SinkrOn
Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

Evaluating Kubernetes Progressive Delivery in Constrained Environments: Flagger vs. Argo Rollouts

Gagah Syuja Saka Abdullah[1)], **Rama Aria Megantara**[2)*]
[1,2)]Teknik Informatika, Fakultas Ilmu Komputer, Universitas Dian Nuswantoro
[1)]111202214259@mhs.dinus.ac.id, [2)]aria@dsn.dinus.ac.id

Submitted :May 11, 2026 | **Accepted** : July 7, 2026 | **Published** : July 9, 2026

Abstract: Cloud-native progressive delivery orchestrators reduce deployment risk by automating canary deployment rollback procedures, dramatically reducing mean time to recover from deployment failures. However, existing research predominantly evaluates these tools in hyperscale environments, masking the transient computational overhead they introduce in resource-constrained edge deployments. This study empirically evaluates and compares the automated incident mitigation latency and computational resource volatility of Flagger and Argo Rollouts within a strictly resource-constrained Kubernetes environment. A low virtual central processing unit Kubernetes testbed was provisioned using Talos Linux with strict hypervisor-level central processing unit pinning, simulating edge computing conditions. Deterministic fault injection spanning four fault classes, two workload runtimes, and two network topology configurations was executed across thirty trials. A Shapiro-Wilk normality assessment, Welch t-test, Mann-Whitney U test, Cohen's d, and 95% confidence intervals were applied to compare temporal and computational metrics. Memory utilization remained statically bounded, averaging 24.01 megabytes for Flagger and 35.47 megabytes for Argo Rollouts. Under standard fault conditions, neither orchestrator demonstrated a consistent temporal advantage. However, under memory exhaustion progressing to CrashLoopBackOff, Argo Rollouts recovered in a mean of 29.67 seconds against Flagger's 166.79 seconds, a statistically significant 5.6-fold degradation with a large effect size. Argo Rollouts sustained transient central processing unit surges of 159 to 168 millicpu against Flagger's bounded ceiling of 17 to 18 millicpu. Progressive delivery automation introduces non-negligible and fault-type-dependent computational overhead in resource-constrained environments. Flagger is recommended for strict resource predictability in threshold-breach environments, while Argo Rollouts is recommended where broader fault-type resilience is operationally critical.

Keywords: automated incident mitigation; canary deployment; edge computing; Flagger; Kubernetes; progressive delivery; resource-constrained environment

INTRODUCTION

Cloud-native architecture is fundamentally oriented toward enabling rapid, continuous software delivery through DevOps and CI/CD practices, with operational agility as a core goal (Amaro, Pereira, & Silva, 2025; Deng et al., 2024). To mitigate the risk of catastrophic system outages during high-velocity updates, the software engineering industry has widely adopted Progressive Delivery. Progressive Delivery extends Continuous Delivery by gating each deployment increment on real-time observability signals, enabling risk-controlled releases that can be automatically halted or rolled back without human intervention. By utilizing Canary Deployment strategies, Progressive Delivery automates deployment safety by incrementally shifting user traffic while monitoring real-time health metrics, and automatically halting the rollout if anomalies are detected, and reducing Time to Recover (TTR) from hours to mere seconds (Malhotra, Elsayed, Torres, & Venkatraman, 2024). A countervailing operational priority has emerged: extreme infrastructure efficiency. As organizations increasingly deploy Kubernetes across resource-constrained edge environments, the co-scheduling of heterogeneous workload types with competing latency and throughput requirements creates inherent resource contention that demands careful orchestration (Shen et al., 2024). In such constrained settings, the additional computational overhead introduced

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

by deployment automation tooling cannot be treated as negligible, as it directly competes with critical application workloads for the same limited CPU and memory budget.

Systems research extensively validates the functional benefits of automated fault recovery and dynamic monitoring mechanisms within container orchestration platforms (Zhao, He, Luo, Li, & Wu, 2025). The industry practically implements these theoretical paradigms through progressive delivery orchestrators such as Flagger and Argo Rollouts, which represent architecturally distinct approaches to automated rollout management that form the central comparative subject of this study. However, the existing literature exhibits three profound gaps.

First, resource prediction models already exist for Kubernetes container systems at the infrastructure level (Turin et al., 2023), and functional evaluations of canary tools focus on deployment decision quality rather than operator-level resource overhead (Malhotra, Elsayed, Torres, & Venkatraman, 2024). Yet no comparative research has evaluated the CPU and memory consumption of progressive delivery controllers themselves under active fault injection, leaving tool adoption driven more by vendor ecosystem alignment, such as the choice between Flux and Argo CD, rather than by peer-reviewed telemetry (Ghosh, Mavromatis, Antonakoglou, & Katsaros, 2025).

Second, these orchestrators are predominantly evaluated in hyperscale public cloud environments that assume operator overhead is negligible (Aqasizade, Ataie, & Bastam, 2025; Sathish, Avula, B U, & Bhat, 2025). Edge-focused research, however, shows controller overhead accumulates significantly under constrained resources (Sebrechts et al., 2022).

Finally, existing methodologies rely on coarse-grained, long-term resource averaging (Liang, Govindan, & Park, 2025; Navarro, Garcia-Pineda, & Gutierrez-Aguado, 2024). This masks the transient, high-amplitude surges produced by the discrete reconciliation events through which these controllers periodically query Prometheus to evaluate canary health (Mondal, Zheng, & Cheng, 2024; Zhang, Guo, Tan, Guan, & Jiang, 2025). Consequently, no empirical benchmarks connect internal operator architectures directly to absolute Time-to-Recover (TTR), CPU volatility, and memory utilization in constrained clusters.

To address these gaps, this study measures the orchestration tax of progressive delivery on low-vCPU Kubernetes edge environments through a systematically expanded experimental design. Rather than comparing functional features, it quantifies the trade-offs between recovery speed, transient resource penalties, and fault-type-dependent resilience for Flagger and Argo Rollouts across four fault injection scenarios (including memory exhaustion progressing to CrashLoopBackOff), two workload runtimes (a lightweight Go binary and an interpreted Python service), and two network topologies (standard ingress and a Linkerd service mesh). Specifically, this study asks: (1) How do the controllers' distinct internal architectures affect absolute incident mitigation latency (Total Recovery Time) across fault scenarios, workloads, and mesh configurations? and (2) What is the true magnitude of their orchestration tax during active fault mitigation, as measured via high-resolution CPU and memory telemetry? By isolating a deterministic, low-vCPU edge environment, this study exposes the hidden computational costs and fault-type-specific sensitivities of deployment automation, giving platform engineers peer-reviewed data to balance reliability against strict infrastructure budgets at the edge.

METHODS

Experimental Framework and Workflow

To ensure experimental reproducibility and a structured comparison between the two orchestrators, the research follows a phased pipeline as depicted in Figure 1, designed to evaluate orchestrator behaviour under both nominal and adversarial conditions across four fault injection scenarios, two application workload runtimes, and two network topology configurations. In the Success Phase, the orchestrator performs a standard canary deployment to establish baseline metrics for CPU surge and memory stability. Conversely, the Failure Phase involves the injection of synthetic faults to trigger the orchestrator's automated recovery mechanisms. Throughout these cycles, high-resolution telemetry was continuously scraped via Prometheus and visualized in Grafana. Run logs capturing exact state transition timestamps were automatically generated by the execution script, with Grafana metric exports subsequently aligned to those timestamps to ensure temporal consistency between telemetry and event data. Individual trial records were then consolidated into per-scenario datasets using an automated Python summarisation script. Each scenario was executed for 36 raw trials per configuration, from which incomplete records and cold-start transients were excluded, yielding 30 clean trials per scenario. A minor capture window asymmetry between orchestrators is acknowledged as a methodological limitation and discussed in the limitations section. All deployment configurations, experimental scripts, and dataset artefacts are publicly available at https://github.com/gagahsyuja/progressive-delivery-research.

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

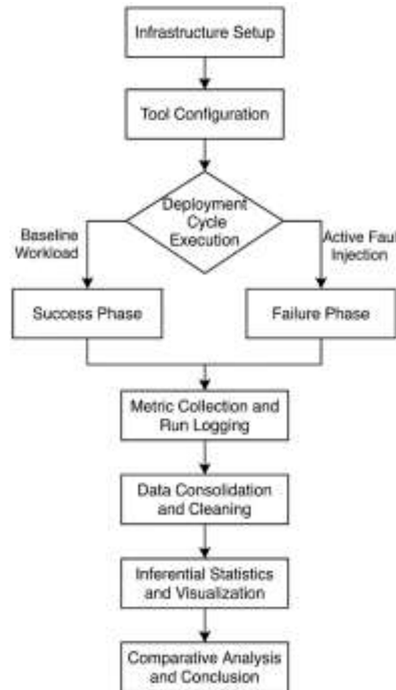

Fig. 1 Research methodology workflow

Experimental Testbed Configuration

To establish a reproducible baseline, the foundational hardware infrastructure and the specific software versions utilized to construct the experimental environment are detailed in Table 1.

