Concept Paper

Governance Models for Agentic Software Delivery

Francis Kagai *

Posted Date: 26 May 2026

doi: 10.20944/preprints202605.1737.v1

Keywords: agentic software delivery; governance; bounded autonomy; autonomous DevOps; runtime governance; operational invariants; policy-constrained automation; human-in-the-loop systems; verifiable software delivery

 doi:10.20944/preprints202605.1737.v1

Concept Paper

Governance Models for Agentic Software Delivery

Francis Kagai

School of Engineering, Swinburne University of Technology, Melbourne, Australia; fkagai@swin.edu.au

Abstract

Software delivery is moving from deterministic pipelines toward autonomous environments where AI agents make runtime deployment decisions. Current DevOps governance assumes predictable execution and offers no mechanisms for constraining agents that generate plans on the fly. This leaves critical gaps in trust, accountability, policy enforcement, and failure containment. We present a conceptual architecture for bounded autonomous delivery, in which agents operate within externally enforced operational, security, reliability, and compliance constraints. The architecture separates planning, execution, policy enforcement, runtime verification, and human oversight into composable layers. We propose a taxonomy of autonomy levels and define operational invariants that limit what agents can do at runtime. A recurring scenario (deploying a payment microservice on an e-commerce platform during peak traffic) grounds the concepts in operational practice. The perspective positions governed autonomous delivery as an emerging discipline that demands new assurance models before organizations can trust agents with production systems.

Keywords: agentic software delivery; governance; bounded autonomy; autonomous DevOps; runtime governance; operational invariants; policy-constrained automation; human-in-the-loop systems; verifiable software delivery

---

1. Introduction

A deployment pipeline that cannot surprise you is easy to govern. Every step is defined in code, reviewed before merge, and executed identically each time [1]. For over a decade, this determinism has been the implicit contract underlying DevOps governance: if you control the pipeline definition, you control what happens in production [2,3].

That contract is breaking. Agentic AI systems, autonomous agents built on large language models and planning architectures, can now reason about deployment strategies, adapt to runtime conditions, and execute multi-step operational workflows without predefined logic for each scenario [4,5]. These are not simple automation scripts with conditional branches. They are goal-directed systems that generate execution plans at runtime based on observations, constraints, and learned behavior [6].

Consider a scenario we will return to throughout this article. A production e-commerce platform needs to deploy a new version of its payment microservice during a major sales event. The service handles mobile money integrations [7,8], processes thousands of transactions per minute, and depends on infrastructure spanning multiple availability zones [9]. A traditional pipeline would execute a predefined sequence (build, test, canary at fixed percentage, wait, promote) regardless of current conditions. An agentic system, by contrast, might assess real-time traffic, choose a conservative 2% canary because transaction volumes are elevated, monitor M-PESA callback latencies as a deployment health signal, and autonomously pause the rollout when callback failures spike. None of this was explicitly programmed for this situation.

When that agent's autonomous decision to proceed with a later rollout stage degrades payment processing for thousands of customers, the governance questions become urgent: *who authorized this action*, *what reasoning produced it*, and *what constraints should have prevented it*? [10]

Existing DevOps governance [11,12] cannot answer these questions because it was never designed for systems that generate their own execution plans. Recent work on agentic AI security [13], trust

 doi:10.20944/preprints202605.1737.v1

in autonomous systems [14], and responsible AI-assisted development [15] recognizes the need for governance, but focuses on general AI rather than the specific operational realities of production deployment, where a wrong decision can cascade into service outages, transaction loss, or regulatory violations within minutes.

We address this gap by proposing a framework organized around one principle: *bounded autonomy*, defined as autonomous operational behavior constrained by externally enforced invariants, runtime verification, and unconditional human override authority. An agent operating under bounded autonomy can act independently within defined limits but cannot exceed those limits regardless of its internal reasoning or confidence.

This article contributes:

1. A governance architecture that separates planning, execution, policy enforcement, and human oversight into independently auditable layers.
2. A taxonomy of autonomy levels for delivery systems, from deterministic pipelines to adaptive multi-service orchestration.
3. An operational invariant model categorizing security, cost, reliability, compliance, and blast radius constraints.
4. Practitioner-oriented implications for DevOps, SRE, platform engineering, and organizational governance.

2. From Pipelines to Autonomous Delivery

The path from scripted automation to autonomous delivery is a graduated expansion of decision authority. Each stage widens what the system can decide on its own, and each stage requires correspondingly stronger governance.

