ISSN: 2754-6659

Journal of Artificial Intelligence & Cloud Computing

SCIENTIFIC Research and Community

Review Article

Open Access

Self-Governing Cloud Performance: an MCP-Orchestrated Multi-Agent Blueprint for Autonomous SLA Assurance in Multi-Tenant Systems

Manvitha Potluri

Independent Researcher, Cloud Architecture and DevOps Engineering, USA

ABSTRACT

This paper presents a practitioner-oriented architectural blueprint for deploying self-governing performance management in multi-tenant cloud systems. Unlike prior theoretical treatments of AI-driven operations, this work provides DevOps Cloud Solutions Architects with concrete implementation patterns, deployment runbooks, infrastructure-as-code templates, and design decisions for constructing a multi-agent system orchestrated through the Model Context Protocol (MCP). The proposed architecture employs five coordinated agents, each bound by SLA-aware governance policies, that autonomously manage the full performance lifecycle from telemetry ingestion through remediation execution. We detail practical integration patterns for connecting agents to existing observability stacks (Prometheus, Grafana, Datadog), container orchestrators (Kubernetes, ECS), and CI/CD pipelines (ArgoCD, GitHub Actions). Based on published industry benchmarks and analytical modeling, the framework projects a 60-75% reduction in Mean Time to Resolution, SLA compliance exceeding 99.95%, and a 30-40% decrease in infrastructure spend. This paper provides the missing implementation bridge between academic agentic AI research and production-grade cloud operations.

***Corresponding author**

Manvitha Potluri, Independent Researcher, Cloud Architecture and DevOps Engineering, USA.

Received: March 03, 2026; **Accepted:** April 09, 2026; **Published:** June 27, 2026

Keywords: Multi-Agent Systems, Model Context Protocol, SLA-Aware Optimization, Cloud Performance Management, DevOps Architecture, Self-Governing Infrastructure, Kubernetes, Observability, Infrastructure-As-Code, Production Deployment Patterns

Introduction

The operational burden of managing performance in multi-tenant cloud systems has reached an inflection point. As organizations deploy hundreds of interconnected microservices across elastic infrastructure, the sheer volume of performance signals (metrics, logs, traces, and events) has exceeded human cognitive capacity for real-time synthesis [1]. DevOps teams routinely manage environments producing over 10 million metric data points per minute, yet the median time to detect and resolve a performance degradation event remains measured in hours, not minutes [2].

Recent advances in agentic AI, referring to systems that autonomously perceive, reason, plan, and execute actions, offer a path forward [3]. However, the existing literature on AI-driven performance engineering has focused predominantly on theoretical architectures and algorithmic contributions, leaving DevOps architects without actionable guidance for translating these ideas into production deployments. The gap between "research prototype" and "production system" in autonomous operations remains substantial: questions of credential management, blast radius containment, rollback choreography, tenant isolation enforcement, and integration with existing CI/CD pipelines are rarely addressed [4].

This paper bridges that gap. We present a complete implementation blueprint for a multi-agent performance management system orchestrated through the Model Context Protocol (MCP), specifically designed for DevOps Cloud Solutions Architects operating multi-tenant infrastructure [5]. Our contributions are: (1) a proposed five-agent architecture with detailed MCP server specifications; (2) concrete integration patterns for prevalent DevOps toolchains; (3) an SLA-aware governance engine with implementable policy schemas; (4) a phased deployment runbook with infrastructure-as-code templates; and (5) projected performance analysis grounded in published industry benchmarks.

The remainder of this paper is organized as follows. Section II surveys the practitioner landscape and identifies specific implementation gaps. Section III presents the system architecture. Section IV details MCP server design and integration patterns. Section V describes the SLA-aware governance engine. Section VI provides the deployment runbook. Section VII presents operational workflows. Section VIII presents projected performance analysis. Section IX discusses anticipated challenges and recommendations, and Section X concludes.

Practitioner Landscape and Gap Analysis

Current State of Performance Operations

Published industry reports reveal consistent patterns across enterprise DevOps organizations. According to the PagerDuty State of Digital Operations report, the majority of teams rely on static

threshold alerting as their primary anomaly detection mechanism, generating hundreds of alerts per week with high false-positive rates [6]. The DORA State of DevOps report indicates that most performance remediation actions, including scaling, configuration changes, and traffic shifting, still require manual execution by on-call engineers, even when the diagnosis and prescribed action are well-understood [4]. Furthermore, Gartner research on AIOps platforms highlights that fewer than 15% of organizations maintain tenant-specific performance baselines; the majority apply uniform thresholds across heterogeneous workloads [2].