Table 1. Infrastructure and Software Environment

Component	Specification
Hardware Platform	Lenovo ThinkPad T480
Physical CPU	Intel Core i5-8350U @ 2.4 GHz (up to 3.6 GHz Boost)
Physical Memory	24 GB DDR4
Physical Storage	256 GB SATA 3 Solid State Drive (SSD)
Kubernetes Version	v1.35.2
Container Network Interface (CNI)	Cilium
Container Runtime	containerd (via Talos Linux)

To replicate the resource constraints characteristic of edge computing deployments, the experimental environment was provisioned as a set of resource-limited virtual machines managed by Kernel-based Virtual Machine (KVM) and QEMU, with each node explicitly capped at 2 vCPUs and 4 GB RAM to reflect the computational boundaries of typical edge hardware. The cluster was bootstrapped using Talos Linux, an immutable, minimal, and API-driven container operating system purpose-built for securely hosting Kubernetes clusters, reducing the underlying OS footprint to only the binaries required by the Kubernetes runtime. The topology consisted of four distinct virtual machines: one control plane node (2 vCPUs, 4 GB RAM), two worker nodes (each 2 vCPUs, 4 GB RAM), and an isolated load-generation node running Alpine Linux (1 vCPU, 1 GB RAM) dedicated to the k6 testing framework, as detailed in Table 2.

To ensure deterministic performance and eliminate virtualization scheduler latency, strict CPU pinning was enforced across the hypervisor. Each virtual CPU (vCPU) was bound directly to a dedicated physical core on the host machine, leaving exactly one physical core entirely isolated to handle the host operating system's background processes. This pinning strategy is mandatory; not only does it mitigate the Noisy Neighbour effect (Muro, Baena, Fortes, Nielsen, & Barco, 2023), but recent empirical studies on edge compute footprints confirm that pinned workload placement fundamentally impacts absolute performance and computational tradeoffs in resource-constrained servers (Veitch, MacNamara, & Browne, 2025). By utilizing strict hypervisor-level pinning, the measured performance metrics accurately reflect the orchestrators' true algorithmic behavior rather than arbitrary resource contention. Within this isolated cluster, two distinct microservices were deployed to evaluate workload-specific resource footprints: podinfo representing a lightweight Go-based application and httpbin representing a

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

e-ISSN : 2541-2019
p-ISSN : 2541-044X

more resource-intensive Python-based application. Both applications were deployed as single replicas to maximise the observability of orchestrator-induced resource surges. Furthermore, the cluster's networking stack was segmented into two distinct configurations to assess the orchestration tax of service mesh adoption: a baseline configuration utilising the NGINX Ingress Controller Community Edition, and a service mesh configuration incorporating Linkerd alongside NGINX to evaluate the overhead of sidecar proxy injection.

Table 2. Experimental Testbed Specifications

Node	Operating System	vCPU	Memory
Control Plane	Talos Linux	2 Cores	4 GB
Worker 1	Talos Linux	2 Cores	4 GB
Worker 2	Talos Linux	2 Cores	4 GB
Load Generator	Alpine Linux	1 Core	1 GB

Progressive Delivery and Fault Injection Strategy

Incident emulation relied on deterministic fault injection executed during active canary rollout phases. Both Flagger and Argo Rollouts were configured to execute an identical canary strategy, advancing in structural increments and increasing the routing weight by 10% every minute until reaching a maximum canary weight of 50%. The data plane traffic splitting was actively managed by the NGINX Ingress Controller, both standalone and within the Linkerd mesh environment, dynamically adjusting routing weights at each canary step consistent with the ingress-based weight-routing model for zero-downtime canary deployments in Kubernetes (Malhotra, Elsayed, Torres, & Venkatraman, 2024). During the progression, the orchestrators were configured with a 10-second analysis interval, evaluating a 30-second rolling window of Prometheus telemetry (e.g., rate[30s]). Both orchestrators were strictly configured to tolerate a maximum of two consecutive metric analysis failures before initiating an automatic deployment abort.

To simulate a comprehensive array of critical application incidents consistent with failure modes documented in container orchestration fault recovery literature (Zhao, He, Luo, Li, & Wu, 2025), the fault injection phase incorporated four distinct disruption scenarios: HTTP 500 Internal Server Errors, latency spikes, memory pressure, and CPU saturation. The memory pressure scenario was specifically designed to exhaust available container memory and induce CrashLoopBackOff, a compound fault state characterised by oscillating pod restarts, as distinct from the threshold-breach signatures produced by the remaining three fault classes. All four scenarios were executed across both application workload runtimes, the lightweight Go-based podinfo and the Python-based httpbin, and across both network topology configurations, the standard NGINX ingress and the Linkerd service mesh, yielding a fully factorial experimental design. For each scenario, continuous baseline traffic was first generated using 50 constant Virtual Users (VUs) via the k6 testing framework, consistent with established k6-based Kubernetes performance benchmarking methodologies (Khamdani, Muslikh, & Affandi, 2025). Immediately following this, the specific fault was injected against the new canary release while maintaining the k6 load, forcing the orchestrators to detect the anomaly within their defined Prometheus polling windows and trigger automated mitigation.

Data Collection and Measurement Procedure

The data collection methodology prioritized high-resolution state tracking across a controlled and reproducible set of automated deployment events. Thirty-six independent trials were executed for each orchestrator configuration across all experimental conditions. Following data consolidation, a two-stage cleaning procedure was applied: incomplete records arising from kubectl log capture window timeouts were excluded, and a trailing window selection retained the final 30 consecutive trials per scenario to isolate steady-state behaviour from cold-start transients, yielding a final clean sample of 30 valid trials per condition for quantitative analysis. A formal a priori power analysis was not conducted prior to data collection, as the exploratory nature of the comparison and the strict computational constraints of the experimental environment imposed a practical ceiling on feasible trial repetitions. This limitation is acknowledged; however, the post-hoc inter-trial variance and the consistently large effect sizes observed across all statistically significant comparisons support the adequacy of the sample for the purposes of this study.

Custom shell scripts polled the Kubernetes API at one-second intervals to capture precise timestamps for state transitions. The primary dependent variable, Time-to-Recover (TTR), was strictly defined as the Incident Mitigation Latency. As shown in Equation (1), this is calculated mathematically as the temporal difference between the fault injection and the mitigation execution:

$$TTR = t_{abort} - t_{inject} \qquad (1)$$

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

where t_{inject} denotes the exact timestamp at which the synthetic fault was activated by the load generator or fault injection mechanism, and t_abort represents the orchestrator log timestamp confirming the deployment abort and the immediate rerouting of traffic away from the canary. This definition was applied consistently across all four fault injection scenarios, with t_inject corresponding to the initiation of HTTP 500 error injection, CPU stress activation, latency spike injection, or memory pressure initiation respectively.

CPU utilization was captured as the instantaneous consumption (millicpu) of the orchestrator's controller pod at each Prometheus scrape interval during the failure phase exclusively. For each trial, two values were recorded: the mean CPU consumption, representing the average utilization across the active failure phase window between t_{inject} and t_{abort}, and the peak CPU consumption, representing the maximum observed value within the same window.

Memory utilization was recorded as a parallel dependent variable during the failure phase. For each trial, the mean memory consumption (MB) of the orchestrator's controller pod was computed across the active failure phase window and retained for cross-scenario and cross-orchestrator comparison. Success phase memory data was retained only for contextual reference and excluded from all inferential comparisons.

Both memory and CPU metrics were aggregated using Prometheus and visualized via Grafana, a combined monitoring stack in which Prometheus handles metric collection while Grafana provides real-time dashboard visualization (Nurmadewi, Mulyadi, & Al Hakim, 2025). A strict 10-second scraping interval was enforced, as scrape interval selection is a critical configuration parameter that directly determines the resolution and accuracy of Kubernetes monitoring data (Huang & Pierre, 2024). High-frequency metric monitoring is essential in constrained environments to successfully capture the transient, high-amplitude computational surges that traditional long-term metric polling entirely masks (Hsu, Bhardwaj, & Gavrilovska, 2024).

