

Toward Self-Healing Cloud-Native Infrastructure for Production Large Language Model Workloads

Rajeev Chevuri

Campbellsville University, USA

Abstract: Production large language model (LLM) infrastructure is a cloud-native system that combines container orchestration, accelerated compute, model-serving runtimes, storage, networking, and deployment pipelines into a single operational surface, and a fault in any one layer can degrade the availability of the whole inference service. Reactive operations, in which monitoring raises an alert and an engineer manually diagnoses and resolves the condition, remain necessary for genuinely novel failures but scale poorly against the recurring, well-understood conditions that dominate day-to-day LLM operations. This paper proposes a structured alternative: a seven-domain failure taxonomy spanning the container, scheduling, node, accelerator, networking, storage, provisioning, and deployment-pipeline layers, paired with a six-layer reference architecture, instrumentation, correlation, classification, policy, execution, and validation and escalation, that governs when automated remediation is permitted and when a condition must reach a human operator. The architecture treats observability as a decision layer rather than a passive monitoring function, combining signals across layers to establish operational state and event correlation before any corrective action executes. Remediation itself is bounded: actions carry retry limits, validation checks, and reversibility requirements, and the architecture explicitly addresses how automation can itself introduce instability through repeated retry loops, conflicting controllers, or premature intervention during transient conditions. The paper argues that self-healing must be designed as an architectural capability from the outset rather than retrofitted after reliability problems emerge, and that automation quality, measured by whether actions are bounded, validated, and reversible, matters more than automation coverage. The resulting model gives infrastructure teams a reusable basis for scaling LLM operations without either relying entirely on manual intervention or ceding recovery decisions to unrestricted autonomous systems.

Keywords: Self-healing infrastructure; cloud-native systems; Kubernetes orchestration; large language model operations; automated remediation; observability

1. Introduction

A production Kubernetes cluster restarts an unhealthy pod automatically, but nothing in that same cluster automatically recognizes that the node underneath it has been silently starved of GPU memory for the past ninety minutes. This gap, between what container orchestration already reconciles and what an LLM-serving platform actually needs to stay healthy, is where most infrastructure teams still fall back on a page to an on-call engineer. Reactive operations, in which telemetry crosses a threshold, an alert fires, and a person investigates, remains the default operating model for infrastructure supporting large language model workloads even though a substantial share of the conditions that trigger those pages are recurring and already well understood. Formal treatments of production reliability practice, including Google's own account of site reliability engineering, have long argued that operations teams should treat repetitive manual work as a signal to automate rather than as an inevitable operating cost [3]; the same argument applies with particular force to LLM-serving infrastructure, where the operational surface is wider and the cost of a missed recovery window is higher than in a conventional web-serving environment.

Large language model platforms are not conventional web services. As cloud-native systems, they combine cloud infrastructure, container orchestration, accelerated compute, model-serving runtimes, storage, networking,



deployment pipelines, and application interfaces into a single operational surface, and a failure in any one of these layers can degrade the availability or performance of the entire inference service [4]. As the number of models, workloads, and concurrent users grows, an operating model that depends entirely on human diagnosis for every recurring condition becomes a structural bottleneck rather than a temporary inconvenience. Recent operational studies of large-scale training and inference clusters bear this out directly: infrastructure-related failures have been found to consume over 82 percent of total GPU-hours lost to failure while accounting for only 11 percent of failed job quantity [9], and analysis of production inference incidents shows that a substantial fraction of high-severity events still require manual intervention even where automated detection exists [11].

Existing reconciliation mechanisms address only part of this problem. Kubernetes already recreates unhealthy containers and reschedules failed pods, a capability that traces back to the reconciliation-loop design first articulated in Google's own transition from Borg to Kubernetes [1]. But container-level reconciliation says nothing about whether the node underneath that container has adequate accelerator capacity, whether the provisioning system can actually deliver a replacement node in time, or whether a cascading failure across the scheduling layer is about to turn a single unhealthy pod into a service-wide outage. The gap between what container orchestration already automates and what a production LLM platform actually needs is precisely where infrastructure teams still fall back on manual, reactive operations.

Closing that gap, this paper argues, requires treating self-healing as a structured engineering discipline rather than a collection of ad hoc scripts layered onto an existing cluster. The contribution here is twofold. First, the paper introduces a seven-domain failure taxonomy, covering the container, scheduling, node, accelerator, networking, storage and artifact, provisioning, and deployment-pipeline layers, that gives infrastructure teams a shared vocabulary for classifying where a given failure actually originates before deciding how to remediate it. Second, the paper proposes a six-layer reference architecture, instrumentation, correlation, classification, policy, execution, and validation and escalation, that binds that taxonomy to a disciplined, policy-governed remediation process. The emphasis throughout is on bounded automation: an architecture that recognizes common, well-understood failure patterns and remediates them automatically, while escalating genuinely uncertain conditions to a human operator rather than attempting unrestricted autonomous control.

