Demand Prediction and Service Scaling Methods for Microservices in Dynamic Computing Environments

Ming Chen

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE, AUSTRALIA

December 2025

ORCID: 0000-0002-6120-6630

Produced on archival quality paper.

Demand Prediction and Service Scaling Methods for Microservices in Dynamic Computing Environments

Ming Chen
Principal Supervisor: Prof. Rajkumar Buyya
Co-Supervisors: Dr. Maria Read and Dr. Muhammed Tawfiqul Islam

---

Abstract

Cloud- and edge-hosted microservice applications have become the de facto architecture for latency-sensitive online services such as social networks, e-commerce, and analytics platforms. These applications exhibit multiple, intertwined forms of runtime dynamics: (i) non-stationary workloads with bursty and diurnal patterns, (ii) request-type- and workload-mix-dependent call graphs, (iii) imbalanced traffic distributions across microservice replicas, and (iv) time-varying cross-node latency and bandwidth in dynamic cloud clusters. These dynamics make it challenging to maintain end-to-end (E2E) SLA/SLO compliance while utilising cluster resources efficiently.

This thesis develops telemetry-driven algorithms, frameworks, and control policies for dynamics-aware management of microservice applications in cloud and cloud–edge environments. Building on a taxonomy of dynamics and control mechanisms, the central goal is to exploit correlations in historical resource usage and fine-grained runtime telemetry to improve forecasting accuracy, enable faithful and repeatable policy evaluation, and translate observed traffic and network variability into effective scheduling decisions via adaptive placement and scaling under real-world dynamics. The thesis mainly makes the following key contributions:

1. It proposes *EN-Beats*, a correlation-aware ensemble method for multi-resource prediction in clouds. RCorrPolicy selects informative metrics using Spearman correlation, and EN-Beats aggregates N-BEATS-based learners to jointly forecast CPU utilisation, memory usage, and network traffic on the Bitbrains production trace, improving prediction accuracy over deep-learning baselines.

2. It introduces *iDynamics*, a configurable and extensible emulation framework for evaluating microservice scheduling policies under controllable dynamics in the

cloud–edge continuum. iDynamics reconstructs call-graph dynamics and quantifies traffic imbalances, and it injects and measures destination-specific cross-node latency and bandwidth via distributed agents, enabling repeatable and realistic evaluation across cluster scales and dynamic scenarios.

3. It designs *TraDE*, a traffic- and network-aware adaptive rescheduling framework for containerised microservices. TraDE combines service-mesh-based traffic stress analysis, destination-specific delay injection and measurement, and a Parallel Greedy Algorithm (PGA) service–node mapper to trigger safe, replica-level migrations while preserving service availability, reducing response time by up to 48.3% and improving throughput by up to 1.4× over network-aware baselines.

4. It develops *AdaScale*, a two-timescale, SLO-aware scaling and placement framework for microservices under multi-source dynamics. AdaScale integrates call-graph reconstruction, per-edge demand estimation, and inter-node latency measurements into a unified Monitor–Analyze–Plan–Execute (MAPE) loop; it couples a slower workload-mix-aware autoscaling loop with a fast reactive placement loop, reducing average response time by up to 1.93× and increasing throughput by up to 2.16× compared with NetMARKS while meeting SLO targets.

Collectively, these contributions provide a unified, telemetry-driven approach to understanding and managing microservice applications under realistic workload, call graph, traffic, and networking dynamics, enabling more robust SLA/SLO compliance and more resource-efficient microservice deployments in dynamic computing environments.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Ming Chen, December 2025

Preface

Main Contributions

This thesis research has been carried out in the Quantum Cloud Computing and Distributed Systems (qCLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in Chapters 2-6 and are based on the following publications:

- **Ming Chen**, Muhammed Tawfiqul Islam, Maria A. Rodriguez, and Rajkumar Buyya, "Adaptive Management of Microservices in Dynamic Environments: A Review, Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)* [Plan to submit, Jan 2026].

- **Ming Chen**, Maria A. Rodriguez, Patricia Arroba, and Rajkumar Buyya, "ENBeats: A Novel Ensemble Learning-Based Method for Multiple Resource Predictions in Cloud," *Proceedings of the 16th IEEE International Conference on Cloud Computing (CLOUD)*, Pages: 144-154, Chicago, IL, USA, July 2-8, 2023.

- **Ming Chen**, Muhammed Tawfiqul Islam, Maria A. Rodriguez, and Rajkumar Buyya, "iDynamics: A Configurable Emulation Framework for Evaluating Microservice Scheduling Policies under Controllable Cloud–Edge Dynamics", *IEEE Transactions on Service Computing (TSC)* [Under Review, Nov 2025].

- **Ming Chen**, Muhammed Tawfiqul Islam, Maria A. Rodriguez, and Rajkumar Buyya, "TraDE: Network and Traffic-Aware Adaptive Scheduling for Microservices Under Dynamics", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, Volume:

37, Number: 1, Pages: 76-89, January, 2026.

- **Ming Chen**, Muhammed Tawfiqul Islam, Maria A. Rodriguez, and Rajkumar Buyya, "AdaScale: An Adaptive Scaling and Placement Framework for Microservices Under Dynamics", *IEEE International Conference on Distributed Computing Systems (ICDCS)* [Under Review, Dec 2025].

Acknowledgements

My PhD journey has been shaped and sustained by the support of many people. First and foremost, I express my deepest gratitude to my supervisors, Professor Rajkumar Buyya, Dr. Maria Read, and Dr. Muhammed Tawfiqul Islam, for their guidance, encouragement, and continuous support over the years. I also sincerely thank the Chair of my PhD Advisory Committee, Professor Udaya Parampalli, for his valuable advice and support that helped me complete my degree in a timely manner.

I am also grateful to the past and present members of the qCLOUDS Laboratory at the University of Melbourne for their support and camaraderie throughout my candidature. In particular, I thank Dr. Mohammad Goudarzi, Dr. Samodha Pallewatta, Dr. Amanda Jayanetti, Dr. Anupama Mampage, Dr. Tharindu B. Hewage, Jie Zhao, Siddharth Agarwal, Hoa Nguyen, Zhiyu Wang, Duneesha Fernando, Thakshila Imiya Mohottige, Qifan Deng, TianYu Qi, Hootan Zhian, Murtaza Rangwala, Yifan Sun, Prabhjot Singh, Haoyu Bai, Abhishek Sawaika, and Avishka Sandeepa for their help and the friendships we have built over the years.

My heartfelt thanks go to my family, especially my wonderful mother, Xiulan, and my father, Yuhong, for their unconditional love and support. I am particularly grateful to my partner, Cici (Yaxi), for standing by my decision to pursue a PhD, being there through both highs and lows, and for the resilience and love we have shared throughout this journey.

I gratefully acknowledge the University of Melbourne for providing the resources and environment that enabled my doctoral studies. I also thank the past and present administrative staff of the School of Computing and Information Systems for their kind assistance. In addition, I would like to express my sincere appreciation to my previous research supervisors, Dr. Renfa Li, Dr. Chengzhong Xu, Dr. Lujia Wang, and Dr. Kejiang Ye, for their guidance and for preparing me for the challenges of doctoral research.

Finally, I would like to acknowledge the people of China. Decades ago, a group of brave individuals worked to make education accessible and affordable to every citizen, creating equal opportunities for people to pursue their aspirations. TMy PhD dream became a reality because of the efforts of many. I am grateful to all the hardworking people of China whose contributions supported my education. This achievement is as much yours as it is mine.

Ming Chen
December 2025, Melbourne, Australia

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CDF	Cumulative Distribution Function
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
E2E	End-to-End
GPU	Graphics Processing Unit
gRPC	gRPC Remote Procedure Call
HPA	Horizontal Pod Autoscaler
IaaS	Infrastructure as a Service
IoT	Internet of Things
JSON	JavaScript Object Notation
K8s	Kubernetes
KEDA	Kubernetes Event-Driven Autoscaling
LLM	Large Language Model
LSTM	Long Short-Term Memory
ML	Machine Learning
NRMSE	Normalised Root Mean Squared Error
P99	99th Percentile (Latency)
QoS	Quality of Service
QPS	Queries Per Second

REST Representational State Transfer

RL Reinforcement Learning

RNN Recurrent Neural Network

SLA Service-Level Agreement

SLO Service-Level Objective

TCN Temporal Convolutional Network

TCP Transmission Control Protocol

VM Virtual Machine

VPA Vertical Pod Autoscaler

WAN Wide Area Network

Chapter 1
Introduction

1.1 Background and Motivation

Cloud computing has become the primary platform for deploying large-scale, latency-sensitive services across domains such as social media, e-commerce, online gaming, and the Internet of Things (IoT) [4, 5]. A key enabler of this shift is the emergence of cloud-native application stacks, where services are built as independently deployable components and are operated through automated orchestration and observability pipelines. In this ecosystem, *microservice architectures* decompose an application into fine-grained services that communicate using lightweight APIs, enabling rapid development and continuous delivery [6, 7]. However, the same decomposition amplifies the impact of tail latency and cross-service dependencies on end-to-end performance: a small fraction of slow requests or a congested call graph dependency edge can dominate user-perceived response times [8]. Consequently, maintaining SLOs for microservice applications requires resource management mechanisms that reason about (i) the multi-resource demand of individual services and (ii) the graph of interactions among services and infrastructure resources.

This thesis is motivated by a central observation: in production environments, microservice performance is shaped by *multiple intertwined sources of dynamics* rather than by a single fluctuating workload. As illustrated in Figure 1.1, multiple concurrent dynamics co-exist in different layers, including the interaction layer, management layer, and evaluation layer. Modern traces and benchmarks show that workloads exhibit burstiness and diurnal patterns, that applications expose diverse request types that tra-

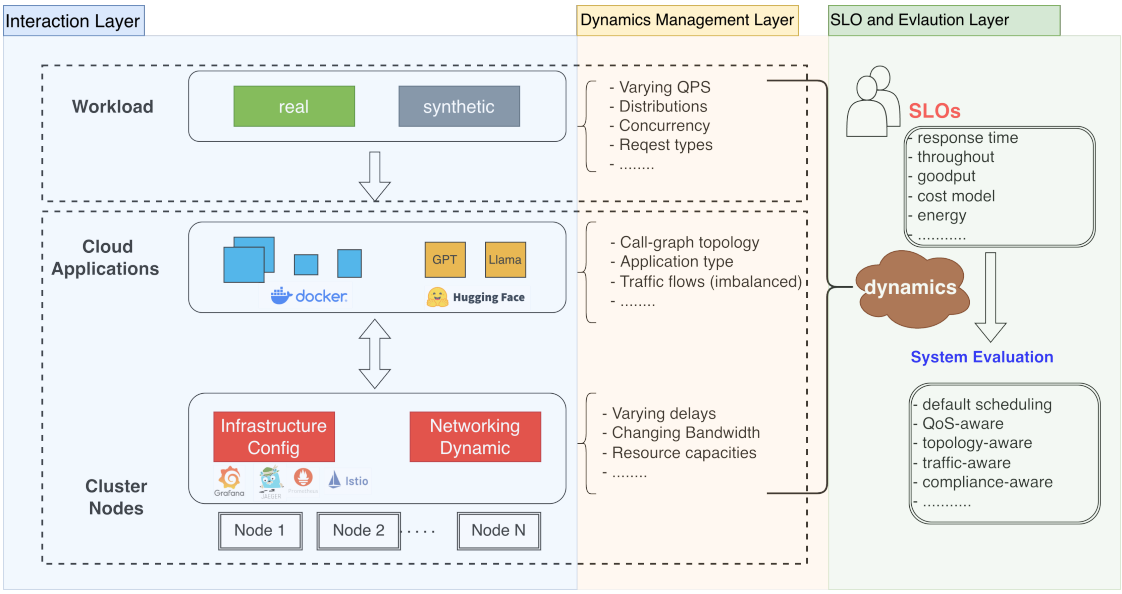


Figure 1.1: Different dynamics spanning across various layers.

verse different call graphs, and that cross-node network conditions and interference can change during execution [2, 4, 5, 9]. While orchestration platforms provide primitives for monitoring, scheduling, and scaling, bridging the gap between these primitives and robust SLO compliance in dynamic environments remains an open research problem. The remainder of this section introduces the technical context for the thesis and highlights why existing approaches are often insufficient when dynamics are multi-dimensional.

1.1.1 From Monoliths to Cloud-Native Microservices

Microservice architectures (MSA) decompose monolithic applications into collections of loosely coupled services, each implementing a specific business capability [10]. In practice, microservices are almost always operated as cloud-native applications: services are containerized, deployed as pods, and managed by an orchestration platform such as Kubernetes [11, 12]. Orchestrators provide core control primitives—deployment, placement, scaling, and failure recovery—over clusters of physical or virtual machines. Their designs are informed by earlier large-scale cluster managers (e.g., Borg [11, 13]), which demonstrated that high utilization and high availability require careful integration of admission control, task packing, and failure-handling policies [13].

However, microservices alter key assumptions behind classic cluster management. Instead of scheduling independent tasks with well-defined lifetimes, platforms must manage dependency graphs of interacting services, each with its own scaling and placement dynamics, and with tight coupling through synchronous communication and tail-latency amplification [2, 8]. As a result, effective resource management must extend beyond CPU and memory provisioning to consider *inter-service communication* and *dependency structure* as primary determinants of end-to-end (E2E) performance.

1.1.2 Microservices, Containers, and the Cloud–Edge Continuum

Microservices are typically containerized and orchestrated by platforms such as Kubernetes [12]. Kubernetes, inspired by earlier large-scale cluster managers such as Borg [11, 13], provides a common abstraction for deploying services, managing their lifecycle, and placing service replicas across a cluster of nodes. It also exposes extensibility mechanisms (e.g., scheduler plugins and custom controllers) that enable researchers and operators to implement specialised placement and scaling logic without re-architecting the entire platform.

Alongside centralised cloud deployments, microservices are increasingly deployed across the *cloud–edge continuum*, where compute nodes span cloud data centres and geographically distributed edge or fog sites. These deployments are driven by latency-sensitive and bandwidth-intensive applications (e.g., 5G/IoT analytics and interactive services) that benefit from proximity to users and data sources [14, 15]. Cloud–edge environments introduce heterogeneity in compute capacity and network connectivity, making placement decisions particularly consequential: when dependent services are separated by slow or variable links, network latency and bandwidth become first-class determinants of end-to-end performance.

Microservice applications are intrinsically *graph-structured*. A single user request typically triggers a chain of RPC/HTTP calls across multiple services, often with branching and repeated calls to downstream services. This structure is captured by the application *call graph*. Realistic microservice benchmarks, such as social-network and e-commerce workloads, include complex call graphs and demonstrate that small changes in depen-

dency paths can shift bottlenecks and tail latency behaviour [2, 16]. Moreover, production traces show that the mix of request types can shift over time, leading to evolving effective call graphs and per-edge traffic patterns [4, 5]. Recent work has begun to explore how to generate or model microservice graphs with production characteristics to support more realistic scaling and placement studies [17]. These observations motivate management approaches that incorporate both service-level resource usage and graph-level interactions.

1.1.3 Service-Level Objectives and Resource Management Loops

Cloud providers and application operators must satisfy Quality of Service (QoS) targets such as response time, throughput, and availability, which are commonly formalised as Service Level Agreements (SLAs) or SLOs. For microservice applications, SLO compliance is complicated by cascaded dependencies along execution paths, shared services that serve multiple applications, and the coupling between compute and network resources [18–20]. A local resource bottleneck at a downstream service can propagate upstream via queueing and retries, producing tail latency spikes even when average utilisation appears moderate [8].

In practice, microservice resource management is implemented through a set of control loops operating at different layers. At the pod level, Kubernetes offers reactive horizontal scaling (HPA) [21]. Many deployments complement HPA with vertical scaling (VPA) to adjust resource requests or limits [22], cluster autoscaling to add or remove nodes [23], and event-driven scaling mechanisms for bursty or queue-based workloads [24, 25]. These loops interact with placement and scheduling, and can lead to oscillatory behaviour or delayed convergence if not coordinated, because scaling decisions change resource contention and network communication patterns [26, 27].

Beyond default orchestration mechanisms, a growing body of research has explored SLO-aware and learning-assisted management for microservices. Sinan uses machine learning models to select resource configurations that meet QoS targets under changing workloads [28], and FIRM proposes fine-grained resource management to enforce SLOs for microservices [18]. Systems for shared microservices address the challenge of co-

locating multiple tenants while maintaining SLA guarantees [19, 20]. Other systems aim to reason explicitly about service dependencies when diagnosing QoS violations or allocating end-to-end SLO budgets across a call graph [29–31]. Despite these advances, two limitations persist in dynamic cloud–edge settings. First, many approaches assume that demand is sufficiently stationary for reactive thresholding or slow reconfiguration to remain effective. Second, network effects are often abstracted away, even though cross-node delays and traffic imbalance can dominate end-to-end behaviour in distributed microservices.

Scheduling and placement decisions are a key part of this context. Kubernetes' default scheduler focuses primarily on compute feasibility, but research has extended scheduling to incorporate QoS, traffic, and network considerations. For example, NetMARKS leverages service-mesh metrics to perform network-aware scheduling for containerised workflows [32], OptTraffic aims to reduce cross-machine traffic in multi-replica microservices [33], and extensions of Kubernetes have explored additional network-aware placement policies [34, 35]. These schedulers illustrate the benefit of incorporating communication patterns, but they also highlight the challenge of reacting to *dynamic* network conditions and traffic distributions without destabilising the overall system.

1.1.4 Observability and Telemetry-Driven Automation

Microservice operation depends on rich telemetry to diagnose and adapt to changing conditions. Modern observability stacks export time-series metrics, logs, and distributed traces [36, 37]. Metrics systems such as Prometheus provide low-overhead, high-frequency signals [38]; distributed tracing systems such as Jaeger and Zipkin capture request paths and latency breakdowns across services [39, 40]; and dashboards such as Grafana support interactive exploration and operational decision-making [41]. Importantly, these signals are both (i) inputs to runtime controllers (e.g., to detect SLO violations or traffic shifts) and (ii) measurement channels for evaluating management policies.

Service meshes extend observability and control by inserting sidecar proxies that standardise routing, enforce policies, and provide uniform telemetry [42–44]. Mesh telemetry can reveal bidirectional traffic between service replicas, enabling the extrac-

tion of per-edge traffic rates and the identification of stressed dependency edges in the application call graph. Mesh control primitives (e.g., traffic shifting, retries, and circuit breaking) can also interact with scheduling and scaling, creating opportunities for integrated management but also increasing the space of possible failure modes.

The availability of telemetry has stimulated the use of *learning-based* methods for prediction and control. Workload prediction and capacity planning have long used statistical forecasting models such as ARIMA [45–47]. More recently, deep learning approaches such as LSTM-based predictors [48, 49], temporal convolutional networks (TCN) [50], and modern architectures such as N-BEATS and transformer variants [1, 51] have demonstrated improved accuracy under non-linearity and long-range dependencies. In microservices and edge settings, learning-based controllers have been explored for dynamic provisioning and placement [52, 53]. However, learning-driven automation in microservice clusters introduces two practical challenges: models must remain robust to non-stationarity and changes in request mix, and control policies must be evaluated in repeatable environments that capture realistic call-graph, traffic, and network dynamics.

1.1.5 Benchmarking and Evaluation for Microservice Management

Developing and validating new management mechanisms requires careful evaluation. However, microservice systems are difficult to experiment with: real deployments are large, multi-tenant, and operationally sensitive, while testbed deployments often fail to capture the scale and dynamics that drive production failures. To bridge this gap, the community has developed a range of benchmarks, emulators, and simulators. Benchmark suites such as DeathStarBench provide realistic microservice applications and workloads for cloud and edge studies [2]. More recent efforts, such as μBench, support systematic construction of benchmark microservice applications [16]. At the same time, simulators and hybrid approaches aim to improve repeatability and reduce experimentation cost, including cloud-native simulation environments [54] and "Kubernetes-in-the-loop" approaches that combine simulation with authentic orchestration behaviour [55].

Despite these tools, reproducing the *combined* effects of call-graph changes, traffic imbalance, and destination-specific network dynamics remains challenging. Generic network emulators can manipulate topologies and link properties, but often lack integration with microservice telemetry and scheduler control hooks [56]. Conversely, microservice benchmarks provide realistic applications but do not, by themselves, offer controlled mechanisms to systematically vary and measure network and traffic dynamics. This gap motivates the need for an extensible evaluation framework that can *inject* and *measure* relevant dynamics in a controlled way while remaining compatible with cloud-native monitoring and scheduling stacks—a need that is addressed by the iDynamics framework developed in this thesis.

1.2 Challenges in Dynamics-aware Microservice Resource Management

Microservice resource management is fundamentally complicated by the fact that *multiple forms of dynamics co-exist and interact* at runtime. Chapter 2 systematically reviews these dynamics and the design space of adaptive management techniques. We highlight the key challenges that motivate the research questions addressed in this thesis.

1.2.1 Multi-Resource Prediction under Non-stationary Workloads

Cloud workloads are often bursty and heavy-tailed, and can change over time due to diurnal patterns, flash crowds, and shifting user behaviour [3, 9]. For microservice applications, these changes translate into non-stationary resource demand at multiple layers: VM-level CPU utilisation, container-level memory pressure, and network traffic induced by service-to-service calls. Traditional autoscalers frequently rely on single-resource signals (e.g., CPU thresholds) and reactive rules, which can lag behind demand spikes and lead to SLO violations. Predictive scaling aims to mitigate this by forecasting future demand, but accurate prediction is difficult when the statistical properties of the workload shift.

A further complication is that microservice demand is *multi-resource* and often corre-

lated: CPU and network usage can co-vary during bursts of RPC traffic, while memory demand may lag behind compute demand due to caching or buffering. Many forecasting approaches treat each metric independently, which ignores cross-metric correlations that could improve robustness. Even when accurate single-metric predictors are available (e.g., ARIMA- or LSTM-based models [45, 48]), downstream control decisions can be fragile because scaling and placement depend on *multiple* metrics simultaneously. This motivates prediction models that (i) explicitly model correlations across resource metrics and (ii) are robust under non-stationary behaviour and heterogeneous time scales. Addressing these challenges is central to the thesis's first contribution, ENBeats.

1.2.2 Call-Graph Dynamics and Traffic Imbalance

Unlike monolithic applications, microservice performance depends on the structure of request execution paths. Production systems typically serve multiple request types (e.g., different API endpoints or user journeys), and each type may exercise a different call graph with distinct bottlenecks and latency sensitivity [2]. Changes in the request mix therefore induce changes in the *effective* call graph observed at runtime. In addition, service meshes and load balancers can produce imbalanced traffic distributions across replicas, which may overload specific service instances or service pairs even when average utilisation appears balanced [32].

From a management perspective, call-graph and traffic dynamics raise three challenges. First, controllers require mechanisms to reconstruct dependency graphs and per-edge traffic rates from telemetry, which is non-trivial in the presence of retries, fan-out/fan-in patterns, and diverse communication modes. Second, the impact of a traffic shift is often *graph-local*: a stressed edge between two services may drive end-to-end latency even if other services are underutilised. Third, scheduling and placement decisions must account for the fact that moving a service instance changes not only compute contention but also network paths, traffic localisation, and the propagation of latency through the service chain. Recent work has started to model such chain effects for provisioning and scaling [31], but practical integration with cloud-native schedulers under

dynamic network conditions remains challenging.

1.2.3 Time-Varying Network Conditions in Cloud–Edge Clusters

In cloud–edge deployments, cross-node latency and bandwidth fluctuate due to congestion, background traffic, and heterogeneous connectivity. Because microservices depend on synchronous RPCs and data-access paths, these network conditions directly affect end-to-end QoS. However, many evaluation tools and resource managers model the network as static or homogeneous, making it difficult to reason about destination-specific latency or bandwidth constraints.

A practical obstacle is the lack of *repeatable* evaluation environments that support both controllable network dynamics and fine-grained measurement. Generic network emulation frameworks such as Kollaps provide decentralised topology emulation [56], but they are not tailored to microservice telemetry and scheduler integration. Likewise, network-aware schedulers such as NetMARKS incorporate service-mesh metrics into Kubernetes scheduling [32], and other extensions explore traffic-aware placement [33, 34]. However, existing schedulers typically assume that network conditions change slowly or rely on aggregate metrics that do not capture destination-specific perturbations. Designing and validating dynamics-aware schedulers therefore requires evaluation frameworks that (i) expose network dynamics as first-class parameters, (ii) capture their interaction with call-graph and traffic dynamics, and (iii) support rapid prototyping of scheduling policies. Addressing this gap motivates the iDynamics framework and the TraDE scheduler developed in this thesis.

1.2.4 Coordinating Scheduling and Scaling under Multiple Dynamics

Cloud-native resource management is inherently multi-loop: placement, rescheduling, autoscaling, and traffic routing can all change the system state. When dynamics are multi-dimensional (e.g., simultaneous workload shifts and network perturbations), treating these loops independently can lead to unstable or inefficient behaviour. For example, rescheduling a service to reduce cross-node latency may temporarily increase compute contention on a target node; reactive autoscaling may add replicas without considering

whether those replicas are placed in latency-optimal locations; and different controllers may react on different timescales, amplifying oscillations and wasting resources.

Addressing this challenge requires a principled control architecture that separates *fast* reactions to imminent QoS violations (e.g., placement adjustments when network latency spikes) from *slower* adaptations to demand trends (e.g., scaling replica counts). The autonomic computing literature formalises such architectures through feedback loops, such as Monitor–Analyse–Plan–Execute (MAPE) models [57]. In the microservice context, this motivates multi-timescale frameworks that integrate telemetry, prediction, and optimisation to coordinate scheduling and scaling decisions. The thesis's final contribution, AdaScale, explores this approach by combining a reactive placement loop with a steady-state autoscaling loop that accounts for workload mixtures and communication modes.

1.3 Research Methodology and Thesis Scope

The thesis adopts an empirical, systems-oriented methodology that combines trace-informed characterisation, prototype implementation on cloud-native platforms, and controlled experimentation. Concretely, we use production-inspired traces to motivate the forms of dynamics that matter in practice (workload non-stationarity, call-graph/request-mix variation, traffic imbalance, and time-varying inter-node network conditions) [4, 5, 9]. We then design algorithms and system mechanisms that operate on the telemetry already available in cloud-native stacks (metrics, traces, and service-mesh observations) [37, 38, 42].

For evaluation, the thesis emphasises repeatability and comparative measurement across alternative policies. Where possible, experiments are trace-driven and are executed on Kubernetes-based testbeds, using realistic microservice benchmarks and workloads [2]. To isolate causal effects and to study interactions between dynamics, we complement deployment-based evaluation with controllable emulation capability through the iDynamics framework and related tooling [54, 56]. Across chapters, the primary evaluation metrics are end-to-end response time (including tail latency), throughput, SLA/SLO violation probability, and overall resource efficiency (e.g., replica counts and

resource consumption).

In terms of scope, this thesis focuses on runtime resource management for microservice applications deployed on container orchestration platforms in cloud and cloud–edge environments. The emphasis is on *prediction*, *scheduling and placement*, and *autoscaling* decisions that impact performance and cost. The scope of this thesis does not include security, data privacy, or developer-facing software engineering aspects of microservice design, except where these factors affect telemetry availability or operational constraints. Similarly, the thesis targets general-purpose microservice applications with request/response interactions; it does not attempt to comprehensively address specialised domains such as serverless function orchestration or tightly coupled HPC workloads.

Finally, the thesis contributions are structured to form a cohesive research. The prediction component (EN-Beats) provides forecasts that can inform proactive provisioning decisions. The evaluation component (iDynamics) provides the experimental control needed to stress-test scheduling policies under combined call-graph, traffic, and network dynamics. The scheduling component (TraDE) demonstrates how fine-grained traffic and network telemetry can be exploited for microservice rescheduling in response to QoS violations. The final component (AdaScale) integrates these insights into a multi-timescale control architecture that jointly reasons about replica counts and placement under multiple concurrent dynamics.

1.4 Research Questions and Objectives

The overarching research problem addressed in this thesis is:

> *How can we design telemetry-driven prediction, evaluation, scheduling, and scaling mechanisms that maintain end-to-end objectives and resource efficiency for microservice applications in cloud and cloud–edge environments under multiple, concurrent runtime dynamics (workload, call-graph, traffic, and network)?*

To address this problem, the thesis investigates the following research questions (RQs):

RQ1: How can we design correlation-aware *multi-resource* prediction models that ex-

ploit correlations among resource metrics to improve prediction accuracy and robustness under non-stationary cloud workloads?

RQ2: How can we construct a configurable and extensible framework to *evaluate* microservice scheduling policies under controllable call-graph, traffic, and network dynamics in cloud–edge environments?

RQ3: How can we develop practical, traffic- and network-aware (re)scheduling mechanisms that adapt microservice placements in response to QoS violations driven by dynamic cross-node delays and changing traffic distributions, while preserving service availability?

RQ4: How can we formulate and implement an integrated, multi-timescale scaling and placement architecture that preserves end-to-end SLOs under concurrent workload, call-graph, traffic, and network dynamics?

The corresponding objectives of this thesis are:

- develop telemetry-driven, correlation-aware prediction methods for CPU, memory, and network demand that are suitable for proactive control;
- design a reproducible evaluation environment that can emulate and measure microservice call-graph, traffic, and network dynamics in a controlled way;
- design and validate adaptive scheduling mechanisms that incorporate traffic stress and destination-specific network conditions; and
- design an SLO-aware control architecture that coordinates fast placement reactions with slower autoscaling to achieve robust performance and efficient resource usage.

1.5 Thesis Contributions

In answering the above research questions, this thesis makes the following contributions:

1. **EN-Beats: Correlation-aware multi-resource prediction in clouds.** Chapter 3 proposes EN-Beats, an ensemble learning-based model that selects correlated resource metrics (via *RCorrPolicy*) and aggregates N-BEATS weak learners into a strong learner for joint prediction of CPU utilisation, memory usage, and network traffic. Using the Bitbrains production dataset, the evaluation demonstrates improved prediction accuracy compared with representative deep learning and statistical baselines.

2. **iDynamics: Evaluation framework for scheduling under controllable microservice dynamics.** Chapter 4 introduces iDynamics, a configurable framework for evaluating microservice scheduling policies under controllable workload, call-graph, traffic, and network dynamics in the cloud–edge continuum. iDynamics integrates a Graph Dynamics Analyzer (call-graph and traffic analysis), a Networking Dynamics Manager (destination-specific latency and bandwidth emulation/measurement), and a Scheduling Policy Extender (policy implementation and comparison).

3. **TraDE: Traffic- and network-aware adaptive scheduling for microservices.** Chapter 5 designs TraDE, a scheduling and rescheduling framework that leverages service-mesh telemetry to detect traffic stress between dependent microservices and adapts placements under changing cross-node network delays. TraDE combines traffic analysis, controllable network dynamics, and a low-overhead mapping algorithm to reduce response time and improve throughput while preserving service availability.

4. **AdaScale: Two-timescale SLA-aware scaling and placement under multiple dynamics.** Chapter 6 develops AdaScale, an adaptive framework that jointly scales and places microservice replicas under heterogeneous dynamics, including network variations, call-graph shifts, and traffic distribution changes. AdaScale decomposes the control problem into a fast reactive placement loop and a slower steady-state autoscaling loop, reducing SLA violation probability and resource consumption compared with representative baselines in cloud–edge experiments.

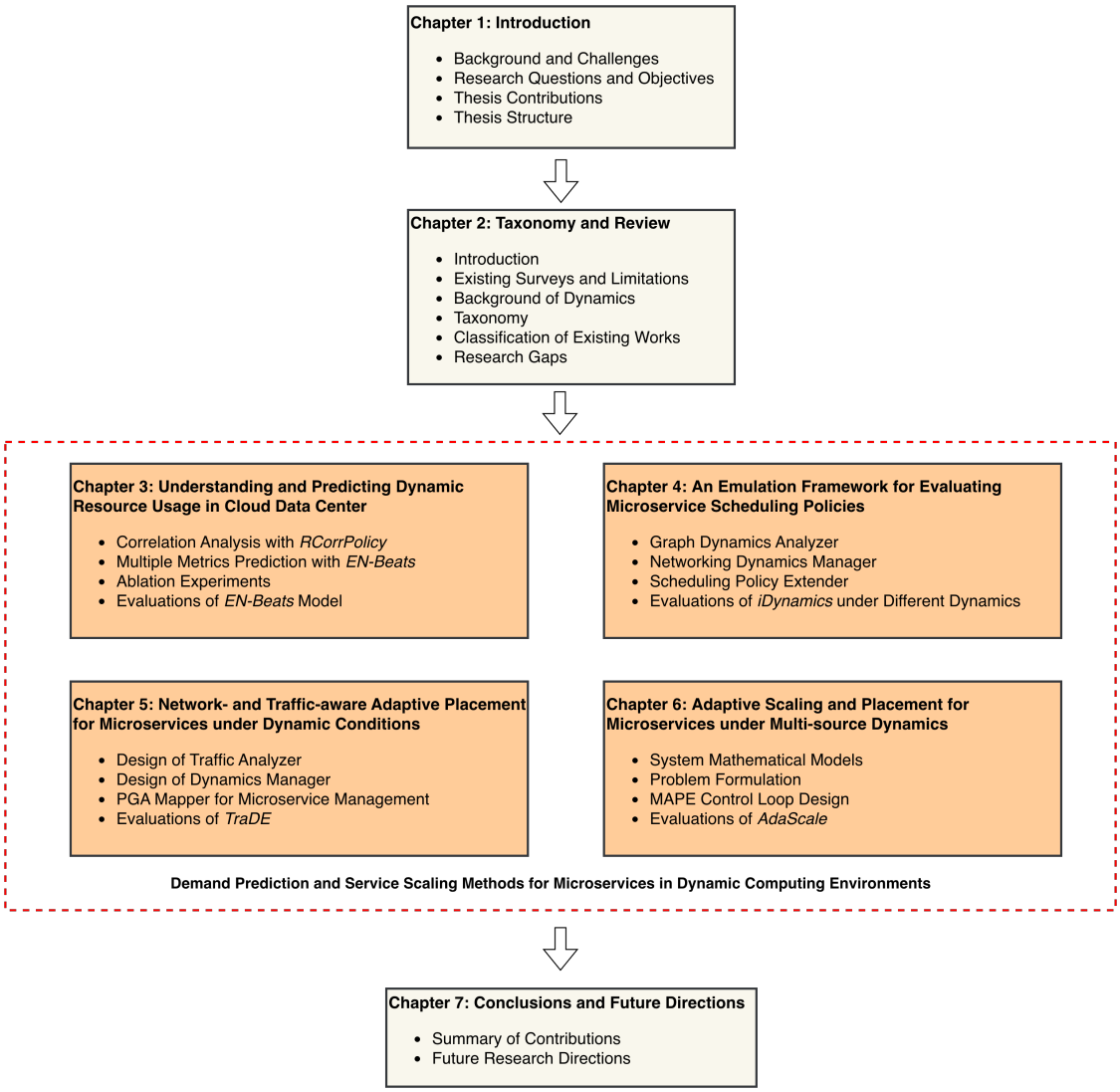


Figure 1.2: Thesis structure.

1.6 Thesis Organisation

This thesis follows a progression from *understanding and modelling dynamics*, to *building controllable evaluation capability*, to *designing adaptive scheduling and scaling mechanisms* that operate reliably under those dynamics. The thesis organisation can be found in Figure 1.2, and the remainder of this thesis is organised as follows:

- **Chapter 2** presents a taxonomy and literature review on dynamics-aware resource

management for microservice applications in cloud and cloud-edge environments. It introduces the main forms of dynamics, proposes taxonomy dimensions, classifies existing techniques, and identifies research gaps that motivate the subsequent chapters.

- **Chapter 3** describes the design and evaluation of RCorrPolicy and EN-Beats for multiple resource prediction in clouds. This chapter details feature engineering, the ensemble learning architecture, and trace-driven experiments on real cloud workloads.

- **Chapter 4** presents the iDynamics framework for evaluating microservice scheduling policies in cloud-edge continuum environments. It explains the architecture and components of iDynamics, including the Graph Dynamics Analyzer, Networking Dynamics Manager, and Scheduling Policy Extender, and provides extensive experimental results under controllable dynamics.

- **Chapter 5** introduces the TraDE framework for traffic- and network-aware adaptive scheduling of microservices. It details the design of the traffic analyzer, dynamics manager, parallel service–node mapper, and microservice rescheduler, and then evaluates them against existing methods.

- **Chapter 6** develops AdaScale, a two-timescale scaling and placement framework for microservice applications under multiple forms of dynamics. This chapter presents the system model, adaptation problem formulation, control-loop design, and evaluation on real testbeds.

- **Chapter 7** concludes the thesis by summarising the main contributions and outlining directions for future research on dynamics-aware microservice resource management.

Chapter 2

A Taxonomy on Adaptive Management of Microservices in Dynamic Environments

Microservices and container orchestration have become popular for building and operating large-scale cloud applications, but their performance and cost-efficiency are increasingly constrained by dynamics*, such as workload bursts, request-path (call-graph) changes, multi-tenant interference, and time-varying network conditions. This chapter (i) reviews existing surveys and clarifies the gap they leave for* dynamics-aware *microservice management, (ii) summarises the key control and observability primitives in modern cloud-native stacks, and (iii) proposes a taxonomy for adaptive management of microservices in dynamic environments.*

Our taxonomy is organized as four primary *dimensions—**(D1) abstraction level**, **(D2) dynamics type**, **(D3) adaptation mechanism**, and **(D4) deployment/evaluation environment**—and two* cross-cutting *axes that repeatedly appear across the literature: **objectives/SLOs** (e.g., tail-latency, cost, energy/carbon) and **telemetry data sources** (metrics, logs, traces). Using this taxonomy, we classify around one hundred representative research works and widely used industry tools/frameworks, and position the thesis contributions (EN-Beats, iDynamics, TraDE, and AdaScale) within the broader design space.*

2.1 Introduction

This chapter is derived from:

- **Ming Chen**, Muhammed Tawfiqul Islam, Maria A. Rodriguez, and Rajkumar Buyya, "Adaptive Management of Microservices in Dynamic Environments: A Review, Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)* [Plan to submit in Jan 2026].

Microservices decompose monolithic applications into a set of independently developed and deployed services, typically communicating via RPC and message-based interactions [10]. In production, microservices are almost always operated as *cloud-native* applications, i.e., containerized and managed by an orchestration platform (most commonly Kubernetes) [11, 12]. This shift improves modularity and operational agility, but it also amplifies *dynamic effects* that were less visible in monoliths: request traffic is bursty and non-stationary, request paths evolve as features are rolled out, and performance becomes sensitive to noisy-neighbor interference and network variability [8, 58–62].

Modern cloud-native stacks provide powerful runtime control primitives (e.g., autoscaling, rescheduling, and traffic shifting), but using them effectively requires *closing the control loop* under partial observability and changing dynamics. Kubernetes exposes reactive controllers such as the Horizontal Pod Autoscaler (HPA) [21], and is increasingly complemented by node- and resource-level controllers (e.g., cluster autoscaling, vertical autoscaling, and event-driven autoscaling) [22–25, 63]. At the same time, production operators rely on observability pipelines to provide the telemetry needed for diagnosis and control, including metrics (Prometheus/Grafana) [38, 41], distributed tracing (Dapper/Jaeger/Zipkin) [36, 39, 40], and open standards such as OpenTelemetry [37]. Service meshes and sidecar proxies (e.g., Istio/Envoy/Linkerd) provide additional control points for routing and latency-aware traffic management [42–44].

These capabilities motivate a growing body of research on *adaptive management* of microservices. However, compared with classic cloud auto-scaling or cluster scheduling, microservices introduce unique challenges: (i) the application is a *dependency graph* rather than a set of independent tasks [29–31, 64]; (ii) decisions at different layers (VM, container, service, graph) interact across different time scales [13, 65–67]; and (iii) realistic evaluation requires capturing both system dynamics and orchestration dynamics under representative benchmarks and traces [2, 4, 16, 55, 68–70].

This chapter provides a thesis-oriented taxonomy that helps to understand the design space and position the contributions of later chapters. This chapter makes the following contributions:

- A review of existing surveys related to microservices, orchestration, and cloud resource management, and a clear identification of the gap for *dynamics-aware* mi-

croservice management.

- Background on cloud-native control and observability primitives (orchestration, autoscaling, service meshes, and telemetry stacks) that support modern adaptive management.
- A taxonomy along four primary dimensions (D1–D4) and two orthogonal dimensions (objectives and telemetry), together with a classification of representative research systems, tools, and evaluation methodologies.
- A positioning of the thesis contributions: EN-Beats (Chapter 3), iDynamics (Chapter 4), TraDE (Chapter 5), and AdaScale (Chapter 6).

The remainder of this chapter is organized as follows. Section 2.2 reviews existing surveys in the cloud-native management stack. Section 2.3 introduces background on microservices and types of dynamics. Section 2.4 presents our taxonomy and classification. Section 2.5 summarizes research gaps. Finally, Section 2.6 concludes the chapter.

2.2 Existing Surveys and Limitations

A substantial number of surveys cover parts of the cloud-native management stack, including Kubernetes scheduling, container orchestration, auto-scaling, serverless platforms, and cloud resource management. Examples include surveys on Kubernetes scheduling and orchestration [71–73], containerized microservice resource-management frameworks [20, 74], machine-learning-based resource management [75, 76], performance-aware cloud resource management [77], and emerging topics such as carbon-aware management [78]. In parallel, surveys in adjacent areas (e.g., serverless computing) provide complementary perspectives on elasticity and multi-tenant platforms [79, 80].

2.2.1 What is missing for dynamics-aware microservice management?

While these surveys are valuable, three recurring limitations motivate a dedicated taxonomy for dynamics-aware microservice management.

(Limitation 1) Dynamics are often simplified or treated implicitly. Many surveys focus on mechanisms (e.g., schedulers or auto-scalers) without explicitly modeling the *types* of dynamics they address (workload, call-graph, network, interference) or how these dynamics interact. For instance, Kubernetes scheduling surveys [71–73] typically describe scheduling policies and extensibility points, but provide limited structure for classifying *dynamic* network and application-dependency effects.

(Limitation 2) Cross-layer interactions are under-emphasized. Microservice management spans multiple abstraction levels: host/VM, container/pod, service, and application call graph. Surveys that focus on a single layer (e.g., the Kubernetes scheduler) often ignore cross-layer feedback loops such as "autoscaler $\leftrightarrow$ scheduler" coupling or "traffic routing $\leftrightarrow$ placement" coupling [33, 67, 81, 82].

(Limitation 3) Evaluation realism and reproducibility remain fragmented. Many papers evaluate on synthetic workloads or simplified simulators, while production deployments increasingly depend on realistic request-path and orchestration behavior [2, 4, 16, 68, 69]. Recent work highlights the need for *Kubernetes-in-the-loop* simulation/emulation to bridge the realism gap [55, 56, 70, 83].

2.2.2 Survey comparison

Table 2.1 summarizes representative surveys and the aspects they emphasize. In contrast, this chapter explicitly structures the literature around (i) *what level* is controlled, (ii) *which dynamics* are modeled, (iii) *how adaptation* is performed, and (iv) *how systems are evaluated*.

2.3 Background: Microservices and Multi-Level Dynamics

2.3.1 Cloud microservices and orchestration platforms

Microservices are typically deployed as containers and managed by an orchestration platform that provides: (i) resource abstraction (pods, services, namespaces), (ii) placement and scheduling, (iii) elastic scaling, and (iv) fault recovery and rollout primitives

Survey	Primary focus	Typical limitations
[71–73]	Kubernetes scheduling and orchestration	Limited explicit treatment of call-graph dynamics and end-to-end SLOs; cross-layer coupling often implicit.
[20, 74]	Resource management for (shared) microservices	Emphasis on mechanisms; taxonomy for dynamics and evaluation environments less systematic.
[75, 76]	ML-based resource management/orchestration	Often technology-centric; less focus on microservice dependency graphs and network dynamics.
[77]	Performance-aware cloud resource management	Broad cloud view; microservice-specific request paths and service meshes not central.
[78]	Carbon/energy-aware management	Focus on sustainability objectives; microservice-specific dynamics and orchestration constraints less detailed.
[79, 80]	Serverless platforms and elasticity	Different abstraction (functions); insights useful but not directly addressing microservice call graphs.
[55]	Kubernetes-in-the-loop simulation/tools	Strong on evaluation realism; does not provide a full taxonomy of dynamics and adaptation mechanisms.
This chapter	Dynamics-aware microservice management	Explicit taxonomy across D1–D4 plus objectives and telemetry; unified view spanning modelling, control, and evaluation.

Table 2.1: Representative surveys and their limitations from the perspective of dynamics-aware microservice management.

[11, 12]. Kubernetes, inspired by earlier cluster managers such as Borg [11, 13], has become the dominant orchestrator for microservice operation, including in hybrid cloud-edge settings [14, 84, 85].

Autoscaling and control loops. At the pod level, Kubernetes provides reactive horizontal scaling (HPA) [21]. In practice, production deployments often use additional controllers: vertical scaling (VPA) to adjust resource requests and limits [22], cluster autoscaling to add or remove nodes [23, 63], and event-driven scaling to handle bursty or queue-based workloads [24, 25]. This naturally leads to *multi-loop* control: scaling decisions interact with placement/scheduling and can create oscillation or delayed convergence if not coordinated [26, 27].

Service meshes and traffic management. Service meshes (e.g., Istio/Envoy/Link-

erd) insert sidecar proxies to provide uniform routing, telemetry, and policy enforcement [42–44]. They enable fine-grained traffic shifting, retries, and circuit breaking, which can be leveraged for SLO management or combined with placement strategies [33, 81, 82, 86].

Observability pipelines. Microservice operation depends on telemetry to diagnose and adapt to changing conditions. Metrics systems (e.g., Prometheus) [38] provide low-cost time-series signals; distributed tracing systems (e.g., Dapper/Jaeger/Zipkin) [36, 39, 40] reveal request paths and latency breakdowns; dashboards (Grafana) [41] support exploratory analysis; and OpenTelemetry provides vendor-neutral instrumentation and export for metrics/logs/traces [37]. Observability signals are therefore both an *input* to controllers and a *measurement* channel for evaluation (e.g., SLO violations).

2.3.2 Types of dynamics

We use the term *dynamics* to denote time-varying behaviors that materially impact microservice performance, cost, or reliability. Dynamics can originate from (i) external demand, (ii) internal software evolution, (iii) shared hardware/software resources, and (iv) the network and infrastructure fabric. Below we summarize common classes of dynamics that appear across the literature:

- **Workload and resource-usage dynamics.** Cloud workloads are well known to be bursty and heavy-tailed, with non-stationary patterns across seconds to days [3, 9, 87, 88]. Workload prediction is therefore widely used in proactive scaling and capacity planning. Classic statistical methods (e.g., ARIMA) [45–47, 89] are still competitive for certain regimes, while deep learning approaches (LSTM/TCN/Transformer variants) improve robustness under non-linearity and long-range dependencies [48, 49, 90–94]. In microservices, workload dynamics propagate through the dependency graph: an upstream burst can amplify downstream contention, causing tail-latency spikes [8, 30, 95].

- **Application, call-graph, and communication-mode dynamics.** Unlike monoliths, microservices often change their request paths due to feature rollout, canary releases, A/B testing, and adaptive routing [29–31, 64, 96]. These changes lead to *call-*

graph dynamics (which services are involved and how frequently) and *communication-mode dynamics* (e.g., synchronous RPC vs. async messaging, fan-out patterns). Call-graph and dependency inference also underpins debugging and incident management [97–99].

- **Network and traffic dynamics.** Network variability includes bandwidth contention, queuing in shared switches, cross-rack latency, and dynamic routing. Such effects become more visible as microservices communicate frequently and may be distributed across zones or edge sites [32–35, 66, 100–102]. Recent work explores joint scheduling and routing/traffic engineering to manage network dynamics and improve SLOs [67, 81, 82].

- **Resource contention, co-location, and interference.** Microservices are commonly *shared* across tenants or co-located with batch jobs, leading to interference on CPU, memory, storage, and accelerators [59–62, 103–106]. Both system-level mechanisms (isolation, bandwidth control) and scheduler-level policies are used to mitigate contention [26, 27, 107, 108]. At the microservice layer, specialized resource managers aim to improve utilization while meeting SLA/SLO constraints [18–20, 28, 109, 110].

- **Additional dynamics: failures and sustainability.** Dynamic environments also include failure events (node failures, stragglers, overload, QoS degradation) [62, 65, 97], and sustainability-related variability such as time-varying carbon intensity or energy budgets [78, 111, 112]. These dynamics motivate controllers that incorporate robustness, risk, and long-term objectives beyond short-term latency.