Data Analysis and Ethical Considerations

Data analysis involved descriptive and inferential statistics to evaluate the significance of TTR and resource consumption differences across orchestrators, fault scenarios, workload runtimes, and network topology configurations. To validate the strict isolation of the KVM testbed and eliminate the possibility of environmental anomalies, the Coefficient of Variation (CV) was calculated for each metric and experimental condition using Equation (2):

$$CV = \frac{\sigma}{\mu} \times 100 \quad (2)$$

where σ is the sample standard deviation and μ is the sample mean. CV values are reported in the results section alongside descriptive statistics to serve as a testbed stability indicator, confirming that observed between-orchestrator differences reflect genuine architectural behaviour rather than environmental measurement noise.

Prior to mean comparison, the Shapiro-Wilk test was applied to assess the normality of the data distribution for each metric and condition independently, informing the selection of parametric or non-parametric comparison tests. For normally distributed data, an independent two-sample Welch's *t-test* was employed. This specific test was selected over the traditional Student's t-test because it provides a more robust control of Type I error rates when the assumption of equal population variances cannot be definitively met, a common characteristic of heterogeneous software systems (Delacre, Lakens, & Leys, 2017). The t-statistic is calculated as defined in Equation (3):

$$t = \frac{\overline{X_F} - \overline{X_A}}{\sqrt{\frac{s_F{}^2}{n_F} + \frac{s_A{}^2}{n_A}}} \quad (3)$$

where $\bar{X}$, s^2 and n represent the sample mean, sample variance, and sample size for Flagger (F) and Argo Rollouts (A), respectively. In instances where the telemetry data violated normality assumptions, the non-parametric Mann-Whitney U test was applied as the primary comparison, calculated using Equation (4):

$$U = n_1 n_2 + \frac{n_1(n_1+1)}{2} - R_1 \quad (4)$$

where n_1 and n_2 are the sample sizes for the respective orchestrators, and R_1 is the sum of the ranks for the first sample. In cases where Welch's t-test and Mann-Whitney U produced conflicting significance conclusions, the Mann-Whitney U result was treated as primary given the non-normality violation. Furthermore, to quantify the magnitude of the performance differences beyond mere statistical significance, Cohen's d effect size was calculated for all parametric comparisons using Equation (5):

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

$$d = \frac{\bar{x}_1 - \bar{x}_2}{s_p} \quad (5)$$

where s_p represents the pooled standard deviation of the two samples. Alongside the effect size, 95% Confidence Intervals (CI) were calculated for all comparative metrics to characterize the precision and practical significance of observed differences. A strict significance level of $\alpha = 0.05$ was established for all inferential tests.

As this research involves automated infrastructure and non-sensitive system metrics, no human subjects were involved, and no ethical approvals regarding informed consent or personal data were required.

RESULT

Fundamental Orchestrator Dynamics (The Baseline Control)

To evaluate the pure architectural performance of the orchestrators without compound variables, baseline metrics were isolated using the lightweight Go microservice and the standard ingress topology across all fault scenarios. Under standard fault conditions including HTTP 500 errors, CPU stress, and latency injection, Flagger recorded nominally lower total recovery times across all three scenarios, though the between-orchestrator differences were narrow and statistically significant only under HTTP 500 error conditions ($t(39.8) = -4.11, p < .001, d = -1.06$). Total recovery times varied within a narrow band of 0.26 to 3.88 seconds between orchestrators depending on fault type, with Flagger recording 37.15s, 41.95s, and 34.71s against Argo Rollouts' 41.03s, 43.13s, and 34.97s for HTTP 500, CPU stress, and latency scenarios respectively with the exception of latency spike where the primary Mann-Whitney U test detected a statistically significant but operationally negligible difference ($U = 631, p = .007, d = -0.08$), as illustrated in Figure 2.

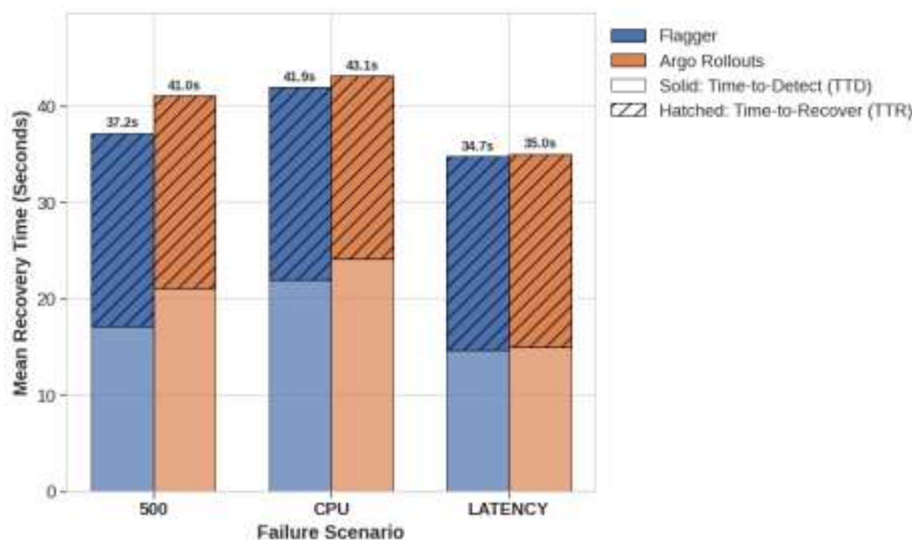

Fig. 2 Breakdown of mean detection latency (Time-to-Detect) and rollback latency (Time-to-Recover) across standard fault scenarios under baseline control conditions

However, this equivalence collapsed entirely under memory exhaustion conditions, exposing a fundamental architectural divergence between the two orchestrators. As described in the methodology, sustained memory exhaustion in this experimental design progressed into CrashLoopBackOff, a compound fault state characterised by oscillating pod restarts rather than the clean metric threshold breaches present in standard fault scenarios. Argo Rollouts executed the complete mitigation lifecycle with a mean total recovery time of 29.67 seconds, maintaining a compact TTD phase consistent with its performance profile observed in standard fault scenarios. Flagger, by contrast, exhibited catastrophic detection latency under these conditions, recording a mean total recovery time of 166.79 seconds, representing a 5.6-fold degradation relative to Argo Rollouts, a difference confirmed as statistically significant with a large effect size ($t(30.7) = 25.55, p < .001, d = 6.60, 95\%\ CI[126.17, 148.07], U = 900, p < .001$), as illustrated in Figure 3.

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

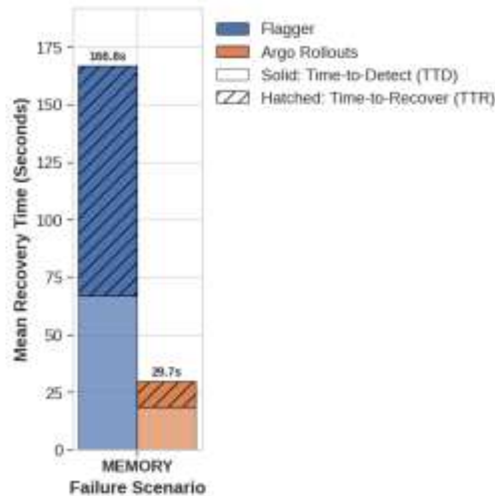

Fig. 3 Breakdown of mean detection latency (Time-to-Detect) and rollback latency (Time-to-Recover) under memory exhaustion fault conditions

Examining mean ambient CPU utilization during the mitigation lifecycle revealed a structurally consistent pattern across standard fault conditions that inverted sharply under memory exhaustion. Flagger maintained a tightly bounded and stable ambient CPU footprint across all standard scenarios, recording 14.17m, 13.01m, and 12.65m under HTTP 500, CPU stress, and latency fault conditions respectively, demonstrating computational predictability regardless of fault type. Argo Rollouts, by contrast, sustained elevated ambient CPU consumption throughout the same scenarios, recording 35.67m, 38.81m, and 34.73m, reflecting the higher baseline processing demands of its reconciliation architecture during active mitigation, a difference confirmed as statistically significant with a large effect size across all three conditions ($p < .001, d$ range -9.00 to -9.22; see Table 4). Under memory exhaustion conditions, however, this relationship inverted unexpectedly. Argo Rollouts recorded a dramatic reduction to just 7.33m, falling below Flagger's 11.87m for the first and only time across all evaluated scenarios ($p < .001, d = 5.56$; see Table 4), as illustrated in Figure 4.

