International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Cost-Optimal Dynamic Provisioning of Ephemeral Microservice Environments in Distributed Continuous Integration Pipelines

Gangadhar Chalapaka

Abstract: Integration testing within distributed microservice CI pipelines presents a persistent infrastructure dilemma: shared static staging environments serialize developer workflows and generate cross-team contamination, while statically replicated parallel environments consume cloud resources linearly with team growth. Neither approach scales economically or operationally at enterprise scale. This paper presents an automated framework that resolves this trade-off by provisioning per-pull-request ephemeral environments constrained to a blast radius sub-graph derived from a statically analyzed service dependency Directed Acyclic Graph (DAG). A custom Kubernetes Operator maps each incoming code mutation to the minimal set of affected services, provisions them onto serverless spot-instance backends, and replaces out-of-scope dependencies with dynamically generated mock interfaces, achieving near-zero idle infrastructure cost. Evaluation against the DeathStarBench Social Network workload — 420 pull requests simulated over 24 hours across a 27-service polyglot topology — demonstrates 74.02% infrastructure cost savings against static staging and consistent provisioning latency below four minutes regardless of total service catalog size. Total infrastructure spend under the proposed framework was $73.80, compared to $284.10 for the static baseline and $198.50 for a naive ephemeral model. These results establish that dependency-aware orchestration eliminates the staging dichotomy without reintroducing the scheduling bottlenecks characteristic of naive full-environment ephemeral strategies.

Keywords: *Blast Radius Provisioning, Continuous Integration, Directed Acyclic Graph, Ephemeral Environments, Kubernetes Operator, Microservices, Spot Instances*

1. Introduction

1.1 The Operational Reality of Distributed Microservice CI

Twelve years into the microservices era, the architectural premise has largely delivered on its promise of team autonomy and deployment velocity [1]. What it has not resolved — and what practitioners at enterprise-scale organizations encounter daily — is the infrastructure challenge created at the integration testing boundary. A distributed team cannot validate its changes in isolation. The modified service must execute alongside its runtime dependencies, and provisioning those dependencies for every pull request is where the cost and latency math breaks down [2].

At organizations operating large-scale hybrid cloud platforms, two provisioning patterns dominate integration test environments: shared static staging and static parallel replication. Both fail under distinct but equally disqualifying conditions. Shared environments introduce serialization queues that grow with developer count and contamination events that are structurally unavoidable when concurrent teams modify overlapping state. Parallel static environments eliminate queuing but introduce cost scaling that is linear with team count and continuous regardless of whether any testing is occurring — a particularly wasteful property given that most cloud infrastructure sits idle during off-peak hours and weekends [6].

The framework presented here addresses this structural mismatch by treating environment provisioning as a graph computation problem. Rather than replicating service topology, the system reasons about topology: constructing a DAG of service dependencies, computing the exact blast radius of each code change, and provisioning only the nodes within that radius. Everything outside the radius is replaced by mock interfaces at the service mesh layer. The result is an ephemeral environment that is minimal by construction — not by manual curation.

1.2 Why Existing Tooling Does Not Bridge This Gap

Helm, Kustomize, and comparable declarative tooling have made environment templating tractable for steady-state deployments [3][4]. The limitation of these tools in a CI context is not expressiveness — it is granularity. Existing ephemeral environment tooling operates at the topology level, provisioning

Netskope Inc., USA
Email: MeetGangadharC@gmail.com

a copy of the full service catalog per pull request. For a 27-service application such as the Social Network profile used in this evaluation, that means spinning up MongoDB shards, Redis clusters, and a dozen unrelated microservices for a change that touches a single front-end template [5]. The waste is structural: it is not possible to eliminate it through configuration without introducing the very dependency reasoning this framework automates.

Research on CI/CD pipeline optimization has examined reactive auto-scaling and VM-level bin-packing, both of which assume slowly varying load profiles mismatched to the bursty, short-lived workloads generated by pull request pipelines [2][10]. The dependency-aware orchestration tier developed in this work addresses the gap these approaches leave unresolved.

1.3 Contributions

Three technical contributions form the core of this work. First, an automated static analysis pipeline extracts service dependency structure from multi-repository infrastructure code, service mesh manifests, and API contract definitions, producing a live DAG reflecting the actual service topology. Second, a blast radius algorithm maps code mutations to DAG nodes and computes the minimal provisioning sub-graph G' under a configurable execution radius R, bounding the provisioned footprint to direct upstream and downstream dependencies of the mutated service. Third, a Go-based Kubernetes Operator operationalizes this computation as a continuous reconciliation loop, generating Custom Resource Definition (CRD)-driven ephemeral namespaces on serverless spot infrastructure with deterministic teardown on test completion.

