

Intelligent Runtime Adaptation of Cloud-Native Microservices through Design-Principle-Aware Resource Management

Syed Fakhar Abbas

38276@students.riphah.edu.pk

Riphah International University

Musharif Ahmed

Riphah International University

Nabil Almashfi

Jouf University

Asim Bakhshi

National University of Sciences and Technology

Sajjad Aamir

National University of Sciences and Technology

Mamoona Humayun

University of Roehampton

Maram Fahaad Almufareh

Jouf University

Research Article

Keywords: Cloud-native microservices, Kubernetes autoscaling, Runtime adaptation, Resource management, Reinforcement learning, Performance optimization

Posted Date: October 13th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10516053/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Intelligent Runtime Adaptation of Cloud-Native Microservices through Design-Principle-Aware Resource Management

Syed Fakhar Abbas^{1*}, Musharif Ahmed¹, Nabil Almashfi^{2*},
Asim Bakhshi³, Sajjad Aamir³, Mamoonah Humayun⁴,
Maram Fahaad Almufareh⁵

^{1*} Department of Computer Science, Riphah International University,
Rawalpindi, Pakistan .

² Department of Software Engineering, College of Computer and
Information Sciences, Jouf University, Sakaka, Saudi Arabia .

³ Department of Computer Science, National University of Sciences and
Technology, Islamabad, Pakistan .

⁴ Department of Computing, School of Arts, Humanities and Social
Sciences, University of Roehampton, London, United Kingdom .

⁵ Department of Information Systems, College of Computer and
Information Sciences, Jouf University, Sakaka, Saudi Arabia .

*Corresponding author(s). E-mail(s): 38276@students.riphah.edu.pk;
nalmashfi@ju.edu.sa;

Contributing authors: musharraf.ahmed@riphah.edu.pk;
asim.dilawar@mcs.edu.pk; saamir.mscs24mcs@student.nust.edu.pk;
mamoonah.humayun@roehampton.ac.uk; mfalmufareh@ju.edu.sa;

Abstract

Cloud-native microservice systems are predominantly managed through reactive autoscaling that responds only after performance has degraded. Although orchestration platforms such as Kubernetes provide elastic resource provisioning, they seldom account for architectural design principles, namely coupling, cohesion, and service granularity, whose effects become pronounced under fluctuating workloads. We present an empirical extension of the Adaptive Microservices Optimization Framework (AMOF), a hybrid learning-based runtime controller for Kubernetes that augments performance telemetry with design-principle-aware proxies. AMOF

31 couples a Random Forest classifier, used as a forward-looking degradation detector,
32 with a Deep Q-Network that selects adaptation actions under a multi-objective
33 reward balancing CPU utilization, memory consumption, latency, and adaptation
34 cost. The framework is evaluated through a controlled multi-case study on two
35 structurally distinct benchmark applications, Google Online Boutique and Weave-
36 works Sock Shop, spanning four adaptation strategies, three workload regimes,
37 and five independent runs, yielding 120 experiments and 89,901 telemetry obser-
38 vations. The detector anticipates degradation thirty seconds ahead with an area
39 under the receiver operating characteristic curve (AUC) above 0.97 on unseen
40 stress and extreme workloads, and the design-principle proxies supply most of its
41 predictive importance. Relative to the Kubernetes Horizontal Pod Autoscaler,
42 AMOF lowers CPU utilization by up to 52.2% and memory usage by up to 49.1%,
43 while exposing a resource-efficiency versus latency trade-off under extreme load.
44 Cohesion proves more amenable to runtime adjustment than coupling, which
45 remains constrained by structural service dependencies. These findings indicate
46 that design-principle-aware telemetry can complement conventional autoscaling
47 and support more informed runtime management of cloud-native deployments.

48 **Keywords:** Cloud-native microservices, Kubernetes autoscaling, Runtime adaptation,
49 Resource management, Reinforcement learning, Performance optimization

50 1 Introduction

51 Cloud-native computing has transformed the design and operation of modern dis-
52 tributed applications by promoting containerization, service decomposition, automated
53 deployment, and elastic resource provisioning [30]. Within this paradigm, microser-
54 vice architecture has become widely adopted for building scalable and independently
55 deployable cloud applications [24]. By decomposing complex systems into smaller ser-
56 vices with decentralized ownership and deployment pipelines, microservices enable
57 operational flexibility, continuous delivery, and fine-grained resource management in
58 cloud environments [12].

59 Loose coupling, high cohesion, and precise granularity are fundamental design
60 principles in microservices systems [10, 11] and have been thoroughly studied at design
61 time [8, 37]; however, their collective dynamic adaptation under varying workloads
62 remains underexplored in operational environments [18].

63 Microservices management strategies are predominantly reactive [42, 3]. Resource
64 scaling mechanisms and tools, such as Kubernetes’ Horizontal Pod Autoscaler (HPA),
65 respond only after performance degradation occurs [3], resulting in Service-Level
66 Agreement (SLA) violations and over-allocation of infrastructure resources [46]. Early
67 work on predictive autoscaling showed that reactive policies introduce a provisioning
68 lag that grows with workload intensity, and that incorporating workload forecasting
69 into the scaling loop reduced Service-Level Objective (SLO) violations by roughly
70 25% [5]. Despite the wide adoption of industry-standard observability tools such as
71 Prometheus and Grafana for analysis and monitoring [41, 9], actionable architectural
72 adjustment based on monitoring data has remained limited [14].

Recent studies have explored machine learning for predictive autoscaling [40, 22], however, most approaches focused on resource metrics like CPU utilization or memory usage in isolation. The literature identifies reactive, proactive, and hybrid scaling as three successive generations, with hybrid methods achieving better resource efficiency under heavy workloads [7, 4]. Moreover, architectural principles such as service granularity, cohesion and coupling were rarely incorporated into runtime adaptation strategies, leaving a gap between software design modeling and runtime behavior of microservices system [18, 19]. Empirical evidence indicates that coupling metrics correlate directly with independence as well as inter-service call volume [23], reinforcing the need for a framework that treats design principles as runtime orchestrators rather than static design-time artifacts.

This study extends our previously introduced **Adaptive Microservices Optimization Framework (AMOF)** [2, 1] by moving from initial framework validation toward a broader cloud-native empirical evaluation. The earlier work established the conceptual basis of design-principle-aware runtime adaptation and demonstrated its feasibility on a single benchmark. The present study evaluates AMOF across two structurally distinct Kubernetes microservice applications, employs a larger experimental matrix, compares against multiple cloud autoscaling baselines, and analyses the extent to which design-principle-aware runtime proxies contribute to cloud resource management under normal, stress, and extreme workload regimes. To support independent verification and extension, the complete AMOF implementation, experiment orchestration scripts, case-study configurations, logs, and statistical analysis code are made openly available as a replication package [1].

Contributions. This paper makes four contributions. First, it extends AMOF to cloud-native Kubernetes microservices by combining runtime telemetry with proxies for coupling, cohesion, and granularity, and by coupling a Random Forest forward-looking degradation detector with a Deep Q-Network that selects adaptation actions under a multi-objective reward over CPU, memory, latency, and adaptation cost. Second, it reports a controlled multi-case evaluation on Google Online Boutique and Weaveworks Sock Shop, two architecturally distinct benchmarks deployed on Kubernetes with Istio-based observability, comprising 120 experiments and 89,901 telemetry observations against Static, Horizontal Pod Autoscaler, and Rule-Based baselines. Third, it quantifies the resulting resource versus Quality-of-Service (QoS) trade-off, showing that AMOF improves CPU and memory efficiency relative to reactive autoscaling at the cost of a workload-dependent latency penalty under extreme load. Fourth, it examines the runtime tractability of architectural parameters and finds that cohesion responds to short-horizon control whereas coupling remains structurally constrained, a distinction further supported by the detector’s reliance on design-principle features.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the AMOF runtime adaptation framework and its design-principle-aware telemetry model. Section 4 describes the Kubernetes experimental environment, benchmark applications, workloads, baselines, metrics, and statistical procedures. Section 5 presents the empirical results and their analysis. Section 6 discusses theoretical implications and practical considerations, and Section 8 concludes the work with future research directions.

2 Related Work

2.1 Runtime Adaptation in Microservices

The wide adoption of microservice architecture (MSA), driven by the architectural principles of service granularity, loose coupling, and high cohesion, has attracted sustained research attention because of the advantages it offers, including independence, scalability, and flexibility, particularly in dynamic cloud environments [20]. The proposed approaches, however, have mainly focused on the reactive scaling of resources without collectively incorporating architectural design principles. Foundational literature established that research on MSA is strongly skewed toward design-time measures like Domain-Driven Design (DDD) [43] or Model-Driven Development (MDD) [50], with runtime reconfiguration remaining underexplored relative to its operational importance [12]. Hassan et al. [18] highlighted granularity adaptation at runtime; Vural & Koyuncu [43] confined the contribution of coupling, cohesion, and granularity to static analysis or design-time decisions.

Recent advances have incorporated Machine Learning (ML) algorithms for microservices optimization. Trabelsi et al. [39] performed a systematic review of ML-based migration of monolithic systems to microservices, identifying coupling and cohesion as key adaptation targets. Similarly, Narváez et al. [29] investigated deep learning techniques to optimize microservice boundaries. These studies supported the argument that runtime adaptation can be enhanced by utilizing AI models trained on design metrics.

A persistent limitation of reactive strategies is the provisioning lag they introduce during sudden workload surges. Xie et al. [49] addressed this by proposing PBScaler, a bottleneck-aware autoscaling framework that uses the TopoRank algorithm to contextualize performance bottlenecks across service interaction graphs before invoking scaling decisions. The Random Forest-based SLO-violation predictor in their approach achieved a precision of 0.92 on real-world Kubernetes deployments. Ahmad et al. [6] extended horizontal pod autoscaling with hierarchical resource sharing across microservices, demonstrating that proactive, MAPE-K-driven policies outperform both reactive and Kubernetes baseline autoscalers in resource-constrained environments. These studies reinforce the need for a framework that combines predictive classification with policy-level control instead of relying on threshold triggers alone.

2.2 Machine Learning and Reinforcement Learning for Systems Optimization

Reinforcement learning (RL) has been applied for runtime improvement of microservices systems, including predictive scalability and resource allocation, where controllers must make sequential decisions under changing conditions [26, 38]. Al-Dhuraibi et al. [7] distinguished reactive, proactive, and hybrid mechanisms, and determined the mitigation of provisioning lag as a principal motivation for predictive control. Similarly, Abdullah et al. [5] showed that trace-driven predictive autoscaling can reduce SLO violations in fog micro-data centers by learning from reactive policy traces. Abdullah et al. [4] further extended their work to fluctuating workloads of containerized microservice and reported substantial gains over reactive baselines under non-stationary demand.