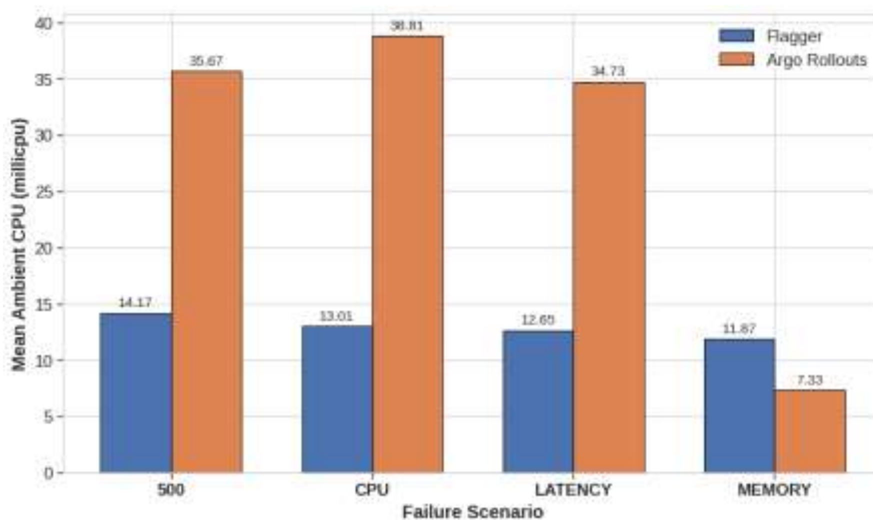

Fig. 4 Mean ambient CPU utilization during failure mitigation lifecycle across all fault scenarios

Peak CPU surge dynamics revealed an even starker architectural divergence that followed a broadly consistent directional pattern across both standard and memory exhaustion conditions, differing only in magnitude. Flagger demonstrated near-absolute computational ceiling stability across all four fault scenarios, recording peak surges of 18.44m, 17.11m, and 16.74m under HTTP 500, CPU stress, and latency conditions respectively, with only a marginal elevation to 17.83m under memory exhaustion. This consistency indicates that Flagger's rollback execution mechanism imposes a structurally bounded transient computational cost that is largely insensitive to the nature of the triggering fault. Argo Rollouts exhibited the opposite characteristic entirely, sustaining extreme transient peak surges of 160.90m, 159.07m, and 167.67m across standard fault scenarios, representing an

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

approximately eight-fold amplification relative to Flagger's ceiling under equivalent conditions, a divergence confirmed as statistically significant with a large effect size across all three conditions ($p < .001, d$ range -8.31 to -13.31; see Table 4). Under memory exhaustion, Argo Rollouts recorded a substantial reduction to 45.16m, and even this attenuated peak remains approximately 2.5 times the absolute maximum peak recorded by Flagger across any scenario, reinforcing that the between-orchestrator divergence in computational burst behaviour is a robust architectural characteristic rather than a scenario-specific artefact, as illustrated in Figure 5.

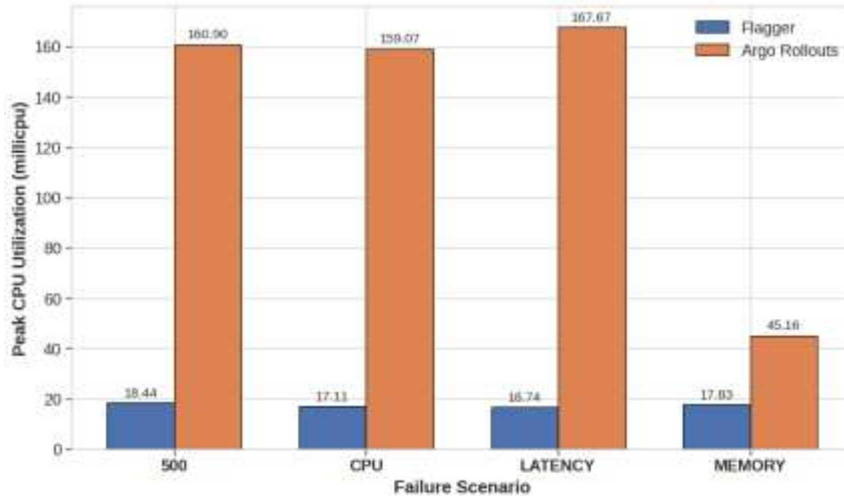

Fig. 5 Peak transient CPU surge utilization during active rollback execution across all fault scenarios

Memory utilization patterns across all four fault scenarios revealed the most structurally stable and consistent findings of the entire baseline evaluation, presenting a marked contrast to the volatile CPU dynamics documented above. Flagger maintained a tightly bounded memory footprint throughout all fault conditions, recording 25.24MB, 23.89MB, 23.44MB, and 23.47MB under HTTP 500, CPU stress, latency, and memory exhaustion scenarios respectively, a range of just 1.80MB across entirely distinct fault classes. Argo Rollouts demonstrated equivalent intra-scenario stability, recording 35.10MB, 35.23MB, 35.12MB, and 36.42MB across the same conditions, a range of 1.32MB. Critically, unlike the CPU metrics documented above, memory utilization produced no inversion, no compression effect, and no fault-sensitive modulation under memory exhaustion conditions. Both orchestrators maintained their respective footprints with near-identical consistency regardless of whether the injected fault was a simple HTTP error or a compound CrashLoopBackOff state. The persistent between-orchestrator gap of approximately 10 to 13MB across all scenarios further confirms this as an inherent architectural characteristic rather than a scenario-dependent artefact, a difference confirmed as statistically significant with a large effect size across all conditions ($p < .001, d$ range -4.30 to -8.21; see Table 4), as illustrated in Figure 6.

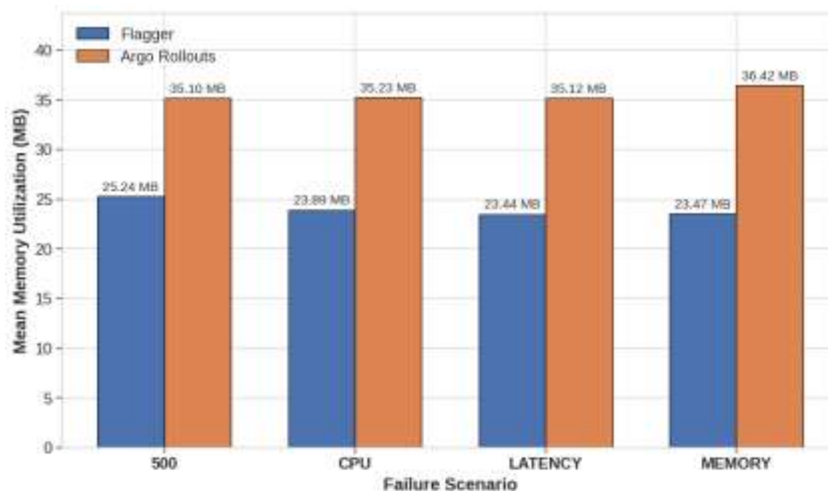

Fig. 6 Mean memory footprint stability across all fault scenarios during failure mitigation

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

SinkrOn
JURNAL & PENELITIAN TEKNIK INFORMATIKA
Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

Table 3 and Table 4 present the complete descriptive and inferential statistics across all metrics and fault scenarios evaluated under baseline conditions.