2.3.3 Benchmarks, traces, and tooling for evaluating dynamics

A recurring challenge in dynamics-aware management is evaluation: synthetic workloads are easy to reproduce but often miss microservice coupling, network effects, or orchestration behavior. Table 2.2 summarizes commonly used benchmarks and traces and what types of dynamics they help reproduce.

Benchmark/trace	Type	Dynamics covered
DeathStarBench [2]	benchmark suite	realistic microservice request paths; tail-latency sensitivity; resource contention.
μBench [16]	benchmark suite	microservice scaling/placement scenarios; configurable bottlenecks.
TrainTicket [68]	benchmark app	realistic service graph and RPC patterns; supports load testing for scaling studies.
Alibaba traces [4, 87]	production traces	non-stationary workload; co-location and interference; cluster-scale patterns.
Meta traces [69]	production traces	microservice-level request patterns and dependencies (trace-driven evaluation).
Bitbrains [3]	cloud traces	VM-level workload and resource-usage dynamics; used for forecasting baselines.
wrk2 [113]	load generator	controlled workload bursts; used in real-cluster experiments.
Network emulation [56, 83]	emulation	time-varying network delay/bandwidth; fault injection.
K8s-in-the-loop [55]	simulation	simulated events; coupling among schedulers/scalers.
iDynamics [70]	evaluation framework	authentic orchestration control; emulated network; controllable dynamics; coupling among scheduling policies.

Table 2.2: Representative benchmarks/traces/tools for evaluating microservices under dynamics.

2.4 Taxonomy of Dynamics-Aware Cloud and Microservice Management

2.4.1 Design rationale

The goal of our taxonomy is to provide a *multi-dimensional* structure that helps compare prior work, expose under-explored regions in the design space, and connect the literature to the thesis contributions. We follow two principles, including **(i): Explicitly separate *dynamics* from *mechanisms*.** In practice, the same mechanism (e.g., scaling) can be used for different dynamics (workload vs. call-graph), and the same dynamics (e.g., interference) can be handled by different mechanisms (isolation vs. scheduling). Our taxonomy therefore treats dynamics type (D2) and adaptation mechanism (D3) as separate axes.**(ii): Treat evaluation environment as a first-class dimension.** Because orchestration and network dynamics are hard to reproduce, evaluation choices (real clus-

ter vs. simulation vs. emulation; synthetic vs. trace-driven workloads) significantly impact the credibility and transferability of results [55, 56, 70, 83]. We therefore include the deployment/evaluation environment (D4) as a primary dimension, rather than an afterthought. Specific taxonomies and relations between D1, D2, D3, and D4 dynamics dimensions can be found in Figure 2.1.

2.4.2 Taxonomy overview

The four primary dimensions (D1–D4) in Figure 2.1 form the main classification used throughout this chapter. They capture (i) *where* control decisions are applied in the system stack, (ii) *what* dynamics are explicitly modeled, (iii) *how* adaptation is performed (models, controllers, and actuators), and (iv) *how* proposed systems are evaluated. In addition, two *orthogonal dimensions* repeatedly shape design choices across all dimensions: *objectives/SLOs* (e.g., tail-latency, cost, energy, reliability) and *telemetry* (metrics/logs/-traces), which together determine what is observable and what can be optimized.

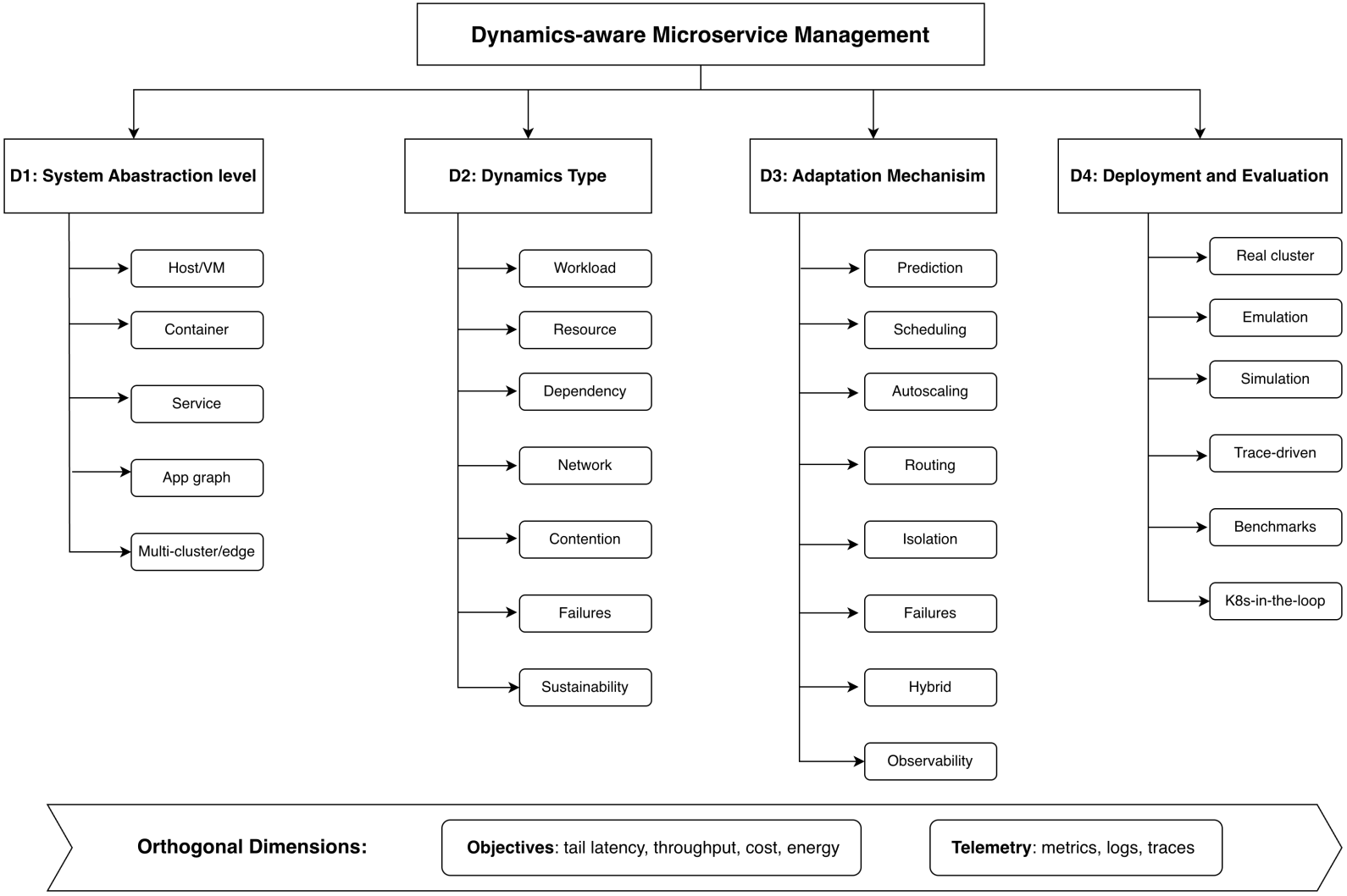


Figure 2.1: Taxonomy for dynamics-aware microservice management (D1–D4) and two orthogonal dimensions (objectives and telemetry).

2.4.3 Dimension 1: system abstraction level and deployment scope

Adaptive management decisions can be applied at different abstraction levels (not mutually exclusive). We distinguish the following common levels.

Host/VM level. Controls at this level manage physical servers or VMs, typically through capacity provisioning, consolidation, and performance isolation. Representative mechanisms include interference and QoS managers for co-located workloads and isolation techniques (e.g., Bubble-Up/Bubble-Flux, PARTIES, CPI^2, Heracles, PIM-Cloud, and Dirigent) [27, 59, 60, 103–105, 114], as well as VM-level anomaly prediction and prevention systems (e.g., PREPARE) [115]. Workload and resource prediction is frequently used to enable proactive provisioning and reduce reaction lag [89, 91, 92, 94, 116, 117]. Thus, host/VM control is significant for cloud providers (no application changes) but often only influences end-to-end SLOs indirectly, motivating coordination with higher-layer policies.

Container and cluster-control level. This level includes pod container placement and scheduling, cluster autoscaling and node provisioning, and coordination among Kubernetes controllers. Research spans scheduler extensions and policy designs (including network-aware and service-mesh-informed scheduling) [32, 34, 35, 66, 118], QoS-aware and heterogeneity-aware resource management inspired by large-scale cluster managers [13, 65, 119, 120], and survey work consolidating the Kubernetes scheduling design space [72]. In practice, widely used primitives such as the Cluster Autoscaler interact with scheduling and scaling decisions [23]. Therefore, cluster-level control exposes powerful actuators (placement, node scale-out/in), but its effectiveness depends on accurate models of the network and interference, and on careful multi-controller coordination.

Service level. Controls are applied per microservice (or per replica), often targeting SLO/SLA satisfaction with improved utilization. Representative systems include FIRM, Sinan, GrandSLAm, and ERMS [18, 19, 28, 109], and recent designs for shared microservices and scalable SLA management [20, 74, 110]. Beyond "hard" resources (CPU/memory), some approaches adapt *soft resources* such as concurrency limits (e.g., ConAdapter) [121], and multifaceted RL scaling frameworks (e.g., CoScal) combine several knobs [122]. Safe learning-based designs aim to reduce exploration risk under rare

events [52]. Thus, per-service control offers fine-grained control but must account for dependency structure (D2 call graph dynamics) to avoid local decisions that harm end-to-end latency.

Application-graph level. Here, the unit of management is a *request path* or dependency graph, and the goal is end-to-end performance across services. Trace studies and datasets characterize dependency structures and workflow variability in production microservice deployments [4, 69, 123]. Systems such as Parslo and ChainsFormer explicitly model microservice graphs to allocate SLO budgets or scale critical chains [29, 31], while Sage and Seer leverage telemetry to diagnose dependency-induced QoS issues [30, 97]. Graph-aware service placement and routing further combine dependency and network effects [64, 67, 124], and graph generation supports scalable what-if evaluation [96]. Therefore, graph-level control is the most "SLO-aligned" abstraction, but depends heavily on high-quality tracing and telemetry and can be sensitive to partial observability.

Multi-cluster and geo-distributed edges. Some works consider microservices spanning multiple clusters or edge sites, where placement interacts with WAN dynamics, mobility, and failures. This includes cloud–edge deployment and migration [14, 15, 125, 126], reliability-aware fog and IoT placement [127, 128], and Kubernetes-oriented edge scheduling under dynamic network conditions [85, 124]. Thus, multi-cluster and geo-distributed edge settings introduce additional dynamics (WAN latency, partitions, churn) and amplify the need for realistic evaluation (D4).

Takeaway of (D1). D1 captures *where* the control loop is closed. Moving upward (host $\rightarrow$ graph) increases SLO alignment and semantic awareness, but typically requires richer telemetry and stronger cross-layer coordination.

2.4.4 Dimension 2: dynamics type

Dynamics-aware management systems differ primarily in *which* dynamics they explicitly represent and respond to.

Workload and resource-usage dynamics. Systems model arrival rates, request mix, and multi-resource consumption that vary over time. Many proactive approaches fore-

cast workload or resource usage using statistical and learning-based models [89–92, 94, 116], and translate predictions into provisioning or scaling decisions. Complementarily, analytical and hybrid performance models (e.g., queueing-based predictors) connect demand to latency and can support SLO planning [64, 95, 115]. These dynamics are often non-stationary (bursts, diurnal cycles, workload shifts), which motivates robust online adaptation rather than static tuning.

Call-graph and dependency dynamics. Microservice request workflows and critical paths can change due to feature rollouts, autoscaling, failures, or traffic shifts. Trace studies provide empirical evidence of dependency variability and performance heterogeneity in large deployments [4, 69, 123], motivating methods that explicitly model microservice graphs. Parslo and ChainsFormer incorporate the dependency structure into SLO allocation and scaling decisions [29, 31], while diagnosis systems such as Sage and Seer leverage telemetry to reason about dependency-induced QoS violations [30, 97]. At the actuation layer, deployment and routing designs adapt to changing dependencies and critical paths [64, 67], and graph generators help evaluate controllers under diverse evolving workflows [96].

Network and traffic dynamics. Systems model time-varying network states (such as cross-node latency and bandwidth) and their impact on microservice performance. Classic datacenter transport and bandwidth control mechanisms address flow-level dynamics [100–102], while recent cloud-native work integrates network signals into orchestration and scheduling [32, 34, 35, 66, 85]. On the traffic-management side, load balancers and service meshes enable traffic shifting and collaborative routing [42, 43, 81, 82, 86]. Joint placement-routing optimization explicitly couples network dynamics with dependency structure [33, 64, 67].

Contention and interference dynamics. Systems address noisy-neighbor and colocation effects that vary with other workloads sharing CPU, memory, cache, I/O, and network resources. Representative mechanisms include interference detection and prediction, resource shaping, and isolation in shared clusters [27, 59, 60, 103–106], as well as heterogeneity- and interference-aware cluster management [119, 120]. Public-cloud measurement studies reveal how such interference appears as QoS degradation at scale [62], and microservice-specific managers incorporate per-service latency sensitivity [19,

110]. A key observation is that interference often manifests primarily in tail latency, making SLO-aware control essential.

Failure and QoS degradation dynamics. Some systems explicitly model failures, stragglers, or degradation events, linking them to mitigation actions such as reconfiguration or migration. Large-scale cluster managers incorporate failure recovery and blacklisting mechanisms [13, 65], and proactive anomaly systems predict degradation and trigger preventative actions [97, 115]. In microservices, debugging studies and benchmarks characterize common failure and debug patterns [68], and reliability-aware placement in fog and edge treats failures and partitions as first-class dynamics [85, 124, 127].

Sustainability dynamics. Time-varying energy budgets, power caps, renewable availability, or (increasingly) carbon intensity appear as system objectives and model constraints. Energy-aware autoscaling and VM allocation trade off performance and power [111], and power management for latency-critical services illustrates how energy control interacts with tail latency [112, 129]. Geo-distributed workload management can exploit green-energy heterogeneity [130], while recent survey work highlights carbon-aware resource management as an emerging direction for latency-sensitive clouds [78].

Takeaway of (D2). Most systems model one or two dynamics explicitly. However, handling *coupled* dynamics (e.g., call-graph + traffic + interference) is significantly harder due to partial observability and interacting controllers, motivating the multi-dynamics focus of this thesis.

2.4.5 Dimension 3: adaptation mechanism

Adaptation mechanisms implement how a system *senses*, *decides*, and *acts*. Most designs can be viewed through an MAPE-K or autonomic-computing lens [57], but they differ in the modelling formalism, optimization type, and available actuators.

Prediction and performance modelling. Systems build models to forecast demand, resource needs, or latency, then use these models to guide proactive decisions. Forecasting methods range from statistical and classical time-series models to deep learning predictors [89–92, 94, 116], while performance prediction can rely on analytical/hybrid models such as queueing-based approaches [64, 95, 115]. *Practical implication:* model

accuracy and robustness under distribution shift directly determine whether proactive control is beneficial or harmful.

Scheduling, placement, and migration. Systems optimize where microservices run, taking into account system resource capacity, interference, reliability, and network costs. This includes Kubernetes scheduler extensions and network-aware scheduling [32, 34, 35, 66, 118], reliability-aware fog/edge placement [124, 127, 128], cloud–edge deployment and migration [14, 15, 125, 126], and cost-efficient deployment formulations [53]. *Practical implication:* placement changes can also reshape call graphs and traffic patterns, creating feedback loops that must be accounted for.

Elastic scaling. Scaling adapts the number of replicas and/or the resources allocated to each replica. Reactive autoscaling is common in practice (e.g., HPA and VPA) [21, 22], while research explores proactive scaling driven by prediction and modelling [94, 115, 116]. Learning-based scaling and adaptation apply to both hard resources (e.g., per-service replicas and resources) and soft resources (e.g., concurrency) [52, 121, 122]. Cluster-level scaling with node provisioning interacts with pod scheduling and service-level scaling, often via the Cluster Autoscaler [23]. *Practical implication:* scaling policies are prone to oscillations under multiple interacting controllers, which motivates stability-aware designs.

Traffic management and routing. Service meshes and load balancers enable traffic shifting, request steering, and collaborative routing. Research explores traffic-aware optimization for replicated microservices [33], traffic-aware load balancing and routing mechanisms [81, 82, 86], and joint placement-routing under network and call-graph dynamics [64, 67]. Industry service-mesh stacks (e.g., Istio/Envoy) provide the operational primitives for such traffic control [42, 43]. *Practical implication:* traffic shifting can mitigate hotspots quickly, but can also amplify instability if combined naively with autoscaling and rescheduling.

Isolation and resource control. Mechanisms such as CPU isolation, admission control, and bandwidth allocation mitigate interference and network dynamics. Representative approaches include co-location management and CPU isolation in shared clusters [59, 60, 103–106], and datacenter network control for bandwidth isolation and deadline-aware transport [100–102, 107]. *Practical implication:* isolation is often the "safety layer"

that prevents tail-latency blowups when predictions or policies are wrong.

Learning-based and hybrid controllers. Reinforcement learning and hybrid approaches are increasingly used to handle complex dynamics and multi-objective trade-offs, including microservice scaling and safe RL variants [52, 112, 121, 122]. Beyond microservices, learning-based cluster schedulers for ML workloads highlight challenges of stability, fairness, and robust generalization [131–133]. *Practical implication:* learning-based control must confront safety, exploration cost, and non-stationarity in real clusters.

Observability-driven diagnosis and remediation. Systems use telemetry to detect anomalies, infer dependencies, and guide remediation actions. Metrics tooling (e.g., Prometheus) and service-mesh telemetry support online monitoring and anomaly signals [32, 38, 134], while tracing frameworks enable request-path and critical-path latency breakdowns [36, 37, 39, 40]. Representative diagnosis and AIOps-style systems leverage this telemetry to localize incidents and performance bottlenecks [30, 97–99]. *Practical implication:* converting telemetry into actionable *causal* control signals remains difficult under sampling, noise, and changing call graphs.

Takeaway of (D3). D3 captures *how* adaptation is realized. The dominant challenge is not the availability of knobs, but ensuring that multi-knob control remains stable and SLO-effective under partial observability.

2.4.6 Dimension 4: deployment and evaluation environment

The choice of evaluation methodology significantly influences the outcomes in dynamics-aware management, as achieving realism, controllability, and reproducibility often involves trade-offs.

Real clusters and production traces. Evaluations on real Kubernetes clusters (including public clouds) provide realism but are hard to reproduce. Many works therefore combine real deployments with trace-driven workloads and production traces, including microservice trace releases and workflow traces [4, 5, 69], and large-scale cluster traces [9, 87]. *Practical implication:* traces capture realistic dynamics but can omit key context (e.g., hidden policies, missing telemetry), motivating careful experimental design.

Microservice benchmarks and workload generators. Benchmark suites such as DeathStarBench and μBench provide realistic microservice behavior [2, 16], and microservice fault and debug benchmarks (e.g., TrainTicket) enable systematic failure studies [68]. Tools such as `wrk2` support load generation [113], and call graph generators support controlled variation of dependency structures [96, 123]. *Practical implication:* benchmarks improve reproducibility but may under-represent production call-graph churn and multi-tenant interference.

Simulation and emulation. Simulation improves scalability and repeatability but can lose fidelity if orchestration and network effects are simplified. Network emulation frameworks (e.g., Kollaps, Thunderstorm) help capture network dynamics [56, 83], and cloud simulators support broader what-if studies [54, 135]. *Practical implication:* emulation and simulation are often necessary to explore large design spaces, but must be validated carefully against real-cluster behavior.

Kubernetes-in-the-loop evaluation. Recent tools integrate authentic orchestration into simulation [55] and emulation [70] to improve realism. This direction is particularly relevant for evaluating cross-layer controllers where scheduling, autoscaling, and traffic management interact. *Practical implication:* "in-the-loop" evaluation is a promising compromise with realistic control-plane behavior with repeatability and scalability.

Takeaway of (D4). D4 determines what conclusions are trustworthy: many performance gains may disappear when realistic orchestration, network dynamics, and interference are introduced.

2.4.7 Orthogonal dimensions: objectives and telemetry

Although not treated as primary dimensions in our classification table, objectives and telemetry repeatedly shape the design of dynamics-aware systems.

Objectives and SLO models. Most microservice management is SLO-driven, with tail latency and throughput as dominant objectives [2, 8, 58]. Cloud Systems (such as FIRM [18], ERMS [19], DERM [110])target SLO/SLA assurance while improving utilization, and cost-aware deployment and consolidation introduce explicit cost–performance trade-offs [20, 53]. Sustainability and energy objectives are increasingly considered,

e.g., energy-efficient autoscaling and power management [111, 112, 129], and emerging carbon-aware resource management directions [78]. A recurring concern is that objectives overlap with trade-offs. For instance, minimizing latency can increase cost or energy, while aggressive consolidation can exacerbate interference and tail latency.

Telemetry signals. The available telemetry constrains what dynamics can be detected and controlled. Metrics systems such as Prometheus provide time-series signals used in autoscaling and anomaly detection [38], and service-mesh telemetry can expose fine-grained network interaction signals useful for scheduling and co-location [32, 134]. Tracing frameworks reveal request paths and critical-path latency breakdowns [36, 37, 39, 40], enabling call-graph-aware debugging and dependency-aware control [29–31, 97]. However, telemetry is noisy, sampled, and incomplete, which means converting it into robust causal signals remains a core challenge [30, 97, 99].

2.4.8 Taxonomy overview

The four primary dimensions (D1–D4) in Figure 2.1 form the main classification used throughout this chapter. Two orthogonal dimensions frequently appear in the related literature works, including *objectives* (e.g., latency, cost, carbon) and *telemetry* (metrics/logs/traces), which influence both modelling and control.

Table 2.3: Classification of representative dynamics-aware microservice management works.

Work	Level	Dynamics	Mechanism & evaluation
Prediction and modelling for dynamic workloads			
EN-Beats [94]	VM / host	workload, resource usage	deep time-series forecasting for proactive decisions; evaluated on real traces (e.g., Bitbrains/Alibaba)
CloudInsight [116]	VM / host	workload	workload forecasting for cloud workloads; trace-driven evaluation
DeepTCN [91]	VM / host	workload	temporal convolutional forecasting; evaluated on public traces
esDNN [92]	VM / host	workload	ensemble deep learning for workload prediction; trace-driven evaluation
Resource Central [136]	cluster	resource usage	predicts resource demand in large-scale clusters for provisioning/planning; production-scale study
PREPARE [115]	service / app	workload, latency	performance prediction + planning; analytical/hybrid model; prototype + trace-driven evaluation

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
Forecasting model baselines [1, 45–49, 51, 90, 93, 137]	VM / host	workload	LSTM/N-BEATS/Transformer/GNN and ARIMA baselines commonly used across proactive management studies
Efficient autoscaling via prediction [47]	service / cluster	workload	prediction-driven autoscaling policy; evaluated with cloud workloads
Industry autoscaling and orchestration primitives			
Kubernetes HPA [21]	cluster / service	workload	reactive horizontal autoscaling (metrics-driven); widely deployed in production
Kubernetes VPA [22]	cluster / service	resource usage	vertical right-sizing of CPU/memory requests; production tool with controller interactions
Cluster Autoscaler [23]	cluster	workload, capacity	node-scale out/in based on pending pods; interacts with scheduling and HPA/VPA
Karpenter [63]	cluster	capacity, heterogeneity	dynamic node provisioning with flexible instance selection; production-oriented autoscaling

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
KEDA [24]	cluster / service	event-driven workload	event-driven autoscaling for Kubernetes; supports queue/stream triggers
Knative autoscaling [25]	serverless / service	bursty workload	scale-to-zero and rapid scale-out for request-driven services; production framework
Call-graph, dependency, and workflow dynamics			
Sage [30]	service / app graph	call graph, latency	ML-driven performance debugging and diagnosis; trace/metric-driven; focuses on end-to-end SLOs
Parslo [29]	service / app graph	call graph, workload	infers service-level performance under dependency coupling; evaluated with microservice workloads
ChainFormer [31]	service / app graph	call graph	dependency modelling for chain-aware management; evaluated on microservice benchmarks
Microservice dependency characterization [123]	app graph	dependency, heterogeneity	characterizes dependency structure and performance variability in production microservices; informs graph-aware control

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
Task-dependency synthesis [138]	workflow / app	dependency	characterizes and synthesizes task-dependency graphs for what-if analysis and scheduler evaluation
Microservice graph generation [17]	app graph	call graph	generates realistic microservice graphs with production characteristics; enables scalable controller testing
Production trace releases [4, 5, 69]	datasets	call graph, workload	distributed trace datasets for microservice analysis and evaluation (dependency + performance dynamics)
Placement, scheduling, and network/edge-aware management			
Borg [13]	cluster	workload, failures	large-scale cluster management (scheduling, admission, recovery); production system
Fuxi [65]	cluster	workload, failures	fault-tolerant resource management and scheduling in Alibaba clusters; production-scale evaluation

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
Mercury [139]	cluster	workload	hybrid centralized/distributed scheduling for shared clusters; production-oriented evaluation
Quasar [119]	cluster	interference, workload	QoS-aware scheduling for heterogeneous workloads; mitigates interference via classification and placement
Paragon [120]	cluster	heterogeneity, interference	QoS-aware scheduling across heterogeneous machines; performance isolation via placement decisions
NetMARKS [32]	cluster	network, traffic	network-metrics-aware Kubernetes scheduling; prototype + experiments
Diktyo (K8s) [66]	cluster	network, traffic	Kubernetes network-aware scheduling; evaluates traffic locality and latency
Extending K8s networking [34, 35]	cluster	network, traffic	network-aware placement/scheduling extensions for Kubernetes; experimental evaluation

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
Interference-aware K8s placement [26]	cluster	interference	microservice placement with interference awareness in Kubernetes; experimental evaluation
Cost-aware microservice placement [53, 118]	cluster	cost, workload	placement policies/formulations for cost optimization under dynamic demand; evaluated via experiments/simulation
OptTraffic [33]	cluster / app	traffic	traffic-aware scheduling/optimization for replicated microservices; cluster evaluation
TraDE [67]	app graph / cluster	call graph, network	joint deployment and request routing under dynamic traffic and dependencies; evaluated with trace-driven environment
Joint deployment & routing [64]	app graph / cluster	call graph, network	optimizes placement and request routing to meet end-to-end objectives; trace/benchmark-driven evaluation
Reliability-aware placement [124, 127]	cluster / edge	network, failures	reliability modelling + placement for microservices/IoT; fog/edge evaluation

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
Cloud–edge placement [14, 15]	edge / multi-cluster	mobility, network	adaptive placement across cloud and edge under WAN/mobility dynamics; simulation/testbed evaluation
Delay-aware migration [125, 126]	edge	mobility, latency	probabilistic or delay-aware migration/scheduling for MEC scenarios; experimental/simulation evaluation
Latency-aware industrial fog orchestration [85]	edge / fog	WAN latency, load	Kubernetes-oriented fog orchestration under dynamic network conditions; prototype evaluation
Survey: custom K8s scheduling [71–73]	survey	—	taxonomy of Kubernetes scheduling/orchestration techniques; summarizes open issues and extensions
Traffic management, service meshes, and collaborative routing			
Concury [81]	service / network	traffic	software load balancing for cloud applications; evaluated with real workloads
Collaborative mesh orchestration [82, 86]	service mesh	traffic, latency	probabilistic/collaborative orchestration across mesh instances; large-scale evaluation

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
Istio / Envoy / Linkerd [42–44]	service mesh	traffic, failures	industry meshes enabling traffic shifting, retries, and policy control; widely deployed
Service mesh performance study [140]	service mesh	latency overhead	quantifies mesh overheads and implications for SLO management; empirical evaluation
Interference-aware and SLO/SLA-aware resource management			
Heracles [114]	host / service	contention, workload	improves efficiency for latency-critical services via resource management; production-style evaluation
FIRM [18]	service	contention, workload	SLA-aware resource management for microservices; cluster prototype evaluation
ERMS [19]	service	contention, workload	SLA-guaranteed management for shared microservices; cluster experiments
Sinan [28]	service	contention, workload	bottleneck detection + resource adjustment; evaluated with microservice workloads
GrandSLAm [109]	service	workload, latency	SLA management via resource adaptation; evaluation on microservices

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
DERM [110]	service / host	interference	microservice resource management with interference awareness; experimental evaluation
Shared microservice management [20, 74]	service	contention	designs for shared microservices and scalable SLA management; prototype + survey evidence
Clite [106, 141]	host / service	contention	QoS-aware co-location for latency-critical services; evaluated on cluster workloads
PIMCloud [104]	host	contention	QoS-aware management for latency-critical jobs; evaluated on production-like setup
PARTIES [105]	host	contention	QoS-aware resource allocation for co-located workloads; cluster evaluation
Bubble-Up / Bubble-Flux [59, 60]	host	contention	online QoS management for co-location; evaluated in warehouse-scale settings
CPI^2 [103]	host	contention	CPU performance isolation for shared compute clusters; experimental evaluation

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
PerfIso [142]	host	contention	performance isolation for commercial latency-sensitive clouds; production-inspired evaluation
MemGuard [143]	host	memory contention	memory bandwidth reservation/isolation to reduce interference; experimental evaluation
Cross-core interference prediction [144]	host	contention	predicts cross-core interference for consolidation decisions; empirical evaluation
DeepDive [61]	VM / host	contention	identifies/manages performance interference in virtualized environments; evaluation on shared setups
Co-location characterization [87, 145]	production trace	contention, workload	characterizes co-located workloads and interference in large clusters; informs interference-aware control
Cloud QoS degradation [62]	public cloud	degradation	detects/estimates QoS degradation in public clouds; measurement-driven evaluation
Dirigent [27]	cluster	workload, contention	QoS-aware management for latency-critical workloads; evaluated in shared clusters

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
Co-locating containerized workloads [134]	cluster	contention, traffic	service-mesh-informed co-location for container workloads; experimental evaluation
Network isolation and transport mechanisms			
D2TCP [100]	network/transport	traffic, deadlines	deadline-aware datacenter transport; evaluated in datacenter settings
Fastpass [101]	network/transport	traffic	centralized scheduling for datacenter networks; experimental evaluation
EyeQ [107]	network	contention	practical network performance isolation for multi-tenant clouds; experimental evaluation
Preemptive flow scheduling [102]	network	traffic	finishing flows quickly via preemptive scheduling (bandwidth control); evaluated in datacenter networks
Sustainability: energy/carbon-aware management			
Carbon-aware computing [146]	datacenter	carbon intensity	models carbon signals and motivates carbon-aware control and scheduling in datacenters

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
Carbon-aware scheduling survey [78, 147]	survey	carbon, energy	taxonomy/review of carbon-aware scheduling and resource management for cloud/edge
Energy-aware autoscaling [111]	VM / cluster	workload, energy	proactive autoscaling balancing performance and energy; evaluated via experiments/simulation
Rubik [129]	host / service	power, tail latency	power management for latency-critical services; evaluates performance–power trade-offs
Hipster [112]	host / service	workload, energy	hybrid task management for latency-critical workloads with energy considerations; evaluated on hardware platform
Sustainable datacenters [130]	datacenter	energy heterogeneity	heterogeneity-aware workload management for sustainable datacenters; trace-driven evaluation
Learning-based and hybrid controllers			

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
ConAdapter [121]	service / cluster	workload, latency	RL-based adaptive microservice management (e.g., soft/hard resource knobs); benchmark evaluation
CoScal [122]	service / cluster	workload, latency	multifaceted learning-based scaling for microservices; evaluated with microservice benchmarks
Safe RL for microservices [52]	service / edge	workload, delay	safe reinforcement learning for adaptive control under constraints; evaluated on cluster/edge setups
ML-based container orchestration [76]	cluster	workload, contention	survey/techniques for ML-driven orchestration decisions; highlights robustness and stability issues
Pollux [131]	cluster	workload, contention	co-adaptive cluster scheduling for ML jobs (goodput-aware); production-oriented evaluation
ONES [133]	cluster	GPU scheduling	online evolutionary orchestration for distributed ML workloads; highlights stability under dynamics

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
Online job scheduling [132]	cluster	workload	online scheduling for distributed ML; exposes fairness/robustness challenges
Q-learning task scheduling [148]	edge / cloud	workload, energy	Q-learning based dynamic task scheduling; evaluated via simulations
Telemetry-driven diagnosis, debugging, and AIOps			
Seer [97]	service / app	failures, call graph	telemetry-driven detection/diagnosis for complex distributed systems; evaluated on real systems
ART [98]	service / app	incidents	unsupervised incident triage using operational telemetry; evaluated on real incident data
PerfScope [99]	host / service	performance bugs	online performance diagnosis/inference for server systems; telemetry-driven evaluation
Orca [149]	service / app	performance bugs	differential bug localization for large-scale systems; evaluated with production traces

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
Microservice fault/debug benchmark [68]	benchmark	failures	fault analysis and debugging for microservices using benchmark application and injected faults
Evaluation frameworks, benchmarks, traces, simulation, and emulation			
iDynamics [70]	evaluation	multi-dynamics	trace-driven simulation + microservice emulation; supports call-graph and network dynamics
Kubernetes-in-the-loop [55]	evaluation	orchestration	integrates authentic Kubernetes control plane into simulation loop; improves realism
Kollaps [56]	network emulation	network	scalable network emulation for distributed systems; used for microservice evaluation
Thunderstorm [83]	network emulation	network	network emulation for resilience testing; experimental evaluation
CloudSim / CloudNativeSim [54, 135]	simulation	workload	cloud simulation frameworks for what-if studies; widely used in resource management research

Taxonomy classification (continued).

Work	Level	Dynamics	Mechanism & evaluation
DeathStarBench [2]	benchmark	workload, tail latency	open-source microservice benchmark suite for realistic end-to-end latency evaluation
μBench [16]	benchmark	microservice behavior	factory of benchmark microservices; supports configurable microservice patterns and stress tests
wrk2 [113]	workload generator	request rate dynamics	closed/open-loop load generation with accurate request rates; used in microservice performance studies
Production traces [4, 5, 9, 69, 87]	datasets	workload, interference, call graph	public traces for cluster scheduling, colocation/interference, and microservice dependency analysis

2.4.9 Summary of taxonomy and positioning of thesis contributions

Table 2.3 classifies representative research works and tools using the four primary taxonomy dimensions. The table is *not* exhaustive; instead, it covers a broad set of systems that collectively illustrate the design space. We highlight the thesis contributions in bold.

Positioning the thesis contributions. EN-Beats (Chapter 3) provides accurate workload forecasting for proactive decisions (D3: modelling/prediction) under real workload dynamics (D2: workload/resource). iDynamics (Chapter 4) provides an emulation environment (D4) that enables realistic evaluation of call-graph and network dynamics. TraDE (Chapter 5) addresses joint deployment and adaptive scheduling with call-graph and traffic dynamics (D2). AdaScale (Chapter 6) further advances adaptive scaling under diverse system dynamics.

2.5 Research Gaps

This taxonomy highlights both the gaps addressed by this thesis and broader open challenges for *dynamics-aware* microservice management.

2.5.1 Gaps addressed in this thesis

Gap 1: Accurate and robust workload prediction for proactive control. Most production autoscaling remains largely reactive (e.g., HPA/VPA and node autoscalers) [21–23], which can lag behind rapid demand changes and cause transient SLO violations. Prior work has explored predictive management using classical forecasting (e.g., ARIMA), recurrent models (LSTM), and deep time-series predictors [89–92, 116, 136], but prediction accuracy and calibration often degrade under bursty and event-driven traffic, regime shifts, and limited history. Since prediction errors directly enter the control loop, unreliable forecasts can also *destabilize* scaling and provisioning policies. Chapter 3 addresses this gap with EN-Beats [94], improving robustness for proactive decisions under non-stationary workloads.

Gap 2: Joint handling of call-graph and network dynamics. Call-graph-aware systems infer and reason about request dependencies using tracing to allocate SLO bud-

gets, identify critical paths, and guide adaptation [29–31]. In parallel, network-aware scheduling and service-mesh routing optimize placement and traffic steering using network signals [32, 35, 66, 86], and joint deployment–routing formulations have been studied for microservice call graphs [64]. However, most existing designs either (i) assume relatively static dependencies when optimizing network and service placement, or (ii) ignore network dynamics when optimizing for dependency changes. This separation has limitations because deployment changes reshape traffic matrices, which in turn alter latency and the effective critical path. Chapter 5 addresses this gap via TraDE [67] by adaptive rescheduling of microservices under changing call graphs, traffic dynamics, and varying cross-node delays.

Gap 3: Realistic and reproducible evaluation for multi-dynamics. Microservice benchmarks such as DeathStarBench and μBench improve comparability [2, 16], while emulation and simulation tools capture network dynamics at scale [56, 83]. Nevertheless, many studies still evaluate under simplified assumptions about orchestration policies, control-plane timing, and network behavior, or rely on proprietary production environments that are hard to reproduce. Although production traces and datasets are increasingly available [4, 69], they are difficult to replay faithfully without authentic orchestration and controller interactions. Chapter 4 addresses this gap by providing iDynamics [70], enabling reproducible evaluation of *multi-dynamics* (workload, call-graph, and network) with Kubernetes-aware control-plane behavior.

Gap 4: Adaptive scaling under diverse interaction patterns. Many microservice resource managers and autoscalers rely on per-service signals and assume relatively stable interaction patterns [18, 19, 21, 28, 109, 110]. However, real deployments exhibit changing request mixes and dependency structures due to feature updates, traffic shifts, and failure recovery [97, 123], which can move bottlenecks across services and invalidate static scaling heuristics. Learning-based approaches can improve adaptivity but introduce additional stability and safety concerns when deployed in closed loops [121, 122]. Chapter 6 addresses this gap with AdaScale, which targets scaling decisions that remain effective under diverse root request patterns and dependency shifts.

2.5.2 Broader open challenges and future directions

Beyond the thesis scope, several challenges remain under-explored.

(C1) Cross-layer coordination and stability. Modern cloud-native stacks often run multiple control loops concurrently—HPA/VPA, node autoscaling, scheduler re-placement, and service-mesh traffic shifting—operating at different time scales and with different objectives. These controllers can interfere, causing oscillations or abnormal system behaviors [22, 23, 26, 27, 72]. For example, traffic shifting in a service mesh (Istio/Envoy) can change per-service load distribution and trigger autoscaling, while rescheduling and node scaling change locality and feed back into routing decisions [42, 43, 86]. A principled framework for hierarchical coordination (or unified multi-actuator control) with stability guarantees remains an open problem.

(C2) Telemetry-to-control: causality and partial observability. Metrics and traces enable diagnosis and troubleshooting [36–40], and support dependency-aware systems [30, 97]. However, converting telemetry into *actionable* and *causal* signals for controllers is challenging under sampling, noise, missing spans, and changing call graphs [97–99]. Bridging observability and control will likely require causal inference, robust estimation, and controller designs that explicitly reason about uncertainty.

(C3) Multi-objective management (latency–cost–carbon). Many systems optimize tail latency or utilization alone, even though practitioners face explicit cost and sustainability constraints. Cost-efficient deployment and consolidation highlight performance–cost coupling [20, 53], while carbon-aware computing and surveys emphasize time-varying carbon intensity as an emerging constraint [78, 146, 147]. Integrating these objectives into microservice control loops, including power/energy management for latency-critical services, remains an active direction [111, 112, 129].

(C4) Safe learning-based controllers. RL-based approaches promise adaptivity in complex environments but raise safety and robustness issues under rare events, distribution shift, and exploration cost [52, 121, 122]. Related experience from learning-based cluster schedulers for ML workloads highlights additional concerns such as fairness and stability in shared clusters [131–133]. Developing learning-based controllers with verifiable constraints, safe exploration, and reliable offline evaluation remains an open research problem.

(C5) Standardized benchmarks and reproducible evaluation pipelines. While benchmark suites exist [2, 16, 68], reproducing *orchestration* and *network* dynamics, as well as evolving call graphs, remains difficult. Recent directions include generating microservice graphs with production characteristics [17, 123] and integrating authentic orchestration into the evaluation loop [55, 70]. Community-standard "dynamic scenarios" (workload bursts, dependency shifts, network events, interference) and reproducible pipelines that package configs, traces, and controller settings are still needed.

2.6 Summary

This chapter reviewed existing surveys and background on cloud-native orchestration, autoscaling, service meshes, and observability, and then proposed a taxonomy for *dynamics-aware* microservice management. The taxonomy is organized along four primary dimensions (abstraction level, dynamics type, adaptation mechanism, and evaluation environment), with SLO objectives and telemetry as orthogonal dimensions. Using this taxonomy, we classified around one hundred representative research works and widely used tools/frameworks, and identified research gaps and open challenges. The following chapters build on this foundation to develop new predictive models (EN-Beats), realistic evaluation environments (iDynamics), and dynamics-aware deployment and scaling mechanisms (TraDE and AdaScale).

Chapter 3

Understanding and Predicting Dynamic Resource Usage in Cloud Data Center

Cloud data centers host diverse, rapidly fluctuating workloads whose CPU, memory, and network demands are difficult to predict yet critical for proactive resource management. Existing predictors typically target a single metric and treat resources independently, thereby failing to exploit latent cross-metric correlations and limiting their accuracy in dynamic environments. This chapter introduces ENBeats, an ensemble learning-based model for multi-resource usage prediction, coupled with RCorrPolicy, a Spearman-correlation-based metric selection policy that automatically identifies the most informative resource metrics from historical traces. RCorrPolicy filters correlated inputs for ENBeats and can also enhance existing models, while ENBeats aggregates multiple weak learners into a strong learner to jointly forecast CPU utilization, memory usage, and network incoming traffic using production traces. Experimental results show that ENBeats achieves lower prediction error and higher goodness-of-fit than existing baselines. By providing accurate short-term forecasts for multiple resources, ENBeats lays the foundation for more informed autoscaling and scheduling strategies in dynamic computing environments.

3.1 Introduction

Recently, a variety of devices and cloud-supported applications are being used in every aspect of our life. Smart devices, including surveillance cameras, augmented reality hel-

This chapter is derived from:

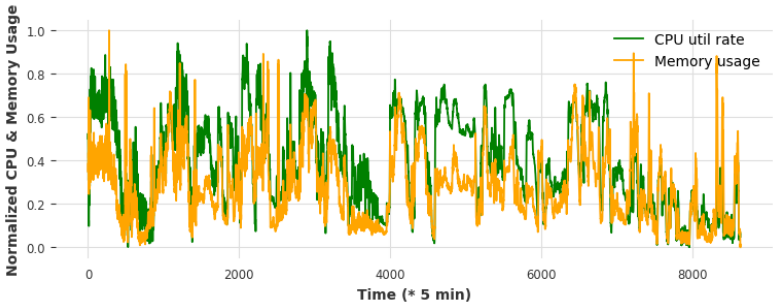
- **Ming Chen**, Maria A. Rodriguez, Patricia Arroba, and Rajkumar Buyya, "EN-Beats: A Novel Ensemble Learning-Based Method for Multiple Resource Predictions in Cloud," *Proceedings of the 16th IEEE International Conference on Cloud Computing (CLOUD)*, Pages: 144-154, Chicago, IL, USA, July 2-8, 2023.

mets, cell phones and autonomous vehicles, are generating massive data, irrespective of structured data or unstructured data [80]. Similarly, there are also arising complex cloud applications (e.g., WeChat, Instagram, Netflix, and TikTok) producing a huge amount of data. The more we use these applications or services, the smarter of these applications we may feel. It is mainly because the application itself can learn from historical data to automate the decision-making process, like the recommendation features of TikTok. Due to such a trend, numerous new devices and complex applications call for a great surge of vibrant cloud services, thus increasing the consumption of these cloud-supported applications. Moreover, during the global COVID-19 pandemic, many companies and government sectors have accelerated the speed of digitization of their business or services, which also invokes the need for surging cloud resource demands and efficient resource management techniques in cloud systems.

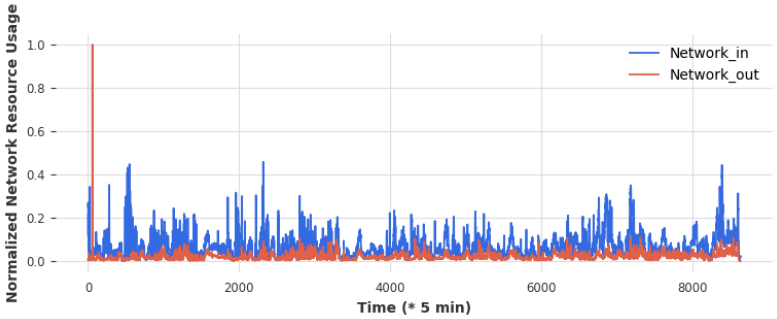
To better utilize and manage cloud resources, efficient resource management strategies in cloud data centers are important. We conducted an in-depth experimental analysis of multiple resource utilization based on open-source `Bitbrains` [3] dataset[1] from a real cloud production environment and we observed that the resource usage fluctuate greatly and demonstrate great uncertainties, as shown in Figure 3.1. Inefficient resource predictions for fluctuating resource consumption bring out challenges like QoS violations due to resource under-subscription, or inefficient resource utilization when oversubscribing too many resources.

To address the problem of how to efficiently predict resource utilization in dynamic cloud computing environments, different approaches are proposed for different scenarios. However, most of these approaches fail to capture the latent correlations among different resource metrics and have limitations on multiple resource predictions. A few of the latest research works, like[150], do consider resource correlations, but their algorithms for correlation calculation are based on `pearson` correlation which assumes two calculated data vectors are uniformly distributed, while some resources in real clouds might not have these patterns. Moreover, the proposed techniques in [150] fail to make multiple resource usage predictions.

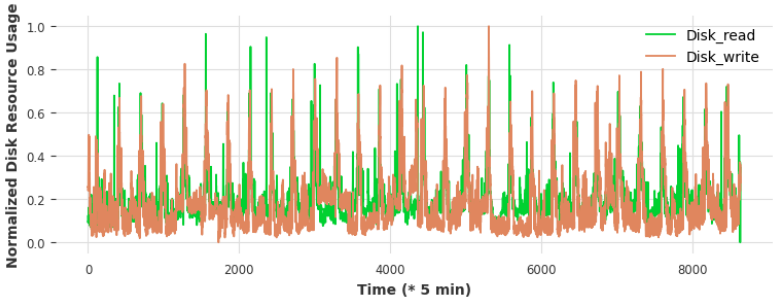
[1]The open-source Bitbrains dataset can be downloaded from http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains



(a) Normalized historical usages of CPU and Memory.



(b) Normalized historical data of network incoming/outgoing traffic.



(c) Normalized historical usages of disk reading and writing.

Figure 3.1: An illustration of normalized average resource usages across the 1,250 Virtual Machines (VMs) from a modern cloud at `5 min` interval index in dataset from `Bitbrains`.

In this chapter, we proposed an efficient ensemble learning-based approach *EN-Beats* along with the resource metric selection policy *RCorrPolicy* for multiple resource metrics predictions in cloud computing environments. *RCorrPolicy* is used for selecting the correlated resource metrics, which are adopted as the inputs to *EN-Beats* model for future resource prediction. The ablation experiments have shown the effectiveness of *RCorrPol-*

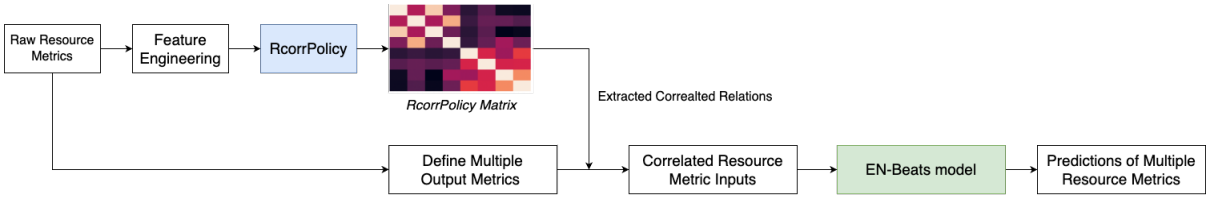


Figure 3.2: The overall workflow of *RCorrPolicy* for selecting resource metrics and *EN-Beats* for resource prediction.

icy and the improvement in workload predictions by *EN-Beats* model. Our contributions can be summarized as:

- We proposed a resource metric selection policy *RCorrPolicy* by choosing the correlated metrics from raw data traces.
- We designed the ablation experiments and evaluated the effectiveness of *RCorrPolicy* by presenting ablation experimental results via widely-used current models.
- We proposed the *EN-Beats* model based on ensemble learning by aggregating multiple weak learners into a strong learner for making multiple resource metric predictions.
- We found that the recently proposed *N-Beats* model is a good candidate for cloud resource usage prediction. To the best of our knowledge, *N-Beats* hasn't been analyzed and adopted in the scenario of cloud resource utilization, which we are able to do.