Wang et al. [44] and Kaloudis [21] pointed out the value of telemetry-driven learning for runtime decision making.

More recent work has moved from prediction only towards learned control policies. DeepScaling [46] employed neural models to stabilize CPU utilization in large-scale production cloud systems while limiting over-provisioning. Similarly, gym-hpa [34] formulated Kubernetes autoscaling as a reinforcement-learning problem for complex microservice systems. Both these studies showed that learned controllers can improve resource management in dynamic environments. At the same time, most existing approaches remain centered on performance and resource metrics. The literature has so far given comparatively little attention to integrating predictive degradation detection with reinforcement-learning-based control.

2.3 Architectural Principles at Runtime

Coupling, cohesion, and granularity have remained fundamental concepts in MSA [32], yet they were treated predominantly as design-time attributes. Although the literature provided different ways to quantify these attributes, their formalization at runtime needs further investigation [18]. Lenarduzzi et al. [23] highlighted the association of coupling-related measures with increased density and the volume of inter-service communication observed during migration to microservices, suggesting that these properties have operational significance in addition to architectural meaning. Related studies by Fritzsche [13] and Mazlami et al. [27] explored monitoring and restructuring decisions with reference to architectural quality, but their focus was primarily on refactoring and decomposition rather than runtime control.

Zhong et al. [50] provided empirical support for interaction-based measures, showing that inter-service call rate and dependency ratio can serve as practical runtime proxies for structural coupling in microservice systems. Sellami and Saied [36] further elaborated that deep reinforcement learning can support service decomposition decisions, thereby reinforcing the broader case for learning-based architectural optimization. Their work, however, was concerned with design-time decomposition rather than runtime reconfiguration.

2.4 Observability in Cloud-Native Systems

Giamattei et al. [14] reported, in a Grey-literature review (GLR) of DevOps monitoring tools, that Prometheus and Grafana have become the prevailing observability stack for microservice environments. This observation was consistent with the systematic mapping study by Gomes et al. [16], which found metrics-based monitoring to be the dominant approach in the literature.

Liang et al. [25] provide a recent example of learning-based autoscaling built on the Prometheus and Kubernetes pipeline. Their framework combined a dense-connectivity spatio-temporal graph neural network with Q-learning and showed that richer spatio-temporal representations improve degradation forecasting accuracy over simpler baselines. Although their focus is workload prediction and autoscaling rather than architectural control, the study was relevant in two aspects. Firstly, it confirmed the practical value of Prometheus-derived telemetry as an input to learned controllers.

202 Secondly, it supported the broader premise that reinforcement-learning-based control
 203 can be effective in Kubernetes-based microservice environments.

204 2.5 Positioning of This Work

205 As shown in Table 1, most prior works targeted the improvement of resource usage or
 206 code modularity in isolation. AMOF differs by introducing a hybrid architecture that
 207 integrates predictive temporal forecasting with reinforcement-based policy optimization
 208 to actively tune architectural configurations in response to telemetry insights. Unlike
 209 prior studies, AMOF provides empirical validation across two distinct benchmarks
 210 on a single-node Kubernetes cluster using calibrated workloads. The incorporation
 211 of design-principle (coupling, cohesion, granularity) into the DQN state vector sets
 212 AMOF apart from performance-only autoscaling approaches like PBScaler [49] and
 213 DeepScaling [46], which did not account for architectural quality during adaptation.

214 3 AMOF: Design-Principle-Aware Runtime 215 Adaptation of Microservices

216 3.1 Framework Overview

217 AMOF is organized into three stages [2], shown in Figure 1. Stage 1 covers design
 218 and deployment initialization, real-time monitoring, trade-off analysis, and decision
 219 triggering. Stage 2 contains a hybrid AI decision engine in which a Random Forest
 220 Classifier produces an adaptation flag and a Deep Q-Network selects the next control
 221 action. Stage 3 performs post-adaptation evaluation and continuous optimization
 222 through monitoring feedback and model updates.

223 Our work distinguishes between the framework’s generic action families and the
 224 concrete action space used in the experiments. In the framework diagram, the policy
 225 layer is represented by generic actions such as SPLIT, MERGE, RECONFIGURE, and
 226 DECOUPLE. In the case-study instantiation reported here, these are operationalized as
 227 SPLIT, MERGE, RECONFCOUPLING, and ADJUSTCOHESION, matching the controller
 228 configuration used throughout the experiments. At this level, the actions denote runtime
 229 controller decisions over deployment and interaction behavior rather than source-code
 230 refactorings.

231 3.2 Runtime State Representation

232 Let $s_t \in \mathbb{R}^6$ denote the AMOF state at time t for a service instance:

$$s_t = \langle \widehat{cpu}_t, \widehat{mem}_t, \widehat{lat}_t, \widehat{cpl}_t, \widehat{coh}_t, \widehat{gran}_t \rangle. \quad (1)$$

233 The first three components capture container-level performance signals (CPU uti-
 234 lization, memory utilization, and request latency), collected at fixed intervals from
 235 cluster telemetry. The last three components operationalize design-principle proxies
 236 derived from service-mesh observations and deployment state: (i) coupling as a normal-
 237 ized inter-service call-rate signal, (ii) cohesion as an internal-versus-external work ratio

Table 1: Comparative analysis of related studies (2017–2025).

Study	Runtime Adaptation	ML/AI Integration	Design Principles	Production Validation	Year	Focus Area	Key Limitations
<i>Foundational & Theoretical Work (2017–2020)</i>							
Bogner et al. [8]	No	No	Coupling only	No	2017	Metrics	Limited to static code analysis; no runtime consideration
Mazlami et al. [27]	No	No	Granularity only	No	2017	Extraction	Static analysis for monolith decomposition; no adaptation
Jamshidi et al. [20]	No	No	Conceptual	No	2018	MSA Evolution	Theoretical framework without empirical validation
Hassan et al. [18]	Limited	No	Granularity only	No	2020	Mapping	Design-time focus; lacks runtime reconfiguration
<i>Machine Learning-Augmented Approaches (2020–2023)</i>							
Hassan et al. [19]	Yes	Statistical	Granularity only	Limited	2022	Adaptation	Addresses single principle; no hybrid ML
Rubak et al. [33]	Yes	Gradient Boosting	No	Cloud	2023	Prediction	Architecture-agnostic; reactive post-degradation
Wang et al. [44]	Partial	Deep Learning	Coupling only	SLR	2023	Governance	Literature review; limited empirical validation
Santos et al. [34]	Yes	RL (Q-learning)	No	Kubernetes	2023	RL Autoscaling	No design-principle integration; resource-only reward
<i>Advanced AI & Reinforcement Learning (2024–2025)</i>							
Fritzsche [13]	Yes	Partial ML	All three	Quality-driven	2024	Refactoring	Manual quality assessment; no predictive capability
Kaloudis [21]	Yes	Statistical + ML	Coupling/Cohesion	Case studies	2024	Resilience	Limited to statistical methods; no RL integration
Xie et al. [49]	Yes	Random Forest	No	Kubernetes	2024	Bottleneck Scaling	RF used for SLO prediction only; no RL or design principles
Sellami & Saied [36]	Design-time	Deep RL	All three	No	2025	Extraction	No runtime application; design-time extraction only
Narváez et al. [29]	Partial	Multiple AI	Conceptual	Review	2025	AI Design	Systematic review; limited empirical validation
Trabelsi et al. [39]	No	ML Overview	Coupling/Granularity	Literature	2025	Migration	Review paper; no runtime adaptation framework
Ahmad et al. [6]	Yes	Hierarchical HPA	No	Kubernetes	2025	Proactive Scaling	No design-principle state; reactive/proactive only
This work (AMOF)	Full	RFC + DQN	All three	Kubernetes	2026	Predictive	Single-node cluster; two benchmark apps

Runtime Adaptation: Full = continuous, policy-driven reconfiguration execution; Partial = semi-automated orchestration; Limited = threshold-triggered response that requires manual validation; No = design-time analysis only.

ML/AI Integration: RFC = Random Forest Classifier; DQN = Deep Q-Network; RL = Reinforcement Learning; HPA = Horizontal Pod Autoscaler; Partial ML = ML-assisted analysis with manual refactoring steps; Statistical = regression or rule-induction without learned policy; Hierarchical HPA = layered rule-based pod sharing without RL.

Design Principles: “All three” denotes coverage of coupling, cohesion, and granularity; Conceptual = principles discussed as design goals only.

Production Validation: Kubernetes = evaluated on a production-grade Kubernetes deployment; Quality-driven = validated against software quality assessments.

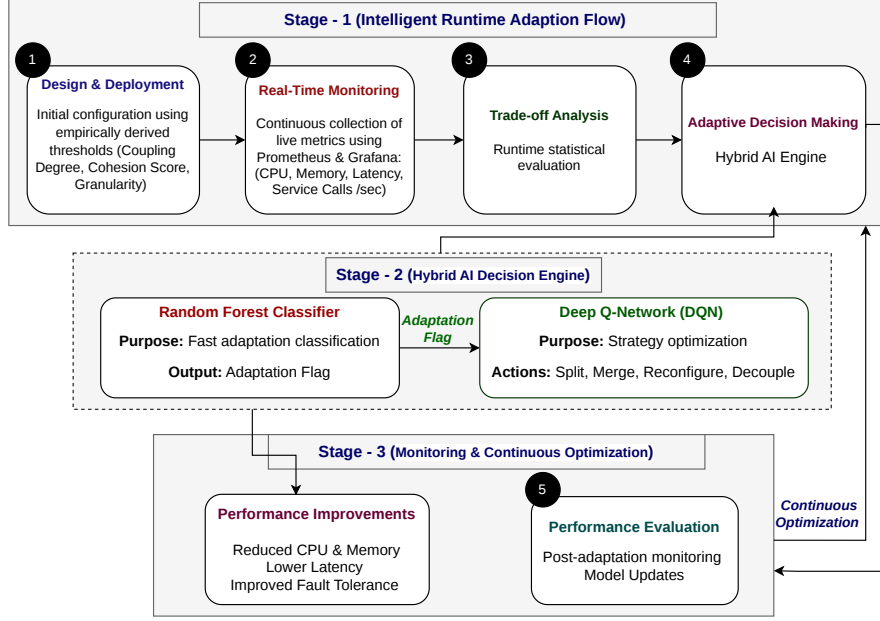


Fig. 1: Overview of the Adaptive Microservices Optimization Framework (AMOF).

proxy, and (iii) granularity as a normalized dependency ratio. Such runtime proxies are consistent with the existing literature [50] and further explained in Section 3.3.