Table 3. Descriptive statistics of temporal and computational metrics across all fault scenarios under baseline control conditions

Metric	Scenario	*n*	Flagger (SD)	*CV* (%)	*n*	Argo Rollouts (SD)	*CV* (%)
Total Recovery Time (TRT)	HTTP 500	30	37.15 (2.08)	5.59	30	41.03 (4.73)	11.53
	CPU Pressure	30	41.95 (2.37)	5.64	30	43.13 (6.02)	13.96
	Request Latency	30	34.71 (2.92)	8.41	30	34.97 (3.78)	10.82
	Memory Spike	30	166.79 (28.97)	17.37	30	29.67 (4.94)	16.66
Mean CPU (millicpu)	HTTP 500	30	14.17 (0.48)	3.42	30	35.67 (2.94)	8.24
	CPU Pressure	30	13.01 (0.60)	4.62	30	38.81 (4.01)	10.33
	Request Latency	30	12.65 (0.64)	5.06	30	34.73 (3.33)	9.57
	Memory Spike	30	11.87 (0.67)	5.68	30	7.33 (0.94)	12.78
Peak CPU (millicpu)	HTTP 500	30	18.44 (1.41)	7.66	30	160.90 (20.78)	12.92
	CPU Pressure	30	17.11 (1.43)	8.34	30	159.07 (24.11)	15.16
	Request Latency	30	16.74 (1.66)	9.92	30	167.67 (15.95)	9.51
	Memory Spike	30	17.83 (2.26)	12.68	30	45.16 (11.33)	25.09
Memory Usage (MB)	HTTP 500	30	25.24 (0.39)	1.55	30	35.10 (3.22)	9.18
	CPU Pressure	30	23.89 (0.58)	2.41	30	35.23 (3.93)	11.17
	Request Latency	30	23.44 (0.79)	3.36	30	35.12 (3.60)	10.25
	Memory Spike	30	23.47 (0.62)	2.65	30	36.42 (2.14)	5.88

Table 4. Inferential statistics of temporal and computational metrics across all fault scenarios under baseline control conditions

Metric	Scenario	SW Flagger (W, *p*)	SW Rollouts (W, *p*)	Welch t (df), p	Cohen's *d*	95% CI	U, p
TRT (s)	HTTP 500	.924, .035*	.675, < .001*	-4.11 (39.8), < .001	-1.06 (large)	[-5.79, -1.97]	265, .006
	CPU Spike	.800, < .001*	.789, < .001*	-1.00 (37.8), .323 *ns*	-0.26 (small)	[-3.57, 1.21]	482, .639 *ns*
	Latency Spike	.826, < .001*	.546, < .001*	-0.30 (54.5), .766 *ns*	-0.08 (negl.)	[-2.01, 1.49]	631, .007‡
	Memory Spike†	.789, < .001*	.832, < .001*	25.55 (30.7), < .001	6.60 (large)	[126.17, 148.07]	900, < .001
Mean CPU (m)	HTTP 500	.945, .124	.954, .214	-39.52 (30.6), < .001	-10.20 (large)	[-22.62, -20.40]	0, < .001

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

	CPU Spike	.956, .241	.946, .134	-34.86 (30.3), < .001	-9.00 (large)	[-27.32, -24.30]	0, < .001
	Latency Spike	.978, .768	.960, .301	-35.72 (31.1), < .001	-9.22 (large)	[-23.34, -20.82]	0, < .001
	Memory Spike†	.967, .448	.975, .667	21.54 (52.7), < .001	5.56 (large)	[4.11, 4.96]	900, < .001
Peak CPU (m)	HTTP 500	.970, .551	.948, .145	-37.46 (29.3), < .001	-9.67 (large)	[-150.24, -134.68]	0, < .001
	CPU Spike	.928, .044*	.938, .081	-32.19 (29.2), < .001	-8.31 (large)	[-150.97, -132.94]	0, < .001
	Latency Spike	.979, .797	.968, .496	-51.55 (29.6), < .001	-13.31 (large)	[-156.91, -144.95]	0, < .001
	Memory Spike†	.936, .070	.994, > .999	-12.96 (31.3), < .001	-3.35 (large)	[-31.63, -23.03]	3, < .001
Mean Memory (MB)	HTTP 500	.894, .006*	.940, .093	-16.65 (29.9), < .001	-4.30 (large)	[-11.08, -8.65]	0, < .001
	CPU Spike	.834, < .001*	.926, .039*	-15.62 (30.2), < .001	-4.03 (large)	[-12.82, -9.86]	0, < .001
	Latency Spike	.953, .198	.940, .092	-17.36 (31.8), < .001	-4.48 (large)	[-13.05, -10.31]	0, < .001
	Memory Spike†	.788, < .001*	.961, .320	-31.80 (33.8), < .001	-8.21 (large)	[-13.78, -12.12]	0, < .001

† Memory exhaustion progressed to CrashLoopBackOff compound fault. * Shapiro-Wilk indicates non-normality (p < .05); Mann-Whitney U treated as primary test. ‡ Discrepancy between Welch t-test (ns) and Mann-Whitney U (p = .007) for Latency TRT; Mann-Whitney U is treated as primary per the pre-specified test selection rule given non-normality in both groups. The negligible effect size (d = -0.08) confirms no meaningful operational consequence despite statistical significance. ns = not significant. negl. = negligible effect size. U = Mann-Whitney U statistic. All metrics measured during failure phase only.

The Modulating Impact of Workload Runtimes

Isolating the impact of application runtime by holding the standard ingress network topology constant and excluding memory exhaustion scenarios to prevent fault sensitivity confounds revealed that substituting the lightweight Go binary with the heavier Python runtime produced no meaningful computational modulation for either orchestrator under standard fault conditions. Mean ambient CPU utilization remained stable within each orchestrator regardless of runtime, with Flagger recording 12.65m to 14.17m under Go and 13.92m to 14.42m under Python, and Argo Rollouts recording 34.73m to 38.81m under Go and 32.50m to 40.02m under Python, preserving the same between-orchestrator gap observed in the baseline evaluation. Peak CPU surge behaviour was equally unaffected by the runtime substitution, with Flagger recording 16.74m to 18.44m under Go and 18.46m to 18.99m under Python, and Argo Rollouts sustaining 159.07m to 167.67m under Go and 154.33m to 171.43m under Python regardless of application language. The temporal dimension produced a more nuanced picture. Argo Rollouts demonstrated no consistent directional change in TRT across runtimes, recording 41.03s, 43.13s, and 34.97s under Go against 43.37s, 38.03s, and 32.47s under Python with no consistent directional pattern across fault types. Flagger, however, exhibited a consistently higher TRT under Python across all three standard fault scenarios, recording 37.15s, 41.95s, and 34.71s under Go against 46.61s, 49.38s, and 35.36s under Python respectively, suggesting a modest runtime sensitivity in Flagger's mitigation timing that was absent in Argo Rollouts. Consequently, application runtime did not emerge as a meaningful computational performance modulator under standard fault conditions, with orchestrator architecture remaining the dominant differentiating factor, though a directionally consistent temporal elevation was observed for Flagger under the Python workload, as illustrated in Figure 7.

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

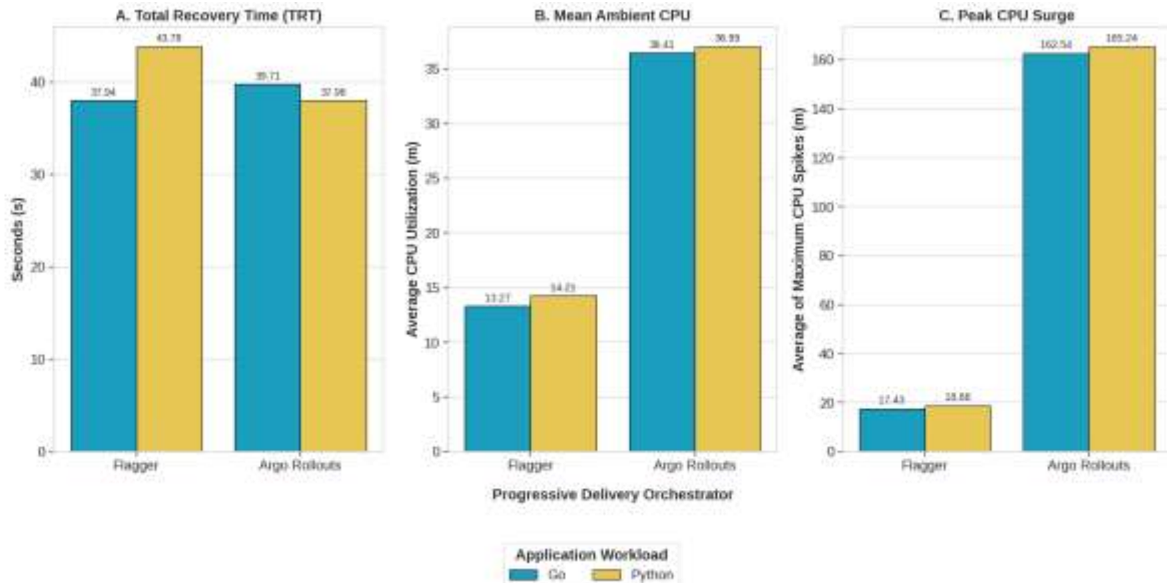

Fig. 7 Workload runtime impact on total recovery time (TRT), mean ambient CPU utilization, and peak CPU surge across progressive delivery orchestrators

The Modulating Impact of Service Mesh Topologies

Isolating the impact of service mesh topology by holding the workload constant and excluding memory exhaustion scenarios to prevent fault sensitivity confounds revealed that the introduction of Linkerd produced no meaningful temporal modulation for either orchestrator under standard fault conditions, with total recovery times remaining broadly stable and directionally inconsistent across both topologies. The more substantive finding was computational and followed opposing directions for each orchestrator. Linkerd's sidecar proxy architecture introduced a modest but consistent increase in Flagger's mean ambient CPU across all standard scenarios, rising from approximately 12.65m to 13.01m under standard ingress to approximately 14.95m to 15.30m under Linkerd mesh. Concurrently, Argo Rollouts demonstrated the opposing response, recording measurable reductions in both mean ambient CPU from approximately 34.73m to 38.81m down to approximately 29.23m to 33.91m, and in peak CPU surge from approximately 159.07m to 167.67m down to approximately 119.91m to 155.07m across the same standard fault scenarios. Flagger's peak CPU ceiling remained effectively stable across both topologies. Consequently, service mesh topology did not emerge as a meaningful temporal modulator, but produced asymmetric computational effects, introducing modest ambient CPU overhead for Flagger while partially attenuating the peak computational demand of Argo Rollouts, as illustrated in Figure 8.

