International Journal of AI, BigData, Computational and Management Studies
Noble Scholar Research Group | Volume 6, Issue 3, PP 137-147, 2025
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V6I3P116

Original Article

Runtime Causal Fingerprinting for Root-Cause-Aware Autoscaling and Adaptive Resource Governance in Kubernetes Service Meshes

Nilesh Mutyam
Senior Software Development Engineer, PayPal Inc, Dallas, TX, USA.

Received On: 23/06/2025 ***Revised On: 11/07/2025*** ***Accepted On: 24/07/2025*** ***Published On: 10/08/2025***

Abstract: *Kubernetes-orchestrated microservice systems increasingly operate under volatile workload patterns, strict latency service-level objectives, and complex inter-service dependencies introduced by service mesh communication layers. Although Kubernetes provides established autoscaling mechanisms, conventional reactive autoscaling policies frequently interpret symptoms such as CPU saturation, request latency, or queue growth as direct scaling triggers without distinguishing whether the observed degradation is caused by workload demand, downstream dependency propagation, resource contention, noisy-neighbor effects, configuration drift, or release-induced regression. This limitation often leads to unnecessary scaling, delayed remediation, over-provisioning, and repeated service-level objective violations. This paper proposes a runtime causal fingerprinting framework for root-cause-aware autoscaling and adaptive resource governance in Kubernetes service meshes. The framework constructs continuously updated causal fingerprints from distributed traces, service mesh telemetry, container metrics, event streams, and workload signals. These fingerprints encode not only anomaly signatures but also causal propagation patterns across services, pods, nodes, and resource dimensions. The proposed approach integrates causal graph inference, temporal fingerprint matching, SLO-risk estimation, and policy-constrained scaling actions within a closed-loop governance architecture. Unlike conventional autoscalers that scale based primarily on local resource utilization, the proposed framework differentiates demand-driven saturation from causally propagated degradation and selects remediation strategies accordingly. The paper develops the conceptual model, methodological pipeline, evaluation criteria, and analytical discussion necessary for implementing root-cause-aware autoscaling in production-grade Kubernetes environments. The study contributes a technically grounded framework for improving autoscaling precision, reducing false-positive scaling, mitigating cascading failures, and strengthening operational trust in cloud-native platforms.*

Keywords: *Kubernetes, Service Mesh, Autoscaling, Causal Inference, Root-Cause Analysis, Distributed Tracing, Microservices, Adaptive Resource Governance, SLO-Aware Resource Management, Cloud-Native Systems.*

1. Introduction

Cloud-native software systems are increasingly built as distributed microservice ecosystems deployed on Kubernetes clusters. This architectural shift improves modularity, release velocity, independent scalability, and infrastructure portability, but it also introduces operational complexity. A single user-facing transaction may traverse dozens of services, pass through sidecar proxies, interact with caches and databases, and trigger asynchronous events. When latency increases or error rates rise, the immediate symptom may appear at an upstream service even though the causal source is a downstream dependency, overloaded node, resource-throttled container, network policy misconfiguration, or release-related regression. Traditional autoscaling techniques were not designed to reason over this level of causal ambiguity, even though early autoscaling research recognized that elasticity must balance responsiveness, cost, and stability under changing workload conditions [1].

Kubernetes provides an extensible resource-control substrate through abstractions such as Deployments, ReplicaSets, Pods, Services, and control loops. The architecture inherits ideas from large-scale cluster management systems that emphasized declarative desired state, automated reconciliation, and workload placement across shared infrastructure [3]. However, Kubernetes autoscaling remains predominantly metric-triggered in many deployments. The Horizontal Pod Autoscaler commonly scales replicas based on CPU utilization or custom metrics, while vertical and cluster autoscaling mechanisms adjust pod resource requests or node capacity. These mechanisms are powerful but largely symptom-oriented: they react to observed utilization or demand without determining whether scaling is causally justified. In a service mesh, the problem becomes more subtle because sidecar proxies provide rich telemetry while also introducing additional latency, retry behavior, circuit-breaking policies, and traffic-shaping effects that can alter causal propagation.

The operational challenge is particularly acute for latency-sensitive applications. Tail latency has long been recognized as a dominant determinant of user experience in large-scale distributed systems, because even rare slow components can compound across fan-out request paths [11]. In microservice architectures, tail latency is not merely a local performance artifact; it is an emergent property of dependency graphs, queueing interactions, shared infrastructure, and runtime policies. Autoscaling a service that only manifests propagated latency may waste resources and fail to improve end-to-end SLO compliance. Conversely, failing to scale the true bottleneck may amplify cascading degradation. Therefore, the central research problem is not simply how to scale faster, but how to scale the causally relevant component with the appropriate resource action at the appropriate time.

Distributed tracing has made this problem more tractable by providing request-level visibility into cross-service interactions. Systems such as Dapper demonstrated that production-scale tracing can be deployed with low overhead and can support operational debugging across complex distributed systems [6]. Modern service meshes extend this visibility by emitting request metrics, response codes, retries, circuit-breaker events, and upstream-downstream latency distributions. Yet observability alone does not produce causality. A spike in latency, CPU throttling, retry count, and queue depth may occur together, but correlation does not identify whether CPU saturation caused retries, retries caused CPU saturation, or both were caused by a downstream dependency. Runtime autoscaling must therefore move from correlation-aware telemetry aggregation toward causal telemetry interpretation.