Continuous delivery pipelines are deterministic: given the same inputs, they produce the same execution trace [1,16]. Governance is straightforward because the pipeline configuration *is* the governance artifact; it specifies what happens, when, and in what order. For the payment service, a traditional pipeline runs a fixed sequence: build, test, stage, approve, deploy canary at 10%, wait 30 minutes, promote. The pipeline cannot deviate. Securing it means securing its definition and execution environment [3].

Infrastructure as code extended automation to provisioning by introducing goal-directed execution [17]: specify the desired state, and let the system determine how to converge toward it. This prefigures the agentic pattern, since the system determines *how* to reach a target, but remains bounded by its declarations and cannot invent states on its own [18]. The payment service infrastructure [9] is provisioned declaratively, yet provisioning cannot autonomously redesign the network topology.

Machine learning in operations added limited adaptive behavior: anomaly detection, auto-scaling within bounds, and automated alerting [19,20]. These systems assist or execute narrow responses. They might scale payment service replicas during a load spike, but their decision authority is confined to specific, well-understood operational domains. They do not reason about deployment strategy.

Large language models and autonomous agent architectures [5,6] enable a qualitatively different class of system [4]. Returning to our scenario: an agentic system observes that mobile money transaction volumes [8] are unusually high, decides on a 2% canary instead of 10%, selects M-PESA callback latency as the primary health signal, and halts the rollout when callbacks degrade. These are decisions generated from goals and observations, not from predefined logic [21].

The system is no longer executing a known plan; it is producing plans at runtime. Pre-reviewing every possible execution path is infeasible because the path space is combinatorially large and context-dependent [10]. This challenge parallels problems in autonomic computing [22], where self-managing systems must be governed despite adaptive behavior, and in self-adaptive software architectures [23, 24], where design-time analysis cannot anticipate all runtime decisions.

 doi:10.20944/preprints202605.1737.v1

3. Why Current Governance Falls Short

Production teams adopting autonomous delivery encounter governance gaps that existing DevOps practices were not built to address [11,12]. Six operational concerns stand out.

The first is trust. In traditional pipelines, trust is binary: the definition was reviewed, so execution is trusted. Autonomous systems require graduated trust because the same agent might be safe to manage canary rollout percentages but dangerous if allowed to modify database schemas or PCI-DSS network isolation rules [14]. Current platforms provide no mechanism for encoding and enforcing these graduated trust boundaries.

The second is accountability. When a pipeline ships a faulty release, responsibility traces directly to the code change and its author. When an agent autonomously deploys a version that corrupts settlement calculations, attribution is unclear [25]. The fault could lie in the agent's reasoning, the policy constraints that failed to intervene, or the environmental signals that misled the decision. Meaningful accountability demands decision logging and causal traceability that current systems do not provide [26].

Third, deployment policies (change windows, promotion gates, compliance checks) typically live inside pipeline definitions [2]. An agent generating plans at runtime can inadvertently bypass these unless enforcement is architecturally external to the agent's decision logic [10]. For payment services operating under financial regulation, this separation is a hard requirement.

Fourth, agents that chain multiple actions (deploy, reconfigure, scale) can produce cascading failures larger than any single pipeline step would cause [13]. Containment must limit not just individual actions but their aggregate impact, following principles established in dependable systems engineering [27,28].

Fifth, regulatory and compliance frameworks increasingly demand that operational decisions be explainable and traceable [29]. A payment system under PCI-DSS cannot accept deployment decisions made by opaque inference. Every action requires documented rationale and a complete evidence chain [30].

Finally, blast radius becomes a concern at a scale traditional pipelines never created. A conventional pipeline deploys one service to one environment, so its blast radius is bounded by construction. An autonomous orchestrator coordinating payment, settlement, and notification services can amplify a single failure across the entire order processing chain [31,32]. Scope constraints must be explicit and enforced.

4. Architecture for Bounded Autonomous Delivery

Our central contribution is an architecture where autonomous agents operate within externally imposed, continuously verified operational boundaries. Figure 1 illustrates the components and their relationships.

The architecture separates concerns that traditional pipelines collapse into a single artifact. A planner agent receives high-level objectives, such as "deploy payment-service v2.3 to production," and reasons about deployment strategy, ordering, resource requirements, and risk [5]. It generates structured plans with preconditions, postconditions, and rollback specifications, but does not execute them directly [33]. This separation of planning from execution creates a natural governance checkpoint. In our scenario, the planner observes peak traffic and proposes a conservative strategy: canary at 2%, 15-minute observation windows, and automatic rollback if transaction errors exceed 0.1%.