The remainder of this chapter is organized as follows. Section II discusses the background and motivation of this work. Related work is discussed in the following Section III. In Section IV, we explain feature engineering and the resource correlation selection policy *RCorrPolicy* for choosing the ranked correlated resource metrics. Section V presents the design of *EN-Beats* model by illustrating the model architecture with weak learners and the aggregation process of constructing a strong learner. The evaluation and experimental part are discussed in Section VI, which includes the ablation experiments of *RCorrPolicy* and the performance evaluation of *EN-Beats*. Section VII concludes the chapter with the summary.

3.2 Background and Motivation

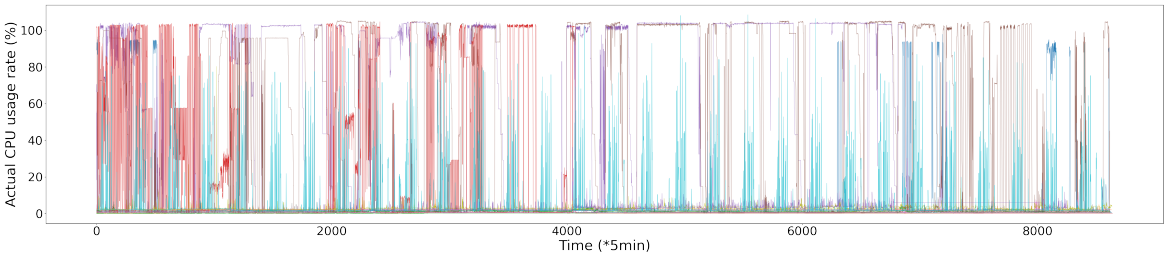


Figure 3.3: An illustration of actual CPU utilization rate from 20 randomly selected VMs across 30 days from `Bitbrains` dataset. Different color line represents different CPU utilization rate in the corresponding VM.

3.2.1 Background

Cloud computing usually refers to the delivery of different resources over the Internet, including resources like memory, CPU, bandwidth, disk, and applications/services. It has emerged as a pivotal driving force in the economy, serving as the foundational infrastructure for digital transformation. Despite the numerous advantages it offers, modern cloud computing also faces a range of challenges that hinder its overall efficiency. One of the primary obstacles lies in accurately predicting workload resource usage in cloud data centers, which is exacerbated by the escalating demands of diverse cloud applications. Consequently, workloads in cloud computing exhibit dynamic and uncertain characteristics, presenting a formidable prediction problem for resource usage. Unfortunately, existing research primarily focuses on single resource metric prediction and fails to capture the underlying correlations among different resource metrics.

3.2.2 Motivation

These challenges motivate us to propose a new method for tackling the problem of efficient resource predictions in cloud computing environments. As observed from Figure 3.3, the resource utilization characteristics within randomly selected Virtual Machines (VMs) are distributed unevenly, where different colors represent different VMs. It is

worth noting that while some VMs are highly utilized, others demonstrate a markedly low utilization rate. In addition, the sub-figures in Figure 3.1 depict that the utilization of correlated resources (e.g., CPU and memory, network incoming and outgoing traffic) are showing similar utilization patterns across time, which means that these resource metrics are correlated to some extent. The correlation matrix is shown in Figure 3.4. These figures from the real cloud production environment demonstrate the imbalance and dynamics in resource allocation and inefficiency in task scheduling.

To alleviate these issues, methods of correlation analysis among different resource metrics and efficient learning techniques should be proposed. Our main motivations can be summarized as follows:

- Most current research works fail to analyze the correlations among different resource metrics and there is no suitable correlation selection policy. Therefore, latent correlation analysis and correlation selection policy should be proposed.
- Some traditional methods like Principal Component Analysis (PCA) and Support Vector Regression (SVR) reduce the feature space largely, resulting in the loss of some valuable features, thus the system model performance is degraded.
- There are limitations to the dominant RNN-based methods (such as LSTM, Bi-LSTM, GRU) for estimating dynamic and long-term resource utilizations due to issues like gradient vanishing and exploding.

3.3 Related Work

In this section, we summarize the methods employed in previous research on predicting workload resource usage in cloud computing environments. The contributions of these studies can be primarily categorized into three types: regression-based models, learning-based models, and hybrid-based models. Regression-based approaches typically incorporate linear regression and auto-regression models. In terms of learning-based models, machine learning and deep learning-based models are widely utilized across various scenarios. Hybrid-based models generally integrate multiple models to tackle challenges in complex cloud environments.

Table 3.1: A comparison of related work.

Work	Resource Types			Resource Prediction		correlation	colocated	Cloud
	cpu	memory	network	single	multiple	analysis	workload	Environments
RPTCN[150]	✓	✗	✗	✓	✗	✓	✗	✓
esDNN[92]	✓	✓	✗	✓	✓	✗	✓	✓
[151]	✓	✗	✗	✓	✗	✗	✗	✓
[152]	✓	✗	✗	✓	✗	✗	✗	✓
ARIMA[89]	✗	✗	✓	✓	✗	✗	✗	✓
N-BEATS[1]	✗	✗	✗	✓	✗	✗	✗	✗
LSTM[49]	✗	✗	✓	✓	✗	✗	✗	✓
TCN[153]	✗	✗	✗	✓	✓	✗	✗	✓
EN-Beats (This work)	✓	✓	✓	✓	✓	✓	✓	✓

3.3.1 Regression-based Models

In recent years, many researchers have conducted extensive research on resource usage prediction from grid computing to cloud computing. The widely-used regression model for time-series forecasting is ARIMA (Autoregressive integrated moving average) model, which inspired lots of researchers to design ARIMA-based predictive models. Calheiros et al [89] introduced a regression model based on auto-ARIMA for predicting workloads by using real traces from web server requests. Other similar ARIMA-based models addressed this challenge for other scenarios [45] [46] [47]. Bi et al. [154] combined Savitzky-Golay filter and wavelet decomposition for workload prediction in the following time slot. Doulamis et al. [155] proposed a non-linear task prediction by incorporating Quality of Service oriented resource management in grid computing. These regression-based methods showed their effectiveness in terms of resource prediction. Farahnakian et al. [151] introduced a prediction approach by linear regression technique based on the history of resource usage in each host.

3.3.2 Learning-based Models

Considering the characteristics of complex workloads, an increasing number of researchers have applied numerous learning-based approaches for addressing this challenge. The learning-based models mainly include machine learning based-models and deep learning-

based models.

For machine learning-based models, Gao et al. [152] introduced a clustering-based workload prediction method by clustering all tasks into different categories and training a single prediction model for each category respectively. In [156], authors introduced Bayesian-based model for predicting short and long-term virtual resources by learning workload patterns from several data centers.

Deep learning-based models have been widely used in recent years. Qiu et al. [157] introduced the VM workload prediction model, i.e., Deep Belief Network (DBN), which consists of multiple-layered Restricted Boltzmann Machines (RBMs) and a regression layer. By integrating Extreme Gradient Boosting Tree, Eli et al. [158] introduced a system, known as *Resource Central*, to collect VM characteristics in Azure and learn VM behaviors offline and predict the over-subscription of VM resources. Boris N. Oreshkin et al [1] proposed a novel time-series forecasting model N-BEATS, which solves the problem of predicting univariate time series. LSTM/BiLSTM-based [49] [90] [159] [160], GRU-based [92] and TCN-based [150] [153] deep learning methods are also widely stuied in academia. Kumar et al [49] adopted LSTM-RNN to predict cloud workload for solving issues of power consumption and resource scaling. Bai et al [153] conducted a systematic experiment evaluation for sequence modeling and found that a simple convolutional architecture outperforms traditional recurrent networks, which inspired the author to design an efficient temporal convolutional neural network for sequence modeling.

3.3.3 Hybrid-based Models

To be adaptive to different resource prediction challenges, some researchers attempted to integrate multiple models for tackling sophisticated challenges. In [161], Janardhanan et al. proposed methods for forecasting of CPU usage of machines using LSTM and ARIMA. Ceticik et al. [162] introduced an Advanced Model for Efficient Workload Prediction in the Cloud (AME-WPC) by combing statistical and learning methods. Bankole et al. [163] developed prediction models for cloud client prediction in terms of TPC-W benchmark web application through the implementation of Linear Regression (LR),

Neural Networks (NN), and Support Vector Machine (SVM). Recently, the Quantum Neural Network-based approach for predicting cloud resource usage has also gained attention. Singh et al. [164] proposed hybrid models by implementing an Evolutionary Quantum Neural Network (EQNN) and integrating a Self Balanced Adaptive Differential Evolution (SB-ADE) model for optimizing qubit network weights, which are implemented for cloud resource prediction as well.

3.3.4 Comparisons with EN-Beats

When comparing with the proposed method *EN-Beats*, these models have limitations on correlated latent feature learning and adaptions for multiple resource patterns. Firstly, these models are mainly considering single resource metrics as inputs, thus such models are weak in taking the latent correlations among different resource metrics. Secondly, in the modern cloud, different resources show different time patterns. However, these models mentioned above are usually designed for specific workload pattern learning like periodical web requests modeling and single CPU utilization prediction, while our proposed models show good performance on learning different patterns of resource metrics, like CPU utilization rates, memory usage, and network incoming traffic.

As noted in Table 3.1, compared to related works, our method is unique because it is able to support different usage patterns from multiple types of resources and predictions along with handling correlation analysis and co-located application workloads.

3.4 Feature Engineering with RCorrPolicy

In this section, we will first describe the pre-processing of raw cluster traces through normalization. Secondly, we will compare the `Pearson` correlation and `Spearman` correlation. Finally, we will detail our proposed resource metric selection algorithm, referred to as *RCorrPolicy*.

3.4.1 Dataset Preprocessing and Feature Scaling

The original `Bitbrains` dataset consists of raw resource provision and utilization metrics from different virtual machines (VMs) in the cloud. As shown in Figure 3.3, we can observe that, in some VMs, the CPU resource utilization rate remains nearly zero for a significant of time span. Thus, firstly, we pre-processed the raw data by calculating the mean values of the same metrics at the same timestamp among all VMs. Then, we can get the mean resource usage metrics for all VMs across the observed time span. Lastly, to make the model training process more efficient, we normalized the pre-processed data via the min-max method, as indicated in Equation (3.1)

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.1}$$

where x is the original value, x_{min} is the minimal value, x_{max} is the maximum value and x_{scaled} is the scaled value. Another nonnegligible step is to scale back the predicted values. After finishing model training, if the predicted scaled values are taken as the resource utilization values, it would be wrong for calculating errors and there are biases for real value predictions, while some researchers may forget to rescale the values especially when predicting CPU or memory utilization rate, which have the same value range (from 0 to 1) as the scaled values. Thus, it is also important to re-scale the predicted scaled value with the following Equation (3.2).

$$\hat{x} = \hat{x_{scaled}} \cdot (x_{max} - x_{min}) + x_{min} \tag{3.2}$$

where $\hat{x}$ is the final predicted value, $\hat{x_{scaled}}$ is the predicted scaled value from the trained model.

3.4.2 Comparison of Pearson Correlation and Spearman Correlation

We compared the two widely-used statistical correlation methods. The main differences between these two correlation methods lie in the assumptions of the data distribution and the calculated results. `Pearson` correlation assumes that the data are uniformly distributed and the results show linear relations, while there is no requirement

for `Spearman` correlation to assume the data to be uniformly distributed and its correlation results show monotonicity between calculated data instead of linearity calculated by `Pearson`. The Equation 3.3 represents how correlations between two data vectors are calculated by `Spearman` correlation.

$$\rho = 1 - \frac{6\sum_{i\in n}(\boldsymbol{x}-\boldsymbol{y})_i^2}{n(n^2-1)} \tag{3.3}$$

where ρ represents the correlations between data vector $\boldsymbol{x}$ and vector $\boldsymbol{y}$, i is the ith observation, and n is the total number of observations.

3.4.3 Correlation selection policy for resource metrics

Considering the actual cloud computing environments, we design *RCorrPolicy* to analyze the VM metrics correlations and select the metrics for predicting either single or multiple future resource utilization. The policy procedure is described in Algorithm 3.1.

Algorithm 3.1 *RCorrPolicy* algorithm for resource-metric selection from raw resource-metric matrix

Input: Trace logs of raw resource metrics from timestamps 1 to t. The raw resource-metric matrix is $\mathbf{R}_{\text{res}} = \{\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_t\}$, where $\mathbf{R}_t = (r_1^t, r_2^t, \ldots, r_m^t)$ contains m resource metrics at time t.

Output: A selected resource-metrics list **List_Corr** that contains correlated metrics.

Choose a threshold T with $0 < T \leq 1$ as the correlation selection boundary.
For each metric i, define its time series $\mathbf{r}_i = (r_i^1, r_i^2, \ldots, r_i^t)$.
Initialise $List\_Corr_i \leftarrow \emptyset$ for all $i \in \{1, \ldots, m\}$.
for $i \leftarrow 1$ to m **do**
    for $j \leftarrow 1$ to m **do**
        $Corr_{ij} \leftarrow \text{Spearman}(r_i, r_j)$
        if $|Corr_{ij}| \geq T$ **then**
            $List\_Corr_i \leftarrow List\_Corr_i \cup \{Corr_{ij}\}$
List_Corr $\leftarrow \bigcup_{i=1}^{m} List\_Corr_i$
return List_Corr

Algorithm 3.1 demonstrates the resource metrics correlation selection process from raw data traces. Firstly, from timestamp 1 to t, we collect the raw resource metrics vector $\boldsymbol{R_{res}} = \{\vec{\boldsymbol{R}}_1, \vec{\boldsymbol{R}}_2, \vec{\boldsymbol{R}}_3, ..., \vec{\boldsymbol{R}}_t\}$ from data traces in cloud production environment. Usually,

at each timestamp, there are multiple different resource metrics, thus m resource metrics at timestamp t can be formulated as $\vec{R_t} = \{r_1^t, r_2^t, r_3^t, ..., r_m^t\}$. Secondly, based on the observations from the correlation heat map in Figure 3.4, in our scenario, we define the threshold T, which can be modified to other values under different scenarios. With the defined threshold T, we can select the correlated resource metrics for the target resource metric. Then, by adopting `Spearman` correlation analysis, we calculate the correlations between $\vec{r_i} = \{r_i^1, r_i^2, r_i^3, ..., r_i^t\}$ and $\vec{r_j} = \{r_j^1, r_j^2, r_j^3, ..., r_j^t\}$, which represents the i^{th} and j^{th} resource metric from timestamp 1 to t respectively. Finally, by following the loop process in Algorithm 3.1, the correlated metric list for i^{th} resource metric will be represented as $List\_Corr_i$ and the final all correlated lists can be represented as $\bigcup_{i=1}^{m} List\_Corr_i$.

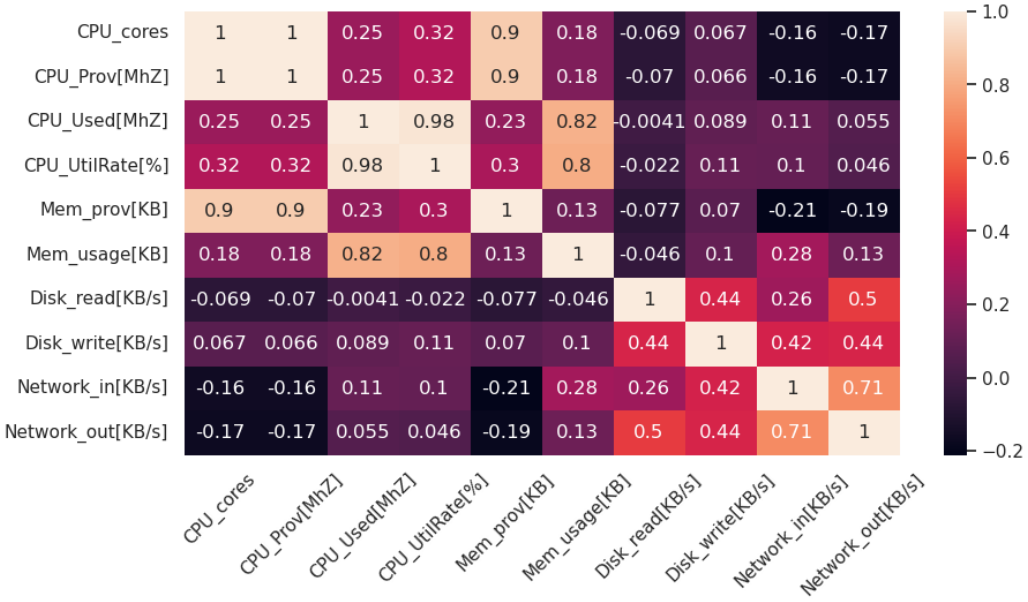


Figure 3.4: The `Spearman` correlation heatmap for different resource metrics in our experimental scenario. The metrics targeted for predictions are CPU_UtilRate(%), Mem_usage(KB), and Network_in(KB/s).

3.5 Design of EN-Beats model for resource prediction

In this section, we will introduce the proposed *EN-Beats* method. According to the observations of traces from `Bitbrains` cloud and intrinsic characteristics of neural networks,

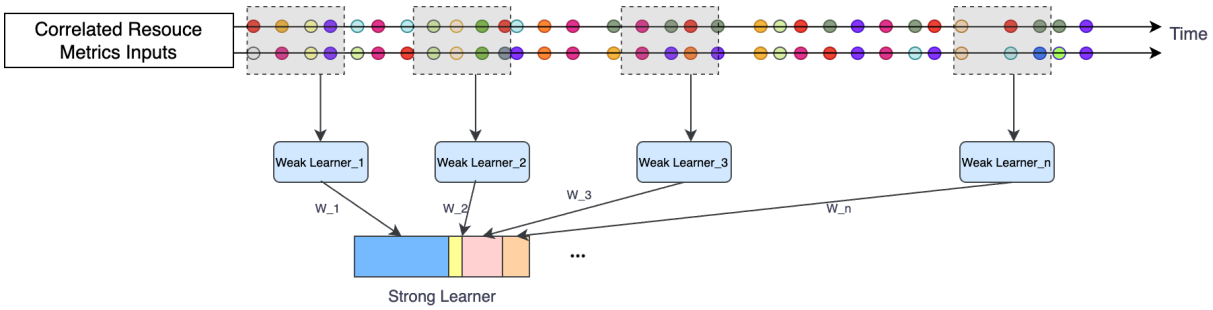


Figure 3.5: The overall design framework of *EN-Beats* with strong learner and weak learners.

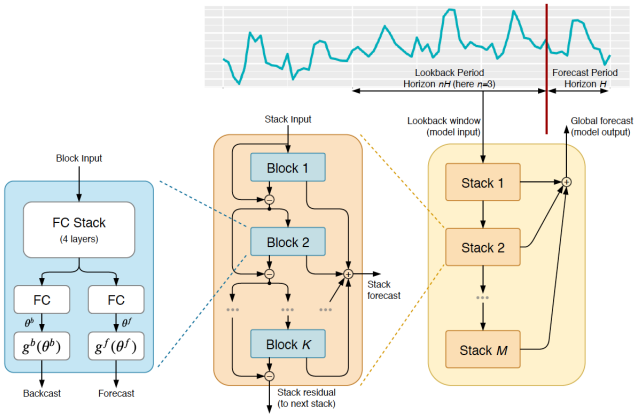
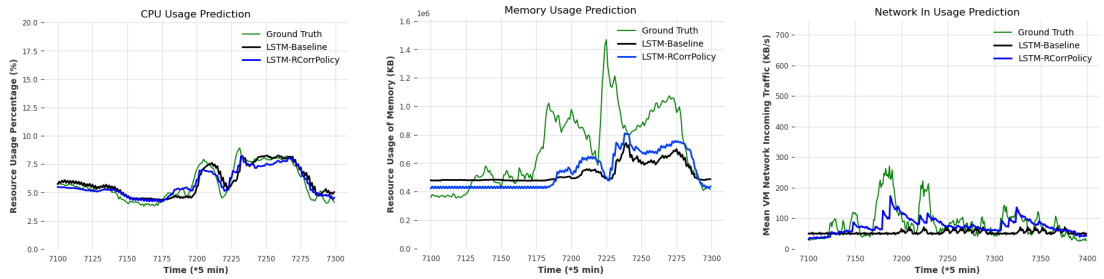


Figure 3.6: The architecture of weak learner (N-Beats [1] model) implemented in *EN-Beats* design.

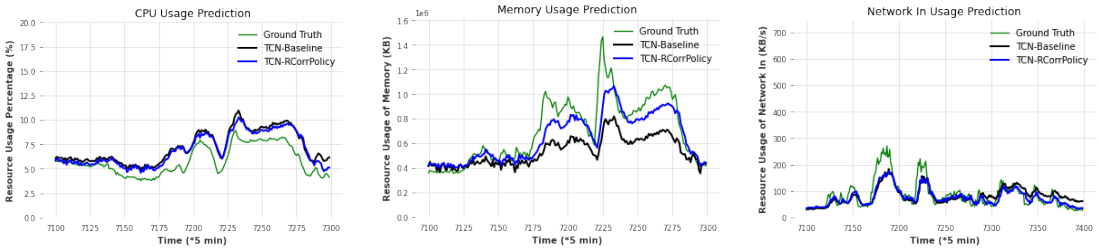


(a) LSTM CPU utilization rate prediction.

(b) LSTM Memory usage prediction.

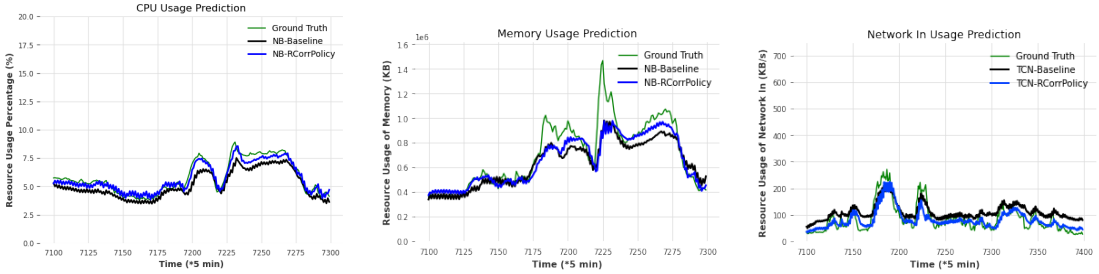
(c) LSTM Network incoming traffic prediction.

Figure 3.7: Resource usage predictions by the LSTM baseline model with *RCorrPolicy* implemented.



(a) TCN CPU utilization rate prediction. **(b)** TCN Memory usage prediction. **(c)** TCN Network incoming traffic prediction.

Figure 3.8: Resource usage predictions by the TCN baseline model with *RCorrPolicy* implemented.



(a) N-BEATS CPU utilization rate prediction. **(b)** N-BEATS Memory usage prediction. **(c)** N-BEATS Network incoming traffic prediction.

Figure 3.9: Resource usage predictions by the N-BEATS baseline model with *RCorrPolicy* implemented.

we make the following assumptions:

- Assumption1: Data between current time and previous time are correlated and the generated data in near time are the same data structure, i.e., the same dimension and data type.
- Assumption2: The data distribution shows differences as time goes by, i.e., not the same data distribution all the time.
- Assumption 3: Data are generated in a continuous way and the time consumed by model training is longer than the time for data generation, thus the model could be continually trained with new accumulated data.

3.5.1 Framework of EN-Beats

To learn the behavior of the use of resources that fluctuate continually we propose the *EN-Beats* model with a global strong learner stacked by different weak learners with different weights, as shown in Figure 3.5. For a basic weak learner, we adopted an N-Beats (Neural basis expansion analysis for interpretable time series forecasting) model is a deep learning model for time series forecasting that was introduced by Oreshkin et al. [1] in 2020. By using ensemble learning, we integrate weak learners into a strong learner, as indicated in Figure 3.5.

Weak learner

The basic building block of *EN-Beats* is the N-Beats-based weak learner (WL). Oreshkin et al. [1] proposed N-Beats model for providing interpretable time-series forecasting. It is based on a stack of fully connected layers, where each layer consists of a set of basis functions that are learned from the data. These basis functions can be thought of as a kind of dictionary of time series patterns, and the model learns how to combine them to make predictions.

To build up the weak learner, N-Beats is chosen as the basic model for the following reasons. Firstly, the N-BEATS model is highly modular and can be easily scaled to handle large and complex time series data and can also be easily adapted to different types

of time series data and forecasting tasks. Secondly, it uses a decomposition of the time series into a sequence of basis functions, which makes it highly interpretable and helps to understand the contribution of each component to the forecast and identify trends, seasonal patterns, and other factors that may affect the time series. Thirdly, it has a simple architecture that makes it fast to train compared to other deep learning models. This makes it suitable for applications where speed is important, such as real-time forecasting. Fourthly, it has been shown to achieve state-of-the-art performance on several benchmark time series datasets, outperforming other popular deep learning models like LSTMs and GRUs.

Strong learner

Inspired by ensemble learning, we build the *Strong Learner* (SL) from *WL*s with different weights. The aggregation process of *WL*s into SL can be summarized as the following process. Firstly, we partition the original dataset D into m chunks of sub-dataset with equal length, which can be represented by $D = \bigcup_{i=1}^{m} Data_i$. Secondly, we group these sub-dataset into different groups by following $Group_n = \bigcup_{i=1}^{n} Data_i$, in which $1 <= n <= m$. Thirdly, each *WL* will be trained by different groups of data. Fourthly, we train a linear regression model to aggregate the *weak learners* into the *strong learner*. When new data is coming, the newly accumulated data will be added as a new sub-dataset to update the recent data groups and train new weak learners.

3.6 Performance Evaluation

In this section, we describe the performance evaluation for *RCorrPolicy* and *EN-Beats*, including dataset introduction, baseline model explanations, experimental setup, ablation experiments for *RCorrPolicy* and evaluations for *EN-Beats* model.

Table 3.2: Statistical characteristics of the GWA-T-12 Bitbrains fastStorage dataset [3] used in our evaluations.

Properties	Mean	Min	Max	SDev
CPU requested [GHz]	8.9	2.4	86	11.1
CPU usage [GHz]	1.4	0.0	64	4.4
Memory requested [GB]	10.7	0.0	511	29.3
Memory usage [GB]	0.6	0.0	384	1.8
Disk, Read throughput [MB/s]	0.3	0.0	1,411	5.2
Disk, Write throughput [MB/s]	0.1	0.00	188	1.1
Network receive [MB/s]	0.1	0.0	859	0.7
Network transmit [MB/s]	0.1	0.0	3,193	1.5
CPU cores	3.3	1	32	4.0

3.6.1 Dataset

`Bitbrains` [3] dataset[2] was collected from the cloud of Bitbrains, which is a cloud service provider and specializes in providing financial computation services for enterprise customers, including many major banks, credit card operators, and insurers. In `Bitbrains` fastStorage dataset, it includes workload traces of 1,250 VMs that were used for co-located business software applications within Bitbrains cloud data center. These data are stored as CSV files across 30 days with `5 min` sampling interval. In Table 3.2, the collected resource metrics and the corresponding statistical characteristics are summarized. As observed in Figure 3.1 and Figure 3.3, the `Bitbrains` dataset shows the evolving resource usage trends with high dynamics and uncertainties, making the predictions of resource usages difficult.

3.6.2 Baseline Models

In the comparison experiments, we evaluated the effectiveness of baseline models with the proposed *RorrPolicy*. There are three models including RNN-based LSTM [49], TCN

[2]The open-source Bitbrains dataset can be downloaded from http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains

Table 3.3: The evaluation of **RCorrPolicy** with different models for predicting different resource metric types with evaluation metrics including MSE, NRMSE, and R^2.

COMPARISON		Evaluation Metrics					
		Without RCorrPolicy			With RCorrPolicy		
Baseline models	Resource metrics	MSE	NRMSE	R^2	MSE	NRMSE	R^2
LSTM[49]	CPU util. rate	0.44	4.8%	0.79	0.24	3.5%	0.89
	Memory usage	$83.19*10^9$	20%	-0.23	$67.69*10^9$	18.49%	-0.01
	Network_in	3669.55	8.2%	-0.38	1781.82	5.7%	0.33
TCN[153]	CPU util. rate	2.2	10%	-0.04	1.68	9.4%	0.20
	Memory usage	$55.74*10^9$	16%	0.17	$17.04*10^9$	9.2%	0.74
	Network_in	1163.26	4.6%	0.56	915.43	4.1%	0.65
N-BEATS[1]	CPU util. rate	0.68	5.9%	0.68	0.15	2.8%	0.93
	Memory usage	$17.41*10^9$	9.4%	0.74	$15.48*10^9$	8.6%	0.78
	Network_in	1548.45	5.3%	0.42	885.75	4.1%	0.67

(temporal convolutional neural network)[153] and N-BEATS model [1]. Many researchers have used RNN-based models like LSTM for cloud workload prediction [49][92], and other deep learning models like TCN [150] and N-BEATS model [1] for tackling new sophisticated scenarios.

LSTM model

Long Short-Term Memory (LSTM) is a special form of Recurrent Neural Network (RNN) that could learn long-term dependencies from historical data. The recurring module is achieved through a combination of four layers exchanging information with each other. In a typical LSTM cell, there are three gates (input gate, forgot gate, and output gate) and a timely-changing cell state. The gates and cell state empowers LSTM to selectively learn, unlearn and retain information as time goes by.

TCN model

A TCN (Temporal Convolutional Network) model consists of dilated, causal 1D convolutional layers with the same input and output lengths. This model was recently proposed by Lea et al.[153] for modeling sequential data samples with previously ig-

nored convolutional modules. While previous RNN-based models such as LSTM, bi-LSTM, and GRU have demonstrated favorable performance with certain data samples, they often encounter issues with exploding or vanishing gradients, which hampers their overall efficiency. By integrating a convolutional module rather than a recurrent one can improve the overall performance because it allows for parallel computation of outputs. TCN model and its modified version have gained popularity in the recent few years.

N-BEATS model

For univariate time series prediction, N-BEATS (Neural Basis Expansion Analysis) for interpretable Time Series is a type of recently proposed neural network with deep architectures based on backward and forward residual links, with deep stacks of fully-connected layers[1]. The N-BEATS model is capable of making more accurate predictions and interpretable results without sacrificing training time. N-BEATS has attracted increasing attention and is treated as a comparison model by state-of-the-art works [51][165]. However, the original N-BEATS model is designed for predicting univariate time series, while our scenario is multiple resource usage prediction, thus we made some changes to it to suit the multivariate inputs and outputs.

3.6.3 Evaluation Metrics

To evaluate the effectiveness of the proposed methods, we adopted three different performance metrics as follows.

MSE (Mean Square Error)

$$MSE(y_i, \hat{y}_i) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.4}$$

NRMSE (Normalized Root Mean Square Error)

$$NRMSE(y_i, \hat{y}_i) = \frac{1}{y_{max} - y_{min}}\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{3.5}$$

R^2 score (Coefficient of determination)

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y_i})^2} \tag{3.6}$$

In Equations (3.4), (3.5), and (3.6), n is the number of all true values, y_i is the value of truth observations, $\hat{y}_i$ is the prediction value, y_{max} and y_{min} are the maximum and minimum values among all y_i values respectively.

3.6.4 Experimental setup

We configured and tuned different parameters for baseline models for evaluating the proposed *RCorrPolicy* policy and *EN-Beats* model. All the algorithms are configured to provide predictions of the target metric two timestamps in advance, which is `10 min` in our scenario because `10 min` prediction in the cloud environment is enough for the system scheduler to make decisions. The threshold in Algorithm 3.1 for *RCorrPolicy* is configured as `T = 0.7`. In TCN model, some key parameters are set as `input_chunk_length = 8`, `output_chunk_length = 2`, `dropout = 0.08`, `dialation_base = 2`, `kernal_size = 5`, `learning_rate=1e-3`. In LSTM model, the key parameters are set up as `input_chunk_length=8`, `output_chunk_length=2`, `dropout=0.01`, `hidden_dim = 10`, `learning_rate = 1e-3`, `batch_size = 32`. For N-BEATS model, the key parameters are configured as `input_chunk_length = 8`, `output_chunk_length = 2`, `dropout = 0.08`, `learning_rate = 1e-3`, `num_blocks = 4`, `num_stacks = 30`, `expansion_coefficient_dim = 5`, `activation_function = ReLU`. For the EN-beats model, the weak learner's key parameters are the same as those used for N-BEATS.

3.6.5 Ablation Experiments for RCorrPolicy

RCorrPolicy Effectiveness

We analyzed the correlations among VM metrics and plotted the heatmap, presented in Figure 3.4. For each resource that we plan to predict, we used the *RCorrPolicy* to select metrics according to our needs and evaluated the effectiveness of the proposed

policy. For the comparison experiments, we selected three baseline models, including TCN[153], LSTM[49], N-BEATS[1], which are widely used for time-series prediction problems. For each baseline model, we selected three different metrics (i.e., CPU utilization rate, Memory utilization, and Network incoming traffic) in experiments and the quantified experiment results are summarized in Table 3.3.

Discussions of RCorrPolicy evaluation

In Figure 3.7, Figure 3.8, and Figure 3.9, we present the ablation experiments by implementing three models for predicting CPU utilization rate, Memory usage, and Network incoming traffic resource metrics. It can be observed that models implemented with *RCorrPolicy* outperform the corresponding models that do not include *RCorrPolicy*. From the experimental results in Figure 3.8b, the results demonstrate a notable reduction in `NRMSE` from 16% to 9.2%, representing a substantial improvement of 44.7%. In addition, from the experimental results shown in Table 3.3, it can be observed that TCN and N-BEATS models show better overall performance in predicting different resource metrics. Furthermore, when comparing TCN and N-BEATS models, according to their designed model structures [153][1], TCN demonstrates suitability for long-term time series sequences, while the N-BEATS model excels at capturing both short-term and long-term time series patterns, making it particularly valuable in dynamic and uncertain cloud environments.

3.6.6 Evaluation of EN-Beats model

We evaluated the proposed *EN-Beats* model for resource metric prediction alongside baseline models. Figure 3.10, Figure 3.11, and Figure 3.12 demonstrate the superior performance of the proposed *EN-Beats* in comparison to the baseline models.

Discussion of experimental results

To evaluate the effectiveness of the proposed *EN-Beats*, we conducted extensive experiments comparing it with existing models in terms of multiple resource predictions for

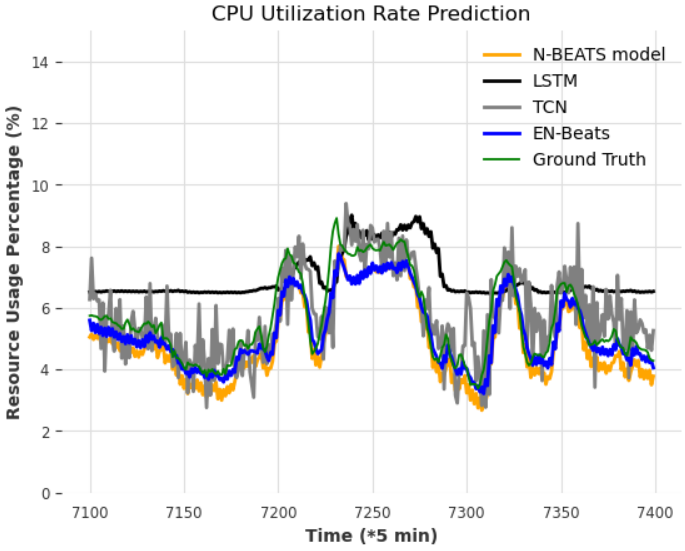


Figure 3.10: A comparison of CPU utilization rate predictions.

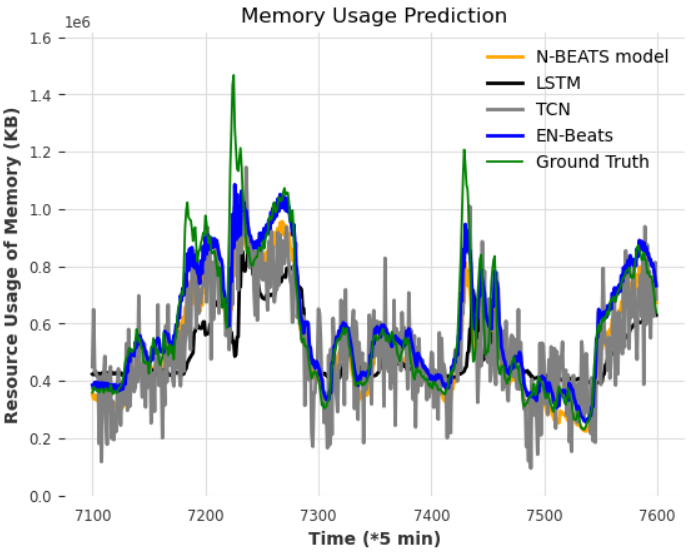


Figure 3.11: A comparison of memory usage predictions.

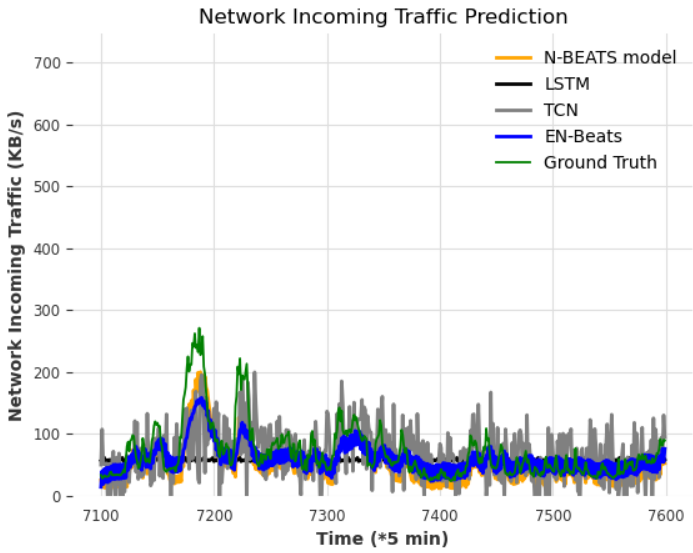


Figure 3.12: A comparison of network incoming traffic predictions

Table 3.4: The evaluation of **EN-Beats** model and baseline models for predicting CPU utilization.

CPU util. rate (%)	MSE	NRMSE	R^2
LSTM[49]	2.92	12.4%	-0.56
TCN[153]	0.80	6.51%	0.62
N-BEATS[1]	0.64	5.84%	0.66
EN-Beats(This work)	0.36	4.4%	0.83

Table 3.5: The evaluation of **EN-Beats** model and baseline models for predicting memory utilization.

Memory usage (KB)	MSE	NRMSE	R^2
LSTM[49]	$38.56*10^9$	13.97%	0.31
TCN[153]	$27.65*10^9$	11.82%	0.51
N-BEATS[1]	$13.37*10^9$	8.22%	0.76
EN-Beats(This work)	$6.96*10^9$	5.93%	0.88

Table 3.6: The evaluation of **EN-Beats** model and baseline models for predicting network incoming traffic.

Network_in traffic (KB/s)	MSE	NRMSE	R^2
LSTM[49]	21.79*10^2	6.37%	-0.08
TCN[153]	24.74*10^2	6.78%	-0.22
N-BEATS[1]	10.85*10^2	4.50%	0.46
EN-Beats(This work)	22.02*10^2	6.40%	0.54

resource usages from `Bitbrains`[3]. As depicted in Figure 3.10, Figure 3.11, and Figure 3.12, *EN-Beats* demonstrates strong performance across various resource metric predictions, exhibiting lower *MSE* (lower values indicate better performance), lower *NRMSE* (lower values indicate better performance), and higher R^2 score (higher values indicate better performance). Therefore, *EN-Beats* outperforms the existing models.

For the prediction of CPU utilization rate, Figure 3.10 illustrates that our proposed *EN-Beats* model closely follows the trend of the ground truth, outperforming other models. In terms of *NRMSE*, the LSTM model, TCN, N-BEATS, and EN-Beats achieved respective experimental results of 12%, 6.5%, 5.8%, and 4.4%. Thus, the proposed *EN-Beats* exhibits lower prediction errors for the CPU utilization rate. Additionally, our method attains the highest R^2 score, indicating superior generalization ability compared to the other methods under consideration.

In terms of resource prediction for Memory usage, Figure 3.11 and Table 3.5 demonstrate that *EN-Beats* effectively captures the changing patterns of the ground truth. Quantitatively, the NRMSE values for LSTM, TCN, N-BEATS, and EN-Beats are 13.97%, 11.82%, 8.22%, and 5.93%, respectively. These results indicate that our proposed method exhibits the lowest error rate and highest accuracy. Furthermore, our method achieves a higher R^2 score (indicating better performance) compared to other methods, further highlighting the superiority of our trained model.

For predicting `network` incoming traffic, Figure 3.12 demonstrates the close estimation achieved by our proposed *EN-Beats* model compared to the ground truth. Regarding the NRMSE values of different models, Table 3.6 presents the normalized errors for LSTM, TCN, N-BEATS, and EN-Beats as 6.37%, 6.78%, 4.50%, and 6.44%, respectively.

Although the NRMSE of our proposed model is slightly higher than that of other models, it is important to note that our model excels in another crucial evaluation metric, the R^2 score, indicating its ability to effectively fit the data.

3.7 Summary

This chapter addressed the challenge of accurately predicting multiple resource usages in cloud data centers, where dynamic and uncertain workloads make it difficult to anticipate CPU, memory, and network demands. We argued that focusing on single metrics and ignoring correlations among resources leaves substantial predictive power untapped and can lead either to overprovisioning or QoS violations. To bridge this gap, the chapter formulated multi-resource prediction as a correlation-aware time-series forecasting problem over real cloud traces; thus, we proposed an efficient metric selection policy, *RCorrPolicy*, and an ensemble learning-based model, *EN-Beats*, for multiple-resource prediction.

Overall, this chapter provides a correlation-aware metric selection policy and a multi-resource ensemble predictor that together improve the fidelity of short-horizon resource forecasts under non-stationary cloud workloads. These forecasts strengthen the thesiss telemetry foundation: they enable proactive decisions and reduce reliance on purely reactive threshold rules. The next chapter builds on this telemetry-driven perspective by shifting from host-level forecasting to *application-level dynamics*, introducing iDynamics to reconstruct call graphs, quantify traffic stress, and emulate/measure network dynamics for repeatable evaluation of microservice scheduling policies.

Chapter 4

An Emulation Framework for Evaluating Microservice Scheduling Policies

Designing and comparing microservice scheduling policies in cloudedge environments is difficult because application behaviour is shaped by multiple interacting dynamics: request mixes change call-graph structures, traffic becomes highly imbalanced across service chains, and cross-node network latency and bandwidth fluctuate over time. Traditional testbeds are costly and hard to repeat, while existing simulators rarely capture these dynamics while running real microservices. This chapter presents `iDynamics`*, a configurable emulation framework that exposes the call graph, traffic, and network conditions as controllable experimental factors on a Kubernetes-based cloudedge cluster.* `iDynamics` *comprises three modular components, including the Graph Dynamics Analyzer, the Networking Dynamics Manager, and the Scheduling Policy Extender. We use* `iDynamics` *to implement and study a call-graph-aware policy and a hybrid traffic-and-latency-aware policy using Social Network workload on a cloudedge cluster with up to 15 worker nodes, demonstrating that the framework can accurately emulate targeted network dynamics and systematically reveal how different policies impact SLA compliance under controlled yet realistic conditions.*

4.1 Introduction

Microservice architectures have become the de facto style for building cloud and edge applications. By decomposing functionality into independently deployable services, they provide elasticity and rapid evolution of individual components. However, when

This chapter is derived from:

- **Ming Chen**, Muhammed Tawfiqul Islam, Maria A. Rodriguez, and Rajkumar Buyya, "iDynamics: A Configurable Emulation Framework for Evaluating Microservice Scheduling Policies under Controllable Cloud–Edge Dynamics", *IEEE Transactions on Service Computing* [Under Review, Nov 2025].

tens or hundreds of microservices are deployed across a cloud–edge continuum, maintaining Service Level Agreements (SLAs) becomes non-trivial: user requests traverse long service chains, services are shared across applications, and traffic patterns vary significantly over time [6, 7, 20, 53, 122, 127, 166].

In such settings, scheduling policies—that is, the placement and migration of microservice *pods* onto available nodes—must react to dynamics at three layers: (i) **workload dynamics**, such as changing mixes and intensities of request types; (ii) **application-level dynamics**, such as evolving call-graph structures and imbalanced traffic between pairs of services; (iii) **infrastructure dynamics**, such as fluctuating cross-node delays and available bandwidth in the cloud–edge continuum. These intertwined effects directly impact the end-to-end latency and throughput of microservice applications deployed on Kubernetes [12] clusters.

Designing scheduling policies that cope with these dynamics already requires substantial effort; rigorously *evaluating* them is even harder. Deploying alternative schedulers in production-scale, geographically distributed clusters is costly, disruptive, and rarely repeatable. Purely simulation-based evaluations, in contrast, often abstract away essential aspects of the Kubernetes stack, the service mesh, and real microservice implementations, which limits fidelity. Therefore, there is a need for an evaluation environment that runs real services while exposing key sources of non-determinism as experimental knobs rather than as uncontrolled environmental factors.

In this work, we use the term *controllable dynamics* to denote this capability: the ability to systematically emulate and vary call-graph topologies, traffic loads along service chains, and cross-node network conditions, while monitoring the effect of these variations on SLA metrics. A framework offering such controllability would allow cloud practitioners and researchers to answer questions such as: How does a given policy behave under increasingly skewed traffic? Under which delay and bandwidth patterns does an otherwise robust scheduler violate latency SLOs? Which microservice pairs become bottlenecks as the mix of request types evolves?

A large body of work has addressed microservice management and scheduling, for example through SLO-oriented resource management [18, 19, 30, 109], trace-based call-graph analysis [29, 123, 138], network dynamics emulation [56, 83, 167], and network-

aware pod placement in Kubernetes clusters [32–35]. Complementary work has proposed microservice simulators and performance-testing tools that couple real orchestrators with synthetic workloads. However, to the best of our knowledge, existing approaches either (i) focus on building new management algorithms and evaluate them in ad hoc environments, (ii) emulate network-level behavior without understanding microservice call-graphs and SLAs, or (iii) provide simulators without a pluggable interface for alternative scheduling policies. None offers an integrated, open, Kubernetes-based framework that jointly (a) observes dynamic call-graphs and traffic, (b) emulates heterogeneous cross-node latency and bandwidth, and (c) exposes a policy-agnostic interface for implementing and comparing schedulers under controllable dynamics.

To fill this gap, we present `iDynamics`, a configurable emulation framework for evaluating microservice scheduling policies on Kubernetes-based cloud–edge clusters. Rather than proposing yet another scheduling algorithm, `iDynamics` focuses on the *evaluation problem*: it instruments the service mesh, injects realistic network dynamics, and provides an extensible policy interface so that both existing and new schedulers can be studied under repeatable conditions. The framework is implemented as a set of modular components, including the Graph Dynamics Analyzer, Networking Dynamics Manager, and Scheduling Policy Extender.

Our main contributions are as follows:

- We introduce the notion of *controllable dynamics* for microservice scheduling evaluation and implement it across the call-graph, traffic, and network layers in cloud-edge Kubernetes clusters.
- We design and implement `iDynamics`, a modular emulation framework that combines (i) service-mesh-based reconstruction and analysis of dynamic call-graphs, (ii) fine-grained cross-node delay and bandwidth emulation with distributed measurement, and (iii) a pluggable scheduling-policy interface with reusable utility functions.
- We demonstrate how `iDynamics` can be used to study scheduling behavior via two representative policies—a call-graph-aware policy and a hybrid dynamics-aware policy—using the DeathStarBench Social Network application on a scalable

cloud–edge cluster with up to fifteen worker nodes. These case studies illustrate how the framework reveals SLA violations and quantifies how different policies mitigate them under diverse dynamic scenarios.

The rest of the chapter is structured as follows. Section 4.2 reviews related work. Section 4.3 discusses background and challenges. Section 4.4 introduces the `iDynamics` framework and its main components. Sections 4.5, 4.6, and 4.7 detail the designs of the Graph Dynamics Analyzer, Networking Dynamics Manager, and Scheduling Policy Extender, respectively. Section 4.8 presents performance evaluation and case studies, and Section 6.8 concludes the chapter with summary.

4.2 Related Work

The dynamics of microservice-based systems in cluster and cloud–edge environments have been studied from multiple angles, including resource management, call-graph analysis, network emulation, and network-aware scheduling. In addition, several frameworks and benchmarks support the simulation or emulation of microservice workloads. This section reviews these research directions and contrasts them with the goals of `iDynamics`.