2. Materials and Methods

This section develops the analytical framework in three parts. The first establishes a systematic method for classifying where LLM infrastructure failures originate. The second develops observability as the decision layer that supplies the evidence a remediation policy needs before it can act. The third describes how remediation policies bind classified failures to bounded, validated corrective actions. Figure 1 previews the six-layer reference architecture, instrumentation, correlation, classification, policy, execution, and validation and escalation, that the remainder of this section develops layer by layer.

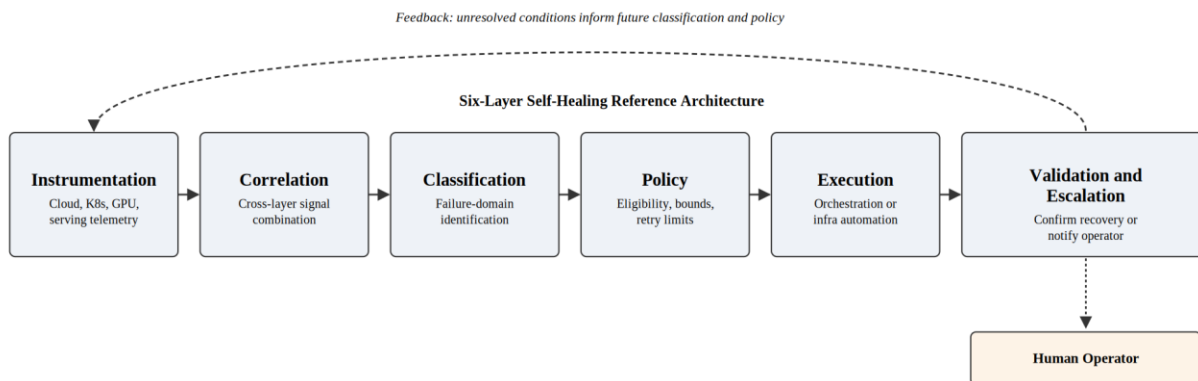


Figure 1. Six-layer self-healing reference architecture, showing the instrumentation-to-validation/escalation pipeline and the feedback path back to instrumentation.

2.1 Failure-Domain Classification Method

A self-healing platform cannot respond intelligently to a failure it cannot correctly locate. Building a remediation policy therefore starts with a systematic account of where failures actually occur across the LLM-serving infrastructure stack, rather than treating every incident as an undifferentiated "system down" event. The classification method applied here proceeds domain by domain: each of the eight operational surfaces identified below is characterized by its typical failure signature, the detection signal that distinguishes it from adjacent domains, and the remediation action it supports, following the same domain-then-signal-then-action structure summarized in Table 1.

The model-serving container layer is the first domain. Containers become unhealthy for reasons ranging from application errors and resource exhaustion to failed initialization and dependency problems, and Kubernetes health mechanisms already detect many of these conditions and recreate the affected workload [1]. Repeated restarts on the same container, however, are themselves a diagnostic signal: they suggest the underlying cause sits outside the container, in scheduling, node health, or an upstream dependency, rather than in the containerized process itself.

Deliberately exercising these failure paths before they occur in production, an approach formalized in the chaos-engineering literature as the practice of injecting controlled faults to verify that a distributed system tolerates the conditions it is expected to encounter [2], gives infrastructure teams confidence that the classification and remediation logic developed in this section actually behaves as intended rather than only appearing correct on paper. The Kubernetes scheduling layer forms a second, distinct domain. A workload can be entirely healthy in configuration and still fail to start because no node in the cluster has sufficient capacity to place it, a condition that is especially common for GPU-enabled workloads where suitable accelerator capacity may not be immediately available [4]. An unscheduled pod is a capacity signal, not an application failure, and conflating the two leads to remediation actions, such as repeatedly restarting a pod that was never unhealthy, that cannot possibly resolve the actual condition.

Node-level failures constitute a third domain, distinguishable from both container and scheduling failures because the appropriate remediation, draining and eventually replacing the node, differs from anything a container-level restart can achieve. A worker node can become unhealthy for reasons including operating system faults, network connectivity loss, resource exhaustion, or underlying cloud infrastructure events, and empirical studies of large GPU clusters show that node failures are neither rare nor evenly distributed: in one 63-node, 504-GPU training cluster, the top three nodes accounted for over 50 percent of all node-exclusion events, rather than failures spreading uniformly across the fleet [13].

The accelerator layer introduces a failure mode that conventional node health checks do not reliably surface. A node can remain nominally online while its GPU resources are degraded or unavailable, which means health checks scoped to CPU, memory, and network reachability may report a node as healthy even when it cannot actually serve inference workloads. Multi-signal detection approaches that go beyond any single metric have been shown necessary precisely because no individual telemetry signal reliably predicts every accelerator failure type across a large GPU fleet [13].