Execution agents carry out the validated plan steps, interacting with container orchestrators, service meshes, and infrastructure APIs [21]. They are deliberately constrained: they execute approved steps and report outcomes but do not improvise. If the payment canary shows unexpected M-PESA callback degradation, the execution agent escalates rather than inventing a workaround.

 doi:10.20944/preprints202605.1737.v1

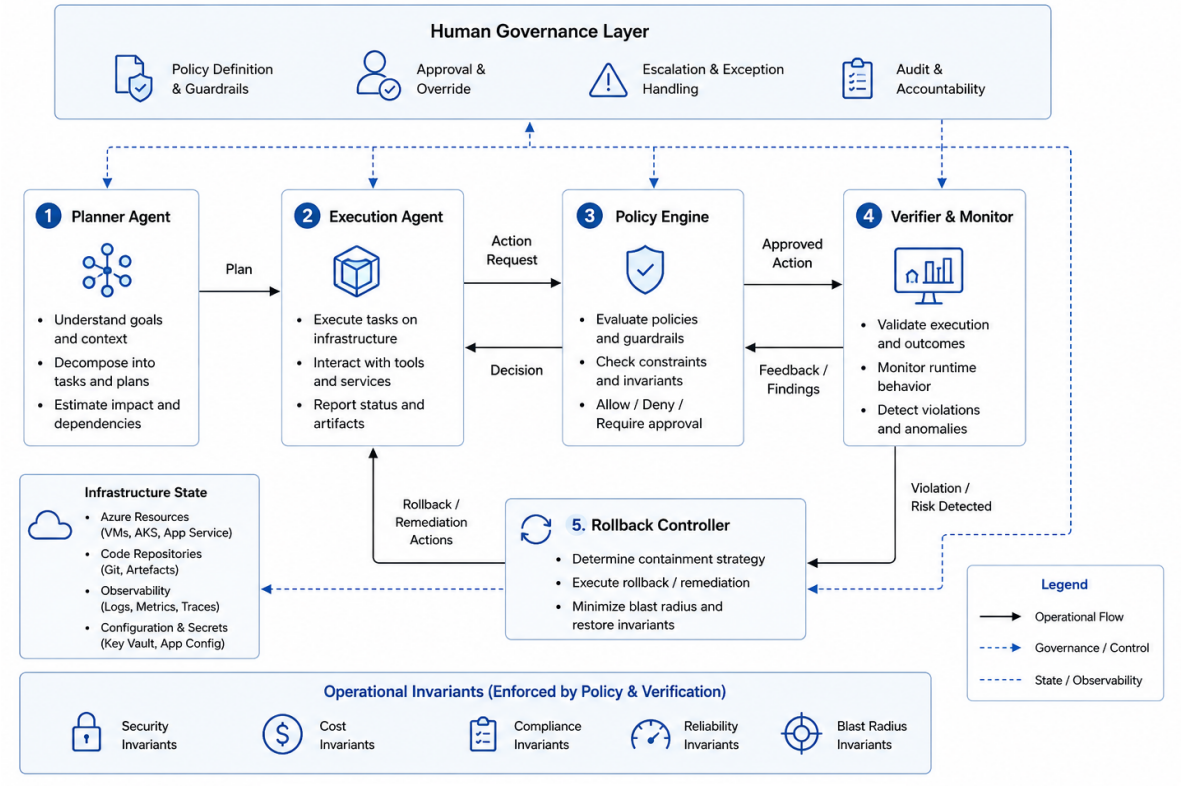

Figure 1. Bounded autonomous delivery architecture. The planner proposes delivery plans that the policy engine validates before execution agents act. The runtime verifier continuously checks invariants, and the human governance layer retains override authority at every stage.

Between the planner and execution sits the policy engine, which evaluates plans and individual actions against organizational constraints before permitting execution [10]. Policies are expressed as declarative constraints (security rules, cost ceilings, compliance checks, change window restrictions) maintained and version-controlled independently of agent logic. For the payment service, policies might specify that no production deployments occur during active settlement windows, PCI-DSS compliance checks must pass before any traffic shift, and initial blast radius is limited to one availability zone. The engine returns one of three verdicts: approve, reject, or escalate to a human decision-maker.

While the policy engine validates intent before execution, the runtime verifier assesses consequences during and after execution [30]. It continuously monitors whether operational invariants hold. When the payment canary causes transaction success rates to dip below 99.5%, the verifier triggers containment by pausing the rollout and alerting the governance layer. This pattern echoes runtime assurance approaches in safety-critical domains [34].