Gaps in Existing AI-Driven Approaches

Current AIOps platforms (Dynatrace Davis, Datadog Watchdog, New Relic AI) provide anomaly detection and correlation capabilities but stop short of autonomous remediation. They operate as advisory systems: they surface insights, but a human must evaluate and execute. Research prototypes for autonomous performance engineering demonstrate algorithmic feasibility but omit critical production concerns: How does the agent authenticate to the Kubernetes API? What happens when two agents simultaneously attempt conflicting scaling actions? How are agent actions audited for SOC 2 compliance? How does the system degrade gracefully when the LLM provider experiences an outage? These are the questions this paper answers [7-9].

Why MCP for Toolchain Integration

The Model Context Protocol was selected as the integration backbone for three practical reasons. First, its tool-description schema allows agents to discover and invoke operational tools without hard-coded API clients, which is critical when toolchains evolve independently of the agent system [5]. Second, MCP's built-in session management and authentication delegation simplify credential lifecycle management. Third, MCP's streaming support enables agents to consume real-time telemetry feeds without polling, reducing latency between signal detection and agent reasoning from minutes to seconds [10].

System Architecture

Architectural Principles

The architecture is governed by five design principles derived from established cloud-native best practices: (1) Separation of perception and action, where agents that observe performance state must not directly execute remediation; a governance layer mediates all actions. (2) Fail-safe defaults, meaning any component failure results in the system reverting to human-in-the-loop mode, never in unconstrained autonomous action. (3) Tenant-first isolation, ensuring every data path and action scope is tenant-aware by construction, not by convention. (4) Idempotent operations, where every agent action can be safely retried without side effects. (5) Observable agents, requiring agents to emit structured telemetry about their own reasoning and actions, enabling human operators to audit and tune agent behavior [11].

Layer Architecture

The system comprises four layers. Table I summarizes each layer, its infrastructure components, and the specific technologies recommended for production deployment.

Table I: Layer Architecture with Recommended Components

Layer	Function	Recommended Stack
Telemetry Bus	Ingest, normalize, tag with tenant context	OpenTelemetry Collector, Kafka, Vector.dev
Intelligence Engine	Anomaly detection, correlation, baselining	Prometheus + custom Recording Rules, Grafana ML, ClickHouse
Multi-agent coordination, reasoning, planning	Multi-agent coordination, reasoning, planning	5 MCP-connected agents, Redis Streams event bus, LangGraph orchestrator
Governance Gateway	Policy enforcement, blast radius, audit	Open Policy Agent (OPA), Argo Rollouts, PostgreSQL audit store

Agent Roster and Responsibilities

Five specialized agents operate within the orchestrator. Each agent is implemented as an independent process with its own MCP client session, enabling independent scaling, fault isolation, and credential scoping. Table II details each agent's role, the MCP servers it connects to, and its maximum permitted autonomy level.

Table 2: Agent Roster With Mcp Connections and Autonomy Levels

Agent	Role	MCP Servers	Max Autonomy	Action Scope
Watchtower	Real-time anomaly detection and triage	Prometheus MCP, PagerDuty MCP	Level 2 (supervised)	Read-only + alert escalation
Elastik	Horizontal and vertical scaling decisions	K8s MCP, Cloud Provider MCP	Level 3 (autonomous)	Pod/node scaling within guardrails
Configurer	Runtime config and tuning optimization	ConfigMap MCP, Feature Flag MCP	Level 2 (supervised)	Non-destructive config changes
Arbitrator	Tenant fairness and SLA enforcement	Billing MCP, OPA MCP	Level 2 (supervised)	Quota adjustment, throttling
Strategist	Capacity planning and cost forecasting	FinOps MCP, All read servers	Level 1 (advisory)	Recommendations only

Agent naming was chosen deliberately to communicate function to operations teams. The Watchtower agent observes; the Elastik agent scales; the Configurer agent tunes; the Arbitrator agent enforces fairness; the Strategist agent plans. This naming convention is designed to help operators quickly identify which agent initiated an action from audit logs.