This paper proposes Runtime Causal Fingerprinting for Root-Cause-Aware Autoscaling and Adaptive Resource Governance, hereafter referred to as RCF-AG. The central idea is to represent runtime degradation patterns as causal fingerprints: structured, time-aware signatures that capture the probable causal origin, propagation path, affected service graph, resource dimension, anomaly morphology, and confidence level of a performance event. These fingerprints are not static labels. They are continuously updated representations derived from traces, service mesh metrics, Kubernetes events, container resource counters, and workload signals. The framework then uses the fingerprint to decide whether to scale a workload horizontally, adjust resource requests, rebalance traffic, isolate a release, throttle a caller, increase queue capacity, or suppress scaling when the symptom is propagated from another layer.

The paper makes four primary contributions. First, it formalizes the concept of runtime causal fingerprints as compact operational representations of root-cause evidence in service mesh environments. Second, it presents a root-cause-aware autoscaling pipeline that combines causal graph construction, temporal anomaly segmentation, fingerprint generation, and policy-constrained action selection. Third, it proposes an adaptive resource governance model that explicitly separates scaling decisions from broader remediation decisions, avoiding the assumption that every SLO risk should be handled by replica expansion. Fourth, it provides an evaluation framework for measuring autoscaling accuracy, SLO protection, causal localization quality, resource efficiency, and governance safety using realistic microservice benchmarks.

The motivation is reinforced by related research in software fault prediction and machine learning-based diagnosis. Deep learning models have shown value in learning complex defect and fault patterns from software artifacts, indicating that high-dimensional operational signals can be transformed into predictive representations when suitable modeling structures are used [2]. However, runtime resource governance requires more than prediction. It requires causal discrimination, because scaling decisions intervene in the system and may alter future telemetry. RCF-AG therefore treats autoscaling as a causal decision problem rather than a simple classification or thresholding problem.

2. Background and Related Work

2.1. Kubernetes Autoscaling and Elastic Resource Control

Autoscaling research has traditionally addressed how cloud systems dynamically add or remove resources to maintain performance while reducing cost. Surveys of cloud autoscaling identify core dimensions such as reactive versus proactive scaling, threshold-based versus control-theoretic policies, single-tier versus multi-tier applications, and application-level versus infrastructure-level metrics [15]. Kubernetes operationalizes autoscaling through controllers that observe metric values, compare them against desired targets, and reconcile deployment state by changing replicas or resource allocations. This design is effective for many stateless workloads, but it is less robust for service graphs in which local utilization is not sufficient to infer global performance impact.

Cluster management systems such as Borg established the importance of large-scale scheduling, resource isolation, admission control, and declarative workload management in production environments [7]. Kubernetes inherited several conceptual principles from this lineage, including control-loop reconciliation and workload abstraction. However, the autoscaling problem in service meshes is more complex than the earlier cluster utilization problem because modern services are highly interdependent, frequently released, and governed by SLOs rather than simple utilization targets. Scaling decisions can also create secondary effects, such as cache dilution, connection churn, cold starts, and database amplification.

Control theory provides another important foundation. Feedback-control approaches emphasize stability, overshoot avoidance, settling time, and control-loop delay, all of which remain relevant to Kubernetes autoscaling [9]. A controller that reacts aggressively to noisy latency signals may oscillate, while a conservative controller may miss the remediation window. In microservices, the feedback signal itself may be causally contaminated: a service's latency may be a delayed effect of another service's saturation. Therefore,

a stable autoscaler must reason not only about control parameters but also about whether the controlled variable is causally appropriate.

Recent resource-management research has advanced beyond simple threshold policies. SLO-oriented microservice frameworks such as FIRM show that fine-grained resource management can localize SLO violations and reprovision resources dynamically, reducing violations while improving utilization [24]. Graph-based autoscaling methods also demonstrate that service dependencies can improve resource estimation compared with isolated per-service decisions. Nevertheless, many approaches still treat dependency information primarily as a predictive feature rather than as a runtime causal object that can support root-cause-aware intervention.

2.2. Microservices, Service Meshes, and Observability

Microservices decompose applications into independently deployable services, but the resulting operational environment is fundamentally graph-structured. Researchers have observed that microservices create new challenges in performance monitoring, failure diagnosis, testing, and governance because failures propagate across service boundaries and because system behavior depends on dynamic call paths [25]. Service meshes further formalize service-to-service communication through sidecar proxies or data-plane components that provide observability, routing, retries, mutual TLS, rate limiting, and policy enforcement.

From an autoscaling perspective, service mesh telemetry is both an opportunity and a risk. It provides high-resolution request metrics, upstream/downstream latency, response status, retry counts, and traffic splits. These signals can reveal propagation paths that infrastructure metrics alone cannot capture. However, mesh behavior can also alter the apparent causal structure. For example, automatic retries may increase downstream load after the first signs of degradation. Circuit breaking may protect a dependency while increasing errors upstream. Traffic shifting during canary release may produce asymmetric latency distributions. A root-cause-aware autoscaler must therefore interpret mesh telemetry as part of the causal system rather than as passive observation.

Distributed tracing is a foundational mechanism for this interpretation. Dapper's design showed that large-scale tracing can expose request paths and timing relationships across distributed systems [6]. In service mesh environments, tracing can be combined with span-level metadata, proxy metrics, and Kubernetes runtime state. However, traces are often sampled, incomplete, or inconsistent across asynchronous boundaries. RCF-AG addresses this limitation by treating traces as one signal family within a multi-signal fingerprint rather than as the sole source of truth.

Benchmarking has also become essential for evaluating microservice resource-governance methods. DeathStarBench provides representative open-source microservice applications that expose realistic architectural complexity, including social networking, media services, e-commerce, banking, and IoT-style workloads [13]. Such benchmarks are valuable because they allow researchers to inject resource contention, workload bursts, network delays, and service failures while measuring end-to-end effects. RCF-AG is designed to be evaluated using similar benchmark environments because causal autoscaling must be validated under controlled but realistic fault and workload scenarios.

