Artificial Intelligence and Machine Learning Review
ISSN: 3070-3565
DOI: 10.69987/AIMLR.2026.70208

Confidence-Gated Arbitration of Automated Control Actions for Stability under Compounded Failures in High-Concurrency Platforms: A Comparative Empirical Study

Linhao Hu[1], Chenhui Hao[1.2]
[1] *Master of Science in Computer Science, Washington University in St. Louis, MO, USA*
[1.2] *Civil Engineering, University of California, CA, USA*

Keywords

automation paradox, tail latency, retry amplification, observability-driven governance

Abstract

High-concurrency platforms rely on automated control actions such as auto-scaling, request retries, circuit breaking, and failover to absorb load and recover from disturbances. Under compounded failures these same actions frequently amplify rather than dampen instability, producing retry surges, oscillating scaling decisions, and recovery trajectories that stall in degraded states. This study presents a comparative empirical evaluation of three operating configurations on a microservice benchmark driven by production-derived workload characteristics: naive automation, queue-based smoothing only, and a full governance arrangement that adds observability-driven confidence gating and rate constraints over the automated actions. Three fault classes are injected under controlled load: dependency timeout, node failure, and traffic spike. Across ten repetitions per configuration-fault pair, the governance arrangement lowers tail latency and recovery time while suppressing retry amplification and scaling oscillation, with the magnitude of the gain depending strongly on the fault class. Queue smoothing alone recovers most of the latency benefit under traffic spikes, while confidence gating contributes the larger share under timeout and node-failure conditions where self-reinforcing retry behavior dominates. The findings delineate where arbitration of automated actions pays off and where buffering suffices, offering an engineering-grade basis for stabilizing large-scale digital infrastructure.

1. Introduction

High-concurrency platforms in finance, commerce, and public services depend on a layered stack of automated control actions to remain available under fluctuating demand. Auto-scaling adjusts capacity, retries paper over transient errors, circuit breakers shed pressure from struggling dependencies, and failover redirects traffic away from unhealthy replicas. Each mechanism is individually sound and designed to suppress a specific class of disturbance. The difficulty arises when several disturbances coincide and the automated responses interact. A localized timeout triggers retries, the retries inflate downstream load, the load lengthens queues, the queues trigger more timeouts, and the loop sustains itself well after the original trigger has passed.

This pattern is a contemporary expression of a long-standing observation about automated control. Bainbridge [1] noted that automating most but not all of a task leaves operators responsible for intervening exactly when the automation reaches the edge of its competence and the operator is least prepared to act. Distributed systems show a similar irony: the more aggressively automated actions defend steady-state throughput, the more violently they can destabilize the system at the boundary of their assumptions. A recent survey of production outages across many organizations documents this as metastable failure, in which a system trapped by a self-sustaining feedback loop stays in a low-throughput state even after the initiating condition is removed, with retry amplification and load-driven feedback among the most common sustaining mechanisms [2].

The operational vocabulary for these dynamics is well established. Cascading failure describes the progressive collapse of capacity as overload propagates across dependent services under positive feedback [3]. Chaos engineering offers a disciplined way to surface such behavior before it reaches users, by injecting representative faults and observing whether steady-state availability holds [4]. What remains underexamined is arbitration: when multiple automated actions are simultaneously eligible to fire during a compounded failure, which should be permitted, which throttled, and on what evidence.

1.1. Background and Motivation

The motivating gap is the absence of coordinating discipline over automated actions during the moments those actions are most likely to conflict. Buffering smooths input but does not decide whether a retry storm should proceed, and monitoring exposes symptoms without constraining the controllers that produce them. The practical need is a lightweight arbitration discipline that reads live telemetry, estimates whether a candidate action will dampen or amplify the current disturbance, gates it accordingly, and escalates to operators when anomalies persist beyond a bounded interval.

1.2. Problem Statement and Contributions

This study frames the question as a controlled comparison rather than the proposal of a new architecture. The central problem is to quantify how much stability is attributable to buffering, how much to arbitration of automated actions, and how that attribution shifts across fault classes.

A. Research Questions

The work asks whether confidence-gated arbitration of automated actions improves tail latency, recovery time, retry amplification, and scaling oscillation relative to naive automation and to queue-based smoothing alone, and whether those improvements depend on the injected fault class. The comparison isolates the contribution of each layer so that the source of any gain is attributable rather than conflated.

