governance boundary is compromised. A generated adapter with excessive permissions or missing evidence emission undermines VAI; a generated policy module with ambiguous logic undermines OpenKedge; a generated tool wrapper that bypasses intent isolation undermines SAL.

PDD also participates in the closed-loop cycle. Runtime evidence reveals protocol gaps, replay identifies unsafe behavior, and simulation refines invariants. Updated protocols constrain future generated artifacts. In this sense, PDD is a learning admission system connected to live operational telemetry.

This is the software counterpart to runtime governance. OpenKedge asks whether a proposed intent may execute now; VAI asks what authority is justified now; PDD asks whether the software artifact is eligible to participate in those decisions at all. If an agent generates an execution adapter, the adapter must satisfy the relevant protocol before it can receive contracts. If a model generates a policy helper, it must satisfy policy-module invariants before influencing decisions. If an agent generates a remediation script, the script must be admitted before it can execute with proof-derived execution identity.

9.9 Relationship to OpenKedge, SAL, and VAI

SAL governs how reasoning crosses into sovereign governance; OpenKedge governs whether a proposed intent may execute; VAI governs the runtime authority issued for approved execution; and PDD governs whether generated code and system components are admissible before participating in the loop.

SAL governs reasoning boundaries; OpenKedge governs intent; VAI governs authority; and PDD governs admissibility.

Table 9.2: *Role of PDD within the Autonomous State Control Plane.*

Layer	Governance Question	Primary Artifact
SAL	May this reasoning output cross into sovereign governance?	Structured intent
OpenKedge	Should this intent be allowed under policy and context?	Execution contract
VAI	What runtime authority is justified by the approved contract?	Execution identity
PDD	Is this generated artifact admissible for use in the system?	Machine-enforceable protocol and admission evidence

This relationship is composable. PDD governs the artifacts used by SAL, OpenKedge, and VAI, including model-context transformation code, policy adapters, identity brokers, execution adapters, evidence emitters, and simulation tools. In return, evidence from those layers refines PDD protocols. This supports architectural coherence: the systems that govern autonomous agents are themselves governed.

This recursive property matters. A control plane built from ungoverned generated components would contradict its own doctrine. It would govern agent actions while leaving the underlying software substrate to informal trust. PDD closes this gap, applying the same discipline to system construction that the control plane applies to execution: define the boundary, produce evidence, make decisions replayable, and refine the boundary from operational telemetry.

9.10 Implementation Patterns

A protocol registry represents the first implementation pattern. This registry houses versioned, owned, machine-enforceable protocols for components, adapters, workflows, policy modules, gen-

erated scripts, and plugins, specifying their invariants, evidence requirements, and applicability scope.

An admission pipeline represents the second pattern. Generated candidate artifacts flow through validation, testing, static analysis, simulation, policy checks, sandbox execution, and evidence generation. The pipeline produces explicit admission outcomes rather than treating build success as admission.

Sandbox execution is mandatory for untrusted generated artifacts. Candidate remediation scripts, adapters, or workflows run in constrained environments that restrict permissions, simulate target systems, observe side effects, and produce empirical evidence before runtime eligibility.

Protocol-constrained generation is useful but insufficient. Prompts and generation tasks include protocol specifications as constraints, examples, and acceptance criteria to help models produce better candidates. However, the admission pipeline must verify outputs independently. Prompts guide generation; protocols govern admission.

An evidence ledger records all admission decisions, connecting them to later runtime evidence. When an admitted artifact participates in an OpenKedge decision or a VAI execution path, the system can trace its origin back to the protocol that admitted it.

CI/CD integration makes PDD operationally practical. Generated artifacts are gated before merge, deployment, or runtime use, using build pipelines, code owners, policy checks, admission controllers, and deployment gates. Generated artifacts should not become operational through informal reviews or unchecked automation.

Runtime feedback closes the loop. Telemetry from OpenKedge decisions, VAI enforcement, incident reviews, and simulations updates protocols, test suites, and admission checks. This helps the admission system improve as the autonomous system encounters real-world workloads.

Implementation must also support artifact quarantine. Generated artifacts that fail admission are isolated from deployment paths and runtime registries. Previously admitted artifacts that later violate operational evidence expectations are suspended until review, preventing failed candidates from lingering in informal repositories where they could be accidentally reused.

For high-impact environments, admission decisions must be reproducible. Given identical artifacts, protocol versions, toolchains, and evidence inputs, the system must reconstruct why it admitted or rejected the artifact. This requires deliberate capture of the admission environment.

9.11 Design Requirements

PDD implementations must satisfy several concrete requirements.

1. **Machine-enforceable protocols.** Protocols must be represented in formats that tools can validate, test, simulate, or enforce.

2. **Structural, behavioral, and operational invariants.** Protocols must cover shape, behavior, and runtime safety rather than relying solely on tests.

3. **Protocol registry.** The system must maintain versioned, owned protocols for generated artifacts and control-plane components.

4. **Candidate artifact submission.** Generated code, workflows, policies, adapters, and configurations must pass through a submission path before admission.
5. **Automated admission checks.** Static analysis, type checks, schema validation, policy checks, simulations, and conformance tests must run automatically.
6. **Evidence generation.** Every admission decision must produce structured evidence detailing the artifact, protocol version, checks, results, and outcome.
7. **Sandbox and simulation support.** Higher-risk artifacts must run in constrained environments before achieving runtime eligibility.
8. **Human review for ambiguous or high-risk artifacts.** The pipeline must escalate to human review when protocol satisfaction is unclear or the institutional consequence is high.
9. **CI/CD integration.** PDD must gate generated artifacts before merge, deployment, registry admission, or operational use.
10. **Runtime feedback for protocol refinement.** Operational telemetry must actively refine protocols, test suites, simulations, and admission rules.
11. **Versioned protocols and artifacts.** Reviewers must be able to verify which protocol version admitted which artifact and under what evidence.
12. **Replayable admission decisions.** Admission outcomes must remain reconstructable for auditing, incident forensic investigations, and protocol evolution.

With SAL, OpenKedge, VAI, and PDD in place, the whitepaper turns from architecture to application: how this control-plane doctrine applies to national-scale AI initiatives such as Vision 2030, sovereign clouds, smart cities, and AI-native public administration.

10 Saudi Arabia Vision 2030 Case Study

Sovereign Leadership / حكمة سيادية

الحزم أبو العزم، ولا حزم إلا بتدبير

"Firmness is the father of resolve, and there is no firmness without deliberate governance."

Saudi Arabia is a useful reference case for sovereign AI infrastructure. Its national transformation agenda connects digital government, infrastructure development, smart-city programs, industrial modernization, and large-scale investment with advanced AI systems. In this setting, deployment alone is not enough: autonomous decisions must remain governable when they touch public administration, financial systems, utility grids, and municipal environments.

National AI Governance

The strategic asset is not only the model. It is the control plane that determines how intelligence is allowed to act.

This chapter outlines how the Autonomous State Control Plane (ASCP) can govern national-scale AI programs, using Saudi Arabia's Vision 2030 and its emerging sovereign AI initiatives as an illustrative framework [11]. This case study is analytical rather than policy-prescriptive; it does not represent official policy, partnerships, or endorsements by any Saudi public or private entity. References to national AI infrastructure, data governance frameworks, or specific smart-city and industrial initiatives serve strictly to ground the architectural discussion in realistic scale and operational complexity.

Executing national AI strategies requires more than licensing frontier models or building localized datacenters. It requires an execution control plane that makes autonomous systems auditable, replayable, and subordinate to sovereign policy. Preserving state sovereignty does not require isolating national networks from global AI; it requires owning the control plane that translates AI-generated reasoning into policy-compliant execution. The models may remain global, but execution authority must remain sovereign.

10.1 Why Saudi Arabia Is a Reference Case

Saudi Arabia serves as an instructive reference environment because Vision 2030 coordinates economic diversification, public-sector modernization, digital transformation, and infrastructural development into a unified strategic framework [11]. In this context, AI systems do not function as isolated consumer applications. Instead, they interact directly with public-sector administrative

queues, cloud infrastructure operations, municipal sensory networks, and large-scale investment execution pipelines.

This makes the national transformation program a useful lens for systemic AI governance, not only a high-growth market for AI tools. A national-scale AI program must coordinate actions across multiple ministries, regulatory agencies, sovereign cloud regions, public data platforms, and critical utility networks. It must foster technical innovation while keeping institutional authority explicit and local. AI reasoning may assist, automate, or expedite administrative tasks, but model outputs should not become unbounded operational or financial authority.

While agencies like the Saudi Data and AI Authority (SDAIA), national infrastructure programs, and smart-city developments represent the scale of institutional domains requiring coherent AI governance, this analysis uses these entities as reference categories to clarify the governance problem [18, 10]. It does not assert adoption of the ASCP by any specific Saudi institution.

The core architectural challenge is managing AI operations across heterogeneous domains with different risk tolerances, data boundaries, and regulatory constraints. An automated action that is low risk in a back-office classification queue becomes high risk when applied to benefit allocations, procurement approvals, grid balancing, or traffic systems. Because model capabilities are acceptable in some contexts and hazardous in others, governance cannot reside only within the model, the prompt, or the application. It needs a control plane that governs the transition from reasoning to action.

Sovereign AI is therefore more than model or data ownership. While model and data sovereignty are necessary foundations, they are incomplete without execution sovereignty. An internally trained model that directly mutates production databases without validation, credential boundaries, or audit trails remains a material risk. Conversely, a global frontier model isolated behind a strong execution control plane can be used for complex reasoning without receiving direct operational authority. The key boundary is the gateway where cognitive output becomes state-mutating execution.

10.2 The Sovereign AI Challenge

The sovereign AI challenge is not whether a state relies on external intelligence, but whether that intelligence is permitted to assume administrative authority. This distinction matters for any state using a heterogeneous mix of proprietary, open-source, and cloud-native AI models. The primary strategic question is not which model family performs the cognitive reasoning, but which institution controls the execution boundary.

External Intelligence as External Authority

The primary systemic risk is allowing external intelligence to transition into active, unvalidated authority over national databases, workflows, critical infrastructure, and administrative decisions.

A national-scale AI architecture must use advanced global reasoning capabilities while safeguarding sensitive internal context, administrative authority, and operational evidence. It must avoid architectural lock-in to any single AI provider or cloud platform, support diverse policy en-

gines, and enforce least-privilege constraints. Autonomous systems should not mutate national infrastructure using static credentials or broad IAM roles simply because an agentic toolchain is technically capable of doing so.

This challenge is most acute in public-sector and regulated environments. An agent might draft an administrative decision, flag missing evidence, recommend a vendor selection, or generate a system configuration patch. These outputs represent useful cognitive inputs, but none constitute official authority. The validity of the action depends not on the semantic confidence of the model, but on whether the proposed intent aligns with institutional policy, current system state, and explicit legal boundaries.

Scale complicates this governance task. A national AI control plane must operate consistently across distributed departments, multi-cloud clusters, municipal platforms, and cross-border data partnerships. This requires model-neutral and cloud-neutral mechanisms for intent verification, proof-derived execution identity, tamper-evident logging, simulation, and protocol-based code admission.

The Autonomous State Control Plane (ASCP) addresses this tension by separating external reasoning from sovereign execution. The Sovereign Agentic Loop (SAL) secures the reasoning interface; OpenKedge evaluates structured intent against policy and context; the Verifiable Agentic Infrastructure (VAI) converts approved contracts into proof-derived execution identities; the Intent-to-Execution Evidence Chain (IEEC) logs the lifecycle; and Protocol-Driven Development (PDD) gates generated artifacts before deployment. These layers allow a sovereign state to use capable reasoning models without relinquishing control over physical and digital systems.

10.3 The Third Path: Global Intelligence, Sovereign Execution

The strategic debate around sovereign AI often presents a narrow choice: rely entirely on global frontier models and vendor-controlled execution paths, or delay deployment until every accelerator, model, cloud layer, and software component is developed domestically. Both approaches impose operational costs. Dependence on external platforms can outsource too much control over national administrative systems. Conversely, insisting on complete technological autarky can slow public-sector modernization, restrict access to current cognitive capabilities, and consume domestic resources before governance systems are ready.

The ASCP establishes a pragmatic third path: use global AI reasoning capabilities while retaining ownership of the control plane that governs execution. State sovereignty is not preserved by isolating national systems from global cognitive tools, but by controlling the gateway through which those tools interface with local environments. By treating AI output as unvalidated, structured intent, this model keeps actual execution subject to local policy, real-time context, sovereign identity systems, and tamper-evident records.

Third Path for Sovereign AI

National execution sovereignty is preserved not by restricting cognitive inputs, but by keeping local control over the execution gateway.

This model is suited for states driving rapid public, industrial, and digital infrastructure mod-

ernization. It allows administrators to deploy capable global reasoning engines for specific analytical tasks while retaining control over context exposure, data residency, authorization boundaries, and memory retention. Sensitive government context is sanitized or minimized before reaching external reasoning endpoints. Runtime execution identities are generated locally after policy validation, and operational evidence remains under sovereign administrative custody.

This decoupled approach also supports strategic and vendor flexibility. Integrating governance mechanisms directly into a single model family or cloud platform binds sovereign authority to a vendor's proprietary architecture. In contrast, the ASCP positions policy engines, identity providers, and evidence ledgers above the infrastructure layer. This architecture allows organizations to change reasoning models, integrate local agents, and coordinate multi-cloud environments under a common governance standard.

The guiding doctrine is straightforward: autonomous systems may advise, simulate, propose, and generate, but they should not hold unilateral authority to mutate sovereign state. Actual execution authority is granted by the local control plane holding the policy and evidence boundary, reducing the risk that cognitive errors cascade into operational failures.

10.3.1 Neuro-Symbolic Governance for National AI

For national-scale AI implementation, the neuro-symbolic split provides a practical architectural standard: neural models accelerate unstructured reasoning, while symbolic policy engines enforce deterministic authority. Large language models and agentic networks are useful for parsing regulations, summarizing administrative records, translating documents, and proposing optimization plans. They should not serve as final arbiters of public decisions, infrastructure changes, or regulatory actions.

National-scale governance needs auditability and predictability beyond what probabilistic neural traces can provide on their own. Symbolic systems, by contrast, encode explicit regional policies, boundary constraints, workflow approvals, and audit requirements as executable rules. When a neural agent proposes a system change, a symbolic policy gate evaluates the resulting structured intent against active rules. If policies are violated, context is insufficient, or the proposed action exceeds the safety envelope, the system rejects or escalates the action instead of permitting silent automation.

This analysis does not claim that any Saudi entity has adopted this model. It uses the neuro-symbolic split as a reference pattern for national-scale AI governance: reasoning can be accelerated by models, while execution authority remains with the institution responsible for policy, evidence, and accountability.

10.4 National Use Case 1: AI-Native Public Administration

AI-native public administration is a clear application of execution governance. Public administration spans diverse administrative workflows, including licensing, permit processing, procurement approvals, benefits eligibility, case triage, and municipal routing. These workflows are highly structured, governed by legal frameworks, and subject to public accountability and appeal mechanisms.

AI agents can significantly improve administrative throughput by summarizing voluminous applications, detecting inconsistencies, and drafting responses. However, a cognitive recommendation is not an administrative decision; a generated draft is not an official act of state; and an automated workflow route is not administrative authority. In public administration, speed must not compromise legal accountability.

The ASCP handles all high-impact AI recommendations as candidate intents. If an agent proposes approving a licensing permit or escalating a benefits case, OpenKedge encapsulates the recommendation into a structured intent outlining the actor, objective, workflow context, requested action, and target database. The control plane then evaluates this intent against machine-executable policies, legal statutes, and case-specific context.

Low-risk operations, such as dispatching informational emails or requesting missing files, may run automatically under tight constraints. High-impact administrative actions—such as final eligibility determinations or financial approvals—require manual sign-off, multi-agency validation, or pre-execution simulation. In all cases, the evaluation path is recorded. Rejections, approvals, constraints, and manual escalations are stored in tamper-evident records, providing the audit trail needed to support public trust and citizen appeals.

This model does not automate administrative discretion. It makes the boundary between machine execution and human judgment explicit, so public administration can remain operationally efficient and accountable.

10.5 National Use Case 2: Smart Cities and Digital Twins

Cyber-physical smart cities and digital twin environments [14] present a distinct set of operational risks. In these ecosystems, autonomous systems are designed to optimize traffic flows, manage utility grids, coordinate emergency services, and schedule municipal maintenance. Connecting cognitive models directly to physical infrastructure may improve efficiency, but it also requires rigorous execution governance.