4.2.1 Microservice Management and Resource Scheduling

FIRM [18] leverages online telemetry and machine learning models to localize SLO violations and resource contentions, then mitigates them through fine-grained reprovisioning actions. Erms [19] formulates scalable resource management models for shared microservices with large call graphs, assigning latency objectives to individual services. GrandSLAm [109] predicts completion times for stages of microservice execution and dynamically batches and reorders requests based on these predictions. Sage [30] uses probabilistic models to diagnose cascading QoS violations in interactive microservices at scale. TraDE [67] further introduces a network- and traffic-aware adaptive scheduling framework that builds a traffic-stress graph from bidirectional microservice traffic,

injects controllable cross-node delays, and redeploys microservice instances to maintain QoS targets under changing workloads and network conditions.

These systems demonstrate sophisticated management logic but treat their execution environment largely as given: they do not expose a generic evaluation framework that allows different scheduling policies to be plugged in and compared under configurable call-graph, traffic, and networking dynamics. In contrast, `iDynamics` does not propose yet another resource manager; instead, it provides the infrastructure required to test such managers under realistic and repeatable cloud–edge dynamics.

4.2.2 Call-Graph Dynamics Analysis

Several works analyze microservice dependencies and their impact on performance. Tian et al. [138] synthesize task-dependency graphs for data-parallel jobs from large-scale cluster traces. Luo et al. [123] characterize microservice call graphs from Alibaba traces, categorizing microservice dependencies into three distinct types. Parslo [29] decomposes end-to-end SLO budgets into node-specific latency targets using gradient descent. Sage [30] also models dependencies when diagnosing QoS violations.

These approaches primarily use call-graph information as input to diagnosis or optimization algorithms. `iDynamics`, instead, treats dynamic call-graphs as part of the *experimental context*. Its Graph Dynamics Analyzer reconstructs call-graphs online from service-mesh telemetry and quantifies bi-directional traffic for each upstream–downstream pair, so that policies and applications can be evaluated under changing call-graph structures and traffic imbalances.

4.2.3 Network Dynamics Emulation

Network dynamics strongly influence microservice performance in cloud–edge systems. THUNDERSTORM [83] evaluates distributed systems under dynamic topologies using micro- and macro-benchmarks. Kollaps [56] provides a decentralized topology emulator, while SplayNet [167] builds on Splay [168] to emulate arbitrary overlay topologies. These tools are effective at reproducing network-level behavior but are largely agnostic to microservice call-graphs, SLAs, and scheduling policies.

At the level of individual nodes, several works inject delays or shape bandwidth using Linux traffic control primitives [14, 15, 18, 125, 126]. Most, however, either employ static or uniform delay matrices or apply configurations at the granularity of a whole node, which makes it difficult to study heterogeneous pair-wise conditions in a cloud–edge cluster. `iDynamics` builds on these ideas by combining classful queueing disciplines and destination-specific filters into an Emulator that can inject distinct delay and bandwidth profiles for each node pair, while preserving default channels for unrelated traffic. A distributed Measurer then validates and tracks these injected dynamics at runtime, enabling closed-loop experiments on scheduling policies.

4.2.4 Network-Aware Microservice Scheduling

Network-aware schedulers exploit infrastructure-level information when placing microservices. NetMARKS [32] uses service-mesh metrics from Istio [42] to place Kubernetes pods for 5G edge applications. Marchese et al. extend the default Kubernetes scheduler with network-aware placement policies [34, 35]. OptTraffic [33] optimizes cross-machine traffic for multi-replica microservices by migrating containers with strong dependencies.

These works confirm the value of incorporating network metrics into scheduling decisions but typically introduce a specific policy and evaluate it against the default Kubernetes scheduler, often in static or lightly controlled environments. In contrast, `iDynamics` is designed to host any such policy and compare them across configurable combinations of call graph, traffic, and network dynamics.

4.2.5 Microservice Simulators and Evaluation Tools

Beyond schedulers and network emulators, several tools support simulation or emulation of microservice systems. Benchmark suites such as DeathStarBench [2] provide realistic microservice applications and workloads. μBench [16] benchmarks microservice applications with dummy services on Kubernetes-based platforms. A Kubernetes-centric simulator [55] has been proposed that couples real container orchestration with simulated workloads to enable “Kubernetes-in-the-loop” performance evaluation. Sur-

veys of Kubernetes scheduling [73] systematically categorize existing policies and open challenges. More recently, CloudNativeSim [54] builds on CloudSim [135] to provide a discrete-event simulator for cloud-native, microservice-based applications, modeling service graphs, pods, and virtual machines and exposing policy interfaces for service management.

These works provide essential building blocks and taxonomies but do not offer a unified, open framework that simultaneously (i) observes dynamic microservice call-graphs, (ii) emulates heterogeneous cross-node network conditions, and (iii) exposes a pluggable interface for arbitrary scheduling policies while running real microservice containers on a Kubernetes-based cloud–edge cluster. `iDynamics` is designed to fill this specific gap.

4.3 Background and Challenges

4.3.1 Background

Microservice architecture decomposes an application into small, independently deployable services that communicate over the network [2, 123]. In cloud–edge computing, these services can be deployed in central cloud data centers, on edge nodes near end users and devices, or across both. In practice, an application may be partitioned so that some microservices run in the cloud (for global data, analytics, or coordination) while latency-sensitive components execute at the edge, communicating via Remote Procedure Calls (RPCs) or REST APIs.

This hybrid deployment improves responsiveness by reducing the distance that latency-critical data must travel and offloading work from the cloud to local nodes. At the same time, it increases sensitivity to network conditions and placement decisions: each network hop between microservices incurs latency and consumes bandwidth, and shared microservices often become hotspots as multiple application workflows converge on them.

In the remainder of this chapter, we use the term *upstream microservice* (UM) to denote the caller in a dependency pair and *downstream microservice* (DM) to denote the callee.

Scheduling decisions determine where microservice pods are placed on cluster nodes and under what conditions pods are migrated between nodes.

4.3.2 Challenges

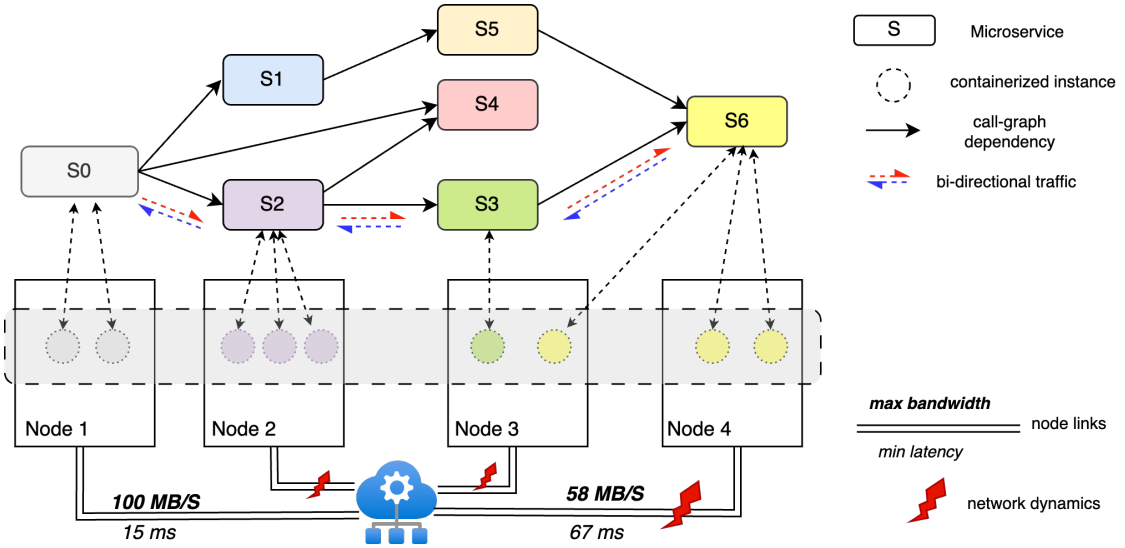


Figure 4.1: An envisioned workflow illustrating containerized microservice execution and communication under networking dynamics in a cloud–edge continuum.

Figure 4.1 illustrates an envisioned workflow of microservices executing across multiple nodes under dynamic network conditions in a cloud–edge continuum. Building effective scheduling policies and rigorously evaluating them requires addressing at least three major challenges.

Dynamic Networking Conditions

In cloud–edge scenarios, microservice applications often experience dynamic, heterogeneous networking conditions. Cross-node delays and available bandwidth can vary due to contention, background traffic, and routing changes. Microservices heavily depend on RPCs or REST APIs, making them highly sensitive to these variations. As a result, fluctuating cross-node latencies and bandwidth constraints significantly affect end-to-end response times and throughput. Evaluating how scheduling policies behave

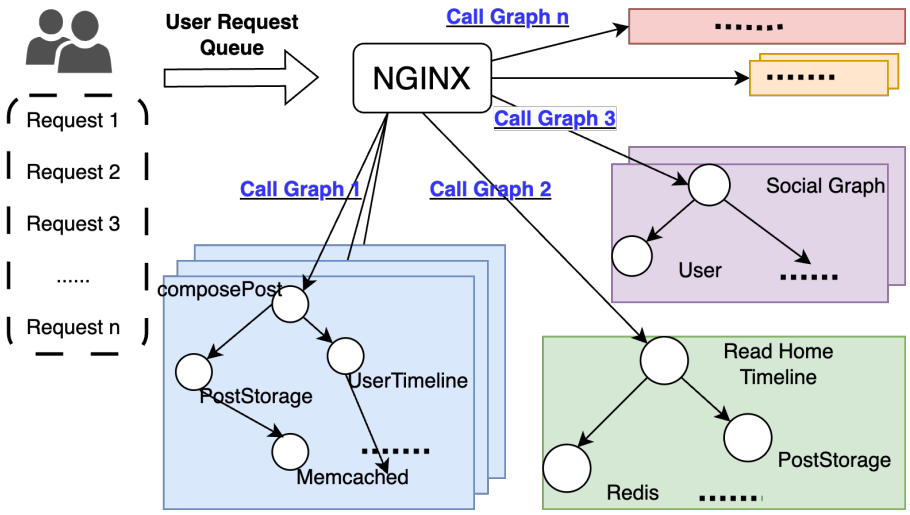


Figure 4.2: Different request types triggering distinct call-graph structures (application source [2]).

under such conditions requires a framework that can emulate and measure realistic, time-varying network profiles, rather than relying solely on the low-latency conditions of a laboratory cluster.

Variability in Call-Graph Structures

Microservice requests traverse complex call graphs, which vary depending on the types of incoming requests. As shown in Figure 4.2, widely used benchmark applications such as Social Network [2] expose different request types (e.g., compose-post, read-home-timeline, read-user-timeline), each triggering a distinct call-graph structure. The relative proportions of these request types change over time, leading to evolving mixtures of call graphs and shifting hotspots. Scheduling policies must therefore handle both structural variability (which services are involved) and workload variability (how frequently each path is exercised). Accurately capturing these variations is essential when evaluating policies targeting end-to-end SLAs.

Imbalanced Traffic Distribution

Even within a fixed call-graph structure, traffic loads across UM–DM pairs can be highly imbalanced. Some service pairs carry orders of magnitude more requests or larger payloads than others, and this imbalance becomes more pronounced under high load. Figure 6.2 shows an example from the Social Network application where a small subset of UM–DM pairs accounts for the majority of traffic at 5k queries per second (QPS). The pair `user-timeline-service` → `user-timeline-mongoDB`, for instance, experiences much higher traffic than other edges in the same call-graph.

Such imbalances can cause localized bottlenecks that dominate end-to-end latency, especially when the corresponding microservice pods are placed on nodes connected by high-latency or low-bandwidth links. Evaluating scheduling policies in this context requires a framework that (i) observes bi-directional traffic between UM–DM pairs, (ii) identifies imbalanced edges, and (iii) can manipulate network conditions to stress specific parts of the call-graph.

These challenges motivate the need for an evaluation framework that can jointly observe and control workload, application, and infrastructure dynamics when studying microservice scheduling policies.

4.4 iDynamics Framework

Motivated by the challenges discussed above, we propose `iDynamics`, a unified emulation framework for implementing and evaluating microservice scheduling policies in cloud–edge Kubernetes clusters. Specifically, `iDynamics` is designed to:

- run real containerized microservices and workloads;
- expose call-graph, traffic, and network conditions as controllable experimental factors; and
- provide a pluggable interface where different scheduling policies can be implemented, executed, and compared.

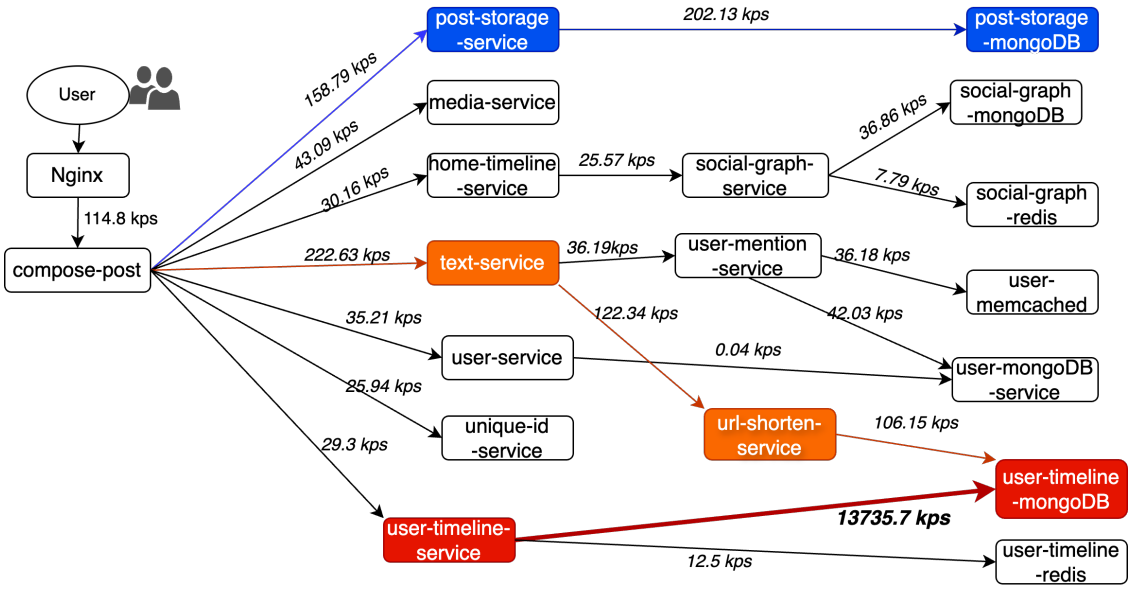


Figure 4.3: Imbalanced traffic loads among UM–DM pairs under high workload (5k Queries Per Second) scenarios.

Figure 4.4 presents the architecture of `iDynamics` and the interactions among its components during evaluation and scheduling. The framework supports managing diverse dynamics and implementing scheduling policies to mitigate SLA violations across the cloud–edge continuum. It currently targets Kubernetes and Istio, but the design principles generalize to other orchestrators and service meshes.

The key components of `iDynamics` are:

- **Graph Dynamics Analyzer**: This component consists of a UM–DM Traffic Profiler and a Call-Graph Builder. It reconstructs the call-graph topology triggered by different request types and quantifies bi-directional traffic between microservices (and their replicas) under varying workloads (Section 4.5).

- **Networking Dynamics Manager**: This component includes an Emulator and a distributed Measurer. The Emulator generates configurable cross-node latency and bandwidth patterns using Linux traffic control, while the Measurer collects actual communication metrics between nodes via lightweight daemon agents deployed across the cloud–edge cluster (Section 4.6).

- **Scheduling Policy Extender**: This component provides a Policy Customization

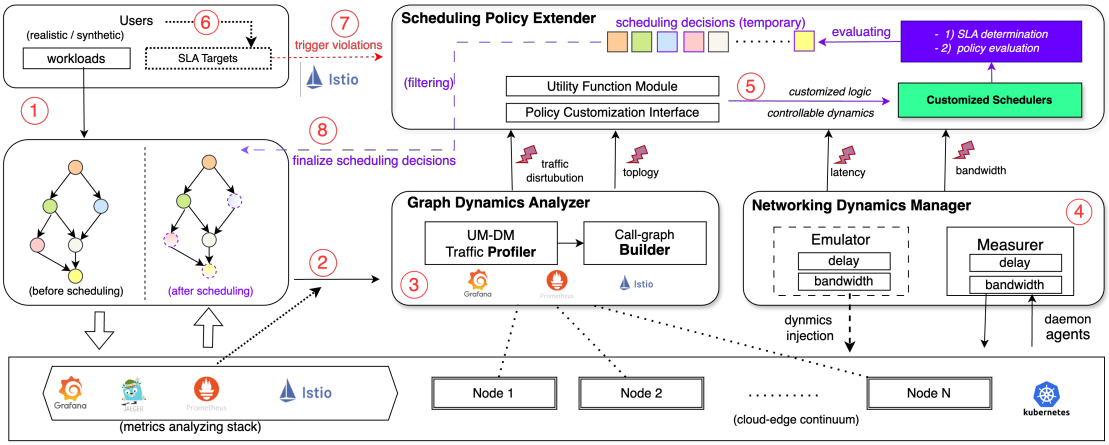


Figure 4.4: `iDynamics` framework architecture and working procedure.

Interface and a Utility Function Module. It enables rapid development and evaluation of scheduling strategies by exposing a policy-agnostic abstraction of nodes, pods, and metrics, and by providing helper functions to access cluster and network state (Section 4.7).

As depicted in Figure 4.4, the working procedure of `iDynamics` consists of multiple stages:

① Users submit realistic or synthetic workloads (with different request types and queries per second) to the deployed microservice applications.

②-③ Performance metrics are collected using a monitoring stack (e.g., Istio [42], Jaeger[**?**], Prometheus [38]). The Graph Dynamics Analyzer then builds the triggered call-graph and analyzes traffic between microservices, forwarding these insights to the Scheduling Policy Extender.

④ In parallel, the Networking Dynamics Manager measures real-time cross-node conditions and can optionally inject dynamic delay and bandwidth patterns into the cloud–edge environment.

⑤ Using the collected dynamics (call-graphs, traffic distributions, and node-level latency and bandwidth), users implement and evaluate customized scheduling policies via the policy interface and utility module. In this context, a *scheduling decision*

is a mapping of one or more pods to target nodes, possibly involving migrations to new nodes.

⑥ Users specify SLA targets (e.g., tail latency thresholds) that determine when a policy should be triggered.

⑦ SLA violations for policy evaluation are induced either by tightening SLA criteria or by injecting more intense dynamics (e.g., increased delay or workload), causing performance degradation. Candidate scheduling decisions produced by the policy are stored in a decision queue and can be filtered based on compliance requirements or constraints on which services are allowed to migrate.

⑧ Finally, the filtered scheduling decisions are executed by reconfiguring pod placements in the Kubernetes cluster, adjusting the deployment according to current conditions.

With `iDynamics`, scheduling policies can thus be rapidly implemented, tested, and refined under realistic yet controllable conditions. For example, traffic- and latency-aware scheduling strategies can be developed using dynamic metrics from the Graph Dynamics Analyzer, emulated network dynamics from the Networking Dynamics Manager, and the extensible interfaces provided by the Scheduling Policy Extender.

4.5 Graph Dynamics Analyzer

Considering the characteristics of microservice call-graphs described in Section 4.3, the *Graph Dynamics Analyzer* is primarily designed to analyze (i) topology dynamics (variations in triggered call-graphs) and (ii) traffic dynamics (imbalanced traffic distributions) of microservice applications deployed in the cloud–edge continuum. To effectively capture these dynamics, we leverage a service mesh to implement two core modules: the *UM–DM Traffic Profiler* and the *Call-graph Builder*, as illustrated in Figure 4.4.

4.5.1 Service Mesh

A service mesh facilitates real-time analysis and bidirectional traffic monitoring between multiple upstream microservices (UM) and downstream microservices (DM), including their replicas. A key requirement for the *Graph Dynamics Analyzer* in `iDynamics` is to efficiently capture and analyze bidirectional traffic between dependent microservices under realistic workloads.

When there is only a single instance of each microservice, metrics such as per-connection bytes and packets can be directly obtained from the Linux filesystem. However, modern microservice applications typically scale individual services horizontally. Multiple UM and DM replicas introduce complex many-to-many communication patterns, making naive per-instance traffic analysis expensive and fragile. To address this, we adopt a service-mesh-based approach that offloads traffic collection and aggregation to sidecar proxies.

Istio Service Mesh Implementation

To obtain fine-grained metrics of bidirectional traffic between UM and DM microservices and their corresponding replicas, we implement the *UM–DM Traffic Profiler* using the Istio service mesh[1] [42]. The service mesh is deployed as an additional infrastructure layer that transparently intercepts all service-to-service traffic via sidecar proxies.

In a Kubernetes cluster with Istio enabled, each microservice pod contains the application container and an Envoy sidecar container. As depicted in Figure 5.5a, the orange rectangles represent the application containers, while the green rectangles denote the Envoy sidecars, which route traffic among microservices and expose detailed telemetry. The green lines indicate the traffic flows and dependency relationships among services. In the control plane, Istio components (e.g., Pilot, Mixer, Citadel) manage configuration, telemetry, and security policies.

[1]Istio is an open-source service mesh that uses injected sidecar containers to provide secure communication, traffic management, and load balancing.

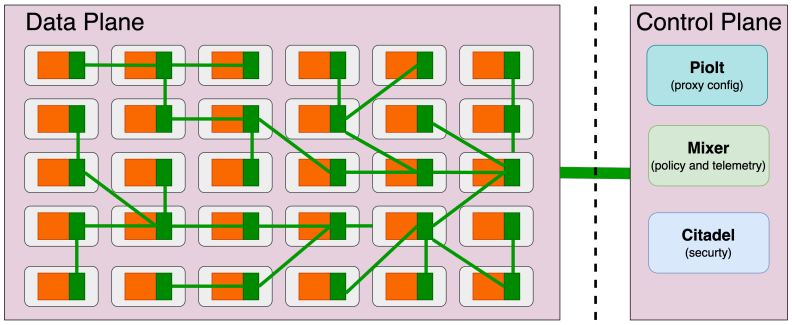


Figure 4.5: Overview of a service mesh structure consisting of data plane and control plane.

Overhead Analysis

Introducing a service mesh inevitably adds some per-request overhead due to the extra sidecar proxy layer. However, this overhead is negligible and acceptable compared to the benefits of fine-grained traffic visibility. According to the performance analysis of Istio v1.21.2 with telemetry v2 [140], each request that traverses both client-side and server-side Envoy proxies experiences an additional latency of approximately 0.18 ms at the 90th percentile and 0.25 ms at the 99th percentile compared to a baseline without proxies, under a 1 kB payload and 1000 requests per second. These measurements were performed on the CNCF Community Infrastructure Lab [169]. Thus, the service-mesh layer introduces negligible, acceptable latency overhead while enabling accurate, scalable traffic telemetry.

4.5.2 Call-graph Builder

The primary function of the *Graph Dynamics Analyzer* is to construct real-time call-graphs that capture both dependencies and traffic flows between deployed microservice instances. Different request types and varying QPS (Queries Per Second) yield distinct call graphs and traffic patterns, creating diverse scenarios for evaluating scheduling policies with specific SLA targets.

We introduce the following terminology.

Stress Element: A Stress Element (SE) models the interaction between two dependent microservices: an upstream microservice (UM) and a downstream microservice

Algorithm 4.1 Build call-graph with dependency topology and associated traffic stress

Input: Application namespace ns, time interval Δt

Output: Directed call-graph G with weighted traffic edges

Initialize G as an empty directed graph
$MS \leftarrow$ GETRUNNINGMS(ns) ▷ all microservices in ns
for each src in MS **do**
    for each dst in MS **do**
        if $src \neq dst$ **then**
            $traffic \leftarrow$ STRESS($src, dst, \Delta t$)
            if $traffic > 0$ **then**
                G.`add_edge`$\big((src, dst), traffic\big)$
return G

(DM). For a given time interval, a Stress Element quantifies the *traffic stress* between a UM–DM pair as the average of the bidirectional traffic (sent and received) between them. We treat both directions as equally important because they jointly influence end-to-end performance and SLA compliance.

Formally, the stress of a Stress Element is defined as:

$$\text{stress}_{\sigma}^{\mu}(\mu^{UM}, \sigma^{DM}, \Delta t) = \frac{\text{BiDirectionalTraffic}(\mu^{UM}, \sigma^{DM})}{2\Delta t}, \tag{4.1}$$

where μ and σ denote the upstream and downstream microservices, respectively, and Δt is the measurement interval. The function BiDirectionalTraffic(μ^{UM}, σ^{DM}) returns the sum of bytes sent from μ to σ and from σ to μ during Δt. In `iDynamics`, we compute this quantity from Prometheus metrics such as istio_tcp_sent_bytes_total and istio_tcp_-received_bytes_total.

A Stress Element is therefore represented as:

$$SE(\mu^{UM}, \sigma^{DM}, \text{stress}_{\sigma}^{\mu}).$$

Given a set of deployed microservices in an application namespace, the *Call-graph Builder* periodically constructs a directed, weighted graph where vertices correspond to microservices and edges correspond to non-zero traffic stress between them. The weight

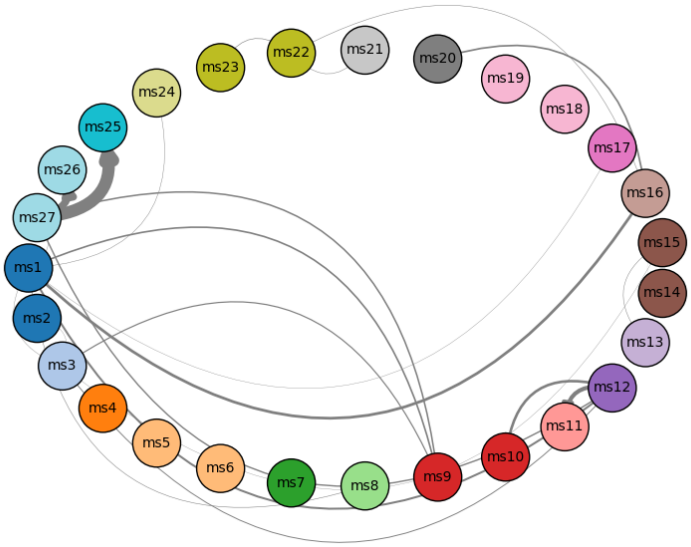


Figure 4.6: Based on Algorithm 4.1, *Graph Dynamics Analyzer* builds a real-time traffic graph of an application with 27 microservices, the call graph shows dependencies and traffic flows.

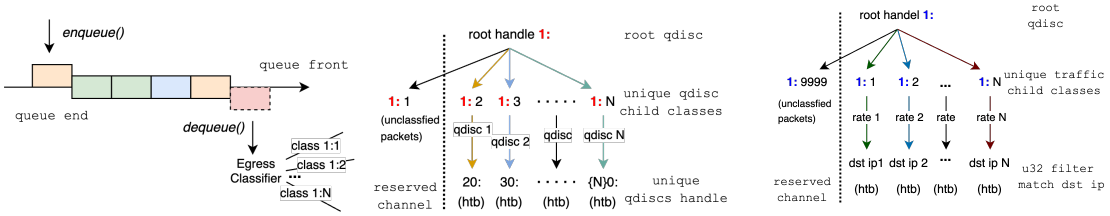
on an edge encodes the magnitude of traffic stress and is used by scheduling policies to explore communication hotspots and bottlenecks.

Algorithm 4.1 shows how the call-graph is constructed within each monitoring interval Δt. For each monitoring interval, the resulting call-graph captures both (i) application-level topology dynamics (which services call which) and (ii) traffic-level dynamics (how much they communicate). Figure 4.6 shows a representative call-graph for a microservice application with 27 services (Social Network [2]), illustrating how `iDynamics` encodes dependencies and traffic flows.

4.6 Networking Dynamics Manager

The *Networking Dynamics Manager* provides two core capabilities in `iDynamics`: (i) emulation of controllable cross-node networking conditions and (ii) accurate measurement of those conditions in the cloud–edge continuum. It consists of two main components: the **Emulator** and the **Measurer**. Together, they enable systematic evaluation of scheduling policies under diverse delay and bandwidth configurations without requiring intru-

sive changes to production environments.



(a) Packet egress with classful disciplines.

(b) Classful `qdisc` for delay emulation.

(c) Classful `qdisc` for bandwidth shaping.

Figure 4.7: The proposed method of destination-oriented networking emulation for (a) Packets are enqueued and dequeued according to classful disciplines. (b) Destination-specific delay emulation. (c) Destination-specific bandwidth shaping.

4.6.1 Linux Traffic Control Primitives

Our emulation logic uses the Linux traffic control (TC) subsystem. At the kernel level, we rely on queueing disciplines, classes, and packet classifiers; at the user level, we configure these primitives via the `tc` utility.

qdisc (queueing discipline): Defines how packets are enqueued and dequeued on a network interface. Classless qdiscs (e.g., `pfifo_fast`) have no internal classes and are easy to configure, whereas classful qdiscs (e.g., `htb`) support multiple traffic classes arranged hierarchically and are suitable for fine-grained shaping.

class: A logical sub-queue within a classful qdisc. Each class can have its own rate and ceiling parameters and can host a child qdisc. In our Emulator, we create one `htb` class per destination node so that delay and bandwidth configurations can be applied independently for each destination, while leaving a default class for non-experimental traffic.

netem qdisc: A classless queueing discipline used to emulate network impairments such as delay, jitter, loss, and reordering. We attach `netem` qdiscs as children of the per-destination `htb` classes to inject customized cross-node delays.

u32 filter: A general-purpose packet classifier that matches fields in the packet header (e.g., source/destination IP address, ports, protocol) by applying mask-and-shift operations on 32-bit words. We use it to direct different flows into different classes within the

classful `htb` qdisc based on their destination IP addresses.

`tc` utility (traffic control): A user-space utility from the `iproute2` suite that configures the kernel traffic control subsystem. It installs, updates, and removes qdiscs, traffic classes, and filters on network interfaces, enabling emulation of bandwidth caps, delay, jitter, or loss.

In `iDynamics`, we combine a classful `htb` root qdisc, per-destination classes with optional `netem` children, and `u32` filters to (i) inject configurable cross-node delays and (ii) shape available bandwidth per destination, while keeping non-experimental traffic (e.g., traffic to the control plane or the public Internet) unaffected.

Algorithm 4.2 Generate Cross-Node Delay Matrix

Input: Number of nodes N, Base latency bl, Maximum additional latency mal
Output: Cross-node delay matrix $delay\_matrix$ of size $N \times N$
Initialize $delay\_matrix \leftarrow [0]_{N \times N}$ ▷ all set to 0
for each node i in $[0, N-1]$ **do**
  for each node j in $[0, N-1]$ **do**
    if $i == j$ **then**
      $delay\_matrix[i][j] \leftarrow 0$ ▷ avoid self-delay
    else
      `/*additional latency*/`
      $al \leftarrow \text{RANDUNIFORM}(0, mal)$
      `/*distance factor*/`
      $df \leftarrow \frac{|i-j|}{N}$
      $emu\_latency \leftarrow bl + (al \times df)$
      `/*congestion factor*/`
      $cf \leftarrow \text{RANDUNIFORM}(0.5, 1.5)$
      `/* convert to integer */`
      $delay\_matrix[i][j] \leftarrow \lfloor emu\_latency \times cf \rfloor$
return $delay\_matrix$ ▷ dynamic cloud–edge delays

4.6.2 Emulator: Destination-oriented Design

Emulation of Customized Delays

Emulating diverse crossnode delays among cluster nodes is crucial for evaluating the robustness of various scheduling policies in the cloudedge continuum. However, imple-

menting customized communication delays from a single source node to multiple destination nodes poses significant challenges. To the best of our knowledge, as discussed in related work [14, 15, 18, 125, 126], none of the existing studies has proposed an efficient method (i.e., one that is both simple and rapid) for injecting tailored communication delays in a controllable manner. In practical cloudedge continuum environments, these approaches generally exhibit two main limitations: (1) the use of uniform communication delays from a single source node to all destination nodes prevents differentiation between node pairs, as delays from the source to all other nodes are identical; and (2) other networking services that do not involve the correlated nodes suffer degradation because the injected delays affect all outgoing traffic.

To address these limitations, we propose a destination-oriented emulation method that classifies packets using filters to distinguish egress packets and directs them based on their IP destinations (see Figure 4.7b). Additionally, we reserve an extra channel for default packet transmission without injected delays, ensuring that the performance of other services remains unaffected, for example, the packets from/to the control-plane node and outside internet services like Google. We implemented this scheme using the traffic control primitives of qdiscs (queuing disciplines) and the htb (Hierarchical Token Bucket) in Linux.

Furthermore, we implemented the destination-oriented method in Figure 5.7 via an algorithm for emulating customized cross-node delays, as presented in Algorithm 4.2. To emulate cross-node delays more realistically, we incorporate several factors: the base latency *bl* (representing the ideal minimal latency); the maximum additional latency *mal*; the distance factor *df* (accounting for latency due to physical separation); and the congestion factor *cf* (which simulates network uncertainties induced by traffic congestion). Here, $\textsc{RandUni}(0, mal)$ denotes the additional delay generated according to a random uniform distribution between 0 and *mal*. Thus, the emulated communication delay from node i to node j in a cluster of N cloud–edge nodes is given by the following equation:

$$\text{delay}_i^j = \left\lfloor \left(bl + \textsc{RandUni}(0, mal) \times \frac{|i-j|}{N} \right) \times cf \right\rfloor \tag{4.2}$$

Emulation of Customized Bandwidths

Shaping customized available bandwidths between different cloud–edge node pairs is also crucial for creating dynamic networking conditions, thereby enhancing the generalizability of evaluated scheduling policies. Prior researchsuch as the work presented in [56] and [83]has demonstrated the feasibility of emulating dynamic network conditions, including bandwidth variations, using decentralized emulation techniques. However, these approaches typically use a global or node-level configuration that does not support fine-grained per-destination control.

In contrast, our proposed approach leverages traffic control primitives, including unique traffic classes and the u32 filter, to match different IP destinations. As illustrated in Figure 4.7c, this approach enables the emulation of distinct bandwidth settings for different destination nodes, even when originating from the same source cluster node. This fine-grained control is essential for accurately emulating heterogeneous network conditions in cloud–edge environments.

4.6.3 Measurer: Distributed Agent Design

In practical cloud–edge computing environments, fluctuating network conditions significantly impact microservice performance and SLA compliance, including varying delays and bandwidths between nodes. High latency between cluster nodes negatively affects microservice communication, while dynamically changing bandwidth can lead to congestion, increased packet loss, and potential SLA violations. To accurately capture these dynamic cross-node conditions, we introduce the **Measurer**, a distributed agent-based measurement module within the *Networking Dynamics Manager* component of `iDynamics` (Figure 4.4).

Design Overview: The **Measurer** uses a unified approach to capture cross-node delays and bandwidths via distributed measurement agents. Although delay and bandwidth measurements follow a similar structure, we illustrate the design using delay measurement for clarity. The module consists of a centralized information processing unit and distributed lightweight agents. The centralized processing unit maintains minimal connections with agents and efficiently aggregates the collected data.

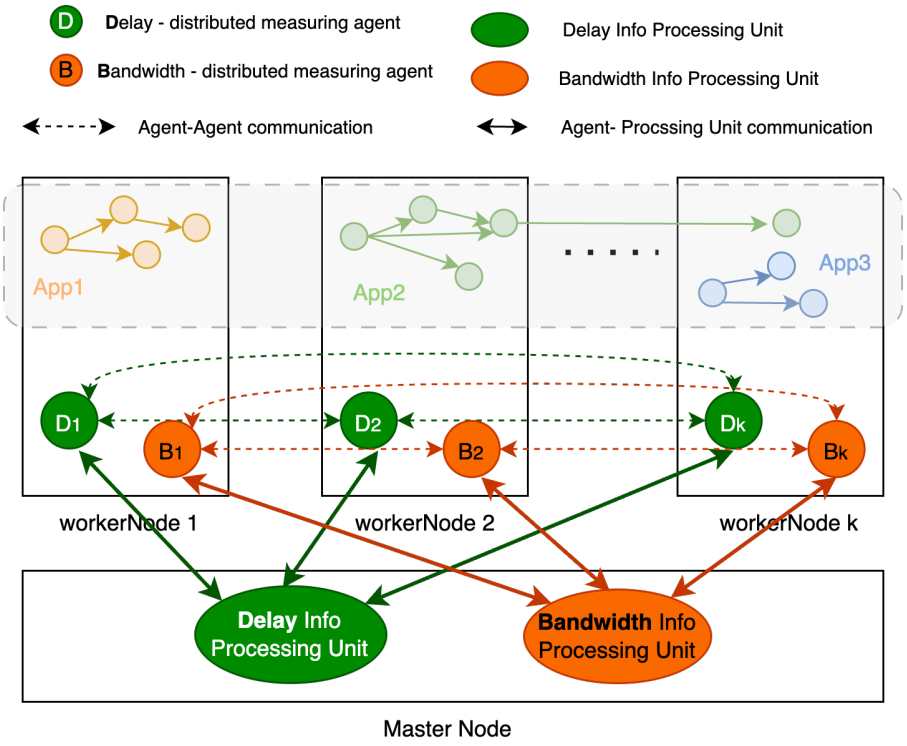


Figure 4.8: Design of *Measurer* for delay and bandwidth measurement through distributed agents across cloud–edge nodes.

The distributed agents run as lightweight containers deployed across cluster nodes, dedicated to measuring and reporting network metrics. We implement an automatic scaling mechanism for these agents to ensure robustness and adaptability during cluster scaling. When new nodes join the cluster, corresponding agents are automatically instantiated, ensuring consistent and accurate measurements. Conversely, when nodes are removed, their associated agents are gracefully terminated.

Implementation Details: The centralized information processing unit is implemented as a plugin running on the control-plane node, collecting measured metrics from distributed agents via standard TCP communications. The distributed agents are deployed as Kubernetes DaemonSet pods, ensuring each node automatically hosts a dedicated measurement pod. These agents continuously measure and report cross-node delays and bandwidths, dynamically adjusting their presence in response to cluster scaling events, as depicted in Figure 4.8.

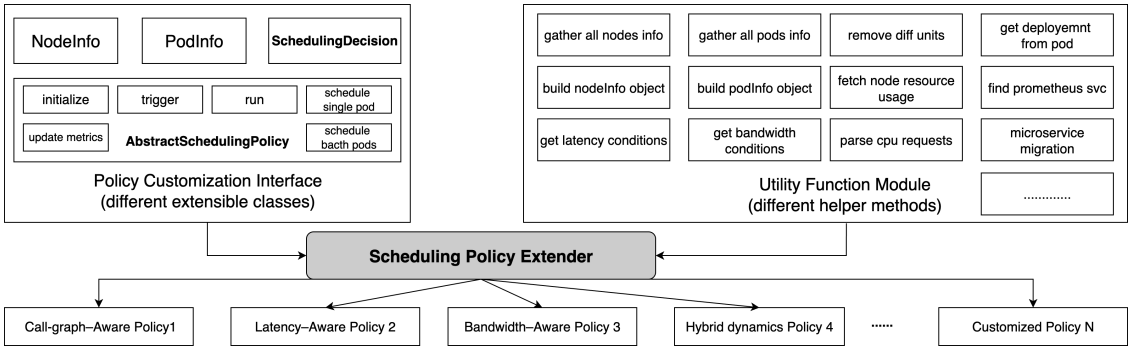


Figure 4.9: Overview of the **Scheduling Policy Extender**, highlighting extensible classes, utility functions, and example policies.

Overhead Analysis: Regarding the overhead of measurement in a real cloud–edge cluster, we optimized the measurement using parallel processing. The measurement tasks for delays and bandwidths are designed to run concurrently, and each measurement result is aggregated into a result dictionary. Additionally, the distributed delay-measuring agents are lightweight (about 0.2 MiB) and stable (running for over 11 months without failure). Each node runs a delay-measuring agent built from the curlimages/curl image, consuming about 0.2 MiB of memory per node. Similarly, each node runs a bandwidth-measuring agent built from the networkstatic/iperf3 image, consuming around 0.84 MiB of memory per node. As the measurement tasks are designed to run in parallel as well, the measuring speed can be adaptively tuned by increasing or decreasing the concurrency, depending on the current load levels in the cluster.

It is worth noting that our use of Linux traffic control and u32-based filters is a practical implementation choice rather than a hard dependency of the framework. These primitives are available in Linux distributions and on Kubernetes nodes without requiring kernel changes, making deployment feasible on commodity clusters. The abstractions in the Networking Dynamics Manager, however, are mechanism-agnostic: the classification and shaping logic could equally be implemented using modern data-plane technologies such as eBPF/XDP programs or hardware offload, as long as they expose per-destination delay and bandwidth control. Exploring such backends is part of our future work to reduce emulation overhead further and broaden the range of supported network behaviors.

4.7 Scheduling Policy Extender

The *Scheduling Policy Extender* provides the abstraction and tooling necessary to design, implement, and evaluate customized scheduling policies on top of `iDynamics`. By extending the provided interfaces and leveraging utility functions, users can efficiently develop tailored scheduling strategies that meet the diverse performance, scalability, and reliability requirements of microservice-based applications in cloud–edge environments.

4.7.1 Policy Customization Interface

As illustrated in Figure 4.9, the policy interface is defined by the abstract class AbstractSchedulingPolicy, which establishes the fundamental structure for creating scheduling strategies. This abstract class includes predefined attributes and abstract methods essential for scheduling decisions. Scheduling decisions are encapsulated into combined NodeInfo and PodInfo objects. The NodeInfo class includes node-specific attributes, such as resource capacities (e.g., CPU and memory) and network characteristics (e.g., latency and bandwidth). The PodInfo class contains essential pod details, including resource requests, limits, and SLA requirements (e.g., throughput, response time), and is extensible to accommodate additional metrics.

The AbstractSchedulingPolicy also provides customization methods, such as single-pod scheduling (for single-service optimization), batch-pod scheduling (for multi-service optimization), and an update-metrics method triggered by adaptive scheduling signals. By overriding these abstract methods, researchers and cloud practitioners can incorporate sophisticated decision-making algorithmsranging from heuristics to advanced learning-based techniqueswhile seamlessly integrating their logic with the underlying scheduling framework.

4.7.2 Utility Function Module

The utility function module complements the policy interface, providing ready-to-use functions that simplify and accelerate the extension of custom policies. At the node level,

it includes functions for collecting real-time node resource usage and network metrics, transforming them into structured objects compatible with scheduling algorithms. At the pod level, utility functions help manage pod resource specifications, SLA requirements, and metric unit conversions. Furthermore, additional helper functions utilize monitoring tools such as Prometheus to gather comprehensive cluster-level metrics and network conditions. By leveraging these pre-built utility functions, practitioners and researchers can rapidly and effectively develop sophisticated scheduling policies tailored to specific operational contexts.

4.7.3 Examples of Customized Policies

To demonstrate the expressiveness of the policy interface, we present several types of policies that can be designed to deal with cloud–edge dynamics:

Policy 1: Call-graph–Aware. This policy leverages the call-graph and traffic stress information from the *Graph Dynamics Analyzer*. Microservice pairs with high traffic stress are preferentially co-located on the same node or on nodes with low inter-node latency. This reduces cross-node communication for hot dependencies and mitigates SLA violations caused by communication bottlenecks.

Policy 2: Latency–Aware. This policy focuses on minimizing end-to-end latency for latency-sensitive services by placing them on nodes that exhibit low delays to their main upstream and downstream dependencies. It is beneficial when cross-node latencies are highly variable.

Policy 3: Bandwidth–Aware. For bandwidth-intensive microservices, this policy assigns services to nodes with high available bandwidth to relevant peers and, when possible, co-locates microservices that exchange large volumes of data.

Policy 4: Hybrid-dynamics–Aware. This policy jointly considers call-graph structure, traffic stress, and cross-node delays. It formulates microservice placement as a Service-Node Mapping Problem: microservices with high mutual traffic are mapped to nodes with low mutual delays to minimize total communication cost. This policy illustrates how multiple dynamics can be integrated within a single optimization formulation.

Policy cost models. To make the example policies more concrete, we briefly formalize their underlying objectives using the notation introduced in Sections 4.5 and 4.6. Let $\mathcal{M}$ denote the set of microservices, and $\mathcal{N}$ the set of worker nodes. The Graph Dynamics Analyzer provides a weighted call graph $G = (V, E)$ with directed edges $e = (u \to v) \in E$ and traffic stress stress_σ^μ between upstream microservice μ and downstream microservice σ (Section 4.5). The Networking Dynamics Manager provides the measured cross-node delay matrix $\text{delay}_{i,j}$ between nodes $i, j \in \mathcal{N}$ (Section 4.6). A placement is a mapping $\pi : \mathcal{M} \to \mathcal{N}$ that assigns each microservice to a worker node.

For *Policy 1 (Call-graph–Aware)*, we can define a simple affinity score that quantifies how preferable a candidate node $n \in \mathcal{N}$ is for placing a microservice μ, given a tentative placement $\hat{\pi}$ of its communication partners:

$$\text{score}_1(\mu, n) = \sum_{\sigma \in \mathcal{M} : \hat{\pi}(\sigma) = n} \text{stress}_\sigma^\mu \tag{4.3}$$

Intuitively, $\text{score}_1(\mu, n)$ is high when many of μ's heavily communicating neighbors are already placed on n. In our implementation, Policy 1 greedily assigns or migrates microservices to nodes that maximize $\text{score}_1(\mu, n)$, subject to per-node resource constraints (e.g., CPU and memory capacity), thereby co-locating hot dependencies and reducing cross-node traffic.

For *Policy 4 (Hybrid-dynamics–Aware)*, we introduce an explicit Service–Node Mapping objective that combines traffic stress and cross-node delays. The total communication cost of a placement π is defined as:

$$C(\pi) = \sum_{(\mu,\sigma) \in \mathcal{E}} \text{stress}_\sigma^\mu \cdot \text{delay}_{\pi(\mu), \pi(\sigma)} \tag{4.4}$$

The ideal service placement then solves $\min_\pi \ C(\pi)$, which is subject to per-node resource-capacity constraints.

In other words, microservices with high mutual traffic are mapped to nodes with low mutual delays to minimize the aggregate communication cost along the call graph. In practice, Policy 4 uses this cost model to evaluate and compare candidate placements produced by a greedy search, rather than attempting to solve the underlying combina-

Table 4.1: Injected versus measured cross-node communication delays (ms). Each cell shows the injected delay (left) and the corresponding measured actual delay (right). The results demonstrate successful injection of varied delays from source nodes to multiple destinations, along with reserved channels without injected delays (i.e., Control-plane node and google.com).

Source Node	Destinations										
	k8s-worker-1	k8s-worker-2	k8s-worker-3	k8s-worker-4	k8s-worker-5	k8s-worker-6	k8s-worker-7	k8s-worker-8	k8s-worker-9	Control-plane Node	google.com
k8s-worker-1	- / -	11.00 / 11.39	9.00 / 9.22	16.00 / 16.01	10.00 / 11.21	35.00 / 35.23	24.00 / 24.91	39.00 / 39.15	41.00 / 41.47	- / 0.34	- / 14.90
k8s-worker-2	13.00 / 13.12	- / -	11.00 / 11.31	4.00 / 4.02	7.00 / 7.11	9.00 / 9.14	2.00 / 2.17	6.00 / 6.11	23.00 / 23.89	- / 0.71	- / 15.20
k8s-worker-3	5.00 / 5.02	12.00 / 12.12	- / -	11.00 / 11.08	13.00 / 13.16	11.00 / 11.89	9.00 / 9.75	28.00 / 28.13	43.00 / 43.19	- / 0.52	- / 14.80
k8s-worker-4	5.00 / 5.13	8.00 / 8.14	3.00 / 3.09	- / -	6.00 / 6.81	16.00 / 16.29	12.00 / 12.17	9.00 / 9.14	21.00 / 21.82	- / 0.31	- / 15.30
k8s-worker-5	15.00 / 15.02	6.00 / 6.13	10.00 / 10.38	8.00 / 8.72	- / -	7.00 / 7.13	5.00 / 5.12	8.00 / 8.18	9.00 / 9.13	- / 0.42	- / 14.90
k8s-worker-6	22.00 / 22.13	7.00 / 7.50	8.00 / 8.12	6.00 / 6.11	6.00 / 6.31	- / -	4.00 / 4.17	9.00 / 9.17	21.00 / 21.13	- / 0.45	- / 15.10
k8s-worker-7	22.00 / 22.11	34.00 / 34.11	10.00 / 10.71	8.00 / 8.01	21.00 / 21.81	13.00 / 13.73	- / -	6.00 / 6.12	7.00 / 7.13	- / 0.21	- / 15.20
k8s-worker-8	35.00 / 35.13	6.00 / 6.28	5.00 / 5.33	3.00 / 3.71	5.00 / 5.21	7.00 / 7.05	6.00 / 6.11	- / -	8.00 / 8.09	- / 0.43	- / 15.00
k8s-worker-9	23.00 / 23.13	13.00 / 13.17	18.00 / 18.19	30.00 / 30.03	14.00 / 14.19	24.00 / 24.19	8.00 / 8.11	6.00 / 6.25	- / -	- / 0.33	- / 15.10

Table 4.2: Saturated versus measured cross-node bandwidths (Mbits/sec). Each cell shows the saturated bandwidth first, followed by the actual measured bandwidth, which is emulated and measured by our proposed components in `iDynamics`.