B. Scope and Contributions

The contributions are threefold. The study designs a reproducible comparative protocol over three configurations and three fault classes on a benchmark driven by production-derived workload characteristics. It specifies an observability-driven gating rule that decides actions from live latency, error, and backlog signals rather than static thresholds. It reports an attribution analysis separating buffering from arbitration and bounds the approach to asynchronous and near-real-time paths while noting the queuing-latency cost on latency-critical lookups.

2. Related Work

Prior work spans the mechanisms that produce instability and the mechanisms that observe and contain it. The relevant strands concern failure amplification, overload control, distributed tracing, and capacity management, each of which informs one component of the comparison reported here.

2.1. Failure Amplification in Automated Control

Two amplification pathways recur in the literature: retry-driven feedback and capacity-driven oscillation. Both convert a bounded trigger into an unbounded disturbance through automated responses that were intended to help.

A. Retry Amplification

Retry policy is the most studied amplifier. Guidance on timeouts and backoff with jitter documents how naive retries multiply downstream load and how capped exponential backoff combined with randomized jitter limits the synchronization of retry waves [5]. Retry budgets and circuit breakers were introduced to cap this multiplication, yet they act on per-client or per-dependency state and remain blind to the aggregate load that many independent actions collectively impose on a shared dependency. The lesson carried into this study is that retry should be treated as an action arbitrated under live evidence rather than a fixed policy executed unconditionally, since the same policy that smooths transient errors can drive a sustaining loop once a dependency is already saturated. A coordinating view across actions, rather than local rate limiting in isolation, is what separates arbitration from the per-call defenses already in common use.

B. Scaling Oscillation

Capacity controllers contribute a second amplification channel. Production autoscaling must balance responsiveness against stability, since reactive scaling under noisy signals can flap between scale-up and scale-down decisions and prolong recovery [6]. Stabilizing measures such as cooldown intervals and hysteresis temper this flapping, although they delay legitimate scaling as much as spurious scaling and trade one failure mode for another. The arbitration question applies symmetrically to scaling: a scale action taken on a transient signal during a compounded failure can deepen the disturbance it was meant to relieve, which motivates gating scale decisions on a confidence estimate that reflects the trend in load rather than an instantaneous threshold reading.

2.2. Observability-Driven Overload and Load Management

A parallel line of work supplies the signals and controls that arbitration depends on. Overload control for large microservice deployments demonstrates that admission decisions keyed to queuing time can protect goodput under sustained pressure [7], and decentralized coordination of such decisions across a call graph can reduce tail latency at high utilization [8]. Low-overhead distributed tracing established that request-scoped telemetry can be collected continuously through sampling and lightweight instrumentation [9], and automated metric analysis can distill a high-dimensional monitoring stream into the few signals that matter for control [10]. Load placement by estimated latency and requests-in-flight, rather than by raw utilization, further shows the value of acting on responsiveness signals close to the decision point [11]. Taken together, these results establish that the signals needed for arbitration are already collectible at low overhead, and that acting on responsiveness rather than utilization yields better decisions under pressure. What remains missing is a discipline that converts those signals into action-level decisions during compounded failures. The configurations compared in this study draw on these signals while differing in whether they arbitrate automated actions or merely observe them.

3. Methodology and Experimental Design

The evaluation is built around a single benchmark application exercised under a production-derived workload, with three operating configurations and three fault classes. The design holds the application, the load shape, and the fault parameters fixed across configurations so that observed differences are attributable to the control discipline rather than to environmental variation.