Inter-Agent Coordination Protocol

Agents communicate through a Redis Streams-based event bus using a structured envelope schema. Each event includes: agent_id, event_type (observation, proposal, approval, execution, outcome), tenant_scope, timestamp, correlation_id, and a typed payload. The correlation_id enables end-to-end tracing of a performance event from initial detection through remediation outcome [12].

A critical design decision is the proposal-approval pattern. When an agent determines that an action is necessary, it publishes a proposal event. The Governance Gateway evaluates the proposal against policies. For Level 2 agents, the Gateway also notifies the Arbitrator agent, which checks for cross-tenant conflicts. Only after approval does the originating agent receive an execution_ authorized event and proceed. This pattern prevents race conditions and ensures every action is auditable.

Mcp Server Design and Integration Patterns

MCP Server Implementation Blueprint

Each operational tool is exposed to agents through a dedicated MCP server that implements four mandatory interfaces: capability_ manifest (machine-readable description of available tools, their parameters, and constraints), health_check (liveness and readiness probes for the underlying tool), rate_limiter (per-agent and per-tenant rate limiting), and audit_emitter (structured logging of every tool invocation to the PostgreSQL audit store) [5].

MCP servers are deployed as sidecar containers alongside the tools they wrap, sharing the same network namespace. This deployment pattern provides three benefits: the MCP server inherits the tool's network policies and service mesh identity; credential injection uses the same secrets management path as the tool itself; and the MCP server can be upgraded independently of the tool without disrupting other agents.

Kubernetes MCP Server: Reference Implementation

The Kubernetes MCP server is the most complex integration point and serves as a reference for other implementations. It exposes seven tools: get_pod_metrics, get_hpa_status, scale_deployment, patch_resource_limits, get_node_allocatable, cordon_node, and get_events. Each tool enforces tenant-scoping through Kubernetes namespace isolation. The agent's MCP session is bound to specific namespaces corresponding to the tenant context, preventing cross-tenant access at the protocol level [13].

The scale_deployment tool implements three safety constraints hardcoded at the MCP server level (not reliant on agent reasoning): a maximum scale-up factor of 3x current replicas per invocation, a minimum of 2 replicas for any production deployment, and a cooldown period of 300 seconds between consecutive scaling actions on the same deployment. These constraints cannot be overridden by agent prompting, providing a defense-in-depth layer against agent reasoning failures.

Observability Stack Integration

The Prometheus MCP server provides agents with structured access to the metrics layer. Rather than exposing raw PromQL execution (which would require the agent to construct syntactically valid queries), the server exposes semantic tools: get_service_ latency_percentiles, get_error_rate, get_resource_saturation, get_ tenant_throughput, and compare_baseline. Each tool encapsulates a battle-tested PromQL query template, parameterized by service name, tenant ID, and time window. This design eliminates an entire class of agent errors, specifically malformed queries, while preserving the agent's ability to flexibly compose multi-signal analyses [14].

Integration with Grafana is implemented through the Grafana MCP server, which exposes tools for programmatic dashboard annotation (marking agent actions on relevant dashboards), snapshot creation (capturing dashboard state at the moment of agent decision for postmortem review), and alert silence management (temporarily silencing alerts during planned remediation windows to prevent alert storms).

CI/CD Pipeline Integration

The ArgoCD MCP server enables agents to participate in the deployment lifecycle. Agents can query deployment history to correlate performance regressions with recent deployments, trigger rollbacks to a previous known-good revision, and annotate deployments with performance impact assessments. The rollback tool requires Level 2 autonomy and triggers an automatic notification to the on-call engineer via the PagerDuty MCP server, ensuring human awareness even when the action is agent-initiated [15].

Sla-Aware Governance Engine

Policy Architecture

The governance engine is built on Open Policy Agent (OPA) with policies authored in Rego. Every agent proposal passes through a policy evaluation pipeline before execution is authorized. The policy stack is organized into four layers: safety policies (hard limits that cannot be overridden, e.g., maximum blast radius, minimum replica counts), SLA policies (tenant-specific constraints derived from contractual SLA definitions), operational policies (time-window restrictions, change freeze periods, concurrent action limits), and cost policies (budget ceilings, reserved instance utilization targets) [16].

Tenant SLA Model

Each tenant's SLA is encoded as a structured policy document containing: latency targets (p50, p95, p99 by service endpoint), availability targets (monthly uptime percentage), throughput guarantees (minimum sustained requests per second), and resource entitlements (guaranteed minimum CPU, memory, and storage IOPS). The Arbitrator agent continuously compares real-time tenant performance against these targets and prioritizes remediation actions for tenants approaching SLA breach thresholds [17].