Source Node	Destinations								
	k8s-worker-1	k8s-worker-2	k8s-worker-3	k8s-worker-4	k8s-worker-5	k8s-worker-6	k8s-worker-7	k8s-worker-8	k8s-worker-9
k8s-worker-1	- / -	594 / 608	659 / 665	437 / 452	345 / 359	550 / 560	277 / 300	755 / 754	659 / 667
k8s-worker-2	251 / 267	- / -	512 / 517	270 / 294	432 / 446	404 / 419	274 / 274	625 / 620	386 / 406
k8s-worker-3	721 / 685	512 / 507	- / -	485 / 485	300 / 305	439 / 439	340 / 342	234 / 245	751 / 711
k8s-worker-4	427 / 434	772 / 769	201 / 206	- / -	692 / 682	238 / 242	427 / 449	594 / 598	252 / 253
k8s-worker-5	385 / 376	398 / 397	467 / 467	622 / 606	- / -	288 / 302	501 / 492	502 / 484	683 / 582
k8s-worker-6	675 / 679	779 / 565	247 / 269	229 / 232	484 / 500	- / -	429 / 422	230 / 243	698 / 553
k8s-worker-7	555 / 569	467 / 477	240 / 259	534 / 545	580 / 589	202 / 221	- / -	449 / 466	737 / 722
k8s-worker-8	313 / 330	361 / 336	427 / 442	628 / 548	419 / 434	707 / 675	564 / 528	-	393 / 411
k8s-worker-9	605 / 619	400 / 389	346 / 362	693 / 562	372 / 389	748 / 710	783 / 736	566 / 576	- / -

torial optimization problem exactly, which would be prohibitively expensive at scale.

Note that these example policies are not meant to be exhaustive but show how `iDynamics` can support a broad spectrum of strategies that leverage different aspects of the exposed dynamics.

4.8 Performance Evaluation

In this section, we empirically validate the proposed `iDynamics` framework along three main questions:

- **RQ1:** How accurately and efficiently can `iDynamics` emulate and measure cross-

Table 4.3: Performance comparison between Kubernetes default scheduling and Policy 1 (Call-graphAware) under varying QPS and call-graph dynamics. Metrics include average (Avg), 99th percentile (p99), and standard deviation (Stdev) of response time (ms). SLA violations are highlighted in red, and SLA-compliant improvements in green.

(a) 5 Cluster Nodes (SLA violations (>100 ms))

Policy Evaluation (5 Cluster Nodes)	**Running time**	0–8 min (Call-graph 1)				8–16 min (Call-graph 2)				16–24 min (Call-graph 3)				24–32 min (Call-graph 4, mixed)			
	QPS (req/s)	30	10	50	70	30	10	50	70	30	10	50	70	30	10	50	70
Kubernetes Policy (default)	Avg	34.96	36.07	34.97	34.64	188.5	191.8	191.8	194.4	80.07	77.36	73.68	68.85	63.08	63.07	60.58	59.84
	p99	63.49	70.33	62.40	61.73	336.7	269.3	349.7	373.3	202.1	202.2	200.9	198.9	232.2	281.86	222.3	218.4
	Stdev	6.020	5.930	12.75	10.27	27.22	18.80	36.97	44.79	45.80	40.86	37.75	30.88	53.05	56.06	49.14	48.12
Call-graph-Aware (Policy 1)	Avg	34.74	35.84	34.26	34.30	188.6	63.59	41.54	40.96	46.03	45.87	42.74	41.12	30.00	32.47	29.43	29.35
	p99	60.22	67.90	60.22	60.00	338.4	463.6	73.92	73.47	182.4	182.0	176.0	159.9	104.5	114.7	109.9	104.5
	Stdev	4.220	4.790	4.120	4.560	27.04	75.87	6.100	6.160	40.88	38.50	35.07	30.05	18.38	20.85	18.54	18.48

(b) 10 Cluster Nodes (SLA violations (>150 ms))

Policy Evaluation (10 Cluster Nodes)	**Running time**	0–8 min (Call-graph 1)				8–16 min (Call-graph 2)				16–24 min (Call-graph 3)				24–32 min (Call-graph 4, mixed)			
	QPS (req/s)	30	10	50	70	30	10	50	70	30	10	50	70	30	10	50	70
Kubernetes Policy (default)	Avg	13.27	14.36	13.22	13.22	168.6	171.8	171.2	174.2	57.33	56.06	49.92	46.11	40.67	44.15	39.31	40.15
	p99	20.24	38.91	18.08	18.64	299.1	243.8	315.6	326.9	197.8	203.1	197.3	187.5	207.7	259.7	203.5	203.5
	Stdev	2.460	4.610	2.510	1.880	24.87	20.17	33.17	39.46	48.30	45.73	38.64	31.42	51.43	60.34	49.63	50.49
Call-graph-Aware (Policy 1)	Avg	13.89	14.84	13.51	13.33	168.88	127.6	117.26	118.4	71.32	70.87	68.40	61.83	34.92	39.37	35.20	35.69
	p99	23.85	42.72	17.38	20.00	300.8	594.4	210.7	218.75	309.8	314.4	310.0	305.4	202.2	251.3	183.4	194.3
	Stdev	3.260	5.260	2.100	2.130	25.27	92.80	19.12	22.33	68.42	68.38	63.62	49.84	43.76	51.00	43.75	43.88

(c) 15 Cluster Nodes (SLA violations (>200 ms))

Policy Evaluation (15 Cluster Nodes)	**Running time**	0–8 min (Call-graph 1)				8–16 min (Call-graph 2)				16–24 min (Call-graph 3)				24–32 min (Call-graph 4, mixed)			
	QPS (req/s)	30	10	50	70	30	10	50	70	30	10	50	70	30	10	50	70
Kubernetes Policy (default)	Avg	41.31	43.12	40.83	40.50	231.4	234.45	236.1	241.4	64.34	65.07	62.22	60.61	71.19	74.00	68.90	67.99
	p99	92.10	103.5	89.21	78.14	425.7	360.9	445.4	492.5	98.88	96.64	104.9	108.0	302.8	384.7	278.0	277.8
	Stdev	7.36	10.40	7.27	6.68	38.79	29.23	53.02	62.90	12.41	11.55	11.92	11.69	116.0	74.33	61.18	60.84
Call-graph-Aware (Policy 1)	Avg	41.32	43.25	40.83	40.56	231.6	151.4	129.8	130.4	76.48	70.24	67.46	64.76	57.69	60.89	55.56	55.78
	p99	91.84	98.24	88.77	78.91	418.6	364.5	243.2	240.4	345.6	349.7	325.9	309.8	239.6	293.9	222.1	223.7
	Stdev	7.670	10.19	6.990	7.310	37.46	101.7	25.57	26.73	79.79	71.62	65.01	58.29	44.53	51.83	40.55	40.90

node networking dynamics (latency and bandwidth) in a controllable manner?

- **RQ2:** Can `iDynamics` support the implementation and evaluation of call-graph–aware scheduling policies that react to dynamic call-graphs and workloads and effectively mitigate SLA violations compared with the default Kubernetes scheduler?

- **RQ3:** Can `iDynamics` evaluate hybrid policies that jointly exploit call-graph information and cross-node network conditions across different cluster scales?

Rather than proposing new state-of-the-art scheduling algorithms, our goal is to demonstrate that `iDynamics` (i) provides accurate control over networking dynamics, and (ii) exposes the impact of different scheduling policies under realistic combinations of call-graph, workload, and network dynamics.

4.8.1 Cloud-edge Testbed Setup

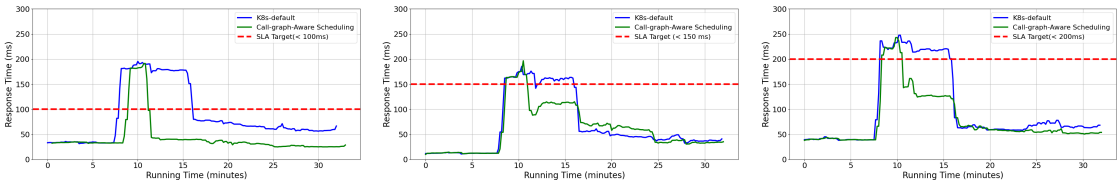
Cluster setup: We implemented and evaluated the proposed `iDynamics` framework on a Kubernetes-based [12] cloud–edge testbed. The cluster consists of one control-plane node and fifteen worker nodes. The control-plane node has 32 CPU cores (AMD EPYC 7763, x86_64), 32 GiB of RAM, and a 16 Gbps network interface. Each worker node has 4 CPU cores from the same processor family, 32 GiB of RAM, and a 16 Gbps link. The software stack comprises Kubernetes v1.27.4 (deployed on ten nodes) and v1.28.4 (on five nodes), Calico v3.26.1 as the CNI plugin, Istio v1.20.3 as the service mesh, and CRI-O v1.27.1 (ten nodes) / v1.28.11 (five nodes) as the container runtime. All nodes run Ubuntu 22.04.2 LTS with the 5.15.0 Linux kernel.

The cluster nodes are virtual machines hosted on the universitys dedicated research cloud, providing ultra-low communication delays between nodes (typically between 0.2 and 1 ms, verified by ICMP ping tests). To emulate realistic and dynamic cloud–edge networking conditions, `iDynamics` injects customized cross-node delays and bandwidth constraints into the cluster nodes. Unless otherwise stated, these conditions are periodically updated during the experiments to evaluate the adaptability of the scheduling policies under varying networking dynamics.

Workload and SLA configuration: We adopted Social Network from DeathStarBench [2] as the microservice application workload and `wrk2` [113] as the request generator. The Social Network benchmark emulates a simplified social media platform. It comprises 27 microservices that collectively support operations such as composing a post, reading the user timeline, and reading the home timeline. Different request types trigger distinct call-graph topologies and traffic patterns, allowing us to stress the *Graph Dynamics Analyzer*.

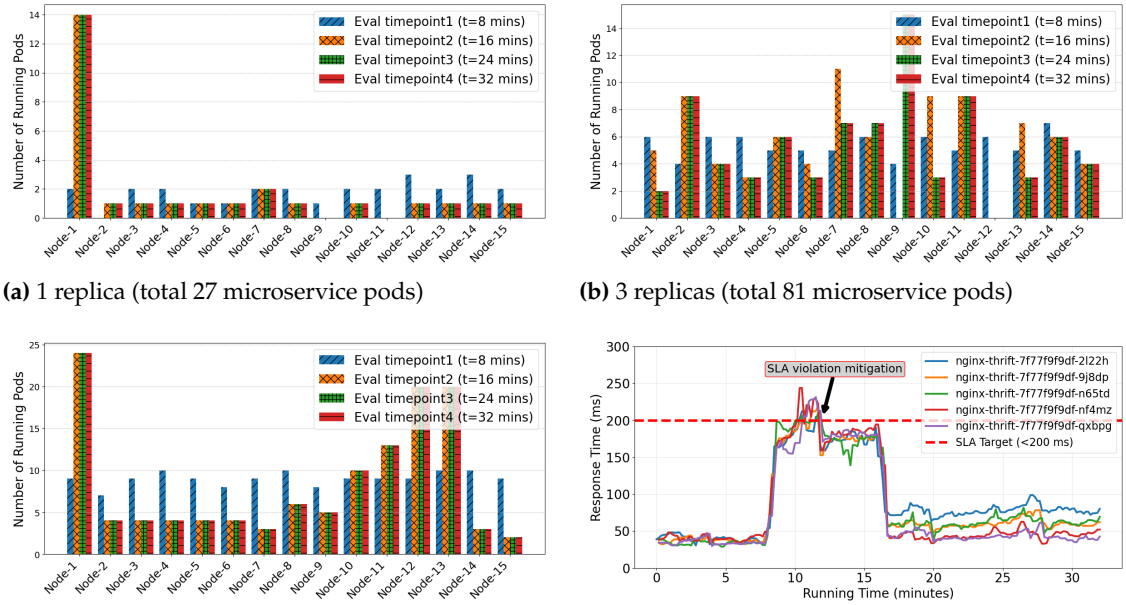
For all policy evaluations, we focus on end-to-end response time for the front-end service (`nginx-thrift`) and report three metrics: average latency (Avg), 99th percentile latency (p99), and standard deviation (Stdev). We define SLA thresholds on average response time that depend on cluster scale: 100 *ms* for five nodes, 150 *ms* for ten nodes, and 200 *ms* for fifteen nodes, matching the thresholds annotated in Table 4.3. For the hybrid-dynamics experiments, a less stringent SLA threshold of 300 *ms* is used to highlight how policies behave under injected high-latency phases.

Replica management and autoscaling: To isolate the effects of the scheduling policies, we disable the Kubernetes Horizontal Pod Autoscaler (HPA) and any other built-in autoscaling mechanisms. Replica counts are fixed within each experiment and are changed only between scenarios (e.g., from 27×1 to 27×3 to 27×5 microservice pods). This ensures that any change in placement or response-time behavior is attributable to the policies implemented via `iDynamics`, not to background autoscaling decisions.



(a) 5 Cluster Nodes **(b)** 10 Cluster Nodes **(c)** 15 Cluster Nodes

Figure 4.10: Under **varying call-graph dependencies**, average response-time comparison between default Kubernetes scheduling and Policy 1 (Call-graphAware) under sustained and varying workloads for (a) 5, (b) 10, and (c) 15 cluster nodes
(Dynamic call graphs ✓, Dynamic queries per second ✓, Dynamic cross-node delays ✗).



(a) 1 replica (total 27 microservice pods)

(b) 3 replicas (total 81 microservice pods)

(c) 5 replicas (total 135 microservice pods)

(d) 5 replicas (SLA guarantee process)

Figure 4.11: In a 15-node cluster, (a)(c) show how the running pods vary across nodes under different replica counts; (d) illustrates SLA-violation mitigation for the 5-replica deployment of the social-network microservice. (Dynamic call graphs ✓, Dynamic queries per second ✓, Dynamic cross-node delays ✗, Varying replicas deployment ✓).

4.8.2 Effectiveness and Validation of Networking Dynamics Manager

To answer **RQ1**, we evaluate the accuracy and overhead of the *Networking Dynamics Manager*, which is responsible for injecting and measuring cross-node latency and bandwidth.

Latency emulation accuracy: We validate the delay-emulation scheme (Figure 4.7b) by injecting distinct delays from each worker node to all other worker nodes and then measuring the actual round-trip time using the distributed measuring agents. Table 5.1 shows, for each source–destination pair, the injected delay and the corresponding measured delay. Across all node pairs, the measured delays closely follow the injected values, with differences typically within about 1 ms. These minor deviations are expected and stem from variability in operating system scheduling and virtualization overheads rather than inaccuracies in the emulator itself. Importantly, we also measure delays from all worker nodes to the control-plane node and to an external host (`google.com`); these remain at approximately 0.5 ms and 14.2 ms, respectively, confirming that non-targeted traffic is unaffected by the injected cross-node delays. In Table 5.1, to show the effectiveness of the destination-oriented delay injection, we did not report the sum of bidirectional delays. When considering the bidirectional sum delays, for each source node s, the bidirectional delay to a destination node d is $\text{delay}^{\leftrightarrow}(s,d) = \text{delay}(s \rightarrow d) + \text{delay}(d \rightarrow s)$. In scheduling evaluation experiments, the distributed measurement agents (Section 4.6.3) directly report the sum of the bidirectional delays.

Bandwidth shaping accuracy: We similarly validate bandwidth shaping by saturating links between node pairs using `iperf3`-based pods. Table 4.2 shows, for each source–destination pair, the configured saturated bandwidth and the measured throughput. The measured throughput consistently tracks the target bandwidth, confirming that the Emulator can partition the cluster into fine-grained bandwidth regions (e.g., high-bandwidth intra-rack links vs. lower-bandwidth cross-rack links) when needed for experiments.

Measurement overhead: The distributed measuring agents are designed to be lightweight and to run in parallel. Each node hosts: (i) a delay-measuring agent built from the `curlimages/curl` image, consuming about 0.2 MiB of memory, and (ii) a bandwidth-measuring agent built from the `networkstatic/iperf3` image, consuming around

0.84 MiB of memory. Measurement tasks are executed concurrently, and the degree of concurrency can be tuned to balance measurement speed and interference. In our experiments, these agents did not saturate CPU or network resources and did not affect application-level performance, confirming that the Networking Dynamics Manager provides accurate dynamic measurements with negligible overhead.

Overall, the results in Tables 5.1 and 4.2 show that `iDynamics` can accurately emulate and measure heterogeneous network conditions, thereby supporting controlled studies of networking-related scheduling policies.

4.8.3 Two Case Studies for Policy Evaluations

To answer **RQ2** and **RQ3**, we now analyze two representative policies implemented using the *Scheduling Policy Extender*: Policy 1 (Call-graph–Aware) and Policy 4 (Hybrid-dynamics–Aware), as introduced in Section 4.7.3. We consider three cluster scales (5, 10, and 15 worker nodes) and different combinations of dynamics. The scheduling logic is implemented via the abstract policy interface and utility functions in Figure 4.9, and scheduling decisions are pushed to Kubernetes through the decision queue mechanism described in Section 4.4 and Section 4.7.

Call-graph–Aware Scheduling Policy Evaluation

In the first case study, we design, implement, and evaluate a Call-graph–Aware policy (Policy 1) that mitigates SLA violations by co-locating microservices with intensive inter-service communication and placing them on nodes with high interconnectivity. Intuitively, this policy minimizes the traffic that traverses cross-node links by using the stress elements and call graphs produced by the *Graph Dynamics Analyzer*.

Dynamic workloads (call graphs, QPS, and replicas) Experiments are conducted under dynamically changing workloads generated by the `wrk2` tool, which creates four distinct call-graph topologies with different proportions of request types. Each topology is exercised under multiple queries-per-second (QPS) levels, generating a matrix of scenarios in which both call graph shape and traffic intensity vary over time. Table 4.3

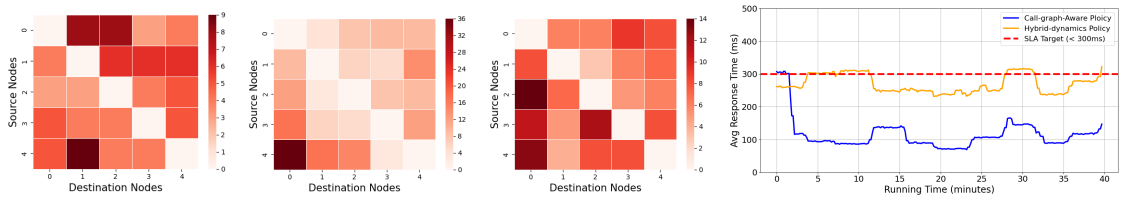
annotates these phases as "Call-graph 1" to "Call-graph 4" and lists the QPS levels (30, 10, 50, 70 req/s).

To examine how Policy 1 behaves as the application scales, we also vary the number of replicas of the Social Network microservice set. Specifically, we evaluate deployments with 27×1, 27×3, and 27×5 microservice pods (i.e., 1, 3, and 5 replicas of each of the 27 microservices), as depicted in Figures 4.11a–4.11c. For a 5-replica deployment, the total number of microservice pods reaches 135, which stresses the schedulers ability to manage complex placement decisions across the 15-node cluster.

Response-time improvements and SLA compliance Figure 4.10 and Table 4.3 compare Policy 1 against the default Kubernetes scheduler under the aforementioned dynamic workloads for 5, 10, and 15 worker nodes. We discuss the main observations:

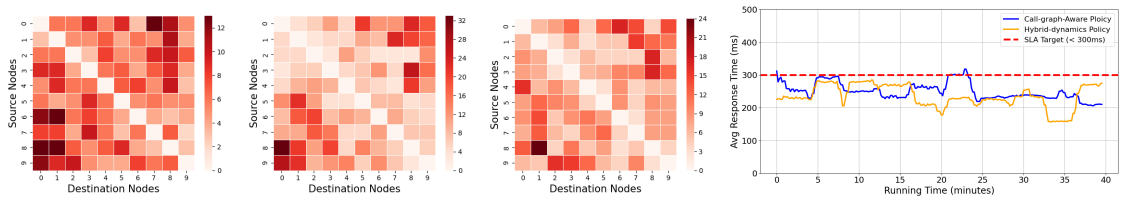
- **Effect under severe call-graph dynamics:** When the call-graph changes to topologies that induce highly imbalanced traffic (e.g., Call-graph 2 and Call-graph 3), the default Kubernetes scheduler exhibits sustained SLA violations. For instance, in the 10-node scenario, once the workload transitions to Call-graph 2, average response time under the default scheduler rises to around 170 ms and remains above the 150 ms SLA threshold across QPS=10–70 until the workload intensity decreases. In contrast, Policy 1 reacts by re-placing pods and brings the average response time down to approximately 120 ms, a reduction of about 2530% that restores SLA compliance.

- **Effect across cluster scales:** The same pattern is observed consistently at 5 and 15 nodes. When the call-graph changes and traffic concentrates on a few microservice pairs, the default scheduler tends to keep communicating services spread across nodes, amplifying cross-node communication delays. Policy 1 systematically pulls those microservices closer together (either onto the same node or onto nodes with lower mutual delays), thereby lowering both mean and tail latencies. The effect is evident in call-graph phases where heavy upstream–downstream pairs emerge, confirming that `iDynamics` can expose the impact of call-graph-aware placement at different scales.



(a) delay matrix 1 (low) **(b)** delay matrix 2 (high) **(c)** delay matrix 3 (medium) **(d)** Two implemented policies manage SLA violations

Figure 4.12: In a 5-node cluster, under sustained workloads and evolving cross-node networking delays (delay matrix 1 ⇒ delay matrix 2 ⇒ delay matrix 3), (d) shows the SLA guarantee process for microservice applications managed separately by Policy 1 (Call-graphAware scheduling) and Policy 4 (Hybrid-dynamicsAware scheduling). (Dynamic call graphs ✗, Dynamic queries per second ✓, Dynamic cross-node delays ✓).

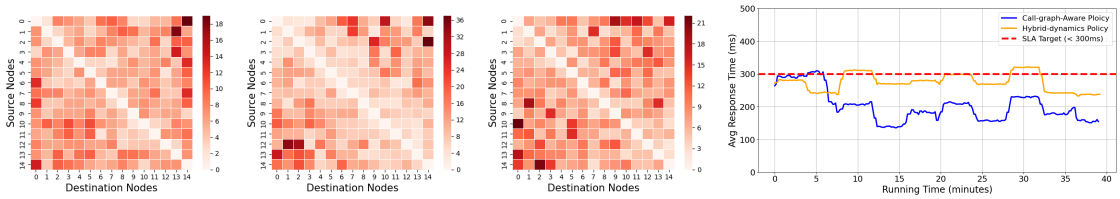


(a) delay matrix 4 (low) **(b)** delay matrix 5 (high) **(c)** delay matrix 6 (medium) **(d)** Two implemented policies manage SLA violations

Figure 4.13: In a 10-node cluster, under sustained workloads and evolving cross-node networking delays (delay matrix 4 ⇒ delay matrix 5 ⇒ delay matrix 6), (d) shows the SLA guarantee process for microservice applications managed separately by Policy 1 (Call-graphAware scheduling) and Policy 4 (Hybrid-dynamicsAware scheduling). (Dynamic call graphs ✗, Dynamic queries per second ✓, Dynamic cross-node delays ✓).

- **Tail latency and variability:** Beyond average latency, Table 4.3 shows that p99 latency and Stdev also benefit from Policy 1. When Policy 1 is active during heavy call-graph phases, p99 latency often decreases by tens to hundreds of milliseconds compared to the default scheduler, and Stdev is reduced as traffic is concentrated on fewer cross-node paths. This indicates that the policy not only improves mean performance but also stabilizes response-time variability, which is important for SLA guarantees.

Runtime redistribution and impact on placement Figures 4.11a–4.11c illustrate how Policy 1 redistributes pods at runtime for different replica counts in the 15-node cluster.



(a) delay matrix 7 (low) **(b)** delay matrix 8 (high) **(c)** delay matrix 9 (medium) **(d)** Two implemented policies manage SLA violations

Figure 4.14: In a 15-node cluster, under sustained workloads and evolving cross-node networking delays (delay matrix 7 ⇒ delay matrix 8 ⇒ delay matrix 9), (d) shows the SLA guarantee process for microservice applications managed separately by Policy 1 (Call-graphAware scheduling) and Policy 4 (Hybrid-dynamicsAware scheduling). (Dynamic call graphs ✗, Dynamic queries per second ✓, Dynamic cross-node delays ✓).

Initially, the Social Network microservices are placed according to Kubernetes default scheduling. As the workload evolves and SLA violations are detected, Policy 1:

1. queries the latest call-graph and stress metrics from the *Graph Dynamics Analyzer*;
2. identifies microservice pairs with high communication stress that are currently hosted on distant nodes; and
3. issues scheduling decisions through the `iDynamics` decision queue, which updates pod placement by adjusting node affinity and triggering controlled pod eviction and re-creation on the selected nodes.

Because these decisions are executed via the Kubernetes API, pods are redistributed without modifying the application code. In the 5-replica experiment (135 pods), Figure 4.11d shows that, around $t \approx 12$ minutes, once SLA violations are detected, Policy 1 migrates several heavily communicating back-end services to less congested or better-connected nodes, thus all the front-end `nginx-thrift` service replicas show decreased response time. After redistribution, the response time of all five replicas (application-side) falls back under the SLA threshold and remains within acceptable bounds for the remainder of the experiment.

It is worth noting that Policy 1 intentionally focuses on communication-aware placement and does not enforce strict load balancing across nodes. As a result, the number of

pods per node can become moderately imbalanced (Figures 4.11a–4.11c). This behavior is consistent with its design and highlights how `iDynamics` can reveal trade-offs between communication efficiency and resource balance, which more advanced policies could explore in future work.

Hybrid-dynamics–Aware Policy Evaluation

The second case study evaluates Policy 4 (Hybrid-dynamics–Aware), which integrates both call-graph dynamics and cross-node latency into a unified Service–Node Mapping cost. This policy uses not only the stress elements between microservices but also the measured delay matrix between nodes (Section 4.6) to choose placements that jointly minimize traffic and network latency.

Dynamic workloads (QPS and networking conditions) To focus on the interaction between scheduling and network dynamics, we fix the call-graph (dominated by a representative request mix) and expose the application to three successive cross-node delay matrices of increasing and then decreasing severity: low $\Rightarrow$ high $\Rightarrow$ medium. These matrices are generated via Algorithm 4.2 by using a base latency of $bl = 5\,\text{ms}$ and varying the maximum additional latency ($mal \in \{10, 30, 20\}$ ms). Figures 4.12a–4.14c show the resulting heatmaps for 5, 10, and 15 nodes, respectively. For each cluster size, we run sustained workloads at fixed QPS levels while gradually switching the delay matrix, thereby stressing the policies with abrupt changes in network conditions.

Comparison between Policy 1 and Policy 4 Figures 4.12–4.14 compare the behavior of Policy 1 (call-graph–only) and Policy 4 (hybrid) under these evolving delay matrices. We highlight the main observations:

- **SLA maintenance under injected delay spikes:** For all three cluster sizes, both policies can respond to increases in cross-node latency and restore the average response time to the target SLA of 300 *ms*. When the delay matrix switches from low to high, the average response time temporarily spikes because many communicating microservices suddenly experience longer paths. After `iDynamics` signals

an SLA violation, each policy computes new scheduling decisions; within a short period, the average latency is reduced again below the SLA threshold.

- **Benefit of incorporating measured delays:** Policy 1 does not explicitly optimize for cross-node latency; it indirectly benefits from network information only because co-locating heavily communicating services reduces the number of cross-node hops. Policy 4, in contrast, incorporates the delay matrix directly into its cost model. As a consequence, the response-time spikes following each delay-matrix change are generally shorter and less pronounced under Policy 4 than under Policy 1. This is especially evident in the 10- and 15-node clusters, where the space of possible placements is larger and the choice of low-latency node pairs yields more gains.

- **Scalability across cluster sizes:** The qualitative behavior of Policy 4 remains stable as we move from 5 to 10 to 15 nodes. In all cases, the policy uses the same inputs (call-graph stress and measured delays) and the same decision interface, yet it successfully restores SLA compliance after considerable changes in network conditions. This suggests that the combination of the Networking Dynamics Manager and the Scheduling Policy Extender scales to larger clusters without requiring policy re-implementation.

Overall, the hybrid-dynamics policy demonstrates how `iDynamics` can support policies that jointly reason about application-layer and infrastructure-layer dynamics, and how the framework exposes the impact of such policies in controlled, repeatable networking scenarios.

4.8.4 Findings and Limitations

Findings

Across all experiments, we obtain the following high-level conclusions:

- The *Networking Dynamics Manager* can accurately inject and measure heterogeneous cross-node delays and bandwidths with low overhead, enabling controlled

experiments on network-aware policies.

- The *Graph Dynamics Analyzer* and *Scheduling Policy Extender* together allow the implementation of call-graph–aware policies such as Policy 1, which significantly reduce SLA violations compared with the default Kubernetes scheduler under dynamic call-graphs and workloads.

- Hybrid policies such as Policy 4 can be implemented by augmenting the cost model with measured network conditions, and `iDynamics` can reveal their behavior under combinations of workload and network dynamics at different cluster scales.

Limitations

Our evaluation also has limitations:

- **Benchmark diversity:** We use the Social Network benchmark as a representative microservice application. While it captures complex call graphs and traffic patterns, other applications (e.g., event streaming or ML inference pipelines) may exhibit different dynamics. Nevertheless, the framework is application-agnostic; additional workloads can be plugged in without changes to `iDynamics`.

- **Platform specificity:** The implementation targets Kubernetes with Calico and Istio. The design, however, is not tied to a specific orchestrator or service mesh. Porting `iDynamics` to other platforms (e.g., Nomad, Docker Swarm, or different CNIs) would require engineering effort but no fundamental changes to the core concepts.

- **Policy optimality:** The evaluated policies are intentionally simple and are used to demonstrate the expressiveness of `iDynamics`, not to claim algorithmic optimality. More sophisticated policies (e.g., solving the underlying Service–Node Mapping optimization more aggressively) could achieve stronger performance improvements; we leave such exploration to future work.

Despite these limitations, the evaluation shows that `iDynamics` achieves its primary goal: providing a controllable and extensible experimentation environment to study microservice scheduling policies under realistic and multi-dimensional dynamics.

4.9 Summary

This chapter tackled the evaluation problem for microservice scheduling policies under realistic yet controllable dynamics in cloudedge environments. We observed that microservice workloads are driven by three intertwined factors: changing call-graph structures as request mixes vary, highly imbalanced bi-directional traffic along service chains, and time-varying cross-node latency and bandwidth. Existing simulators and emulators either focus on infrastructure-level dynamics or on application workloads, but seldom allow researchers to jointly control call-graph, traffic, and network conditions while executing real microservice code on a production-grade orchestration stack.

Overall, `iDynamics` provides the methodological support for the thesis: it makes call-graph, traffic, and network dynamics explicit, controllable, and repeatable on an authentic orchestration stack. This controllability allows policy behaviour to be studied under well-defined dynamic scenarios rather than ad-hoc testbed conditions. Building on the abstractions and measurements enabled by `iDynamics`, the next chapter introduces TraDE, which uses bidirectional traffic stress and destination-specific delay information to perform *runtime rescheduling* that mitigates QoS violations under dynamic cross-node conditions.

Chapter 5

Network- and Traffic-aware Adaptive Placement for Microservices under Dynamics

Efficiently scheduling containerized microservices in a cluster is increasingly challenging due to dynamic workloads, complex service dependencies, and time-varying cross-node communication delays. This chapter introduces `TraDE`*, a network- and traffic-aware adaptive scheduling framework that continuously monitors traffic stress between dependent microservices and dynamically redeploys microservice instances in response to QoS violations.* `TraDE` *builds a traffic-stress graph from service-mesh telemetry, uses a Dynamics Manager to inject and measure controllable cross-node delays via packet-level manipulation and distributed measuring agents, and employs a Parallel Greedy Algorithm (PGA) Mapper and a microservice rescheduler to compute and enact low-overhead service-to-node mappings while preserving service availability. Implemented as an extension to Kubernetes and evaluated using the deployed application under a variety of mixed workloads and dynamic delay matrices,* `TraDE` *reduces average response time by up to 48.3%, improves throughput by 1.2 to 1.5x, and achieves 95.36% goodput compared with lower success rates for baseline methods, demonstrating robust QoS compliance under sustained workloads and changing network conditions.*

5.1 Introduction

In the pervasive cloud computing domain, microservices have emerged as a key architecture, revolutionizing how cloud-based applications are designed and implemented.

This chapter is derived from:

- **Ming Chen**, Muhammed Tawfiqul Islam, Maria A. Rodriguez, and Rajkumar Buyya, "TraDE: Network and Traffic-Aware Adaptive Scheduling for Microservices Under Dynamics", *IEEE Transactions on Parallel and Distributed Systems*, Volume: 37, Number: 1, Pages: 76-89, January, 2026.

Characterized by the modular and decentralized design, microservices provide good flexibility and scalability, catering to the dynamic demands of modern cloud-deployed applications [6, 7, 166]. However, this transition also introduces significant challenges in microservice management and performance optimization, particularly microservices with complex workflow and dependencies [20, 53, 122, 127]. Uncertainty in user requests, combined with varying execution paths across microservices and differing communication delays between nodes, significantly increases the difficulty of ensuring Quality of Service (QoS) compliance for microservices deployed on cluster nodes. In cloud environments, numerous microservices are co-located, interact with each other, and are often shared among various applications [19][121]. This can lead to worsened performance degradation due to resource contention and cascading delays in the execution paths of requests. These issues directly affect latency and throughput, both critical for meeting QoS targets.

Although existing works have proposed different methods to improve the running performances of containerized microservices, there are still some limitations to these methods. OptTraffic [33] optimizes the traffic transmission of containerized microservices across cluster nodes, which fails to consider the cross-node delays under dynamic networking traffic and also introduces complexity by calculating every dependent container replica pair. NetMARKS[32] determines Kubernetes pod scheduling by the dynamic network metrics collected by Istio Service Mesh [42], which have limitations on analyzing the bidirectional metrics between dependent microservices and also may introduce imbalanced load distributions across the cluster nodes. Other existing methods[31, 34, 35, 134] also have similar limitations on tackling dynamic workloads, the awareness of cross-node delays, and imbalanced load distributions in the cluster.

To solve the aforementioned challenges, this chapter proposes `TraDE`, a novel framework that utilizes a traffic and network-aware rescheduling approach. `TraDE` is designed to adaptively redeploy containerized microservices within the cluster by analyzing real-time traffic stress between dependent microservices along with the variations of cross-node delays. By doing so, it seeks to mitigate QoS target violations amid fluctuating user requests and network variations. To implement and evaluate the proposed `TraDE` framework, this chapter seeks to address several crucial challenges in dealing

with network dynamics for meeting application services' QoS targets: (1) How to quantitatively map dynamic bidirectional traffic patterns into traffic stress between upstream and downstream microservices, including all the corresponding replicas?, (2) What approach should be employed to build the traffic stress graph under dynamic workloads within a specific time interval?, (3) How to design a controllable network-dynamics manager to thoroughly evaluate the proposed method via efficiently injecting dynamic cross-node delays and accurately measuring the node-delay matrix?, (4) With the constructed traffic-stress graph and the measured cross-node delay matrix, how to determine the service-to-node mapping under minimal exploration time and a balanced load goal?, (5) With the explored service-to-node mapping result, what strategies should be adopted to ensure zero downtime of the running services when migrating related microservice containers?

The proposed `TraDE` framework resolves these challenges and could adapt to the changing traffic conditions and redeploy the running microservice containers to server nodes when QoS violation is detected. Specifically, `TraDE` builds a traffic stress graph for dependent microservices, a lightweight cross-node delay monitor, a low overhead service-node mapper, and a microservice rescheduler with guarantees of service availability when migrating containers. We demonstrate the effectiveness of our approach through extensive evaluation using practical microservice applications. The results of our experiments, conducted under a variety of scenarios, demonstrate the ability of `TraDE` to maintain the desired QoS target of deployed microservice applications when QoS violations happen. In summary, our main contributions can be summarized as follows.

- We propose a traffic analyzer that dynamically constructs a traffic stress graph. This graph not only illustrates the latest microservice call graphs (dependencies) but also identifies the microservice pairs experiencing higher stress via bidirectional traffic analysis.

- We design and implement a network-dynamics manager which mainly consists of injecting customized cross-node delays in a controllable way via packet-level tagging and also accurately measuring the node communication delays via cluster-

level daemon agents.

- We introduce a parallel algorithm for service-node mapping to minimize the total traffic transmission overhead with guaranteed balanced task chunks and fast convergence.

- We design a microservice rescheduler to migrate microservice instances that experience QoS violations. Specifically, when migrating the microservices, we employ various scheduling schemes to guarantee service availability and resource availability in the cluster.

- We develop a prototype system of the `TraDE` framework as an extension to the Kubernetes platform, and demonstrate system performance in a real computing cluster.

What is new compared to prior work? TraDE (i) models *bidirectional* traffic to expose overloaded pairs as rescheduling targets; (ii) introduces a *controllable*, destination-specific delay generator paired with a multi-agent measurer for accurate cross-node latency under dynamics; and (iii) couples these with a service-node mapper that embeds an overload penalty and preserves availability during service migrations. Together, these enable fast, practical, and repeatable QoS-target compliance under changing traffic and network conditions.

The rest of the chapter is structured as follows: Section 2 discusses related work, providing a comprehensive background on existing methods. The motivation and problem statement are introduced in Section 3. Section 4 presents the proposed `TraDE` framework, explaining its main components. Section 5 focuses on the design of the traffic analyzer. Section 6 details the design of the dynamics manager, which consists of the dynamic delay generator and the cross-node delay measurer. Section 7 introduces the proposed PGA algorithm for microservice placement, along with an overhead analysis. Section 8 offers a performance evaluation and analysis of the proposed `TraDE` framework. Finally, Section 9 concludes the chapter with summary.

5.2 Related Work

The scheduling of microservices in cluster environments has been extensively studied from various perspectives. This section reviews the existing literature, focusing on microservice management, graph analysis, and network-aware scheduling.

5.2.1 Microservice Management

FIRM [18] leverages online telemetry metrics data and machine learning-based models to manage microservices in a fine-grained way by localizing the SLO violations, identifying resource contentions, and then taking reprovision measures to mitigate the SLO violations. Erms [19] builds resource scaling models to calculate the latency objectives for shared microservices with large calling graphs. GrandSLAm [109] estimates the completion time of the requests for individual microservice execution stages and leverages the estimated time to batch and reorder the requests dynamically. However, these existing methods have limitations in considering the dynamic impacts of cross-node communication delays and the changing bidirectional traffic between upstream and downstream microservices with multiple replicas.

5.2.2 Graph Analysis

Sage [30] builds a graphical-based bayesian model to analyze the root cause of cascading QoS violations for interactive microservices focusing on practicality and scalability. Tian et al [138] develop a workload generator to synthesize the DAG jobs with graph workflow by analyzing large-scale cluster traces. Furtherly, Luo et al [123] characterize the call graph of dependent microservices by analyzing Alibaba cluster data and reveal three types of calling dependency graphs for microservice applications. Parslo [29] introduces a gradient descent-based method by breaking the end-to-end SLO budget into smaller units to assign partial SLOs among nodes in a microservice graph under an end-to-end latency SLO. However, these methods are generally time-consuming with high overheads to build the graph and not suitable for dynamic incoming user requests.

5.2.3 Network-aware Scheduling

NetMARKS [32] introduces a network-aware approach to schedule the Kubernetes pods from different 5G edge applications by using the collected dynamic metrics from Istio Service Mesh. Marchese et al [35] introduce a network-aware scheduling extension for the default Kubernetes scheduler by considering the infrastructure network conditions and the interactions among microservices. OptTraffic [33] develops a network-aware scheduling framework by optimizing the cross-machine traffic scheduling for multi-replica microservice containers by migrating the containers with dependent relations. However, these networking-aware scheduling schemes designed for microservices still have limitations on bidirectional traffic analysis for dependent microservice replicas and the changing infrastructure-level conditions, i.e. varied cross-node communication delays.

5.3 Motivation and Problem Statement

5.3.1 Background

An increasing number of modern cloud applications have evolved into microservice-based architectures, which manage applications through a collection of containerized, loosely coupled, fine-grained services [123] [2]. As discussed in previous work [2], transitioning from monolithic designs to microservice designs for cloud applications leads to a higher proportion of processing time being spent in the networking stack compared to monolithic applications. The processing time percentages of different cloud applications are shown in Fig. 5.3a.

Deployed as a set of containerized microservice instances, the application processes user requests through dependent microservice replicas and returns the processed results in the reverse direction. As shown in Figure 5.1, a microservice-based application is decoupled into a collection of containerized microservices along with the bidirectional traffic flows between dependent microservices. Every microservice is supported by one or multiple corresponding container instances to provide the application functionality. These container instances launch from pre-defined container images specified in the cor-

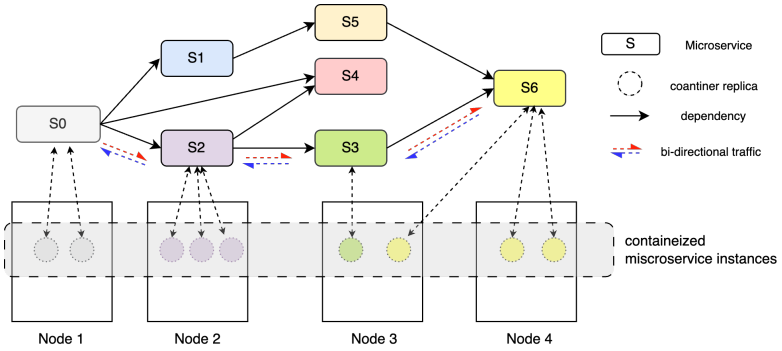
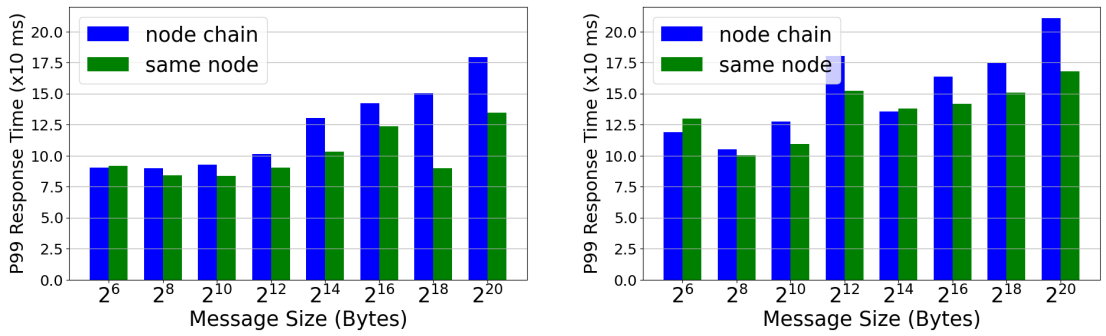


Figure 5.1: An envisioned workflow of containerized microservice executions and communications in a cluster with four nodes.

responding microservice deployment file.

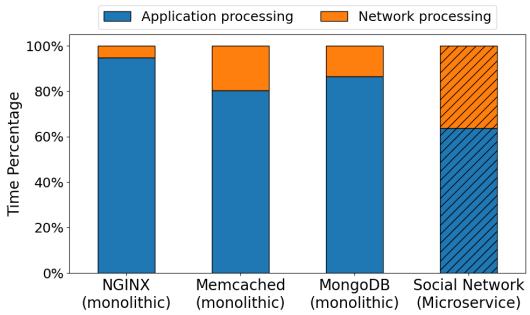
5.3.2 Motivation



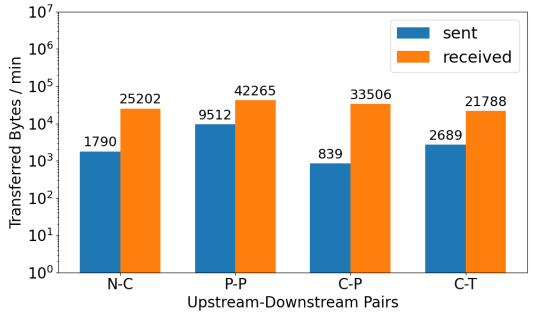
(a) QPS=3000. **(b)** QPS=5000.

Figure 5.2: P99 response time of different QPS from an *upstream* microservice client and a *downstream* microservice server under scenarios where they are either colocated on the same node or running across a node chain.

We deployed synthetic microservice applications in a running cluster to motivate our design and tested them under different scenarios. The deployed applications include a server/client application and a benchmark application. Figure 5.2 shows the performance of two dependent microservices: one functions as a client container sending `PUT` requests with varying message sizes from 2^6 Bytes to 2^{20} Bytes, and the other operates as the corresponding server container to receive the requests. From Figure 5.2a and Figure



(a) Processing comparisons of monolithic and microservice applications (data source [2]).



(b) Bidirectional traffic between UM-DM pairs in Social Network in one minute.

Figure 5.3: (a) Comparison of processing time percentage between monolithic and microservice applications. (b) Traffic of different UM-DM pairs in one minute.

5.2b, we observe that colocating two dependent microservice containers on the same node notably improves the p99 response time. Additionally, the QPS (Queries Per Second) has a significant impact on dependent microservice containers. As shown in Figure 5.2, when QPS increases from 3000 to 5000, the p99 response times for each transferred message size also increase, with a maximum 78.2% performance degradation when the sent message size is 2^{12} Bytes. This implies that, for an *upstream-downstream* pair, the transferred message size, QPS, and cross-node communications have notable impacts on end-to-end performance.

Additionally, we deployed the Social Network benchmark released with DeathStarBench [2] to quantify the transmitted traffic difference between dependent UM-DM pairs. Under different request workloads, we observed notable traffic differences between different UM-DM pairs.

As shown in Figure 5.3b, there are significant differences between sent and received traffic for certain UM-DM pairs, including N-C (`Nginx`→ `Compose-post`), P-P (`Post-storage-service` → `Post-storage-MongoDB`), C-P (`Compose-post` → `Post-storage-service`) and C-T (`Compose-post` → `Text-service`) [2]. In our testing scenario, for example, the bidirectional traffic between the C-P pair shows that the received traffic could be approximately 40x more than the sent traffic, implying that a slight delay increase in transmission among the C-P (pair could degrade the pair's end-to-end performance by up to 40x. Thus, it is significant to analyze the real-time

bidirectional traffic for dependent microservices.

From the above observations, we can conclude that: (1) Unlike monolithic cloud applications, microservice-based applications spend significantly more time on the networking processing stack. (2) Cross-node communication can impact the performance of microservice pairs, particularly for pairs with high volumes of traffic transmission. (3) For dependent *upstream* and *downstream* microservice pairs, the transmitted bidirectional (sent and received) traffic can exhibit significant differences, such as small requests but large payloads.

5.3.3 Problem Definition

In dynamic networking and traffic environments, the deployment of microservice instances across different server nodes plays a critical role in determining the end-to-end performance of distributed applications. In particular, poor placement decisions may incur high communication overhead due to inter-node delays and traffic volumes. To address this, we formulate the optimization task as a **Service-Node Mapping Problem**, where the objective is to assign a set of microservices to a set of server nodes in a manner that minimizes overall communication costcapturing both communication latency and traffic-induced overheadwhile satisfying server resource constraints.

Let $M = \{m_1, \ldots, m_k\}$ be the set of microservices, $N = \{n_1, \ldots, n_p\}$ the set of server nodes, and $P : M \to N$ the servicenode mapping. Over a measurement window Δt, let $T_{i\to j} \geq 0$ and $T_{j\to i} \geq 0$ denote the forward and reverse traffic (bytes or rate) between m_i and m_j. Let $D_{a,b} \geq 0$ be the one-way (ping-like) delay from node a to node b (not necessarily symmetric). We use direction weights $w_f, w_b \in [0, 1]$ (default w_f=w_b=0.5).