2.3. Root-Cause Analysis in Microservice Systems

Root-cause analysis has become a major research area in AIOps and cloud-native reliability. Early graph-based approaches such as CloudRanger constructed impact graphs and used random-walk mechanisms to rank likely culprit services in cloud-native systems [17]. These methods demonstrated the value of modeling dependency relationships rather than diagnosing services independently. However, autoscaling requires a further step: after identifying the likely culprit, the system must decide whether resource scaling is the right intervention and which resource dimension should be changed.

Microscope advanced causal-graph-based diagnosis for microservice environments by identifying abnormal services and ranking possible root causes [10]. This line of work is directly relevant to runtime causal fingerprinting because it shifts diagnosis from correlation ranking toward structural interpretation. Still, many diagnosis systems are designed primarily for human incident response rather than automated resource governance. The latency between diagnosis and action may be acceptable in manual operations but too slow for bursty workloads or cascading SLO risk.

MicroRCA further demonstrated that performance root causes can be localized using application-level response times and system-level resource metrics collected from microservice deployments [19]. This is important because resource governance requires connecting service-level symptoms to infrastructure-level causes. A service may show elevated latency because its own CPU is throttled, because a downstream service is saturated, because a node-level noisy neighbor is consuming shared resources, or because network delay has increased. RCF-AG incorporates this multi-level reasoning by fingerprinting anomalies across service, pod, node, and dependency layers.

Trace-based localization systems such as MicroRank use normal and abnormal traces to identify latency root causes through spectrum analysis and ranking [21]. Such approaches are valuable in environments where traces are reliable and latency issues are the dominant incident type. However, autoscaling decisions must account for resource cost, action reversibility, SLO urgency, and governance constraints. Therefore, RCF-AG uses trace-derived evidence as part of a broader action-selection framework rather than treating localization as the endpoint.

Causal inference-based root-cause analysis has also matured. CIRCA formulates root-cause diagnosis as an intervention-recognition problem using causal Bayesian

networks and identifies root-cause indicators by detecting distributional changes conditioned on causal parents [14]. This perspective is especially relevant to autoscaling because scaling is itself an intervention. An autoscaler changes replica count, CPU request, memory allocation, or traffic distribution, thereby modifying the future causal structure of the system. A root-cause-aware autoscaler should therefore predict not only whether a component is anomalous but also whether an intervention on that component is expected to reduce SLO risk.

2.4. Predictive and Learning-Based Autoscaling

Predictive autoscaling uses workload forecasts, resource-demand models, or learned policies to allocate capacity before degradation occurs. Graph neural network-based autoscaling frameworks such as GRAF capture microservice dependencies and SLO requirements to proactively allocate resources [18]. These approaches are important because microservice resource demand is not independent; a workload change at one entry point can propagate unevenly through downstream services. However, proactive resource prediction does not automatically resolve root-cause ambiguity. A predicted increase in downstream CPU may reflect true demand growth, retry amplification, or inefficient release behavior.

DeepScaler similarly addresses holistic autoscaling by using spatiotemporal graph neural networks and adaptive graph learning to estimate resource requirements under dynamic dependencies [23]. This demonstrates the value of learning hidden service interactions rather than relying exclusively on static architecture. RCF-AG builds on this insight but focuses on causal fingerprints as operationally interpretable objects. The distinction is important: a graph model may predict that a service needs more resources, while a causal fingerprint explains why the service appears resource-constrained and whether scaling it is the right governance response.

Smart HPA proposes a resource-efficient horizontal pod autoscaler that improves upon baseline Kubernetes HPA in resource-constrained microservice architectures [20]. This work highlights that autoscaling should be coordinated across services rather than treated as isolated per-workload adjustment. RCF-AG extends this direction by introducing root-cause-aware coordination. If two services are degraded, the framework should determine whether both require scaling or whether one is the causal source and the other is merely a victim of propagation.

Learning-based software diagnosis also informs the proposed framework. Machine learning studies on fault diagnosis using decision trees and K-nearest neighbor models illustrate how interpretable and instance-based methods can support diagnostic classification when the feature space is well constructed [8]. Ensemble methods for defect prediction further suggest that combining multiple weak or partial signals can improve robustness under heterogeneous conditions [12]. RCF-AG adopts a similar philosophy for runtime telemetry: causal fingerprints should fuse traces, metrics, events, logs, and mesh signals rather than depend on a single metric stream.

3. Problem Statement

The central problem addressed in this paper is the absence of root-cause awareness in runtime autoscaling and resource governance for Kubernetes service meshes. Existing autoscaling policies often answer the question: “Which metric crossed a threshold?” RCF-AG instead asks: “Which component and resource dimension are causally responsible for SLO risk, and what intervention is most likely to reduce that risk without causing instability or waste?”

Let a Kubernetes service mesh be represented as a dynamic directed graph (G_t = (V_t, E_t)), where each node (v V_t) denotes a service, workload, pod group, or infrastructure entity, and each edge (e E_t) denotes runtime communication, dependency, or resource-sharing relation. Let (X_t) represent observed telemetry at time (t), including request rate, latency percentiles, error rate, retry count, CPU usage, memory pressure, throttling, queue depth, network delay, pod restarts, deployment version, and node-level contention. Let (Y_t) represent SLO risk, such as the probability of violating a p95 or p99 latency objective within a future time horizon.