A digital twin must do more than predict urban dynamics; it must govern how automated proposals alter physical systems. Unchecked optimization routines can generate severe real-world hazards. For instance, a model may propose a mathematically optimal traffic signal pattern that inadvertently blocks emergency vehicle corridors, violates municipal safety interlocks, or conflicts with active utility repairs.

Under the ASCP, every cyber-physical modification clears intent governance before reaching actuators. A proposed traffic adjustment or utility load shift is formatted as a structured intent. OpenKedge evaluates it against active safety protocols, dependency state, and local policies. High-impact operations trigger automated simulations to assess downstream effects. Critical changes require human verification, while multi-agency operations require coordinated digital signatures.

VAI then generates a task-scoped, time-bounded execution identity for the specific action. An agent authorized to balance energy loads in a specific microgrid cannot access water treatment systems, nor can a traffic routing agent modify traffic patterns outside its designated window.

Post-incident forensics matters for municipal systems. In the event of an operational failure, the IEEC enables operators to reconstruct what the AI system observed, what actions it proposed,

the context of the policy evaluation, and the physical mutations that resulted. PDD requires AI-generated automation scripts or control configurations to be audited against behavioral invariants before they interact with physical infrastructure.

10.6 National Use Case 3: Sovereign Multi-Cloud AI Infrastructure

National AI strategies rarely rely on a single, homogeneous infrastructure. Instead, they span domestic datacenters, sovereign cloud regions, commercial hyperscalers, edge clusters, and legacy agency systems. Managing governance across this heterogeneous landscape is complex; each provider features distinct IAM semantics, logging formats, and deployment pipelines.

The ASCP establishes a model-neutral, cloud-neutral, and policy-sovereign governance fabric. Model neutrality reduces lock-in by decoupling cognitive engines from the decision boundary. Cloud neutrality allows the control plane to map execution contracts to different infrastructure endpoints using standardized adapters. Policy sovereignty keeps policy semantics, approval rules, and audit records under local institutional control.

For example, when an AI operations agent proposes a cloud infrastructure change, the proposal enters the control plane as a structured intent. OpenKedge evaluates it against organizational policy. VAI then generates short-lived, provider-agnostic execution credentials, which adapters translate into provider-specific API calls. Throughout this cycle, IEEC logs the event in a common format, allowing administrators to audit and replay actions across different clouds.

This decoupling reduces vendor lock-in. Rather than relying on cloud-native security groups to govern agent behaviors, the ASCP establishes a governance plane above the virtualized hardware, using providers as execution substrates. Policy engine pluggability allows agencies to use existing investments in engines like Cedar or OPA/Rego while maintaining a consistent sovereign execution boundary across their digital estate.

10.7 National Use Case 4: Industrial and Critical Infrastructure AI

Critical infrastructure sectors—such as energy distribution, chemical manufacturing, telecommunications, and heavy transport—have very low tolerance for operational error. In these sectors, AI systems are deployed to predict equipment failures, schedule preventative maintenance, and generate system configurations. The potential benefits are substantial, but the cost of an unconstrained action can be severe.

Industrial AI cannot operate based on probabilistic confidence alone. Offline equipment schedules, pipeline pressure adjustments, and network configuration changes can trigger cascading utility failures, heavy financial losses, or physical accidents. The ASCP reduces these risks by classifying the potential blast radius of every proposed action. A maintenance request that is automatically approved during low-demand periods is escalated or denied during peak loads or active emergency conditions.

Standing, broad-scoped operational credentials create a high security risk. The ASCP uses VAI to replace static privileges with short-lived, task-scoped execution identities generated dynamically from validated contracts. An operations agent receives authority only for the approved scope, within the designated time frame, and with a verifiable cryptographic link to the underlying policy decision.

Because industrial environments frequently use generated code and configurations, PDD matters. Every AI-generated automation script or control template must pass automated admission checks before deployment. Structural checks verify syntax and compatibility, behavioral checks simulate execution to detect unauthorized state changes, and operational checks verify rollback capabilities and timeout boundaries. An artifact is admitted to the runtime registry only when evidence shows that it satisfies its designated protocol, reducing the risk that automated updates compromise system integrity.

10.8 Executable Regulation and National Auditability

National AI governance needs more than high-level ethical frameworks and static compliance documents. High-level principles guide policy, but autonomous environments also need executable protocols. The core challenge is translating legal mandates, risk guidelines, and agency policies into machine-enforceable rules, execution boundaries, and verifiable evidence.

This translation must be handled carefully, recognizing that human discretion remains irreplaceable in complex cases. Every autonomous system boundary should explicitly define which actions can be automated, which require multi-party approval, and which are prohibited. A national architecture should codify these rules within the control plane rather than embedding them implicitly within prompts or application code.

Executable regulation does not reduce complex laws to simplistic code blocks. Instead, it defines operational boundaries. For instance, a benefits processing agent may automate document validation but is prohibited from unilaterally denying an application. A database utility may adjust resource allocations within defined limits but must escalate to human administrators if data migration is required.

Under this model, auditability is designed into the system rather than added as an after-the-fact reporting exercise. Every autonomous mutation is linked to a validated intent, a real-time policy evaluation, a cryptographic execution contract, a proof-derived identity, and verified execution evidence. When an audit is conducted, the system can show what action occurred and the chain of authority that permitted it. This level of verification supports regulatory compliance, security audits, and public accountability.

10.9 Reference Architecture for a Sovereign National Deployment

A sovereign national deployment of this architecture comprises seven discrete operational layers. This blueprint illustrates a reference deployment rather than a specific national implementation, showing how ASCP coordinates distributed, high-impact AI systems.

Reasoning Layer → SAL Boundary → OpenKedge Governance → VAI Runtime Trust → Execution Adapters → IEEC Ledger → PDD Admission

1. **Cognitive Reasoning Layer:** Integrates domestic models, global frontier engines, specialized domain agents, translation services, and simulation engines. These components parse datasets and draft operational recommendations. By default, they possess no execution authority.
2. **Sovereign Agentic Loop (SAL) Boundary:** Restricts cognitive interactions by implementing context minimization, data obfuscation membranes, and intent isolation. It allows cognitive models to generate reasoning while preventing the exposure of sensitive sovereign context or the execution of unvalidated instructions.
3. **OpenKedge Intent Governance Layer:** Evaluates candidate intents by binding them to verified administrative actors and defined organizational objectives. It gathers relevant system context, applies active policy files, assesses blast-radius risks, and issues bounded execution contracts.
4. **Verifiable Agentic Infrastructure (VAI) Trust Layer:** Translates approved execution contracts into cryptographic runtime credentials. It validates system proofs, issues short-lived, task-scoped tokens, enforces automated credential revocation, and records identity generation metrics.
5. **Execution Adapter Layer:** Maps approved contracts to target environments, including government databases, municipal controllers, industrial APIs, and cloud resources. These adapters enforce physical contract boundaries and translate abstract governance rules into target-specific operational calls.
6. **Intent-to-Execution Evidence Chain (IEEC):** Registers every step of the transaction lifecycle in a tamper-evident ledger. This logging supports audit dashboards, policy review, and offline simulation.
7. **Protocol-Driven Development (PDD) Admission Layer:** Audits all generated code, workflow modules, and integration adapters before runtime deployment. It maintains a registry of authorized software protocols, executes behavioral and structural conformance checks, and gates the CI/CD pipeline.

These layers form a decoupled governance system. Cognitive models, infrastructure providers, and policy syntax may change over time, but the sovereign control plane remains the stable boundary that prevents probabilistic reasoning from bypassing deterministic administrative control.

10.10 Pilot Program Roadmap

Deploying a national-scale governance architecture requires a staged, evidence-first approach starting with highly bounded pilot programs. These pilots test whether intent governance, cryptographic identities, tamper-evident records, and protocol-based software admission can operate reliably within actual institutional environments while keeping operational risk bounded.

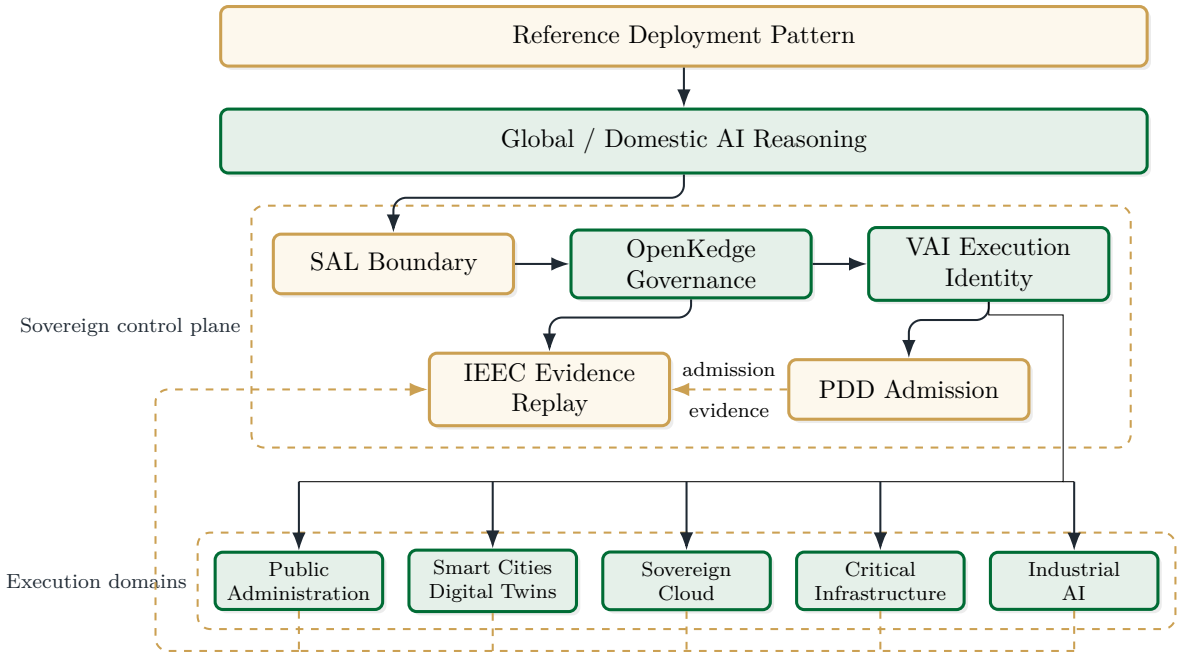


Figure 10.1: *Reference deployment pattern for applying the Autonomous State Control Plane to national-scale sovereign AI programs. The diagram is illustrative and does not imply adoption by any specific institution.*

A credible pilot program should evaluate metrics beyond processing speed or throughput. Useful indicators include policy compliance rates, false approval and denial indices, context freshness, average credential lifespans, policy explainability scores, and human operator trust. These indicators help determine whether the control plane improves security or merely introduces administrative friction.

10.11 Strategic Implications

The core strategic implication of this reference study is that sovereign AI depends heavily on control-plane ownership. Modern states can use a broad spectrum of cognitive technologies—including proprietary frontier systems, open-source models, and specialized agents—without relinquishing administrative authority. Sovereignty is maintained not by isolating local systems from external tools, but by inserting a deterministic control plane that governs how those tools interact with production networks.

This shift reframes the debate around technological sovereignty. The primary strategic asset is not the reasoning model, but the control plane that regulates how that model acts. Models produce probabilistic reasoning; the control plane produces deterministic authority. Execution generates telemetry; the evidence chain produces institutional accountability. Protocols verify software components, and replay systems drive continuous policy improvement.

For any state or regulated enterprise deploying autonomous AI in high-impact environments, the challenge is similar: connecting probabilistic reasoning models to deterministic physical and digital systems. Waiting for more capable models or relying only on post-execution log reviews is

Table 10.1: *Potential pilot programs for sovereign AI control-plane adoption.*

Pilot	**Objective**	**Primary Control-Plane Mechanism**
Public-service workflow	Accountable AI-assisted administration	Intent governance, escalation, evidence chain
Cloud remediation	Safe autonomous infrastructure operations	Execution contracts and proof-derived execution identity
Smart-city digital twin	Governed cyber-physical decision support	Simulation, approval, bounded execution
Generated-code admission	Safe use of AI-generated automation	Protocol-Driven Development and admission evidence

insufficient. Organizations need active governance infrastructure so automated actions are bounded, verified, replayable, and legally accountable.

The next chapter generalizes these case-study patterns into a staged, multi-phase deployment roadmap for organizations adopting the Autonomous State Control Plane.

11 Deployment Roadmap

Transitioning from theoretical control-plane architectures to production deployments requires a structured, staged adoption methodology. Rather than granting autonomous agents direct, unvalidated access to production systems, organizations must adopt an incremental approach that begins with passive observation, shifts to structured intent logging, introduces declarative policy evaluations, and culminates in cryptographically bounded contracts and task-scoped execution identities.

Implementing the Autonomous State Control Plane (ASCP) is not a "big-bang" migration. It is an incremental hardening process that secures existing infrastructures by progressively narrowing the path through which agentic networks can alter system state.

Assess → Capture Intent → Evaluate Policy → Issue Contract → Derive Identity → Execute Boundedly → Record Evidence → Replay → Refine

11.1 Adoption Principles

Initial deployment phases prioritize absolute accountability over complete automation. Before granting autonomous systems mutation authority, operators must analyze the actions agents propose, the tools they invoke, the context they retrieve, the credentials they assume, and the forensic evidence generated during execution. Early phases therefore establish rigorous monitoring and risk-tiering frameworks before active enforcement is enabled.

Deployment Doctrine

Observe before enforcing. Constrain before automating. Replay before scaling.

The ASCP integration methodology is incremental, risk-stratified, vendor-agnostic, and designed to compose with existing identity providers, cloud IAM frameworks, databases, and monitoring networks. Rather than bypassing established access controls, the control plane acts as a strategic orchestration layer that binds diverse infrastructure components under a single governance standard.

Staged adoption allows operations teams to validate policies in a passive "shadow" mode before active blocking is enforced. Similarly, execution contracts can be introduced for low-risk, reversible administrative tasks before governing high-impact operations. Continuous simulation and replay cycles enable administrators to verify policy behavior before scaling the architecture across multiple departments, clouds, or business units.

Risk-tiering dictates that low-risk actions—such as read-only queries, case routing, and fully reversible operations—are governed early. High-impact operations—including financial approvals, infrastructure mutations, safety-critical signals, and regulated data processing—require rigorous policy evaluation, multi-signature approvals, pre-execution simulation, and complete evidence chains. A successful deployment is characterized by the systematic application of context-specific controls to discrete, validated actions.

Finally, the roadmap preserves institutional flexibility. The ASCP operates independently of specific policy engines (supporting Cedar, OPA/Rego, or legacy validators), cloud APIs, and database technologies. This compatibility ensures that security analysts, system operators, and compliance officers receive standard, high-fidelity audit evidence without requiring a total infrastructure overhaul.

11.2 Phase 0: Readiness Assessment

Phase 0 inventories and classifies the pathways through which autonomous or semi-automated systems interact with organizational state. Governance is impossible without mapping the vectors where AI-generated intent can mutate production databases, configure hardware, or process sensitive records. This assessment evaluates formal agent platforms as well as informal workflows that rely on AI-generated scripts, semi-automated operations, or developer assistants.

The readiness assessment maps all AI models, agentic integrations, automation pipelines, cloud remediation daemons, and database access routes. It documents the service accounts, IAM roles, active policy libraries, audit log destinations, and operational compliance mandates across the enterprise. The goal is to define the boundary where probabilistic outputs convert into deterministic state mutations.

Workflows are classified across multiple dimensions, including execution reversibility, potential blast radius, regulatory impact, data sensitivity, human oversight, credential models, and audit depth. Low-risk workflows comprise tasks like internal ticket classification or missing-data requests. Medium-risk operations include non-production resource configurations or reversible system remediations. High-risk paths directly affect production environments, benefits allocations, financial ledgers, public services, and critical infrastructure.

Table 11.1: *Readiness assessment dimensions.*

Dimension	Assessment Question	Output
Autonomous action paths	Where can AI-assisted systems affect state?	Action inventory
Identity and privilege	What credentials or roles can be used?	Privilege map
Policy coverage	What rules govern each action?	Policy map
Auditability	What evidence exists today?	Evidence gap analysis
Risk tier	What is the blast radius and reversibility?	Risk matrix
Pilot suitability	Which workflow is bounded and high-value?	Pilot shortlist

Phase 0 produces five key deliverables: a verified autonomous-action inventory, a privilege and IAM role map, a policy and audit coverage map, a risk-tier matrix, and a pilot workflow shortlist. Pilot workflows must feature high business value, clearly bounded scopes, deterministic policy criteria, available context streams, and easily mitigated failure modes. Workflows characterized by political ambiguity, broad standing credentials, or sparse telemetry are unsuitable for initial pilots.