The rollback controller ensures that every autonomous action remains reversible within defined time bounds [1]. It preserves pre-deployment state, maintains reversal procedures per action type, and executes rollbacks either autonomously on verifier trigger or on human command. For payment services handling live financial transactions, rollback correctness is non-negotiable.

Humans occupy the governance layer. They set objectives, define boundaries, review escalated decisions, and hold unconditional override authority [35]. In our scenario, humans defined the 99.5% transaction success threshold, the PCI-DSS policy rules, and the escalation trigger for cross-zone deployments. They did not need to approve individual canary increments; that is the agent's operational domain within their defined constraints.

Underlying all components is an infrastructure state model: a queryable, consistent representation of the production environment, including services, configurations, dependencies, and health status [9, 17]. The planner and verifier both read from this model. Updates flow through a controlled write path that ensures consistency and retains full state history for audit purposes.

 doi:10.20944/preprints202605.1737.v1

5. Operational Invariants

Invariants are the properties that must hold true no matter what the agent decides. They operationalize bounded autonomy by making constraints concrete, measurable, and enforceable. We draw on dependability concepts [27] and safety engineering [32] to define five categories relevant to delivery systems [10].

Security invariants preserve the posture of production environments. For our payment service, this means no deployment proceeds with known critical vulnerabilities, network policies must maintain PCI-DSS isolation, secrets must not leak into observable channels, and authentication configurations cannot weaken as a result of any autonomous action [7,13]. Violations in this category trigger immediate halt and mandatory escalation.

Cost invariants bound the financial impact of autonomous decisions. Resource provisioning is capped so that an agent optimizing payment service throughput cannot provision unbounded compute capacity [36]. These invariants ensure that no single autonomous decision or sequence of decisions creates uncontrolled financial exposure for the organization.

Reliability invariants define minimum service levels that must hold during and after operations [31]. For the payment service, the transaction success rate must remain above 99.5%, M-PESA callback latency must stay below 3 seconds [8], and end-to-end payment confirmation must complete within 10 seconds. The runtime verifier evaluates these continuously and triggers containment when violations are detected or predicted.

Compliance invariants encode regulatory requirements and represent the most rigid constraint category, since violations carry legal consequences [29]. Data residency, change management audit trails, separation of duty, and evidence retention requirements admit no autonomous override under any circumstance.

Blast radius invariants limit the scope of simultaneous impact: the number of services affected, the percentage of traffic exposed, the geographic scope of changes, and the duration before mandatory validation [31,32]. For our scenario, the initial canary is limited to one zone, traffic exposure cannot exceed 5% without human review, and payment and settlement services cannot undergo simultaneous deployments.

These five categories are enforced at three lifecycle points: plan validation through the policy engine, execution through the runtime verifier, and post-deployment through continuous monitoring. This defense-in-depth follows safety engineering principles [28], catching violations wherever they manifest rather than relying on a single enforcement gate [30].

6. A Taxonomy of Autonomy Levels

To support deliberate organizational decisions about how much authority to delegate, we propose a five-level taxonomy. It draws on autonomic computing classifications [22] and self-adaptive systems research [37,38], adapted for the delivery domain.

The payment scenario makes these levels tangible. At Level 0, deployment follows a fixed pipeline regardless of traffic [1]. At Level 1, automated checks restart a crashed canary but cannot change the deployment strategy [20]. At Level 2, the agent analyzes traffic patterns and proposes a conservative canary, but waits for human approval before acting [33]. At Level 3, the primary target of our architecture, the agent executes autonomously within defined invariants, escalating only when boundaries are approached [10]. At Level 4, it coordinates payment, settlement, and notification deployments together, adapting the entire orchestration in real time [30,39].

The taxonomy is descriptive, not prescriptive. Its value is providing shared vocabulary for governance decisions: "payment infrastructure operates at Level 3 for routine releases and Level 2 for schema migrations" is a concrete organizational commitment that guides engineering, compliance, and incident response.

 doi:10.20944/preprints202605.1737.v1

Table 1. Autonomy levels for software delivery.

Level	Name	What the system can decide	What governance must provide
0	Deterministic Pipeline	Nothing; executes a fixed definition.	Pipeline review; change management.
1	Assisted Remediation	Narrow responses to known failures (restart, retry, scale within limits).	Response policies; bounded action scopes; alerting.
2	Supervised Autonomy	Proposes plans; cannot act without human approval.	Approval workflows; rationale logging.
3	Bounded Delivery	Plans and executes within invariant boundaries; escalates at limits.	Invariants; policy engine; runtime verification; escalation.
4	Adaptive Orchestration	Coordinates multi-service deployments; adapts strategy in real time.	Full architecture; blast radius controls; multi-agent coordination.