Networking constitutes a fifth domain. Model-serving services depend on communication between routing layers, containers, storage systems, and clients, and a connectivity fault anywhere in that path can present as an application-level failure even when the model server itself is functioning correctly. A self-healing design that assumes every failed inference request originates inside the model container will misclassify a meaningful share of its incidents.

Storage and model-artifact availability form a sixth domain, and this domain carries measurable latency consequences that are specific to large language model workloads. Retrieving and mounting multi-gigabyte model weights before a serving container becomes ready can dominate cold-start time: node-cached delivery of a 70-billion-parameter-class model artifact has been measured at roughly 11.7 seconds, compared with approximately 40.7 minutes when the same artifact is pulled through generic object storage, a difference of roughly two orders of magnitude that

has direct consequences for how quickly a self-healing platform can actually recover a failed inference pod [8]. Multi-tier checkpoint loading strategies that stage weights across GPU memory, DRAM, and SSD have similarly been shown to cut cold-start times by several times relative to naive loading for models across a range of parameter counts [12].

The infrastructure-provisioning layer is a seventh domain, distinct from scheduling because provisioning failures occur one step earlier, when a scaling system requests additional capacity and that request itself fails due to quota limits, configuration errors, or upstream dependency problems. A platform that interprets an unfulfilled provisioning request as a simple retry-until-success condition risks generating repeated failed attempts rather than surfacing the actual constraint to an operator.

Deployment pipelines constitute an eighth operational surface that, while not a runtime failure domain in the same sense as the seven above, is where a meaningful share of production incidents actually originate. Empirical analysis of industrial continuous-integration pipelines has found that roughly sixteen percent of executed jobs are rerun despite an initial success signal, and that silent failures, in which a job reports success while the underlying artifact or state is actually broken, account for a nontrivial share of those reruns [16]. A self-healing model that treats deployment-triggered instability as indistinguishable from steady-state runtime failure will misdiagnose the recovery path in exactly the cases where rollback, not remediation, is the correct response.

A concept that ties these domains together is failure propagation: a fault originating in one layer frequently produces symptoms in several others. Insufficient accelerator capacity, for instance, can leave workloads unscheduled, which reduces effective serving capacity, which increases request queue depth, which ultimately surfaces to operators as elevated user-facing latency. Table 1 summarizes the seven runtime failure domains, their characteristic detection signals, and the remediation actions each domain typically supports.

Failure Domain	Characteristic Detection Signal	Typical Remediation Action
Container	Repeated pod restarts, application-level errors	Recreate container (existing K8s reconciliation)
Scheduling	Pod remains unscheduled despite healthy configuration	Diagnose capacity vs. configuration mismatch
Node	Node health degraded; repeated failures across pods on same node	Drain workloads, replace node
Accelerator (GPU)	Node reports healthy but GPU degraded/unavailable	Multi-signal detection; isolate and replace accelerator
Networking	Application-level failures with healthy model server	Diagnose routing/connectivity path before container-level action
Storage / model artifact	Elevated cold-start latency, mount/retrieval failures	Node-local caching, multi-tier checkpoint loading
Provisioning	Scaling request fails (quota, config, dependency)	Surface constraint to operator rather than blind retry
Deployment pipeline	Instability immediately following a release; silent CI failures	Rollback or halt rollout rather than runtime remediation

Table 1. Failure Domains, Detection Signals, and Remediation Actions

2.2 Observability-as-Decision-Layer Method

Acting safely on an unverified assumption is exactly what a self-healing platform cannot afford to do, which is why observability has to go beyond conventional threshold-based monitoring and provide the correlated evidence needed to determine both whether a problem exists and which failure domain it most likely originates in.

The distinction between monitoring and observability is functional rather than semantic. Monitoring evaluates predefined conditions and raises alerts when a threshold is crossed; observability provides the broader context needed to understand relationships among infrastructure state, workload behavior, and service performance, which is what a remediation policy actually needs before it can select an action. A layered observability model for LLM environments spans four levels: cloud infrastructure telemetry describing compute availability, network health, and provisioning activity; Kubernetes-layer signals covering node status, pod health, scheduling events, and resource pressure; accelerator-layer telemetry describing GPU availability and utilization; and serving-layer signals reflecting request processing, queue depth, latency, and error rates.

Correlation, not raw signal volume, is the central methodological point here. A single elevated-latency signal does not, by itself, indicate whether the cause is model execution behavior, insufficient replica count, scheduling delay, compute scarcity, or network degradation. Combining signals resolves this ambiguity: if request queues grow while workloads remain unscheduled and accelerator capacity is simultaneously exhausted, the platform can infer a compute-scarcity condition; if workloads restart repeatedly immediately after a new deployment, the more likely cause is the deployed change itself rather than an unrelated infrastructure fault.