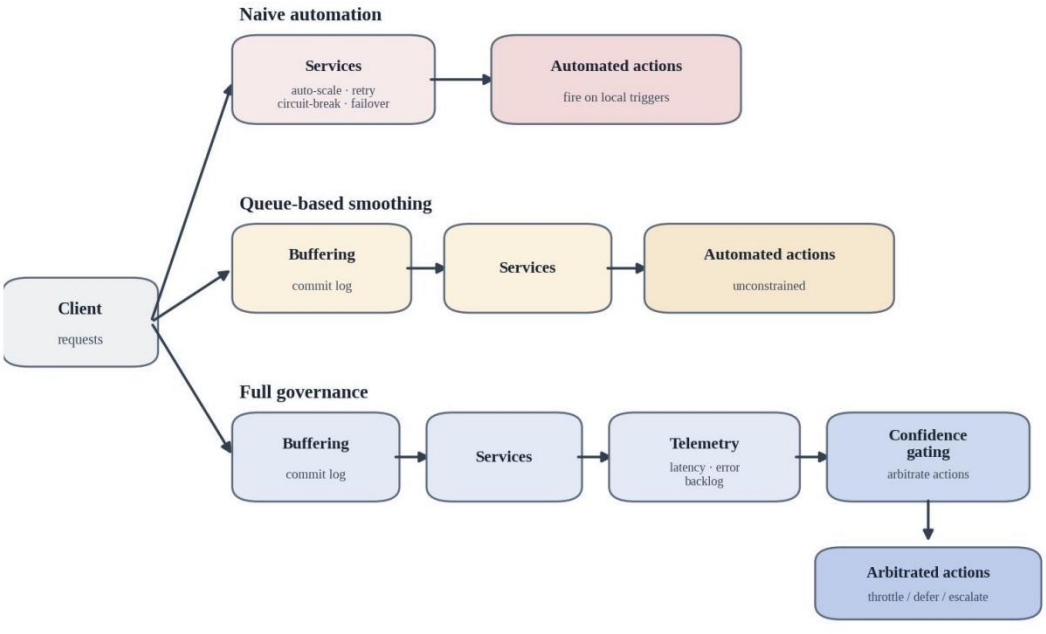


Figure 1. Experimental Configurations and the Closed-Loop Arbitration Path

3.1. System Boundary and Buffering Layer

The mechanism studied here targets asynchronous and near-real-time paths, where a small amount of buffering is acceptable in exchange for smoother downstream load. A partitioned commit log serves as the decoupling and buffering layer, drawing on the durable, pull-based design that lets consumers absorb bursts at their own pace [12] and on the replicated logging practices that make such a layer dependable in production pipelines [13]. The boundary is explicit: latency-critical point lookups, such as checkout authorization or core authentication, are not routed through the log, since full enqueueing would add queuing latency that such paths cannot tolerate. For those paths the arbitration discipline operates over a bounded in-memory ring buffer and a local cache rather than the durable log, gating actions without persistent-queue delay and keeping the buffering benefit where it is affordable.

Figure 1 situates the three configurations within this boundary, showing the shared request path, the buffering layer, the telemetry signal flow, and the point at which automated actions are arbitrated.

Schematic of the three evaluated configurations over a common request path. In the naive configuration, automated actions for retry, scaling, and failover fire directly on local triggers. The queue-only configuration inserts a buffering layer that decouples producers from consumers and smooths bursts, leaving the automated actions otherwise unconstrained. The full governance configuration adds a telemetry-fed gating stage that reads latency, error rate, and backlog signals and arbitrates each candidate action before it executes, escalating to manual intervention when anomalies persist beyond the bounded window. The diagram highlights that the buffering layer and the gating stage act at different points: buffering shapes input flow, while gating constrains the controllers that respond to it.

3.2. Comparative Configurations and Confidence Gating

The three configurations form a cumulative progression, with each adding one layer to the previous so that the marginal contribution of buffering and of arbitration can be separated.

A. Three Configurations

The naive automation configuration represents common practice: auto-scaling, retry, circuit breaking, and failover each fire on their own local triggers with default policies and no coordination among them. The queue-based smoothing configuration adds the buffering layer described above, smoothing input bursts and applying backpressure, while leaving the automated actions unconstrained once requests are admitted. The full governance configuration retains the buffering layer and adds the gating stage, which arbitrates every candidate automated action against live telemetry before it executes. The progression is deliberately additive so that any difference between the second and third configurations isolates the effect of arbitration alone.

C. Confidence-Gated Arbitration Logic

The gating stage estimates, from a sliding telemetry window, whether a candidate action is likely to dampen or amplify the current disturbance. The estimate combines the recent slope of tail latency, the error rate, and the backlog depth into a confidence score for the action. When the score indicates that a retry or scale action would compound an active amplification, the action is throttled or deferred rather than executed, and the rate of admitted retries is bounded to prevent synchronized waves. When anomalies persist beyond a bounded interval despite gating, the stage stops attempting automated recovery and escalates to operators, preserving the residual-intervention role that automated systems must leave open.

3.3. Workload, Fault Injection, and Metrics

The workload and fault parameters are drawn from public sources and held constant across configurations, with all reported figures produced by the load tests described here rather than read from any external dataset.