The root-cause-aware autoscaling problem can be formulated as selecting an intervention (a_t A), where (A) includes horizontal scaling, vertical resource adjustment, traffic shifting, rate limiting, queue tuning, canary rollback, pod eviction, or no-action. The optimal intervention should minimize expected SLO risk and resource cost while satisfying governance constraints:

[a_t^* = _{a A} [R(Y_{t:t+h} do(a), F_t)] + C(a) + S(a)]

Where (F_t) is the runtime causal fingerprint, (C(a)) is the resource or operational cost of the action, and (S(a)) is a stability penalty capturing oscillation risk, policy violation, or unsafe intervention. The use of (do(a)) emphasizes that autoscaling is an intervention, not merely a prediction problem, following the broader causal reasoning tradition in which interventions must be distinguished from passive observations [4].

The research gap is therefore threefold. First, existing autoscaling systems often lack explicit causal representation of degradation patterns. Second, existing RCA systems often stop at localization and do not translate root-cause evidence into safe autoscaling or resource-governance actions. Third, current service mesh observability pipelines provide rich telemetry but do not consistently transform it into compact, action-oriented causal fingerprints that can be used by controllers.

The proposed framework addresses these gaps by introducing runtime causal fingerprints as the intermediate representation between observability and governance. A fingerprint summarizes the causal identity of an incident or emerging risk: source component, affected path, resource dimension, temporal shape, propagation direction, release

context, confidence score, and recommended intervention class. This representation allows the autoscaling controller to distinguish among superficially similar telemetry patterns. For example, high CPU and high latency in a service may produce one fingerprint when caused by legitimate demand growth, another when caused by retry storms, another when caused by inefficient new code, and another when caused by node-level contention. Each case requires a different governance response.

4. Proposed Framework / Methodology

4.1. Overview of RCF-AG

The proposed Runtime Causal Fingerprinting and Adaptive Governance framework consists of six methodological stages: multi-signal telemetry ingestion, topology-aware causal graph construction, anomaly segmentation, causal fingerprint generation, intervention selection, and feedback-based governance learning. The framework operates continuously and integrates with Kubernetes control loops, service mesh telemetry, and policy engines.

The first stage ingests telemetry from Prometheus-compatible metrics, OpenTelemetry traces, service mesh proxies, Kubernetes events, container runtime statistics, node exporters, and deployment metadata. Unlike conventional autoscalers that focus mainly on CPU or memory, RCF-AG treats telemetry as a multi-layer observational record. Each observation is aligned by timestamp, service identity, pod identity, node identity, version, namespace, and request path. This alignment is critical because causal reasoning fails when signals are aggregated at incompatible levels.

The second stage constructs a topology-aware causal graph. The graph includes service invocation edges from traces, runtime traffic edges from service mesh metrics, resource-sharing edges from pod-node placement, deployment edges from release metadata, and policy edges from mesh configuration. This graph is dynamic: edges may appear or disappear as traffic shifts, deployments roll out, pods restart, or services scale. The graph is not assumed to be fully causal at construction time; rather, it provides structural constraints for causal inference and fingerprinting.

The third stage detects and segments anomalies. Instead of treating every threshold violation as an autoscaling trigger, RCF-AG identifies anomaly windows and classifies their morphology: gradual saturation, sudden spike, retry amplification, cold-start delay, memory pressure, throttling burst, downstream propagation, or release-correlated regression. This segmentation reduces noise and improves causal interpretation by grouping related observations into coherent runtime episodes.

The fourth stage generates causal fingerprints. A fingerprint is defined as:

$$[F_t = r, p, d, m, , c, q,]$$

Where (r) is the probable root component, (p) is the propagation path, (d) is the dominant resource dimension, (m) is anomaly morphology, () is temporal lag structure, (c) is causal confidence, (q) is SLO-risk estimate, and () is the recommended action class. The fingerprint is intentionally compact so that it can be consumed by runtime controllers, dashboards, and incident workflows.

The fifth stage selects an intervention. If the fingerprint indicates demand-driven CPU saturation at the service where latency originates, horizontal scaling may be appropriate. If it indicates CPU throttling due to low container requests, vertical adjustment may be more effective. If it indicates downstream database saturation, scaling upstream callers may worsen the problem. If it indicates release-induced regression, rollback or traffic reduction may be safer than scaling. If it indicates retry amplification, mesh retry policy adjustment may be required. Thus, RCF-AG reframes autoscaling as one action within adaptive resource governance rather than as the universal response to performance risk.

The sixth stage updates governance knowledge after intervention. The framework observes whether the selected action reduced SLO risk, shifted the bottleneck, increased cost, or caused oscillation. This feedback updates fingerprint-action mappings and improves future decisions. This idea is consistent with machine learning and simulation-based approaches in high-performance and scientific computing environments, where iterative model refinement is essential for improving prediction and decision quality under complex system behavior [5].

4.2. Causal Fingerprint Construction

Causal fingerprint construction begins by identifying an anomaly episode (E), defined as a bounded time interval in which one or more SLO indicators deviate from baseline. For each episode, the framework extracts a subgraph (G_E) containing services, pods, nodes, and dependencies active during the episode. The subgraph includes both request-path dependencies and resource-sharing dependencies. For example, two services that do not call each other may still be causally related through node-level CPU contention.

The framework then computes temporal precedence relationships. If downstream latency increases before upstream latency, the downstream service becomes a stronger root-cause candidate. If upstream request rate increases before downstream CPU saturation, the workload source becomes relevant. If pod restarts precede error spikes, runtime instability becomes part of the fingerprint. If a new deployment version appears shortly before the anomaly, release context increases in weight. These temporal relationships are not sufficient for causality, but they provide evidence when combined with graph constraints and conditional independence tests.

Next, the framework estimates causal contribution scores. For each candidate root component (v), the system evaluates how much of the SLO-risk signal remains unexplained after conditioning on its parents in the runtime graph. Distributional shifts conditioned on causal parents are

especially informative because they help distinguish root indicators from downstream effects. This logic is aligned with intervention-recognition approaches that identify root-cause variables by examining conditional distribution changes [14].