The pairwise communication cost for (i, j) under a placement P is

$$\begin{cases} \mathcal{C}_{i,j}(P) \;= w_f \, T_{i\to j} \, D_{P(m_i),P(m_j)} \;+\; w_b \, T_{j\to i} \, D_{P(m_j),P(m_i)}, \\ \qquad\qquad (w_f, w_b) \in [0, 1]^2. \end{cases} \tag{5.1}$$

and the total communication cost aggregates pairwise costs:

$$\text{TotalCost}(P) = \sum_{i=1}^{k} \sum_{j=1}^{k} \mathcal{C}_{i,j}(P). \tag{5.2}$$

Each node $n \in N$ has a capacity vector C_n (e.g., CPU, memory, GPU), and each microservice i has a resource demand vector R_i. A placement is feasible if, element-wise,

$$\sum_{i:\, P(m_i)=n} R_i \;\leq\; C_n \qquad \forall\, n \in N. \tag{5.3}$$

Penalty form used in our solver. We enforce (5.3) via a soft penalty, consistent with our implementation. Let $\Phi(P) = \sum_{n \in N} \max\{0, \sum_{i:P(i)=n} R_i - C_n\}$ denote the aggregated overflow (applied element-wise and summed across resources), and let $\lambda > 0$ be a large coefficient. The penalised objective is

$$\min_{P} \;\; \text{TotalCost}(P) \;+\; \lambda\, \Phi(P), \tag{5.4}$$

which is equivalent in practice to the constrained form for sufficiently large λ.

Why bidirectional traffic? An RPC is typically a requestresponse round trip; even with one-way (ping-like) delays, both directions (forward $i \to j$, return $j \to i$) contribute to user-perceived time and to total network time over Δt. Equation (5.1) therefore weights both directions.

When one-way delays are symmetric ($D_{a,b} = D_{b,a}$), (5.1) reduces to:

$$(w_f T_{i \to j} + w_b T_{j \to i})\, D_{P(m_i),\, P(m_j)}.$$

Setting $(w_f, w_b) = (1, 0)$ yields the one-direction variant.

Implementation in TraDE. In our prototype system, $T_{i \to j}$ and $T_{j \to i}$ are computed from Istio byte counters over Δt; we take w_f=w_b=0.5 by default and optimise (5.4) using a parallel greedy search.

The problem is thus to find a mapping P that minimizes (5.2) (or equivalently (5.4)) subject to (5.3), promoting traffic locality and reducing cross-node latency under dynamic workloads and network conditions.

5.4 Framework of TraDE

Based on our observations, we were motivated to design TraDE, a network and traffic-aware rescheduling framework for containerized microservices when the deployed service experiences QoS violations due to dynamic requests and varying cross-node communication delays in dynamic computing environments. As shown in Fig. 5.4, the figure illustrates the main modules and how each module works together at different stages to complete the adaptive scheduling process. The key modules of the proposed framework are as follows:

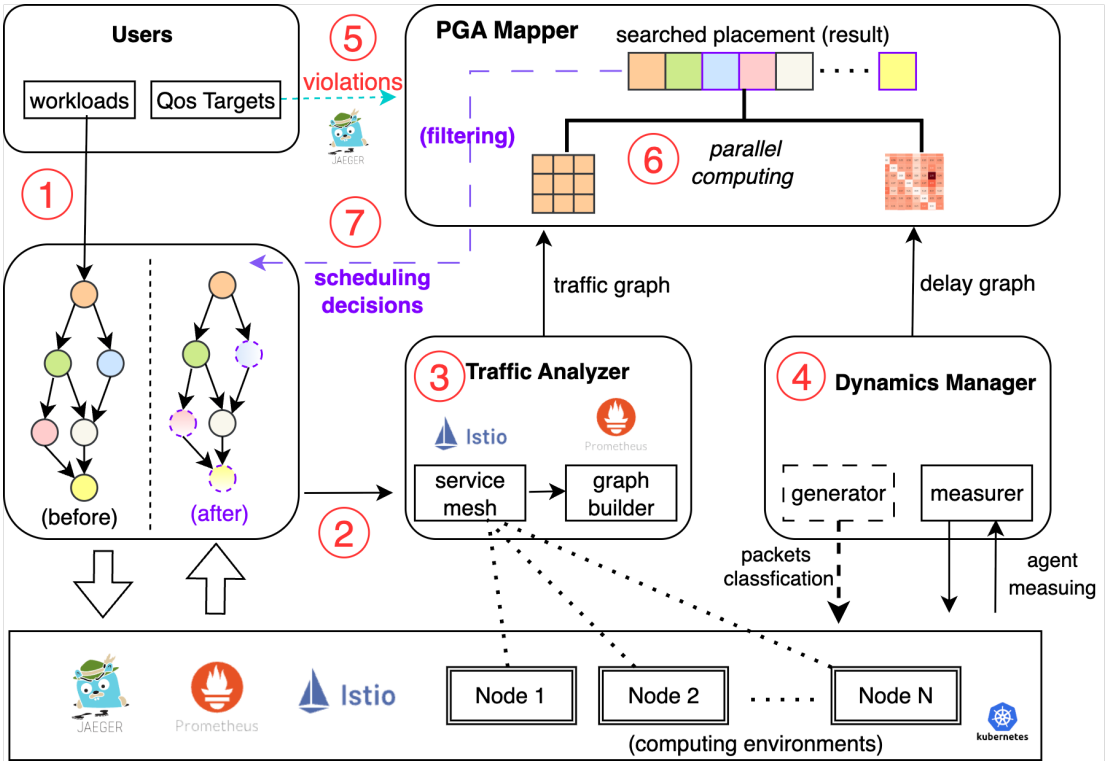


Figure 5.4: The proposed TraDE framework.

- **Traffic Analyzer**: This module consists of a service mesh and a graph builder, which are designed for analyzing the real-time bidirectional traffic flows between the dependent *upstream* microservice containers (including all corresponding replicas) and *downstream* microservice containers through the service mesh and the proposed graph builder algorithm.

- **Dynamics Manager**: This module consists of a *generator* and a *measurer* to manage the cross-node communication delays in the computing environments. The *generator* is designed to generate different practical delays to the computing nodes to validate the proposed `TraDE` through packet-level manipulation. The *measurer* is designed to measure the communication delays between different nodes across the computing environments through multiple daemon agents.

- **PGA Mapper**: This module is designed to tackle QoS violations of the running applications. Specifically, when there are violations to the predefined QoS targets, the PGA (Parallel Greedy Algorithm) mapper computes the service placements on the computing nodes to achieve the lowest overhead as defined in Eq.5.2.

As shown in Fig. 5.4, at the beginning stage ①, users define the QoS targets and send different workloads (i.e., different types of requests and QPS) to the deployed microservice applications. At stages ② and ③, the performance metrics of the running microservices are collected by `Jaeger` and `Istio`. Within a predefined monitoring time interval, the traffic analyzer analyzes the bidirectional traffic between dependent microservices and builds the traffic graph, which is then sent to the PGA mapper. Meanwhile, at stage ④, the node dynamics manager measures the cross-node communication delay graph, which is sent as another graph to the PGA mapper. As the microservice application runs, if there are any QoS violations (at stage ⑤) to the predefined QoS targets, `TraDE` is triggered to run the PGA mapper. As shown at stage ⑥, the PGA mapper finds the service-node placement through a constructed traffic graph and delay graph from the traffic analyzer and dynamics manager, respectively. Designed to run in a parallel computing manner to speed up the process, the PGA mapper provides the searched placement result, specifying the placement of each microservice to a node. Placement results are filtered before being used as the scheduling decision, as some microservices are already placed in the optimal node, and some microservice instances, such as Jaeger agent instances, are not supposed to be migrated. At the final stage ⑦, the filtered placement results are taken as the adaptive scheduling decision in response to the current dynamic computing environments.

5.5 Design of Traffic Analyzer

5.5.1 Service Mesh

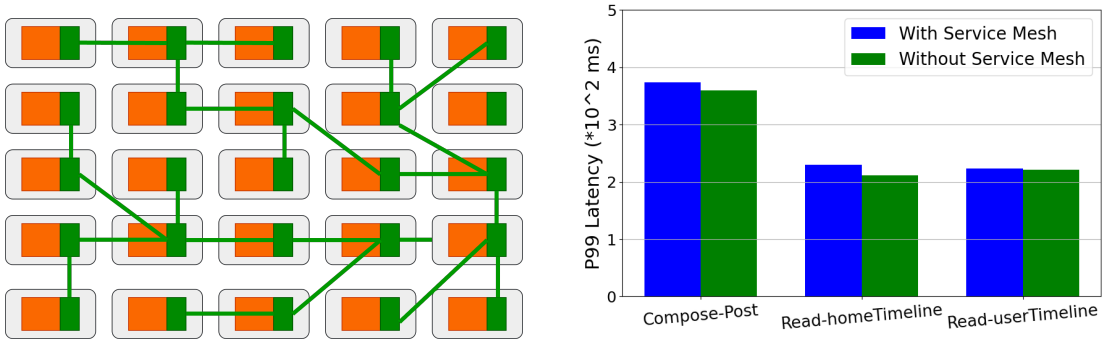
The key aspect of implementing the Traffic Analyzer for `TraDE` lies in efficiently and minimally analyzing the bidirectional traffic between dependent container pairs, i.e., the upstream and downstream microservice containers. When there is only one microservice instance for both upstream and downstream services, it is straightforward to collect and analyze bidirectional traffic metrics from the Linux *proc* file system. However, when multiple replicas exist for either the downstream or upstream microservices, analyzing the bidirectional traffic between all upstream and downstream microservice replicas becomes time-consuming. Thus, we implemented the service mesh to better observe and manage the complex traffic flows.

Istio Service Mesh Implementation

To obtain finer-grained metrics of bidirectional traffic between upstream microservices and their downstream counterparts, along with their corresponding replicas, we implemented *Traffic Analyzer* by analyzing traffic metrics with Istio Service Mesh [42]. The service mesh is designed as a dedicated infrastructure layer that can be added to containerized microservice applications. With the service mesh deployed, the traffic for each microservice container is proxied by an injected sidecar container.

In a Kubernetes-based cluster with the deployed Istio service mesh, a microservice pod comprises both the application and sidecar containers. As depicted in Figure 5.5a, the dedicated service mesh structure proxies the ingress and egress traffic among the microservice containers. The orange rectangles represent the containerized microservices, and the green rectangles represent the sidecar containers, which function as a dedicated layer managing traffic among various microservices. The green links illustrate the data traffic connections and dependency relationships between microservices. Specifically, as depicted in Figure 5.6, the `Envoy` [43] containers (acting as sidecar containers) in Pod A and Pod B proxy the bidirectional traffic between Container A and Container B. Within each pod, the `Envoy` container communicates with its corresponding microservice con-

tainers.



(a) Service mesh graph.

(b) p99 tail latency.

Figure 5.5: (a) An overview of how service mesh with sidecar containers (green) works with microservice containers (orange). (b) Overhead comparisons of p99 tail latency of different workloads to Social Network applications with and without service mesh.

Overhead Analysis

We realize that adding an additional service mesh layer to the microservice application may introduce extra delays in the traffic flow. However, the introduced delay only contributes a slight proportion to tail latency. Based on [140], within the Istio 1.21.2 service mesh, utilizing telemetry v2, each request is processed by both a client-side and server-side Envoy proxy. These proxies collectively increase latency at the 90th percentile by approximately 0.182 milliseconds and at the 99th percentile by about 0.248 milliseconds, compared to the baseline latency of the data plane. These findings were conducted with a 1kB payload, a rate of 1000 requests per second, and varying client connections (2, 4, 8, 16, 32, 64) at the CNCF Community Infrastructure Lab [169].

Additionally, we evaluated and compared the performance of the Social Network [2] with and without the implemented service mesh. We separately deployed the Social Network application in two different namespaces to guarantee the requests and services would not interfere with each other. Three types of requests (i.e., Compose-Post, Read-homeTimeline, and Read-userTimeline) were sent as the workloads to the deployed Social Network application, each with a QPS of 200. As depicted in Figure 5.5b, it can be observed that the service mesh introduces only minimal delays. Therefore, the service

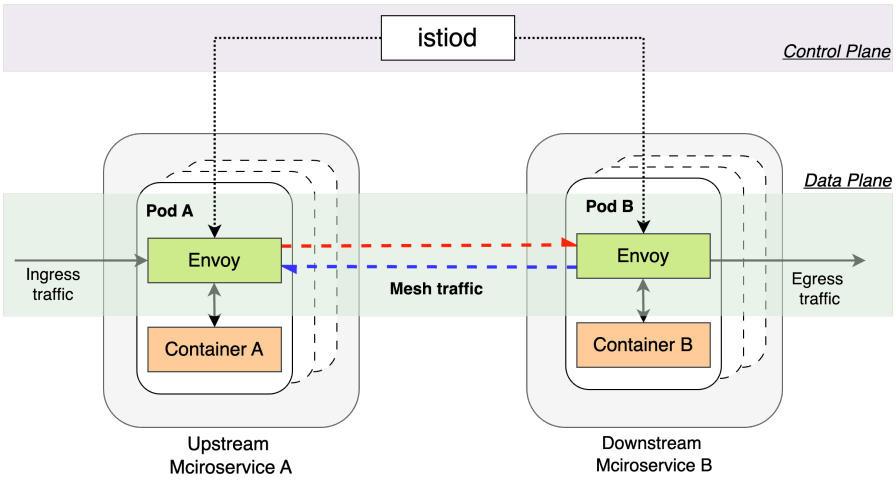


Figure 5.6: An architecture of how upstream microservice A interacts with downstream microservice B with Istio service mesh enabled.

mesh layer does not add significant overhead to the network traffic of the microservice application. As the tradeoff to better manage the complex traffic flows, the introduced minimal delays are acceptable.

5.5.2 Traffic Stress Graph

One of the main goals of the proposed `TraDE` is to construct real-time traffic flows along the triggered call graphs of the deployed microservices. Multiple request types and varied QPS would trigger different structures of call-graph with different amounts of traffic flows, which complicates the adaptive scheduling of microservices in response to QoS violations. In our work, we define the following terminology.

Stress Element: Stress Element (SE) defines the dependencies between two dependent microservices, i.e., the UM (Upstream Microservice) and DM (the Downstream Microservice), and the traffic stress between the two corresponding microservices. The traffic stress of a SE is calculated by the average traffic of *sent* and *received* in a given time interval, as we believe that the traffic of *sent* and *received* between the dependent microservices both contribute significant stress to the end-to-end performance of the deployed application. Written mathematically, the stress of a Stress Element can be expressed by:

$$\text{stress}_{\sigma}^{\mu}(\mu^{UM}, \sigma^{DM}, \Delta t) = \frac{\text{Bi-direction\_traffic}(\mu^{UM}, \sigma^{DM})}{2\Delta t} \tag{5.5}$$

In Eq. 5.5, μ and σ refer to the dependent upstream and downstream microservices. Δt defines the measurement time interval for microservices μ and σ.

The term

$$\text{Bi-Direction\_traffic}(\mu^{UM}, \sigma^{DM})$$

measures the bidirectional traffic transmitted between μ and σ during the time interval Δt.

Thus, a Stress Element can be represented as:

$$SE(\mu^{UM}, \sigma^{DM}, \text{stress}_{\sigma}^{\mu}).$$

In our implementation, `TraDE` analyzes metrics data retrieved from:

`istio_tcp_sent_bytes_total` and `istio_tcp_received_bytes_total`[1],

which respectively measure the total bytes sent during response and the total bytes received during request.

Before analyzing the traffic flow patterns among microservices, it is essential to build a stress graph that represents the dependencies between microservices and the associated stress on those dependencies. Additionally, sorting the microservice pairs from the constructed traffic graph is important for efficient service-node mapping in subsequent sections. Therefore, we propose algorithms for constructing a traffic stress graph and sorting the microservice pairs based on the constructed stress graph.

Building the Traffic Stress Graph

Algorithm 5.1 constructs the traffic stress graph, *Graph*, using the stress elements defined by Eq. 5.5. The algorithm begins by retrieving a list of deployed microservices at the current monitoring time. It then initializes the *Graph* with zeros for each row and

[1]Different versions of Istio service mesh may use different standard metrics.

Algorithm 5.1 Build Traffic Stress Graph

```
1: Input: List of Stress_Elements with SE objects
2: Output: A traffic Graph with dependencies and stress
3: MS ← Stress_Elements                          ▷ list of deployed microservices
4: Initialize Graph ← zeros(|MS| × |MS|)
5: for each μ in MS do                           ▷ upstream microservices
6:     for each σ in MS do                       ▷ downstream microservices
7:         if μ ≠ σ then
8:             stress ← BI_TRAFFIC(μ, σ, Δt)
9:             Graph[μ][σ] ← stress
10: return Graph
```

column. In the next step, it iterates through all deployed microservices and calculates the traffic stress between each pair of microservices over a given time interval. Once the matrix iteration is complete, the traffic stress graph is obtained.

Sorting Microservice Pairs by Stress Level

After constructing the traffic stress graph, the next step is to identify the stress level among the microservice pairs in the stress graph. Algorithm 5.2 is responsible for sorting all the microservice pairs with traffic values from the stress graph *Graph* in descending order to identify the microservice pairs with the higher stress and also the pairs with lower stress, which will be used for designing scheduling policies in the proposed `TraDE`. The microservice pairs with higher stress are the pairs that contribute more to the total communication cost in Eq. 5.2. A quicker localization of the microservice pairs under higher stress would be good for a quicker convergence to find the satisfied microservices (M) to nodes (N) mapping ($P : M \rightarrow N$).

5.6 Design of Dynamics Manager

5.6.1 Dynamic Delay Generator

Customized Delay Generation. Injecting different cross-node delays to the cluster nodes is crucial for evaluating the effectiveness of `TraDE`. However, implementing various

Algorithm 5.2 Sort Microservice Pairs by Traffic Stress

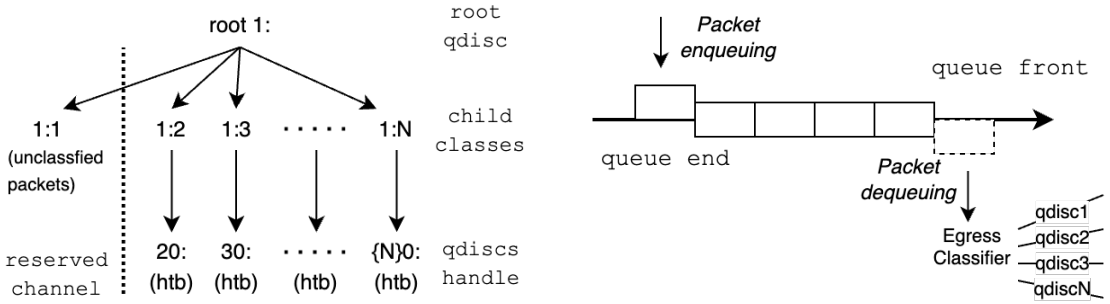
```
1: Input: Traffic Stress Graph Graph (traffic stress matrix)
2: Output: Sorted list of microservice pairs by traffic stress
3: Initialize pairs ← {}                                  ▷ empty list of pairs
4: for each μ in Graph do
5:     for each σ in Graph[μ] do                          ▷ corresponding DM
6:         if Graph[μ][σ] > 0 then
7:             Append (μ, σ, Graph[μ][σ]) to pairs
8: SortPairs(pairs)                                       ▷ sorting in descending order
9: return Sorted pairs
```

communication delays from one source node to multiple destination nodes poses significant challenges. To the best of our knowledge, no existing work has proposed a method to inject customized communication delays in a controllable manner. Some related works [14, 15, 18, 125, 126] mention delay injection to server nodes, but these works either use a static delay matrix or implement uniform communication delays for every egress traffic packet on the node's network interface. This approach leads to two main limitations in evaluation experiments: (1) Uniform communication delays from one source node to all destination nodes prevent distinguishing differences between node pairs, as delays from one source node to all other nodes share the same settings; (2) All other networking services not involving correlated nodes are degraded because the injected delays affect all egress traffic.

To address these limitations in current research, as shown in Fig. 5.7a, we propose a customized delay injection scheme that classifies packets using filters to differentiate egress packets and distribute them based on their IP destinations. Additionally, an extra channel is reserved for default packet transmission without injected delays, ensuring that the performance of other services is not affected. We implemented this scheme using the `tc` networking tool and `htb` (Hierarchical Token Bucket).

5.6.2 Cross-node Delay Measurer

In a large-scale computing cluster with high traffic, communication delays among infrastructure nodes are not negligible, as microservice applications are decoupled and distributed across different connected computing nodes. When significant delays occur



(a) Traffic packets classifying disciplines. **(b)** Packets egress with different class disciplines.

Figure 5.7: (a) Unclassified packets (not sent to certain destinations) are transmitted through a reserved channel without injected delays, while classified packets are assigned different delays for different destinations. (b) Classified packets are sent to different destinations with varying delays.

among cluster nodes, the communication among microservice containers running on these nodes will be negatively impacted. Consequently, some microservices may experience QoS violations.

Design of Delay Measurer. To address this problem, we designed a lightweight module called the *Cross-node Delay Measurer* for our proposed `TraDE` framework. The module consists of a centralized information processing unit and multiple distributed measuring agents. To keep the module lightweight, the processing unit maintains minimal connections with all measuring agents and summarizes the measured communication delays. The measuring agents run as lightweight containers, maintaining only a simple communication function for measuring delays. Additionally, to enhance the module's robustness during cluster-level upgrades or downgrades, an auto-scaling mechanism for the distributed measuring agents is implemented. Specifically, when the cluster adds more nodes, a measuring agent is automatically added to the new nodes to maintain the consistency and accuracy of communication delay measurements across the cluster. Similarly, when the cluster removes nodes, the corresponding measuring agents are automatically removed as well.

Implementation of Delay Measurer. We implemented an efficient, lightweight, and auto-scalable cluster-level measuring scheme that analyzes cross-communication delays across infrastructure nodes. As shown in Fig. 5.8, we introduced the design schemes of

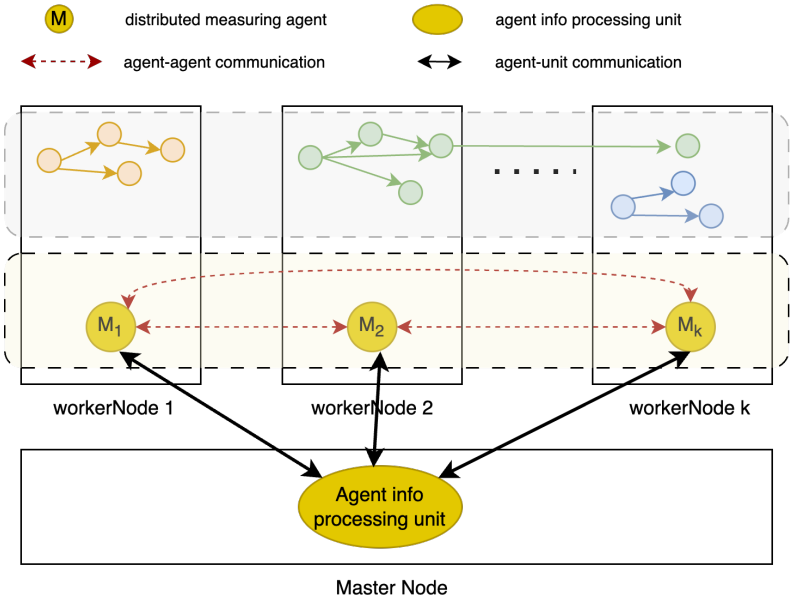


Figure 5.8: The implementation of *Cross-node Delay Measurer* across cluster nodes.

Table 5.1: Injected delay versus measured cross-node communication delay (ms). In each cell, the first number represents the injected delay, and the second number represents the measured actual delay. This table demonstrates the successful injection of different delays from one source node to various destination nodes, as well as the reserved channel for destinations (i.e., Master node and google.com) without injected delays.

Source Node	Destinations										
	Node1	**Node2**	**Node3**	**Node4**	**Node5**	**Node6**	**Node7**	**Node8**	**Node9**	**Master Node**	**google.com**
Node1	- / -	3.00 / 3.21	8.00 / 8.51	10.00 / 11.01	14.00 / 14.21	6.00 / 6.73	27.00 / 28.98	13.00 / 13.89	21.00 / 22.31	- / 0.64	- / 14.10
Node2	8.00 / 8.22	- / -	4.00 / 4.52	13.00 / 14.02	14.00 / 15.21	18.00 / 19.02	38.00 / 38.22	31.00 / 32.13	29.00 / 31.02	- / 0.89	- / 14.20
Node3	4.00 / 4.03	12.00 / 12.37	- / -	8.00 / 9.05	18.00 / 18.66	11.00 / 11.89	4.00 / 5.75	12.00 / 13.41	15.00 / 16.89	- / 0.43	- / 14.20
Node4	17.00 / 17.04	15.00 / 15.11	7.00 / 7.21	- / -	5.00 / 5.87	6.00 / 6.79	22.00 / 23.15	13.00 / 13.76	25.00 / 26.82	- / 0.51	- / 14.30
Node5	20.00 / 21.02	12.00 / 12.23	11.00 / 11.68	10.00 / 11.27	- / -	9.00 / 9.73	7.00 / 7.13	4.00 / 4.11	9.00 / 9.23	- / 0.45	- / 14.10
Node6	17.00 / 17.12	26.00 / 26.06	18.00 / 18.87	16.00 / 17.08	6.00 / 6.11	- / -	5.00 / 5.71	10.00 / 10.19	5.00 / 5.86	- / 0.53	- / 14.20
Node7	20.00 / 20.96	10.00 / 10.52	10.00 / 10.91	9.00 / 9.86	11.00 / 12.92	5.00 / 5.67	- / -	5.00 / 5.71	9.00 / 9.39	- / 0.45	- / 14.10
Node8	21.00 / 21.11	25.00 / 25.72	4.00 / 4.53	10.00 / 10.71	12.00 / 12.73	15.00 / 16.02	10.00 / 10.13	- / -	6.00 / 7.08	- / 0.34	- / 14.10
Node9	36.00 / 36.93	22.00 / 22.70	40.00 / 40.39	9.00 / 10.08	25.00 / 25.69	8.00 / 8.16	7.00 / 7.23	6.00 / 6.18	- / -	- / 0.30	- / 14.30

the *Cross-node Delay Measurer*, consisting of an agent information processing unit and a set of distributed measuring agents.

Specifically, we designed the agent information processing unit to operate as a running plugin on the master node, collecting measured cross-node delay information from all active agents via `TCP` messages. Meanwhile, the distributed measuring agents are implemented as a group of interconnected running pods, managed by a DaemonSet de-

ployment with a lightweight pre-configured image. In this way, the measuring agents on worker nodes continuously measure cross-node delays with each other and send the measured information to the information processing unit plugin on the master node. Additionally, when the number of cluster nodes changes, the agents are automatically added when a new node joins or deleted when a node drains from the existing cluster through the DaemonSet deployment mechanism. With this mechanism, the *Cross-node Delay Measurer* ensures that each node will have a dedicated pod for delay measurement, regardless of changes in the number of cluster nodes.

Fig. 5.8 shows how the distributed measuring agents communicate with each other and how the agent information processing unit interacts with each measuring agent container.

Overhead analysis of Delay Measurer. Regarding the overhead of delay measurement in a real system, we optimized the measurement using parallel processing. The latency measurement tasks are designed to run concurrently, and the results are aggregated into a latency results dictionary, completing the process in just a few seconds. Additionally, the distributed measuring agents are lightweight (about 0.2 MiB) and stable (running over 6 months without any failure). Each node runs a measuring agent developed from the `curlimages/curl` image, consuming around 0.2 MiB of memory per node. Thus, the cluster-level delay measurement consumes minimal memory and completes within a few seconds.

5.6.3 Effectiveness of Dynamics Manager

Injecting different communication delays from one source node to various destination nodes and accurately measuring these delays is challenging. By adopting the proposed injection scheme shown in Fig. 5.7, we injected different delays to various destination worker nodes and measured the actual communication delays from the source node to these destinations. As shown in Table 5.1, the measured delays in each cell exhibit high accuracy, with only around a 1 *ms* difference, which can be attributed to the randomness of data packet transfers in cloud networking stacks. Additionally, to demonstrate that other traffic is not influenced, we tested the communication delays from all worker

nodes to other destinations. One destination is the master node in the same cluster, and the other is the Google host (google.com), showing average delays of 0.50 *ms* and 14.20 *ms*, respectively.

From these measured data, we can confirm the effectiveness of the proposed schemes and implementation for the *Dynamics Manager* in `TraDE`. It should be noted that the delay injection scheme is optional and can be switched off when `TraDE` is deployed in actual computing environments. The primary design aim of the customized delay injection scheme is to generate various dynamics to validate and evaluate our proposed adaptive scheduling framework `TraDE`. Additionally, the proposed delay injection mechanism with customized delays can be easily adopted to evaluate systems in different computing environments, such as cloud-edge continuum and pervasive computing.

5.7 PGA Mapper for Microservice Placement

To solve the defined problem in Section 3.3, we proposed the following PGA (Parallel Greedy Algorithm) parallel algorithm to address the problem. We propose a Parallel Greedy Algorithm to optimize microservice placement by leveraging parallel computing to minimize communication costs. This method ensures efficient and effective placement of microservices, reducing inter-node communication overhead and improving system performance.

```
ParallelPlace(T, D, P, res, cap, workers)
|-- Sorted_MS_pairs(T)
|-- for each chunk task:
|   |-- PlaceWorker(T, D, P, res, cap, tasks)
|   |   |-- CalcCost(T, P, D, res, cap)
|   |   |-- CalcCost(T, new_P, D, res, cap)
|-- Choose best result from all workers
```

Figure 5.9: Function call hierarchy of the PGA placement Algorithm 5.3.

Algorithm 5.3 PGA Algorithm for Microservice Placement under Dynamic Traffics and Cross-node Delays.

Input: Matrix T for microservice traffic stress graph, Matrix D for cross-node delay graph, Placement P for service to node mapping list.
Output: Microservice-node *placement*, Lowest *cost*
Define $res\_list \leftarrow \{\texttt{cpu}, \texttt{memory}, \texttt{gpu}, \dots\}$
$res \leftarrow$ GET_MS_DEMANDS(res_list, T)
$cap \leftarrow$ GET_NODE_CAPACITIES(res_list)
`// Cost and Overloads Penalty.`
function CALCCOST(T, P, D, res, cap)
    $cost \leftarrow 0$
    for all (u, v) in T **do**
        $cost \leftarrow cost + T[u][v] \times D[P[u]][P[v]]$
    Initialize penalty factor pf ▷ pf can be adaptive
    $loads \leftarrow [0] \times \text{len}(cap)$
    for each u in P **do** ▷ Calculate server resource load
        $loads[P[u]] \leftarrow loads[P[u]] + res[u]$
    $penalty \leftarrow 0$
    for each server j in $loads$ **do** ▷ Check for overloads
        if $loads[j] > cap[j]$ **then**
            $penalty \leftarrow penalty + (loads[j] - cap[j]) \times pf$
    return $cost + penalty$
`// Microservice Placement Worker.`
function PLACEWORKER$(T, D, P, res, cap, tasks)$
    $current\_cost \leftarrow$ CALCCOST(T, P, D, res, cap)
    $nodes\_num \leftarrow |D|$ ▷ number of server nodes
    for each $(u, v, stress)$ in $tasks$ **do**
        $new\_P \leftarrow$ other nodes for u and v
        $\_cost \leftarrow$ CALCCOST(T, new_P, D, res, cap)
        if $\_cost < current\_cost$ **then**
            Update P and $current\_cost$
    return $P, current\_cost$
`// Parallel Placement Computing.`
function PARALLELPLACE$(T, D, P, res, cap, workers)$
    $pairs \leftarrow$ SORTED_MS_PAIRS(T) ▷ Algorithm 5.2
    $num\_pairs \leftarrow |pairs|$ ▷ Get the number of pairs
    $num\_workers \leftarrow |workers|$
    $size \leftarrow \lceil \frac{num\_pairs + num\_workers - 1}{num\_workers} \rceil$
    `/* Distribute tasks to workers */`
    Initialize $tasks \leftarrow []$ ▷ Initialize chunk tasks
    for $i = 0$ **to** $num\_workers - 1$ **do**
        $tasks \leftarrow pairs[i \times size : (i + 1) \times size]$
    $results \leftarrow$ PLACEWORKER$(T, D, P, res, cap, tasks)$
    Choose the best $\{P, cost\}$ from $results$
    return $best\_P, best\_cost$

5.7.1 PGA Algorithm Explanation

The algorithm aims to iteratively refine the placement of microservices to minimize the total communication cost, as defined in Equation 5.2. It leverages parallel processing to handle multiple microservices concurrently, improving the efficiency of the optimization process. The algorithm integrates three core functions: calculating communication costs, optimizing placements for chunks of microservices, and iteratively refining the overall placement until no further improvement is achievable. As shown in Figure 5.9, the *ParallelPlace* function coordinates the execution by calling *PlaceWorker* on distributed microservice pairs, which in turn invokes *CalcCost* to evaluate each candidate placement.

The key steps of the proposed method for microservice placement are as follows:

1) Initialization: The algorithm begins with an initial placement of microservices across server nodes, alongside the input of a traffic stress graph and a cross-node communication delay graph.

2) Resource List Definition: As outlined in Eq. 5.3, resource constraints such as CPU, memory, and GPU availability are critical when migrating microservices. A resource list `res_list` is defined at the start, specifying the resources to be considered throughout the placement process.

3) Cost Calculation: The total communication cost for the initial placement is computed using the *CalcCost (T, P, D, res, cap)* function. This function also accounts for server overloads by introducing a penalty factor, which adds to the communication cost if any server's capacity is exceeded.

4) Parallel Processing: The set of microservices is divided into chunks, and the *PlaceWorker* function is applied to each chunk in parallel via *ParallelPlace* function.

- In the *PlaceWorker* function, for each microservice pair (u, v), the algorithm attempts to reassign u and v to every possible pair of server nodes (excluding their current assignments) across the cluster. This is done exhaustively over all node pairs (i, j), ensuring that all candidate placements are evaluated. The new placement is only accepted if it results in a strictly lower total cost (including penalty for resource violations) compared to the current placement.

- In the *ParallelPlace* function, the full set of sorted microservice communication

pairs is first partitioned into disjoint chunks, where each chunk is assigned to a separate worker (e.g., CPU thread). Each worker executes its own instance of the *PlaceWorker* function independently. These workers operate in parallel, concurrently exploring local placement refinements for their assigned microservice pairs. After all workers (CPU threads) complete, their outputseach representing a locally optimized placement and costare gathered, and the best result (i.e., the one with the lowest overall cost) is selected as the final placement. This design improves scalability by enabling simultaneous local search across multiple parts of the system.

5) Iterative Refinement: The results from all parallel workers (i.e., CPU cores) are gathered, and the overall placement is updated if a better solution is found. This process continues until no further improvement can be made in the placement.

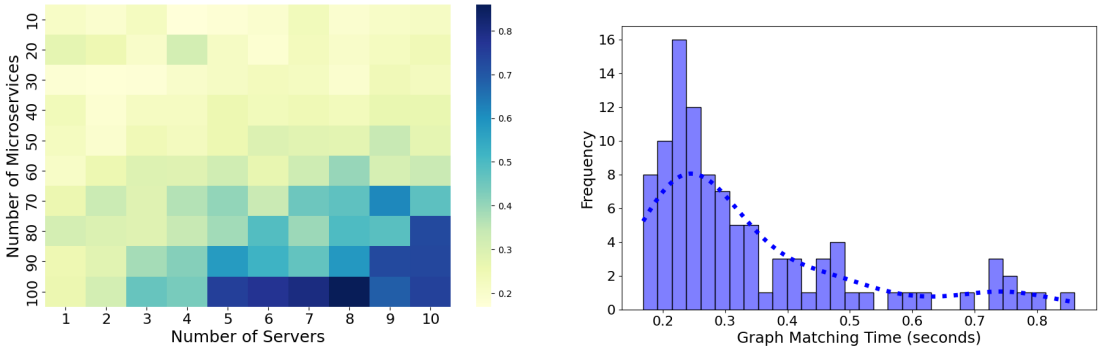
6) Output: The algorithm concludes by returning the satisfied microservice placement across the server nodes, along with the minimized communication cost.

This approach leverages parallel processing to efficiently handle large-scale microservice deployments, ensuring efficient placement with reduced communication overhead.

5.7.2 Balanced Chunks and Fast Convergence

Balanced Task Chunks: To balance the computation tasks at each parallel worker, we adopted the chunk size calculated by $size = \lceil \frac{num\_pairs+num\_workers-1}{num\_workers} \rceil$ to ensure a more accurate and even distribution of tasks among workers, especially when the number of tasks is not perfectly divisible by the number of workers. This approach correctly handles the remainder and avoids overestimating the chunk size, ensuring a more balanced task load distribution for the parallel computing process.

Fast Convergence: Not all microservices contribute equally to the total communication cost. Some pairs of microservices may have significantly higher traffic between them compared to others. As shown in line 40 of the Algorithm 5.3, high-traffic microservice pairs will be prioritized. By focusing on these high-traffic pairs first, the algorithm addresses the microservice placements that contribute the most to the total cost. Thus, improvements in these placements will have a disproportionately large impact on



(a) PGA Overhead Heatmap.

(b) PGA Overhead Distributions.

Figure 5.10: Processing time overheads for exploring satisfied placements via the PGA algorithm under different numbers of server nodes and microservices.

reducing the overall cost compared to optimizing low-traffic pairs.

5.7.3 Overhead Analysis

Exploring the satisfied placement result is time-consuming. To address this, we designed and implemented the proposed algorithm as a parallel algorithm and quantified the overheads in our cluster. When high-impact microservice pairs are grouped and processed in parallel, it reduces the overhead of synchronization between parallel tasks, thus reducing the need for frequent inter-worker communications. As shown in Fig. 5.10a, the running time of the proposed PGA algorithm is able to complete the matching process in less than 1 second. In Fig. 5.10b, it can be observed that the majority of the graph matching time is approximately 0.3 seconds, which indicates a promising scheduling decision time with a quick response to dynamic changes.

5.7.4 Adaptive Scheduler

Adaptive Scheduler is responsible for rescheduling the deployed microservices containers based on the satisfied placement results from PGA mapper. When there are traffic stresses on the running application or notable communication delays among certain server nodes, the pre-defined QoS targets might be violated in practical computing environments. Thus, designing adaptive scheduling schemes to tackle QoS violations is

significant.

The rescheduling process involves microservice instance migration and eviction across different server nodes in the cluster. However, this will lead to the following problems: (1) how to guarantee the performance of microservice instances that are not affected during the rescheduling process?, (2) when service consistency is guaranteed, how to avoid the overloads on certain server loads?

Asynchronous launching: At the beginning of microservice migration, it is crucial to ensure the service availability of the affected microservices. We initially launch new microservice containers on the target cluster node, and once these new containers are in ready states, the old containers are evicted from the previous nodes. This approach guarantees zero downtime for specific microservices, whose backend-supported microservice instances require migration.

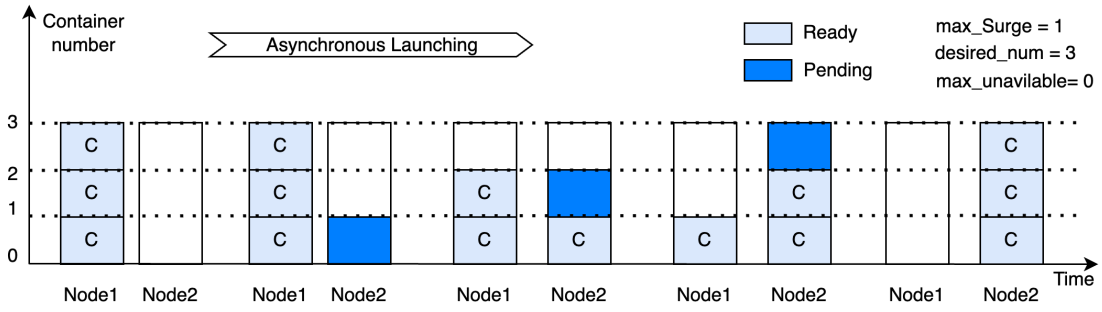


Figure 5.11: The process of asynchronous launching for container migration (evicting old while launching new containers) between two nodes to guarantee service high availability.

Constraints-based scheduling: The proposed rescheduling scheme computes a servicenode placement based on microservice dependencies and the cross-node delay matrix, and also enforces server-node resource constraints (e.g., CPU, memory, GPU), as mathematically summarized in Eq. 5.3. In line 2 of the Algorithm 5.3, a resource constraints list is defined for consideration during the whole rescheduling process. Additionally, the microservice demands (e.g., `requests` and `limits` in deployment yaml file) and server node resource capacities are considered as conditions at Algorithm 5.3.

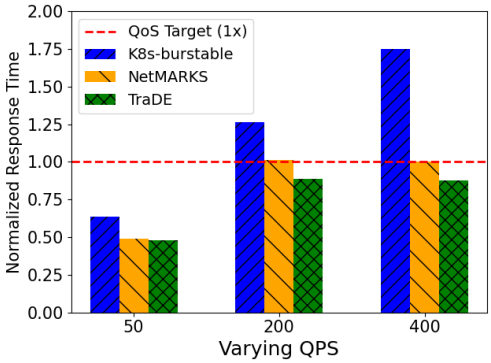
5.8 Performance Evaluation

5.8.1 Testbed Setup

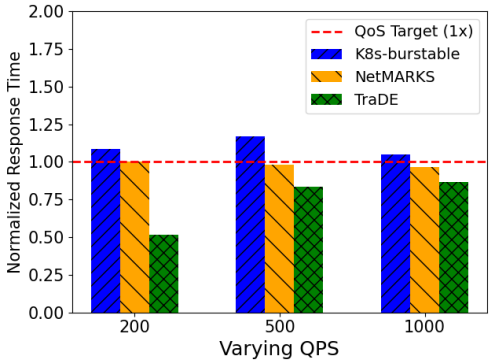
Cluster Setup: We validated our design by implementing `TraDE` using the de facto standard container orchestration platform, Kubernetes [12]. We deployed `TraDE` on 10 server nodes without any preset anti-colocation rules, such as taints for server nodes and affinity for pods. The Kubernetes cluster configuration includes one master node and nine worker nodes. The master node features 32 CPU cores with x86_64 AMD EPYC 7763 series processors, 32GiB of RAM, and a network bandwidth of 16 Gbps. Each of the nine worker nodes is equipped with 4 CPU cores from the same AMD series as the master node, 32 GiB of RAM, and a network bandwidth of 16Gbps. Regarding software versions, the cluster runs Kubernetes v1.27.4, the Container Network Interface (CNI) plugin Calico v3.26.1, the service mesh Istio v1.20.3, and uses CRI-O v1.27.1 as the container runtime. All server nodes operate on Ubuntu 22.04.2 LTS with the Linux kernel 5.15.0.

Besides, each of the cluster nodes is running on a Virtual Machine instance at the dedicated research cloud platform from the University, thus the typical communication delay among each of the nodes is ultra-low, usually ranging from 0.2 to 1 milliseconds (*ms*), tested by sending `ICMP` messages between nodes. To make our evaluations more realistic and emulate the changing cluster networking environment, the cross-node delay matrix can be automatically updated every t (i.e., $t = 5$) minutes. In Table 5.1, we demonstrated the customized cross-node communication delays to the worker node destinations and also avoided injecting delays to other destinations like the master node and external sites.

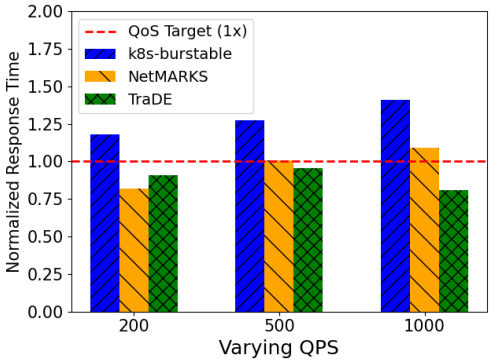
Benchmark Application: We adopted `Social Network` from DeathStarBench[2] as the microservice application to evaluate our proposed scheduling framework. `Social Network` benchmark emulates a simplified social media platform similar to popular social networking services. It is structured to replicate the intricate interactions and communication patterns in such type of applications. The benchmark comprises 27 different microservices that collectively offer functionalities such as composing a post, writing a user timeline, and writing a home timeline.



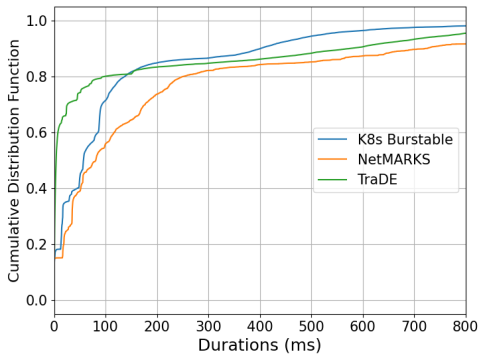
(a) `compose-post`.



(b) `read-user-timeline`.



(c) `read-home-timeline`.

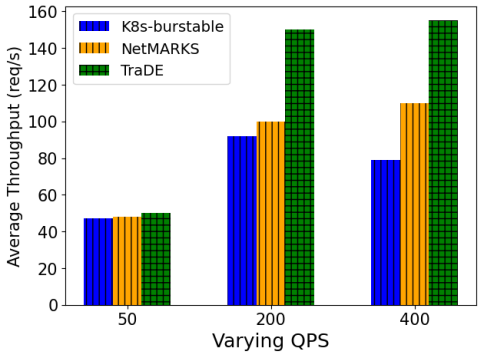


(d) Durations for execution time.

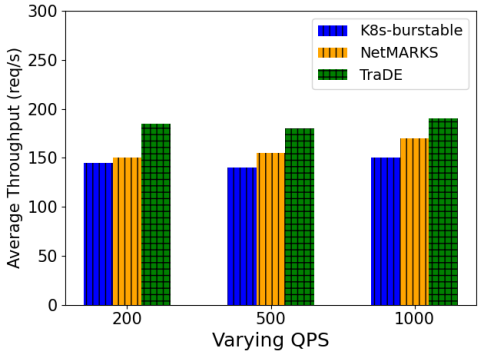
Figure 5.12: In (a)(b)(c), different response time comparisons under varying QPS and request types. In (d), the cumulative distribution function (CDF) figures of all triggered microservices execution time by mixed workload requests (with 6:2:2 ratio of `compose-post`, `read-user-timeline`, and `read-home-timeline` requests), showing TraDE has an overall reduced execution time, thereby leading to better end-to-end performance.

5.8.2 Workload Generator

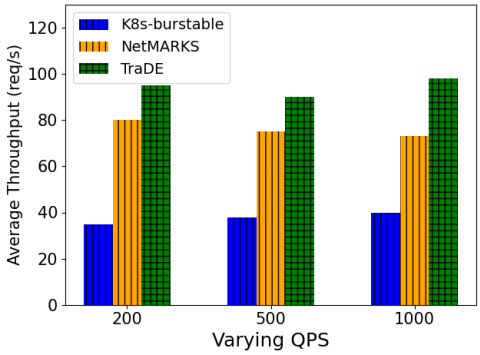
We adopted wrk2 [113] as the workload generator. As a modern HTTP benchmarking tool, wrk2 is capable of generating different types and proportions of workload requests for performance testing and measuring how well the cloud applications can handle varied traffic. Its ability to maintain a constant request rate makes it particularly useful for understanding the end-to-end performance of a server under controlled workload conditions.



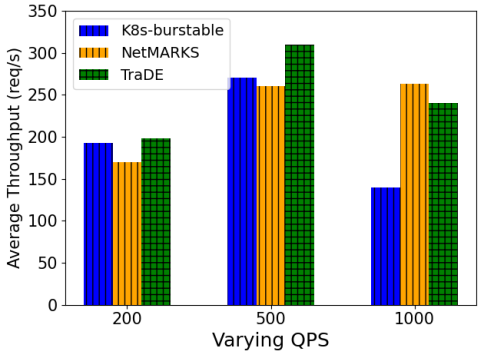
(a) `compose-post`.



(b) `read-user-timeline`.



(c) `read-home-timeline`.



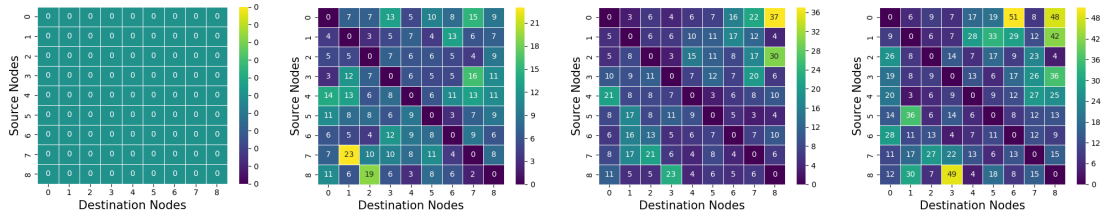
(d) Mixed requests (6:2:2).

Figure 5.13: In (a)(b)(c), the evaluation comparisons of average throughput are shown under varying QPS and different request call-graphs. In (d), the throughput under mixed workload requests is displayed with the proportion of 6:2:2 for `compose-post`, `read-user-timeline`, and `read-home-timeline`, respectively.