2. Related Work and System Architecture

2.1 Positioning in the CI/CD Cost Optimization Literature

Cloud cost management in DevOps pipelines has received sustained research attention. Early work concentrated on VM-level auto-scaling and bin-packing algorithms that adjust cluster capacity to observed CPU and memory utilization [10][11]. These reactive mechanisms suit steady-state serving workloads but are structurally mismatched to CI pipeline profiles: a validation burst lasts minutes, reactive scaling responds over a similar timescale, and by the time capacity is available the workload has terminated [2].

Container-level provisioning tools — Helm, Kustomize, and Infrastructure as Code frameworks more broadly — have shifted the automation boundary closer to the application tier [3][4][12]. Their adoption has been wide, but the dominant implementation pattern when used for ephemeral environments remains naive: every pull request triggers a full-topology replica regardless of which service is being tested [5]. The blast radius constraint introduced in this work is the missing layer between topology declaration and dependency-aware provisioning.

Research on spot instance scheduling has established that binding workloads to preemptible capacity at submission time — rather than migrating them after the fact — produces more reliable cost savings across bursty workloads [6]. The Operator architecture developed here applies this insight at the environment level, assigning ephemeral namespaces to spot node pools as part of the CRD reconciliation rather than as a post-hoc optimization.

2.2 System Architecture

The proposed framework decomposes into three functionally independent processing layers: the Git Ingestion and Diff Engine, the Graph-Based Orchestration Controller (comprising the Graph Discovery Engine and Delta Provisioner), and the Ephemeral Target Cluster Runtime [7][8].

The Git Ingestion and Diff Engine operates as an event-driven webhook receiver. Upon detecting a pull request event or a push to an open branch, it isolates modified file paths and identifies which microservice image requires recompilation. The Graph Discovery Engine runs as a persistent background daemon, continuously parsing service mesh manifests — Istio VirtualServices, Linkerd TrafficSplits — and multi-repository Helm chart dependency declarations to maintain an in-memory DAG of the live service topology. The Delta Provisioner Engine consumes the diff payload, queries the Graph Discovery Engine for blast radius computation, generates Helm values overrides scoped to the computed sub-graph, and drives the Kubernetes Operator through the cluster API.

Component boundaries are intentionally loose-coupled: the Ingestion Engine emits structured events over an internal queue, the Graph Discovery Engine exposes a read-only query interface, and the Provisioner holds all cluster-mutation logic. This separation allows the topology model to evolve independently of provisioning decisions — a critical

property when service catalogs change rapidly across parallel development tracks [9].

2.3 Blast Radius: Formal Definition

The service topology is represented as G = (V, E), where each node in V corresponds to a discrete microservice and each directed edge in E represents a synchronous RPC call or an asynchronous event dependency [5][21]. When pull request p mutates service v_m, the Delta Provisioner computes the minimal provisioning sub-graph G'_p under execution radius R = 1:

V'_p = {v_m} ∪ {v ∈ V | (v, v_m) ∈ E} ∪ {v ∈ V | (v_m, v) ∈ E}

Services in V \ V'_p are not omitted — omission would produce connection failures — but are actively replaced by a mock container. The Istio service mesh routes traffic destined for any v_out ∉ V'_p to a response-generating daemon via namespace-scoped VirtualService rewrites [13]. This daemon parses the target service's OpenAPI specification or gRPC Protobuf schema and returns structurally valid, deterministic responses without pulling or initializing the actual service image.

Table 1 (below) — EphemeralEnvironment CRD field specification

CRD Field	Type	Description
sourceBranch	string	Feature branch ref triggering the PR
targetService	string	Microservice image requiring recompilation
blastRadiusDepth	integer	Execution radius R; default = 1
ttlMinutes	integer	Namespace TTL before automatic teardown

3. Cost Model and Operator Implementation

3.1 Formal Cost Analysis

The economic basis for delta-driven provisioning is expressible as an optimization problem. Let M denote the complete microservice set and C_i the infrastructure cost per unit time for service i. Static staging accrues cost across all services for the full operational window T, independent of test activity:

Cost_static = Σ_{t=0}^{T} Σ_{i ∈ M} C_i · t

The dynamic model ties cost to active test execution windows. For each pull request p in the set P processed within T, let τ_p denote the integration test execution duration and M_p ⊂ M the blast-radius-constrained service subset:

Cost_dynamic = Σ_{p ∈ P} Σ_{j ∈ M_p} (C_j · τ_p) + Σ_{p ∈ P} δ_provision · |M_p|

The term δ_provision · |M_p| captures per-environment initialization overhead, which is bounded and short-lived. Because stateful backends — database clusters, caches, and message brokers — typically carry the highest per-minute cost C_i in a microservice topology, and because these services almost never appear within the blast radius of a typical front-end or business-logic change, the |M_p| ≪ |M| inequality translates directly into large cost differentials [6]. The empirical results in Section 5 confirm this analytically derived expectation.