A critical implementation detail is the SLA headroom calculation. The Arbitrator maintains a real-time "SLA burn rate" metric for each tenant, analogous to an error budget burn rate in SRE practice. When a tenant's SLA burn rate exceeds 1.5x the sustainable rate, the Arbitrator automatically elevates the priority of any pending optimization proposals affecting that tenant and can preempt lower-priority optimizations for other tenants. This mechanism ensures that SLA-critical situations receive immediate agent attention.

Blast Radius Controls

Every action proposal includes a blast radius declaration specifying: affected tenant(s), affected service(s), affected resource type, maximum percentage change, and estimated recovery time if rollback is required. The governance engine evaluates blast radius against a matrix of thresholds indexed by autonomy level and time window. Table III presents the blast radius matrix.

Table 3: Blast Radius Control Matrix

Dimension	Level 1 (Advisory)	Level 2 (Supervised)	Level 3 (Autonomous)
Max tenants affected	Unlimited (recommend only)	3 tenants per action	1 tenant per action
Max capacity change	N/A	±50%	±30%
Max services affected	N/A	5 services	2 services
Change freeze respect	N/A	Hard block	Hard block
Rollback time budget	N/A	15 minutes	5 minutes

Audit Trail and Compliance

Every agent decision, from initial observation through reasoning, proposal, policy evaluation, execution, and outcome verification, is recorded in an immutable audit store (PostgreSQL with append-only tables and row-level security). Each audit record includes: the full agent reasoning chain (LLM prompt and response), the MCP tool calls made with parameters and responses, the OPA policy evaluation result with the specific rules that matched, and the execution outcome with before/after metrics. This audit granularity satisfies SOC 2 Type II and ISO 27001 control requirements for automated change management [18].

Deployment Runbook

Prerequisites Checklist

Before initiating deployment, the following infrastructure prerequisites must be satisfied: an operational Kubernetes cluster (v1.27+) with Horizontal Pod Autoscaler and Metrics Server enabled; a deployed OpenTelemetry Collector with tenant-aware attribute processors; Prometheus with at least 30 days of historical metrics (required for baseline generation); OPA/Gatekeeper deployed with a functioning admission webhook; and a PostgreSQL instance (v15+) provisioned for the audit store with WAL-level replication for durability [19].

Phased Deployment Strategy

Deployment follows a four-phase progression. Table IV details each phase with specific entry criteria, deliverables, and validation checkpoints.

Table 4: Phased Deployment Strategy with Validation Checkpoints

Phase	Weeks	Deliverables	Entry Criteria	Exit Validation
1: Observe	1–4	Telemetry bus, MCP read servers, Watchtower agent	Prerequisites met, team trained	95% metric coverage, <5s ingestion latency
2: Advise	5–10	Elastik + Configurer in advisory mode, Governance engine	Phase 1 exit validated	80% recommendation accuracy vs. human decisions
3: Assist	11–18	Level 2 autonomy for Elastik + Configurer, Arbitrator active	Phase 2 exit validated, OPA policies approved	Zero SLA violations from agent actions
4: Govern	19–26	Level 3 for Elastik, Strategist active, full dashboard	Phase 3 exit validated, 90-day clean record	MTTR < 8 min, cost reduction > 25%

Infrastructure-as-Code Templates

All deployment artifacts are packaged as Helm charts with values files for each deployment phase. The chart structure includes: a base chart for the telemetry bus (OpenTelemetry Collector DaemonSet, Kafka StatefulSet, Vector.dev DaemonSet); a per-agent chart template parameterized by agent name, MCP server connections, resource limits, and autonomy level; and a governance chart (OPA deployment, policy ConfigMaps, audit store schema migrations). Phase transitions are implemented as Helm values overrides. Promoting an agent from Level 1 to Level 2 requires only a values change and a Helm upgrade, not a redeployment [20].

Rollback and Circuit Breaker Design

The system implements three rollback mechanisms at different granularities. Action rollback: every executed action records a compensating action in the audit store; if the outcome verification fails within the rollback time budget (Table III), the compensating action is automatically executed. Agent rollback: if an agent's error rate (failed actions / total actions) exceeds 10% within a 1-hour sliding window, the agent is automatically demoted to Level 1. System rollback: a global circuit breaker can be triggered by any operator via a dedicated Slack command (/agents-pause), instantly demoting all agents to Level 1 [21].