This leads to the concept of operational state as distinct from component-level health. A platform can have every individual component report a green health check while still being unable to satisfy actual workload demand, a distinction that matters because self-healing decisions should be driven by whether the system as a whole meets its service objective, not merely by whether each component individually reports healthy. Event correlation extends this further by considering temporal relationships among scaling events, deployment changes, node failures, and application errors together, so that a fault appearing immediately after a deployment is treated differently from an equivalent fault appearing during otherwise stable operation.

Signal confidence is the final element of the observability model. Automated remediation should not respond to every isolated anomaly, because transient fluctuations are routine in distributed systems and a policy that reacts to single-sample noise will itself become a source of unnecessary intervention. Persistence checks, repeated observations across a defined window, and requiring multiple correlated signals before triggering remediation all raise confidence that an observed condition reflects a genuine failure rather than momentary variance. Observability must also extend to the remediation process itself: when an automated action executes, the platform should record what condition triggered it, what action was taken, and whether the system returned to a healthy operating state, so that automated decisions remain auditable rather than opaque.

2.3 Policy-Driven Remediation Design Method

Once a failure has been classified with sufficient confidence, the architecture requires a controlled mechanism for selecting and executing the corresponding recovery action. A remediation policy defines what failure type is eligible for automated correction, what action may be taken, how many attempts are permitted, how success is validated, and under what condition the issue escalates to a human operator instead.

Container-level remediation is the most direct case: when an isolated container fails, Kubernetes' existing reconciliation behavior can recreate it automatically, extending a mechanism that already exists rather than introducing a new one [1]. Node-level remediation follows a different workflow, since restoring health requires draining workloads away from the affected node and, in a dynamically provisioned cluster, requesting a replacement resource before rescheduling; empirically observed retry-and-recovery patterns in large training clusters show that automated retry mechanisms achieved a 33.3 percent success rate across 73 retry attempts, roughly 2.7 times the 12.5 percent success rate observed for manual recovery attempts under comparable conditions [13]. Capacity-related remediation addresses the case where workloads remain unscheduled because of insufficient accelerator availability, triggering a provisioning request rather than a workload-level action; empirical tuning of the standard Kubernetes horizontal pod autoscaler against real workload traces has demonstrated that calibrating autoscaler parameters to an application's actual load profile, rather than relying on default thresholds, can reduce the number of instantiated compute nodes by roughly fifteen percent without degrading service performance, which illustrates the practical payoff available when

a remediation policy is tuned to observed conditions rather than left at generic defaults [7]. Deployment-related remediation follows yet another pattern: where service health degrades immediately following a release, the correct action is typically to halt the rollout or revert to the previously stable version, connecting deployment automation directly to operational health signals rather than treating the two as separate concerns [6].

Two design principles govern how these actions are bounded. First, remediation actions must carry explicit limits: a policy should not permit indefinite retries, unbounded infrastructure requests, or repeated execution of an action that has already failed, since unconstrained automation of this kind is itself a documented source of instability in distributed systems, manifesting as retry storms, thundering herds, and cascading failure across dependent components [5]. Second, remediation actions must be validated and, where possible, reversible: executing a corrective action does not by itself confirm recovery, and the architecture must confirm that the original failure condition has actually cleared before considering the incident resolved, while favoring actions such as deployment rollback that can be undone if they produce unexpected results. Table 2 in Section 3 compares three operating modes, reactive, assisted, and policy-driven automated, against these design principles to clarify where the boundary between appropriate automation and necessary human judgment should sit.

3. Results

Applying the failure-domain taxonomy and the six-layer architecture from Section 2 to the operational patterns reported in recent production and research literature yields two principal results: a clarified boundary between the three viable operating modes for LLM infrastructure remediation, and a set of concrete mechanisms by which unbounded automation becomes a source of instability rather than a solution to it.

3.1 Comparing Three Operating Modes

The literature on production infrastructure operations describes three broad operating modes that differ in where remediation authority sits. In traditional reactive operations, monitoring generates an alert and a human operator manually diagnoses and resolves the condition; this mode remains appropriate for conditions that are genuinely novel or insufficiently understood, but empirical incident analyses show it scales poorly against high incident volume, with seventy-fifth-percentile time-to-mitigation ranging from approximately 32 hours to approximately 49 hours across a large sample of production inference incidents [11]. In assisted operations, monitoring identifies a likely cause and recommends a remediation path without executing it, reducing diagnostic burden while preserving human judgment over the actual corrective action. In policy-driven automated remediation, the platform executes an approved corrective action once clearly defined conditions are met, without waiting for human confirmation on each occurrence.