7. Human Oversight

The human role does not disappear in autonomous delivery. Rather, it moves from executing operational steps to governing what agents are permitted to do [35]. Five mechanisms maintain oversight without reintroducing the manual bottlenecks that automation was designed to eliminate.

Approval requirements scale with operational impact [40]. Staging deployments may need no approval at all. Production canaries operating within defined parameters might require only post-hoc review. Changes touching PCI-DSS security boundaries require pre-approval from designated authority. Each organization calibrates these thresholds to its regulatory exposure and risk tolerance.

Escalation triggers define the conditions under which autonomous operation must pause and defer to human judgment [10]. These include invariant near-violations, agent confidence falling below a defined threshold, novel failure patterns outside policy coverage, and multi-service impact exceeding blast radius limits. When the payment agent encounters unfamiliar M-PESA error codes that do not match any known failure mode, escalation is the correct response: governance working as designed, not a system malfunction.

Override authority is unconditional. Operators retain the ability to halt any autonomous action immediately, force rollback to a known-good state, or suspend autonomous operation entirely [31]. For systems processing live financial transactions, override latency is a design constraint, and the architecture must support sub-second halt commands. This aligns with established requirements in safety-critical system design [28,32].

Accountability distributes across architectural layers [25]. The agent is accountable for operating within its boundaries. The policy engine is accountable for correct constraint enforcement. The human governance layer is accountable for defining appropriate boundaries. The organization is accountable for overall governance posture. This layered model prevents both accountability gaps and misplaced blame after incidents.

Every decision, policy evaluation, escalation, and override is recorded in an immutable audit trail with full context: inputs, reasoning, action, and observed outcome [26]. For regulated payment services, these records serve directly as compliance evidence, produced as a natural byproduct of governed operation rather than as a separate reporting burden.

8. Implications for Practice

The architectural ideas above have concrete consequences for how teams build, operate, and govern software delivery. This section examines the changes practitioners will encounter.

 doi:10.20944/preprints202605.1737.v1

8.1. Governance Engineering Replaces Pipeline Engineering

The central DevOps artifact shifts from pipeline definitions to governance specifications [2,11]. Instead of writing Jenkinsfiles or GitHub Actions workflows that encode execution steps, engineers write invariant definitions, policy rules, trust boundaries, and escalation protocols that encode operational limits. This mirrors the earlier shift from imperative scripts to declarative infrastructure [12], but applied to the governance layer itself.

The pipeline does not disappear. It becomes one mechanism among several through which agents interact with production. The governance specification becomes the authoritative control surface: the artifact that determines what is permitted, not what is executed.

8.2. SRE Alignment

Site reliability engineering provides strong conceptual alignment [31,36]. Service level objectives map naturally to runtime invariants that the architecture enforces continuously. Error budgets become autonomy budgets, representing quantified risk the agent may consume before human intervention is required.

Incident response playbooks become escalation protocols. The SRE concept of toil elimination extends naturally: agents absorb routine deployment toil while SRE teams govern the boundaries and respond to escalations that exceed autonomous authority.

8.3. Governance as Testable Code

If governance specifications are code, they require engineering rigor [18]. Policy unit tests verify that security invariants correctly block non-compliant deployments. Integration tests validate policy composition, confirming that cost constraints and blast radius limits interact as expected when triggered simultaneously.

Chaos tests probe enforcement under adversarial conditions: injecting a metric anomaly and verifying that the runtime verifier halts the rollout. For the payment service, policy tests confirm that PCI-DSS constraints prevent deployment of unvalidated configurations and that cost limits engage before budget thresholds are exceeded [7].

8.4. Continuous Assurance

Governance effectiveness requires ongoing validation, not one-time verification [25,29]. Invariant definitions must keep pace with changing regulations. Policy engines must be tested against evolving constraint specifications. Escalation paths must be verified for responsiveness: do the right people receive alerts within acceptable latencies?

These operational assurance workflows are the governance equivalent of chaos engineering: deliberately stressing governance mechanisms to confirm they hold under realistic operational pressure. Organizations that treat governance as a static configuration rather than a living system will discover gaps only during incidents.

8.5. Organizational Structures

Technical architecture alone is insufficient. Organizations adopting autonomous delivery need governance bodies, such as AI deployment review boards, that oversee the expansion of autonomous authority [35]. These boards evaluate whether governance infrastructure is mature enough to support proposed autonomy increases, and authorize boundary expansions based on demonstrated safe behavior.