3.3 Design-principle-aware runtime proxies

AMOF uses design-time principles (coupling, cohesion, and granularity) as runtime indicators of performance and quality, alongside CPU, memory, and latency, reflecting the flexibility and maturity of cloud-native microservice systems under fluctuating loads. Let $\mathcal{S}$ denote the set of services, $s \in \mathcal{S}$ a service instance, and t a monitoring interval. We write $c_s^{\text{in}}(t)$ and $c_s^{\text{out}}(t)$ for the inbound and outbound inter-service call rates of service s , $w_s^{\text{int}}(t)$ for its internally served request work, $c_s^{\text{ext}}(t)$ for its outbound (externally forwarded) call volume, and $D_s(t)$ for the number of distinct services it depends on. The coupling proxy estimates the relative communication pressure associated with service s :

$$\text{Cpl}_s(t) = \frac{c_s^{\text{in}}(t) + c_s^{\text{out}}(t)}{\sum_{j \in \mathcal{S}} (c_j^{\text{in}}(t) + c_j^{\text{out}}(t))}. \quad (2)$$

Cohesion estimates the degree to which the observed work of a service is handled internally rather than through external service interactions:

$$\text{Coh}_s(t) = \frac{w_s^{\text{int}}(t)}{w_s^{\text{int}}(t) + c_s^{\text{ext}}(t)}. \quad (3)$$

Granularity estimates dependency density relative to the size of the deployed service graph:

$$\text{Gran}_s(t) = \frac{D_s(t)}{|\mathcal{S}| - 1}. \quad (4)$$

These parameters are not intended to replace source-level architectural metrics. Instead, they provide runtime-observable approximations of architectural pressure that can be incorporated into cloud-native adaptation decisions.

Normalization.

All six features are normalized to $[0, 1]$ for supporting cross-service comparability and stable learning dynamics:

$$\hat{x}_t = \frac{x_t - \min(x)}{\max(x) - \min(x)} \quad (\text{min-max over the experiment window, per service}). \quad (5)$$

3.4 Degradation Detection (RFC Gate)

AMOF uses a Random Forest Classifier (RFC) as a fast, forward-looking degradation detector that gates the reinforcement-learning decision engine. Rather than classifying the current instant, the detector *anticipates* degradation: it maps the state vector s_t to a binary label $\hat{y}_t \in \{0, 1\}$ that indicates whether the service will enter a degraded condition within a short prediction horizon H . A service is treated as degraded at a given interval when it violates any of the empirically selected thresholds (CPU > 80%, memory > 85%, P95 latency > 500 ms, or error rate > 5%), consistent with common performance-engineering practice in which latency and saturation act as primary early-warning indicators [14]. The training label for interval t is therefore the degradation state observed at interval $t + H$, computed independently per service and per run to prevent any leakage across run boundaries. We set $H = 30$ s (three samples at the 10-second scrape interval), which is short enough to remain actionable yet long enough to precede saturation. Defining the target prospectively, rather than as a concurrent function of the input features, ensures that the detector learns genuine precursors of degradation instead of re-deriving the labeling rule from the same-instant metrics. The RFC is trained on normal-load (W1) traces and evaluated on previously unseen stress (W2) and extreme (W3) workloads, reflecting the workload shifts encountered in operation. It uses 100 trees with bounded depth and balanced class weights (Table 5).

3.5 Action Selection using Deep Q-Learning

After degradation prediction by RFC, AMOF calls Deep Q-Network (DQN) in order to select an architectural control action a_t from action space:

$$a_t \in \{\text{SPLIT}, \text{MERGE}, \text{RECONFCOUPLING}, \text{ADJUSTCOHESION}\}. \quad (6)$$

Conceptually, SPLIT/MERGE aim for (fine or coarse) service granularity through runtime configuration, whereas RECONFCOUPLING and ADJUSTCOHESION adjust

Table 2: Mapping of AMOF adaptation actions to cloud-native control semantics

AMOF action	Cloud-native interpretation	Runtime realization	Boundary limitation
Split	Fine-grained execution or scaling mode	Increase service-level execution capacity or isolate overloaded service paths	Does not perform source-code decomposition
Merge	Consolidated execution mode	Reduce unnecessary runtime dispersion and stabilize low-load services	Does not merge code-bases or databases
Reconfigure coupling	Interaction-pressure mitigation	Adjust routing, traffic distribution, or dependency-related runtime behavior	Cannot remove mandatory business dependencies
Adjust cohesion	Workload-locality improvement	Reduce redundant external calls or favour localized service execution paths	Limited by existing service responsibilities

interaction patterns as well as internal/external work distribution. The proposed design aligns with empirical findings indicating that inter-service communication and dependency pressure are critical quality factors of microservice systems [50]. Table 2 maps each abstract action to the orchestration operations used during the case-study.

Reward Function.

AMOF optimizes a multi-objective reward that balances resource efficiency and QoS simultaneously:

$$R(s_t, a_t, s_{t+1}) = 0.4 \Delta \widehat{cpu} + 0.3 \Delta \widehat{mem} + 0.3 \Delta \widehat{lat} - \lambda, \quad (7)$$

where $\Delta \widehat{x} = \widehat{x}_t - \widehat{x}_{t+1}$, so that a positive reward indicates an improvement in system behavior through lower resource consumption and lower latency. The term λ denotes an adaptation penalty that discourages unnecessary or excessive reconfiguration. DQN training is carried out using experience replay and a target network, consistent with established practice in deep reinforcement learning [28]. Prior work on RL-based resource management has shown that learned policies can surpass hand-crafted heuristics in non-stationary scheduling and control environments [26]. This provides a clear rationale for AMOF’s reliance on learned action selection when workload conditions evolve over time. In both case studies the policy network training converged: the temporal-difference loss stabilized by approximately episode 200 and settled at a low residual value (0.0057 for Online Boutique and 0.0087 for Sock Shop), as shown in Figure 2.

3.6 AMOF Control Loop

AMOF operates as a persistent monitor–analyze–plan–execute cycle with continuous monitoring, illustrated in Figure 3. In each iteration, the controller first gathers runtime metrics, then obtains the state features for decision making, predicts degradation probability, selects an adaptation action when intervention becomes necessary, and

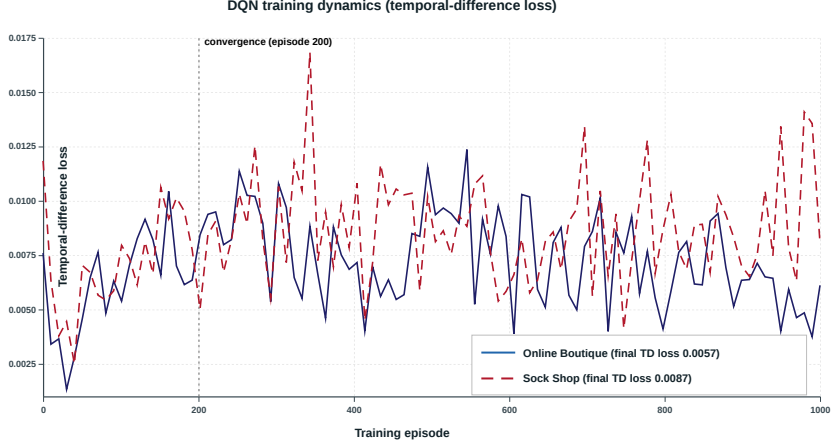


Fig. 2: Temporal-difference loss of the Deep Q-Network during training on both benchmarks. The loss stabilizes by approximately episode 200, indicating policy convergence.

finally applies that action using DevOps tools like Kubernetes and service-mesh primitives. This iterative structure enables AMOF to move systematically from observation to prediction and from planning to execution, while maintaining a clear separation between detection and adaptive response.

To keep the control process computationally manageable, AMOF restricts the number of adaptations permitted within a given control window. Moreover, the controller state is reset between independent experimental runs in order to avoid carry-over effects and preserve the validity of the evaluation design (details are shown in Section 4).

4 Empirical Study Design

4.1 Research Questions

The evaluation of our proposed work is structured around three research questions:

RQ1 (*Detection generality*). Does AMOF’s degradation detector generalize from normal-load observations to unseen stress and extreme workloads across distinct microservice systems?

RQ2 (*Runtime effectiveness*). Compared to established scaling baselines (Static, HPA, Rule-Based), how does AMOF affect CPU utilization, memory utilization, and request latency under increasing workload intensity?

RQ3 (*Design-principle outcomes*). Does AMOF measurably influence coupling, cohesion, and granularity proxies, and are the effects consistent across case studies?

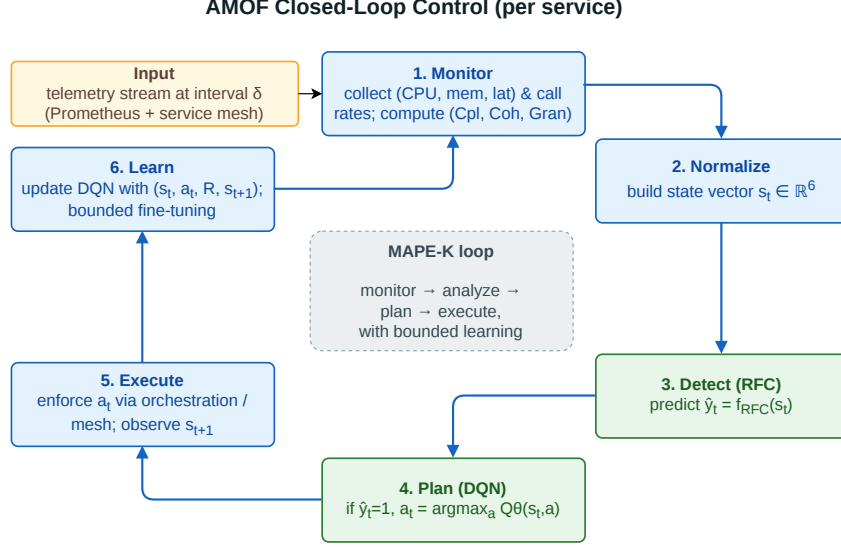


Fig. 3: AMOF closed-loop control on a per-service basis. Each monitoring interval runs a six-step MAPE-K cycle: collect telemetry and compute the design-principle proxies (Monitor), assemble the normalized state vector (Normalize), predict impending degradation with the Random Forest gate (Detect), select an adaptation with the Deep Q-Network when degradation is anticipated (Plan), apply it through orchestration and service-mesh controls (Execute), and perform bounded learning updates (Learn).

4.2 Experimental Environment and Subject Systems

All experiments were executed on a single bare-metal workstation to reduce inter-node variability and improve reproducibility. The host runs Ubuntu 22.04.5 with a single-node MicroK8s Kubernetes distribution, Istio service mesh, and a Prometheus–Grafana observability stack with a 10-second scrape interval (Table 3). AMOF is implemented in Python and uses scikit-learn for the RFC and PyTorch for the DQN.