3.2 Kubernetes Operator: Reconciliation Design

The Go-based Operator implements the standard Controller-Runtime reconciliation pattern against the EphemeralEnvironment CRD [8]. Each CRD instance encodes four fields: the source branch reference, the target microservice identifier, the blast radius depth R, and the namespace TTL in minutes. The reconciliation loop executes four sequential phases: parsing the DAG mapping from the Graph Discovery Engine's query cache; generating Helm values overrides scoped to V'_p; compiling and applying namespace-scoped Istio VirtualService and DestinationRule manifests for mock routing; and submitting pod specifications to the serverless node pool with AWS Fargate profile selectors and spot interruption handling annotations. TTL enforcement uses a finalizer-based garbage collection mechanism. When the TTL expires or a pipeline-complete webhook is received, the

Operator removes the EphemeralEnvironment CRD instance, triggering cascading namespace deletion and pod termination through the standard Kubernetes garbage collection path. All teardown logic resides at the control plane level; no cleanup code runs inside test pods [8][14].

3.3 Mesh-Layer Traffic Isolation

Precise traffic isolation within G'_p depends on the mock routing layer's accuracy. The Operator generates namespace-scoped Istio DestinationRules that map each out-of-scope service hostname to the mock daemon's cluster-internal address [13]. Envoy sidecar proxies — injected automatically by the Istio control plane into every pod within the ephemeral namespace — intercept outbound requests at the TCP layer before they leave the namespace boundary. When the destination hostname resolves to a mock-mapped DestinationRule entry, the proxy forwards the request to the daemon rather than the real service.

From the perspective of services within G'_p, observable network behavior is identical to behavior in a full-topology deployment. API contract compliance, error handling logic, and retry policies are all exercised against structurally correct response payloads [13][15]. Latency characteristics differ — mock responses are faster — but latency-sensitive correctness tests are outside the integration validation scope this framework targets.

4. Experimental Methodology

4.1 AWS Testbed Configuration

The evaluation infrastructure was provisioned entirely within Amazon Web Services to maximize reproducibility and eliminate hardware variability as a confounding factor [16]. An Amazon Elastic Kubernetes Service cluster running Kubernetes version 1.30 served as the primary orchestration platform, with a three-node control plane using m6i.xlarge instances distributed across availability zones us-east-1a, us-east-1b, and us-east-1c. This configuration guaranteed control plane availability throughout the 24-hour evaluation window and isolated the Graph Discovery Engine, Git Ingestion Daemon, and Operator controller processes from ephemeral workload scheduling pressure [8].

Two data plane pools handled distinct workload classes. The static staging baseline ran on managed EC2 node groups using c6i.large on-demand instances, maintaining consistent pricing for cost comparison purposes. Ephemeral workloads under the proposed framework executed on a mixed pool of AWS Fargate profiles and Knative-managed Spot node groups using c6i.large and m6i.large target instance families. These configurations offer substantially lower per-hour costs than equivalent on-demand capacity, at the cost of preemption risk — a risk the Operator's TTL-bound workload lifecycle is specifically designed to tolerate [6]. Network isolation between pull request namespaces was enforced via Kubernetes NetworkPolicies applied by the AWS VPC CNI provider on private subnets.

4.2 DeathStarBench as Validation Surface

DeathStarBench was selected over purpose-built synthetic benchmarks because its service graph reflects the structural properties of production enterprise microservice deployments [5]. The Social Network application profile presents 27 tightly coupled microservices implemented in a polyglot stack of C++, Go, Python, and NodeJS, backed by heterogeneous stateful storage including Redis clusters, MongoDB shards, and Memcached layers.