D. Workload Traces and Benchmark Systems

The benchmark application is the Hotel Reservation service from an open microservice suite whose tail-at-scale characteristics are documented in its originating study [14]. Request mix and call-graph fan-out are shaped to match characteristics of a public production microservice trace, so that the synthetic load reflects realistic dependency structure rather than uniform traffic. A constant-throughput load generator drives the system and records latency with high-dynamic-range histograms to avoid coordinated-omission bias. Table 1 lists the workload and configuration parameters held fixed across all runs.

Table 1. Benchmark and workload configuration.

Parameter	Value
Benchmark application	Hotel Reservation microservice suite (gRPC)
Workload shaping source	public production microservice trace (call-graph fan-out and request mix)
Load generator	constant-throughput generator with high-dynamic-range latency recording
Nominal request rate	9,500 req/s
Peak (spike) request rate	19,000 req/s
Baseline replicas per service	6
Telemetry scrape interval	1 s
Confidence-gating window	10 s sliding
Repetitions per configuration-fault pair	10 runs

E. Fault Injection and Evaluation Metrics

Faults are injected with a cloud-native fault-injection controller, and the injection design parallels the controlled performance-anomaly approach used for resource-management evaluation on this class of benchmark [15]. Three fault classes are applied in isolation: a dependency timeout that adds latency and aborts a fraction of a specific call path, a node failure that terminates a subset of data-tier pods for a fixed outage window, and a traffic spike that steps the request rate upward within a few seconds. Table 2 records the injection mechanisms and parameters. Five metrics are evaluated: 99th- and 99.9th-percentile tail latency, recovery time measured as mean time to recovery, retry amplification factor defined as the ratio of total attempted requests to original requests during the disturbance, scaling oscillation measured as the count of scale actions and the peak replica swing in the recovery window, and sustained throughput.

Table 2. Fault injection matrix.

Fault	Injection mechanism	Parameters	Target scope
Dependency timeout (F1)	added latency with partial abort	+800 ms latency, 12% abort	reservation-to-profile call path
Node failure (F2)	pod termination	2 of 6 data-tier pods, 90 s outage	random data-tier replicas
Traffic spike (F3)	load step	9,500 to 19,000 req/s over 5 s	ingress

4. Results and Analysis

Results are reported as means over ten repetitions per configuration-fault pair. The analysis emphasizes where each layer contributes, since the attribution between buffering and arbitration shifts markedly across the three fault classes. Reported reductions are measured against the naive configuration unless stated otherwise, and the framing remains deliberately conservative given the bounded scale of the evaluation.

4.1. Tail Latency and Throughput under Faults

Tail latency and sustained throughput respond differently to buffering and to arbitration depending on whether the dominant amplifier is retry feedback or input burstiness. Table 3 reports the full set of latency, recovery, amplification, and throughput figures, and Figure 2 visualizes the tail-latency comparison across the three fault classes.

Table 3. Tail latency, recovery, retry amplification, and throughput (mean of 10 runs).

Fault	Configuration	p99 (ms)	p999 (ms)	MTTR (s)	RAF	Throughput (req/s)
F1 Dependency timeout	Naive	842	2,150	148	3.7	6,200
F1 Dependency timeout	Queue-only	610	1,480	96	2.4	7,400
F1 Dependency timeout	Full governance	470	980	58	1.5	8,350
F2 Node failure	Naive	910	2,480	205	2.9	5,800
F2 Node failure	Queue-only	720	1,760	150	2.1	6,900
F2 Node failure	Full governance	560	1,180	92	1.4	7,950
F3 Traffic spike	Naive	1,180	3,050	178	4.2	5,400
F3 Traffic spike	Queue-only	680	1,520	88	2.0	8,100
F3 Traffic spike	Full governance	520	1,050	61	1.6	8,600

A. Dependency Timeout and Node Failure

Under the dependency timeout, where retry feedback is the principal amplifier, arbitration carries most of the benefit. Queue smoothing lowers p99 latency from 842 ms to 610 ms, a reduction of about 28 percent, while full governance brings it to 470 ms, about 44 percent against naive automation. The step from queue-only to full governance is comparable in size to the step that buffering provides, because gating bounds the retry waves that buffering admits but does not constrain. Node failure shows the same ordering with a smaller spread: full governance reduces p99 latency from 910 ms to 560 ms, about 38 percent, and throughput recovers from 5,800 req/s to 7,950 req/s. The gain here is more moderate, since a node outage removes capacity that no control discipline can restore until replacement pods are ready.