This phase also highlights existing telemetry gaps. Standard system logs record the execution of API calls, but they fail to document why an agent proposed a specific action, the context considered, the policies applied, the alternatives evaluated, or whether the execution aligned with the initial intent. Identifying these blind spots defines the initial evidence requirements for the control plane.

11.3 Phase 1: Intent Capture and Shadow Governance

Phase 1 implements passive shadow governance by capturing AI-generated proposals as structured intents without altering operational flows. The control plane observes agent proposals, database requests, and script executions, converting them into standardized, structured intent records for analysis.

Passive intent capture is critical because real-world agent behaviors frequently diverge from idealized designs. Agents often propose out-of-sequence operations, request excessive permissions, omit critical context parameters, repeat failing tool calls, or generate ad-hoc scripts rather than direct API calls. Collecting these patterns in shadow mode mitigates deployment risks when active enforcement is enabled.

The intent schema encapsulates the reasoning source, requester identity, target resource, specific action, operational scope, context assumptions, expected state changes, and explicit justification. The schema is designed to be highly extensible, allowing it to adapt as new agentic behaviors are documented.

During this phase, existing execution paths remain unchanged. The control plane operates strictly as a passive listener, recording intent and generating shadow decision metrics. This low-risk deployment model allows policy authors, security teams, and database owners to draft, test, and refine policy rules against actual production intents rather than abstract assumptions.

Phase 1 delivers the intent gateway, the core intent schema, shadow execution logs, baseline evidence records, and initial policy requirements. Key baseline metrics include intent volume, targeted resources, risk distribution, context-retrieval latency, common parsing failures, and credential usage patterns.

This shadow phase also aligns the vocabularies of diverse stakeholders. Domain owners, security administrators, and systems engineers can collaborate around a shared, structured object: the intent record. This represents the first practical step toward governed autonomy.

11.4 Phase 2: Policy Evaluation and Shadow Decisions

Phase 2 integrates active policy engines and real-time context providers to evaluate captured intents, recording the resulting decisions on a shadow evidence ledger. The control plane determines whether proposed actions would be allowed, denied, modified, or escalated, providing a rigorous testbed for rule validation before active enforcement.

This phase deploys the context provider layer, policy engine adapters (such as Cedar or OPA/Rego), risk classification engines, and the pre-execution stages of the Intent-to-Execution Evidence Chain (IEEC). Context providers aggregate operational facts, including database status, dependency health, system load, active change windows, data classification levels, and transaction histories.

Policy engines evaluate structured intents against these context streams to determine policy compliance.

Although the control plane does not yet block physical execution, it generates explicit governance decisions (allow, deny, constrain, escalate, or defer). Every decision produces structured telemetry. A denial record details the specific policy rules violated or context fields missing; a constraint record specifies how the action scope must be restricted; and an escalation record identifies the manual sign-off required for execution.

The IEEC begins registering critical milestones: intent receipt, context provider queries, context snapshots, policy inputs, decision records, and escalation metrics. These records allow security engineers to debug policy sets, operators to analyze decisions, and compliance teams to verify governance efficacy.

Phase 2 deliverables include the unified policy input model, the policy engine adapter, context retrieval schemas, decision event schemas, the initial IEEC event store, and a comprehensive policy gap report. The policy gap report highlights actions lacking clear policies, missing context sources, or poorly defined escalation boundaries.

Shadow policy evaluation must remain highly visible, allowing operators to build confidence in the control plane's reasoning. Once the shadow decision metrics demonstrate high fidelity and zero false positives, selected low-risk workflows can transition to active enforcement.

11.5 Phase 3: Bounded Execution Contracts

Phase 3 transitions the control plane from advisory evaluations to active enforcement by binding selected low- and medium-risk workflows to strict execution contracts. The execution contract is the primary boundary of the ASCP, converting approved intents into cryptographic constraints that govern actual execution.

Premature Automation

Deployments must never transition directly from advisory models to unbounded autonomous execution. The execution contract serves as the necessary, deterministic boundary separating governance from execution.

An execution contract defines the approved action, target resource scope, parameter limits, temporal window, context assumptions, evidence obligations, rollback procedures, and revocation hooks. A vague contract is useless. The control plane must issue narrow, precise contracts, such as: "optimize resource allocations on database cluster X, restricted to CPU limits between 2 and 4 cores, valid for 15 minutes, contingent on database read-latency exceeding 200ms, emitting pre- and post-optimization latency telemetry."

Initial deployment targets focus on simple, reversible tasks: automated instance cleaning, non-production configuration updates, service ticket routing, sandboxed system scans, and bounded remediation scripts. These targets validate contract enforcement without exposing critical production states.

Execution adapters enforce contract boundaries directly at the target interface, blocking any API call or instruction that deviates from the approved parameters, targets, or timelines. This

moves governance from policy text to active runtime enforcement, preparing the system for integration with verifiable identity modules.

Phase 3 delivers the contract schemas, the contract issuer, execution adapter specifications, negative-enforcement test suites, and manual approval loops. Test suites must aggressively validate negative paths: expired contracts, mismatched target resources, unauthorized parameters, and missing evidence.

Enforcement rollout must be gradual, backed by continuous monitoring. Operators must have immediate override capabilities and escalation paths to handle edge cases where contracts are misaligned with shifting operational demands.

11.6 Phase 4: Proof-Derived Execution Identity

Phase 4 eliminates standing privileges for autonomous agents, replacing them with task-scoped, time-bounded execution identities derived dynamically from approved contracts. This introduces the Verifiable Agentic Infrastructure (VAI) pattern, ensuring that agents possess zero standing authority and assume only the minimal credentials required for a specific, contracted task.

Traditional IT architectures often assign broad, long-lived IAM roles and API keys to agent services, decoupling credentials from intent and policy. In an autonomous environment, this configuration is highly dangerous. A reasoning error or compromised prompt can translate broad standing credentials into systemic operational failures.

Phase 4 deploys the VAI identity broker, contract verification services, dynamic credential issuers, and short-lived session token generators. When an agent attempts an action, the identity broker verifies the corresponding execution contract and issues a transient, scoped credential—such as a single-use token or temporary IAM session—limited strictly to the approved scope and duration.

Implementation paths depend on the target environment, leveraging cloud security token services (STS), temporary service account impersonation, workload identity federation, or cryptographically signed capability tokens. Regardless of the technology stack, runtime authority remains a dynamic consequence of contract proof.

Phase 4 deliverables include the dynamic identity broker, resource credential mappings, rapid revocation pipelines, and privilege reduction matrices. Key operational metrics include the percentage of actions governed by dynamic identities, the reduction in active standing credentials, dynamic token lifespans, and dynamic authorization rejections.

Deploying dynamic identities requires precise mapping of credential scopes. Teams should begin with a single, well-understood workflow, progressively expanding the VAI pattern to cover broader operational domains as verification metrics stabilize.

11.7 Phase 5: Replay, Simulation, and Certification

Phase 5 leverages the IEEC to enable forensic replay, policy simulation, and formal governance certification. Replay converts audit logs into an active, continuous learning loop. Rather than simply logging events, the replay engine reconstructs decision paths to verify whether identical

inputs, policies, and contexts yield identical, deterministic outcomes.

The replay engine ingests the complete execution lifecycle: intents, context snapshots, policy versions, contracts, dynamic identities, execution telemetry, and verification metrics. This allows security teams to investigate operational anomalies, test policy changes against historical events, and generate comprehensive compliance reports.

Simulation scales this capability by testing governance structures against hypothetical contexts and policies. For example, database administrators can simulate how a grid-balancing agent behaves under degraded infrastructure dependencies, or test how new regulatory rules affect active administrative workflows before they are deployed to production.

Certification packages consolidate this evidence to support formal audits and compliance reviews, aligning with standards like the NIST AI Risk Management Framework [13] and Zero Trust Architecture [17]. These packages provide cryptographic proof that every autonomous change was authorized by an active policy, validated against real-time context, bound to a contract, and executed under a verified, short-lived identity.

Phase 5 deliverables include the replay engine, simulation harnesses, policy diff utilities, and automated compliance generators. Replay telemetry directly informs the optimization of policies, context providers, and dynamic execution constraints.

11.8 Phase 6: Protocol-Driven Software Admission

Phase 6 implements Protocol-Driven Development (PDD), governing the admission of AI-generated code, configurations, workflows, and policy modules before they enter production registries or deployment pipelines. Runtime governance is incomplete if autonomous systems can inject unverified code into the execution path.

The admission gate evaluates all AI-generated automation scripts, infrastructure-as-code templates, configuration patches, agent adapters, and workflow definitions. PDD enforces that no generated artifact is deployed simply because it compiles; it must cryptographically prove compliance with its designated protocol.

The PDD pipeline evaluates candidate artifacts against structural, behavioral, and operational invariants:

- **Structural invariants** verify interface signatures, syntax correctness, dependency constraints, and module boundaries.
- **Behavioral invariants** simulate execution within secure sandboxes to detect unauthorized state changes, security vulnerabilities, or logic errors.
- **Operational invariants** verify timeout limits, rollback capabilities, telemetry emission, and resource consumption boundaries.

Admission decisions yield distinct states: absolute admission, sandboxed execution only, constrained admission, or escalation to manual review. A rejected artifact represents a successful enforcement of the admission boundary, preventing unverified code from lingering in production repositories.

Phase 6 deliverables include the centralized protocol registry, the automated PDD pipeline, sandboxed simulation environments, and CI/CD integration plugins. Performance metrics track admission and rejection rates, protocol violation categories, escaped defect indices, and delivery velocity.

Runtime telemetry feeds back into the protocol registry. If an admitted script encounters operational errors, PDD updates the corresponding behavioral and operational invariants to prevent similar failures in future generations.

11.9 Phase 7: Institutional and National-Scale Expansion

Phase 7 scales the ASCP from bounded pilot environments to a federated, cross-domain governance fabric. At national or enterprise scale, the control plane functions as shared, strategic infrastructure, providing common schemas, evidence ledgers, and identity frameworks across distributed departments and business units.

Institutional Control

Scale requires more than technical integration; it requires absolute institutional ownership of policies, identities, evidence ledgers, and software protocols.

Enterprise expansion encompasses diverse operational domains: public administrative queues, multi-cloud platforms, security operations centers, cyber-physical municipal networks, and financial transaction systems. The architecture supports federated policy ownership, allowing individual domains to manage localized rules while adhering to unified intent schemas and evidence models.

Key architectural requirements at scale include federated policy synchronization, unified intent schemas, standardized environment adapters, high-availability evidence repositories, and role-based audit dashboards. The governance structure must prevent bottlenecks by automating routine, low-risk operations while routing ambiguous, high-impact intents to designated manual approval channels.

Expansion proceeds systematically based on domain maturity. Workflows featuring clean context streams, explicit policy coverages, and established verification paths are scaled first. The goal is not to maximize the volume of automated tasks, but to ensure that all consequential actions are validated, bounded, and auditable.

Decoupling the control plane from underlying technologies ensures strategic portability. Organizations can swap LLM providers, migrate cloud environments, or update database engines without redefining their fundamental governance boundaries.

11.10 Operating Model

Successful deployment depends on a clearly defined operating model. Autonomous governance requires designated organizational owners for policies, identities, evidence ledgers, escalations, and software protocols to ensure that control-plane telemetry drives continuous improvement.

These responsibilities are assigned to specific teams based on organizational structure. In public administration, policy authority resides with regulatory agencies and central digital transformation

Table 11.2: *Operating roles for governed autonomous execution.*

Role	Responsibility
Control-plane owner	Operates the governance substrate and integration architecture
Policy authority	Defines machine-enforceable policy and approval thresholds
Domain owner	Defines operational context, risk tiers, and escalation paths
Security/IAM team	Manages identity integration, credential scope, and revocation
Evidence and audit team	Maintains evidence schemas, replay, and audit reporting
Protocol owner	Defines PDD invariants for generated artifacts and components
AI platform team	Integrates models and agents through governed intent interfaces
Incident response team	Uses replay and evidence for after-action analysis

ministries. In commercial enterprises, it is shared between business units, platform engineering, and risk compliance offices. Regardless of the organizational layout, these roles must be explicitly defined rather than left implicit.

11.11 Success Metrics

ASCP maturity is measured by the proportion of automated actions that are justified, bounded, replayable, and governed. Success metrics track policy coverage, privilege minimization, evidence fidelity, safety enforcement, and PDD compliance.

Table 11.3: *Success metrics for deployment maturity.*

Metric Category	Example Metrics
Governance coverage	Intent capture rate, governed workflow count, policy evaluation coverage
Privilege reduction	Long-lived credential reduction, credential lifetime, proof-derived execution identity adoption
Evidence quality	IEEC completeness, replay success rate, audit evidence availability
Safety and risk	Unsafe actions blocked, constrained actions, escalations, contract violations prevented
Operational efficiency	Audit preparation time, incident reconstruction time, policy iteration time
Software admission	PDD admission rate, protocol violations, generated-artifact defect escape rate

Metrics must be evaluated contextually. A rising rejection rate may indicate improving policy enforcement or deteriorating agent behavior. A high escalation index could signal weak policy definition or incomplete context providers. Consequently, these metrics must be analyzed in partnership with domain owners rather than treated as isolated dashboards.

The definitive indicator of maturity is replay fidelity. If the system can consistently reconstruct exactly why authority was granted, confirm that execution remained within the contract, and verify all supporting evidence, the control plane is performing its core security function.

11.12 Common Failure Modes

The most severe deployment failure is treating an AI automation project as a governance architecture. Connecting agents directly to systems, adding basic telemetry logs, and building monitoring dashboards is not governed autonomy. True governance requires the active, deterministic restriction of the pathways through which actions become executable.

Deployment Failure Mode

The most common failure is mistaking an AI automation project for a governance architecture. The objective is not simply to automate more actions, but to govern the path by which actions become executable.

- **Standing Privilege Overlook:** Implementing execution contracts while leaving standing, broad-scoped IAM roles assigned to agents. If agents retain permanent credentials, contracts become mere documentation rather than active enforcement gates.
- **Treating Logs as Control:** Relying on post-execution audit logging without implementing pre-execution intent evaluation, contract binding, or dynamic identity derivation. Logs explain failures after they occur; they cannot prevent them.
- **Absolute Vendor Lock-in:** Hard-coding policy evaluation and identity management into a specific LLM platform or cloud service. This binds organizational sovereignty to proprietary vendor roadmaps and prevents technology portability.
- **Context Minimization Failures:** Transmitting sensitive operational context or database states directly to external reasoning models without sanitization or minimization, exposing the organization to data leaks.
- **Neglecting Replay Loops:** Accumulating vast volumes of IEEC evidence without deploying active replay engines to validate policies, investigate anomalies, and audit system state.
- **Treating PDD as Optional:** Implementing robust runtime governance while allowing unverified AI-generated code, adapters, and configuration files to bypass protocol-based admission checks.

11.13 Roadmap Summary

This phased roadmap provides a disciplined pathway from initial readiness assessments to federated, sovereign autonomy. Each phase yields concrete, testable artifacts that build the organizational confidence required to securely scale autonomous operations.

In initial deployments, Phases 0–2 can be rapidly completed within a single shadow-governance cycle. Production integrations for Phases 3–5 require deeper system access, as execution contracts, dynamic identities, and replay systems interface directly with live databases and networks. Finally, scaling Phases 6–7 demands organizational maturity as protocol-driven software admission and federated policies become standard components of enterprise compliance.

Table 11.4: *Phased deployment roadmap for the Autonomous State Control Plane.*

Phase	Primary Goal	Key Deliverable
0. Readiness Assessment	Discover autonomous action paths and risks	Action inventory and risk matrix
1. Intent Capture	Observe agent proposals in shadow mode	Intent gateway and shadow logs
2. Policy and Evidence	Evaluate intents and record decisions	Policy adapter and IEEC events
3. Execution Contracts	Create bounded contracts	Contract issuer and adapter interface
4. Execution Identity	Replace standing privilege with dynamic credentials	Identity broker and scoped credentials
5. Replay and Simulation	Reconstruct and test decisions	Replay engine and simulation harness
6. Protocol Admission	Govern generated artifacts before use	Protocol registry and admission pipeline
7. Scale Expansion	Extend governance across domains	Shared governance fabric and operating model

The deployment roadmap establishes a secure, structured framework for adopting governed autonomy. The final chapter outlines the collaborative steps required across research, engineering, and regulatory bodies to standardize these control-plane architectures for the autonomous era.