Table 2 (below) — DeathStarBench Social Network benchmark profile summary (Author's experimental configuration)

Property	Detail
Application Profile	Social Network
Total Microservices	27
Implementation Languages	C++, Go, Python, NodeJS
Stateful Storage Backends	Redis Cluster, MongoDB Shards, Memcached
Blast Radius Depth Tested	R = 1 (direct upstream + downstream)

License	GPL (publicly available)

The depth and variety of the dependency graph — including upstream API gateway chains, synchronous RPC fan-outs, and asynchronous event flows — is precisely what makes blast radius isolation algorithmically meaningful. A two-service test application would provide no meaningful evaluation of the DAG computation or mock routing logic [5][21].

4.3 Simulated Developer Workload

A synthetic commit generator modeled a medium-to-large distributed engineering organization. 150 active engineers committed code concurrently under a Poisson arrival distribution peaking during simulated mid-day business hours and dropping to low single digits during overnight cycles. Over the 24-hour epoch, the pipeline processed 420 discrete pull requests, each targeting a randomly selected service with mutations ranging from front-end template changes to modifications of deep internal graph-processing nodes. Each pull request triggered an automated end-to-end integration testing suite of 500 deterministic API calls covering principal user interaction paths, payload schema validation, and error boundary behavior [5].

This workload profile produced realistic utilization curves for both provisioning models: the static staging baseline maintained constant capacity throughout the epoch, while the ephemeral models experienced bursty provisioning activity correlated with the Poisson arrival peaks. The contrast between the utilization curves — flat for static, bursty for ephemeral — is directly reflected in the CPU utilization and cost figures reported in Section 5.

5. Results and Discussion

5.1 Cloud Infrastructure Cost

Cost tracking over the 24-hour evaluation window produced the three-way comparison shown in Table 3. The static staging baseline maintained five fully provisioned instances of the 27-service DeathStarBench suite around the clock. The naive ephemeral model provisioned all 27 services for every incoming pull request. The proposed DAG-based framework constrained provisioning to the blast radius sub-graph at depth R = 1, yielding a typical provisioned footprint of 3–5 services per pull request.

Table 3 (below) — Three-way infrastructure cost comparison over 24-hour evaluation window (Author's primary experimental data)

Metric	Static Staging Baseline	Naive Ephemeral Model	Proposed DAG Framework
Total Compute Hours	3,240.00 hrs	1,890.00 hrs	491.40 hrs
Avg. Cluster CPU Utilization	14.2%	48.6%	81.3%
Total Infrastructure Spend	$284.10	$198.50	$73.80
Capital Savings vs. Baseline	0.00% (Reference)	30.13%	74.02%

Total infrastructure spend reached $284.10 under the static baseline, $198.50 under the naive ephemeral model, and $73.80 under the proposed framework — representing capital savings of 74.02% against the static baseline and 62.82% against the naive ephemeral model. Average cluster CPU utilization of 14.2% under static staging confirms that the majority of provisioned capacity was idle throughout the evaluation window. The DAG-based framework achieved 81.3% average utilization, meaning nearly all provisioned capacity executed active test work at any given moment [6].

The efficiency gap between the naive and DAG-based ephemeral models — $198.50 versus $73.80 — is the most analytically significant finding. Spinning services up and down without restricting which services are provisioned captures only approximately 30% of the available savings. The

blast radius constraint, not the ephemeral lifecycle, drives the remaining 44 percentage points of cost reduction. This confirms the core claim of the work: the critical missing layer in CI cost optimization is dependency reasoning, not lifecycle management.

5.2 Pipeline Latency

Developer adoption depends on latency characteristics as much as cost. A system that saves money by making CI slower trades one bottleneck for another. Table 4 presents wall-clock latency measurements — from code push to integration test suite completion — as total cluster service count scaled from 5 to 35 services.

Table 4 (below) — Pipeline wall-clock latency vs. total cluster service count (Author's primary experimental data)

Total Cluster Service Count (N)	Naive Ephemeral Latency (min)	DAG-Based Framework Latency (min)
5	~2.1	<4
10	~3.8	<4
15	~5.4	<4
20	~6.9	<4
25	~8.2	<4
30	~9.6	<4
35	>10.0	<4

The naive ephemeral model's latency grew from approximately 2.1 minutes at 5 services to over 10 minutes at 35 services, driven by container scheduling contention, image pull cascades, and DNS propagation delays accumulating across the full service manifest. The DAG-based framework's latency held flat below four minutes throughout the scaling range, because the blast radius constraint bounds the provisioned footprint at 3–5 services regardless of total catalog size [5][8]. This flatness is the framework's defining operational property: pipeline latency becomes a function of blast radius depth, not service catalog size. For organizations scaling from tens to hundreds of microservices, this predictability is as operationally valuable as the cost savings.