Operational Workflows

Autonomous Incident Response Scenario

We illustrate the proposed system's expected behavior through a representative scenario. Consider a case where the Watchtower agent detects a p99 latency increase on a critical API endpoint from 180ms to 1,240ms for an enterprise-tier tenant. The agent correlates this with three concurrent signals: a 340% increase in garbage collection pause time on 3 of 8 pod replicas, a memory utilization increase from 71% to 94% on those same pods, and a deployment event 47 minutes prior that modified JVM heap settings [22].

Watchtower publishes a structured observation event to the event bus. The Elastik agent consumes the event and formulates a two-phase remediation proposal: (1) immediate horizontal scale-out of the affected deployment from 8 to 12 replicas to restore SLA compliance while investigation continues, and (2) a rollback of the recent deployment to the previous revision. The Arbitrator agent verifies that the scaling action would not exceed the tenant's resource entitlement and would not impact other tenants on the same node pool.

The Governance Gateway evaluates the proposal: Phase 1 (scaling from 8 to 12, a 50% increase) falls within Level 3 blast radius limits; Phase 2 (deployment rollback) requires Level 2 approval, which is granted after the on-call engineer is notified via PagerDuty. The projected time from detection to SLA restoration is under 5 minutes. The equivalent manual workflow typically averages over 2 hours for similar incidents according to industry benchmarks [4].

Proactive Capacity Optimization Scenario

The Strategist agent operates on a weekly analysis cycle, consuming 30-day rolling metrics windows. In a representative scenario, it identifies that a batch processing workload exhibits a consistent pattern: CPU utilization peaks at 92% between 02:00–04:00 UTC on business days but averages only 23% during other hours. The Strategist recommends a time-aware HPA policy with pre-scale triggers 30 minutes ahead of predicted peaks, with the goal of reducing both peak-hour SLA violations and off-peak infrastructure costs [23].

Noisy Neighbor Mitigation Scenario

In this scenario, the Arbitrator agent's tenant fairness monitoring detects that a tenant's workload is consuming 3.2x its CPU entitlement during a traffic spike, causing p95 latency degradation for three co-located tenants. The Arbitrator publishes a throttling proposal: apply a CPU limit enforcement to the offending tenant's namespace and simultaneously trigger a horizontal scale-out for the affected neighbors to restore their SLA compliance. The proposal is executed within the Level 2 governance framework, with the offending tenant notified via the billing integration that their usage has exceeded contracted entitlements [24].

Projected Performance Analysis

Reference Deployment Scenarios

To evaluate the expected impact of the proposed framework, we define two reference deployment scenarios representative of common multi-tenant cloud environments: Scenario A, a multi-tenant SaaS application with 40-60 tenants across 200-400 microservices on a managed Kubernetes platform; and Scenario B, a financial data processing platform with 15-25 tenants across 50-100 services. Both scenarios assume mature observability stacks (Prometheus, Grafana, distributed tracing) as prerequisites [25].

Performance Projections

Table V presents projected performance improvements based on published industry benchmarks from the DORA State of DevOps report [4], PagerDuty operational data [6], and Gartner AIOps research [2]. These projections assume full Phase 4 (Govern) deployment maturity.

Table 5: Projected Performance Targets (Based on Industry Benchmarks)

Metric	Industry Baseline	Projected with Framework	Source
MTTD (median)	15-30 min (DORA elite)	1-3 min (automated)	DORA 2024 [4]
MTTR (median)	1-4 hours (typical)	5-15 min (60-75% reduction)	PagerDuty [6]
SLA Compliance	99.5-99.9% (typical)	>99.95% (target)	Gartner [2]
False Positive Alert Reduction	70-80% false positive	70-85% reduction	Gartner [2]
Infrastructure Cost Savings	25-40% overprovisioned	30-40% reduction	Flexera 2024 [27]
Target Agent Accuracy	N/A	>93% (with rollback safety)	Estimated [26]

Expected Accuracy Trajectory

Based on patterns observed in comparable autonomous systems, agent accuracy is expected to improve over time as the RAG knowledge base accumulates organizational knowledge [26]. Initial deployment (Phase 2) may exhibit accuracy in the 85-90% range, improving to 93-97% as the system processes more incidents and refines its OPA policies through rollback-driven feedback. The automated policy review workflow, triggered on every rollback event, provides the mechanism for this continuous improvement.