Two independent case studies were used, as illustrated in the experimental design overview (Figure 4):

1. **Case Study 1: Google Online Boutique [17].** An e-commerce microservice benchmark comprising 11 polyglot services (10 scalable), deployed under namespace `gob`. Services communicate via gRPC with a gateway frontend. The application has been widely adopted in autoscaling and microservice research as a representative polyglot workload spanning Go, Python, Java, C#, and Node.js services [6, 34].
2. **Case Study 2: Weaveworks Sock Shop [47].** A microservice reference application with 14 services (8 scalable, plus datastores), deployed under namespace `sock-shop`, using HTTP/REST communication. Sock Shop is an established benchmark for evaluating container orchestration and autoscaling strategies, and its

Table 3: Platform and case-study characteristics (single-node Kubernetes).

Item	Configuration
Host hardware	Intel Core i7-10610U (4C/8T), 32 GB RAM, NVMe SSD
OS / Kernel	Ubuntu 22.04.5 LTS; Linux 6.8.0-94-generic
Orchestration	MicroK8s v1.32.9; containerd 1.6.36; Calico CNI
Service mesh	Istio 1.18.2 (Envoy sidecars injected per pod)
Monitoring	Prometheus + Grafana (scrape interval: 10 s)
ML stack	Python 3.10; scikit-learn 1.7.2; PyTorch 2.0.0
Case Study 1	Online Boutique (11 svcs; 10 scalable; ns <code>gob</code>)
Case Study 2	Sock Shop (14 svcs; 8 scalable; ns <code>sock-shop</code>)

REST-based topology offers a structurally distinct contrast to the gRPC stack of Online Boutique [49].

These systems represent distinct implementation stacks and service graphs, supporting the analytic generalization argument commonly expected in multi-case empirical research [45].

4.3 Baselines, Workloads, and Experiment Matrix

We compare AMOF against three widely used scaling strategies:

- **Static baseline:** fixed single replica per scalable service (no autoscaling).
- **Kubernetes HPA baseline:** CPU target 70% and memory target 80%, with 1–5 replicas.
- **Rule-based baseline:** threshold rules (e.g., scale up if CPU > 80%; scale down if CPU < 30%; scale up if memory > 85%).

Workloads were calibrated per application to obtain comparable stress regimes, acknowledging that distinct microservice systems have different resource footprints [31]. Each workload runs for 10 minutes after a brief ramp-up (Table 4). For each combination of approach and workload level, we performed 5 independent runs and reset the cluster to a 1-replica baseline between runs. Overall, the study comprises 120 experiments (60 per application), yielding 89,901 data points.

The five-run per-condition design follows the independent-replication principle for controlled experiments in empirical software engineering [48], whereby repeated trials under reset conditions bound the influence of scheduling non-determinism on outcome estimates. The resulting corpus of 89,901 telemetry observations, collected at a 10-second scrape interval across 120 experiments, is commensurate with (and in aggregate exceeds) the data volumes reported in comparable Kubernetes autoscaling studies [35].

Table 4: Workload configurations and experiment matrix (c = concurrent clients; rps = requests per second).

Workload	Description	Online Boutique (c/rps)	Sock Shop (c/rps)
W1 Normal	Normal operating load	100 / 50	200 / 100
W2 Stress	High-load stress test	300 / 150	400 / 200
W3 Extreme	Peak overload	500 / 300	600 / 300

Matrix: 4 approaches $\times$ 3 workloads $\times$ 5 runs = 60/app; 120 total.
Collection: 10 s interval; warmup: 60 s; cooldown: 30 s; gap: 120 s; reset: yes.

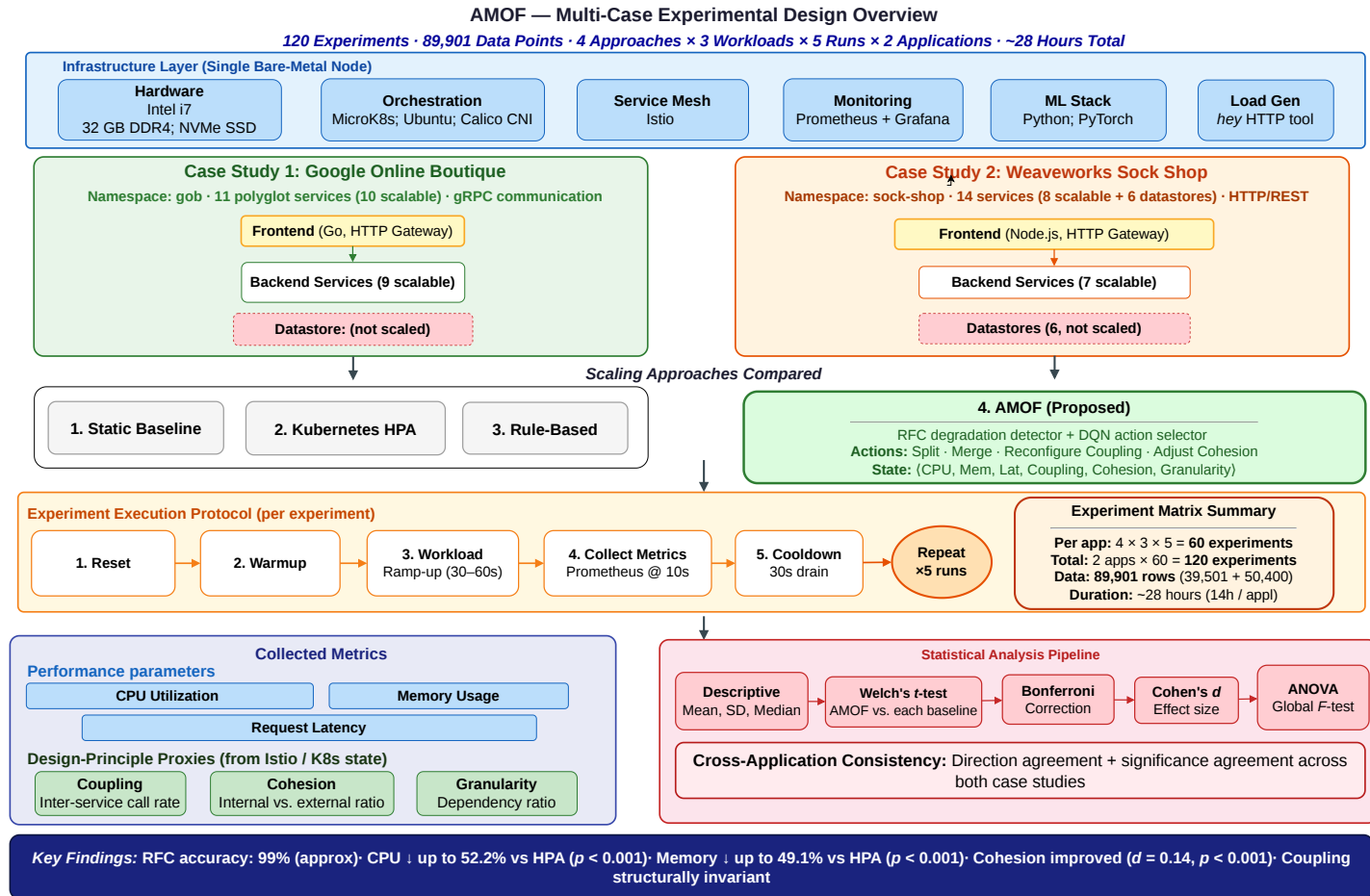


Fig. 4: Multiple case-study experimental design overview including infrastructure platform, details of both applications and workloads, four scaling approaches, per-experiment execution protocol and the statistical analysis pipeline.

Table 5: AMOF configuration (RFC + DQN) and metric definitions.

Component	Configuration
RFC (detector)	100 trees; max depth 10; min split/leaf 5/2; class weight balanced; 30 s forward-looking labels ($H = 3$ samples); trained on W1 baseline; tested on unseen W2+W3; degradation thresholds: CPU >80%, Mem >85%, P95 Lat >500 ms, Err >5%
DQN (policy)	Input(6)→128→64→Output(4); ReLU; dropout 0.2; $\gamma=0.95$; offline lr=0.001; online lr=0.0001; replay buffer 10k/5k; batch 32; max 1,000 episodes; target update every 10 episodes
Reward	$0.4 \Delta \widehat{cpu} + 0.3 \Delta \widehat{mem} + 0.3 \Delta \widehat{lat} - \lambda$; $\lambda=0.1$; CPU threshold 0.70; memory threshold 0.80; latency threshold 500 ms; max 5 adaptations per 300 s
Metrics	CPU/memory from cAdvisor; latency and request rates from Istio/Envoy; coupling from inter-service call rate; cohesion from internal vs. external work proxy; granularity from normalized dependency ratio

4.4 AMOF Configuration and Collected Metrics

Table 5 summarizes the AMOF configuration. The RFC is trained on normal-load baseline data and tested on unseen stress/extreme workloads. The DQN uses a 2-hidden-layer network and is trained offline with bounded episodes, then fine-tuned online with a reduced learning rate. To maintain controller stability, AMOF bounds the number of adaptations within a time window and includes an adaptation penalty in the reward (Eq. 7).

4.5 Statistical Analysis

To evaluate differences between AMOF and the comparison approaches, we employed both descriptive and inferential statistical analyses. Descriptive statistics are first used to summarize the central tendency and dispersion of the observed outcomes across experimental conditions. For pairwise comparisons, we have used Welch’s t -test, which is appropriate when group variances cannot be assumed to be equal, to compare AMOF against each baseline method. As multiple comparisons are conducted across several metrics and baselines, Bonferroni correction is applied to control the family-wise error rate and reduce the likelihood of spurious significance; unless stated otherwise, significance is assessed at a family-wise $\alpha = 0.05$ after correction. Additionally, we assess practical importance using Cohen’s d , guided by established effect-size recommendations [15].

Table 6: Forward-looking (30s ahead) degradation-detection performance on Online Boutique. The detector is trained on W1 and evaluated on unseen W2 and W3. AUC denotes the area under the receiver operating characteristic (ROC) curve.

Workload	Accuracy	Precision	Recall	F1	AUC
W1 Normal (in-sample)	0.995	0.970	0.989	0.979	1.000
W2 Stress (unseen)	0.935	0.939	0.749	0.834	0.984
W3 Extreme (unseen)	0.948	0.936	0.815	0.871	0.979

5 Results

5.1 RQ1: Detection Generality

We evaluate the detector as a thirty-second-ahead early-warning predictor (Section 3.4): trained only on normal-load (W1) traces and tested on previously unseen stress (W2) and extreme (W3) workloads. Because the target is the degradation state at $t + 30$ s rather than a concurrent function of the input features, the metrics reflect genuine anticipation rather than re-derivation of the labeling rule. Table 6 reports the outcome for Online Boutique, the case study that exhibits a non-trivial degradation rate across all regimes (11.9% under W1, rising to roughly 21.5% under W2 and W3).