Reliability studies of large-scale training clusters reinforce why this mode boundary matters operationally rather than only theoretically: analysis of operational data spanning eleven months and four million jobs across two large ML cluster environments found that large training jobs are disproportionately vulnerable to failure even though smaller jobs make up the majority of cluster activity, a distinction the authors captured through a dedicated failure taxonomy and an effective-training-time-ratio metric built specifically to estimate mitigation effectiveness across job sizes [10]. That kind of size-aware failure modeling is itself an instance of the classification discipline this paper's Section 2.1 taxonomy generalizes beyond training workloads to the full LLM-serving stack. Table 2 summarizes how these three modes compare across automation scope, operator burden, and recovery-latency characteristics, drawing on the reliability figures reported across the empirical sources reviewed in Section 2.

Operating Mode	Automation Scope	Operator Burden	Recovery-Latency Characteristic
Reactive (manual)	None — human diagnoses and executes all actions	High — every incident requires manual investigation	~32–49 hrs (p75 TTM) [11]

Assisted	Detection and recommendation only; human executes	Moderate — human confirms/executes recommended action	~90 min baseline before assistance [15]
Policy-driven automated	Detection, classification, and execution for well-understood conditions	Low for covered conditions; escalates uncertain cases	Seconds to low minutes for covered conditions [14]

Table 2. Comparison of Three Operating Modes

The evidence supports a mode-selection principle rather than a universal preference for one mode over the others: reactive operations remain necessary for conditions the platform cannot yet classify with confidence, assisted operations serve conditions where the classification is likely correct but the consequences of an incorrect action are severe, and policy-driven automation is appropriate specifically for the well-understood, recurring conditions identified in Section 2's failure taxonomy. Analysis of production inference incidents illustrates this boundary concretely: automated health-probe detection was found to auto-detect approximately 74 percent of incidents, yet approximately 28 percent still required a manual hotfix, indicating that even mature automated-detection environments retain a residual category of conditions that assisted or reactive handling must still cover [11]. Figure 2 illustrates the comparative recovery-time characteristics reported across the reactive, assisted, and policy-driven-automated literature reviewed above.

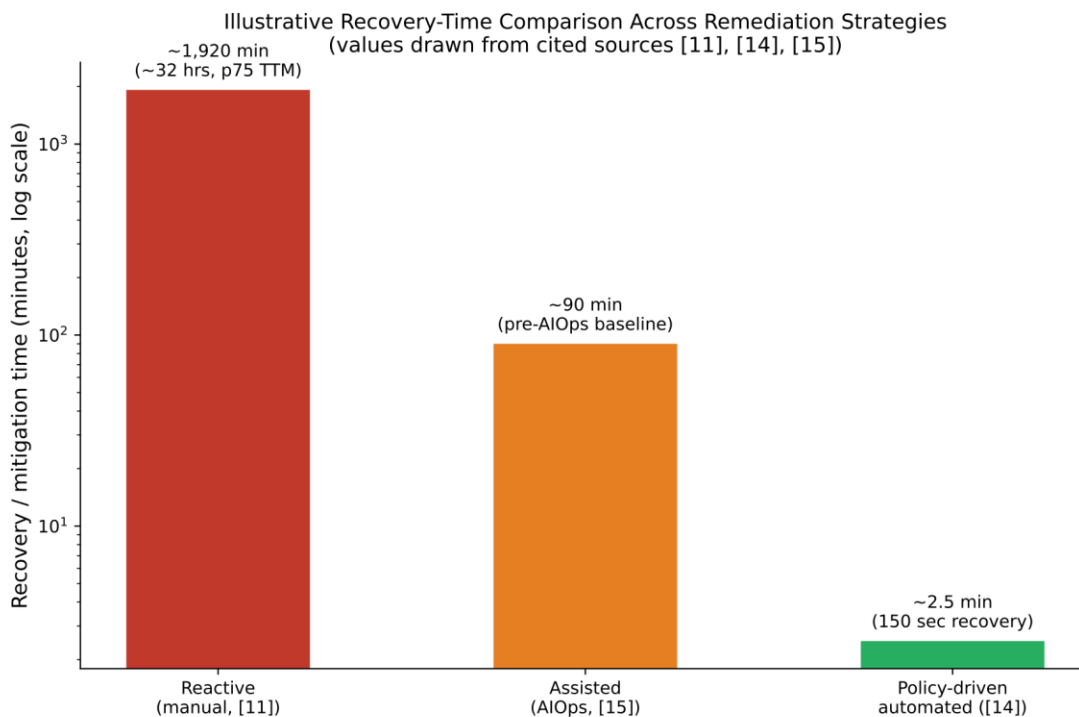


Figure 2. Illustrative recovery-time comparison across reactive, assisted, and policy-driven automated remediation strategies (log scale), drawn from the empirical sources reviewed in Sections 2 and 3.