Autonomy readiness assessments complement these boards [37]. Before granting wider authority, organizations evaluate several dimensions: whether invariants are well-defined, whether monitoring can detect violations reliably, whether rollback mechanisms have been tested under production conditions, whether escalation paths respond within required latencies, and whether teams are culturally prepared to trust-but-verify autonomous systems. Premature autonomy that outstrips governance ca-

 doi:10.20944/preprints202605.1737.v1

pability is how organizations discover the risks of autonomous delivery through production incidents rather than through deliberate engineering.

9. Open Research Challenges

Several problems remain unsolved and warrant sustained research attention [4,6].

The most fundamental is formal verification of governance properties. Can we prove that a specific governance configuration actually constrains agent behavior as intended? Demonstrating that invariants are never violated, or that violations are always detected and contained, requires formal methods applied at the boundary of agent reasoning and policy enforcement [26,34]. This extends known verification challenges from autonomic systems [22] into the delivery domain, where the interaction between learned decision logic and declarative constraints creates a verification surface that existing tools do not address well.

Decision explainability is closely related but distinct. LLM-based agents produce natural language rationales for their decisions, but these may not faithfully represent the actual decision process [35]. Operators investigating a failed deployment need explanations they can trust, not post-hoc rationalizations generated to sound plausible. Developing methods for faithful explanation of delivery decisions, where the explanation provably corresponds to the reasoning that produced the action, remains an open problem.

Adversarial resilience introduces a threat model that governance must account for. When an adversary deliberately corrupts the signals an agent relies on (poisoning production metrics, fabricating health indicators, or manipulating cost data) the agent may make decisions that appear locally rational but serve the adversary's objectives [13]. Governance architectures must remain effective even when the operational environment is adversarial rather than merely noisy.

Multi-agent coordination creates governance challenges at a higher level of abstraction. When organizations deploy multiple autonomous systems across different delivery domains, governance boundaries must compose correctly [33,38]. How conflicts between agents are detected and resolved, and what collective properties must hold beyond individual agent constraints, are questions without established answers.

Trust calibration over time is another open area. An agent demonstrating consistently safe behavior over hundreds of deployments might deserve expanded autonomy, while one that frequently approaches invariant boundaries might warrant tighter constraints [14]. Principled mechanisms for adjusting trust dynamically, and for knowing when to revoke previously expanded authority, require both theoretical foundations and practical implementation strategies.

Finally, stateful rollback raises correctness questions that do not arise for stateless services. Rolling back a payment processor that has been committing live transactions during the canary period cannot simply restore a previous container image [30]. Partial rollbacks, state-entangled reversals, and correctness verification during the rollback process itself (where the rollback may introduce inconsistencies worse than the original problem) remain difficult open problems with direct operational consequences.

10. Conclusions

Autonomous delivery is a governance problem before it is an automation problem. The question is not whether agents *can* deploy software (they demonstrably can) but whether organizations can constrain, verify, and hold accountable the autonomous decisions those agents make in production.

Bounded autonomy provides the organizing principle: agents operate freely within externally enforced, continuously verified, human-overridable limits. The architecture, invariant categories, autonomy taxonomy, and practitioner implications presented here give teams a starting point for adopting autonomous delivery deliberately rather than reactively.

 doi:10.20944/preprints202605.1737.v1

Organizations that adopt governance-first will deploy autonomous systems confidently. Those that do not will discover the boundaries their agents needed through the production incidents that define them.