12 Call to Collaboration

Autonomous AI is an infrastructure governance challenge. As agentic systems move from advisory recommendations toward high-impact, real-world mutations, organizations need control planes that define how intent becomes authority, how authority translates into execution, and how execution produces evidence.

The transition from AI-assisted workflows to autonomous execution requires cooperative governance. No single model provider, cloud platform, government agency, enterprise, or research team can address the architectural complexity alone. Progress will depend not only on model capability, but on the control planes that make autonomous systems governable.

Governed Intelligence

The enduring strategic asset is not intelligence alone. It is governed intelligence.

Intent Standards → Policy Evaluation → Execution Contracts → Proof-Derived Execution Identity → Evidence Chains → Replay and Certification

The Autonomous State Control Plane (ASCP) reference architecture addresses this gap. It separates probabilistic reasoning from deterministic system execution, converts automated proposals into structured intents, derives runtime credentials from policy proofs, records the transaction lifecycle, enables replay verification, and gates AI-generated code through software protocols. The priority is to build control substrates that let institutions use model capability without sacrificing accountability.

12.1 Autonomy as a Shared Infrastructure Challenge

Autonomous AI creates a governance challenge across execution authority, policy enforcement, identity systems, evidence, replay verification, software admission, auditability, and state sovereignty. Cognitive models excel at processing unstructured text but are poorly suited to serve as authority gates. Cloud platforms offer mature runtime environments but usually lack the semantic context to verify whether an agent's proposed mutation is justified. Standard compliance audits demand operational logs but often miss the context snapshots, policy inputs, and execution identities required for replay verification.

Unchecked model capability creates operational risk, while inflexible governance limits useful automation. High-impact sectors, including municipal grids, public administration, financial ledgers, software supply chains, and industrial networks, need an architecture that balances both. Execution must remain governed even when cognitive reasoning is probabilistic. This doctrine is not anti-agentic; it recognizes that autonomous execution shifts the unit of risk from semantic error to unvalidated physical and digital state mutation.

Securing this ecosystem is a cooperative task:

- **Sovereign States** require strict execution boundaries to protect public services and national databases.
- **Enterprises** need risk-tiered controls to safely automate complex business workflows.
- **Infrastructure Providers** require standard APIs for identity and evidence integration to support bounded execution.
- **Researchers** can develop formal verification models and benchmarks to assess governance systems.
- **Open-Source Contributors** must produce reference implementations and standardized schemas.
- **Standards Bodies** must establish common formats for intents, contracts, evidence logs, and software admission.

12.2 Core Architectural Requirements

Rather than developing another agentic framework, the field needs the governance substrate beneath agentic execution. Standard agent frameworks coordinate tasks and invoke APIs, but they do not usually verify whether an action aligns with corporate policies, dependency states, or audit obligations. A secure control plane requires nine core components:

1. **Standardized Intent Interfaces:** Common formats for agents to declare objectives, actors, targets, parameters, risk tiers, and evidence requirements before execution.
2. **Trusted Context Providers:** Secure pipelines that aggregate real-time system state, dependency health, incident records, and data classifications to inform policy decisions.
3. **Pluggable Policy Adapters:** Interfaces that translate structured intents and context snapshots for diverse engines (such as Cedar or OPA/Rego) without forcing a single policy language.
4. **Execution Contract Formats:** Cryptographically signed agreements defining approved actions, parameter limits, temporal windows, rollback conditions, and evidence duties.
5. **Dynamic Identity Brokers:** Credential systems that convert execution contracts into short-lived, task-scoped tokens, reducing dependence on permanent credentials.
6. **Tamper-Evident Evidence Chains:** Registries logging the lifecycle (intent, context, policy, identity, execution, and verification) in a standardized format.
7. **Forensic Replay Engines:** Analytical tools to reconstruct decision paths and verify whether identical context inputs yield identical policy outcomes.
8. **Software Admission Gates:** Pipelines that audit AI-generated code and configurations against structural, behavioral, and operational invariants before deployment.

9. **Operator Override Mechanisms:** Bounded, manual escalation paths to resolve policy ambiguities and secure safety-critical anomalies.

12.3 OpenKedge as an Open Foundation

OpenKedge is an open-source reference architecture for standardized vocabularies and implementations around intent governance, execution contracts, dynamic identities, and protocol-driven software admission. The architecture is model-neutral, cloud-neutral, policy-pluggable, and adapter-oriented, allowing institutions to deploy and inspect these control boundaries across diverse operational environments.

Decoupling the control plane keeps adoption portable. Organizations can use OpenKedge to verify a clear sequence: model outputs remain unvalidated intents, contracts bind those intents to policy, credentials stay task-scoped, and execution is recorded for replay. These checks can remain independent of any single cloud provider or model vendor.

To maintain credibility, the project must emphasize executable software over abstract descriptions. Practical adoption depends on the continuous development of open-source SDKs, validation tools, reference adapters, identity brokers, and replay interfaces, backed by clear demonstration scenarios showing how the control plane handles allow, deny, escalate, and sandbox events.

12.4 Collaboration Across Stakeholders

Governing autonomous systems requires cooperation among model builders, infrastructure engineers, security practitioners, policy authors, and compliance officers. The control plane's effectiveness depends on integrating these perspectives into shared standards and interoperable implementations.

Table 12.1: *Collaboration areas for governed autonomous infrastructure.*

Stakeholder	Potential Contribution
Governments and national agencies	Policy guidelines, audit standards, sovereign boundaries, and escalation protocols
Cloud and infrastructure providers	Identity integrations, provider adapters, policy hooks, sandboxes, and evidence APIs
Enterprises and regulated industries	Production workflows, risk models, compliance requirements, and integration feedback
Academic researchers	Formal semantics, verification proofs, replay metrics, and governance benchmarks
Open-source contributors	SDKs, API adapters, replay dashboards, schemas, and developer tooling
Standards bodies	Portable intent formats, evidence schemas, risk taxonomies, and certification metrics

12.5 Research Agenda

As model capabilities scale, the research frontier shifts toward verifying the governance systems around autonomous agents. This agenda prioritizes six disciplines:

- **Formal Intent Semantics:** Establishing mathematical models to analyze intent equivalence, parameter nesting, scope limits, and parsing ambiguities.
- **Policy-Context Consistency:** Developing algorithms to verify context freshness, validate data sources, and resolve conflicting state snapshots across distributed databases.
- **Contract Constraint Proofs:** Designing techniques to verify that execution contracts restrict actual runtime API behaviors.
- **Proof-Derived Identity Models:** Creating cryptographic protocols that bind dynamic security tokens to intent schemas, ensuring that credentials cannot be hijacked or reused.
- **Fidelity Replay Semantics:** Refining deterministic simulation frameworks to reconstruct historical states and audit policy drift over time.
- **Cyber-Physical Governance:** Engineering safety interlocks, latency-tolerant bounds, and real-time rollbacks for autonomous physical systems.

12.6 Implementation Agenda

The implementation agenda focuses on concrete reference components that practitioners can integrate and evaluate:

- **Intent SDK:** Multi-language client libraries, validators, and schemas enabling applications to submit structured intent objects rather than raw execution commands.
- **OpenKedge Ingestion Engine:** High-throughput services to capture intents, query context providers, build policy inputs, normalise decisions, and issue contracts.
- **Unified Policy Adapters:** Production-ready wrappers for major policy engines (such as Cedar and OPA/Rego) and legacy approval systems.
- **Workload Identity Brokers:** Integration modules that translate approved contracts into short-lived IAM credentials using cloud token services and workloads federation.
- **IEEC Store and Replay Dashboard:** A tamper-evident database and visual dashboard allowing operators to query, simulate, and replay decision paths.
- **PDD CI/CD Pipelines:** Automated test suites, sandbox orchestrators, and static code analyzers to gate generated adapters and configurations.

12.7 Standards and Governance Agenda

Without interoperable, open standards, organizations will deploy fragmented, proprietary boundaries. This fragmentation hinders audits, prevents portable compliance packaging, and complicates certification. The industry requires common formats for intents, contracts, evidence event logs, and software admission proofs.

Fragmented Governance

Without interoperable governance artifacts, each autonomous system will reinvent its own accountability boundary, making audit, certification, replay, and institutional trust harder to scale.

Standardization must focus on defining portable accountability primitives—such as schemas for intents, contracts, and evidence logs—while leaving organizations free to implement custom policies and context providers. Establishing these primitives also improves technology procurement, allowing enterprises and governments to verify that third-party agent platforms natively support execution contracts, dynamic identities, and replay audits.

12.8 Toward Sovereign and Governed AI

The sovereign AI thesis remains clear: states, enterprises, and regulated institutions can use powerful global reasoning models while maintaining local control over execution boundaries. Sovereignty in the agentic era is defined not by access to cognitive models alone, but by ownership of the control plane that regulates how those models act.

Sovereign AI

Models may be global. Execution authority must remain sovereign.

This operational boundary matters across high-consequence domains. Utilities require control planes for grid balancing; financial institutions require bounded contracts to prevent capital leakage; cloud engineers require proof-derived identities; and compliance teams require durable evidence. The ASCP renders these needs composable: SAL secures the cognitive interface, OpenKedge evaluates intent, VAI enforces dynamic least privilege, the IEEC captures evidence, replay supports policy improvement, and PDD secures the software supply chain.

12.9 Closing Statement

The autonomous era will not be governed by raw cognitive capability alone. It will depend on systems that translate that capability into accountable, policy-compliant action. The immediate task is to deploy governance infrastructure before autonomous execution becomes deeply embedded in digital and physical networks.

The Autonomous State Control Plane provides a reference blueprint for this work and an invitation to build the missing governance substrate: interfaces, policy adapters, execution con-

tracts, dynamic identities, evidence stores, and software protocols. The practical aim is to make autonomous action governable before it becomes routine.

A Formal Models and Invariants

The Autonomous State Control Plane can be modeled as a governed transition system in which probabilistic reasoning produces candidate intents, but only validated intents may induce bounded execution events. This appendix defines the core objects used by the architecture and states the invariants that should hold across implementations.

Formalization Goal

The goal of the formal model is not to prescribe one implementation, but to define the invariants that any trustworthy implementation must preserve.

To complement the systems and institutional descriptions in the main text, this appendix establishes the formal mathematical skeleton of the control plane. This model does not claim to prove absolute safety or dictate specific verification engines; instead, it establishes the minimum set of relational invariants necessary to govern, rather than merely automate, autonomous execution.

A.1 Notation

Table A.1 summarizes the symbols used throughout the appendix.

Table A.1: *Core notation for the Autonomous State Control Plane.*

Symbol	Meaning
A	Actor, agent, or requesting principal
M	Reasoning model or external agentic system
I	Structured intent proposed for governance
C	Context snapshot used for policy evaluation
Π	Policy set or policy evaluation function
D	Governance decision
K	Execution contract
EID	Proof-derived execution identity
X	Execution event or bounded state mutation
$\mathcal{E}$	Evidence chain
$\mathcal{P}$	Protocol in Protocol-Driven Development
$\mathcal{S}$	Structural invariants
$\mathcal{B}$	Behavioral invariants
$\mathcal{O}$	Operational invariants
t	Time or validity interval

These symbols represent logical abstractions rather than specific implementation constraints. An execution contract K, for instance, may be encoded as a cryptographically signed token, a database record, a policy-engine artifact, or a capability descriptor. The invariant lies not in the

concrete representation, but in the strict confinement of execution authority to the bounds of the approved contract.

A.2 System Model

At a high level, the Autonomous State Control Plane can be represented as:

$$\Sigma = (S, I, C, \Pi, D, K, EID, X, \mathcal{E})$$

where S is system state, I is a structured intent, C is a context snapshot, Π is a policy set or evaluation function, D is a governance decision, K is an execution contract, EID is proof-derived execution identity, X is an execution event, and $\mathcal{E}$ is the evidence chain.

$$I \to C_t \to D \to K \to EID \to X \to \mathcal{E} \to Replay$$

A governed transition can be written as:

$$S_t \xrightarrow{I,C,\Pi,D,K,EID} S_{t+1}$$

Admissibility of a state transition requires prior intent evaluation, a formal policy decision, contract issuance, derivative identity binding, and non-repudiable evidence recording. Crucially, the reasoning model M lacks the authority to induce state transitions directly. While M may generate proposals, plans, or candidate configurations, it cannot establish authorization.

This distinction formalizes the control-plane doctrine: M may influence the candidate intent I, but it must never induce execution X directly. State mutation occurs exclusively through the governed transition relation.

This architecture isolates probabilistic uncertainty from deterministic authority. While the reasoning layer proposing I may be non-deterministic, heuristic-driven, or external, the governance relation remains deterministic: given the same intent, context snapshot, policy definition, and contract parameters, the control plane must reconstruct an identical decision. Conformant implementations need not employ identical engines, but they must guarantee the reproducibility of the decision path.

Furthermore, the transition system treats non-execution outcomes (such as denial, escalation, simulation, or deferral) as first-class events. Recording blocked or constrained candidate transitions is essential for system auditing, policy refinement, and continuous context adjustments. Consequently, the transition system tracks both state-changing executions and evidence-updating rejections.

A.2.1 Compositionality of Governed Transitions

Governed workflows often contain multiple steps. A single high-level task may decompose into sub-intents, delegated actions, tool calls, or nested agentic loops. A governed transition can be represented as:

$$g_i : S_i \to S_{i+1}$$

A composed workflow is then:

$$G = g_n \circ \cdots \circ g_2 \circ g_1$$

Let $Gov(g_i) = true$ denote that transition g_i satisfies the governance requirements of this appendix: intent precedes execution, policy precedes contract, identity is no broader than contract, execution satisfies contract, and evidence is complete. A governance-preserving composition can be stated as:

$$\forall i,\ Gov(g_i) \wedge ComposeSafe(g_1, \ldots, g_n) \Rightarrow Gov(G)$$

A composed workflow preserves governance if and only if every constituent transition is governed and the composition prevents authorization creep, evidence loss, or policy evasion. Composition must not become a mechanism for policy bypass. This compositionality requires explicit sub-intents, distinct contracts for delegated actions, nested evidence chains that preserve lineage, and explicit policy controls governing delegation.

For foundational approaches to these formalizations, we refer readers to categorical semantics [16], compositional verification [3], and denotational semantics [19].

A.3 Intent Model

An intent is the boundary artifact between reasoning and governance. It captures a claim that a state transition should be considered; it is not itself authority. A compact intent model is:

$$I = (a, o, r, q, j, \alpha)$$

where a is the actor or requester, o is the objective, r is the requested action, q is the target scope, j is the justification, and α is the set of assumptions or constraints proposed by the reasoning layer.

The intent object decouples semantic intent from executable authority. Unlike raw tool invocations, an intent encapsulates the objective, contextual assumptions, risk categorization, and institutional justification required for governance.

Define:

$$ValidIntent(I) = true$$

only if the intent is structurally well-formed and contains sufficient fields for governance evaluation. Structural validity does not imply approval. It means the control plane has enough information to proceed to context acquisition and policy evaluation. A syntactically valid intent may still be denied, constrained, escalated, deferred, or returned for additional context.

A.4 Context and Policy

Policy evaluation depends on context. A context snapshot can be modeled as:

$$C_t = snapshot(S_t, R, t)$$

where S_t is current system state, R is the relevant resource or domain scope, and t is the time of evaluation. The context snapshot may include system health, ownership metadata, dependency state, user or workflow state, risk signals, incident status, change windows, or domain-specific constraints.

Policy evaluation is written:

$$D = \Pi(I, C_t)$$

where:

$$D \in \{\texttt{allow}, \texttt{deny}, \texttt{constrain}, \texttt{escalate}, \texttt{simulate}, \texttt{defer}, \texttt{request\_context}\}$$

Policy decisions are intrinsically bound to the context snapshot under which they are evaluated. The same intent may yield divergent decisions as context evolves—for example, permitting resource termination when a node is out of rotation, but denying it when serving active traffic. To maintain accountability, the evidence chain must store the complete context snapshot and the exact policy version used during evaluation.

A.5 Execution Contracts

An execution contract is the bounded artifact that connects governance to runtime authority. It can be modeled as:

$$K = contract(I, C_t, D, \beta, \tau)$$

where β denotes execution bounds and τ denotes the validity interval. The contract specifies the allowed operation, target resources, parameter bounds, time bounds, context assumptions, evidence requirements, and revocation conditions.