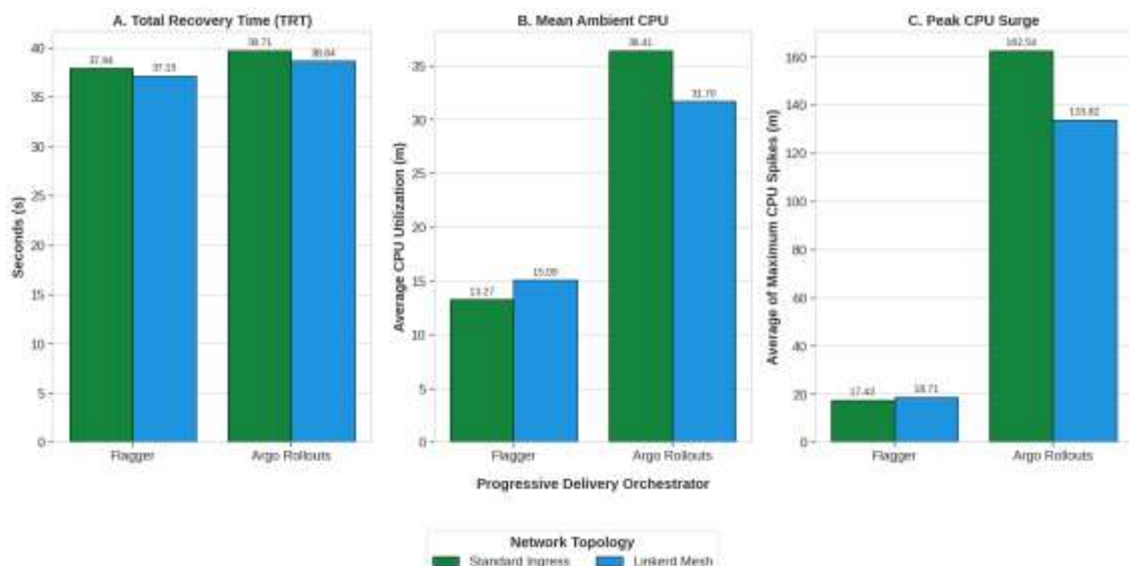

Fig. 8 Network topology impact on total recovery time (TRT), mean ambient CPU utilization, and peak CPU surge across progressive delivery orchestrators

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

DISCUSSION

Recap of Core Findings and Architectural Root Causes

This empirical evaluation provides clear evidence that the choice of progressive delivery orchestrator dictates a fundamental trade-off between computational overhead predictability and fault-type resilience. Under standard fault conditions, both orchestrators demonstrated broadly equivalent temporal mitigation performance, with neither consistently outperforming the other across HTTP 500, CPU stress, and latency fault scenarios. The primary architectural differentiator under these conditions was computational rather than temporal, consistently favouring Flagger's structurally bounded CPU profile against Argo Rollouts' substantially higher and more volatile consumption pattern. However, this apparent equivalence proved contingent on fault type. Under memory exhaustion conditions progressing to CrashLoopBackOff, a fundamental architectural sensitivity in Flagger's detection mechanism produced a 5.6-fold degradation in total recovery time relative to Argo Rollouts, a difference confirmed as statistically significant with a large effect size ($t(30.7) = 25.55$, $p < .001$, $d = 6.60$).

This fault-type-dependent temporal divergence and the persistent CPU volatility of Argo Rollouts are not arbitrary, but rather direct reflections of their distinct internal software architectures and their interactions with the Kubernetes control plane. Flagger's architecture implements a synchronous control loop that periodically analyzes metrics and promotes or rolls back the canary based on predefined thresholds, which is documented in the official Flagger project documentation (Flagger, 2025). Because the detection and the mitigation occur synchronously without requiring the generation of secondary Kubernetes resources, Flagger minimizes Kubernetes API write amplification, resulting in a tightly bounded CPU footprint and consistent temporal performance under threshold-breach fault conditions. However, this synchronous, metric-threshold-based detection mechanism is structurally ill-suited to CrashLoopBackOff fault states, where service degradation manifests as oscillating pod restarts rather than sustained metric violations, explaining the catastrophic detection latency observed under memory exhaustion conditions.

Conversely, Argo Rollouts utilizes a more complex, decentralized architectural pattern, in which metric evaluation is managed through dedicated Custom Resources (AnalysisTemplate and AnalysisRun) that are separate from the primary Rollout object, as documented in the Argo Rollouts architecture guide (Argo Rollouts, 2025). To execute this analysis, the operator leverages the Go runtime's concurrency model, spawning multiple concurrent goroutines to poll background metrics asynchronously. When an anomaly is detected, Argo Rollouts cannot instantly abort the traffic. Instead, it must execute a graceful termination sequence: sending context cancellation signals to all active goroutines, waiting for work queues to drain, updating the AnalysisRun resource status to a "Failed" state via a Kubernetes API PATCH request, and only then updating the parent Rollout object to initiate the traffic shift. This multi-stage state reconciliation fundamentally explains the persistent CPU volatility observed across all standard fault scenarios. Critically, however, Argo Rollouts' pod-level state monitoring architecture, which operates independently of metric threshold evaluation, enables it to detect and respond to CrashLoopBackOff conditions with substantially lower latency than Flagger's probe-based mechanism, directly accounting for the 5.6-fold temporal advantage observed under memory exhaustion. Recent systems research on fault-tolerant Kubernetes operator design demonstrates that multi-stage state reconciliation and repeated API interactions introduce measurable computational overhead in controller-based architectures (Schmidt, Rejiba, Eidenbenz, & Förster, 2023).

Methodological Limitations

While this study utilized a highly controlled environment, several general methodological limitations must be acknowledged. First, regarding hardware scale constraints, the experimental testbed was intentionally limited to 6 vCPUs and 12 GB of RAM to emulate resource-rationed edge environments. Consequently, the proportional impact of the CPU surges might manifest differently on large-scale enterprise clusters equipped with high-frequency processors and extensive CPU headroom. Second, while this study extended fault injection coverage to include HTTP 500 errors, CPU stress, latency degradation, and memory exhaustion progressing to CrashLoopBackOff, additional critical failure domains remain unevaluated, including network partitions, cascading service failures, and storage I/O degradation, which may trigger distinct reconciliation behaviours and expose additional architectural sensitivities not captured by the fault classes evaluated here. Third, while this study extended workload coverage to include both a lightweight Go microservice and a Python runtime to partially address interpretive runtime concerns, real-world enterprise applications often involve heavy Java Virtual Machine (JVM) frameworks such as Spring Boot, which suffer from significantly longer start-up times compared to lightweight native binaries, as empirically demonstrated in containerized microservice environments (Wyciślik, Latusik, & Kamińska, 2023). The introduction of these heavier runtimes into the progressive delivery pipeline could substantially alter the baseline TTR measurements established in this study. Fourth, the service mesh evaluation was limited to a single implementation, Linkerd, which emphasises a lightweight sidecar proxy architecture. Alternative service mesh implementations such as Istio employ substantially more complex control plane architectures and may produce different computational overhead profiles and temporal modulation effects

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

than those observed in this study. Finally, a minor data collection asymmetry is acknowledged between the two orchestrators. The kubectl log capture window was extended for Flagger configurations following pilot observation but not uniformly applied to Argo Rollouts, introducing a conservative bias in which excluded trials represent the subset with longer recovery durations. This asymmetry marginally understates Argo Rollouts' worst-case temporal performance and should be addressed in future replications.