The detector generalizes across workload regimes that it never observed during training. On unseen stress and extreme traffic it sustains an area under the ROC curve above 0.97 and a precision of roughly 0.94, while recall declines to 0.75–0.81. This precision–recall profile is consistent with a conservative early-warning gate: when the detector raises an adaptation flag it is rarely wrong, and the missed events are predominantly transient threshold crossings rather than sustained degradation. The result substantiates the gated architecture, in which the Deep Q-Network is activated only when degradation is anticipated, thereby avoiding unnecessary reconfiguration during periods of operational stability.

The relative contribution of each input feature is informative. Figure 5 reports the Gini importance of the six state dimensions for the forward-prediction task. Latency remains the single strongest predictor, as expected from saturation theory, yet the three design-principle proxies (granularity, coupling, and cohesion) together account for 52.5% of the total importance, exceeding the combined contribution of the raw resource signals. The detector therefore relies substantially on architectural state when anticipating future degradation, which supports the central premise of this work: design-principle telemetry carries early-warning signal that conventional resource-only monitoring does not capture. Because these design-level attributes evolve through service interactions, they remain informative even when resource utilization is momentarily stable while latent structural pressure continues to build.

Two boundary conditions qualify this finding. First, cross-workload generalization is a prerequisite for deployable adaptive control: operational microservices rarely encounter traffic that matches the training distribution, so a detector confined to its training

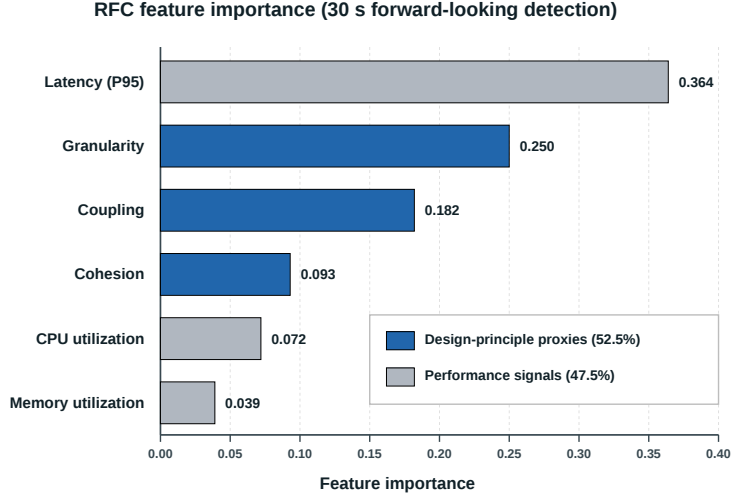


Fig. 5: Feature importance of the forward-looking degradation detector on Online Boutique. Design-principle proxies (granularity, coupling, cohesion) jointly account for the majority (52.5%) of predictive importance.

regime would offer little practical value. The results indicate that AMOF meets this requirement on Online Boutique. Second, the Sock Shop case study could not support an equivalent train-on-normal evaluation, because it produced no threshold-crossing degradation events under normal load and only 5.8–7.0% under stress and extreme load. In other words, under the calibrated workloads Sock Shop retained sufficient resource headroom that the detector had no positive examples to learn from at W1. We treat this as a transparency point rather than a detection result, and we return to it in Section 7: the degradation-detection claim is established on the application that actually degrades, while Sock Shop primarily exercises the resource-efficiency and design-principle analyses that follow.

5.2 RQ2: Runtime Effectiveness Relative to Baselines

Figure 6 compares the cross-application performance of AMOF relative to Kubernetes HPA across the three workloads. In Case Study 1 (Online Boutique), AMOF reduced CPU utilization by 14.5%, 52.2%, and 49.7% under W1, W2, and W3 respectively, and memory utilization by 49.1%, 46.4%, and 48.4% over the same workloads, while latency increased by 42.6%, 33.0%, and 20.2% (all percentage changes are computed from unrounded per-run means, so they can differ marginally from ratios of the rounded values in Table 7). In Case Study 2 (Sock Shop) the corresponding CPU reductions were 8.8%, 4.0%, and 33.1%, and the memory reductions were 4.0%, 9.1%, and 17.2%. Sock Shop latency remained marginally below that of HPA under W1 and W2 (−10.6% and −1.9%) but rose sharply under W3 (+198.6%). The two case studies therefore exhibit a common pattern: AMOF delivers consistent improvements

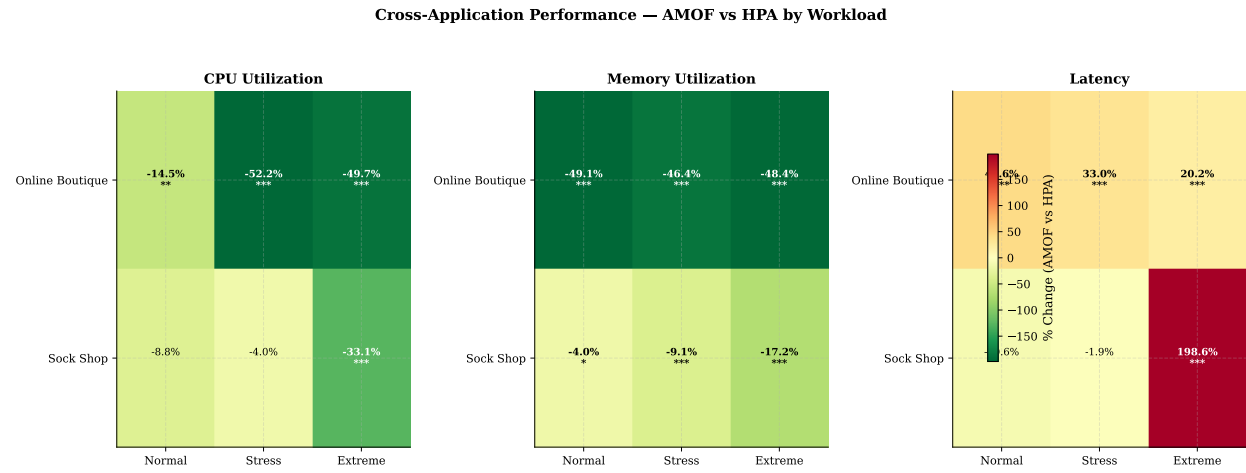


Fig. 6: Cross-application performance of AMOF relative to Kubernetes HPA by workload regime (W1–W3). Negative percentages denote lower resource utilization for CPU and memory; for latency, positive values denote an increase relative to HPA. Significance markers follow the convention established in Table 8.

Table 7: Descriptive statistics (mean over five runs) for the runtime metrics by application, approach, and workload. CPU is reported in millicores (mc), memory as a percentage of the container limit, and P95 latency in milliseconds. Per-run data and dispersion statistics are provided in the replication package [1].

Approach	WL	CPU (mc)	Mem (%)	P95 lat. (ms)
<i>Case Study 1: Online Boutique (gRPC)</i>				
Static	W1	10.20	2.7	227
	W2	15.56	2.8	380
	W3	9.64	3.0	409
HPA	W1	22.96	3.4	212
	W2	56.63	3.6	251
	W3	70.93	3.8	329
Rule-Based	W1	10.12	2.8	240
	W2	15.49	2.8	363
	W3	10.08	3.0	397
AMOF	W1	19.63	1.7	302
	W2	27.09	1.9	334
	W3	35.71	2.0	396
<i>Case Study 2: Sock Shop (HTTP/REST)</i>				
Static	W1	6.31	6.5	29
	W2	10.24	6.6	62
	W3	12.86	7.2	130
HPA	W1	6.73	6.7	32
	W2	10.85	7.3	63
	W3	18.97	8.8	38
Rule-Based	W1	6.14	6.5	29
	W2	10.57	6.6	63
	W3	12.97	7.0	110
AMOF	W1	6.14	6.4	28
	W2	10.42	6.7	62
	W3	12.70	7.3	115

441 in resource efficiency relative to the reactive baseline, accompanied by a workload-
 442 dependent latency penalty that grows under extreme overload. This penalty should not
 443 be read as a universally unfavorable outcome; rather, it reflects an explicit resource-
 444 versus-QoS trade-off introduced by AMOF’s resource-efficient adaptation policy, which
 445 weights CPU and memory improvements at 0.7 against 0.3 for latency (Eq. 7). The
 446 pattern suggests that AMOF is best suited to cost-sensitive or resource-constrained
 447 environments, whereas latency-critical deployments would require SLO-aware reward
 448 weighting or a hybrid fallback to HPA under extreme load.

449 The +198.6% Sock Shop latency figure under W3 warrants explicit interpretation,
 450 because in relative terms it appears severe. In absolute terms it is not: the corresponding

mean P95 latencies are approximately 115 ms for AMOF, 130 ms for the Static baseline, 110 ms for Rule-Based, and only 38 ms for HPA, all far below the 500 ms degradation threshold. The large percentage therefore arises from an unusually small denominator rather than a large numerator: under extreme load HPA scaled out aggressively and drove its own P95 latency *below* its W2 value (63 ms), over-provisioning replicas to minimise queueing. Against this compressed HPA baseline, AMOF’s latency, which is statistically indistinguishable from the Static and Rule-Based baselines under W3 (Table 8), registers as a large relative increase. Three mechanisms on AMOF’s side reinforce the effect: the resource-leaning reward weighting noted above, a merge-dominated control policy on Sock Shop (Section 5.4) that consolidates rather than expands capacity, and the cap of five adaptations per 300 s, which deliberately limits the pace of scale-out while Sock Shop’s synchronous HTTP/REST call chains amplify queueing. The result is consistent with the design intent: AMOF trades a modest, sub-threshold latency increase for substantial and repeated resource savings, and is not intended to outperform HPA on latency under peak overload.

The Online Boutique CPU comparison against the Static and Rule-Based baselines in Table 8 also warrants attention. AMOF recorded *higher* CPU utilization than both the fixed-replica Static baseline and the Rule-Based baseline on Online Boutique, with Cohen’s $d = 0.35$ and $p < 0.001$ in both comparisons, even though it reduced CPU by 45.2% relative to HPA when pooled across the three workloads. The two observations are not contradictory: the Static and Rule-Based baselines perform little or no reconfiguration, whereas AMOF actively manages the deployment, executing 333 adaptations across the Online Boutique runs (about 22 per experiment, Section 5.4). This active management, together with the dense inter-service traffic of the gRPC stack, consumed CPU cycles that a do-nothing controller avoids entirely, so AMOF costs more than passivity but far less than HPA’s reactive scale-out. The CPU premium over passive baselines is a calculable cost that practitioners under strict compute budgets should weigh when configuring AMOF’s adaptation frequency and scrape interval; raising the inter-adaptation gap through the λ penalty reduces it.