The execution contract translates policy decisions into enforceable bounds. A decision that denies execution must never yield a contract. A decision that allows or constrains execution generates a contract K defining the parameters, resources, and temporal boundaries of permissible operations.

Contract compliance can be written:

$$X \models K$$

meaning that the observed execution event satisfies the bounds specified by the contract. Admissibility requires that the observed execution event strictly satisfies the contract ($X \models K$). Any deviation—such as exceeding a traffic-shifting percentage, targeting an unapproved resource, or executing outside the temporal window—constitutes a violation that must be blocked by runtime enforcement. The contract is a dynamic boundary enforced at runtime, not mere retrospective documentation.

A.6 Proof-Derived Execution Identity

Execution identity is derived from proof artifacts rather than standing entitlement. A compact model is:

$$EID = f(I, C_t, D, K, \tau)$$

where identity is computed from validated intent, context snapshot, policy decision, execution contract, and validity interval. The identity may be implemented as a short-lived credential, signed capability, scoped token, session policy, workload identity claim, or adapter-enforced authorization. The implementation may vary, but the authority must be derived from the approved contract.

Identity validity can be expressed as:

$$Valid(EID, K, t) \iff t \in \tau \land EID \preceq K$$

where $EID \preceq K$ means that the authority granted by the identity is no broader than the authority specified by the contract.

The inequality $EID \preceq K$ formalizes the principle of proof-derived least privilege: the issued execution identity must never convey authority exceeding the limits set by the contract.

Execution identity must be transient, task-scoped, revocable, and tightly coupled to the evidence chain. If an identity outlives its contract, targets unauthorized resources, or permits unapproved operations, the system collapses back into a model of standing privilege.

A.7 Evidence Chain

The evidence chain records the governance path. A minimal sequence is:

$$\mathcal{E} = \langle e_I, e_C, e_D, e_K, e_{EID}, e_X, e_V \rangle$$

where e_I is the intent event, e_C is the context snapshot event, e_D is the policy decision event, e_K is the execution contract event, e_{EID} is the identity issuance event, e_X is the execution event, and e_V is the verification event.

Define evidence completeness:

$$Complete(\mathcal{E}, X) = true$$

if the chain contains all evidence required for the executed mutation. Evidence completeness ensures that an independent auditor can reconstruct the entire authorization lineage. This completeness must extend to rejections, escalations, and simulations, as these negative paths contain vital information regarding policy efficacy and system safety.

While concrete implementations may utilize append-only logs, cryptographic signatures, or distributed ledgers to guarantee non-repudiation, this model remains mechanism-neutral. The invariant requires only that the recorded evidence is sufficient to verify, audit, and reconstruct the decision path.

A.8 Replay Semantics

Replay reconstructs the governance decision path from recorded evidence. A simple replay function is:

$$Replay(\mathcal{E}) \rightarrow D'$$

A replay is consistent if:

$$D' = D$$

under the recorded intent, context snapshot, and policy version. Replayability guarantees that a past governance decision can be deterministically reproduced from the evidence chain under the corresponding context snapshot and policy version.

Replay does not require the reasoning model to reproduce its internal probabilistic generation paths. Instead, replay reconstructs the deterministic governance boundary that validated and authorized the execution. Replayability isolates the probabilistic reasoning layer from the deterministic control plane, focusing exclusively on the structured intent, context snapshot, policy version, and contract parameters.

A.9 Protocol-Driven Development Model

Protocol-Driven Development governs generated software and system components before they participate in operational execution. The protocol is modeled as:

$$\mathcal{P} = (\mathcal{S}, \mathcal{B}, \mathcal{O})$$

where $\mathcal{S}$ denotes structural invariants, $\mathcal{B}$ denotes behavioral invariants, and $\mathcal{O}$ denotes operational invariants.

An implementation is admissible when:

$$\text{impl} \models \mathcal{P}$$

meaning that the implementation satisfies the structural, behavioral, and operational invariants required by the protocol. Under Protocol-Driven Development, source code is treated as a candidate realization of a protocol, rather than the primary source of authority.

A.9.1 Type-Theoretic View of Protocol Admission

A type-theoretic interpretation of PDD treats protocol admission as the judgment that a candidate implementation inhabits the admissible space defined by structural, behavioral, and operational invariants.

$$\text{impl} : \mathcal{P} \quad \Rightarrow \quad Admissible(\text{impl})$$

where impl : $\mathcal{P}$ represents the judgment that the implementation code inhabits the space defined by the protocol.

Here, structural invariants ($\mathcal{S}$) map to interface type constraints, behavioral invariants ($\mathcal{B}$) map to refinement types or behavioral contracts, and operational invariants ($\mathcal{O}$) map to effect types and resource bounds. While advanced type systems (such as refinement, dependent, or effect types) can statically verify some of these properties, they complement rather than replace sandboxing, runtime simulation, and audit capabilities.

Advanced formalizations of the symbolic layer can draw upon foundational work in refinement types [4], dependent types [20], and typed effect systems [12].

Admission may be verified through static analysis, type checking, schema validation, property-based testing, sandboxed execution, simulation, or manual review. The critical requirement is that a generated artifact only transitions to operational status when explicit evidence demonstrates conformity to the governing protocol.

A.10 Core Safety Invariants

The following invariants formalize the core constraints of governed execution and constitute the implementation compliance criteria.

Invariant 1 – No Direct Reasoning-to-Execution.

$$M \not\rightarrow X$$

A reasoning model must never induce a state mutation directly. Reasoning models generate proposals and candidate actions; execution authority is established solely by the control plane.

Invariant 2 – Intent Precedes Execution.

$$X \Rightarrow \exists I$$

Every execution event must map to a prior structured intent. Unattributed or direct execution violates the governance boundary.

Invariant 3 – Policy Decision Precedes Contract.

$$K \Rightarrow \exists D = \Pi(I, C_t)$$

Execution contracts must derive from a deterministic policy decision evaluated against the active context. A contract generated without a corresponding decision has no standing authority.

Invariant 4 – Identity Is No Broader Than Contract.

$$EID \preceq K$$

The authority granted by the proof-derived execution identity must never exceed the bounds of the active contract, enforcing runtime least privilege.

Invariant 5 – Execution Must Satisfy Contract.

$$X \models K$$

Observed execution must conform strictly to the active contract. Runtime enforcement must block any out-of-bounds mutation.

Invariant 6 – Evidence Completeness.

$$X \Rightarrow Complete(\mathcal{E}, X)$$

Every execution event must produce an evidence chain sufficient for complete decision replay and audit, preserving the authorization path.

Invariant 7 – Replayability.

$$Replay(\mathcal{E}) = D$$

The original governance decision must be reconstructable from the evidence chain using the recorded context snapshot and policy version.

Invariant 8 – Protocol Admission.

$$Operational(\text{impl}) \Rightarrow \text{impl} \models \mathcal{P}$$

A generated artifact transitions to operational status if and only if it is formally admitted under its governing protocol.

Table A.2: *Core safety invariants.*

Invariant	Meaning
No direct reasoning-to-execution	Models may propose, but cannot directly mutate governed systems
Intent precedes execution	Every execution event is linked to a structured intent
Policy precedes contract	Contracts are derived from policy decisions over context
Identity no broader than contract	Runtime authority cannot exceed approved execution bounds
Execution satisfies contract	Observed execution remains within approved scope
Evidence completeness	Execution has sufficient evidence for replay and audit
Replayability	Governance decisions can be reconstructed from evidence
Protocol admission	Generated artifacts are operational only after satisfying protocol invariants

A.11 Discussion

This formal model remains strictly implementation-neutral. The logical components—such as contracts, decisions, and evidence—can be instantiated across diverse environments, policy engines, and identity layers. For example, one deployment may represent contracts as cryptographically signed capability tokens, while another may use database-backed records enforced by resource adapters. Similarly, policy evaluation may utilize languages like Cedar, OPA/Rego, or bespoke institutional workflow engines.

The core invariants represent portable governance obligations that survive changes in underlying infrastructure. Whether deployed on domestic, global, sovereign, public, or hybrid clouds, the control-plane properties remain identical. The appendix serves as a formal interface contract between architectural design and concrete implementations. It acknowledges the realities of engineering systems: context snapshots may be incomplete, reasoning models will remain probabilistic, and policy configurations will evolve. The control plane handles this incomplete information through deterministic mechanisms like escalation, context requests, or constraints.

Future formalizations can extend this model by defining equivalence relations over intents, refinement relations over contracts, temporal decay models for context, and composition rules for multi-agent validation. Regardless of these extensions, the fundamental flow of authority remains invariant: reasoning models propose, policies decide, contracts bound, identities authorize, execution emits evidence, and replay enforces accountability.

The specific mechanisms may vary across institutions and infrastructures, but the invariants should remain stable: intent before execution, proof before privilege, evidence before trust, and protocol before implementation.

B Threat Model

Conventional security models prioritize preventing unauthorized access; governed autonomous execution must address a more insidious threat: authorized-looking execution derived from untrusted, manipulated, or semantically unsafe reasoning. A conventional security system asks whether a principal is authenticated and permitted to invoke an operation. Governed autonomous execution, by contrast, must verify whether a machine-generated intent is legitimate, contextually safe, bounded by policy, authorized by proof, and replayable after the fact.

Threat Model Principle

The control plane does not assume that the reasoning layer is malicious. It assumes something broader and more useful for security design: the reasoning layer is not authoritative.

This appendix establishes a threat model for the Autonomous State Control Plane. It maps the specific threat vectors arising when AI agents, external reasoning models, generated code, and dynamic policy engines interface with real-world infrastructure. Rather than cataloging generic AI risks, this model covers the transition path from initial reasoning to intent, policy, contract, identity, execution, evidence, replay, and protocol admission.

B.1 Scope and Assumptions

This threat model covers threats to AI-generated intent, the reasoning-to-execution boundary, context acquisition, policy evaluation, execution contracts, proof-derived execution identity, runtime execution adapters, evidence chains, replay and simulation, and generated-code admission. It supports the architecture described by Sovereign Agentic Loops, OpenKedge intent governance, Verifiable Agentic Infrastructure, Intent-to-Execution Evidence Chains, Protocol-Driven Development, replay, simulation, and audit.

Autonomy represents an authority-amplification surface: minor reasoning discrepancies, prompt injections, stale context inputs, or ambiguous contracts can propagate into material operational changes if model outputs translate directly into system execution. The control plane reduces this risk by ensuring that the reasoning layer generates proposals rather than authoritative actions.

This security framework operates under the following axioms: AI reasoning is inherently untrusted, external models reside outside the sovereign execution boundary, agents may be compromised or confused, and human operators are fallible. The architecture does not attempt to enforce perfect reasoning; instead, it confines the execution blast radius by enforcing deterministic boundaries external to the model.

This model does not guarantee model correctness, promise perfect prompt injection detection, or replace foundational cloud security. It defines the threats emerging at the intersection of autonomous reasoning and system mutation, and identifies architectural controls intended to prevent

reasoning failures from becoming unbounded execution authority.

The primary assets requiring protection include sovereign execution authority, system state, policy definitions, execution contracts, short-lived credentials, and tamper-evident evidence logs. The core of this defense lies in protecting the authority boundary itself; flawed reasoning remains contained when the transition interface prevents unauthorized execution, maintains context freshness, and retains replayable evidence.

Table B.1: *Protected assets in governed autonomous execution.*

Asset	Protection Goal
Execution authority	Prevent unauthorized or unjustified real-world mutation
Context data	Limit disclosure and prevent stale or manipulated decisions
Policy definitions	Preserve institutional rules and approval boundaries
Execution contracts	Ensure approved bounds cannot be widened or forged
Execution identity	Prevent misuse, reuse, or privilege expansion
Evidence chain	Preserve auditability, replayability, and accountability
Generated artifacts	Prevent unsafe code from entering operational workflows
Human approval authority	Prevent social, procedural, or system-level bypass

The protected assets are interdependent. If context can be manipulated, policy decisions may be wrong. If contracts can be widened, identity may become overbroad. If identity is reusable, runtime execution may escape its task. If evidence can be omitted, replay cannot establish accountability. The control plane therefore treats the full governance path as the security surface.

B.2 Actors and Trust Boundaries

We model the system across multiple actors and distinct trust boundaries. The relevant actors include the external model provider, reasoning agents, user requestors, agent runtimes, intent gateways, context providers, policy engines, governance brokers, execution adapters, and evidence stores.

The model defines multiple trust boundaries:

- **Reasoning boundary:** external model output enters as a non-authoritative proposal.
- **Intent boundary:** model output becomes structured intent.
- **Policy boundary:** intent is evaluated under local policy and context.
- **Identity boundary:** execution authority is created.
- **Execution boundary:** actions affect real systems.
- **Evidence boundary:** events are recorded for replay and audit.
- **Protocol admission boundary:** generated artifacts become eligible for operational use.

Reasoning Boundary → Intent Boundary → Policy Boundary → Identity Boundary → Execution Boundary → Evidence Boundary

The security of this architecture does not depend on agent compliance or model-level alignment. Instead, the control-plane components (the runtime adapter, identity broker, policy engine, and evidence store) must independently enforce boundaries, even when presented with a well-formed but unsafe intent proposal.

B.3 Threat Category 1: Reasoning-Layer Threats

Reasoning-layer threats emerge from prompt injections, tool-use manipulation, instruction hierarchy confusion, and hallucinated justifications. Sovereign Agentic Loops mitigate these vectors by treating all model outputs as non-authoritative proposals. Obfuscation membranes limit context exposure, structured intent validation sanitizes outputs, and policy evaluation is decoupled from the inference runtime. Model-neutral governance prevents single-provider dependencies from compromising the authorization path.

B.4 Threat Category 2: Intent-Layer Threats

Intent-layer threats arise when model outputs cross the governance boundary as candidate intents, manifesting as malformed intents, smuggled actions, scope inflation, or replayed requests. To neutralize these threats, the control plane enforces strict schema validation, risk categorization, temporal nonces, and context freshness verification before generating any execution contract.

The intent layer must reject ambiguity: if an objective is unclear, the scope is overbroad, or the requester is unauthenticated, the control plane escalates, constrains, or denies the intent rather than converting it into execution authority.

B.5 Threat Category 3: Context and Policy Threats

Context and policy threats occur when governance decisions rely on stale, manipulated, or inconsistent inputs. Mitigating these vectors requires context provenance tracking, narrow freshness windows, policy versioning, and deny-by-default behavior on missing context. High-risk intents require real-time context revalidation immediately preceding execution.

B.6 Threat Category 4: Execution Contract Threats

Execution contract threats involve forgery, contract widening, or ambiguity. Because the contract constitutes the primary enforcement boundary, it must be cryptographically signed, immutable, and parameterized with explicit resource bounds and expiration times. If an action cannot be deterministically validated against the contract, the runtime adapter must block execution.

B.7 Threat Category 5: Execution Identity Threats

Execution identity threats attempt to exploit standing privileges, leaked credentials, or long-lived tokens. The identity broker must enforce proof-derived workload identity, issuing short-lived, task-

scoped, and non-reusable tokens that are strictly bounded by the execution contract ($EID \preceq K$).

Standing Privilege

Standing privilege turns an agent error into reusable authority. Proof-derived execution identity limits authority to the validated intent, contract, and time window.

The identity broker must fail closed: any validation failure regarding the contract, decision, context, or evidence requirements must prevent authority issuance.

B.8 Threat Category 6: Runtime Execution Threats

Runtime execution threats target the physical mutation interface via adapter bypass, parameter substitution, or race conditions. The execution adapter constitutes a critical element of the trusted computing base (TCB); it must verify the contract at the point of execution, fail closed, enforce atomic rollback, and emit detailed runtime evidence.

B.9 Threat Category 7: Evidence and Replay Threats

Evidence and replay threats aim to obscure accountability through omission, tampering, or selective logging. To support compliance in high-consequence environments, the system should employ append-only, hash-chained logs that record rejections and escalations alongside successful executions, supporting replayability.

B.10 Threat Category 8: Generated Software Threats

Generated software threats occur when AI-synthesized adapters, policy modules, or tools introduce malicious behavior or subtle bugs. Protocol-Driven Development (PDD) mitigates these threats by treating code as a candidate artifact. Artifacts are admitted to operational environments only after verifying structural, behavioral, and operational invariants within sandbox and property-testing pipelines.

Generated adapters and policy modules deserve special scrutiny because they participate directly in the active governance path. An unsafe adapter or compromised policy module can completely bypass runtime contract enforcement.