B. Traffic Spike

Under the traffic spike the attribution inverts. Buffering alone captures most of the latency benefit, lowering p99 latency from 1,180 ms to 680 ms, a reduction of about 42 percent, because absorbing the input burst addresses the dominant cause directly. Full governance adds a moderate further improvement to 520 ms, about 56 percent against naive automation, by preventing the scaling controller from overreacting to the transient. Throughput under the spike rises from 5,400 req/s to 8,100 req/s with buffering and to 8,600 req/s with full governance, indicating that the marginal value of arbitration over buffering is real yet limited when input shaping already resolves the primary disturbance.

4.2. Recovery Time and Retry Amplification

Recovery time and retry amplification track each other closely, consistent with the metastable-failure account in which self-sustaining retry loops keep a system degraded after the trigger passes [16]. The retry amplification factor falls from 3.7 to 1.5 under timeout, from 2.9 to 1.4 under node failure, and from 4.2 to 1.6 under traffic spike when moving from naive automation to full governance. Recovery time follows the same pattern, with the largest absolute reduction under node failure, from 205 s to 92 s, a reduction of about 55 percent. Figure 3 presents the recovery-time and amplification comparison side by side. The correspondence supports the interpretation that bounding the rate of admitted retries, rather than buffering input, is what shortens recovery once an amplification loop has formed.

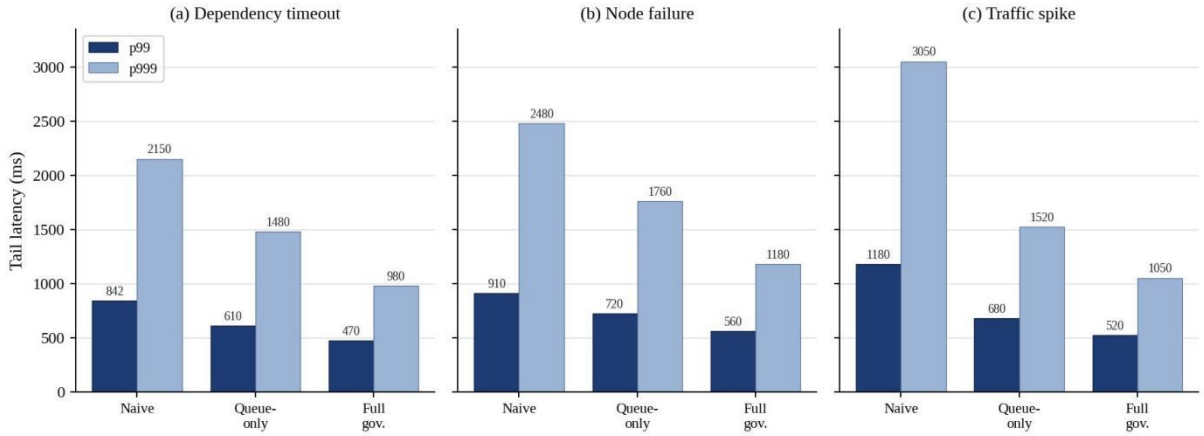


Figure 2. Tail Latency Across Three Configurations and Three Fault Classes

Comparison of 99th- and 99.9th-percentile latency for the three configurations under (a) dependency timeout, (b) node failure, and (c) traffic spike. Across all three faults the ordering is consistent, with naive automation highest and full governance lowest. The largest p99 reduction occurs under the traffic spike, from 1,180 ms to 520 ms, while the largest contribution from arbitration relative to buffering occurs under the dependency timeout, where p99 falls from 610 ms to 470 ms after gating is added. The 99.9th-percentile gap is widest under naive automation, reaching 3,050 ms under the spike, and narrows to roughly 1,000 ms under full governance across all faults.