5.3 RQ3: Design-Principle Outcomes

Table 8 reports the effect sizes for AMOF versus each baseline on the three architectural proxies. The pattern is consistent in its central message across both case studies: coupling does not move under runtime control, whereas cohesion is the more tractable principle. On Online Boutique, where the service mesh provided live interaction telemetry, AMOF produced a small but highly significant improvement in cohesion (up to $d = 0.14$, $p < 0.001$ against every baseline), while coupling remained statistically invariant ($d \leq 0.003$, all non-significant) and granularity showed only a marginal negative effect against HPA ($d = -0.04$, $p < 0.05$). On Sock Shop the coupling and cohesion proxies registered no change; as discussed below and in Section 7, these values reflect the application’s static service topology in the recorded telemetry rather than a measured runtime response, so the Sock Shop rows characterise the structural baseline rather than the effect of adaptation. We retain the coupling row precisely because its invariance is itself the finding: across both applications and every baseline, runtime control left coupling unchanged.

Table 8: Effect size (Cohen’s d) for AMOF relative to each baseline. For CPU, memory, and latency, negative values indicate lower values for AMOF; for cohesion and granularity, positive values indicate higher (better) values for AMOF. Significance markers: * $p < 0.05$, *** $p < 0.001$, ns: not significant.

Metric	Online Boutique			Sock Shop		
	vs S	vs H	vs R	vs S	vs H	vs R
CPU utilization	0.35***	-0.25***	0.35***	-0.00 ns	-0.06***	-0.00 ns
Memory usage	-0.58***	-0.89***	-0.58***	0.01 ns	-0.12***	0.02 ns
Latency (normalized)	0.01 ns	0.13***	0.02 ns	-0.02 ns	0.10***	0.00 ns
Coupling	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns	0.00 ns
Cohesion	0.11***	0.13***	0.14***	0.00 ns	0.00 ns	0.00 ns
Granularity	0.02 ns	-0.04*	0.02 ns	0.00 ns	-0.03*	0.00 ns

Abbreviations: S = Static, H = HPA, R = Rule-Based. The vs-S and vs-R columns are near-identical because the Rule-Based thresholds rarely fired under these calibrated workloads, so Rule-Based behavior converged to that of the fixed-replica Static baseline. The Sock Shop coupling and cohesion rows reflect static topology-derived proxy values in the recorded telemetry (Section 7).

The cohesion result, although modest in magnitude, is the clearest evidence that a narrowly scoped runtime intervention can shift an architectural signal. The ADJUST-COHESION action reduces redundant downstream invocations and redistributes internal versus external request handling; its measurable positive influence on the Online Boutique cohesion proxy confirms that local execution-policy adjustments can move the observable cohesion signal even within a fixed service topology. Notably, this effect emerged despite ADJUSTCOHESION being a comparatively infrequent action on Online Boutique (41 of 333 adaptations, Section 5.4), whereas the far more frequent RECONF-COUPPLING action (179 of 333) failed to move coupling at all. The contrast is instructive: action frequency does not determine architectural effect; tractability does. Coupling resisted adjustment even under sustained intervention, while cohesion responded to a handful of targeted actions.

The granularity effect is small by construction. Granularity is measured through the normalized dependency ratio, which reflects the structural consistency of service interconnections within the deployment graph. Because AMOF’s SPLIT and MERGE actions operate at the runtime-configuration level rather than at design or code boundaries, their measurable effect on the observed dependency ratio within a ten-minute experiment window is inherently limited. Substantial granularity adjustment would require either longer observation horizons or complementary mechanisms; we examine this limitation directly in Section 7 and identify hour-scale adaptation windows as the natural setting in which granularity effects could become detectable.

5.4 Qualitative Adaptation Behavior

Figure 7 presents the per-action frequencies executed by AMOF across the two benchmarks. Online Boutique produced the heavier controller footprint, dominated by coupling reconfiguration (179 actions), followed by service merges (68), service splits

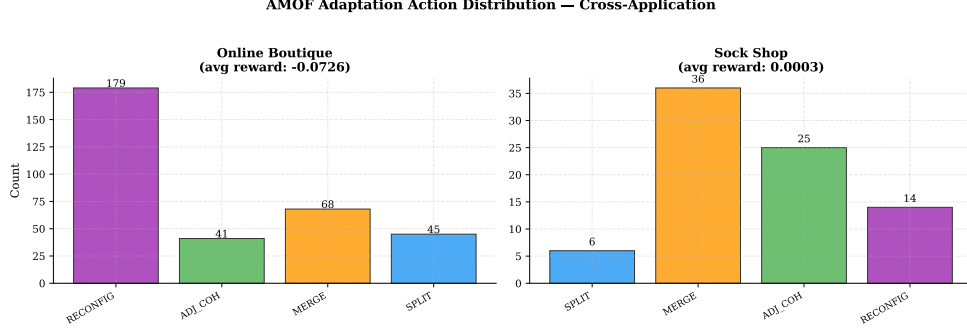


Fig. 7: Distribution of AMOF adaptation actions across both case studies. Online Boutique triggers substantially more controller actions than Sock Shop (333 versus 81).

(45), and cohesion adjustments (41). Sock Shop triggered far fewer actions overall (81), led by merges (36) and cohesion adjustments (25), with coupling reconfiguration (14) and splits (6) trailing. These counts should be read alongside the effect sizes in Section 5.3 rather than as a direct proxy for them. Although Sock Shop devoted a larger *share* of its limited action budget to cohesion adjustment (25 of 81), the measurable cohesion improvement appeared on Online Boutique, where the service mesh exposed live interaction telemetry; Sock Shop’s cohesion actions operated against statically derived proxy values and therefore left no measurable trace. On Online Boutique the controller leaned heavily on coupling reconfiguration (179 actions), yet coupling did not move, whereas the comparatively infrequent cohesion adjustments (41 actions) did. Action frequency and architectural effect are thus only loosely related; the binding constraint is tractability, not how often an action fires.

6 Discussion

6.1 Implications for Adaptive Resource Allocation

The results indicate that AMOF is particularly beneficial when practitioners require a *balanced* operational objective that accounts for resource efficiency and architectural stability at the same time. Particularly, memory reductions are consistent and substantial in the resource-intensive case study, indicating that AMOF can curb the memory over-allocation that may emerge under reactive autoscaling. In contrast, HPA remains competitive, and at times better, for latency minimization under heavy load, reflecting its inclination toward horizontal expansion.

In operational settings, practitioners whose primary concern is reducing infrastructure cost under heavy load benefit most from AMOF’s CPU and memory efficiency gains. Therefore, organizations operating under strict SLA compliance may prefer to retain HPA as the primary scaling mechanism and apply AMOF for architectural quality optimization only in order to prevent over-allocation of resources.

546 6.2 Analytic Generalization across both Case Studies

547 AMOF exhibits a consistent qualitative pattern across the two case studies. CPU and
548 memory reductions relative to HPA are observed in both Online Boutique and Sock
549 Shop, while latency trade-offs emerge under high load in both systems. The intensity of
550 runtime effects varies, with Online Boutique exhibiting substantial resource gains (up
551 to 52.2% CPU reduction) in comparison with Sock Shop (up to 33.1%). This difference
552 reflects the distinct architectures of the two applications: Online Boutique’s gRPC-based
553 polyglot stack presents more optimization opportunities for design-principle-aware
554 control than Sock Shop’s simpler HTTP/REST protocol. The consistency of this
555 pattern across both case studies and all three workloads, despite differing magnitudes,
556 strengthens the analytic generalization of our findings.

557 The architectural divergence between the two heterogeneous benchmarks strength-
558 ens the case for design-principle monitoring. gRPC’s binary framing and multiplexed
559 flows generate a higher inter-service call volume than REST’s stateless request model.
560 The choice of communication protocol therefore carries operational weight for runtime
561 adaptation: deployments built on, or migrating toward, gRPC-style interaction present
562 richer signal for AMOF’s control than predominantly REST-based deployments.

563 6.3 Why Coupling Is Hard to Move at Runtime

564 Figure 8 summarizes the maximum observed effect sizes for the three design principles
565 across both case studies. The effect on service independence (coupling) is minor in
566 both applications, reflecting a structural constraint in which coupling is determined
567 by business logic. This finding is consistent with empirical observations that coupling
568 is a persistent quality factor rooted in service boundaries and cross-service commu-
569 nication [50]. In contrast, cohesion responds to local execution policies (for example,
570 reducing redundant downstream calls or altering in-service request handling), which is
571 reflected in the measurable cohesion improvement on Online Boutique.

572 Moving coupling at runtime would require the controller to alter the service call
573 mechanism itself. Such an intervention reaches into service contracts and API boundaries
574 rather than deployment configuration alone. Processes such as routing policies, traffic
575 shaping at node level, or optional-call suppression in specific scenarios/ flags represent
576 valuable directions for extending AMOF’s coupling effects. Without such extensions,
577 the current RECONF COUPLING action modifies call-rate targets at the mesh level
578 but cannot eliminate structurally mandated dependencies, explaining the observed
579 limitation on service coupling traces and inter-service communication mechanisms.

580 6.4 Performance Trade-Offs

581 AMOF has shown adaptable behavior when it comes to trade-off among performance
582 parameters (CPU, memory and latency). If latency is non-negotiable, increasing the
583 latency weight in Eq. 7 or decreasing the adaptation penalty can encourage more
584 interventions (converging toward HPA). If cost efficiency dominates, the observed
585 memory gains motivate keeping the CPU/memory weights substantial, while ensuring
586 stability through bounded adaptations per window.

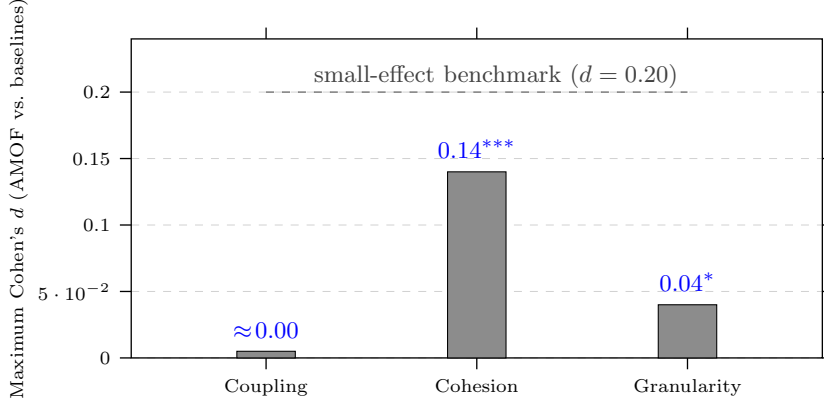


Fig. 8: Effect sizes (Cohen's d) observed for AMOF-induced adjustments to the three design-principle, aggregated across both case studies. Significance markers follow Table 8: * $p < 0.05$, *** $p < 0.001$. Coupling remains invariant under runtime control; cohesion attains the largest response and granularity exhibits marginal effects. The dashed line marks the conventional small-effect benchmark of $d = 0.20$, corresponding to $r \approx .10$ in the guidelines of Gignac and Szodorai [15].