Coherence with Existing Literature

These findings strongly align with existing cloud-native engineering documentation regarding controller performance. The observed state reconciliation overhead corroborates findings on the computational costs introduced by multi-stage API interactions in complex Kubernetes operator architectures (Schmidt, Rejiba, Eidenbenz, & Förster, 2023). Furthermore, these findings build upon recent comparative systems research which establishes that the baseline control plane components in Kubernetes (such as the API and Metrics Server) inherently consume higher foundational CPU and memory resources than isolated container engines (Suryayusra, Destarina, Negara, Supratman, & Ulfa, 2024). The collected data demonstrates that injecting asynchronous progressive delivery orchestrators on top of this already heavy Kubernetes baseline creates extreme computational transients, reinforcing the practical latency costs and strict resource management required for these closed-loop control systems in resource-constrained environments. The fault-type-dependent divergence observed under CrashLoopBackOff conditions further extends existing literature by demonstrating that probe-based metric threshold detection mechanisms exhibit structural limitations under oscillating pod failure states, a failure class not extensively evaluated in the context of progressive delivery orchestrator comparative performance.

Incidental Observations

An unexpected but noteworthy finding during the data collection phase was the complete decoupling of CPU volatility from memory consumption. While Argo Rollouts experienced extreme transient CPU surges reaching 159.07m to 167.67m across standard fault scenarios, its memory footprint remained remarkably static across the same conditions. This pattern suggests that the AnalysisRun overhead manifests as a processing-bound rather than memory-bound bottleneck, consistent with computational costs driven by serialization, network I/O, and concurrent scheduling operations rather than heap allocation.

A secondary incidental observation was the unexpected inversion of mean ambient CPU consumption under memory exhaustion conditions, where Argo Rollouts recorded a dramatic reduction to 7.33m, falling below Flagger's 11.87m for the first and only time across all evaluated scenarios. This inversion was not a product of improved efficiency but rather a compression artefact attributable to Argo Rollouts' rapid 29.67 second mitigation window, which inherently limited the duration of measurable CPU activity during the failure phase.

A further incidental observation arising from the service mesh evaluation was the asymmetric and opposing directional response of the two orchestrators to Linkerd's sidecar proxy architecture. Rather than producing a uniform overhead increase across both orchestrators as might be anticipated, Linkerd introduced a modest CPU increase for Flagger while simultaneously reducing both mean ambient and peak CPU consumption for Argo Rollouts, suggesting that Linkerd's traffic interception layer partially offloaded the metric polling workload otherwise performed by Argo Rollouts' goroutine-based concurrency model.

Implications for Practice and Policy

For Site Reliability Engineering (SRE) teams and platform administrators, this data carries immediate operational implications. Policymakers designing cluster resource quotas must account for the dynamic nature of these tools. SREs utilizing Argo Rollouts must size the controller's CPU requests and limits based on the peak surge metrics, reaching 167.67m under latency fault condition, rather than the idle average. Recent research highlights that when orchestration controllers exceed their allocated resource quotas, the underlying platform enforces strict CPU throttling; this abrupt suspension freezes the container context, forcing developers to implement complex retry logic and catastrophically delaying automated incident recovery (Gajanin, Marcelino, & Nastic, 2025). Conversely, teams operating in environments where fault signatures are well-defined and dominated by threshold-breach conditions such as HTTP errors, CPU stress, and latency degradation should consider Flagger for its structurally bounded and predictable CPU ceiling, accepting the trade-off of elevated vulnerability to CrashLoopBackOff fault states. Teams operating in environments where pod-level instability and memory pressure are anticipated failure modes should instead prefer Argo Rollouts for its superior CrashLoopBackOff resilience, accepting its substantially higher transient CPU demands as the operational cost of broader fault-type coverage.

Threats to Validity

To ensure academic rigour, potential threats to validity were systematically categorised and mitigated across four dimensions. Regarding internal validity, hypervisor-level CPU pinning enforced one-to-one vCPU-to-

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026 e-ISSN : 2541-2019
DOI : https://doi.org/10.33395/sinkron.v10i3.16201 p-ISSN : 2541-044X

physical-core binding across all nodes, and the k6 load generator was isolated on a dedicated node, eliminating CPU contention and ensuring observed metrics reflect orchestrator behaviour rather than environmental noise. Regarding construct validity, the 10-second Prometheus scraping interval and 30-second rolling window are representative of production-grade constrained monitoring stacks, and the mathematical decomposition of Total Recovery Time into Time-to-Detect and Time-to-Recover components strengthens measurement precision by isolating detection latency from mitigation execution. Regarding external validity, the explicit focus on a low-vCPU edge topology limits direct extrapolation to hyperscaler-managed environments; however, the multi-workload and multi-topology design preserves the generalisability of the relative architectural trade-offs between orchestrators for comparative decision-making in constrained clusters. Regarding conclusion validity, a rigorous two-stage data cleaning procedure established 30 valid trials per configuration, and normality violations were addressed through Shapiro-Wilk testing with Mann-Whitney U applied as the primary test where parametric assumptions were violated, ensuring all reported significance levels are mathematically sound.

CONCLUSION

This study undertook an empirical evaluation of automated incident recovery performance in Kubernetes progressive delivery, specifically contrasting Flagger and Argo Rollouts within a strictly resource-constrained environment. The research successfully quantified the fault-type-dependent relationship between recovery latency and architectural resource volatility in a setting that departs from traditional hyperscale assumptions.

These findings directly address the two core research questions posed by this study. In response to RQ1, the architectural divergence between Flagger's synchronous, metric-threshold-based detection mechanism and Argo Rollouts' event-driven, pod-level state monitoring is identified as the root cause of the fault-type-dependent mitigation latency divergence. Under standard fault conditions including HTTP 500 errors, CPU stress, and latency injection, neither orchestrator demonstrated a consistent temporal advantage, with total recovery times varying within a narrow band across scenarios. However, under memory exhaustion conditions progressing to CrashLoopBackOff, Argo Rollouts executed complete mitigation in a mean of 29.67 seconds while Flagger's probe-based detection produced a mean total recovery time of 166.79 seconds, a 5.6-fold degradation directly attributable to the structural incompatibility between Flagger's threshold-breach detection and the oscillating pod failure signature of CrashLoopBackOff. Application workload runtime and service mesh topology produced no meaningful temporal modulation under standard fault conditions, confirming orchestrator architecture as the dominant performance determinant. In response to RQ2, the empirical data quantifies the true orchestration tax in a constrained environment. Argo Rollouts generated extreme transient peak CPU surges of 159.07m to 167.67m across standard fault scenarios, representing an approximately eight to ten-fold amplification relative to Flagger's tightly bounded ceiling of 16.74m to 18.44m. Notably, memory utilization remained statically bounded for both orchestrators across all fault scenarios, confirming that the observed computational volatility is exclusively a processing overhead phenomenon rather than a memory allocation issue.

The findings confirm that the choice of progressive delivery orchestrator dictates a fundamental operational trade-off between computational overhead predictability and fault-type resilience rather than a simple universal performance hierarchy. Flagger provides a highly predictable, bounded computational footprint with consistent temporal performance under threshold-breach fault conditions but exposes the edge node to multi-minute service degradation during CrashLoopBackOff events. Conversely, Argo Rollouts demands substantial CPU headroom during active deployments but delivers superior fault-type resilience across a broader class of failure modes.

The data provides strong empirical evidence against the prevailing industry assumption that deployment automation tooling is computationally negligible. For organizations operating edge-computing nodes or low-vCPU environments, resource quotas must be deliberately sized to accommodate transient orchestration surges, as failure to do so risks triggering CPU throttling precisely during active incident mitigation. Based on the empirical results across multiple fault scenarios, workload runtimes, and network topology configurations, there is no universally superior orchestrator; rather, there is an optimal tool for specific infrastructure profiles and anticipated failure modes. Flagger is recommended for hyper-constrained edge nodes dominated by threshold-breach fault signatures, while Argo Rollouts is recommended where broader fault-type resilience is operationally critical and sufficient CPU headroom exists.