5.3 Database Schema Migration Handling

Schema migration changes expose a structural challenge for mocking-based isolation. When a service modifies its relational or document store schema, a generic mock response engine cannot exercise the schema-dependent code paths being validated. The framework addresses this through automated detection of migration file changes in the Git diff payload — matching paths under recognized migration tool conventions for Flyway, Liquibase, and raw SQL migration directories.

When migration files are detected, the mock routing for the affected data node is overridden. A dedicated single-pod ephemeral database container is provisioned, initialized with the production schema baseline, and the proposed migration scripts are applied before the mutated service connects. This override path introduces an additional 45–90 second

initialization penalty relative to fully mocked environments. The trade-off is deliberate: a false-positive test pass against a mock that cannot validate schema behavior, or a full stateful backend provisioning that eliminates cost advantages entirely, are both worse outcomes than a bounded latency penalty on a minority of pull requests [15].

5.4 Limitations and Failure Mode Analysis

Two failure modes with direct operational consequences require explicit documentation. Static dependency graph extraction produces incorrect blast radius calculations when services use runtime-configured RPC endpoints, dynamic service discovery patterns, or reflection-based protocol routing. In these cases, hidden communication edges are absent from the DAG. The mock layer routes traffic for those edges to a generic response engine rather than the correct service, producing false-pass integration test results — the pull request is marked green while a real dependency contract violation exists [14].

The second failure mode is mock drift. When API contract definitions held in the mock daemon's index fall out of synchronization with actual service interfaces — because upstream teams publish schema changes without updating the registry — the mock returns responses that are structurally valid against an outdated schema. Services validating against the current contract will fail in production on calls that passed cleanly in the ephemeral environment. Both failure modes converge on a common architectural need: a dynamic, runtime-instrumented dependency and contract discovery layer to replace the current static analysis baseline [2][13].

6. Conclusion

6.1 Future Research Directions

The two documented failure modes define the immediate research roadmap. Replacing static graph extraction with trace-driven topology mapping — consuming OpenTelemetry spans from production clusters via a streaming pipeline — would ground the DAG in observed runtime behavior rather than inferred static structure, eliminating the hidden-edge failure mode [2][14]. For mock fidelity, automated contract synchronization between the daemon's index and live service deployments could be implemented through schema registry integration: Confluent Schema Registry for Kafka-based async dependencies, Buf for Protobuf-defined gRPC services. These extensions would transform the framework from a static-analysis-bounded tool into a continuously self-updating infrastructure primitive.

A third research direction, not directly motivated by a failure mode but identified during the evaluation, is predictive environment provisioning. Machine learning models trained on developer commit frequencies, branch creation patterns, and cross-team dependency histories could warm spot-instance node pools and pre-stage service images before pull requests are formally submitted, reducing observed initialization latency from the current 2–4 minute range toward sub-minute thresholds [16].

6.2 Conclusion

Distributed microservice CI pipelines cannot scale economically on staging paradigms designed for monolithic deployment models. Static staging accumulates idle infrastructure cost continuously; naive ephemeral provisioning replaces idle cost with per-PR provisioning cost that scales with service catalog size rather than with change scope. The framework presented in this paper resolves both failure modes by introducing structural dependency awareness at the container orchestration tier.

Automated DAG construction from static analysis, blast-radius-constrained sub-graph provisioning, and serverless spot-instance execution combine to produce integration test environments that cost 74.02% less than static staging while delivering per-pull-request isolation and sub-four-minute pipeline latency at scale. Total infrastructure spend of $73.80 across 420 pull requests — against $284.10 for the static baseline — demonstrates that the economic case for per-PR integration testing is not merely theoretical. For engineering organizations operating at the scale where CI infrastructure costs constitute a meaningful budget line, dependency-aware orchestration offers a path to continuous integration discipline that does not require choosing between test coverage and infrastructure economics.

Acknowledgments

The author acknowledges the open-source contributions of the DeathStarBench project at Cornell University, whose Social Network application profile served as the primary evaluation benchmark for this work.

Author Contributions

Gangadhar Chalapaka conceived the framework architecture, designed the blast radius algorithm,

implemented the Kubernetes Operator, constructed the experimental testbed, and conducted all evaluations reported in this work.

Conflicts of Interest

The author declares no conflict of interest.