3.2 Instability Mechanisms in Unbounded Automation

A second and separate result concerns the specific mechanisms through which automated remediation, if left unbounded, becomes a source of instability rather than a source of reliability, and three of these recur across the literature reviewed in Section 2.

The first is the repeated-retry loop. If a workload fails because of a persistent configuration error rather than a transient condition, restarting the affected container repeatedly does not correct the underlying cause and instead consumes cluster resources while generating operational noise that obscures the actual problem. Formal treatments of this failure mode in distributed systems describe it under the general heading of retry storms and identify architectural countermeasures, including exponential backoff with jitter, circuit breakers, and bulkhead isolation, each of which the six-layer architecture's policy layer can enforce directly by bounding retry count and introducing randomized backoff intervals [5].

The second mechanism is capacity-provisioning overreaction, where a workload fails to schedule because of an incorrectly specified resource request rather than an actual capacity shortage. A policy that cannot distinguish these two conditions will respond to a configuration problem by requesting additional infrastructure, which does not resolve the scheduling failure and instead increases operating cost without restoring service. This is precisely the distinction the failure-domain classification in Section 2.1 is designed to make explicit before any remediation action executes.

The third mechanism is conflicting automation, in which multiple independent controllers make uncoordinated decisions about the same workload or resource, one system scaling a service up while another simultaneously reduces its capacity, producing oscillation rather than convergence toward a stable state. The architecture's policy layer addresses this by assigning clear ownership of each remediation domain, container reconciliation to the orchestration layer, capacity requests to the provisioning layer, and rollout control to the deployment layer, so that no two automated systems act on the same failure domain without a defined arbitration path between them.

Empirical evidence for both the risk and the achievable mitigation of unbounded automation is available from production deployment-safety research. An integrated canary, blue-green, and feature-flag deployment framework evaluated across a production CI/CD pipeline reported a measured forty-percent improvement in mean time to recovery alongside system availability exceeding 99.98 percent, a result attributable specifically to bounding automated rollout actions with validation and rollback rather than allowing unconstrained progressive delivery [6]. Comparable discipline in remediation scope, bounding what an automated action is permitted to attempt and validating its outcome before proceeding, is the mechanism this paper's six-layer architecture generalizes across all seven failure domains rather than deployment automation alone. Single-step checkpoint restoration techniques developed for large-scale LLM training pipelines illustrate the same bounding principle applied to the recovery mechanism itself rather than only to the decision of whether to trigger it: an approach that eliminates dependence on traditional checkpoint replay was shown to restore a multi-thousand-device training cluster in well under three minutes regardless of cluster scale, which matters for this paper's architecture because a validation step that itself takes hours to complete undermines the practical value of fast failure detection upstream of it [14].

4. Discussion

The results in Section 3 support a central claim: the value of self-healing infrastructure for production LLM operations depends less on how many failure conditions are automated and more on whether the automation that exists is bounded, validated, and reversible. That claim follows directly from the failure-domain taxonomy and six-layer architecture developed in Section 2's Materials and Methods, since bounded automation is precisely what the policy and validation-and-escalation layers were designed to enforce. This section situates that claim against the broader literature on infrastructure reliability and considers the organizational and design implications that follow from it.

4.1 Self-Healing as an Architectural Capability, Not a Retrofit

A recurring theme across the reliability literature reviewed in Section 2 is that recovery mechanisms work best when designed into a platform from the outset rather than added after reliability problems have already surfaced in production. Kubernetes' own reconciliation model illustrates this: container-level self-healing exists because the reconciliation loop was a foundational design decision in the transition from Borg through Omega to Kubernetes, not an afterthought bolted onto an existing scheduler [1]. The seven-domain taxonomy and six-layer architecture proposed in this paper extend that same design principle across accelerator, provisioning, and deployment-pipeline layers that

current container-level reconciliation does not address, on the premise that recovery paths, telemetry, and escalation policy belong in the initial platform design rather than in a subsequent hardening effort.

LLM-serving platforms feel this design choice more acutely than most systems, given how measurably different the cost structure of failure recovery is at this scale. The order-of-magnitude difference between cached and non-cached model-artifact delivery documented in Section 2.1 illustrates why: a platform that did not anticipate cold-start latency as a first-class reliability concern cannot retrofit that concern away without redesigning its storage and artifact-distribution layer, whereas a platform that treats artifact delivery as part of its observability and remediation design from the start can incorporate node-local caching as a designed capability rather than a later patch [8].

4.2 Automation Quality Over Automation Coverage

The instability mechanisms identified in Section 3.2, retry storms, capacity overreaction, and conflicting controllers, all share a common root cause: automation that responds to a classified condition without the bounding, validation, and ownership discipline the six-layer architecture requires. This supports treating automation quality as the primary design variable rather than automation coverage. A platform that automates a narrow set of well-understood conditions with strict bounds, validation, and clear escalation paths is more reliable in practice than a platform that attempts to automate a broader set of conditions without those controls, because the latter risks converting a recoverable failure into a self-inflicted outage through exactly the mechanisms documented in the distributed-systems literature on self-induced denial of service [5].