The fingerprint also includes anomaly morphology. Consider two cases with the same p99 latency violation. In the first, CPU utilization rises gradually with request rate, queue depth increases, and latency follows a queueing curve. This indicates demand-driven saturation. In the second, request rate remains stable, but latency increases after a version rollout and CPU instructions per request rise. This indicates release-induced inefficiency. In the third, upstream latency increases after downstream errors trigger retries. This indicates retry amplification. A conventional autoscaler may scale the upstream service in all three cases. RCF-AG produces different fingerprints and therefore different actions.

4.3. Action Mapping and Governance Constraints

The action-mapping layer translates fingerprints into interventions. A governance policy defines permissible actions by namespace, workload criticality, deployment stage, cost budget, SLO priority, and safety constraints. For example, production payment services may permit conservative scale-out but disallow automatic rollback without approval. Batch services may permit aggressive downscaling but require queue-depth protection. Canary workloads may permit traffic reduction when release fingerprints appear.

RCF-AG supports five main action classes. The first is horizontal scaling, which changes replica count. The second is vertical adjustment, which changes CPU or memory requests and limits. The third is traffic governance, including load shedding, rate limiting, retry adjustment, and traffic shifting. The fourth is release governance, including canary pause, rollback recommendation, or progressive delivery adjustment. The fifth is suppression, where the system deliberately avoids scaling because the symptom is propagated or the confidence is insufficient.

This action taxonomy prevents a common autoscaling failure mode: treating every performance anomaly as a capacity shortage. If the causal fingerprint indicates that an upstream service is slow only because a downstream service is timing out, scaling the upstream service may increase pressure on the downstream dependency. If the fingerprint indicates memory leak behavior, horizontal scaling may temporarily hide the issue but increase aggregate memory cost. If the fingerprint indicates node-level contention, pod rescheduling may be more effective than service scaling.

4.4. Integration with Kubernetes Controllers

RCF-AG can be implemented as a Kubernetes operator that watches telemetry streams and emits scaling recommendations through custom resources. A Custom Resource Definition may define CausalScalingPolicy objects, each specifying target workloads, SLO objectives, allowed actions, minimum and maximum replicas, cost budgets, confidence thresholds, and fallback policies. The operator computes fingerprints and writes CausalScalingDecision resources that can be audited.

Integration with existing autoscaling mechanisms can occur in three modes. In advisory mode, RCF-AG provides recommendations to human operators and dashboards. In guarded mode, it acts as a policy layer that approves, suppresses, or modifies HPA decisions. In autonomous mode, it directly executes scaling and governance actions within predefined safety constraints. The guarded mode is likely most appropriate for early production adoption because it improves autoscaling intelligence without fully replacing existing Kubernetes controllers.

The framework is also compatible with hierarchical scaling approaches. Multi-level Kubernetes scaling frameworks show that centralized and local controllers can be combined to coordinate decisions across services [20]. RCF-AG can provide the causal context needed by such controllers, allowing local service autoscalers to act only when the fingerprint indicates local causality and allowing global controllers to coordinate interventions across dependency paths.

5. System Architecture or Conceptual Model

The proposed architecture contains seven logical layers. The first layer is the telemetry collection layer. It receives service mesh metrics, distributed traces, Kubernetes events, pod metrics, node metrics, and release metadata. Each telemetry record is normalized into a common schema containing timestamp, workload identity, service identity, pod identity, node identity, version, request path, and metric family.

The second layer is the runtime graph layer. It constructs a multiplex graph with several edge types: request invocation edges, retry edges, traffic-shift edges, resource co-location edges, deployment-version edges, and policy edges. This graph differs from a static service dependency map because it reflects actual runtime behavior. For example, during canary deployment, only a subset of traffic may reach a new version; during failure, retry edges may intensify; during autoscaling, pod placement may change node-level resource relations.

The third layer is the anomaly and SLO-risk layer. This component computes service-level and path-level SLO risk using latency percentiles, error-budget burn rate, saturation metrics, and workload forecast residuals. It detects not only current SLO violations but also emerging risks. This is essential because autoscaling actions have delay: new pods need scheduling, image availability, readiness checks, warm-up, and load-balancer propagation. Therefore, the framework must act before SLO risk becomes irreversible.

The fourth layer is the causal inference and fingerprinting layer. This component generates candidate causal graphs, computes temporal lags, identifies

propagation direction, scores root candidates, and emits fingerprints. It uses structural constraints from topology, temporal precedence from time series, and conditional shifts from metric distributions. The output is not a black-box anomaly label but a structured explanation suitable for governance.

The fifth layer is the policy and action layer. This component maps fingerprints to actions while enforcing constraints. It ensures that scaling stays within replica limits, avoids unsafe rapid oscillation, respects cost budgets, and prevents conflicting actions. If confidence is low, it may select a safe exploratory action or request human approval. If confidence is high and SLO risk is severe, it may execute immediate scale-out or traffic adjustment.

The sixth layer is the feedback-learning layer. After intervention, it evaluates the effect on latency, error rate, resource usage, and downstream pressure. If the action reduces SLO risk without excessive cost, the fingerprint-action mapping is reinforced. If the action fails, the mapping is weakened and the system records a counterexample. This layer allows the framework to adapt to changing architectures, workload patterns, and release behavior.

The seventh layer is the audit and explainability layer. Every decision includes a fingerprint summary, causal evidence, action rationale, expected effect, confidence score, and policy constraints. This auditability is essential for production adoption because platform engineers must trust automated controllers. It also supports post-incident review by showing why the controller scaled, suppressed scaling, shifted traffic, or recommended rollback.