5.8.3 QoS Targets and Compared Methods

QoS Target Determination (Trigger-Oriented)

In our experiments, we use a fixed latency QoS target as the control-plane trigger for `TraDE`: every $\tau = 30\,\text{s}$ the scheduler queries Prometheus over a sliding window W (default $W = 1\,\text{min}$) for Istio metrics in the target namespace with `response_code=200`, `istio_request_duration_milliseconds_sum` and `istio_request_duration_milliseconds_count` (via `rate` and `custom_query_range` functions)and computes the windowed average response time $\bar{L} = \frac{\sum \texttt{sum\_values}}{\sum \texttt{count\_values}}$ (ms); if $\bar{L} > T$ with $T = 300\,\text{ms}$ (we also test $T \in \{250, 300, 350\}$ ms in Fig. 5.15), `TraDE` sets *Trigger*=`True` and exe-



(a) delay_matrix1_zero. **(b)** delay_matrix2_light. **(c)** delay_matrix3_heavy. **(d)** delay_matrix4_-medium.

Figure 5.14: In (a)(b)(c)(d), dynamic delays are injected to the nine worker nodes.

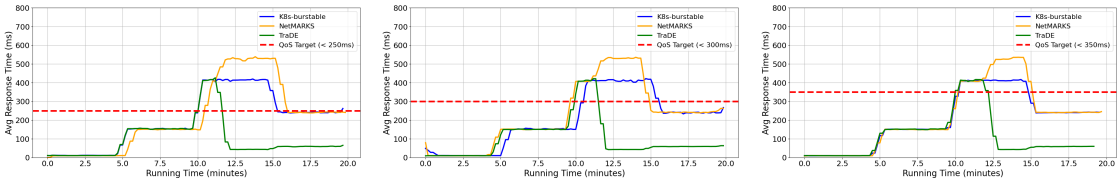
cutes the rescheduling pipeline implemented in `TraDE` (traffic-graph construction, current placement extraction, and PGA-based remapping/migration); if the window has no data or the traffic count is zero, no trigger is raised.

Compared Methods

To evaluate our proposed `TraDE` framework, we compared it with the default Kubernetes scheduling policy and the recent traffic-aware NetMARKS [32] which also targets network-aware scheduling for microservice applications.

K8s default policy: In Kubernetes, the default scheduling and Quality of Service (QoS) policies are designed to evenly distribute workloads across the cluster without further rescheduling policies even when microservice performance is violated. QoS policies in the k8s cluster classify pods into three classes: *Guaranteed*, *Burstable*, and *BestEffort*. *Guaranteed* provides the highest priority ensuring pods always get the requested resources, *Burstable* offers a flexible middle ground where pods have guaranteed minimum resources but can consume more if available, while *BestEffort* has the lowest priority, where pods have no resource guarantees and can be preempted first during resource contention. In the evaluation experiments, we will use the default scheduling policy with *Burstable* QoS class.

NetMARKS: The recent work NetMARKS [32] introduces a microservice pod scheduling scheme that leverages dynamic network metrics collected from the Istio Service Mesh. The main idea of NetMARKS is the proposed node scoring algorithm, which calculates node scores for a target pod by iteratively analyzing all pods running on each



(a) QoS Target $<$ 250 ms **(b)** QoS Target $<$ 300 ms **(c)** QoS Target $<$ 350 ms

Figure 5.15: Under three different QoS targets and varying cross-node delays, the adaptive response of k8s-burstable, NetMARKS, and TraDE. In 20 minutes of the test time, $0 \sim 5$ mins, there are no injected cross-node delays; 5$\sim$10 mins, light cross-node delays are injected; 10$\sim$15 mins, heavy cross-node delays are injected; 15$\sim$20 mins, medium cross-node delays are injected.

node and identifying those with traffic connections to the target pod. Each node's score is calculated based on the sum of traffic flows between the target pod and the selected pods on that node. The node with the highest score is then chosen to host the target pod.

With the implemented modules of *Traffic Analyzer* in Section V, *Dynamics Manager* in Section VI and *PGA Mapper* in Section VII, we evaluated the proposed `TraDE` with benchmark microservice application and compared the end-to-end performance with default k8s QoS policy [12] and NetMARKS [32]. In the evaluation experiments, we implemented the K8s QoS policy with *Burstable* and the pod scheduling policy without any pre-set rules like affinity and taints. For NetMARKS [32], we implemented the node scoring algorithm proposed by NetMARKS and adopted it for rescheduling the predefined target microservice pods experiencing QoS violations.

To ensure fair evaluations, three separate namespaces are created for three identical `social network` applications from [2], each managed by a different method (i.e., K8s default, NetMARKS, and TraDE). Additionally, sustained workloads are sent concurrently to each of the three identical `social network` applications. This setup isolates the workloads for each method while maintaining the same cross-node delay settings for the three applications at the cluster level.

We generated multiple workloads with different requests and varying QPS to evaluate the end-to-end performance of our proposed `TraDE` and existing methods.

Response Time and Durations. For the `social network` benchmark application, we used the wrk2 tool to generate three request types: `compose-post`, `read-user-`

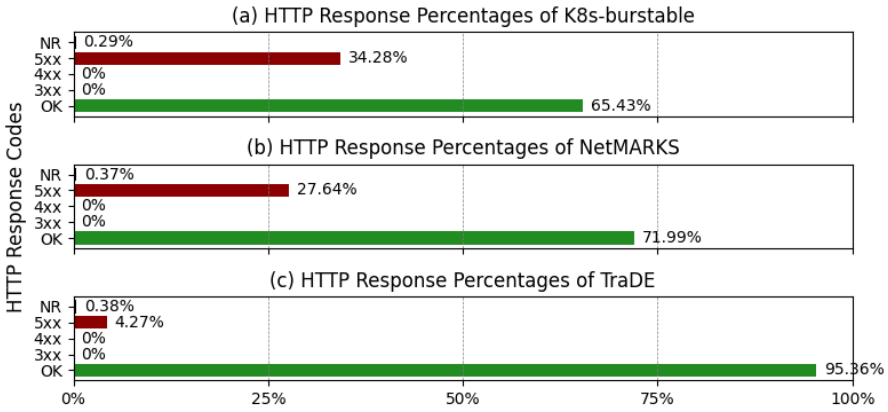


Figure 5.16: In each service mesh, HTTP response percentage distributions show the overall microservice application performance under different deployment methods. The higher the percentages of 'OK' responses, the higher the goodput ratio.

`timeline`, and `read-home-timeline`. Each type shows distinct call graphs and traffic patterns across the applications dependency graph. Fig. 5.12 compares the average response times under various workloads, including changes in QPS and user requests. For these request types, `TraDE` consistently outperforms existing methods, meeting QoS targets across scenarios. Compared to NetMARKS [32], `TraDE` achieves up to 12.3% lower response times for `compose-post`, 48.3% for `read-user-timeline`, and 25.8% for `read-home-timeline` requests. Notably, `TraDE` adapts effectively under varied workloads, where K8s-burstable and NetMARKS do not always meet QoS.

Fig. 5.12d illustrates the CDF of execution times for all triggered microservices under a mixed workload (6:2:2 ratio of `compose-post`, `read-user-timeline`, and `read-home-timeline` requests). Here, `TraDE` demonstrates reduced microservice execution time, leading to lower response times and higher throughput.

5.8.4 End-to-end Performance

Throughput and Goodput. Throughput and goodput are key metrics for assessing end-to-end performance in dynamic environments. Throughput represents total data transmitted, including overhead, while goodput measures only the useful data successfully received at the application layer.

In terms of *throughput*, we conducted experiments to measure the average throughput (requests/second) across different QPS and mixed workloads. As shown in Fig. 5.13, `TraDE` achieves higher throughput than NetMARKS [32]up to 1.5x for `compose-post`, 1.2x for `read-user-timeline`, 1.4x for `read-home-timeline`, and 1.2x for mixed requests, indicating superior throughput in various scenarios.

For *goodput*, we analyzed response types using Istio Service Mesh across separate deployments (K8s-burstable, NetMARKS, and TraDE) for isolation. Fig. 5.16 shows `TraDE` achieves a 95.36% success rate, outperforming NetMARKS (71.99%) and K8s-burstable (65.43%). These results show that `TraDE` surpasses existing methods in both throughput and goodput across dynamic workloads.

5.8.5 Adaptive Performance Under Changing Delays

To assess the adaptive capability of the proposed `TraDE` framework, we evaluated its performance under fluctuating cross-node communication delays. Specifically, in Fig. 5.14, four different cross-node delays were injected to the cluster nodes every five minutes. As shown in Fig. 5.15, `TraDE` effectively responds to these changing delays by adaptively redeploying microservice instances to meet QoS targets once detecting QoS violations. Comparing with the existing two methods, it is clear to observe that TraDE can consistently maintain response times within the QoS targets (i.e., $< 250ms, 300ms, < 350ms$) throughout the remaining runtime, while the other two methods (K8s-burstable and NetMARKS) failed to meet these QoS targets.

5.9 Summary

In this chapter, we designed a traffic and network-aware framework, `TraDE`, to address the challenges of QoS violations in containerized microservices running in dynamic computing environments. Our framework primarily consists of three components: a traffic stress analyzer, a network dynamics manager, and an efficient service node mapper. We evaluated our proposed `TraDE` against existing solutions and demonstrated that our framework effectively meets the QoS targets under various dynamic condi-

tions, outperforming the existing method NetMARKS by reducing response time by up to 48.3%, improving throughput by up to 1.4x and showing robust adaptiveness under sustained workloads.

Overall, TraDE demonstrates how telemetry about traffic stress and network variability can be translated into concrete, Kubernetes-compatible rescheduling decisions that improve end-to-end QoS under dynamic conditions. However, rescheduling alone cannot deal with sustained workload growth or long-run shifts in request mix. The next chapter would extend the control scope from placement-only adaptation to *joint scaling and placement*.

Chapter 6

Adaptive Scaling and Placement for Microservices under Multi-source Dynamics

Microservice applications are increasingly deployed across cloud–edge environments, where heterogeneous nodes and time-varying inter-node delays amplify the impact of placement decisions. At the same time, these applications face non-stationary traffic, shifts in the mix of root request operations that exercise different call graphs, and heterogeneous communication modes that determine how network latency and queuing propagate to end-to-end (E2E) performance. Existing autoscalers and network-aware schedulers typically handle only a subset of these dynamics, leading to either compute bottlenecks or inflated cross-node latency and thus SLO violations. We present AdaScale, an adaptive framework that jointly scales and places microservice replicas under such multidimensional dynamics. AdaScale implements a Monitor–Analyze–Plan–Execute (MAPE) loop that extracts per-edge and per-service demand from distributed traces and service-mesh metrics, identifies the most critical root operation under a mixed workload, computes SLO-aware replica targets, and then places replicas to minimize a demand-weighted latency objective given the current inter-node latency matrix. To react quickly to networking perturbations, AdaScale triggers a reactive placement loop, while a steady-state autoscaling loop handles demand shifts.

This chapter is derived from:

- **Ming Chen**, Muhammed Tawfiqul Islam, Maria A. Rodriguez, and Rajkumar Buyya, "AdaScale: An Adaptive Scaling and Placement Framework for Microservices Under Dynamics", *IEEE International Conference on Distributed Computing Systems (ICDCS)* [Under Review, Dec 2025].

6.1 Introduction

Microservice architecture has become a de facto approach for building large-scale online services by decomposing an application into loosely coupled services that can be developed and deployed independently [7, 29, 123]. Modern deployments increasingly span a cloud–edge continuum, where services are placed across heterogeneous nodes to satisfy latency, cost, and locality requirements. In such environments, end-to-end (E2E) service quality is shaped not only by per-service compute provisioning, but also by where replicas are placed: time-varying inter-node delays can quickly turn previously acceptable placements into latency bottlenecks [32, 67, 124].

Managing microservices under dynamics is challenging because multiple sources of variability interact. First, user demand is non-stationary and can exhibit bursts, requiring timely replica provisioning. Second, production microservice applications rarely serve a single request type. Instead, they serve a mix of root request operations (e.g., read vs. write paths), each exercising a different call graph and often having a different E2E SLO target; changes in the workload mix therefore change which services and edges dominate E2E latency. Third, microservice dependencies involve heterogeneous communication modes (e.g., blocking RPC, message queues, and storage I/O), meaning that the same network perturbation can manifest differently across edges and services. Finally, these effects are coupled: a bad placement magnifies cross-node delay even with sufficient replicas, while poor replica provisioning creates compute bottlenecks even under favorable network conditions.

Existing orchestration mechanisms typically handle only a subset of these factors. Kubernetes Horizontal Pod Autoscaler (HPA) [21] scales replica counts based on resource utilization signals, but is blind to the network and to which root operations are currently critical. Network-aware schedulers [32, 124] can improve placement using service-mesh telemetry, but commonly assume fixed replica budgets and do not explicitly reason about changing root-request mixes. Conversely, call-graph and SLO-analysis techniques can localize bottlenecks, but are often not designed to drive joint scaling and placement decisions under time-varying network states. These limitations motivate a unified framework that continuously translates runtime call-graph behavior and net-

working state into coordinated scaling and placement actions.

This chapter presents AdaScale, an adaptive scaling and placement framework for distributed microservice applications under dynamics. AdaScale follows a Monitor–Analyze–Plan–Execute (MAPE) design that (i) continuously collects distributed traces, service metrics, and inter-node latency measurements, (ii) infers per-edge and per-service demand summaries from the observed call-graph behavior, (iii) evaluates SLO risk under the current workload mix and derives SLO-aware replica targets, and (iv) places replicas to reduce demand-weighted cross-node latency under current network conditions. A key design principle is two-timescale adaptation: AdaScale uses a steady-state autoscaling loop to handle demand and workload-mix shifts, and a reactive placement loop to promptly mitigate networking perturbations that threaten E2E SLO compliance.

We implement AdaScale on Kubernetes with a service-mesh and tracing stack, and evaluate it on a cloud–edge cluster using the DeathStarBench Social Network benchmark [2], which exposes three root operations with distinct call graphs. Across a range of request rates and workload mixes, AdaScale consistently meets SLO targets and improves both latency and throughput compared with Kubernetes HPA and NetMARKS. This chapter mainly makes the following contributions:

- We formulate joint scaling and placement as an optimization problem that balances per-service resource cost against demand-weighted network latency under E2E and per-service quantile SLO constraints, and we decompose it into tractable scaling and placement subproblems suitable for online control.

- We design and implement AdaScale as a two-timescale MAPE controller with (i) an SLO- and demand-aware autoscaler that identifies the most critical root operation under mixed workloads and computes per-service replica targets, and (ii) a greedy, latency- and capacity-aware placer that maps replicas to nodes using the current inter-node latency matrix.

- We demonstrate the effectiveness of AdaScale on a real Kubernetes-based cloud–edge testbed, where it reduces average response time by up to 1.93$\times$ and increases throughput by up to 2.16$\times$ compared with NetMARKS, while satisfying SLO targets across dynamic conditions.

The remainder of this chapter is organized as follows: Section 6.2 outlines the background and key challenges. Section 6.3 introduces the system model and telemetry-based demand estimation. Section 6.4 formulates the joint scaling and placement problem, while Section 6.5 presents the design and algorithms of AdaScale. Section 6.6 provides the evaluation results. Section 6.7 reviews related work, and Section 6.8 concludes the chapter with summary.

6.2 Background and Challenges

6.2.1 Background

An increasing number of modern cloud applications have evolved into microservice-based architectures, which manage applications through a collection of containerized, loosely coupled, fine-grained services [2, 123]. In dynamic computing environments, these services can run in central cloud data centers, on fog/edge nodes (near end-devices), or across both. In practice, the deployed microservice application is usually deployed across the computing hosts. For example, in a cloud-edge continuum environment, the application would be decoupled into multi-layer business logic, so that some microservices run with centralized tasks in the cloud, while other microservices, which are responsible for user-facing services with strict latency requirements, run in the edge nodes. Although such a hybrid deployment improves application performance by lowering user-perceived latency, the dynamic and evolving networking environments may degrade the end-to-end performance even if the application is optimally deployed at the beginning. Moreover, in addition to dynamic networking conditions, workload dynamics are crucial factors that affect microservice performance by triggering different proportions of call graphs and imbalanced traffic.

6.2.2 Challenges

Networking Uncertainty. Varied networking conditions are a common feature of a dynamic computing environment, such as the cloud-edge clusters and interconnected

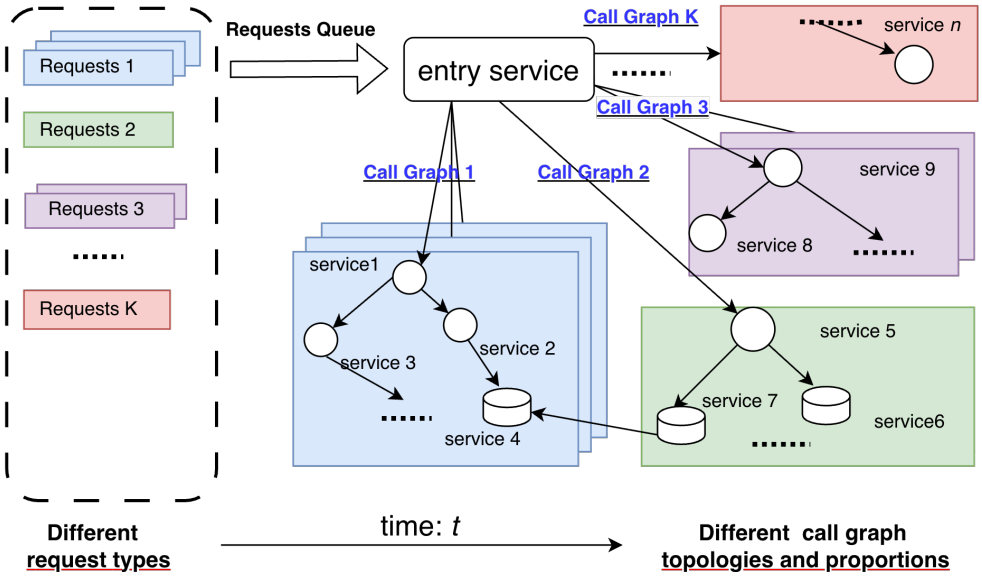


Figure 6.1: Different root requests trigger a timevarying mixture over microservice call graphs $\mathcal{G} = \{G_1, \ldots, G_K\}$ with proportions $\boldsymbol{\pi}(t) = (\pi_1(t), \ldots, \pi_K(t))$ and $\sum_{k=1}^{K} \pi_k(t) = 1$.

fog devices. In fact, microservice applications allocate a higher proportion of processing time to the networking stack than monolithic applications do [2], indicating more time is required for microservice applications to send and process requests over RPC or other REST APIs.

For the application, deployed as a set of containerized microservice instances, user requests are processed through dependent microservices, and the results are returned. In dynamic networking conditions, it is notable that latencies and available bandwidths vary between pairs of nodes, thereby microservices spend a significant amount of time sending and processing requests. Therefore, dynamic networking conditions, characterized by varying node-to-node latencies and available bandwidths, can significantly degrade the performance of these microservices.

Dynamics of Call Graphs. In the microservice dependency call graph, if one service triggers another, we refer to them as *UIM* (upstream microservice) and *DM* (downstream microservice), respectively. Each microservice can have one or multiple instances under different workloads to guarantee SLOs. After deploying the microservice instances

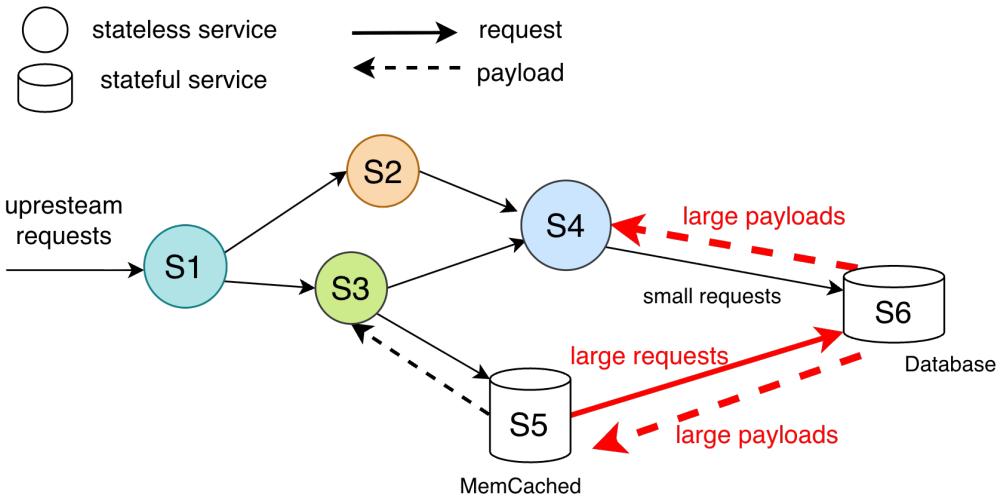


Figure 6.2: Inside one part of a call graph, different communication modes exist along the call graph edge, exhibiting different traffic load patterns.

across the cluster nodes, the entry microservice receives incoming requests to the application, which are then routed to a series of downstream microservices to trigger further complex calls. Thus, a better understanding and in-depth modeling of call graphs with quantified details are also crucial for designing efficient microservice scaling policies.

In Figure 6.1, at time t, under different types of incoming root requests (we refer to the requests served by the entry service of the application as *root requests*), there are different mixes of the triggered call graphs depending on the business logic of the application. As an illustration of microservice call-graph dynamics, the mix of various call-graphs makes the management of the microservice application even more difficult.

The mix of various call-graphs can be explained from two perspectives: *Graph topologies* and *Graph proportions*. For *Graph topologies*, it can be observed that there are different structures of graphs, such as *Call Graph 1*, *Call Graph 2*,..., and *Call Graph K*, triggered by the corresponding type of requests. For *Graph proportions*, under different volumes of request loads for each type, the higher the load of specific types of request, the higher the replica numbers of the corresponding graphs. For example, in Figure 6.1, a large number of users might concurrently send different proportions of different requests and trigger different proportions of call-graphs (*Call Graph 1*: *Call Graph 2*: *Call Graph 3* = 3: 1: 2). For a better understanding of the different call-graphs here, we simply demonstrate that

higher requests trigger more replicas of call graphs. Moreover, for shared microservices, like stateful *service 4* in Figure 6.1, among different call graphs, such sharing mechanisms would make the modelling and understanding of the call graphs at time t even more difficult. In the large-scale deployment of microservice applications, fine-grained management for shared microservices and scaling strategies for microservice containers are typically employed, which makes the overall call graph topology more complex and subject to change over time.

Communication Modes. As revealed in the microservice traces from Alibaba and Meta's production environments [4, 5, 69, 123], the communication modes (i.e., HTTP, gRPC, DB (Database), MQ (Message Queue), and MC (Memcached)) among dependent *upstream* and *downstream* microservice pairs are important factors that affect end-to-end performance, such as latency and throughput. For dependent microservice pairs with DB or MC communication mode, these pairs are usually heavy in traffic load and thus more prone to networking and load variations, which easily violates the end-to-end performance of upstream services. These communication modes would typically lead to imbalanced traffic loads across microservice call graphs. These imbalances indicate that specific service pairs could *become bottlenecks*, negatively affecting overall application performance under increased workloads.

Moreover, traffic flow is usually in one direction, from *Upstream* to *Downstream* services. When considering the traffic from *Downstream* to *Upstream* services, the transmitted bi-directional (sent and received) traffic can exhibit significant differences, such as small requests but large payloads, which implies the traffic load distribution would be another landscape. Thus, considering the communication modes that influence bidirectional traffic loads between each dependent microservice pair would be a better perspective to design the microservice placement and scaling strategies.

Thus, considering the challenges from different perspectives, accurately estimating the networking conditions and modeling the characterization the call graphs require a systematic framework that considers: (1) dynamic network conditions, including variable latency and bandwidth, could heavily impact the deployed microservices spanning across different nodes; (2) Under different types and volumes of request loads, the call

graph structures can be dynamic both in topology and proportions. (3) For a single structure of call graph, the heterogeneity of the communication modes should be characterized and modeled, as the traffic load flows along the graph could be heavily impacted, and a holistic view of bi-directional traffic flows needs to be considered. When designing efficient microservice placement and scaling strategies, the communication mode is a non-negligible factor.

6.3 System Model

We consider a *microservice application* composed of stateless and stateful microservices $\mathcal{S} = \{s_1, \ldots, s_N\}$ deployed on a cluster of compute nodes $\mathcal{N} = \{n_1, \ldots, n_M\}$. Each microservice $s \in \mathcal{S}$ has $x_s(t) \in \mathbb{N}$ replicas (e.g., Kubernetes Pods) at time t. A single replica of s consumes a per-replica resource vector $R_s = [c_s, m_s]$ (CPU shares and memory), while node n offers capacity $C_n = [\bar{c}_n, \bar{m}_n]$. For network conditions of the cluster, we primarily consider the inter-node communication delays and assume these delays evolve randomly as time goes by.

6.3.1 Replica placement matrix.

In AdaScale implementation, a service may spread its replicas across multiple nodes. We therefore represent a placement at time t by a matrix

$$A(t) = \left[A_{s,n}(t)\right]_{s\in\mathcal{S},\, n\in\mathcal{N}},$$

where $A_{s,n}(t) \in \mathbb{N}$ is the number of replicas of service s hosted on node n. The total replica count of s is $x_s(t) = \sum_{n\in\mathcal{N}} A_{s,n}(t)$. In the implementation, the JSON fields `assignments`:{`node` $\mapsto$ `replica count`} and `replicas` in the placement files encode $A_{s,n}(t)$ and $x_s(t)$, respectively.

6.3.2 Workflows as mixtures of call graphs

In a deployed microservice application, multiple root request types can coexist and thus trigger different call graphs. We refer to a root request as a request entering the application at its entry service (e.g., a front-end gateway). Incoming root requests arrive at a total rate $\lambda_{\text{root}}(t)$ and are partitioned into K request classes (e.g., compose-post, read-home-timeline, read-user-timeline) with per-class rates $\lambda_k(t)$ and proportions

$$\pi_k(t) \;=\; \frac{\lambda_k(t)}{\max\{\lambda_{\text{root}}(t), 1\}}, \quad \sum_{k=1}^{K} \pi_k(t) = 1.$$

Each root-request class k triggers its own call graph $G_k = (V_k, E_k)$. Thus, incoming root requests $\lambda_{\text{root}}(t)$ trigger a time-varying mixture over the microservice call graphs $\mathcal{G} = \{G_1, \ldots, G_K\}$ with proportions $\boldsymbol{\pi}(t) = (\pi_1(t), \ldots, \pi_K(t))$. Each call graph $G_k = (V_k, E_k)$ has $V_k \subseteq \mathcal{S}$ and directed edges $e = (u \to v) \in E_k$, where u and v denote upstream and downstream microservices. Over a time window W_t, we compute the following edge-level statistics from tracing spans:

Each edge e is annotated with an attribute

$$\text{attr}(e) = \langle \text{mode}_e,\ r_e(t),\ \theta_e,\ w_e(t),\ b_e(t) \rangle, \tag{6.1}$$

where:

- $\text{mode}_e \in \{\text{HTTP}, \text{gRPC}, \text{MQ}, \text{DB}, \text{MC}\}$ captures the communication mode; different modes have distinct traffic tramit patterns.
- $r_e(t)$ is the estimated call rate (calls/second) on edge e;
- θ_e identifies the callee interface (e.g., HTTP path, gRPC method, DB collection/op);
- $w_e(t)$ is the average edge-local service time (milliseconds), which can be obtained from child span durations in raw metrics traces;
- $b_e(t)$ is the byte rate (bytes/s) on e, which obtained from service-mesh traffic counters.

6.3.3 Edge statistics from traces

From each metrics span (e.g., Jaeger traces), we estimate, over a time window W_t, the following edge statistics:

$$\begin{aligned} p_e(t) &\triangleq \frac{\#\text{root requests that traverse } e}{\max\{\#\text{root requests}, 1\}}, \\ r_e^{\text{per-req}}(t) &\triangleq \frac{\#\text{occurrences of } e}{\max\{\#\text{requests that traverse } e, 1\}}, \\ w_e(t) &\triangleq \text{mean child-span duration for edge } e. \end{aligned} \tag{6.2}$$

A root request is a request entering the application at its entry service. The quantity $p_e(t)$ is the probability that a root request exercises edge e, $r_e^{\text{per-req}}(t)$ captures repeated invocations *conditional* on traversing e, and $p_e(t) \cdot r_e^{\text{per-req}}(t)$ is the expected number of occurrences of e per root request. For the trace sampling strategies, we use probabilistic sampling, which means sampling traces based on a defined probability (e.g., 0.1 for 10% sampling). If the root request rate is $\lambda_{\text{root}}(t)$ and the tracing sampling probability is ρ_{sample}, then the call rate on edge e is estimated as

$$r_e(t) \approx \lambda_{\text{root}}(t) \cdot \frac{p_e(t)\, r_e^{\text{per-req}}(t)}{\rho_{\text{sample}}}.$$

Here $\lambda_{\text{root}}(t)$ denotes the true root request rate (requests/s) for the application. We assume sampling is approximately uniform over requests, so rescaling by $1/\rho_{\text{sample}}$ debiases the per-edge counts.

6.3.4 Service-level demand

Under sustained workloads on the deployed microservice application, we can obtain per-service demand metrics by aggregating call graph edge statistics. For each service

$s \in \mathcal{S}$ we define:

$$R_s^{\text{in}}(t) = \sum_{e=(u,s)} r_e(t), \qquad R_s^{\text{out}}(t) = \sum_{e=(s,v)} r_e(t),$$
$$W_s^{\text{in}}(t) = \frac{\sum_{e=(u,s)} r_e(t)\, w_e(t)}{\max\{R_s^{\text{in}}(t), 1\}}, \qquad B_s^{\text{in}}(t) = \sum_{e=(u,s)} b_e(t),$$

where $R_s^{\text{in}}(t)$ and $R_s^{\text{out}}(t)$ are the total incoming and outgoing call rates of service s, $W_s^{\text{in}}(t)$ is the average incoming work per call (ms), and $B_s^{\text{in}}(t)$ is the incoming byte rate which can be used as an importance weight when prioritizing services under stateful backends or high network overhead. In practice of AdaScale implementation, these quantities are produced by the `service_edge_demand` module as the `services_demand` table, while the per-edge quantities $r_e(t)$, $w_e(t)$, $b_e(t)$ appear in the `edges_demand` table.

An approximate CPU demand (CPU-seconds per second) for service s is then

$$\text{CPU\_demand}_s(t) \approx R_s^{\text{in}}(t) \cdot \frac{W_s^{\text{in}}(t)}{1000}$$

which directly benefits the decisions of resource provisioning for replicas of service s. Here $w_e(t)$ and $W_s^{\text{in}}(t)$ are measured in milliseconds, so $R_s^{\text{in}}(t) \cdot W_s^{\text{in}}(t)/1000$ has units of CPU-seconds per second, i.e., an effective fraction of one CPU core under the simplifying assumption that $w_e(t)$ is dominated by CPU service time.

6.3.5 Networking state (latency-only)

For each node pair $(i, j) \in \mathcal{N} \times \mathcal{N}$, a set of distributed agents continuously measures inter-node latency $L_{i,j}(t)$ using ICMP probes at a configurable interval. An inter-node latency matrix maintains these cross-node delay measurements:

$$L(t) = \{L_{i,j}(t) \mid i, j \in \mathcal{N}\}.$$

AdaScale intentionally does not construct or use an explicit bandwidth prior to the current stage for three reasons: (i) most latency-sensitive microservice applications in our

target setting are bottlenecked by CPU and queueing rather than raw network bandwidth; (ii) considering varying bandwidth across cluster nodes would significantly complicate the scaling decision process; (iii) evolving cross-node latencies across the cluster nodes is enough to emulate a dynamic computing environment. Considering the stateful services existing in the deployed microservice application, byte counters are used through $b_e(t)$ and $B_s^{\text{in}}(t)$ to weight the contributions of the cross-node latencies, while the control objective is driven entirely by *latency* and *resource* metrics.

Given a placement matrix $A(t)$, the expected network latency for a single invocation on edge $e = (u, v)$ between service u and service v is approximated as

$$\bar{L}_e\big(A(t), L(t)\big) \approx \sum_{i,j \in \mathcal{N}} \frac{A_{u,i}(t)}{x_u(t)} \cdot \frac{A_{v,j}(t)}{x_v(t)} \cdot L_{i,j}(t). \tag{6.3}$$

Intuitively, $\bar{L}_e(A, L)$ is the expected network delay per invocation on edge $e = (u, v)$ if caller and callee replicas are chosen uniformly at random among the replicas of u and v under placement A.

6.3.6 Latency composition

For a root request that traverses call graph G_k, its end-to-end latency can be estimated by aggregating contributions along invocation paths:

$$\begin{aligned} Y_k(t) \approx \sum_{(u,v) \in \mathcal{P}_k} \Big[& \bar{L}_{(u,v)}\big(A(t), L(t)\big) \\ & + \phi_{\text{mode}_e}\big(x_u(t), x_v(t), r_e(t)\big) \Big]. \end{aligned} \tag{6.4}$$

where $\mathcal{P}_k$ denotes the multiset of edges on the request's execution path within G_k, $\bar{L}_{(u,v)}$ is the expected replica-averaged inter-node delay in Equation (6.3), and $\phi_{\text{mode}_e}(\cdot)$ captures callee-side queueing and protocol semantics for edge e (e.g., blocking RPC delay, MQ enqueue/dequeue delay, storage I/O) as a function of replica counts and call rate. Note that $\phi_{\text{mode}_e}(\cdot)$ depends on the call rate $r_e(t)$ (through utilization and queueing), whereas $w_e(t)$ is used only in the demand estimates of Section 6.3.4.

In practice, AdaScale does not attempt to solve Equation (6.4) exactly. Instead, it uses

span-level decomposition, which attributes per-request latency contributions to individual services using traces. The approximations, together with the demand matrices described above, are sufficient to drive the optimization in real time.

6.3.7 Correlations with runtime dynamics

The end-to-end latency distribution is shaped by several coupled factors:

- Replica counts $x(t) = \{x_s(t)\}$ determine per-service queueing and contention, and thus affect the ϕ_{mode_e} terms in Equation (6.4).
- Placement $A(t)$ fixes which inter-node latencies $L_{i,j}(t)$ each edge (dependent service pair) experiences in Equation (6.3).
- Call-graph demand, as summarized by $\{r_e(t)\}$ and $\{R_s^{\text{in}}(t)\}$, reshapes both CPU demand and the contribution of each edge (dependent service pair) to end-to-end latency.

These couplings make purely placement-only or scaling-only strategies suboptimal: a bad placement magnifies network delay even with sufficient replicas, while poor replica provisioning creates compute bottlenecks even under favorable network conditions. Therefore, **AdaScale treats *placement* and *replica provisioning* as a joint optimization problem driven by call-graph statistics, demand matrices, latency measurements, and resource constraints.**

6.4 Problem Formulation

Let the measured and estimated system state at time t be

$$S(t) \;=\; \big\{\, L(t),\; \{r_e(t), w_e(t), b_e(t)\}_{e\in E},\; \{R_s^{\text{in}}(t), W_s^{\text{in}}(t)\}_{s\in\mathcal{S}} \,\big\},$$

where $L(t) = \{L_{i,j}(t)\}$ is the inter-node latency matrix, $r_e(t), w_e(t), b_e(t)$ are edge-level demand statistics from Section6.3.3, and $R_s^{\text{in}}(t), W_s^{\text{in}}(t)$ are service-level demands from Section6.3.4.

6.4.1 Latency-weighted objective

Given state $S(t)$, AdaScale jointly determines a replica allocation matrix $A = [A_{s,n}]$ and the replica counts $x_s = \sum_n A_{s,n}$ that minimize total system cost while meeting service-level objectives.

We define a constant value for per-service resource cost with per-replica unit cost c_s (e.g., normalized CPU cost) for running service s, and a network latency cost that weights edges by call rate:

$$\min_{A} \quad \mathrm{Cost}(A; S(t)) \triangleq \underbrace{\sum_{s\in\mathcal{S}} c_s x_s}_{\text{resource cost}} + \underbrace{\lambda_L \sum_{e=(u,v)\in E} r_e(t)\, \bar{L}_e\big(A, L(t)\big)}_{\text{latency cost}} \tag{6.5}$$

s.t.

$$\sum_{s\in\mathcal{S}} A_{s,n}\, R_s \le C_n, \qquad \forall\, n \in \mathcal{N}, \tag{6.6}$$

$$Q_q[Y(x(A), A; S(t))] \le \tau_{\text{e2e}}, \tag{6.7}$$

$$Q_q[Y_s(x(A), A; S(t))] \le \tau_s, \qquad \forall\, s \in \mathcal{S}_{\text{SLO}}, \tag{6.8}$$

$$A_{s,n} \in \mathbb{N}, \quad A_{s,n} \ge 0. \tag{6.9}$$

Here:

- $\lambda_L > 0$ balances resource usage against network latency.
- $\bar{L}_e(A, L(t))$ is the replica-averaged edge latency defined in Equation (6.3).
- $Q_q[Y(\cdot)]$ denotes the q-quantile (e.g.,p50, p90, and p95) of the end-to-end latency distribution; τ_{e2e} is the configured end-to-end SLO for each types of root request.
- $Y_s(\cdot)$ denotes the per-service latency (from raw collected metrics spans), and τ_s are per-service p95 SLOs (from the SLO configuration file). In practice, $Y(x(A), A; S(t))$ and $Y_s(x(A), A; S(t))$ are not evaluated via the analytic model in Equation (6.4), but via recent latency samples collected from the tracing and metrics stack.
- $\mathcal{S}_{\text{SLO}} \subseteq \mathcal{S}$ is the subset of services with explicit local SLOs.

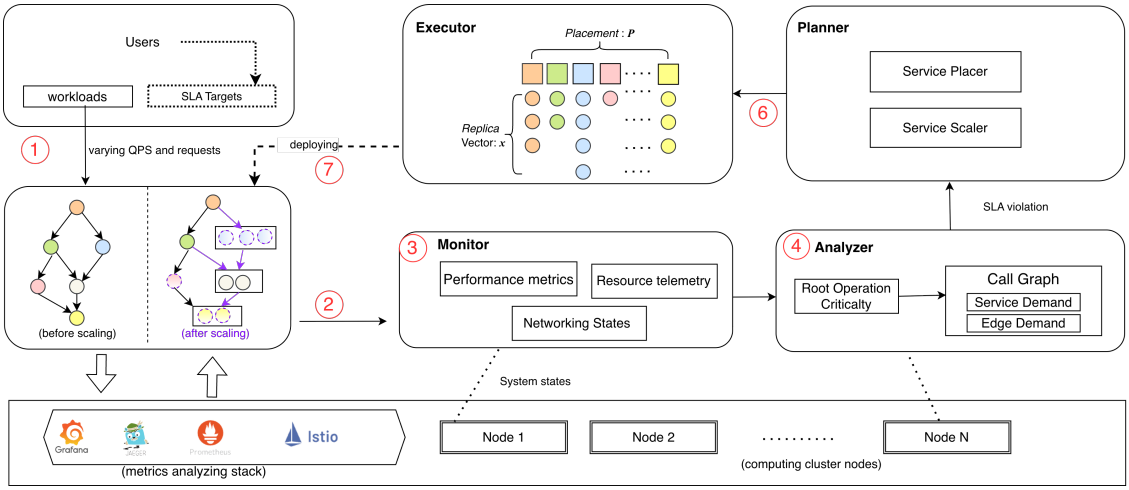


Figure 6.3: AdaScale framework architecture and components.

Constraint Equation (6.6) enforces per-node CPU and memory capacities. Constraint Equation (6.7) requires the q-quantile of end-to-end latency to stay below the global SLO, while Equation (6.8) enforces per-service SLOs where configured.

6.4.2 Demand-driven capacity estimation

Directly solving Equation (6.5)–(6.8) in terms of $A_{s,n}$ is combinatorial and NP–hard. Instead, AdaScale separates the calculation of **Scaling** (i.e., *how many replicas*) from **Placement** (i.e., *where to place them*).

Given service-level demand $\text{CPU\_demand}_s(t)$ from Section 6.3.4 and $\rho_s^{\max} \in (0,1)$ is the target max utilization per replica (e.g., 0.7 for CPU), we first estimate the required replica count for service s as

$$x_s^{\text{dem}}(t) \;=\; \left\lceil \frac{\text{CPU\_demand}_s(t)}{\mu_s\, \rho_s^{\max}} \right\rceil, \tag{6.10}$$

where μ_s is the estimated processing capacity (CPU-seconds per second) of a single replica of s at the full utilization, [1] and per-replica safe capacity will be estimated by $\mu_s\, \rho_s^{\max}$.

This yields a demand-driven replica vector $x^{\text{dem}}(t) \;=\; \{x_s^{\text{dem}}(t)\}$, which is then

[1] In practice, μ_s is calculated by recent Prometheus CPU usage and per-replica throughput metrics.

rounded up to the closest integer and adjusted by the autoscaler to satisfy the SLO constraints.

Conditional on $x^{\mathrm{dem}}(t)$, the placement subproblem chooses a feasible A with $\sum_n A_{s,n} = x_s^{\mathrm{dem}}$ that approximately minimizes the network latency term in Equation (6.5). This two-stage decomposition is realized by the two-timescale adaptation mechanism via: (i) a slower demand-driven autoscaling loop that updates $x^{\mathrm{dem}}(t)$; Compared with the current replica count $x^{\mathrm{current}}(t)$ the $x^{\mathrm{dem}}(t)$, the scaling actions of service s at time t will accordingly be chose from

$$ScaleActions(s,t) \;=\; \{\, Up(s,t), Down(s,t), Hold(s,t)\}$$

; and (ii) while keeping the replica count x_s of service s fixed, a faster reactive placement loop that updates the replica placement matrix A.

6.4.3 Dynamic cross-node delays

The latency matrix $L(t)$ is inherently time-varying due to congestion, background traffic, and externally injected dynamics (e.g., emulated delays for experiments). AdaScale treats $L(t)$ as part of the observed state and reoptimizes placement whenever: (i) the change in average or high-percentile latency exceeds a threshold; or (ii) the end-to-end SLO Equation (6.7) is violated persistently.

Concretely, the reactive placement loop reduces an approximate objective

$$\min_{A'} \sum_{e=(u,v)} r_e(t)\, \bar{L}_e\big(A', L(t)\big)$$

By weighting links by $r_e(t)$, the controller focuses migration budget on edges that both (i) are heavily exercised by the current workload and (ii) suffer from increased cross-node delays.

6.5 Proposed AdaScale Framework

AdaScale is a demand- and latency-aware microservice resource manager that implements a Monitor–Analyzer–Planner–Executor (MAPE) control loop. At a high level, AdaScale repeatedly (i) monitors application and cluster state, (ii) analyzes SLO risk and demand under the current workload mix, (iii) plans a joint scaling and placement update, and (iv) executes the update safely using native cluster mechanisms. A key design principle is a *separation of concerns*: the analysis logic determines *how many* replicas are required per service to satisfy SLOs, while the placement logic determines *where* to place those replicas to minimize latency under time-varying inter-node delays.

At each control epoch t, AdaScale produces a placement plan consisting of: (i) desired replica totals $\{\hat{x}_s(t)\}_{s\in\mathcal{S}}$ and (ii) a replica–node assignment matrix $A^\star(t) = \left[A^\star_{s,n}(t)\right]$ (Section 6.3), where $A^\star_{s,n}(t)$ is the number of replicas of service s planned to run on node n. The plan is computed from the measured state $S(t)$ (Section 6.4), including demand estimates and the inter-node latency matrix $L(t)$.

The control loop mainly comprises five stages:

- **Step 1 (Monitor: cluster exporters).** AdaScale collects cluster state including node capacities, the current replica assignment matrix $A(t)$, and the inter-node latency matrix $L(t)$.
- **Step 2 (Monitor/Analyze: application telemetry and demand tables).** AdaScale collects application telemetry (distributed traces and service metrics) and constructs demand summaries: per-edge demand statistics (call rates/work/bytes) and per-service demand statistics (aggregate call rates and CPU-demand proxies).
- **Step 3 (Analyze: SLO- and demand-aware scaling decisions).** AdaScale evaluates end-to-end and per-service SLO status, identifies the most critical root operations under a mixed workload, and derives per-service scaling actions (replica targets $\hat{x}_s(t)$) using a combination of trace-based criticality and demand estimates.
- **Step 4 (Plan: latency-aware placement).** Given replica targets and the current cluster network state $L(t)$, AdaScale computes a new assignment matrix $A^\star(t)$ that reduces the demand-weighted latency objective while respecting node capacities.

- **Step 5 (Execute: actuation).** AdaScale enforces $\{\hat{x}_s(t)\}$ and $A^\star(t)$ through safe rollouts and scheduling constraints, optionally with bounded parallelism $\kappa \in \mathbb{N}$ to reduce actuation time.

This pipeline aligns with the state definition $S(t)$ and the optimization goal in Section 6.4. We next detail each component.

6.5.1 Monitor

The Monitor produces a time-indexed snapshot of the system state used by subsequent stages. Conceptually, it maintains a cache $\mathcal{D}(t)$, which includes current replica placement matrix $A(t)$, current inter-node latency matrix $L(t)$, node capacities, service metrics, and demand tables.

Workload and performance telemetry. AdaScale collects (i) distributed traces to recover call-graph structure and end-to-end latency samples, (ii) per-service performance metrics such as p50/p90/p95 latencies and error rates, and (iii) resource telemetry such as CPU and memory usage. These measurements are aggregated over a short window W_t to provide a stable estimate of the current operating regime.

Demand tables (call-graph demand estimator) To make scaling and placement explicitly *demand-aware*, AdaScale computes two demand summaries from the same telemetry window:

- an *edge-demand* table that estimates, for each directed service dependency $e = (u, v)$, its call rate $r_e(t)$ (calls/s), expected work $w_e(t)$ (ms, derived from span durations), and (when available) byte rate $b_e(t)$ (bytes/s);
- a *service-demand* table that aggregates incoming/outgoing rates and work, yielding $R_s^{\text{in}}(t)$, $R_s^{\text{out}}(t)$, $W_s^{\text{in}}(t)$, and a CPU-demand proxy $\text{CPU\_demand}_s(t) \approx R_s^{\text{in}}(t) \cdot W_s^{\text{in}}(t)/1000$ (Section 6.3.4).

These demand tables instantiate the quantities in our system model and are reused by both the scaling logic (Analyzer) and the placement logic (Planner).