For teams considering AMOF in production, three practical observations follow from the results. First, AMOF provides its most pronounced benefit during sustained high-load periods on Online Boutique, where reduced CPU utilization and memory savings translate directly into lower infrastructure expenses. Second, the latency trade-off under extreme load on Sock Shop should inform capacity planning: extreme traffic spikes remain better served by HPA's rapid horizontal expansion, so a hybrid control loop that defers to HPA under W3-class conditions is a natural engineering extension. Third, the CPU premium that AMOF incurs over the passive Static and Rule-Based baselines on Online Boutique reflects the cost of active management; configuring a minimum adaptation cool-down of at least 300 s between consecutive interventions, consistent with the adaptation bound already incorporated into the reward function, keeps this overhead in check.

7 Threats to Validity

We discuss the principal threats to validity following the established categories for empirical software engineering [48].

Construct validity.

Coupling, cohesion, and granularity are abstract design properties that cannot be measured directly at runtime, so AMOF estimates them through observable proxies derived from inter-service call rates and dependency structure (Eqs. 2–4). These proxies approximate, rather than define, the underlying principles. This limitation is most consequential for Sock Shop. Although the case study's telemetry is available in the

replication package, the recorded coupling and cohesion values for Sock Shop correspond to its static service topology rather than to live, time-varying mesh measurements: across all runs these proxies took a small set of discrete, dependency-count-derived values with zero temporal variance, indicating that the metric pipeline fell back to a topology-based estimate for that namespace. Consequently, the Sock Shop coupling and cohesion rows in Table 8 characterise a structural baseline and should not be interpreted as evidence that adaptation had no runtime effect on those principles. On Online Boutique, by contrast, the cohesion proxy varied over time for the services that AMOF acted upon, so the cohesion result reported in Section 5.3 rests on genuinely dynamic measurements. A second construct consideration concerns latency, which is min-max normalised by a fixed 2000 ms divisor before being used in the reward and effect-size computations. Relative latency changes expressed as percentages can therefore appear large when the absolute values are small, as in the +198.6% Sock Shop figure discussed in Section 5.2; we report absolute millisecond values in Table 7 to support correct interpretation. Finally, the degradation detector is trained against a prospective, threshold-based definition of degradation evaluated thirty seconds ahead. The forward horizon removes the label leakage that a concurrent definition would introduce, but the resulting detector is calibrated to that specific horizon and threshold set; different operational definitions of degradation would yield different detectors.

627 *Internal validity.*

The experiments were executed on a single-node MicroK8s cluster. This choice was deliberate: confining all services to one node removes cross-node network variability, scheduler placement effects, and inter-node bandwidth contention as confounding factors, which strengthens the internal comparability of the four adaptation strategies under identical conditions. The cost is that the study does not capture phenomena that arise only in distributed clusters, such as multi-node scheduling latency, network partitions, or node failures. To limit other sources of internal variation we repeated every configuration across five independent runs, applied identical workload generators across approaches, and used Welch’s *t*-test with Bonferroni correction to guard against spurious significance under unequal variances (Section 4.5).

638 *External validity.*

The evaluation covers two benchmark applications, Online Boutique and Sock Shop, selected for their contrasting communication models (gRPC versus HTTP/REST) and their wide use in microservices research. Two systems cannot establish statistical generalization to the population of cloud-native applications; our claims are therefore framed as analytic rather than statistical generalization [48], and the consistency of the resource-efficiency pattern across two architecturally distinct systems is what we take as evidence of transferability. The single-node setting further bounds external validity, and multi-node and multi-cloud replication remains necessary before the findings can be assumed to hold at production scale. The reliance on synthetic, calibrated workloads is a related limitation: under these workloads Sock Shop never crossed the degradation thresholds during normal load, which both prevented a train-on-normal detection evaluation for that application and signals that the workload calibration left

651 Sock Shop with substantial resource headroom. Production traffic with different burst
652 characteristics could exercise the system differently.

653 *Conclusion validity.*

654 Several reported effect sizes are small in absolute terms. The cohesion improvement
655 on Online Boutique, although statistically significant at $p < 0.001$ after correction,
656 reaches only $d \approx 0.14$, and the granularity effects are smaller still. We interpret these
657 as bounded results rather than strong practical effects, and we rely on Cohen’s d with
658 the effect-size guidelines of Gignac and Szodorai [15] (small $r = .10$, corresponding to
659 $d \approx 0.20$) to avoid overstating their magnitude. The coupling invariance is reported as a
660 null result, which we regard as substantive given its consistency across both applications
661 and all baselines and its alignment with prior observations that coupling is rooted in
662 service boundaries [50]; nonetheless, a null result obtained through a particular proxy
663 cannot exclude the possibility that a different runtime mechanism could move coupling.

664 8 Conclusion and Future Work

665 This paper presented an empirical evaluation of AMOF, a design-principle-aware
666 runtime adaptation framework for cloud-native Kubernetes microservices. By combining
667 service-level telemetry with runtime proxies for coupling, cohesion, and granularity,
668 AMOF extends conventional autoscaling with architecture-aware decision support. The
669 framework integrates a Random Forest forward-looking degradation-detection gate
670 with a Deep Q-Network adaptation policy, enabling runtime decisions that consider
671 CPU utilization, memory consumption, latency, and adaptation cost.

672 The multi-case evaluation shows that the detector anticipates degradation thirty
673 seconds ahead with an area under the ROC curve above 0.97 on unseen workloads,
674 with design-principle proxies supplying the majority of its predictive signal, and that
675 AMOF reduces CPU and memory utilization relative to Kubernetes HPA under selected
676 workload conditions. At the same time, the results reveal a measurable latency trade-
677 off, particularly under extreme workload regimes. This finding indicates that AMOF
678 is most appropriate for cloud environments where resource efficiency, infrastructure
679 cost, and adaptive runtime management are primary concerns, while latency-critical
680 deployments require SLO-aware reward tuning or hybrid integration with conventional
681 autoscaling mechanisms.

682 Future work will extend AMOF in three directions. First, we will validate the
683 framework on multi-node and multi-cloud deployments to strengthen external validity.
684 Second, we will incorporate explicit SLO constraints into the reward function to improve
685 latency-sensitive adaptation. Third, we will investigate stronger runtime mechanisms
686 for coupling and granularity adaptation, including service-mesh policies, contract-aware
687 routing, and longer-horizon adaptation strategies that extend the observation window
688 beyond minutes to the hour scale at which granularity effects are likely to become
689 measurable.

690 *Replication Package.*

691 For verification and extension, the complete AMOF implementation, including the RFC
692 degradation detector, the DQN policy network, the observability stack configuration
693 (Prometheus and Grafana), Kubernetes deployment manifests for both benchmark
694 applications, experiment scripts, raw telemetry data, and the statistical analysis code,
695 is available in the replication package [1].

696 **Abbreviations**

697 AMOF: Adaptive Microservices Optimization Framework; RFC: Random Forest
698 Classifier; DQN: Deep Q-Network; HPA: Horizontal Pod Autoscaler; MAPE-K: Monitor-
699 Analyze-Plan-Execute over a shared Knowledge base; AUC: Area under the receiver
700 operating characteristic curve; ROC: Receiver operating characteristic; QoS: Quality
701 of Service; SLO: Service-Level Objective; SLA: Service-Level Agreement; ML: Machine
702 Learning; RL: Reinforcement Learning; MSA: Microservice Architecture; DDD: Domain-
703 Driven Design; MDD: Model-Driven Development; gRPC: Google Remote Procedure
704 Call; REST: Representational State Transfer; CNI: Container Network Interface; P95:
705 95th percentile.

706 **Declarations**

707 **Availability of data and materials.** The datasets generated and analysed during
708 the current study, together with the complete AMOF implementation, experiment-
709 orchestration scripts, Kubernetes and observability configurations, raw telemetry, and
710 statistical-analysis code, are openly available in the AMOF-Case-Study repository:
711 <https://github.com/Syed-Fakhar-Abbas/AMOF-Case-Study>.

712 **Competing interests.** The authors declare that they have no competing interests.

713 **Funding.**

714 The authors declare that no funds, grants, or other support were received during
715 the preparation of this manuscript.

716 **Author's contributions.**

717 S.F.A. has conceived the idea, designed and implemented the framework, conducted
718 the experiments and drafted the manuscript. M.A. supervised the study, provided
719 methodological guidance. N.A. contributed to perform data analysis and comparison of
720 both industrial benchmarks included in our work. A.B. contributed to the refinement
721 of the content and critical review of the manuscript. S.Aa. contributed in building
722 DevOps pipeline for the conduct of experiments and statistical analysis, M.H. reviewed
723 the manuscript and added a mathematical validations/ multiobjective reward function
724 in the script. M.F.A. contributed in adding suitable tools for observability stack for
725 performance evaluation. All authors read and approved the final manuscript.

726 **References**

- 727 [1] Abbas SF, Ahmed M, Aamir S (2025) AMOF: Adaptive microservice optimiza-
728 tion framework replication package. <https://github.com/Syed-Fakhar-Abbas/>