References

1. Humble, J.; Farley, D. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deploy Automation*; Addison-Wesley Professional, 2010.
2. Kim, G.; Debois, P.; Willis, J.; Humble, J. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*; IT Revolution Press, 2016.
3. Bass, L.; Holz, R.; Rimba, P.; Tran, A.B.; Zhu, L. Securing a Deployment Pipeline **2018**. pp. 4–7. https://doi.org/10.1109/RELENG.2015.7087454.
4. Acharya, D.B.; Kuppan, K.; Divya, B. Agentic AI: Autonomous Intelligence for Complex Goals—A Comprehensive Survey. *IEEE Access* **2025**, *13*, 18912–18936. https://doi.org/10.1109/ACCESS.2025.3532853.
5. Wang, L.; Ma, C.; Feng, X.; Zhang, Z.; Yang, H.; Zhang, J.; Chen, Z.; Tang, J.; Chen, X.; Lin, Y.; et al. A Survey on Large Language Model Based Autonomous Agents. *Frontiers of Computer Science* **2024**, *18*. https://doi.org/10.1007/s11704-024-40231-1.
6. Khalid, O.; Farooqi, A.U.H.; Bilal, M. Agentic AI: A Review, Applications and Open Research Challenges. *Computing* **2026**, *108*. https://doi.org/10.1007/s00607-026-01676-3.
7. Kagai, F.K. Design and Development of a Secure Mobile Money Integration for E-Commerce Platforms: A Software Engineering Approach **2018**. https://doi.org/10.5281/zenodo.13369096.
8. Wairimu, S.W.; Kagai, F. A Secure IoT–Cloud Framework for Mobile Payments Using M-PESA **2026**. https://doi.org/10.5281/zenodo.19696284.
9. Kagai, F.K. ICT Infrastructure for Campus Big Data: Network, Storage and Security Design and Implementation, 2019. https://doi.org/10.13140/RG.2.2.13175.88481.
10. Karuppuchamy, S. Governed Agentic AI for Software Platforms: A Reference Architecture for Safe Autonomy at Scale. In Proceedings of the Conference Proceedings - IEEE SOUTHEASTCON, 2026. https://doi.org/10.1109/SoutheastCon63549.2026.11475963.
11. Ebert, C.; Gallardo, G.; Hernantes, J.; Serrano, N. DevOps. *IEEE Software* **2016**, *33*, 94–100. https://doi.org/10.1109/MS.2016.68.
12. Kagai, F.K. Adapting Agile DevOps for Strategic Information Systems Development. *TechRxiv Preprint Server* **2019**, *2*. https://doi.org/10.36227/techrxiv.10279193.v3.
13. Chhabra, A.; Datta, S.; Nahin, S.K.; Mohapatra, P. Agentic AI Security: Threats, Defenses, Evaluation, and Open Challenges. *IEEE Access* **2026**, *14*, 49455–49482. https://doi.org/10.1109/ACCESS.2026.3675554.
14. Rodriguez-Barroso, N.; Garcia-Marquez, M.; Luzon, M.V.; Herrera, F. From Privacy to Trust in the Agentic Era: A Taxonomy of Challenges in Trustworthy Federated Learning Through the Lens of Trust Report 2.0. *Information Fusion* **2026**, *132*. https://doi.org/10.1016/j.inffus.2026.104236.
15. Elgendy, I.A.; Dwivedi, Y.K.; Al-Sharafi, M.A.; Hosny, M.; Helal, M.Y.I.; Crick, T.; Hughes, L.; Alwahaishi, S.; Mahmud, M.; Dutot, V.; et al. Responsible Vibe Coding: Architecture, Opportunities, and Research Agenda. *Journal of Computer Information Systems* **2026**. https://doi.org/10.1080/08874417.2026.2621186.
16. Fitzgerald, B.; Stol, K.J. Continuous Software Engineering: A Roadmap and Agenda. *Journal of Systems and Software* **2017**, *123*, 176–189. https://doi.org/10.1016/j.jss.2015.06.063.
17. Bass, L.; Weber, I.; Zhu, L. *DevOps: A Software Architect's Perspective*; Addison-Wesley Professional, 2015.
18. Rahman, A.; Parnin, C.; Williams, L. The Seven Sins: Security Smells in Infrastructure as Code Scripts. *IEEE Transactions on Software Engineering* **2023**, *49*, 102–119. https://doi.org/10.1109/TSE.2021.3137803.
19. Amershi, S.; Begel, A.; Bird, C.; DeLine, R.; Gall, H.; Kaez, E.; Nagappan, N.; Nushi, B.; Zimmermann, T. Software Engineering for Machine Learning: A Case Study. In Proceedings of the Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 291–300. https://doi.org/10.1109/ICSE-SEIP.2019.00042.
20. Leite, L.; Rocha, C.; Kon, F.; Milojicic, D.; Meirelles, P. A Survey of DevOps Concepts and Challenges. *ACM Computing Surveys* **2019**, *52*, 1–35. https://doi.org/10.1145/3359981.
21. Praneesh Roshan, P.; Thavasi, M.; Jaslin, C.Q. SDLC AutoPilot AI: Agentic Automation of Software Development Life Cycle. In Proceedings of the Proceedings of the 4th International Conference on Intelligent Computing, Information and Control Systems, ICOIICS 2025, 2025, pp. 1237–1242. https://doi.org/10.1109/ICOIICS67115.2025.11390299.