B.11 Mitigation Matrix

Table B.2 summarizes the primary architectural mitigations for each threat category.

The mitigations must be implemented as layered controls. A single mechanism is insufficient: intent validation without policy evaluation is incomplete, and contracts without proof-derived execution identity are vulnerable to standing privilege bypass. Evidence without replay remains mere logging, and PDD without runtime enforcement cannot govern post-admission execution.

Table B.2: *Threat categories and primary architectural mitigations.*

Threat Category	Primary Mitigations
Reasoning-layer threats	SAL, obfuscation membrane, intent isolation, no direct execution
Intent-layer threats	Intent schema validation, scope checks, risk classification, expiration
Context and policy threats	Context provenance, freshness checks, policy versioning, replay
Execution contract threats	Signed contracts, narrow bounds, expiration, revocation conditions
Execution identity threats	Proof-derived execution identity, short-lived credentials, no broader than contract
Runtime execution threats	Adapter enforcement, contract verification, fail-closed behavior
Evidence and replay threats	Append-only evidence, correlation ids, completeness checks, replay tests
Generated software threats	PDD, invariant checks, sandboxing, admission evidence, CI/CD gates

B.12 Residual Risks

Residual risks, including misconfigured policies, human operator errors, insider threats, and physical infrastructure compromise, must be managed through defense-in-depth, recurring red-team exercises, and periodic replay drills. Mature deployments should treat threat modeling as a continuous feedback loop, using incident evidence and validation failures to refine policy and adapter designs.

High-consequence domains must implement domain-specific safety cases, continuous system simulations, and clear manual escalation pathways. The ultimate objective of threat modeling is not to claim absolute security, but to delineate where authority must be bounded, where evidence must be produced, and where human sovereignty must be explicitly retained.

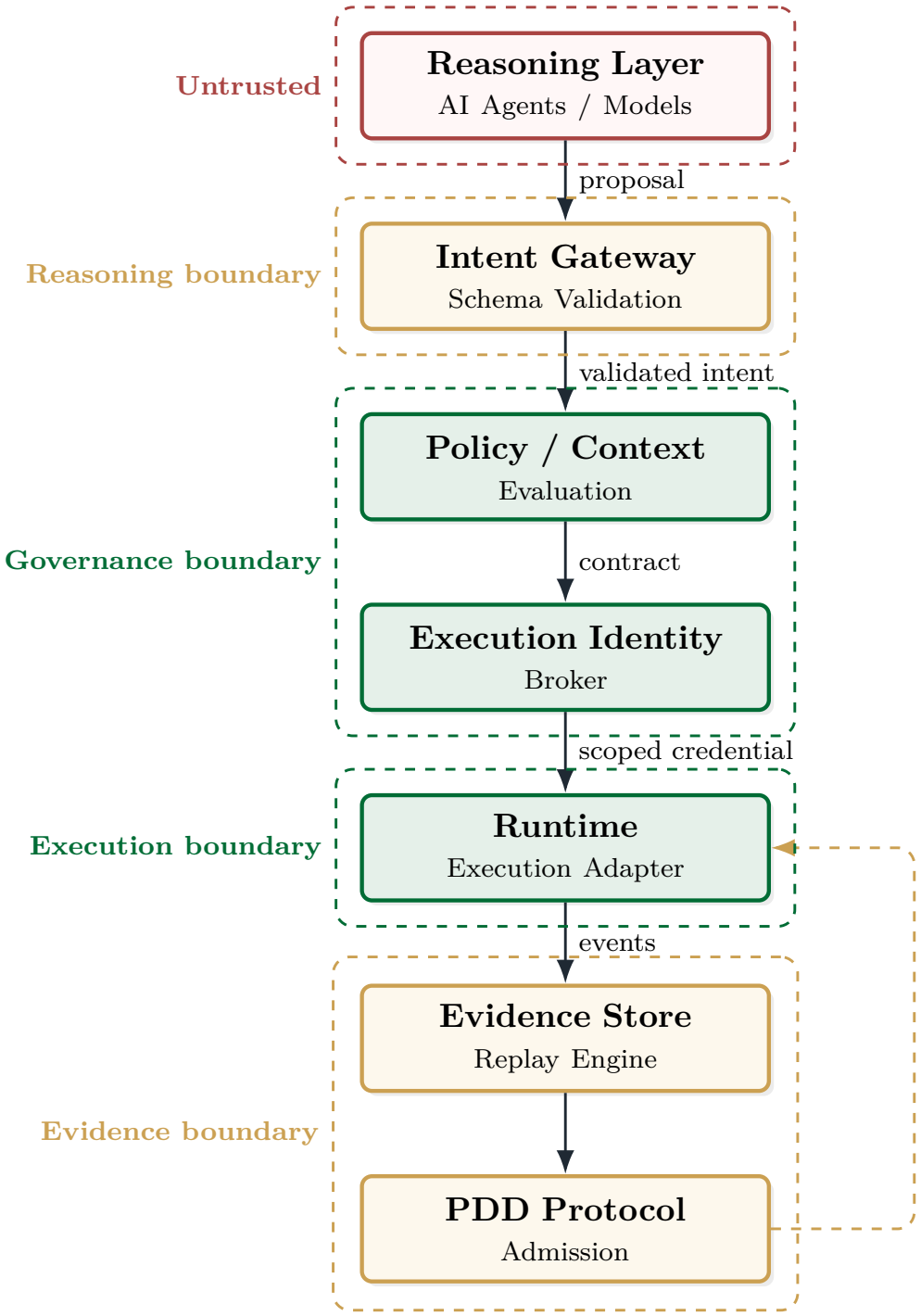


Figure B.1: *Trust boundaries in the Autonomous State Control Plane threat model.*

C Reference Implementation

Implementing the Autonomous State Control Plane as a modular, adapter-based governance runtime establishes a control boundary between AI-generated intent and domain-specific execution systems. This reference implementation makes the architecture executable while decoupling core governance from specific model providers, policy engines, or cloud backends.

Implementation Doctrine

OpenKedge should provide the governance substrate, not replace every model, policy engine, identity system, or cloud API.

This appendix provides a reference implementation blueprint for OpenKedge and related components. Rather than serving as an exhaustive operations manual, it defines the core runtime interfaces, TypeScript specifications, adapter boundaries, evidence structures, and deployment topologies required to translate these architectural concepts into functional code.

C.1 Implementation Goals

The reference implementation aligns the main runtime responsibilities: agents propose intents, OpenKedge evaluates them against policy, adapters enforce execution contracts, and evidence is recorded automatically. The system governs structured intents rather than dynamic prompts; while prompts direct probabilistic reasoning, they are unsuitable as stable governance artifacts. The primary runtime primitive must be a structured, verifiable intent.

For compatibility, the control plane uses established policy engines rather than introducing a proprietary policy language. A unified policy-engine interface integrates with engine runtimes such as Cedar and OPA/Rego, alongside custom or legacy validators. OpenKedge normalizes intents and context snapshots into policy-evaluable inputs, translating engine responses into deterministic decisions.

The architecture accommodates multiple operational domains. Cloud infrastructure management serves as an ideal baseline deployment due to its mature identity frameworks and deterministic resource APIs. The same interface boundaries govern Kubernetes resources, workflow engines, database migrations, serverless functions, case management systems, and smart-city control adapters.

Every decision path, including denials, constraints, simulations, and escalations, should emit evidence. Negative paths matter for auditability because they show active security enforcement.

Replayability requires that past decisions be reconstructable from the evidence chain regardless of inference non-determinism. Replay verifies the intent, context snapshot, policy decision, contract parameters, and derivative identity from the immutable log, bypassing the need to invoke the original reasoning model.

Execution authority relies on short-lived contracts. Rather than granting agents standing credentials, the runtime generates task-specific execution contracts, derives scoped identity tokens, and requires adapters to validate these contracts before executing mutations. Conforming implementations remain modular, enabling organizations to deploy local, private, or sovereign nodes without external SaaS dependencies.

C.2 Component Architecture

The reference architecture uses component boundaries to enforce modular decoupling. While the reasoning client generates candidate plans, it receives no direct execution authority. The 'IntentTranslator' processes natural language or model outputs to produce an 'IntentCandidate' which the 'IntentGateway' then validates and normalizes before evaluation. The 'IntentTranslator' resides outside the trusted computing base (TCB); its output remains untrusted until validated.

The orchestration logic resides in the governance engine, which coordinates context acquisition, policy evaluation, risk classification, and contract generation. The contract issuer produces a signed, bounded execution contract that defines the parameters and validity window of the operation. The identity broker then transforms the contract into a short-lived execution identity, which the execution adapter consumes to perform the mutation. All transitions emit structured evidence to the evidence store for verification.

Agent → Intent Translator → Intent Gateway → Context Providers → Policy Adapter → Governance Engine → Contract Issuer → Identity Broker → Execution Adapter → Evidence Store → Replay

This component model should remain small enough for an open-source implementation to be understandable, testable, and extensible. The goal is not to build a monolith. The goal is to define a stable governance path from proposed intent to bounded execution and replayable evidence.

C.3 Core Interfaces

TypeScript interfaces formalize these structural boundaries: translation must not be confused with validation, intent is decoupled from context, context remains separate from policy, and contracts bind derivative execution identity.

Core TypeScript interface families should cover intent translators, intents, context providers, policy engines, governance decisions, execution contracts, identity brokers, execution identities, execution adapters, evidence events, evidence stores, replay engines, protocols, and admission results. The concrete names are illustrative, but the implementation should preserve the separation of responsibility: translation is not validation, intent is not context, context is not policy, policy is not contract, contract is not identity, identity is not execution, and execution is not evidence.

Listing C.1: *Core OpenKedge interface families.*

```
interface IntentTranslator {
  translate(input: ReasoningOutput): Promise<IntentCandidate>
}
interface Intent { ... }
```

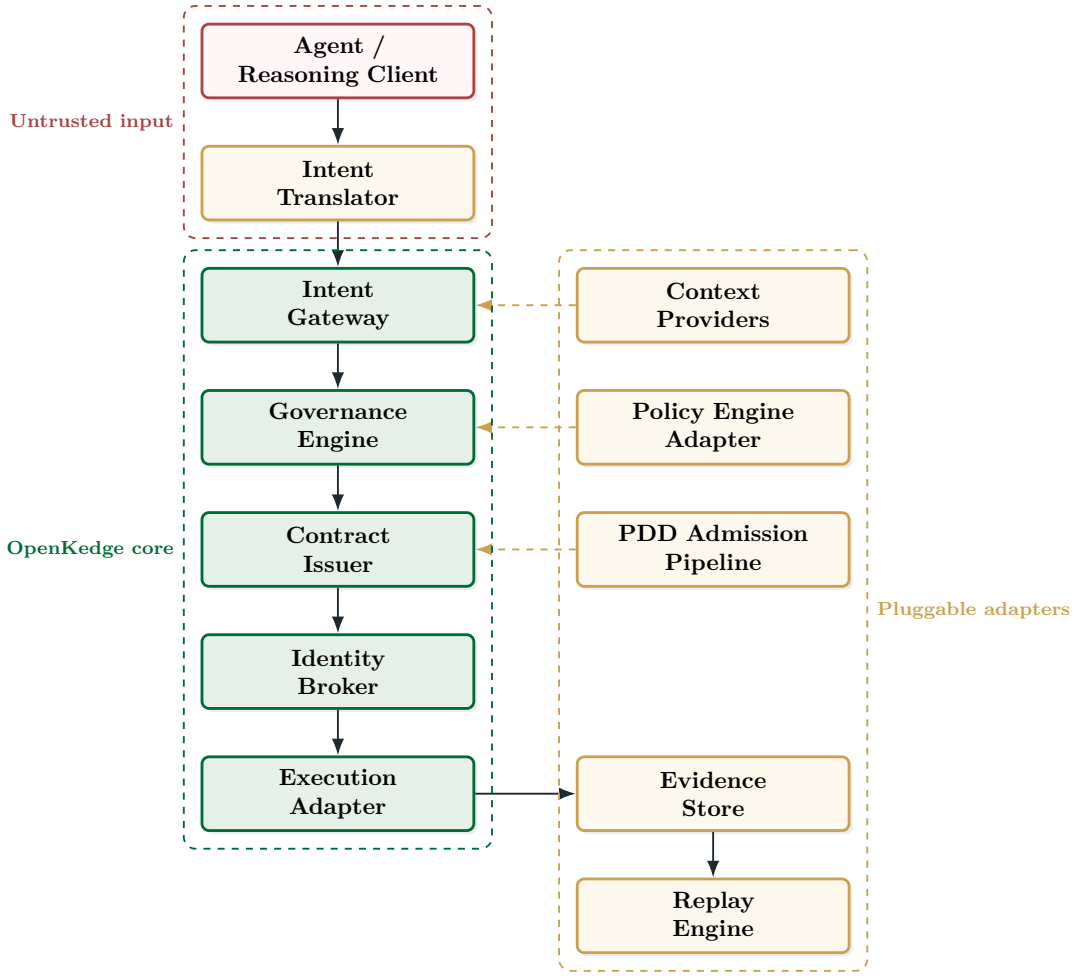


Figure C.1: *Reference implementation components for OpenKedge.*

```
interface ContextProvider {
  collect(intent: Intent): Promise<ContextSnapshot>
}
interface PolicyEngine {
  evaluate(input: PolicyInput): Promise<GovernanceDecision>
}
interface ContractIssuer {
  issue(intent, context, decision): ExecutionContract
}
interface IdentityBroker {
  derive(contract): Promise<ExecutionIdentity>
}
interface ExecutionAdapter {
  execute(contract, identity): Promise<ExecutionResult>
}
interface EvidenceStore {
  append(event): Promise<void>
}
```

```
interface ReplayEngine {
  replay(chainId): Promise<ReplayResult>
}
```

Listing C.2: *Intent translator interface.*

```
interface IntentTranslator {
  translate(input: ReasoningOutput): Promise<IntentCandidate>;
}

interface IntentCandidate {
  rawInputRef: string;
  proposedIntent: Partial<Intent>;
  confidence?: number;
  assumptions?: string[];
}
```

An intent candidate is untrusted until validation succeeds. Validation of the intent candidate at the gateway enforces structural completeness, schema correctness, and objective-action consistency. Each valid intent is bound to a unique chain identifier, enabling cryptographic correlation of all subsequent context snapshots, policy decisions, contracts, and execution events.

Conforming implementations must define rigorous conformance tests. Policy engine adapters must guarantee deterministic decision translation, execution adapters must reject actions that fall outside the contract bounds, and replay engines must detect evidence omission.

All schemas and interfaces are explicitly versioned. Structural specifications for intents, contracts, and evidence payloads must support backward compatibility, while allowing domain-specific schema extensions that do not weaken the core invariants. Adapters declare supported contract and evidence versions, enabling the governance engine to reject execution paths if an adapter lacks the capacity to enforce the active contract.

C.4 Intent Schema

Intent is the boundary artifact between reasoning and governance. The intent schema should be expressive enough to support policy evaluation, but narrow enough to prevent arbitrary model output from becoming authority. It should make the proposed action explicit, bind it to an actor and source, describe the objective, define the target and scope, capture assumptions and constraints, include justification, and carry creation and expiration metadata.

Reference fields should identify the intent, actor, source, objective, requested action, target, scope, assumptions, constraints, justification, risk hint, creation time, expiration time, and metadata. Implementations may add domain fields, but the governance engine should not accept unbounded model text as executable input. The illustrative schema below shows one compact representation.

Listing C.3: *Illustrative intent schema.*

```
{
```

```
  "intentId": "intent-123",
  "actor": "ops-agent",
  "source": "external-reasoning-client",
  "objective": "restore checkout service health",
  "requestedAction": "shift_traffic",
  "target": "service.checkout",
  "scope": {
    "region": "us-west-2",
    "maxPercentage": 20
  },
  "assumptions": [
    "current_error_rate_above_threshold",
    "secondary_capacity_healthy"
  ],
  "constraints": {
    "expiresIn": "10m",
    "requirePostVerification": true
  },
  "justification": "Reduce elevated 5xx rate by shifting traffic.",
  "riskHint": "medium"
}
```

Intent validation should reject missing actors, ambiguous targets, expired requests, broad scopes without justification, unsupported requested actions, and mismatches between objective and requested action. It should also attach a chain identifier so later context, policy, contract, identity, execution, and verification events can be correlated.

C.5 Context Provider Interface

Context providers turn operational state into policy-evaluable evidence. They may retrieve cloud resource state, topology, health metrics, current traffic, ownership metadata, resource tags, compliance state, policy metadata, change windows, incident state, deployment state, and previous related intents.