How infrastructure teams sequence automation rollout follows directly from this principle. Rather than pursuing automation breadth first, and layering bounding and validation logic in afterward, the architecture proposed here suggests the reverse sequence: establish the observability and classification layers first, so that the platform can reliably distinguish failure domains and assess signal confidence, then extend automated remediation only to the specific conditions where classification confidence and validation coverage are both already established. Applied AIOps systems built along this same sequence, pairing an anomaly-detection ensemble with automated log correlation and a language-model-based synthesis layer, have been reported to reduce time-to-actionable-insight for cloud-infrastructure incidents from roughly ninety minutes to under five minutes in a five-month production deployment, while root-cause analyses reached 80 to 85 percent accuracy across assessed incidents, illustrating in practice how the observability-as-decision-layer method from Section 2.2 translates into measurable operator benefit once classification confidence is established [15].

4.3 Organizational Scalability and the Feedback Loop Between Operations and Platform Engineering

As the number of LLM applications an organization operates grows, manually monitoring every infrastructure component at all times becomes organizationally infeasible regardless of how skilled the operations team is. Standardized, reusable self-healing capabilities, provided at the platform level rather than reimplemented per application, allow operational teams to concentrate their attention on genuinely novel or high-impact failures rather than repeatedly diagnosing conditions the platform has already encountered and resolved before.

A natural feedback loop between operational experience and platform capability follows from this: an incident that initially requires manual intervention becomes, once its cause and recovery procedure are sufficiently well understood and validated, a candidate for inclusion in the policy-driven remediation layer. Over time, this feedback loop grows the library of conditions the platform can safely remediate on its own, while the human-operator escalation path continues to absorb the genuinely novel conditions that arise as the underlying technology, GPU generations, model architectures, serving frameworks, continues to change.

4.4 Limitations and Future Work

This paper presents a conceptual and architectural framework rather than an empirically validated deployment; the seven-domain taxonomy and six-layer architecture synthesize patterns documented across the reliability, autoscaling, and deployment-safety literature reviewed in Sections 2 and 3, but the architecture as an integrated whole

has not itself been benchmarked against a production LLM-serving deployment. The quantitative figures cited throughout this paper come from the specific empirical studies referenced at each point and describe those studies' own measured systems rather than a validation of this paper's proposed architecture directly. Future work should evaluate an implementation of the six-layer architecture against a production or representative LLM-serving workload, measuring whether the bounded-automation design principle actually reduces both incident count and the instability mechanisms documented in Section 3.2 relative to an unbounded or purely reactive baseline. A second direction concerns the provisioning and deployment-pipeline domains specifically, where the failure-propagation dynamics described in Section 2.1 are comparatively less studied in the existing literature than container- and node-level failures, suggesting these domains as a priority for future empirical work. This gap is visible even outside LLM-specific contexts: a large-scale empirical study of industrial continuous-integration pipelines found that a meaningful share of jobs reporting success were later rerun, with silent failures traced to causes such as artifact-handling errors and ignored exit codes rather than any condition the pipeline's own status reporting had surfaced, which is precisely the kind of deployment-adjacent failure mode the architecture proposed here classifies as a deployment-pipeline domain event rather than a runtime failure requiring container- or node-level remediation [17]. A third direction concerns environments where the six-layer architecture's assumption of centralized cluster-level observability does not hold as cleanly, such as edge and distributed IoT deployments, where limited local observability and intermittent connectivity have been identified as conditions requiring federated configuration management and localized alerting rather than the centralized model this paper assumes throughout [18].

5. Conclusions

Production large language model infrastructure is, at its core, a cloud-native system spanning container, scheduling, node, accelerator, networking, storage, provisioning, and deployment layers, and a fault in any one of them can degrade the service as a whole. This paper has proposed a seven-domain failure taxonomy and a six-layer reference architecture, instrumentation, correlation, classification, policy, execution, and validation and escalation, that together give infrastructure teams a structured basis for deciding when a recurring, well-understood failure can be remediated automatically and when a condition should still reach a human operator. The evidence reviewed here supports two central claims: that self-healing capability functions best when designed into a platform from the outset rather than retrofitted after reliability incidents accumulate, and that the quality of automation, whether actions are bounded, validated, and reversible, matters more than the breadth of conditions automated. Reliable production LLM operations depend not on eliminating failure, which distributed cloud systems will continue to produce regardless of design quality, but on building platforms that recognize common failure patterns, restore healthy operating conditions safely, and reserve human judgment for the conditions that genuinely require it.