6. Evaluation Criteria / Performance Metrics

A rigorous evaluation of RCF-AG should measure both autoscaling performance and causal diagnosis quality. Conventional autoscaling metrics such as average CPU utilization or replica count are insufficient because the framework's goal is not simply to reduce resource usage but to improve root-cause-aware governance. The evaluation should therefore include six metric categories.

The first category is SLO protection. Relevant metrics include SLO violation rate, violation duration, error-budget burn reduction, p95 and p99 latency, and recovery time. Because tail latency dominates user experience in large-scale systems, p99 latency should be treated as a primary metric rather than an auxiliary statistic [11].

The second category is causal localization quality. Metrics include top-1 root-cause accuracy, top-k recall, mean reciprocal rank, causal path accuracy, and resource-dimension accuracy. Resource-dimension accuracy is especially important: identifying the correct service is not enough if the system cannot distinguish CPU throttling, memory pressure, network delay, retry amplification, or release regression.

The third category is autoscaling precision. This includes true-positive scaling actions, false-positive scaling suppressions, false scale-outs, missed scale-outs, action latency, and intervention effectiveness. A false-positive scale-out occurs when the system adds replicas to a service that is not causally responsible for SLO risk. Reducing such actions is one of the core objectives of RCF-AG.

The fourth category is resource efficiency. Metrics include CPU-seconds, memory-hours, replica-minutes, over-provisioning ratio, under-provisioning ratio, and cost per SLO-compliant request. These metrics must be interpreted jointly with SLO performance. A system that minimizes cost but violates SLOs is not successful; a system that eliminates violations through extreme over-provisioning is also not optimal.

The fifth category is control stability. Metrics include oscillation frequency, scaling reversal rate, settling time, cooldown violations, and action conflict rate. Control stability is essential because aggressive autoscaling can create instability, especially when multiple services react simultaneously to propagated symptoms.

The sixth category is governance explainability. Metrics include decision trace completeness, fingerprint interpretability score, policy compliance rate, and post-incident usefulness as assessed by operators. Although explainability is difficult to quantify, it is necessary for high-trust automation. Root-cause-aware autoscaling should provide a clear rationale for why a specific action was selected.

The experimental design should include workloads with normal diurnal patterns, sudden bursts, traffic skew, downstream bottlenecks, retry storms, node contention, memory leaks, CPU throttling, and canary regressions. Benchmark platforms such as DeathStarBench provide a suitable basis for these scenarios because they include realistic multi-service applications and allow controlled performance experiments [13].

7. Results and Discussion / Analytical Discussion

Because RCF-AG is proposed as a conceptual and methodological framework, the expected results should be evaluated analytically and through controlled prototype experiments rather than asserted as production outcomes. The primary hypothesis is that causal fingerprinting reduces unnecessary autoscaling and improves SLO recovery by distinguishing root causes from propagated symptoms.

In a demand-driven saturation scenario, RCF-AG should behave similarly to a well-tuned proactive autoscaler. If request rate increases at the entry point, CPU utilization rises in the target service, queue depth grows, and latency increases after saturation, the fingerprint should classify the event as local demand-driven resource exhaustion. The appropriate action is horizontal scaling or vertical adjustment, depending on workload characteristics. In this

case, causal fingerprinting should not delay scaling; rather, it should confirm that scaling is causally justified.

In a downstream bottleneck scenario, conventional HPA may scale upstream services because they show elevated latency or increased concurrency. RCF-AG should instead identify that latency originates downstream and propagates upstream through request waiting time. The correct action may be scaling the downstream service, adjusting connection pools, reducing retry pressure, or applying backpressure at the caller. This distinction is essential because upstream scale-out can intensify downstream overload and worsen the incident.

In a retry amplification scenario, the service mesh may automatically retry failed requests, causing request volume to increase beyond original user demand. A conventional autoscaler may interpret the increased request rate as organic demand and add replicas. RCF-AG should fingerprint this as mesh-policy-induced amplification if retry counts, downstream errors, and traffic expansion align temporally. The proper governance response may involve reducing retry budgets, enabling circuit breaking, or scaling the true downstream bottleneck.

In a release-regression scenario, a new version may increase CPU time per request or introduce inefficient database queries while external workload remains stable. Scaling may temporarily reduce latency but hides the regression and increases cost. RCF-AG should include version metadata in the fingerprint and recommend canary pause, rollback, or traffic reduction when release correlation is strong. This connects autoscaling with release governance, which is necessary because many production incidents arise not from workload growth but from software change.

In a node-contention scenario, multiple pods co-located on a node may experience CPU steal, memory pressure, or network contention. Scaling the affected service may or may not help, depending on whether new replicas land on healthier nodes. RCF-AG should identify node-level resource-sharing edges and recommend rescheduling, tainting, pod anti-affinity, or node scaling. This demonstrates why service-level autoscaling must be integrated with infrastructure governance.

Compared with purely predictive graph autoscalers, RCF-AG's expected advantage lies in intervention specificity. Graph neural network-based approaches can predict resource needs by learning dependency-aware representations [18]. However, a prediction that resource demand will increase does not explain whether the cause is legitimate demand, retry amplification, or release regression. Causal fingerprints provide a decision-oriented explanation that can reduce unsafe or wasteful actions.

Compared with RCA-only systems, RCF-AG's expected advantage lies in closed-loop actionability. Systems such as Sage show that machine-learning-driven performance debugging can identify root causes and support corrective actions in microservice environments [16]. RCF-AG extends this logic specifically into Kubernetes service meshes, where actions must be expressed as Kubernetes and mesh governance operations subject to policy constraints, auditability, and SLO budgets.