Cost-Benefit Projection

The framework's own infrastructure cost is estimated at $10,000-25,000 per month (comprising LLM API usage, agent compute, audit store, and event bus), depending on deployment scale. Against projected infrastructure cost reductions of 30-40% reported in industry benchmarks for intelligent auto-scaling and right-sizing, this represents a favorable return on investment for organizations with monthly cloud spend exceeding $100,000. The use of tiered model selection, assigning a smaller model for routine Watchtower monitoring and a more capable model for complex Elastik and Arbitrator reasoning, is expected to reduce LLM API costs by 30-50% compared to uniform model usage [27].

Implementation Recommendations

Start with Read-Only Agents

The most important deployment recommendation is to resist the temptation to enable autonomous actions prematurely. The 4-6week observation-only phase (Phase 1) generates baseline data, surfaces integration issues, and builds operator trust. Research on human-AI trust calibration suggests that teams that skip observation phases experience higher resistance to automation adoption [28].

MCP Schema Quality Determines Agent Quality

Agent reasoning quality is directly proportional to MCP tool description quality. Detailed, unambiguous tool schemas with explicit parameter constraints, clear error descriptions, and usage examples are expected to significantly reduce agent tool invocation errors. We recommend treating MCP tool schemas with the same rigor as public API documentation [29].

Tenant Isolation is Non-Negotiable

Implementing tenant scoping only at the agent reasoning level (i.e., instructing the agent to "only act on Tenant X") is insufficient. Research on LLM prompt injection vulnerabilities demonstrates that agents can be induced to cross boundaries under adversarial conditions [30]. The proposed architecture enforces isolation at the MCP server level, where tenant scope is bound to the MCP session and the server rejects any request outside that scope regardless of agent intent.

Invest in Agent Observability

Agents that cannot be observed cannot be trusted. The proposed system emits OpenTelemetry traces for every agent reasoning cycle, with spans for LLM inference, RAG retrieval, MCP tool

calls, and governance evaluation. A dedicated Grafana dashboard should display real-time agent activity, proposal approval rates, rollback trends, and per-tenant SLA headroom. This observability layer is expected to become the primary operational interface for DevOps teams, complementing traditional alert consoles [31].

Plan for LLM Provider Outages

LLM provider outages are an operational reality that must be accounted for in system design. The proposed architecture includes graceful degradation that automatically reverts to rule-based automation (pre-configured HPA policies and static alert rules) during LLM unavailability, ensuring no gap in performance management coverage. This fallback path should be designed and tested before any agent is promoted beyond Level 1 [32].

Conclusion

This paper has presented a practitioner-focused implementation blueprint for self-governing cloud performance management in multi-tenant environments. The key contribution is not the theoretical architecture, which builds on established multi-agent and AIOps foundations, but the detailed design decisions, integration patterns, governance policies, and deployment procedures that enable DevOps architects to translate the promise of agentic AI into operational reality.

Based on published industry benchmarks, the proposed framework projects a 60-75% MTTR reduction, SLA compliance exceeding 99.95%, and 30-40% infrastructure cost savings. The phased deployment strategy (observe, advise, assist, govern) provides a risk-managed path that builds organizational confidence incrementally. Future work will focus on validating these projections through controlled deployment studies.

The central argument of this work is that the primary barrier to autonomous cloud performance management is not algorithmic capability but operational integration. The agent reasoning layer is, in many respects, the simplest component to implement. The difficult engineering lies in the governance policies, the MCP server specifications, the tenant isolation enforcement, the rollback choreography, and the human-agent trust calibration. It is these practical concerns, not the AI models themselves, that determine whether an autonomous performance system delivers value or introduces risk.

Acknowledgment

The author acknowledges the open-source communities behind MCP, OpenTelemetry, Kubernetes, Open Policy Agent, and ArgoCD whose projects form the foundation of the proposed architecture. AI-assisted tools were used during the preparation of this manuscript. The author reviewed, edited, and validated all content and takes full responsibility for the accuracy and originality of the final work.