 doi:10.20944/preprints202605.1737.v1

22. Kephart, J.O.; Chess, D.M. The Vision of Autonomic Computing. *Computer* **2003**, *36*, 41–50. https://doi.org/10.1109/MC.2003.1160055.
23. Garlan, D.; Cheng, S.W.; Huang, A.C.; Schmerl, B.; Steenkiste, P. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. In Proceedings of the Proceedings of the 1st International Conference on Autonomic Computing (ICAC), 2004, pp. 276–277. https://doi.org/10.1109/ICAC.2004.1301370.
24. Kramer, J.; Magee, J. Self-Managed Systems: An Architectural Challenge. In Proceedings of the Future of Software Engineering (FOSE), 2007, pp. 259–268. https://doi.org/10.1109/FOSE.2007.19.
25. Radanliev, P.; Santos, O.; Maple, C.; Atefi, K. Operationalising Artificial Intelligence Bills of Materials for Verifiable AI Provenance and Lifecycle Assurance. *Frontiers in Computer Science* **2026**, *8*. https://doi.org/10.3389/fcomp.2026.1735919.
26. Merchant, P.S.; Gaba, S. Verifiable Collaboration among Agentic Code Assistants using Blockchain for Software Engineering Workflows. In Proceedings of the Proceedings - 2025 International Conference on AI x Software Engineering, AIxSE 2025, 2025, pp. 87–90. https://doi.org/10.1109/AIxSE64906.2025.00019.
27. Avizienis, A.; Laprie, J.C.; Randell, B.; Landwehr, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing* **2004**, *1*, 11–33. https://doi.org/10.1109/TDSC.2004.2.
28. Knight, J.C. Safety Critical Systems: Challenges and Directions. In Proceedings of the Proceedings of the 24th International Conference on Software Engineering (ICSE), 2002, pp. 547–550. https://doi.org/10.1145/581339.581406.
29. Zhu, L.; Xu, X.; Lu, Q.; Governatori, G.; Whittle, J. AI and Ethics—Operationalising Responsible AI. *ACM Computing Surveys* **2022**, *55*, 1–43. https://doi.org/10.1145/3585509.
30. Radanliev, P.; Atefi, K.; Santos, O.; Maple, C. Integrating Agentic Risk Signalling in Trusted Research Environments: Automating VEX with Agent2Agent Protocols and Model Context Protocol in SACRO and TREvolution Pipelines. *Computer Standards & Interfaces* **2026**, *96*. https://doi.org/10.1016/j.csi.2025.104079.
31. Beyer, B.; Jones, C.; Petoff, J.; Murphy, N.R. *Site Reliability Engineering: How Google Runs Production Systems*; O'Reilly Media, 2016.
32. Leveson, N.G. *Engineering a Safer World: Systems Thinking Applied to Safety*; MIT Press, 2011.
33. Doropoulos, S.; Vologiannidis, S.; Magnisalis, I. DevNous: An LLM-based Multi-Agent System for Grounding IT Project Management in Unstructured Conversation. *Information and Software Technology* **2026**, *195*. https://doi.org/10.1016/j.infsof.2026.108078.
34. Seshia, S.A.; Sadigh, D.; Sastry, S.S. Towards Verified Artificial Intelligence. *arXiv preprint arXiv:1606.08514* **2016**.
35. Shneiderman, B. Human-Centered Artificial Intelligence: Reliable, Safe & Trustworthy. *International Journal of Human-Computer Interaction* **2020**, *36*, 495–504. https://doi.org/10.1080/10447318.2020.1741118.
36. Forsgren, N.; Humble, J.; Kim, G. *Accelerate: The Science of Lean Software and DevOps*; IT Revolution Press, 2018.
37. Cheng, B.H.C.; de Lemos, R.; Giese, H.; Inverardi, P.; Magee, J.; Andersson, J.; Becker, B.; Bencomo, N.; Brun, Y.; Cukic, B.; et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*; Springer, 2009; pp. 1–26. https://doi.org/10.1007/978-3-642-02161-9_1.
38. Weyns, D. An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective **2020**.
39. Maire, F.; Badmaev, D.; Braik, F.; Gkartzonika, I.; Polymenakos, L.; Sfakianakis, P.; Tirchas, P. Agentic AI Framework for Technical Excellence: A Discipline-Based, Scalable Multimodal Assistant for Subsurface. In Proceedings of the IPTC Summit on AI for the Energy Industry, IPTC 2026, 2026. https://doi.org/10.2523/IPTC-25245-MS.
40. Parikh, N.A. *Agentic AI in Product Management: A Co-Evolutionary Model*; 2025; pp. 1–36. https://doi.org/10.4018/979-8-3373-6207-6.ch001.