The analytical trade-off is computational overhead. Causal inference over high-cardinality telemetry can be expensive, especially in large clusters with thousands of services and millions of traces. RCF-AG addresses this by fingerprinting anomaly episodes rather than continuously analyzing every metric at maximum granularity. It also limits graph expansion to the affected request paths, dependency neighborhoods, and co-location groups. This design should keep runtime overhead compatible with production observability systems.

Another trade-off is confidence calibration. Incorrect causal fingerprints may suppress necessary scaling or trigger inappropriate actions. Therefore, RCF-AG should include conservative fallbacks. When confidence is low and SLO risk is high, the system may permit safe scale-out while marking the action as low-confidence. When confidence is high that a symptom is propagated, the system may suppress local scaling and act on the root component. This confidence-aware behavior is critical for operational safety.

Overall, the expected result is not that RCF-AG always scales less. Rather, it should scale more precisely. It should scale aggressively when the fingerprint indicates causal capacity shortage, avoid scaling when symptoms are propagated, and select non-scaling governance actions when the causal mechanism is release, retry, contention, or policy related.

8. Practical Implications

The proposed framework has several practical implications for platform engineering, site reliability engineering, and cloud cost governance. First, it can reduce infrastructure waste caused by false-positive scaling. In large Kubernetes environments, unnecessary scale-out across multiple services can significantly increase cost, especially when workloads run on reserved high-memory or high-CPU nodes. By identifying whether a service is a causal source or a propagation victim, RCF-AG can prevent replica expansion that does not improve SLOs.

Second, it can improve incident response. Instead of presenting operators with disconnected dashboards, the framework produces causal fingerprints that summarize root component, propagation path, resource dimension, and recommended action. This can reduce mean time to understanding, even when automatic remediation is disabled. Survey research on anomaly detection and root-cause analysis emphasizes that modern multi-service applications require structured techniques to detect failures and identify likely root causes because manual interpretation becomes increasingly difficult at scale [22].

Third, it supports safer progressive delivery. Release metadata is often separated from autoscaling telemetry. RCF-AG integrates deployment version into causal fingerprints, enabling the platform to distinguish workload-induced saturation from release-induced inefficiency. This allows autoscaling systems to cooperate with canary and rollout controllers rather than masking regressions with additional resources.

Fourth, it improves SLO governance. Many organizations define SLOs but operate autoscalers using low-level infrastructure metrics. RCF-AG bridges this gap by treating SLO risk as the primary objective while using causal telemetry to select interventions. This enables resource governance to be aligned with user experience rather than cluster utilization alone.

Fifth, it enables policy-aware automation. Different services require different risk tolerances. A customer-facing checkout service may justify rapid scale-out under uncertainty, while an internal analytics service may prioritize cost. RCF-AG's policy layer allows organizations to encode such differences explicitly.

9. Limitations

RCF-AG has several limitations. First, causal inference from observational telemetry is inherently uncertain. Even with traces, metrics, and topology constraints, some causal relationships may remain ambiguous. Hidden confounders such as external dependencies, shared databases, cloud-provider network events, and kernel-level scheduling behavior may distort fingerprints.

Second, service mesh telemetry may be incomplete or biased. Sampling can remove rare but important traces. Sidecar proxies may aggregate metrics at levels that obscure pod-specific behavior. Asynchronous messaging can break request-path continuity. These issues require careful instrumentation and confidence-aware interpretation.

Third, the framework may introduce operational complexity. Implementing causal fingerprinting requires telemetry normalization, graph maintenance, policy design, and integration with existing autoscalers. Organizations with immature observability practices may need substantial platform investment before adopting autonomous governance.

Fourth, causal action mapping may be environment-specific. A fingerprint-action rule that works in one architecture may fail in another because of different workload patterns, caching layers, database constraints, or deployment policies. The feedback-learning layer partially addresses this, but initial calibration remains necessary.

Fifth, governance safety constraints can reduce responsiveness. If policies require human approval for certain actions, RCF-AG may identify the correct action but still be unable to execute it automatically. This limitation is appropriate in high-risk systems, but it means benefits may vary by operational maturity.

10. Future Research Directions

Future research should investigate several extensions. First, causal fingerprint portability remains an open problem. Researchers should examine whether fingerprints learned in one microservice application can transfer to another with different topology but similar failure morphology. Transferable fingerprints would reduce cold-start effort for new services.

Second, counterfactual evaluation should be developed. Because autoscaling actions are interventions, it is valuable to estimate what would have happened if the controller had chosen a different action. Counterfactual analysis can improve policy learning and support post-incident review.

Third, integration with reinforcement learning should be explored carefully. Reinforcement learning can optimize long-term resource governance, but unsafe exploration is unacceptable in production. Causal fingerprints may provide a safer state representation for constrained reinforcement learning.

Fourth, service mesh policy optimization should be jointly considered with autoscaling. Retry budgets, timeout policies, circuit breakers, and traffic splits strongly affect resource demand. Future controllers should optimize these policies together with replica and resource allocation.

Fifth, causal fingerprinting should be extended to multi-cluster and edge-cloud environments. Many modern systems deploy services across regions, clusters, and edge nodes. Root-cause-aware governance in these environments must account for network latency, data locality, failover, and regional capacity constraints.

Sixth, benchmark standardization is needed. Existing microservice benchmarks are useful, but causal autoscaling requires benchmark scenarios that include labeled root causes, release events, service mesh policies, node contention, and workload bursts. Such benchmarks would allow fair comparison among root-cause-aware autoscalers.