References

1. Waseem M, Liang P, Shahin M (2020) Systematic mapping study on microservices architecture in DevOps, J. Syst. Softw 170: 110750.
2. Gartner (2024) Market Guide for AIOps Platforms, Gartner Research, 2024. Available: https://www.gartner.com.
3. Wang Y, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, et al. (2024) A survey on large language model based autonomous agents, Frontiers Comput. Sci 18: 6.
4. DORA (2024) "Accelerate State of DevOps Report 2024," Google Cloud. Available: https://dora.dev.
5. Anthropic (2024) Model Context Protocol Specification Available: https://modelcontextprotocol.io.
6. PagerDuty (2024) "State of Digital Operations," PagerDuty Inc Available: https://www.pagerduty.com/resources.
7. Chen P (2020) Outage prediction and diagnosis for cloud service systems, in Proc. Web Conf. (WWW) 2659-2665.
8. IBM Research (2025) Towards safe agentic AI performance engineering," presented at SOSP Workshop Available: https://research.ibm.com.
9. NIST (2022) SP 800-204C: Implementation of DevSecOps for a Microservices-based Application with Service Mesh https://csrc.nist.gov/pubs/sp/800/204/c/final.
10. Anthropic (2024) MCP Transport: Streamable HTTP, MCP Specification Available: https://modelcontextprotocol.io/specification.
11. Burns B, Beda J, Hightower K, Evenson L (2022) Kubernetes: Up and Running, 3rd ed. Sebastopol, CA: O'Reilly https://digtvbg.com/files/LINUX/Kubernetes%20Up%20and%20Running,%203rd%20Edition.pdf .
12. Kleppmann M (2017) Designing Data-Intensive Applications. Sebastopol, CA: O'Reilly, https://unidel.edu.ng/focelibrary/books/Designing%20Data-Intensive%20Applications%20The%20Big%20Ideas%20Behind%20Reliable,%20Scalable,%20and%20Maintainable%20Systems%20by%20Martin%20Kleppmann%20(z-lib.org).pdf.
13. Kubernetes Authors (2024) Using RBAC Authorization," Kubernetes Documentation Available: https://kubernetes.io/docs/.
14. Brazil B (2022) Prometheus: Up and Running, 2nd ed. Sebastopol, CA: O'Reilly https://www.oreilly.com/library/view/prometheus-up/9781098131135/.
15. Argo Project (2024) Argo CD: Declarative GitOps CD for Kubernetes Available: https://argo-cd.readthedocs.io.
16. Open Policy Agent Authors (2024) OPA Documentation Available: https://www.openpolicyagent.org/docs/.
17. Beyer B, Jones C, Petoff J, Murphy NR (2016) Site Reliability Engineering. Sebastopol, CA: O'Reilly https://www.oreilly.com/library/view/site-reliability-engineering/9781491929117/.
18. 2023) AICPA SOC 2 Type II: Trust Services Criteria," American Institute of CPAs.
19. (2024) Open Telemetry Authors OpenTelemetry Collector. https://opentelemetry.io/blog/2024/ .
20. (2024) Helm Authors Helm: The Package Manager for Kubernetes. https://helm.sh/docs/.
21. (2018) Netflix Technology Blog Automated Canary Analysis at Netflix with Kayenta https://netflixtechblog.com.
22. Qu L (2021) Automatic cloud resource scaling algorithm based on long short-term network model Soft Comput 25: 4597-4613.
23. Reiss C, Tumanov A, Ganger G R, Katz R H, Kozuch M A (2012) Heterogeneity and dynamicity of clouds at scale: Google trace analysis in Proc. ACM SoCC Art. no 7.
24. Krebs R, Momm C, Kounev S (2012) Architectural concerns in multi-tenant SaaS applications," in Proc. CLOSER 426-431.
25. Sigelman B H, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, et al. (2010) Dapper, a large-scale distributed systems tracing infrastructure Google Tech. Rep.
26. Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, et al (2023) Survey of hallucination in natural language generation ACM Comput. Surv 55: 12- 248.
27. (2024) Flexera State of the Cloud Report Flexera https://www.flexera.com.

28. Lee J D, See K A (2004) Trust in automation: Designing for appropriate reliance Human Factors 46: 50-80.
29. P Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, et al. (2020) Retrieval-augmented generation for knowledge-intensive NLP tasks in Proc. NeurIPS 33: 9459-9474.
30. (2024) OWASP OWASP Top 10 for Large Language Model Applications1.1 https://owasp.org.
31. Datta A, Sen S, Zick Y (2016) Algorithmic transparency via quantitative input influence in Proc. IEEE S&P 598-617.
32. (2023) CNCF Cloud Native Resilience White Paper Cloud Native Computing Foundation https://www.cncf.io.