- 729 [AMOF-Case-Study](#), replication package for “Predictive Runtime Adaptation of
730 Kubernetes Microservices via Design-Principle-Aware Hybrid Learning”
- 731 [2] Abbas SF, Ahmed M, Bakhshi A, et al (2026) Toward intelligent runtime
732 adaptation in microservices: Empirical trade-off analysis and a hybrid ai
733 framework. *Automated Software Engineering* 33(83). [https://doi.org/10.1007/
734 s10515-026-00627-8](https://doi.org/10.1007/s10515-026-00627-8)
- 735 [3] Abdel Khaleq A, Ra I (2023) Intelligent microservices autoscaling module using
736 reinforcement learning. *Cluster computing* 26(5):2789–2800. [https://doi.org/https:
737 //doi.org/10.1007/s10586-023-03999-8](https://doi.org/https://doi.org/10.1007/s10586-023-03999-8)
- 738 [4] Abdullah M, Iqbal W, Berral JL, et al (2020) Burst-aware predictive autoscaling for
739 containerized microservices. *IEEE Transactions on Services Computing* 15(3):1448–
740 1460. <https://doi.org/10.1109/TSC.2020.2995937>
- 741 [5] Abdullah M, Iqbal W, Mahmood A, et al (2020) Predictive autoscaling of microser-
742 vices hosted in fog microdata center. *IEEE Systems Journal* 15(1):1275–1286.
743 <https://doi.org/10.1109/JSYST.2020.2997518>
- 744 [6] Ahmad H, Treude C, Wagner M, et al (2025) Towards resource-efficient reactive
745 and proactive auto-scaling for microservice architectures. *Journal of Systems and
746 Software* 225:112390. <https://doi.org/10.1016/j.jss.2025.112390>
- 747 [7] Al-Dhuraibi Y, Paraiso F, Djarallah N, et al (2017) Elasticity in cloud computing:
748 state of the art and research challenges. *IEEE Transactions on Services Computing*
749 11(2):430–447. <https://doi.org/10.1109/TSC.2017.2711009>
- 750 [8] Bogner J, Wagner S, Zimmermann A (2017) Automatically measuring the main-
751 tainability of service- and microservice-based systems: a literature review. In:
752 Proceedings of the 27th International Workshop on Software Measurement and
753 12th International Conference on Software Process and Product Measurement, pp
754 107–115, <https://doi.org/10.1145/3143434.3143443>
- 755 [9] Chakraborty M, Kundan AP (2021) Grafana. In: *Monitoring cloud-native applica-
756 tions: Lead agile operations confidently using open source software*. Springer, p
757 187–240, <https://doi.org/https://doi.org/10.1007/978-1-4842-6888-9>
- 758 [10] Chen R, Li S, Li Z (2017) From monolith to microservices: A dataflow-driven
759 approach. In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*,
760 IEEE, pp 466–475, <https://doi.org/10.1109/APSEC.2017.53>
- 761 [11] Cojocar MD, Uta A, Oprescu AM (2019) Attributes assessing the quality of
762 microservices automatically decomposed from monolithic applications. In: *2019
763 18th International Symposium on Parallel and Distributed Computing (ISPDC)*,
764 IEEE, pp 84–93, <https://doi.org/10.1109/ISPDC.2019.00021>

- [12] Di Francesco P, Lago P, Malavolta I (2019) Architecting with microservices: A systematic mapping study. *Journal of Systems and Software* 150:77–97. <https://doi.org/10.1016/j.jss.2019.01.001>
- [13] Fritzscht J (2024) Architectural refactoring to microservices: A quality-driven methodology for modernizing monolithic applications. PhD thesis, University of Stuttgart
- [14] Giamattei L, Guerriero A, Pietrantuono R, et al (2024) Monitoring tools for devops and microservices: A systematic grey literature review. *Journal of Systems and Software* 208:111906. <https://doi.org/10.1016/j.jss.2023.111906>
- [15] Gignac GE, Szodorai ET (2016) Effect size guidelines for individual differences researchers. *Personality and Individual Differences* 102:74–78. <https://doi.org/10.1016/j.paid.2016.06.069>
- [16] Gomes F, Rego P, Trinta F (2025) A systematic mapping study on observability of microservices-based applications: fundamentals, classifications, and challenges. *Computing* 107(9):1–25. <https://doi.org/10.1007/s00607-025-01540-w>
- [17] Google LLC (2020) Online boutique: A cloud-native microservices demo application. <https://github.com/GoogleCloudPlatform/microservices-demo>
- [18] Hassan S, Bahsoon R, Kazman R (2020) Microservice transition and its granularity problem: a systematic mapping study. *Software: Practice and Experience* 50(9):1651–1681. <https://doi.org/10.1002/spe.2869>
- [19] Hassan S, Bahsoon R, Minku L, et al (2022) Dynamic evaluation of microservice granularity adaptation. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 16(2):1–35. <https://doi.org/10.1145/3502724>
- [20] Jamshidi P, Pahl C, Mendonça NC, et al (2018) Microservices: the journey so far and challenges ahead. *IEEE Software* 35(3):24–35. <https://doi.org/10.1109/MS.2018.2141039>
- [21] Kaloudis M (2024) Evolving software architectures from monolithic systems to resilient microservices: Best practices, challenges, and future trends. *Journal of Advanced Computer Science and Applications* 15(3):140–157. <https://doi.org/10.14569/ijacsa.2024.0150901>
- [22] Kumar J, Goomer R, Singh AK (2018) Long short term memory recurrent neural network (lstm-rnn) based workload forecasting model for cloud datacenters. *Procedia Computer Science* 125:676–682. <https://doi.org/10.1016/j.procs.2017.12.087>
- [23] Lenarduzzi V, Lomio F, Saarimaeki N, et al (2020) Does migrating a monolithic system to microservices decrease the technical debt? *Journal of Systems and*

- 801 Software 169:110710. <https://doi.org/10.1016/j.jss.2020.110710>
- 802 [24] Lewis J, Fowler M (2014) Microservices: a definition of this new architectural term.
803 MartinFowler com URL <https://martinfowler.com/articles/microservices.html>
- 804 [25] Liang P, Xun Y, Cai J, et al (2026) Autoscaling of microservice resources based
805 on dense connectivity spatio-temporal gnn and q-learning. Future Generation
806 Computer Systems 174:107909. <https://doi.org/10.1016/j.future.2025.107909>
- 807 [26] Mao H, Alizadeh M, Menache I, et al (2016) Resource management with deep
808 reinforcement learning. In: Proceedings of HotNets '16, <https://doi.org/10.1145/3005745.3005750>
809
- 810 [27] Mazlami G, Cito J, Leitner P (2017) Extraction of microservices from monolithic
811 software architectures. In: IEEE International Conference on Web Services (ICWS),
812 pp 524–531, <https://doi.org/10.1109/ICWS.2017.61>
- 813 [28] Mnih V, et al (2015) Human-level control through deep reinforcement learning.
814 Nature <https://doi.org/10.1038/nature14236>
- 815 [29] Narváez D, Battaglia N, Fernández A, et al (2025) Designing microservices
816 using ai: A systematic literature review. Software 4(1):6. [https://doi.org/10.3390/](https://doi.org/10.3390/software4010006)
817 [software4010006](https://doi.org/10.3390/software4010006), URL <https://www.mdpi.com/2674-113X/4/1/6>
- 818 [30] Newman S (2021) Building microservices: designing fine-grained systems, 2nd edn.
819 O'Reilly Media, Sebastopol, CA
- 820 [31] Qu C, Calheiros RN, Buyya R (2018) Auto-scaling web applications in clouds: A
821 taxonomy and survey. ACM Computing Surveys <https://doi.org/10.1145/3148149>
- 822 [32] Richardson C (2018) Microservices Patterns: With Examples in Java. Manning
823 Publications
- 824 [33] Rubak A, Taheri J (2023) Machine learning for predictive resource scaling of
825 microservices on kubernetes platforms. In: Proceedings of the IEEE/ACM 16th
826 international conference on utility and cloud computing, pp 1–8, [https://doi.org/](https://doi.org/10.1145/3603166.3632165)
827 [10.1145/3603166.3632165](https://doi.org/10.1145/3603166.3632165)
- 828 [34] Santos J, Wauters T, Volckaert B, et al (2023) gym-hpa: Efficient auto-scaling via
829 reinforcement learning for complex microservice-based applications in Kubernetes.
830 In: Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS),
831 IEEE, <https://doi.org/10.1109/NOMS56928.2023.10154298>
- 832 [35] Santos J, Reppas E, Wauters T, et al (2025) Gwydion: Efficient auto-scaling for
833 complex containerized applications in Kubernetes through reinforcement learning.
834 Journal of Network and Computer Applications 234:104067. [https://doi.org/10.](https://doi.org/10.1016/j.jnca.2024.104067)
835 [1016/j.jnca.2024.104067](https://doi.org/10.1016/j.jnca.2024.104067)

- [36] Sellami K, Saied MA (2025) Extracting microservices from monolithic systems using deep reinforcement learning. *Empirical Software Engineering* 30. <https://doi.org/10.1007/s10664-024-10547-4>
- [37] Taibi D, Lenarduzzi V, Pahl C (2018) Architectural patterns for microservices: a systematic mapping study. In: *Proc. 8th International Conference on Cloud Computing and Services Science (CLOSER)*, URL <https://hdl.handle.net/10863/5599>
- [38] Tian J, Liu Q, Zhang H, et al (2021) Multiagent deep-reinforcement-learning-based resource allocation for heterogeneous qos guarantees for vehicular networks. *IEEE Internet of Things Journal* 9(3):1683–1695. <https://doi.org/10.1109/JIOT.2021.3089823>
- [39] Trabelsi I, Mahmoudi B, Minani JB (2025) A systematic literature review of machine learning approaches for migrating monolithic systems to microservices. *IEEE Transactions on Software Engineering* <https://doi.org/10.1109/TSE.2025.3603897>, URL <https://ieeexplore.ieee.org/document/11145241>
- [40] Truong HL, Dustdar S (2022) Machine learning for cloud services and resource management: techniques, applications, and research directions. *IEEE Internet Computing* 26(4):42–52
- [41] Turnbull J (2018) *Monitoring with Prometheus*. Turnbull Press
- [42] Vera-Rivera FH, Puerto E, Astudillo H, et al (2021) Microservices backlog—a genetic programming technique for identification and evaluation of microservices from user stories. *IEEE Access* 9:117178–117203. <https://doi.org/10.1109/ACCESS.2021.3106342>
- [43] Vural H, Koyuncu M (2021) Does domain-driven design lead to finding the optimal modularity of a microservice? *IEEE Access* 9:32721–32733. <https://doi.org/10.1109/ACCESS.2021.3060895>
- [44] Wang L, Jiang Y, Wang Z, et al (2022) Operation and maintenance governance of microservices architecture systems: A systematic literature review. *Journal of Software: Practice and Experience* <https://doi.org/10.1002/smr.2433>
- [45] Wang Y, Kadiyala H, Rubin J (2021) Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering* <https://doi.org/10.1007/s10664-020-09910-y>
- [46] Wang Z, Zhu S, Li J, et al (2024) Deepscaling: Autoscaling microservices with stable cpu utilization for large scale production cloud systems. *IEEE/ACM Transactions on Networking* 32(5):3961–3976. <https://doi.org/10.1109/TNET.2024.3400953>

- 871 [47] Weaveworks (2017) Sock shop: A microservices demo application. [https://github.](https://github.com/microservices-demo/microservices-demo)
872 [com/microservices-demo/microservices-demo](https://github.com/microservices-demo/microservices-demo)
- 873 [48] Wohlin C, Runeson P, Höst M, et al (2012) Experimentation in Software Engi-
874 neering. Springer, Berlin, Heidelberg, <https://doi.org/10.1007/978-3-642-29044-2>
- 875 [49] Xie S, Wang J, Li B, et al (2024) PBScaler: A bottleneck-aware autoscaling
876 framework for microservice-based applications. IEEE Transactions on Services
877 Computing 17(2):604–616. <https://doi.org/10.1109/TSC.2024.3376202>
- 878 [50] Zhong C, Zhang H, Li C, et al (2023) On measuring coupling between microservices.
879 Journal of Systems and Software 200. <https://doi.org/10.1016/j.jss.2023.111670>