11. Conclusion

This paper proposed Runtime Causal Fingerprinting for Root-Cause-Aware Autoscaling and Adaptive Resource Governance in Kubernetes Service Meshes. The central argument is that autoscaling should not be driven solely by local metric thresholds or even by predictive resource demand. In complex service mesh environments, performance symptoms often propagate across dependencies, and scaling the symptomatic service may fail to address the true cause. RCF-AG introduces causal fingerprints as compact, action-oriented representations of runtime degradation. These fingerprints encode probable root component, propagation path, resource dimension, temporal morphology, confidence, SLO risk, and recommended governance action.

The proposed framework integrates multi-signal telemetry, dynamic causal graphs, anomaly segmentation, fingerprint generation, policy-constrained action selection, and feedback learning. It distinguishes demand-driven saturation from downstream bottlenecks, retry amplification, release regression, and node contention. By doing so, it expands autoscaling into adaptive resource governance, where horizontal scaling is one possible intervention among several. The framework is designed to improve SLO protection, reduce false-positive scaling, enhance explainability, and support safer cloud-native operations.

The broader contribution is conceptual and methodological. Kubernetes service meshes provide the telemetry required for more intelligent autoscaling, but telemetry must be transformed into causal knowledge before it can safely guide interventions. Runtime causal fingerprinting offers a path toward this transformation. As microservice systems continue to grow in complexity, root-cause-aware autoscaling will become increasingly important for balancing reliability, cost, and operational trust.

References

1. M. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014, doi: 10.1007/s10723-014-9314-7.
2. S. K. Gunda, "A Deep Dive into Software Fault Prediction: Evaluating CNN and RNN Models," in *2024 International Conference on Electronic Systems and Intelligent Computing (ICESIC)*, Chennai, India, 2024, pp. 224–228, doi: 10.1109/ICESIC61777.2024.10846549.
3. H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020, pp. 805–825.
4. P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018, doi: 10.1109/MS.2018.2141039.
5. Spatharakis, D., Dimolitsas, I., Vlahakis, E., Dechouniotis, D., Athanasopoulos, N., & Papavassiliou, S. (2022). Distributed resource autoscaling in Kubernetes edge clusters. In Proceedings of the 18th International Conference on Network and Service Management (CNSM 2022) (pp. 163–169). IFIP. https://doi.org/10.23919/CNSM55787.2022.996505
6. G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "MicroRank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *Proceedings of The Web Conference 2021 (WWW '21)*, Ljubljana, Slovenia, 2021, pp. 3087–3098, doi: 10.1145/3442381.3449905.
7. A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*, Bordeaux, France, 2015, pp. 1–17, doi: 10.1145/2741948.2741964.
8. S. K. Gunda, "Machine Learning Approaches for Software Fault Diagnosis: Evaluating Decision Tree and KNN Models," in *2024 Global Conference on Communications and Information Technologies (GCCIT)*, Bangalore, India, 2024, pp. 1–5, doi: 10.1109/GCCIT63234.2024.10861953.
9. C. Meng, S. Song, H. Tong, M. Pan, and Y. Yu, "DeepScaler: Holistic autoscaling for microservices based on spatiotemporal GNN with adaptive graph learning," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*, Luxembourg, 2023, pp. 569–580, doi: 10.1109/ASE56229.2023.00038.
10. J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing: 16th International Conference, ICSOC 2018*, Hangzhou, China, 2018, pp. 3–20, doi: 10.1007/978-3-030-03596-9_1.
11. J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro)service-based cloud applications: A survey," *ACM Computing Surveys*, vol. 55, no. 3, Article 59, pp. 1–39, 2022, doi: 10.1145/3501297.
12. S. K. Gunda, "Software Defect Prediction Using Advanced Ensemble Techniques: A Focus on Boosting and Voting Method," in *2024 International Conference on Electronic Systems and Intelligent Computing (ICESIC)*, Chennai, India, 2024, pp. 157–161, doi: 10.1109/ICESIC61777.2024.10846550.
13. Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, Providence, RI, USA, 2019, pp. 3–18, doi: 10.1145/3297858.3304013.
14. M. Li, Z. Li, K. Yin, X. Nie, W. Zhang, K. Sui, and D. Pei, "Causal inference-based root cause analysis for online service systems with intervention recognition," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22)*, Washington, DC, USA, 2022, pp. 3230–3240, doi: 10.1145/3534678.3539041.
15. C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys*, vol. 51, no. 4, Article 73, pp. 1–33, 2018, doi: 10.1145/3148149.
16. Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and scalable ML-driven performance debugging in microservices," in *Proceedings of the 26th*

ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), Virtual Event, 2021, pp. 135–151, doi: 10.1145/3445814.3446700.

17. P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "CloudRanger: Root cause identification for cloud native systems," in *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '18)*, Washington, DC, USA, 2018, pp. 492–502, doi: 10.1109/CCGRID.2018.00076.
18. J. Park, B. Choi, C. Lee, and D. Han, "Graph neural network-based SLO-aware proactive resource autoscaling framework for microservices," *IEEE/ACM Transactions on Networking*, vol. 32, no. 4, pp. 3331–3346, 2024, doi: 10.1109/TNET.2024.3393427.
19. L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root cause localization of performance issues in microservices," in *NOMS 2020—2020 IEEE/IFIP Network Operations and Management Symposium*, Budapest, Hungary, 2020, pp. 1–9, doi: 10.1109/NOMS47738.2020.9110353.
20. H. Ahmad, C. Treude, M. Wagner, and C. Szabo, "Smart HPA: A resource-efficient horizontal pod auto-scaler for microservice architectures," in *Proceedings of the IEEE 21st International Conference on Software Architecture (ICSA 2024)*, Hyderabad, India, 2024, pp. 46–57, doi: 10.1109/ICSA59870.2024.00013.