Context collection should be scoped to the intent. A provider should gather the facts needed to evaluate the proposed action, not export broad operational state to the reasoning layer. Context should include provenance, freshness timestamps, source identifiers, and uncertainty. Missing context should be explicit. A governance engine should be able to distinguish between a confirmed safe condition, a confirmed unsafe condition, and an unknown condition.

Listing C.4: *Context provider interface.*

```
interface ContextProvider {
  name: string;
  supports(intent: Intent): boolean;
  collect(intent: Intent): Promise<ContextSnapshot>;
}
```

```
interface ContextSnapshot {
  snapshotId: string;
  collectedAt: string;
  sources: ContextSource[];
  facts: Record<string, unknown>;
  freshness: Record<string, string>;
}
```

Context snapshots should be stored or referenced as evidence. Replay depends on knowing which facts were available at decision time and how fresh they were. For high-risk execution, adapters may perform a final context revalidation immediately before action.

C.6 Policy Engine Interface

OpenKedge is not a replacement for policy engines; it is the substrate that makes autonomous intent evaluable by them. The policy layer should support Cedar, OPA/Rego, institutional approval systems, custom validators, and domain-specific rules through a common adapter shape.

Policy input should include the intent, context snapshot, actor, resource, requested action, risk classification, and evidence references. Policy output should include a normalized decision, constraints, reasons, required approvals, required evidence, policy version, and obligations. The governance engine should treat the policy output as an input to contract issuance, not as executable authority by itself.

Listing C.5: *Policy engine interface.*

```
type DecisionKind =
  | "allow"
  | "deny"
  | "constrain"
  | "escalate"
  | "simulate"
  | "defer"
  | "request_context";

interface PolicyEngine {
  name: string;
  evaluate(input: PolicyInput): Promise<GovernanceDecision>;
}

interface GovernanceDecision {
  decisionId: string;
  kind: DecisionKind;
  reasons: string[];
  constraints?: Record<string, unknown>;
  obligations?: EvidenceObligation[];
  policyVersion?: string;
}
```

Policy adapters should preserve policy provenance. A replay engine must know which policy set, version, bundle, approval workflow, or validator produced the decision. If the policy engine returns uncertainty or missing context, OpenKedge should request context, defer, escalate, or deny rather than convert uncertainty into permission.

C.7 Execution Contract Schema

The execution contract is the machine-enforceable bridge between governance and runtime authority. It converts a policy decision into a bounded artifact that an identity broker and execution adapter can verify.

Reference fields should identify the contract, intent, policy decision, approved action, target, bounds, constraints, validity interval, identity scope, evidence obligations, revocation conditions, rollback plan, and integrity metadata. The contract should be precise enough that an adapter can decide whether a requested runtime action is inside or outside the approved bounds.

Listing C.6: *Illustrative execution contract schema.*

```
{
  "contractId": "contract-456",
  "intentId": "intent-123",
  "decisionId": "decision-789",
  "approvedAction": "shift_traffic",
  "target": "service.checkout",
  "bounds": {
    "region": "us-west-2",
    "maxPercentage": 20
  },
  "validUntil": "2026-05-18T18:00:00Z",
  "identityScope": {
    "actions": ["traffic:Shift"],
    "resources": ["service.checkout"]
  },
  "evidenceObligations": [
    "pre_context_snapshot",
    "policy_decision",
    "identity_issuance",
    "execution_result",
    "post_verification"
  ]
}
```

Contract issuance should be deterministic with respect to recorded intent, context, and decision. If replay reconstructs the same inputs under the same policy version, it should be able to verify that the contract was consistent with the decision.

C.8 Execution Identity Interface

The identity broker derives execution identity from an approved contract. The identity broker should never issue authority broader than the execution contract. Implementation patterns include short-lived cloud credentials, session policies, signed capability tokens, workload identity federation, workflow-scoped credentials, and sandbox capabilities.

Listing C.7: *Execution identity interface.*

```
interface IdentityBroker {
  derive(contract: ExecutionContract): Promise<ExecutionIdentity>;
  revoke(identityId: string, reason: string): Promise<void>;
}

interface ExecutionIdentity {
  identityId: string;
  contractId: string;
  issuedAt: string;
  expiresAt: string;
  scope: IdentityScope;
  materialRef: string;
}
```

The `materialRef` field should avoid storing sensitive credential material directly in evidence. Evidence should record that authority was issued, what scope it represented, which contract justified it, and when it expired. Secrets should remain in the institution's credential management system or runtime secret store.

Identity issuance should fail closed. If the contract is expired, unsigned, inconsistent with the decision, missing evidence obligations, or broader than policy allows, the broker should refuse issuance and emit an evidence event.

C.9 Execution Adapter Interface

Execution adapters enforce contracts rather than executing unstructured instructions. Each adapter maps the contract boundaries to specific infrastructure API mutations. The 'preflight' interface verifies context freshness, resource availability, and contract integrity; the 'execute' method executes the mutation under the short-lived identity; and the 'verify' method validates postconditions and records runtime outcomes. An adapter must reject parameters, resources, or actions not explicitly permitted by the contract even if the identity token is valid.

Listing C.8: *Execution adapter interface.*

```
interface ExecutionAdapter {
  name: string;
  supports(contract: ExecutionContract): boolean;
  preflight(
    contract: ExecutionContract,
    identity: ExecutionIdentity
```

```
  ): Promise<PreflightResult>;
  execute(
    contract: ExecutionContract,
    identity: ExecutionIdentity
  ): Promise<ExecutionResult>;
  verify(
    contract: ExecutionContract,
    result: ExecutionResult
  ): Promise<VerificationResult>;
}
```

The `preflight` method checks context freshness, contract validity, identity scope, and feasibility. The `execute` method performs the bounded action. The `verify` method checks postconditions, records outcomes, and emits evidence. An adapter should reject parameters, resources, or actions not explicitly permitted by the contract even when the identity token appears valid.

C.10 Evidence Chain Interface

Every meaningful governance transition should emit an evidence event. Reference event types should cover intent receipt, context collection, policy evaluation, contract issuance, identity issuance, execution start, execution completion, execution denial, contract violation, verification completion, replay completion, and admission evaluation.

Listing C.9: *Illustrative evidence event types.*

```
intent.received
context.collected
policy.evaluated
contract.issued
identity.issued
execution.started
execution.completed
execution.denied
execution.violated
verification.completed
replay.completed
admission.evaluated
```

Listing C.10: *Evidence event interface.*

```
interface EvidenceEvent {
  eventId: string;
  chainId: string;
  type: string;
  timestamp: string;
  subject: {
    intentId?: string;
    decisionId?: string;
```

```
    contractId?: string;
    identityId?: string;
    executionId?: string;
  };
  payloadHash?: string;
  payloadRef?: string;
  previousEventHash?: string;
  metadata?: Record<string, unknown>;
}

interface EvidenceStore {
  append(event: EvidenceEvent): Promise<void>;
  getChain(chainId: string): Promise<EvidenceEvent[]>;
}
```

Implementations may use hash chains, signatures, append-only logs, or trusted ledgers depending on assurance requirements. The reference implementation should begin with a simple evidence store that is easy to inspect and replay, then allow stronger storage backends as deployments mature.

Evidence storage should also support retention policy, access control, and redaction patterns. Some payloads may contain sensitive operational details, so the evidence event can store a payload hash and reference while keeping the full payload in a protected system. The replay engine should be able to retrieve authorized payloads when needed, but routine audit views should expose only the minimum detail required for review. This keeps the implementation evidence-first without turning the evidence store into an uncontrolled data lake.

C.11 Replay and Simulation Interface

Replay is the mechanism that turns evidence into operational learning. The replay engine should reconstruct intent, context, policy decision, contract, identity issuance, execution, and verification. Replay modes should include exact replay under recorded policy and context, policy replay under updated policy, what-if simulation with alternate context, incident replay, and certification replay.

Listing C.11: *Replay engine interface.*

```
interface ReplayEngine {
  replay(
    chainId: string,
    options?: ReplayOptions
  ): Promise<ReplayResult>;
  simulate(
    intent: Intent,
    scenario: SimulationScenario
  ): Promise<SimulationResult>;
}

interface ReplayResult {
```

```
  chainId: string;
  reconstructedDecision: GovernanceDecision;
  contractValid: boolean;
  identityWithinContract: boolean;
  executionWithinContract: boolean;
  evidenceComplete: boolean;
  findings: ReplayFinding[];
}
```

Replay findings should be actionable. They should identify missing context, missing policy versions, contract inconsistencies, overbroad identity, execution outside the contract, incomplete verification, or evidence gaps. Replay should be usable by engineers, auditors, incident responders, and policy owners.

C.12 Protocol Admission Interface

Generated code enters the system through admission, not trust. The PDD layer should provide a protocol registry, candidate artifact submission, invariant evaluators, admission pipeline, sandbox execution, and evidence output. Admitted artifacts can later be referenced by execution contracts and evidence chains.

Type-theoretic admission affects this interface design. A protocol registry may expose machine-checkable schemas, typed interfaces, refinement predicates, effect constraints, and resource bounds. Invariant evaluators may include type checkers, schema validators, refinement checkers, dependency analyzers, effect checkers, and sandbox tests. Admission evidence should record which checks passed, which checks failed, and which properties were unknown or deferred to runtime verification.

Listing C.12: *Protocol admission interface.*

```
interface Protocol {
  protocolId: string;
  version: string;
  structuralInvariants: Invariant[];
  behavioralInvariants: Invariant[];
  operationalInvariants: Invariant[];
}

interface AdmissionPipeline {
  evaluate(
    candidate: CandidateArtifact,
    protocol: Protocol
  ): Promise<AdmissionResult>;
}

interface AdmissionResult {
  resultId: string;
  kind: "admit" | "reject" | "constrain" | "review";
  evidence: AdmissionEvidence[];
```

```
  findings: string[];
}
```

Listing C.13: *Type-check result for protocol admission.*

```
interface TypeCheckResult {
  artifactId: string;
  protocolId: string;
  judgment: "inhabits" | "rejects" | "unknown";
  evidence: AdmissionEvidence[];
}
```

The initial PDD implementation should prioritize generated automation scripts, execution adapters, policy modules, and workflow definitions. These artifacts can affect the governance path directly and should not become operational simply because they compile or pass a few examples.

Advanced implementations of the symbolic governance layer can draw upon foundational work in refinement types [4], dependent types [20], and typed effect systems [12] to formally bound the behavior of generated code.

C.13 AWS Adapter Example

The AWS execution adapter provides a practical demonstration of intent governance applied to real-world cloud infrastructure. When an AI agent proposes EC2 instance termination for auto-recovery, the control plane intercepts the request. The gateway structures the intent, context providers gather instance metrics and Auto Scaling state, and the policy engine verifies safety bounds. The contract issuer then emits a tightly bounded contract, STS generates a task-scoped credential, and the adapter deregisters the instance, executes the drain, terminates the resource, and verifies replacement health.

The adapter validates all actions against the contract; it does not accept arbitrary API commands. Real-world AWS implementations leverage STS session policies [1], scoped IAM roles, and Config rules to enforce bounds.

Adapter Bypass

A reference implementation is only effective if execution paths are forced through governed adapters. If agents retain direct access to privileged APIs, the control plane becomes advisory rather than enforceable.

Early adapter iterations should focus on a narrow, well-tested set of remediation operations. The objective is to establish an end-to-end integration path that verifies context acquisition, contract generation, transient credentials, and evidence collection before expanding the API surface.

C.14 Repository Structure

A monorepo structure segregates the core governance logic from domain-specific extensions. The ‘core‘ package implements validation, decision normalization, contract logic, and evidence parsing.

Adapter packages integrate with external clouds and policy engines, while ‘pdd‘ manages type-theoretic admission.

C.15 Deployment Modes

Conformant deployments scale from local verification to enterprise and sovereign runtime environments. Local development mode utilizes SQLite and mock adapters for rapid prototyping; enterprise mode integrates with native IAM, change management, and internal telemetry; sovereign mode provides strict air-gapped isolation with offline reasoning and strict data residency controls.

Sovereign Deployment

The architecture should support local, enterprise, and sovereign deployments without requiring a hosted SaaS control plane.

In local development mode, the system can use a file-backed or SQLite evidence store, mock policy engine, mock execution adapter, and replay CLI. This mode is useful for demos, tests, protocol experimentation, and contributor onboarding.

In enterprise mode, the system is deployed inside the institution’s infrastructure. It integrates with existing IAM, policy systems, observability, CI/CD, incident management, and audit tooling. Evidence is retained internally according to institutional retention policy.

In sovereign mode, the system runs in a national, regulated, or high-assurance environment with strict data and context boundaries, local policy ownership, controlled model integration, and long-term evidence retention. External reasoning may be allowed, but execution authority remains inside the sovereign control plane.

In air-gapped or restricted mode, the system can use local models or approved offline reasoning, a local evidence store, manual policy updates, and restricted adapter access. This mode is relevant when external model calls, hosted control planes, or remote telemetry are not permitted.

C.16 Implementation Roadmap

The development roadmap prioritizes end-to-end loop integration over API breadth. Early milestones demonstrate core validation and replay capabilities (v0.1-v0.2) before expanding to native cloud integration (v0.3), protocol admission (v0.4), and production-grade API stability (v1.0).

Version 0.1 should include the core intent schema, governance decision model, evidence event model, simple file or SQLite evidence store, mock policy engine, mock execution adapter, and replay CLI. This creates the minimum visible loop.

Version 0.2 should add OPA and Cedar adapters, AWS read-only context providers, execution contract issuer, and a basic replay UI. This stage should focus on policy-engine pluggability and context evidence.

Version 0.3 should add the AWS execution adapter, STS or session-policy identity broker, CloudWatch or CloudTrail integration, and an outage-prevention demo. This stage should demonstrate a real bounded action with evidence and replay.

Version 0.4 should add the PDD protocol registry, generated-code admission pipeline, sandbox execution, and CI/CD integration. This stage brings generated artifacts into the governance boundary.

Version 1.0 should stabilize APIs, document production deployment patterns, harden security controls, define adapter certification tests, verify evidence chains, and ship reference demos and contributor documentation.

This reference implementation is intentionally modular. The goal is to demonstrate how the architectural invariants can be enforced in real systems while allowing institutions to retain their own policy engines, identity systems, cloud providers, and evidence stores.

D Glossary

Glossary of Terms

Governing autonomous systems as infrastructure requires a precise, shared taxonomy that decouples cognitive reasoning from execution authority. This glossary establishes the formal vocabulary of the Autonomous State Control Plane (ASCP) architecture, aligning executive, engineering, security, and policy frameworks around verifiable execution.

Glossary Principle

Governed autonomy demands the strict separation of cognitive intelligence from operational authority, proposal from execution, and passive logging from active accountability.

Core Concepts

Autonomous AI System. Any computing architecture wherein artificial intelligence models or agents propose, orchestrate, or initiate state transitions within software, infrastructure, databases, or physical systems. Autonomy operates along a spectrum; it does not necessitate the complete exclusion of human oversight. The defining architectural risk is the capacity of probabilistic machine reasoning to influence deterministic real-world state.

Agent. An AI-driven component that ingests contextual telemetry, reasons over specific objectives, synthesizes action plans, invokes tools, and generates output. In the ASCP architecture, agents operate purely as non-authoritative reasoning engines: they propose actions but possess zero inherent execution privileges.

Agentic System. A software system integrating one or more autonomous agents operating statefully over time, executing tools, receiving feedback loops, and dynamically adjusting behavioral plans. Such systems demand strict structural governance, as agentic tool invocation translates cognitive reasoning errors into immediate, unguided operational changes.

Reasoning Layer. The computational tier dedicated to execution planning, natural language translation, and strategic analysis. While functionally essential, the reasoning layer operates without authority: it generates proposed transitions but lacks the deterministic mechanism to authorize execution.

Execution Layer. The physical or virtual environment hosting target systems, databases, cloud resources, CI/CD pipelines, or cyber-physical infrastructure. State changes within the execution layer constitute real-world consequences; thus, failures in the governance plane manifest here as operational or security compromises.

Mutation. Any operation that alters the state of a governed system. Mutations encompass infrastructure provisioning, workflow approvals, credential emissions, configuration updates, financial transfers, data exfiltration, or physical control commands.

Governed State Transition. A state change executed only after validating intent structure, operational context, policy compliance, cryptographic identity, and evidence criteria. A governed transition is actively bounded and validated *prior* to execution, preventing the vulnerabilities of passive post-hoc logging.