Ethics Statement

Conflict of Interest: The author declares no conflict of interest.

Statement of Human and Animal Rights: This study did not involve human participants, animal subjects, or identifiable personal data requiring ethics board review.

Informed Consent: Not applicable.

CRedit Author Contributions

Rajeev Chevuri: Conceptualization, Methodology, Investigation, Writing – Original Draft, Writing – Review & Editing, Visualization.

REFERENCES

1. B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade," *Communications of the ACM*, 59(5), 2016.
2. A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. "Chaos Engineering," *IEEE Software*, 33(3), pp. 35–41, 2016.

3. B. Beyer, C. Jones, J. Petoff, and N. R. Murphy (eds.). *Site Reliability Engineering: How Google Runs Production Systems*, O'Reilly Media, 2016.
4. W. Gao, Q. Hu, Z. Ye, P. Sun, X. Wang, Y. Luo, T. Zhang, and Y. Wen. "Deep Learning Workload Scheduling in GPU Datacenters: Taxonomy, Challenges and Vision," *arXiv:2205.11913*, 2022.
5. T. Gelishvili, G. Kuchava, and T. Sturua. "Self-Inflicted Denial of Service (Self-DDoS) in Distributed Systems, Mechanisms and Mitigating Architectural Strategies: An Analysis of Retry Storms, Thundering Herds, and Cascading Failure," *Computational and Applied Science*, 1(2), pp. 141–148, 2026.
6. N. Alugunuri. "Progressive Delivery in CI/CD Pipelines: Evaluating Canary, Blue-Green, and Feature Flag Strategies," *International Journal of Intelligent Systems and Applications in Engineering*, 11(6s), pp. 922–933, 2023.
7. D. R. Augustyn, Ł. Wyciślik, and M. Sojka. "Tuning a Kubernetes Horizontal Pod Autoscaler for Meeting Performance and Load Demands in Cloud Deployments," *Applied Sciences*, 14(2), Article 646, 2024.
8. G. Kliukovkin. "Cold-Start Model Delivery in Kubernetes Inference Serving: An Empirical Study of OCI-Based Distribution and Its Integrity," *arXiv:2607.16596*, 2026.
9. Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, Y. Wen, and T. Zhang. "Characterization of Large Language Model Development in the Datacenter," in *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)*, 2024.
10. A. Kokolis, M. Kuchnik, J. Hoffman, A. Kumar, P. Malani, F. Ma, Z. DeVito, S. Sengupta, K. Saladi, and C.-J. Wu. "Revisiting Reliability in Large-Scale Machine Learning Research Clusters," *arXiv:2410.21680*, 2024.
11. B. Ranganathan, M. Zhang, and K. Wu. "Enhancing Reliability in AI Inference Services: An Empirical Study on Real Production Incidents," *arXiv:2511.07424*, 2025.
12. H. Ghosh. "Enabling Efficient Serverless Inference Serving for LLM in the Cloud," *arXiv:2411.15664*, 2024.
13. D. Kang, E. Hwang, H. Lee, H. Kim, H. Koo, J. Shin, J. Kang, J. Kang, J. Heo, J. Kim, J. Lee, J. Yang, K. Cho, and Y. Song. "From Detection to Recovery: Operational Analysis on LLM Pre-training with 504 GPUs," *arXiv:2605.09370*, 2026.
14. H. Zhang, J. Wang, Z. Yu, Y. Zhang, X. Ji, K. Mao, J. Zhang, Y. Zhang, T. Wu, F. Jie, X. Huang, Z. Cai, J. Cheng, S. Wang, W. Li, X. Bao, H. Xu, S. Zhao, J. Li, H. Sun, Z. Zhang, Y. Xiong, and C. Li. "FlashRecovery: Fast and Low-Cost Recovery from Failures for Large-Scale Training of LLMs," *arXiv:2509.03047*, 2025.
15. B. Kasinadhuni. "From Detection to Action: AI-Driven Anomaly Detection and Root Cause Synthesis for Cloud Infrastructure Operations," *International Journal of AI, BigData, Computational and Management Studies*, 7(2), pp. 171–184, 2026.
16. H. Aïdasso, F. Bordeleau, and A. Tizghadam. "On the Illusion of Success: An Empirical Study of Build Reruns and Silent Failures in Industrial CI," *arXiv:2509.14347*, 2025.
17. L. Zheng, S. Li, X. Huang, J. Huang, B. Lin, J. Chen, and J. Xuan. "Why Do GitHub Actions Workflows Fail? An Empirical Study," *ACM Transactions on Software Engineering and Methodology*, 2025.
18. H. Allam. "Reliability at the Edge: SRE for Distributed Cloud and IoT Platforms," *International Journal of Emerging Research in Engineering and Technology*, 6(2), pp. 39–52, 2025.