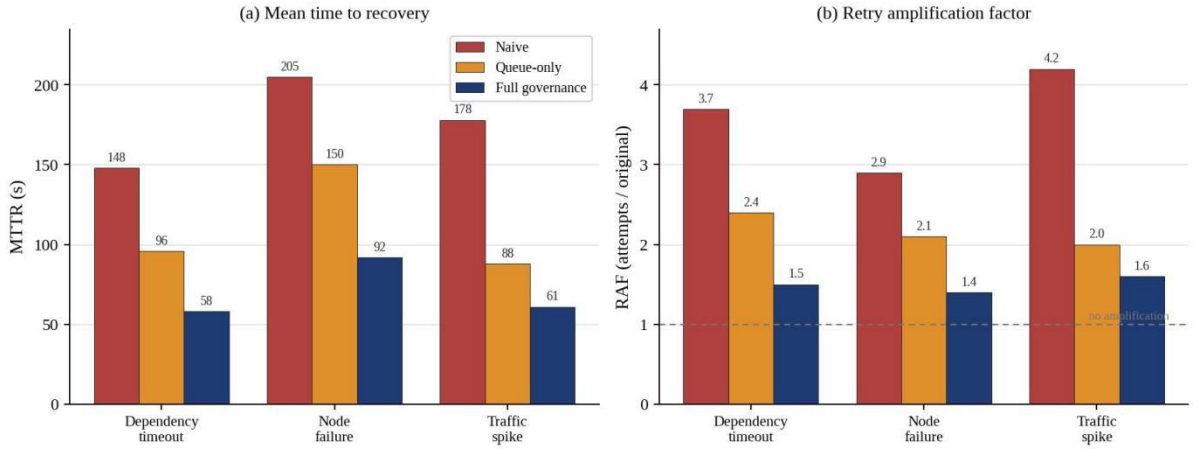


Figure 3. Recovery Time and Retry Amplification by Fault Class

Side-by-side comparison of (a) mean time to recovery and (b) retry amplification factor for the three configurations across the three fault classes. The two panels move together, with the retry amplification factor falling to between 1.4 and 1.6 under full governance across all faults. The largest recovery-time reduction appears under node failure, from 205 s to 92 s, and the largest amplification reduction appears under the traffic spike, from 4.2 to 1.6. The joint movement indicates that suppression of retry amplification is the proximate cause of shortened recovery.

4.3. Scaling Oscillation and Component Ablation

Scaling behavior and the decomposition of the gating contribution complete the picture, clarifying which signal the arbitration rule depends on most.

C. Scaling Oscillation Amplitude

Scaling oscillation appears only under node failure and traffic spike, the two faults that engage the capacity controller. Table 4 reports the count of scale actions and the peak replica swing during the recovery window. Under the traffic spike, naive automation issues 17 scale actions with a peak swing of 11 replicas, queue smoothing reduces this to 8 actions and 5 replicas, and full governance to 3 actions and 2 replicas. The pattern is consistent with production observations that

capacity controllers churn under shifting pressure when scale decisions react to instantaneous load [17], and it shows that gating the scale action on a confidence estimate, rather than on an instantaneous threshold, is what damps the oscillation.

Table 4. Scaling oscillation during recovery (mean of 10 runs).

Fault	Configuration	Scale actions (per 5 min)	Peak replica swing
F2 Node failure	Naive	14	9
F2 Node failure	Queue-only	9	6
F2 Node failure	Full governance	4	3
F3 Traffic spike	Naive	17	11
F3 Traffic spike	Queue-only	8	5
F3 Traffic spike	Full governance	3	2

D. Ablation of Gating Components

Disabling individual signals within the gating rule isolates their contribution. Removing the backlog signal while retaining latency and error signals raises p99 latency under the traffic spike from 520 ms to 605 ms and increases scale actions from 3 to 6, indicating that backlog depth is the dominant input for damping scaling oscillation. Removing the latency-slope signal while retaining the others raises the retry amplification factor under timeout from 1.5 to 2.2, since the rate of admitted retries is no longer bounded by the trend in tail latency. The ablation confirms that the two amplification channels are governed by different signals, both of which the full governance configuration requires.

5. Discussion and Future Work

The comparison yields a practical decision rule for where arbitration is worth its complexity, alongside an honest account of the conditions under which the present evidence holds.

5.1. Discussion, Boundary Conditions, and Threats to Validity

The attribution analysis supports a layered recommendation. Where input burstiness is the dominant disturbance, buffering alone captures most of the available stability, and arbitration adds a moderate increment that may not justify its operational cost. Where retry feedback or scaling oscillation dominates, as under dependency timeout and node failure, arbitration of automated actions contributes the larger share and is the component that shortens recovery. The boundary condition stated in the methodology is material to this conclusion. The buffering layer is appropriate for asynchronous and near-real-time paths, while latency-critical point lookups must rely on bounded in-memory buffering and local caching to avoid the queuing latency that a durable log would impose. Several threats to validity bound these claims. The evaluation runs on a single benchmark application, so the absolute figures should be read as indicative rather than universal. The workload is shaped from public trace characteristics rather than replayed verbatim, leaving open whether verbatim replay would shift the margins. The faults are injected one class at a time, so genuinely simultaneous compounded failures may interact in ways this protocol does not capture. The reported values are load-test output and carry the external-validity limits inherent to any simulation of production behavior.