Governed Transition. A distinct state modification that strictly preserves control-plane constraints. Composed workflows require every constituent transition to remain within the sovereign execution boundary, ensuring transitive integrity across complex operations.

Autonomous Governance. The technical and administrative discipline of regulating how autonomous systems propose intents, secure authorization, execute tasks, generate structured evidence, and refine policies via replay mechanisms. It unifies declarative policy, cryptographically bounded identity, immutable evidence, and runtime enforcement.

Compositional Governance. The structural requirement that nested, delegated, or chained agentic workflows preserve strict governance boundaries upon composition. A composite workflow is admissible only if its constituent sub-intents, execution contracts, active privileges, and replay paths are individually validated and cryptographically linked.

Architecture Layers

Autonomous State Control Plane. A closed-loop governance architecture that translates probabilistic AI reasoning into bounded, evidenced, replayable, and sovereign state mutations. Rather than forcing probabilistic models to be error-free, the control plane ensures that any reasoning failure is contained within predefined policy limits. It mediates the boundary between cognitive reasoning and target execution environments.

Sovereign Agentic Loops, or SAL. An architectural pattern that strictly decouples probabilistic AI reasoning from sovereign, institutional execution authority. SAL allows non-authoritative models to propose transitions while preventing them from directly executing mutations on governed infrastructure.

OpenKedge. The open-source reference implementation of the intent-governance layer. OpenKedge ingests structured intents, evaluates them against active policy engines and context snapshots, issues cryptographically bounded execution contracts, and generates structured evidence. It transforms arbitrary API calls into governed, evidenced, and replayable state transitions.

Neuro-Symbolic Governance. A hybrid governance paradigm wherein neural networks translate ambiguous human objectives into candidate structured intents, while deterministic, symbolic engines evaluate these intents against formal policy, real-time context, execution contracts, and identity constraints. The neural layer proposes semantic intent; the symbolic layer enforces deterministic compliance.

Symbolic Governance Layer. The deterministic computing layer that evaluates structured artifacts (intents, context snapshots, policy rules, contracts, and identity assertions). It remains the final authority for execution, ensuring that neural interpretation errors cannot bypass hard-coded symbolic rules.

Verifiable Agentic Infrastructure, or VAI. The runtime trust layer that converts validated execution contracts into short-lived, proof-derived identities and cryptographically bounded operational authority. VAI enforces lease-based privileges, validates execution contexts, and produces deterministic evidence of all state modifications.

Protocol-Driven Development, or PDD. A software construction and admission paradigm that treats machine-enforceable protocols as the primary development artifact. Under PDD, machine-generated or human-written code is admitted to execution environments only upon proving adherence to structural, behavioral, and operational invariants.

Intent-to-Execution Evidence Chain, or IEEC. The cryptographically linked evidence chain tracking the lifecycle of an autonomous action: from intent proposal, context snapshot, policy decision, and contract issuance, to execution, verification, and replay. The IEEC constitutes the core accountability substrate for governed execution.

Governance Artifacts

Intent. A structured, declarative, and machine-evaluable representation of a proposed state transition. Operating as the boundary artifact between reasoning and governance, an intent defines *what* is proposed without conveying any inherent execution privilege.

Intent Gateway. The ingestion gateway that receives, parses, normalizes, and logs structured intents. It acts as a hard boundary, preventing raw, unstructured model output or arbitrary API payloads from bypassing the symbolic governance pipeline.

Intent Translator. An untrusted cognitive adapter that translates natural language commands, unstructured model outputs, or tool-call proposals into structured schemas representing candidate intents. Its output is treated as a tentative proposal until validated by the symbolic governance layer.

Sub-Intent. A child intent spawned from a parent task, delegated workflow, or composed loop. Every sub-intent must be independently evaluated, bounded, and logged within the evidence chain rather than inheriting execution privilege implicitly from the parent process.

Context Snapshot. An immutable, point-in-time record of the operational and environmental telemetry used to evaluate an intent. Snapshots capture resource health, network topology, resource ownership, deployment windows, active incidents, and identity context.

Context Provider. An integration adapter that gathers real-time system and environmental telemetry for policy evaluation. Context providers convert infrastructure, database, application, and workflow states into structured evidence ingested by the policy engine.

Policy. A declarative, machine-evaluable rule set defining the constraints under which an intent is allowed, denied, simulated, escalated, or deferred. Policies express legal, operational, and security requirements in a mathematically enforceable format.

Machine-Enforceable Policy. Declarative policies compiled into software-interpretable representations for dynamic evaluation against intents and context snapshots. While some qualitative policies require human arbitration, the ASCP boundary requires deterministic, machine-enforceable rules to regulate machine-to-machine interactions.

Policy Engine. The evaluation engine that computes governance decisions by validating intents and context snapshots against active policies. The ASCP is engine-agnostic, supporting standard policy engines (e.g., Cedar, OPA/Rego) alongside custom formal-verification systems.

Governance Decision. The cryptographic output generated by evaluating an intent against a specific policy set and context snapshot. Standard decisions include allow, deny, constrain, escalate, simulate, or defer. Decisions must be cryptographically bound to the specific policy version and context snapshot used during evaluation.

Execution Contract. A cryptographically signed, machine-enforceable contract specifying the precise operational bounds of an approved intent. The contract acts as the bridge between governance and runtime execution, constraining the action, target resources, allowed parameters, lease duration, identity scope, and evidence obligations.

Contract Issuer. The authoritative component that translates a favorable governance decision into a signed execution contract. It enforces strict traceability by linking the contract directly to the original intent, context snapshot, policy version, and active evidence chain.

Delegated Authority. A temporary runtime privilege issued exclusively for a delegated task or sub-intent. Delegated authority is strictly lease-bound and task-scoped; child processes never inherit the parent workflow's broader credentials or permissions.

Blast Radius. The maximum potential impact of an intent across system boundaries. It is calculated by identifying the services, data stores, financial records, infrastructure, or physical components that could be affected by execution failure or compromise.

Risk Classification. The process of assessing an intent's impact, operational reversibility, data sensitivity, and legal consequences. Risk classification maps intents to appropriate governance tiers, dictates required escalation paths, and defines safety parameters.

Semantic Safety. The operational state wherein a proposed action is safe with respect to system intent, state context, and institutional goals, rather than merely complying with API syntax. An agent may possess the technical authorization to invoke an API while the call remains semantically unsafe under current conditions.

Runtime and Identity

Execution Authority. The legitimate permission to perform state-changing mutations on target infrastructure. Execution authority is uniquely derived from the institution owning the relevant system resources, policy layers, and identity boundaries.

Runtime Authority. The active privilege level during execution, materialized via short-lived credentials, signed capability tokens, or adapter-enforced roles. In governed environments, runtime authority is strictly lease-based and bound to the associated execution contract.

Execution Identity. A cryptographically attested, lease-bound identity issued to an execution process. Derived directly from an execution contract, the execution identity serves as the runtime manifestation of validated permission, mapping directly to specific actions.

Proof-Derived Execution Identity. An execution identity whose active privilege is derived dynamically from verifiable evidence, including the intent, context snapshot, policy decision, and contract. Under the ASCP, execution privilege is a consequence of reconstructable proof rather than static, standing credentials.

Identity Broker. The trusted runtime component that translates approved execution contracts into short-lived execution identities. It interfaces with external identity systems to provision scoped API tokens, SPIFFE/SPIRE IDs, or temporary cloud sessions.

Standing Privilege. Persistent, ambient access credentials assigned to a service account, role, or agent independent of a specific intent. Standing privilege represents a severe vulnerability in autonomous systems, as reasoning failures or prompt injections can exploit broad, non-contextual authority.

Bounded Execution. The enforcement paradigm wherein target mutations are restricted to the parameters, resources, duration, and evidence obligations specified in the execution contract. Bounded execution ensures that the runtime, rather than the agent, remains the arbiter of allowed actions.

Execution Adapter. A domain-specific mediation layer that translates approved execution contracts into API calls or system commands on target infrastructure. Execution adapters interact solely with the contract, completely isolating the target system from direct agent commands.

Human Escalation. The structured redirection of ambiguous, high-risk, or policy-violating intents to institutional human review. Escalation is integrated directly into the policy engine's standard state-transition workflow rather than bypassed as an exception.

Evidence and Replay

Evidence Event. An immutable ledger entry capturing a state transition within the governance plane. Evidence events document intent ingestion, context snapshots, policy evaluation results, contract and identity issuance, and execution outcomes.

Evidence Chain. A cryptographically linked chain of evidence events that allows independent auditors to reconstruct the entire lifecycle of an autonomous transition, structurally linking the initial intent proposal to its ultimate execution outcome.

Evidence Completeness. The formal property ensuring an evidence chain contains all telemetry, policy versions, context snapshots, and cryptographic signatures required to fully reconstruct and audit an autonomous state transition.

Replay. The process of executing a deterministic verification pass over a recorded evidence chain to validate the legitimacy of a past action. Replay does not require regenerating the probabilistic reasoning path; it requires proving that the symbolic governance decision was mathematically correct given the recorded context and policy.

Replay Fidelity. The precision with which a replay engine can reconstruct the exact states, inputs, and constraints of a historical governance decision, establishing the validity of the resulting execution.

Simulation. The execution of governance evaluation using hypothetical context snapshots, alternative policy versions, or synthetic intents. Simulation enables the verification of high-risk actions and the testing of policy variations without altering production system state.

Auditability. The structural capacity of a system to provide independent, verifiable proofs justifying every autonomous decision and mutation. True auditability requires a cryptographically validated history demonstrating the precise policies and context that authorized an action.

Certification Package. A self-contained, cryptographically signed bundle containing the complete evidence chain, policy definitions, and verification proofs required to certify an autonomous workflow for regulatory, compliance, or institutional review.

Protocol-Driven Development

Protocol. A machine-enforceable, mathematically rigorous specification governing the structural, behavioral, and operational bounds of a software component. Protocols define the hard invariants that any candidate implementation must satisfy, independent of model generation instructions.

Type-Theoretic Admission. A formal approach to admission control wherein a candidate implementation becomes operational only if it is proven to inhabit the type-space defined by the protocol. This framework shifts verification upstream of runtime, complementing dynamic testing, sandboxing, and evidence compilation.

Refinement Type. A data type enriched with logical predicates that restrict the set of valid values. Within PDD, refinement types enforce precise static bounds on generated code, including resource parameters, rate limits, ownership metadata, and structural scopes.

Dependent Type. A type whose definition depends dynamically on runtime values, parameters, or environment variables. In PDD, dependent typing enables the static verification of properties parameterized by system variables, such as resource bounds or geographic regions.

Typed Effect. A static typing mechanism that explicitly declares and restricts the computational side effects (e.g., system writes, network calls, file system mutations, or credential accesses) a candidate block of code is permitted to perform.

Invariant. A fundamental property or constraint that a system component, runtime execution path, or state transition must preserve under all operations. Invariants govern structural interfaces, operational tolerances, and security boundaries.

Structural Invariant. An invariant defining the physical shape, API schema, interface contract, dependency graph, and module boundaries of a software component, guaranteeing syntactic and architectural compatibility.

Behavioral Invariant. An invariant regulating allowed behaviors, valid state transitions, preconditions, postconditions, and side-effect limits, ensuring that the runtime implementation complies exactly with protocol semantics.

Operational Invariant. An invariant defining required runtime characteristics, including latency tolerances, resource limits, timeout behaviors, observability hooks, rollback procedures, and evidence-emission obligations.

Admission. The gatekeeping process that evaluates a candidate implementation against its governing protocol. Within the ASCP architecture, generated or modified software components are integrated strictly through automated admission verification rather than ambient trust.

Admission Evidence. The structured evidence compiled during the admission process, including static analysis proofs, test suites, sandbox execution telemetry, security vulnerability scans, and manual review signatures.

Candidate Implementation. A newly generated or modified software artifact submitted for verification against a governing protocol. Candidate implementations are isolated and non-operational until they pass the formal admission gate.

Protocol Registry. The versioned repository storing the authoritative schemas, invariants, and protocols that govern system components. The registry ensures that historical admission decisions can be replayed and validated against the precise protocol version active at the time of execution.

Sovereign AI and Institutional Terms

Sovereign AI. An institutional or national capability to deploy artificial intelligence while retaining unilateral control over data residency, policy definition, identity boundaries, evidence chains, audit logs, and infrastructure mutations. While the underlying models may be sourced globally, execution authority remains strictly sovereign.

Execution Sovereignty. The unilateral power of an institution or nation to regulate how AI-generated reasoning affects its internal digital and physical systems. Execution sovereignty is maintained via local policy enforcement, identity brokering, execution contracts, and evidence verification.

Model Sovereignty. The control over the lifecycle, training data, weights, hosting environment, and inference infrastructure of an AI model. While model sovereignty is valuable, execution sovereignty is distinct and paramount; an institution can leverage external, non-sovereign models safely by routing proposals through a sovereign control plane.

Sovereign Execution. An execution runtime governed entirely by the policy engines, identity systems, and audit frameworks of the institution whose infrastructure is affected. It ensures that runtime privileges never escape local jurisdictional boundaries.

Sovereign Execution Boundary. The perimeter inside which all policies, identities, execution adaptors, evidence logs, and audit chains are controlled by the host institution. Non-sovereign, external reasoning systems may only communicate across this boundary via structured, non-authoritative intents.

Obfuscation Membrane. A context-filtering interface that sanitizes and abstracts the operational telemetry exposed to external reasoning models. The membrane provides models with sufficient context to propose useful actions while hiding structural details and preventing the model from exercising direct operational control.

Governance Boundary. The structural perimeter where non-authoritative proposals are subjected to policy validation, context analysis, contract generation, and identity brokering. It acts as the gateway separating cognitive reasoning from physical or virtual mutation.

Institutional Authority. The mandate of an organization, agency, or nation to govern operational systems, define policies, authorize state mutations, and assume accountability. The ASCP translates this institutional mandate into deterministic, machine-enforceable runtime configurations.

Policy Sovereignty. The capability of an organization to define, evaluate, and enforce operational constraints using local policy engines and approval workflows. Policy sovereignty prevents governance structures and risk tolerances from being delegated to external model providers or cloud hosts.

Accountable Automation. The paradigm of system automation that enforces explanatory audit trails, cryptographic evidence chains, and deterministic replay paths. Accountable automation is the prerequisite for deploying autonomous systems within critical, highly regulated, or public-sector infrastructure.

Bibliography

[1] Amazon Web Services. Aws security token service session policies. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html, 2026.

[2] Cedar Policy Language Project. Cedar policy language. https://www.cedarpolicy.com/, 2026.

[3] Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors. *Compositionality: The Significant Difference*. Springer-Verlag, 1998.

[4] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277. ACM, 1991.

[5] Artur d'Avila Garcez, Marco Gori, Luis C Lamb, Luciano Serafini, Michael Spranger, and Son N Tran. Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. *arXiv preprint arXiv:1905.06088*, 2019.

[6] Jun He and Deying Yu. Openkedge: Governing agentic mutation with execution-bound safety and evidence chains. *arXiv preprint arXiv:2604.08601*, 2026.

[7] Jun He and Deying Yu. Protocol-driven development: Governing generated software through invariants and evidence. *arXiv preprint arXiv:2605.12981*, 2026.

[8] Jun He and Deying Yu. Sovereign agentic loops: Decoupling ai reasoning from execution in real-world systems. *arXiv preprint arXiv:2604.22136*, 2026.

[9] Jun He and Deying Yu. Verifiable agentic infrastructure: Proof-derived authorization for sovereign ai systems. *arXiv preprint arXiv:2605.15228*, 2026.

[10] HUMAIN. Humain. https://www.humain.ai/en/index, 2026.

[11] Kingdom of Saudi Arabia. Saudi vision 2030. https://www.vision2030.gov.sa/, 2016.

[12] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57. ACM, 1988.

[13] National Institute of Standards and Technology. Artificial intelligence risk management framework (ai rmf 1.0). Technical Report NIST AI 100-1, National Institute of Standards and Technology, 2023.

[14] NEOM. Neom. https://www.neom.com/, 2026.

[15] Open Policy Agent. Open policy agent. https://www.openpolicyagent.org/, 2026.

[16] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[17] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. Technical Report NIST SP 800-207, National Institute of Standards and Technology, 2020.

[18] Saudi Data and AI Authority. Our strategies and initiatives. https://sdaia.gov.sa/en/SDAIA/SdaiaStrategies/pages/default.aspx, 2026.

[19] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[20] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM, 1999.