References

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in Present and Ulterior Software Engineering, M. Mazzara and B. Meyer, Eds. Cham, Switzerland: Springer, 2017, pp. 195–216. Available: https://doi.org/10.1007/978-3-319-67425-4_12

[2] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," IEEE Access, vol. 5, pp. 3909–3943, 2017. Available: https://doi.org/10.1109/ACCESS.2017.2685629

[3] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: A systematic mapping study," Proceedings of the 8th International Conference on Cloud Computing and Services Science CLOSER, Funchal, Portugal, 2018, pp. 221–232. Available: https://doi.org/10.5220/0006798302210232

[4] C. Kroiß and T. Bureš, "Logic-based modeling of information transfer in cyber–physical multi-agent systems," Future Generation Computer Systems, vol. 56, pp. 317–332, 2016. Available: https://doi.org/10.1016/j.future.2015.09.013

[5] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken and B. Jackson, "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Providence, RI, USA, 2019, pp. 3–18. Available: https://doi.org/10.1145/3297858.3304013

[6] S. G. Domanal and G. R. M. Reddy, "An efficient cost optimized scheduling for spot instances in heterogeneous cloud environment," Future Generation Computer Systems, vol. 84, pp. 11–21, 2018. Available: https://doi.org/10.1016/j.future.2018.02.003

[7] L. Zhu, L. Bass, and G. Champlin-Scharff, "DevOps and its practices," IEEE Software, vol. 33, no. 3, pp. 32–34, May/Jun. 2016. Available: https://doi.org/10.1109/MS.2016.81

[8] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A Kubernetes controller for managing the availability of elastic microservice based stateful applications," Journal of Systems and Software, vol. 175, art. 110924, 2021. Available: https://www.sciencedirect.com/science/article/abs/pii/S0164121221000212

[9] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in Proceedings of the 6th International Conference on Cloud Computing and Services Science, Rome, Italy, 2016, pp. 137–146. Available: https://doi.org/10.5220/0005785501370146

[10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," Communications of the ACM, vol. 53, no. 4, pp. 50–58, 2010. Available: https://doi.org/10.1145/1721654.1721672

[11] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," ACM Queue, vol. 14, no. 1, pp. 70–93, 2016. Available: https://doi.org/10.1145/2898442.2898444

[12] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," IEEE Transactions on Cloud Computing, vol. 7, no. 3, pp. 677–692, 2019. Available: https://doi.org/10.1109/TCC.2017.2702586

[13] M. R. S. Sedghpour, C. Klein, and J. Tordsson, "An empirical study of service mesh traffic management policies for microservices," in ICPE '22: Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering, Beijing, China, 2022, pp. 17–27. Available: https://doi.org/10.1145/3489525.3511686

[14] S. D. Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Within-project defect prediction of infrastructure-as-code using product and process metrics," IEEE Transactions on Software Engineering, vol. 48, no. 6, pp. 2086–2104, 2022. Available: https://doi.org/10.1109/TSE.2021.3051492

[15] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," Concurrency and Computation: Practice and Experience, vol. 32, no. 17, e5668, 2020. Available: https://doi.org/10.1002/cpe.5668

[16] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. How bad is it

really?" Empirical Software Engineering, vol. 24, no. 4, pp. 2469–2508, 2019. Available: https://doi.org/10.1007/s10664-019-09681-1
[17] A. S. Abdelfattah and T. Cerny, "The Microservice Dependency Matrix," in European Conference on Service-Oriented and Cloud Computing, 2023, pp. 285–300. Available: https://link.springer.com/chapter/10.1007/978-3-031-46235-1_19
[18] I. Ayala, A. V. Papadopoulos, M. Amor, L. Fuentes, "ProDSPL: Proactive self-adaptation based on Dynamic Software Product Lines," Journal of Systems and Software, vol. 175, art. 110909, 2021. Available: https://doi.org/10.1016/j.jss.2021.110909
[19] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," ACM SIGAPP Applied Computing Review, vol. 17, no. 4, pp. 29–45, 2018. Available: https://doi.org/10.1145/3183628.3183631
[20] A. Brito, A. Hora and M. T. Valente, "Refactoring Graphs: Assessing Refactoring over Time" 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 2020, pp. 501–511. Available: https://doi.org/10.1109/SANER48275.2020.9054864
[21] N. Rajovic, P. M. Carpenter, I. Gelado, N. Puzovic, A. Ramirez and M. Valero, "Supercomputing with commodity CPUs: are mobile SoCs ready for HPC?," SC '13: Proceedings of the International Conference on High Performance Computing, Denver, CO, USA, 2013, art. 39. Available: https://doi.org/10.1145/2503210.2503281