5.2. Conclusion and Future Work

This study compared naive automation, queue-based smoothing, and a full governance arrangement that arbitrates automated control actions through observability-driven confidence gating, under three injected fault classes on a microservice benchmark driven by production-derived workload characteristics. The governance arrangement lowered tail latency and recovery time and suppressed retry amplification and scaling oscillation, with buffering carrying most of the benefit under traffic spikes and arbitration carrying most of it under timeout and node-failure conditions. The decomposition gives operators a basis for deciding which layer to invest in for a given failure profile rather than adopting all mechanisms uniformly, and it suggests that the operational cost of arbitration is best spent where retry and scaling

feedback dominate. Future work will extend the protocol to simultaneously injected faults, replay verbatim production traces across multiple benchmark applications to test the stability of the attribution, and study adaptive thresholds that let the gating window adjust to changing load without manual retuning.

References

[1]. Bainbridge, L. (1983). Ironies of automation. Automatica, 19(6), 775-779. https://doi.org/10.1016/0005-1098(83)90046-8

[2]. Huang, L., Magnusson, M., Muralikrishna, A. B., Estyak, S., Isaacs, R., Aghayev, A., Zhu, T., & Charapko, A. (2022). Metastable failures in the wild. In Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22) (pp. 73-90). USENIX Association.

[3]. Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). Site reliability engineering: How Google runs production systems. O'Reilly Media.

[4]. Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., & Rosenthal, C. (2016). Chaos engineering. IEEE Software, 33(3), 35-41. https://doi.org/10.1109/MS.2016.60

[5]. Brooker, M. (n.d.). Timeouts, retries, and backoff with jitter. Amazon Builders' Library. https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/

[6]. Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., & Wilkes, J. (2020). Autopilot: Workload autoscaling at Google. In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20). Association for Computing Machinery. https://doi.org/10.1145/3342195.3387524

[7]. Zhou, H., Chen, M., Lin, Q., Wang, Y., She, X., Liu, S., Gu, R., Ooi, B. C., & Yang, J. (2018). Overload control for scaling WeChat microservices. In Proceedings of the ACM Symposium on Cloud Computing (SoCC '18) (pp. 149-161). Association for Computing Machinery. https://doi.org/10.1145/3267809.3267823

[8]. Xing, J., Giannoukos, A., Loh, P., Wang, S., Qiu, J., Demoulin, H. M., Kallas, K., & Lee, B. C. (2025). Rajomon: Decentralized and coordinated overload control for latency-sensitive microservices. In Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25) (pp. 21-36). USENIX Association.

[9]. Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure (Technical Report). Google, Inc.

[10]. Thalheim, J., Rodrigues, A., Akkus, I. E., Bhatotia, P., Chen, R., Viswanath, B., Jiao, L., & Fetzer, C. (2017). Sieve: Actionable insights from monitored metrics in distributed systems. In Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17) (pp. 14-27). Association for Computing Machinery. https://doi.org/10.1145/3135974.3135977

[11]. Wydrowski, B., Kleinberg, R., Rumble, S. M., & Stolyar, A. (2024). Load is not what you should balance: Introducing Prequal. In Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24). USENIX Association.

[12]. Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka: A distributed messaging system for log processing. In Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB '11).

[13]. Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., Rao, J., Kreps, J., & Stein, J. (2015). Building a replicated logging system with Apache Kafka. Proceedings of the VLDB Endowment, 8(12), 1654-1665. https://doi.org/10.14778/2824032.2824063

[14]. Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., ... Delimitrou, C. (2019). An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19) (pp. 3-18). Association for Computing Machinery. https://doi.org/10.1145/3297858.3304013

[15]. Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., & Iyer, R. K. (2020). 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) (pp. 805-825). USENIX Association.

[16]. Bronson, N., Aghayev, A., Charapko, A., & Zhu, T. (2021). Metastable failures in distributed systems. In Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21) (pp. 221-227). Association for Computing Machinery. https://doi.org/10.1145/3458336.3465286

[17]. Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., Harchol-Balter, M., & Wilkes, J. (2020). Borg: The next generation. In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20). Association for Computing Machinery. https://doi.org/10.1145/3342195.3387517