While this study carefully isolated the experimental testbed and evaluated compound variables including heterogeneous workload runtimes and service mesh integration, a primary limitation remains its focus on a specific low-vCPU edge topology and stateless microservice architectures. Future studies should investigate how these orchestrators perform in massively scaled, hyperscaler-managed environments to assess whether the orchestration tax documented here diminishes as compute headroom becomes abundant. Furthermore, it is critical to extend this research to heavier enterprise workloads, particularly JVM-based applications such as Spring Boot, where significantly longer startup times and higher baseline resource consumption may substantially alter the TTR and surge dynamics documented in this study. Finally, longitudinal studies evaluating the compounding latency overhead of complex multi-service incident cascades, stateful application rollbacks, and database-integrated

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

e-ISSN : 2541-2019
p-ISSN : 2541-044X

progressive delivery would provide further visibility into the operational limits of deployment automation in production-grade Kubernetes environments.

REFERENCES

Amaro, R., Pereira, R., & Silva, M. M. da. (2025). Mapping DevOps capabilities to the software life cycle: A systematic literature review. *Information and Software Technology, 177*, 107583. https://doi.org/10.1016/j.infsof.2024.107583

Aqasizade, H., Ataie, E., & Bastam, M. (2025). Kubernetes in action: Exploring the performance of Kubernetes distributions in the cloud. *Software: Practice and Experience, 55*(10), 1711–1725. https://doi.org/10.1002/spe.70000

Argo Rollouts. (2025). *Architecture—Kubernetes progressive delivery controller*. https://argo-rollouts.readthedocs.io/en/stable/architecture/

Delacre, M., Lakens, D., & Leys, C. (2017). Why psychologists should by default use Welch's t-test instead of Student's t-test. *International Review of Social Psychology, 30*(1), 92–101. https://doi.org/10.5334/irsp.82

Deng, S., Zhao, H., Huang, B., Zhang, C., Chen, F., Deng, Y., ... Zomaya, A. Y. (2024). Cloud-native computing: A survey from the perspective of services. *Proceedings of the IEEE, 112*(1), 12–46. https://doi.org/10.1109/JPROC.2024.3353855

Flagger. (2025). *How it works*. Retrieved May 11, 2026, from https://docs.flagger.app/usage/how-it-works

Gajanin, R., Marcelino, C., & Nastic, S. (2025). Performance isolation for serverless functions. *IEEE Transactions on Services Computing, 18*(6), 4408–4424. https://doi.org/10.1109/TSC.2025.3619151

Ghosh, S., Mavromatis, I., Antonakoglou, K., & Katsaros, K. (2025). Performance evaluation of intent-based networking scenarios: A GitOps and Nephio approach. In *2025 IEEE Conference on Standards for Communications and Networking (CSCN)* (pp. 1–7). IEEE. https://doi.org/10.1109/CSCN67557.2025.11230760

Hsu, K.-J., Bhardwaj, K., & Gavrilovska, A. (2024). Colibri: Efficient collection of fine-grained resource metrics necessary for mobile edge computing. In *2024 IEEE/ACM Symposium on Edge Computing (SEC)* (pp. 29–44). IEEE. https://doi.org/10.1109/SEC62691.2024.00011

Huang, C.-K., & Pierre, G. (2024). Aggregate monitoring for geo-distributed Kubernetes cluster federations. *IEEE Transactions on Cloud Computing, 12*(4), 1449–1462. https://doi.org/10.1109/TCC.2024.3482574

Khamdani, A. R., Muslikh, A. R., & Affandi, A. S. (2025). Comparative analysis of performance and efficiency of load balancing algorithms on ingress controller. *Jurnal Teknik Informatika (JUTIF), 6*(1), 453–468. https://doi.org/10.52436/1.jutif.2025.6.1.4040

Liang, Y., Govindan, R., & Park, S. J. (2025). Granular resource demand heterogeneity. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems* (pp. 187–194). Association for Computing Machinery. https://doi.org/10.1145/3713082.3730379

Malhotra, A., Elsayed, A., Torres, R., & Venkatraman, S. (2024). Evaluate canary deployment techniques using Kubernetes, Istio, and Liquibase for cloud-native enterprise applications to achieve zero downtime for continuous deployments. *IEEE Access, 12*, 87883–87899. https://doi.org/10.1109/ACCESS.2024.3416087

Mondal, S. K., Zheng, Z., & Cheng, Y. (2024). On the optimization of Kubernetes toward the enhancement of cloud computing. *Mathematics, 12*(16), 2476. https://doi.org/10.3390/math12162476

Muro, F., Baena, E., Fortes, S., Nielsen, L., & Barco, R. (2023). Noisy neighbour impact assessment and prevention in virtualized mobile networks. *IEEE Transactions on Network and Service Management, 20*(1), 415–425. https://doi.org/10.1109/TNSM.2022.3194137

Navarro, A. S., Garcia-Pineda, M., & Gutierrez-Aguado, J. (2024). PodInsights: A millisecond pod metric collector for Kubernetes. In *Proceedings of the 12th Euro American Conference on Telematics and Information Systems* (pp. 1–4). Association for Computing Machinery. https://doi.org/10.1145/3685243.3685281

Nurmadewi, D., Mulyadi, Y. R., & Al Hakim, S. (2025). Comparative study of PRTG and Grafana-Prometheus for monitoring information technology infrastructure. In *2025 IEEE 2nd International Conference on Cryptography, Informatics, and Cybersecurity (ICoCICs)* (pp. 452–457). IEEE. https://doi.org/10.1109/ICoCICs68032.2025.11384063

Sathish, P., Avula, S. R., B. U., D., & Bhat, A. G. (2025). Bridging the benchmarking gap: Performance evaluation of popular Kubernetes-based orchestration frameworks. In *2025 International Conference on Computing Technologies & Data Communication (ICCTDC)* (pp. 1–5). IEEE. https://doi.org/10.1109/ICCTDC64446.2025.11159041

Schmidt, H., Rejiba, Z., Eidenbenz, R., & Förster, K.-T. (2023). Transparent fault tolerance for stateful applications in Kubernetes with checkpoint/restore. In *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)* (pp. 129–139). IEEE. https://doi.org/10.1109/SRDS60354.2023.00022

Sebrechts, M., Ramlot, T., Borny, S., Goethals, T., Volckaert, B., & De Turck, F. (2022). Adapting Kubernetes controllers to the edge: On-demand control planes using Wasm and WASI. In *2022 IEEE 11th International*

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16201
e-ISSN : 2541-2019
p-ISSN : 2541-044X

Conference on Cloud Networking (CloudNet) (pp. 195–202). IEEE. https://doi.org/10.1109/CloudNet55617.2022.9978884

Shen, S., Feng, Y., Xu, M., Ren, Y., Wang, X., Leung, V. C. M., & Wang, W. (2024). Tango: Harmonious optimization for mixed services in Kubernetes-based edge clouds. *IEEE Transactions on Services Computing, 17*(6), 4354–4367. https://doi.org/10.1109/TSC.2024.3479926

Suryayusra, S., Destarina, N., Negara, E. S., Supratman, E., & Ulfa, M. (2024). Implementation Docker and Kubernetes scaling using horizontal scaler method for WordPress services. *Sinkron, 8*(4), 2192–2196. https://doi.org/10.33395/sinkron.v8i4.14091

Turin, G., Borgarelli, A., Donetti, S., Damiani, F., Johnsen, E. B., & Tapia Tarifa, S. L. (2023). Predicting resource consumption of Kubernetes container systems using resource models. *Journal of Systems and Software, 203*, 111750. https://doi.org/10.1016/j.jss.2023.111750

Veitch, P., MacNamara, C., & Browne, J. J. (2025). CPU pinning impact on performance and power consumption of mixed workloads at edge. In *2025 28th Conference on Innovation in Clouds, Internet and Networks (ICIN)* (pp. 1–5). IEEE. https://doi.org/10.1109/ICIN64016.2025.10942678

Wyciślik, Ł., Latusik, Ł., & Kamińska, A. M. (2023). A comparative assessment of JVM frameworks to develop microservices. *Applied Sciences, 13*(3), 1343. https://doi.org/10.3390/app13031343

Zhang, G., Guo, W., Tan, Z., Guan, Q., & Jiang, H. (2025). KIS-S: A GPU-aware Kubernetes inference simulator with RL-based auto-scaling. In *2025 IEEE International Performance, Computing, and Communications Conference (IPCCC)* (pp. 1–8). IEEE. https://doi.org/10.1109/IPCCC66453.2025.11304654

Zhao, C., He, M., Luo, S., Li, B., & Wu, L. (2025). Research on dynamic monitoring and fault recovery mechanism for container orchestration platforms. In *2025 7th International Conference on Next Generation Data-Driven Networks (NGDN)* (pp. 27–32). IEEE. https://doi.org/10.1109/NGDN66208.2025.11182115

*name of corresponding author
This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.