Algorithm 6.1: *Service Scaler* with multi-root criticality and SLO-/demand-aware scaling

Input: Root request operations $\mathcal{O}$ with $(\tau_o^{\text{e2e}}, \pi_o, \mathcal{S}_o)$; service SLOs $\{\tau_s\}$; observed per-root p95 $\{\widehat{Y}_o^{95}\}$ from traces; observed per-service p95 $\{\widehat{Y}_s^{95}\}$ and utilization u_s from metrics; demand tables providing CPU_demand$_s$ and edge rates $r_{(u,v)}$

Output: Replica targets $\{\hat{x}_s\}$ and weighted edges $\mathcal{E}(t)$

`// Root operation priority`

foreach $o \in \mathcal{O}$ **do**

- $v_o \leftarrow \max\{\widehat{Y}_o^{95} / \max(\tau_o^{\text{e2e}}, 1) - 1, 0\}$
- $\kappa_o \leftarrow \pi_o \cdot v_o$

Select target root op $o^\star$ (e.g., max criticality)

`// Service priorities`

$\mathcal{S}^\star \leftarrow \bigcup \mathcal{S}_{o^\star}$ `// Services triggered by` $o^\star$.

foreach $s \in \mathcal{S}^\star$ **do**

- $\rho_s \leftarrow \widehat{Y}_s^{95} / \max(\tau_s, 1)$ `// pressure ratio`
- Compute trace-based criticality crit$_s$ over W_t
- Compute demand share dem_share$_s$ if demand tables exist (else set dem_share$_s \leftarrow 0$)
- $\eta_s \leftarrow \alpha \cdot \text{crit}_s + (1 - \alpha) \cdot \text{dem\_share}_s$
- score$_s \leftarrow \rho_s \cdot \eta_s$
- Initialize $\hat{x}_s \leftarrow x_s^{\text{cur}}$

`// Propose actions with hysteresis`

foreach $s \in \mathcal{S}^\star$ **do**

- $\hat{x}_s \leftarrow x_s^{\text{cur}}$; `// hold by default`
- **if** $\rho_s > \theta_\uparrow$ ***or*** $x_s^{\text{dem}} > x_s^{\text{cur}}$ **then**
  - action$_s \leftarrow$ `scale_up`
  - $\hat{x}_s \leftarrow x_s^{\text{cur}} + 1$
- **if** $\rho_s < \theta_\downarrow$ ***and*** $u_s < u_\downarrow$ **then**
  - action$_s \leftarrow$ `scale_down`
  - $\hat{x}_s \leftarrow \max\left(x_s^{\min},\ x_s^{\text{cur}} - 1\right)$
- **else**
  - action$_s \leftarrow$ `hold`

`// Scale-up budget`

Let $\mathcal{U} \leftarrow \{s \in \mathcal{S}^\star : \hat{x}_s > x_s^{\text{cur}}\}$

Keep only the top-K services in $\mathcal{U}$ by score$_s$ as scale-up; set others to `hold` ($\hat{x}_s \leftarrow x_s^{\text{cur}}$)

`// Export edge weights for placement`

Construct $\mathcal{E}(t) = \{(u, v, \omega_{u,v})\}$ from demand/trace edges relevant to $\mathcal{O}^\star$

Set $\omega_{u,v} \propto r_{(u,v)}$

return $\{\hat{x}_s\}, \mathcal{E}(t)$

Algorithm 6.2: *Service Placer* with greedy latency- and capacity-aware replica placement

Input: Current placement A; target replicas $\{\hat{x}_s\}$; latency matrix $L(t)$; weighted edges $\mathcal{E}(t)$; node capacities and per-replica requests $\{R_s\}$
Output: Planned placement $A^\star$
$A^\star \leftarrow A$

Function `Cost(`$A^\star$`)`
  LatCost $\leftarrow \sum_{(u,v,\omega)\in\mathcal{E}(t)} \omega \cdot \bar{L}_{(u,v)}(A^\star, L(t))$
  CapPenalty $\leftarrow$ large penalty for exceeds capacity
  return LatCost + CapPenalty

foreach *service* s **do**
  while $\sum_n A^\star_{s,n} < \hat{x}_s$ **do**
    Find node $n^\star \in \mathcal{N}$ minimizing $\textsc{Cost}(A^\star + \mathbf{1}_{(s,n)})$
    $A^\star_{s,n^\star} \leftarrow A^\star_{s,n^\star} + 1$
  while $\sum_n A^\star_{s,n} > \hat{x}_s$ **do**
    Let $\mathcal{N}_s = \{n : A^\star_{s,n} > 0\}$
    Find node $n^\star \in \mathcal{N}_s$ minimizing $\textsc{Cost}(A^\star - \mathbf{1}_{(s,n)})$
    $A^\star_{s,n^\star} \leftarrow A^\star_{s,n^\star} - 1$

return $A^\star$

Cluster resource and placement telemetry. AdaScale snapshots node capacities (CPU and memory) and the current replica assignment matrix $A(t)$. For each service s, $A_{s,n}(t)$ records how many replicas are currently placed on node n, and $x_s(t) = \sum_n A_{s,n}(t)$ is the service replica count. The snapshot also records per-replica resource requests R_s needed for capacity checks in the planner.

Networking state (inter-node latency). AdaScale continuously measures or infers a matrix of inter-node latencies $L(t) = \{L_{i,j}(t)\}$. We treat $L(t)$ as a first-class runtime signal: changes in $L(t)$ can trigger placement updates even if the workload mix is unchanged. This design captures dynamic environments (e.g., congestion, background interference, or injected delays) without requiring explicit bandwidth modeling.

6.5.2 Planner

6.5.3 Analyzer

The Analyzer transforms $\mathcal{D}(t)$ into per-service scaling actions and a compact, demand-weighted call-graph representation for placement. The Analyzer implements two coupled tasks: (i) *SLO risk assessment under mixed root operations* and (ii) *demand-aware scaling decisions*.

Root-operation criticality under mixed workloads. Microservice applications often serve multiple root request classes $\mathcal{O} = \{o_1, \ldots, o_K\}$, each with its own end-to-end SLO target τ_o^{e2e} and workload proportion π_o (from configuration or telemetry, with $\sum_o \pi_o = 1$). The Analyzer computes an observed end-to-end p95 latency $\widehat{Y}_o^{(95)}(t)$ per root operation and defines a workload-weighted SLO risk score:

$$v_o(t) \triangleq \max\left\{\frac{\widehat{Y}_o^{(95)}(t)}{\max\{\tau_o^{\text{e2e}}, 1\}} - 1,\ 0\right\}, \qquad \kappa_o(t) \triangleq \pi_o \cdot v_o(t).$$

The system prioritizes root operations with large $\kappa_o(t)$, ensuring that optimization effort is focused on request classes that are both frequent and SLO-threatening.

Service criticality and scaling actions. To select which services to scale, AdaScale computes a trace-based service criticality score $\text{crit}_s(t)$, and combines it with demand signals such as $\text{CPU\_demand}_s(t)$ or its normalized share. Given per-service SLO thresholds τ_s and observed p95 latencies $\widehat{Y}_s^{(95)}(t)$, the Analyzer derives a pressure ratio $\rho_s(t) = \widehat{Y}_s^{(95)}(t) / \max\{\tau_s, 1\}$ and proposes an action $a_s(t) \in \{\text{scale\_up}, \text{scale\_down}, \text{hold}\}$ together with a replica target $\hat{x}_s(t)$. To avoid oscillations and limit control-plane disruption, AdaScale enforces a budget: only the top-K services by a composite priority (pressure combined with criticality/demand) are allowed to scale up in each epoch, while remaining services are held.

Demand-weighted edge set for placement. For placement planning, the Analyzer exports a weighted edge set $\mathcal{E}(t) = \{(u, v, \omega_{u,v}(t))\}$, where (u, v) is a service dependency

and $\omega_{u,v}(t)$ is proportional to the edge demand (e.g., call rate). This edge set summarizes which dependencies dominate end-to-end performance under the current workload mix and provides the primary signal for latency-aware placement.

The Planner (Controller) computes a replica assignment plan $A^\star(t)$ given (i) replica targets $\{\hat{x}_s(t)\}$ from the Analyzer, (ii) current placement $A(t)$, (iii) node capacities, and (iv) the measured inter-node latency matrix $L(t)$.

Latency- and capacity-aware objective. The Planner minimizes a demand-weighted latency objective subject to capacity constraints. Using the weighted edge set $\mathcal{E}(t)$ and the replica-averaged edge latency $\bar{L}_{(u,v)}(A, L(t))$ (Equation (6.3)), it evaluates:

$$\text{LatCost}(A) = \sum_{(u,v)\in\mathcal{E}(t)} \omega_{u,v}(t) \cdot \bar{L}_{(u,v)}(A, L(t)).$$

To discourage capacity violations, it adds a large penalty when the implied CPU or memory requests exceed any node capacity, yielding $\text{Cost}(A; S(t)) = \text{LatCost}(A) + \text{CapPenalty}(A)$ as described in Section 6.5.2.

Greedy replica placement and removal. As the joint problem is combinatorial, the Planner uses a greedy local search guided by $\text{Cost}(A; S(t))$. For each service s, it compares the current replica count $x_s(t) = \sum_n A_{s,n}(t)$ with the target $\hat{x}_s(t)$: if $\hat{x}_s(t) > x_s(t)$ it adds replicas one-by-one to the node that yields the lowest cost; if $\hat{x}_s(t) < x_s(t)$ it removes replicas one-by-one from the node whose removal causes the smallest cost increase. This produces a new assignment $A^\star(t)$ that is simultaneously demand-aware (through $\omega_{u,v}(t)$), latency-aware (through $L(t)$), and capacity-aware (through resource penalties).

6.5.4 Executor

The Executor actuates the scaling and placement decisions produced by the Planner. Its input is the plan $(\{\hat{x}_s(t)\}, A^\star(t))$ for the current epoch. For each service s, it (i) enforces the replica target $\hat{x}_s(t)$ and (ii) constrains scheduling to the allowed node set $\mathcal{N}_s = \{n \mid A^\star_{s,n}(t) > 0\}$ so that replicas run on planner-chosen nodes. When feasible, it additionally

applies spreading rules that bias replicas toward the intended distribution encoded by $A^{\star}(t)$. The Executor monitors rollout progress and declares success once the cluster converges to a state consistent with the plan (subject to timeouts). To reduce actuation latency, it supports bounded parallelism with a configurable concurrency limit κ.

When putting all stages together, each decision epoch: (i) refreshes $\mathcal{D}(t)$ from telemetry, (ii) computes scaling actions and edge weights in the Analyzer, (iii) computes $A^{\star}(t)$ in the Planner under the current $L(t)$, and (iv) enforces the resulting plan in the Executor. This completes one MAPE iteration and realizes the two-stage decomposition: *how many* replicas are needed is decided by SLO- and demand-driven analysis, while *where* to place replicas is decided by latency-aware planning under dynamic cross-node delays.

6.6 Performance Evaluation

6.6.1 Cluster Testbed

We prototype and evaluate AdaScale on a Kubernetes-based cloud–edge testbed [12]. The testbed comprises 16 virtual machines (VMs): one control-plane node and fifteen worker nodes. The control-plane VM is equipped with 32 CPU cores (AMD EPYC 7763, x86_64), 32 GiB RAM, and a 16 Gbps network interface. Each worker VM provides 8 CPU cores from the same EPYC processor family, 32 GiB RAM, and a 16 Gbps link.

Software stack. All nodes run Ubuntu 22.04.2 LTS with Linux kernel 5.15.0. We deploy Kubernetes v1.27.4 on ten nodes and Kubernetes v1.28.4 on the remaining five nodes. Calico v3.26.1 is used as the CNI, Istio v1.20.3 provides the service mesh, and CRI-O serves as the container runtime (v1.27.1 on ten nodes and v1.28.11 on five nodes).

Intra-cluster connectivity and network dynamics. The VMs are hosted in the universitys dedicated research cloud. As a result, the baseline inter-node round-trip latency is very small (typically 0.2–1 ms), which we confirmed using ICMP ping measurements. To emulate cloud–edge networking dynamics in a controlled and repeatable manner, AdaScale programmatically applies cross-node delay constraints within the cluster. Unless stated otherwise, we refresh these injected conditions periodically during experi-

Table 6.1: Per-edge demand statistics.

Src	Dst	p_e	$r_e^{\text{per-req}}$	w_e [ms]	r_e [calls/s]	b_e [bytes/s]
S0	S1	0.106	1.063	275.376	10.63	–
S0	S2	0.311	3.110	3.884	31.102	–
S0	S3	0.567	5.669	2.618	56.693	–
S1	S2	0.106	1.063	2.401	10.63	8913
S1	S3	0.106	1.063	40.996	10.63	9281
S1	S4	0.106	1.063	87.745	10.63	73395
S1	S5	0.106	1.063	0.023	10.63	8237
S1	S6	0.106	1.063	0.012	10.63	10886
S1	S7	0.106	1.063	0.014	10.63	13201
S1	S8	0.106	1.063	10.161	10.63	52095
S2	S8	0.311	3.110	0.011	31.102	21564
S3	S8	0.559	5.591	0.011	55.906	43380
S3	S9	0.106	1.063	38.724	10.63	7808
S4	S10	0.106	1.063	24.360	10.63	10915
S4	S11	0.106	1.063	1.497	10.63	39275

ments so that we can assess how well the evaluated policies adapt to time-varying network environments.

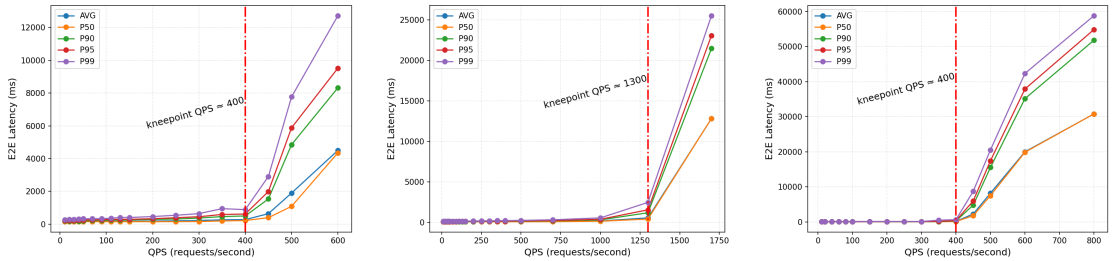
Benchmark application and request generation. We use the Social Network application from DeathStarBench [2] as the representative microservice workload, and we generate client requests using `wrk2` [113]. Social Network implements a simplified social-media service and consists of 27 microservices supporting operations such as composing posts as well as reading user and home timelines. Different request types exercise different call-graph structures and traffic patterns, enabling us to evaluate AdaScale under diverse end-to-end execution paths.

6.6.2 Root Requests Analysis

In the Social Network benchmark, there are three types of root request operations. To define the SLO targets for each root request, different QPS are used to measure end-to-end performance and identify the kneepoint at which it is significantly affected. When QPS (Queries Per Second) increases, the tail latency of the corresponding microservice first gradually increases, then suddenly increases sharply at a specific QPS and latency

value. We refer to such a point as a *kneepoint* to represent the QoS-violation point for a specific root operation request. Thus, under our experiment environment, we define the SLO target of a specific root operation by *kneepoint QPS*, where *max_load* QPS is reached.

To find the *kneepoint QPS* of each root request operation of the deployed social network application, we conducted load test for each root request operation by gradually increasing the corresponding QPS from lower values to higher values, which saturate the deployed microservice application under each root request operation. As shown in Figure 6.4, we demonstrated different percentile metrics including AVG (average), p50 (median), p90 (90th Percentile), p95 (95th Percentile), and p99 (99th Percentile) for the response time under each root request. For all the root request operations, i.e., $\lambda_1(t)$, $\lambda_2(t)$ and $\lambda_3(t)$, each percentile metric shows similar increasing trends as the QPS increases. Specifically, the kneepoint QPS for each root operation is approximately 400, 1300, and 400.



(a) Kneepoint QPS exploration of root request $\lambda_1(t)$ **(b)** Kneepoint QPS exploration of root request $\lambda_2(t)$ **(c)** Kneepoint QPS exploration of root request $\lambda_3(t)$

Figure 6.4: In the Social Network benchmark, there are three types of root requests. To define the SLOs for each root request, different QPS are sent to obtain the end-to-end performance and find the kneepoint, at which the end-to-end performance significantly being affected.

6.6.3 Service and Edge Demands in Call Graphs

To get fine-grained statistics of the call graph at run time, we define the service demand and edge demand in Section 6.3.4. In the evaluation, we present a snapshot of the runtime service and edge demand under a 5-minute sustained mix workload (root requests proportion 1:3:6). Table 6.1 shows per-edge demand statistics inferred from Jaeger traces

and Prometheus traffic for a single time window W_t. For each edge $e = (u \to v)$ we list the empirical probability p_e, the expected number of traversals per root request $r_e^{\text{per-req}}$, the mean edge-local service time w_e in milliseconds, the estimated call rate r_e (calls/s), and the byte rate b_e (bytes/s).

Under 5 minutes sustained mix workload (root requests proportion $\lambda_1 : \lambda_2 : \lambda_3 = 1 : 3 : 6$) to the deployed microservice application, per-service demand statistics aggregated from edge-level demand in Table 6.1 for the same time window W_t. For each service s, we list R_s^{in}, R_s^{out}, W_s^{in}, and B_s^{in} as defined in Section 6.3.4. Besides, Table 6.2 demonstrates the per-service demand statistics aggregated from edge-level demand in Table 6.1 for the same time window W_t. For each service s, we list R_s^{in}, R_s^{out}, W_s^{in}, and B_s^{in} as defined in Section 6.3.4.

In Table 6.1, $S0$ is the entry service which receives root requests and triggers the downstream services $S1$, $S2$ and $S3$. It can be observed that the probability p_e of the root requests exercising the edges ($S0 \to S1$, $S0 \to S2$, $S0 \to S3$) are 0.106, 0.311, and 0.567, which are almost the same as the proportions of each request ($\lambda_1 : \lambda_2 : \lambda_3 = 1 : 3 : 6$). For Table 6.2, the in-degree and out-degree counts of each service exactly match the actual service call graph structure in Social Network [2]. For instance, considering the entry service $S0$, the in-degree count is 0 because $S0$ is the entry service of the call graph, thus there is no upstream service for $S0$, and the out-degree count is 3 because the downstream services of $S0$ consist of $S1$, $S2$, and $S3$. Therefore, both tables accurately approximate the fine-grained statistics of the call graph.

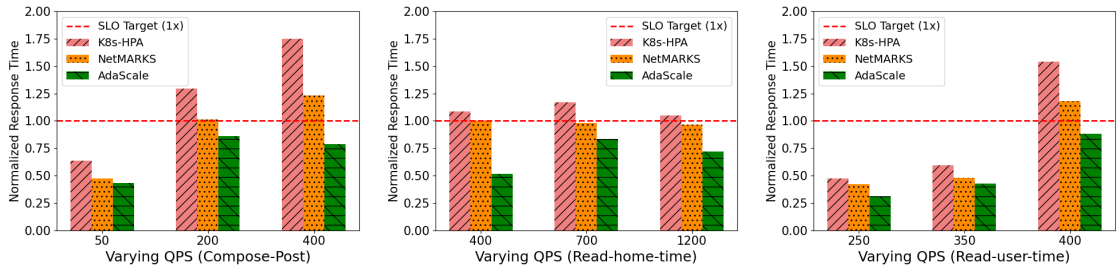
6.6.4 Performance Comparison

Compared Methods. To evaluate our proposed AdaScale framework, we compared it with the Kubernetes Horizontal Pod Autoscaler (HPA) policy and the traffic-aware NetMARKS [32], which targets network-aware management for microservice applications. For K8s HPA, this is mainly achieved control loop that automatically adjusts the number of pod replicas in a workload (such as a Deployment or StatefulSet) to match observed demand. This is done to maintain performance and optimize resource usage without manual intervention. For the method of NetMARKS, it adopts a similar service mesh

Table 6.2: Per-service demand statistics.

Service	in_deg	out_deg	R_s^{in} [calls/s]	R_s^{out} [calls/s]	W_s^{in} [ms]	B_s^{in} [bytes/s]
S0	0	3	0	98.425	0	0
S1	1	7	10.63	74.409	275.376	0
S2	2	1	41.732	31.102	3.506	8913
S3	2	2	67.323	66.535	8.678	9281
S4	1	2	10.63	21.26	87.745	73395
S5	1	0	10.63	0	0.023	8237
S6	1	0	10.63	0	0.012	10886
S7	1	0	10.63	0	0.014	13201
S8	3	0	97.638	0	1.116	117039
S9	1	0	10.63	0	38.724	7808
S10	1	0	10.63	0	24.360	10915
S11	1	0	10.63	0	1.497	39275

with AdaScale, but focuses on optimizing the most communicated microservice pairs.



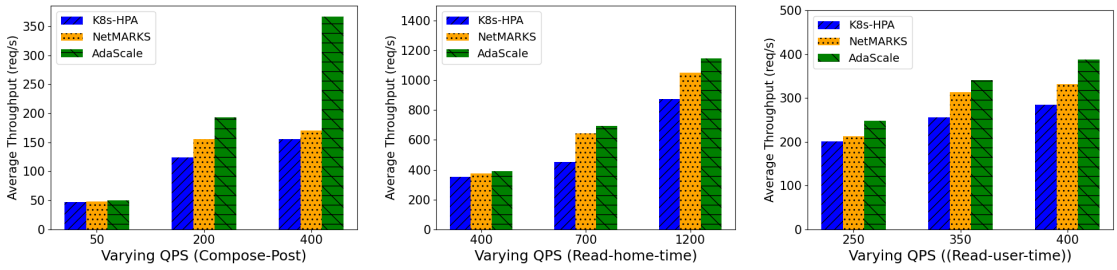
(a) root operation 1 `Compose Post` **(b)** root operation 2 `Read User Timline` **(c)** root operation 3 `Read Home timeline`

Figure 6.5: Under three types of root request operations and varying QPS, AdaScale outperformed existing methods in terms of response time.

End-to-end Performance. The experiment results showed improved throughput and decreased average response time for different root operations (requests).

Response Time. For the `social network` benchmark application, we used the wrk2 tool to generate three request types:compose-post, read-user-timeline, and read-home-timeline. Each type shows distinct call graphs and traffic patterns across the applications

dependency graph. Fig. 6.5 compares the normalized average response times under various workloads, including changes in QPS and root request operations. For these request types, AdaScale consistently outperforms existing methods, meeting SLO targets across scenarios. Compared to NetMARKS [32], AdaScale achieves up to 1.56x lower response times for `compose-post`, 1.93x for `read-home-timeline`, and 1.34x for `read-user-timeline` requests. Obviously, AdaScale outperforms existing methods under varied workloads.



(a) root operation 1 `Compose Post` **(b)** root operation 2 `Read User Timeline` **(c)** root operation 3 `Read Home timeline`

Figure 6.6: Under three types of root request operations and varying QPS, AdaScale outperformed existing methods in terms of throughput.

Throughput. Throughput critical metrics for evaluating the deployed applications' end-to-end performance in dynamic computing environments. Throughput refers to the total quantity of transmitted data, including protocol overhead and retransmissions. In terms of *throughput*, we conducted experiments to measure the average throughput (requests/second) with different QPS of workloads and mixed root request operations. As shown in the three subplots of Fig. 6.6, the proposed AdaScale achieves higher throughput across various scenarios. Specifically, compared with NetMARKS [32], our method shows an overall higher throughput and outperforms it by up to 2.16x for compose-post requests, 1.32x for read-home-timeline requests, 1.36x for read-user-timeline requests. Thus, compared with the existing methods k8s-burstable and NetMARKS, our proposed method AdaScale achieves higher overall throughput across various scenarios.

6.7 Related Work

The management of microservices in different scenarios has been extensively explored. This section reviews existing literature emphasizing microservice management, call-graph analysis, and network-aware scheduling.

6.7.1 Microservice Management

FIRM [18], Erms [19], GrandSLAm [109], and Sage [30] leverage online telemetry and modeling to localize SLO violations, assign latency budgets to services, and trigger fine-grained reprovisioning or request reordering in shared microservice environments. TraDE [67] is proposed to redeploy microservice instances to maintain QoS under changing workloads and network conditions. CoScal [122] is presented as a multi-faceted scaling method integrating workload prediction methods and reinforcement learning. Yi Hu et al. investigate the dynamic service mesh orchestration by using probabilistic routing for microservice call graphs. Kai Peng et al. study large-scale service-mesh orchestration using Jackson queuing network theory and a three-stage heuristic [86]. However, these methods tend to ignore the influences of different types of root requests.

6.7.2 Call-Graph Dynamics Analysis

Several works analyze microservice dependencies and their impact on performance. Tian et al. [138] synthesize task-dependency graphs for data-parallel jobs from large-scale cluster traces. Yi Hu et al. introduce a joint optimization method for service deployment and request routing via fine-grained queuing network analysis [64]. Luo et al. [123] characterize microservice call graphs from Alibaba traces, categorizing microservice dependencies into three distinct types. Parslo [29] decomposes end-to-end SLO budgets into node-specific latency targets using gradient descent. Sage [30] also models dependencies when diagnosing QoS violations. However, these methods are generally time-consuming with high overheads to build the graph and are not suitable for dynamic incoming user requests.

6.7.3 Network-aware Microservice Scheduling

Network-aware microservice management methods exploit infrastructure-level metrics when placing microservice instances. Fangyu Zhang et al introduce a network-aware reliability model and optimized microservice placement algorithms to enhance service reliability and reduce bandwidth consumption in mobile and IoT networks [124]. NetMARKS [32] is proposed as a dynamic Kubernetes scheduling approach leveraging Istio [42] network metrics to optimize containerized workflows for 5G edge applications. Marchese et al. extend the default Kubernetes scheduler with network-aware placement policies [34, 35]. OptTraffic [33] is a network-aware scheduling system that minimizes cross-machine traffic in containerized microservices under multi-replica deployment. However, these research works have limitations in considering the dynamic networking states across the cluster nodes.

6.8 Summary

In this chapter, we designed and implemented AdaScale, an adaptive scaling and placement framework for microservices under multiple concurrent dynamics, including cross-node delays, dynamic call graphs, and evolving request operations. AdaScale is organised as a Monitoring–Analysis–Planning–Execution (MAPE) control loop that integrates telemetry from call-graph reconstruction, traffic stress analysis, and network measurement, and separates adaptation into two timescales: fast placement reactions for imminent SLA violations and slower autoscaling for sustained workload and mix changes. The next chapter concludes the thesis by synthesising how EN-Beats, iDynamics, TraDE, and AdaScale collectively enable a unified telemetry-driven approach to dynamics-aware microservice management, and by outlining open directions for research and practice.

Chapter 7
Conclusions and Future Directions

This chapter concludes the thesis by synthesising its main findings and outlining future research directions for telemetry-driven, dynamics-aware management of microservice-based applications in cloud and cloud–edge environments. It first summarises how the thesis addresses workload/resource-usage dynamics, call-graph and traffic dynamics, and time-varying network conditions through four contributions: EN-Beats for correlation-aware multi-resource prediction; iDynamics for controllable, repeatable evaluation of scheduling policies under cloud–edge dynamics; TraDE for traffic- and network-aware adaptive rescheduling; and AdaScale for two-timescale, SLO-aware scaling and placement under multiple concurrent dynamics. The chapter then discusses promising future research directions, including multi-cluster coordination across the cloud–edge continuum, stronger safety and stability guarantees for learning-based controllers, and sustainability-aware extensions. It closes with final remarks on how treating dynamics and telemetry as first-class concerns enables more robust and cost-efficient microservice deployments.

7.1 Summary of Contributions

Thesis scope and research questions

This thesis investigated how to design, evaluate, and implement dynamics-aware mechanisms for managing microservice-based applications under multiple interacting sources of uncertainty. Building on the taxonomy in Chapter 2, the thesis focused on three recurring forms of dynamics that are particularly influential for microservices: (i) workload and resource-usage dynamics, (ii) call-graph and traffic dynamics, and (iii) network dynamics (cross-node latency/bandwidth variability). The central research question can be stated as:

How can we design prediction, evaluation, scheduling, and scaling mechanisms that maintain application-level SLOs and resource efficiency for microservice-based cloud and cloud-edge systems under multiple, concurrent runtime dynamics?

To answer this question, the thesis addressed four sub-questions (RQ1–RQ4) introduced in Chapter 1. The remainder of this section summarises how the thesis contributions answer these questions and how they collectively advance the state of the art.

7.1.1 EN-Beats (RQ1): Correlation-aware multi-resource prediction for cloud VMs

Chapter 3 introduced EN-Beats, an ensemble learning-based method for predicting multiple correlated resource metrics (CPU utilisation, memory usage, and network traffic) for VMs in dynamic cloud environments. EN-Beats consists of two key ideas: (i) *RCorrPolicy*, a correlation-aware metric selection policy based on Spearman correlation, and (ii) an ensemble of N-BEATS-based predictors aggregated into a strong learner for multi-resource forecasting [1, 94].

Using the Bitbrains production trace, the evaluation showed that correlation-aware metric selection improves forecasting fidelity and that EN-Beats provides accurate short-horizon predictions suitable for downstream control decisions [3]. From the taxonomy perspective, EN-Beats operates at the VM/host abstraction level and directly addresses workload/resource-usage dynamics, filling a gap where many predictors target single metrics or assume fixed statistical workload structure.

7.1.2 iDynamics (RQ2): A controllable evaluation framework for cloud–edge microservice dynamics

Chapter 4 proposed iDynamics, a configurable and extensible emulation framework for evaluating microservice scheduling policies under controllable dynamics in the cloud–edge continuum [70]. iDynamics combines three modular capabilities:

- the *Graph Dynamics Analyzer*, which reconstructs call graphs from traced requests and quantifies traffic imbalances across upstream–downstream pairs;

- the *Networking Dynamics Manager*, which injects and measures destination-specific cross-node latency and bandwidth using distributed agents;
- the *Scheduling Policy Extender*, which provides interfaces and utilities to implement and test custom scheduling policies.

By enabling repeatable experiments under combinations of call-graph, traffic, and network dynamics, iDynamics addresses a critical methodological gap: many prior studies evaluate policies under partial or static dynamics, limiting the credibility and comparability of results. In the taxonomy, iDynamics spans the application-graph and cloud–edge levels and bridges real orchestration stacks with controllable dynamics injection.

7.1.3 TraDE (RQ3): Traffic- and network-aware adaptive scheduling under dynamic delays

Chapter 5 presented TraDE, a traffic- and network-aware adaptive scheduling framework for microservices under dynamic workloads and cross-node network conditions [67]. TraDE extends Kubernetes with four main components:

- a *traffic stress analyser* that captures bidirectional traffic between upstream and downstream microservices (including corresponding replicas),
- a *network dynamics manager* that injects and measures destination-specific delays,
- a *PGA-based service–node mapper* that computes placements to minimise traffic-weighted latency under load-balance constraints, and
- a *microservice rescheduler* that performs safe migrations while preserving service availability.

Evaluations with realistic benchmarks showed that TraDE reduces response time and improves throughput compared with existing approaches (including network-aware baselines such as NetMARKS) by making scheduling decisions that are explicitly informed by traffic stress and network variability [32]. TraDE therefore fills a key gap in practical, replica-level rescheduling under combined traffic and network dynamics.

7.1.4 AdaScale (RQ4): SLO-aware scaling and placement under multiple dynamics

Chapter 6 introduced AdaScale, an adaptive scaling and placement framework that jointly preserves E2E SLOs for microservices under multiple concurrent dynamics. AdaScale formalises a two-timescale adaptation architecture: a fast reactive placement loop to mitigate imminent SLO violations (e.g., latency spikes driven by network perturbations or traffic hotspots) and a slower autoscaling loop that adjusts replica counts based on sustained workload changes and workload-mix effects.

AdaScale integrates telemetry from call-graph reconstruction, traffic stress analysis, and network measurement into a unified Monitoring–Analysis–Planning–Execution (MAPE) control loop. Experiments on cloud–edge clusters showed that the two-timescale decomposition improves robustness: transient events are handled by fast placement reactions while longer-term shifts are absorbed by autoscaling. In the taxonomy, AdaScale spans microservice and application-graph levels and addresses the gap where scaling and placement are often treated as separate, weakly coupled mechanisms.

Cross-cutting synthesis

Taken together, the thesis contributions provide a coherent progression from *forecasting* to *evaluation* to *control*. EN-Beats improves the fidelity of telemetry-driven predictions for proactive decisions; iDynamics makes microservice dynamics controllable and measurable for rigorous evaluation; TraDE translates traffic and network insights into concrete rescheduling actions; and AdaScale unifies scaling and placement via a multi-timescale control architecture. Collectively, these methods demonstrate that treating dynamics and telemetry as first-class concerns enables more robust, SLA-compliant, and resource-efficient microservice deployments in both cloud and cloud–edge settings.

7.2 Future Research Directions

Building on the taxonomy and the four contributions, several promising research directions emerge.

7.2.1 Predictive and generative modelling for evolving call graphs and traffic

While this thesis leverages call-graph reconstruction and traffic stress analysis, future work can explore more predictive models of *call-graph evolution* and *traffic redistribution* under feature rollouts and changing request mixes. Combining trace-derived graph models with forecasting (analogous to EN-Beats at the resource level) could enable proactive placement and scaling decisions that anticipate topology shifts rather than reacting after QoS degradation.

7.2.2 Learning-based controllers with stability and safety guarantees

TraDE and AdaScale primarily rely on algorithmic heuristics and carefully designed control rules. A key direction is to incorporate learning-based controllers (e.g., reinforcement learning or online optimisation) that operate on the abstractions exposed by iDynamics, while providing explicit stability and safety guarantees. This includes avoiding control-loop interference among autoscaling, placement, and traffic routing, and ensuring bounded SLA violation probability under partial observability.

7.2.3 Multi-cluster and wide-area coordination across the cloud–edge continuum

The evaluated systems focus on single clusters or tightly connected cloud–edge environments. As deployments increasingly span multiple clusters and geographic regions, future work should extend dynamics-aware scheduling and scaling to wide-area scenarios. This includes modelling inter-region latency and bandwidth, coordinating placement across clusters, and handling policy constraints such as data locality and compliance.

7.2.4 Sustainability-aware objectives: energy, cost, and carbon

Extending iDynamics, TraDE, and AdaScale to incorporate sustainability objectives is increasingly important. Future work can integrate energy measurement and carbon-intensity signals into the control loop, enabling policies that trade off latency, cost, and

carbon impact. The two-timescale structure of AdaScale is a natural fit: fast loops can maintain SLOs while slow loops optimise energy and carbon budgets over longer horizons.

7.2.5 Support for emerging AI-driven microservice workloads

Microservice platforms are increasingly used to serve AI workloads (e.g., LLM inference pipelines, vector databases, multi-stage retrieval-augmented generation), which introduce new dynamics: heterogeneous accelerators, queueing effects, large payloads, and sensitivity to network jitter. Extending traffic and network-aware scheduling and scaling to such pipelines requires richer performance models, new telemetry signals (e.g., token-rate, queue depth), and co-scheduling of compute and network resources.

7.3 Final Remarks

This thesis presented a systematic exploration of telemetry-driven, dynamics-aware management for microservice-based cloud and cloud–edge systems. Grounded in a taxonomy of dynamics and control mechanisms, the thesis developed four complementary contributions: EN-Beats for correlation-aware multi-resource prediction, iDynamics for controllable evaluation under microservice dynamics, TraDE for traffic- and network-aware adaptive scheduling, and AdaScale for two-timescale SLA-aware scaling and placement.

Collectively, these contributions show that robust SLA/SLO compliance under realistic dynamics requires integrating prediction, observability, evaluation, and control across multiple system layers and timescales. By making dynamics explicit and by exploiting telemetry as a primary input to decision-making, the thesis advances practical foundations for building more predictable, efficient, and resilient cloud and cloud–edge microservice deployments.

Bibliography

[1] B. N. Oreshkin, D. Carpov, N. Chapados, and Y. Bengio, "N-beats: Neural basis expansion analysis for interpretable time series forecasting," in *Proceedings of the International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=r1ecqn4YwB

[2] Y. Gan, C. Delimitrou, and et al, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, p. 318.

[3] S. Shen, V. Van Beek, and A. Iosup, "Statistical characterization of business-critical workloads hosted in cloud datacenters," in *Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 465–474.

[4] Alibaba, "Alibaba microservice distributed traces," https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022, 2025, accessed: 2025-11-02.

[5] Meta, "Facebook distributed traces," https://github.com/facebookresearch/distributed_traces, 2025, accessed: 2025-11-02.

[6] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: current and future directions," *SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, p. 2945, jan 2018.

[7] S. Pallewatta, V. Kostakos, and R. Buyya, "Placement of microservices-based iot applications in fog computing: A taxonomy and future directions," *ACM Computing Surveys*, vol. 55, no. 14s, July 2023.

[8] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013. [Online]. Available: https://doi.org/10.1145/2408776.2408794

[9] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: The next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: ACM, 2020.

[10] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 29–45, 2018.

[11] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[12] K. A. open-source container orchestration system, https://kubernetes.io/, 2024, accessed: 2024-05-25.

[13] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015.

[14] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2021.

[15] K. Fu, W. Zhang, Q. Chen, D. Zeng, X. Peng, W. Zheng, and M. Guo, "Qos-aware and resource efficient microservice deployment in cloud-edge continuum," in *Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 932–941.

[16] A. Detti, L. Funari, and L. Petrucci, "bench: An open-source factory of benchmark microservice applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 968–980, 2023.

[17] F. Du, J. Shi, Q. Chen, P. Pang, L. Li, and M. Guo, "Generating microservice graphs with production characteristics for efficient resource scaling," in *Proceedings of the*

ACM International Conference on Supercomputing (ICS '25), ser. ICS '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 1–16.

[18] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 805–825.

[19] S. Luo, C. Xu, and et al, "Erms: Efficient resource management for shared microservices with sla guarantees," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023. New York, NY, USA: ACM, 2022, p. 6277.

[20] S. Luo, C. Lin, K. Ye, G. Xu, L. Zhang, G. Yang, H. Xu, and C. Xu, "Optimizing resource management for shared microservices: A scalable system design," *ACM Trans. Comput. Syst.*, vol. 42, no. 12, Feb 2024.

[21] "Kubernetes horizontal pod autoscaling." https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2025, accessed: 2025-11-06.

[22] K. Autoscaler, "Vertical pod autoscaler," https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler, 2025, accessed: 2025-12-21.

[23] ——, "Cluster autoscaler," https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler, 2025, accessed: 2025-12-21.

[24] KEDA, "Keda: Kubernetes event-driven autoscaling," https://keda.sh/, 2025, accessed: 2025-12-21.

[25] Knative, "Knative," https://knative.dev/, 2025, accessed: 2025-12-21.

[26] Z. Ding, S. Wang, and C. Jiang, "Kubernetes-oriented microservice placement with dynamic resource allocation," *IEEE Transactions on Cloud Computing*, 2022.

[27] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," in *Proceedings of the Twenty-First International Conference on Ar-*

chitectural Support for Programming Languages and Operating Systems, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, p. 3347.

[28] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: Ml-based and qos-aware resource management for cloud microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: ACM, 2021, p. 167181.

[29] A. Mirhosseini, S. Elnikety, and T. F. Wenisch, "Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, USA: ACM, 2021, p. 442457.

[30] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: practical and scalable ml-driven performance debugging in microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: ACM, 2021, p. 135151.

[31] C. Song, M. Xu, K. Ye, H. Wu, S. S. Gill, R. Buyya, and C. Xu, "Chainsformer: A chain latency-aware resource provisioning approach for microservices cluster," in *Proceedings of the 21st International Conference on Service-Oriented Computing (ICSOC 2023)*. Berlin, Heidelberg: Springer-Verlag, 2023, p. 197211.

[32] Ł. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "NetMARKS: Network metrics-aware kubernetes scheduler powered by service mesh," in *Proceedings of IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–9.

[33] X. Zhu, X. Zhu *et al.*, "On optimizing traffic scheduling for multi-replica containerized microservices," in *Proceedings of the 52nd International Conference on Parallel Processing (ICPP)*, 2023, pp. 358–368.

[34] A. Marchese and O. Tomarchio, "Network-aware container placement in cloud-edge kubernetes clusters," in *Proceedings of the 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 859–865.

[35] Marchese, Angelo and Tomarchio, Orazio, "Extending the kubernetes platform with network-aware scheduling capabilities," in *Proceedings of the 20th International Conference on Service-Oriented Computing (ICSOC 2022)*. Berlin: Springer-Verlag, 2022, p. 465480.

[36] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/, 2010, accessed: 2025-12-22.

[37] OpenTelemetry, "Opentelemetry," https://opentelemetry.io/, 2025, accessed: 2025-12-21.

[38] P. A. open-source technology designed to provide monitoring and alerting functionality, https://prometheus.io/, 2024, accessed: 2024-04-20.

[39] J. Tracing, "Jaeger: open source, end-to-end distributed tracing," https://www.jaegertracing.io/, 2025, accessed: 2025-12-21.

[40] Zipkin, "Zipkin distributed tracing system," https://zipkin.io/, 2025, accessed: 2025-12-21.

[41] G. Labs, "Grafana," https://grafana.com/oss/grafana/, 2025, accessed: 2025-12-21.

[42] Istio: An open source service mesh, https://istio.io/, 2024, accessed: 2024-05-25.

[43] Envoy: An Open Source Edge and Service Proxy, Designed for Cloud Native Apps., https://www.envoyproxy.io/, 2024, accessed: 2024-05-25.

[44] Linkerd, "Linkerd service mesh," https://linkerd.io/, 2025, accessed: 2025-12-21.

[45] J. Bi, L. Zhang, H. Yuan, and M. Zhou, "Hybrid task prediction based on wavelet decomposition and arima model in cloud data center," in *2018 IEEE 15th International Conference on Networking, Sensing and Control (ICNSC)*, 2018, pp. 1–6.

[46] Y. Wang, C. Wang, C. Shi, and B. Xiao, "Short-term cloud coverage prediction using the arima time series model," *REMOTE SENSING LETTERS*, vol. 9, no. 3, pp. 274–283, 2018.

[47] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*, 2011, pp. 500–507.

[48] J. Kumar and A. K. Singh, "Workload prediction in cloud using artificial neural network and adaptive differential evolution," *Future Generation Computer Systems*, vol. 81, pp. 41–52, 2018.

[49] J. Kumar, R. Goomer, and A. K. Singh, "Long short term memory recurrent neural network (lstm-rnn) based workload forecasting model for cloud datacenters," *Procedia Computer Science*, vol. 125, pp. 676–682, 2018, the 6th International Conference on Smart Computing and Communications.

[50] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *arXiv:1803.01271*, 2018.

[51] H. Wu, J. Xu, J. Wang, and M. Long, "Autoformer: Decomposition transformers with Auto-Correlation for long-term series forecasting," in *Advances in Neural Information Processing Systems (NeurIPS '21)*, 2021.

[52] Y. Zeng, Z. Qu, S. Guo, B. Ye, J. Zhang, J. Li, and B. Tang, "Safedrl: Dynamic microservice provisioning with reliability and latency guarantees in edge environments," *IEEE Transactions on Computers*, vol. 73, no. 1, pp. 235–248, 2024.

[53] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *Proceedings of the IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–9.

[54] J. Wu, M. Xu, Y. He, K. Ye, and C. Xu, "Cloudnativesim: A toolkit for modeling and simulation of cloud-native applications," *Software: Practice and Experience*, vol. 55, no. 7, pp. 1185–1208, 2025.

[55] M. Straesser, P. Haas, S. Frank, A. Hakamian, A. van Hoorn, and S. Kounev, "Kubernetes-in-the-loop: Enriching microservice simulation through authentic container orchestration," in *Performance Evaluation Methodologies and Tools*, E. Kalyvianaki and M. Paolieri, Eds. Cham: Springer Nature Switzerland, 2024, pp. 82–98.

[56] P. Gouveia, J. a. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos, "Kollaps: Decentralized and dynamic topology emulation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[57] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[58] Yoav Einav, "Amazon found every 100ms of latency cost them 1% in sales, 2019," 2019, [Online; accessed 18-September-2022]. [Online]. Available: https://www.gigaspaces.com/blog/amazon-foundevery-100ms-of-latency-cost-them-1-in-sales/

[59] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, p. 248259.

[60] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, p. 607618.

[61] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized

environments," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. USA: USENIX Association, 2013, p. 219230.

[62] L. Pons, J. Feliu, J. Sahuquillo, M. E. Gómez, S. Petit, J. Pons, and C. Huang, "Cloud white: Detecting and estimating qos degradation of latency-critical workloads in the public cloud," *Future Gener. Comput. Syst.*, vol. 138, no. C, p. 1325, jan 2023.

[63] Karpenter, "Karpenter," https://karpenter.sh/, 2025, accessed: 2025-12-21.

[64] Y. Hu, H. Wang, L. Wang, M. Hu, K. Peng, and B. Veeravalli, "Joint deployment and request routing for microservice call graphs in data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 11, pp. 2994–3011, 2023.

[65] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: A fault-tolerant resource management and job scheduling system at internet scale," *Proc. VLDB Endow.*, vol. 7, no. 13, p. 13931404, aug 2014.

[66] J. Santos, C. Wang, T. Wauters, and F. D. Turck, "Diktyo: Network-aware scheduling in container-based clouds," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2023.

[67] M. Chen, M. T. Islam, M. R. Read, and R. Buyya, " TraDE: Network and Traffic-Aware Adaptive Scheduling for Microservices Under Dynamics ," *IEEE Transactions on Parallel & Distributed Systems*, vol. 37, no. 01, pp. 76–89, Jan. 2026.

[68] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018.

[69] D. Huye, Y. Shkuro, and R. R. Sambasivan, "Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 419–432.

[70] Ming Chen and Muhammed Tawfiqul Islam and Maria Rodriguez Read and Rajkumar Buyya, "iDynamics: A Configurable Emulation Framework for

Evaluating Microservice Scheduling Policies under Controllable Cloud–Edge Dynamics," 2025. [Online]. Available: https://arxiv.org/abs/2503.16029

[71] K. Senjab, S. Abbas, N. Ahmed, and A. U. R. Khan, "A survey of kubernetes scheduling algorithms," *Journal of Cloud Computing*, vol. 12, no. 1, p. 87, 2023.

[72] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, vol. 55, no. 7, Dec. 2022.

[73] Z. Rejiba and J. Chamanara, "Custom scheduling in kubernetes: A survey on common problems and solution approaches," *ACM Comput. Surv.*, vol. 55, no. 7, dec 2022.

[74] L. M. Al Qassem, T. Stouraitis, E. Damiani, and I. M. Elfadel, "Containerized microservices: A survey of resource management frameworks," *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 3775–3796, 2024.

[75] T. Khan, W. Tian, G. Zhou, S. Ilager, M. Gong, and R. Buyya, "Machine learning (ml)-centric resource management in cloud computing: A review and future directions," *Journal of Network and Computer Applications*, vol. 204, p. 103405, 2022.

[76] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "Machine learning-based orchestration of containers: A taxonomy and future directions," *ACM Comput. Surv.*, vol. 54, no. 10s, sep 2022.

[77] S. K. Moghaddam, R. Buyya, and K. Ramamohanarao, "Performance-aware management of cloud resources: A taxonomy and future directions," *ACM Comput. Surv.*, vol. 52, no. 4, aug 2019.

[78] J. Yang, Z. Saad, J. Wu, X. Niu, H. Leung, and S. Drew, "A survey on task scheduling in carbon-aware container orchestration," *arXiv preprint arXiv:2508.05949*, 2025.

[79] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "A review of serverless use cases and their characteristics," *arXiv preprint arXiv:2008.11110*, 2020.

[80] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, p. 7684, apr 2021.

[81] S. Shi, Y. Yu, M. Xie, X. Li, X. Li, Y. Zhang, and C. Qian, "Concury: A fast and light-weight software cloud load balancer," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: ACM, 2020, p. 179192.

[82] K. Peng, Y. Hu, H. Ding, H. Chen, L. Wang, C. Cai, and M. Hu, "Large-scale service mesh orchestration with probabilistic routing in cloud data centers," *IEEE Transactions on Services Computing*, 2025.

[83] L. Liechti, P. Gouveia, J. Neves, P. Kropf, M. Matos, and V. Schiavoni, "Thunderstorm: A tool to evaluate dynamic network topologies on distributed systems," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019, pp. 241–24 109.

[84] M. Goudarzi, M. Palaniswami, and R. Buyya, "Scheduling iot applications in edge and fog computing environments: A taxonomy and future directions," *ACM Comput. Surv.*, vol. 55, no. 7, Dec. 2022.

[85] R. Eidenbenz, Y.-A. Pignolet, and A. Ryser, "Latency-aware industrial fog application orchestration with kubernetes," in *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, 2020, pp. 164–171.

[86] Y. Hu, H. Ding, H. Chen, J. He, M. Hu, C. Cai, and K. Peng, "Collaborative orchestration with probabilistic routing for dynamic service mesh in clouds," in *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*. IEEE, 2025, pp. 1–10.

[87] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018, p. 347360.

[88] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the wild: Workload analysis and scheduling in Large-Scale

heterogeneous GPU clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 945–960.

[89] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications qos," *IEEE Transactions on Cloud Computing*, vol. 3, no. 4, pp. 449–458, 2015.

[90] C. Nguyen, C. Klein, and E. Elmroth, "Multivariate lstm-based location-aware workload prediction for edge data centers," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 341–350.

[91] Y. Chen, Y. Kang, Y. Chen, and Z. Wang, "Probabilistic forecasting with temporal convolutional neural network," *Neurocomputing*, vol. 399, pp. 491–501, 2020.

[92] M. Xu, C. Song, H. Wu, S. S. Gill, K. Ye, and C. Xu, "Esdnn: Deep neural network based multivariate workload prediction in cloud computing environments," *ACM Trans. Internet Technol.*, vol. 22, no. 3, aug 2022.

[93] Z. Wu, S. Pan, G. Long, J. Jiang, X. Chang, and C. Zhang, "Connecting the dots: Multivariate time series forecasting with graph neural networks," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '20. New York, NY, USA: ACM, 2020, p. 753763.

[94] M. Chen, Y. Li, M. Lin, H. Wang, and A. Y. Zomaya, "EN-Beats: A novel event-driven n-beats for cloud workload prediction," in *Proceedings of the 24th International Middleware Conference*, ser. Middleware '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 103–116. [Online]. Available: https://doi.org/10.1145/3583780.3614832

[95] J. Ruuskanen, T. Berner, K.-E. Arzen, and A. Cervin, "Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing," *SIGMETRICS Perform. Eval. Rev.*, vol. 49, no. 3, p. 6970, mar 2022.

[96] F. Du, J. Shi, Q. Chen, P. Pang, L. Li, and M. Guo, "Generating microservice graphs with production characteristics for efficient resource scaling," in *Proceedings of the 39th ACM International Conference on Supercomputing*, ser. ICS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 895910.

[97] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, p. 1933.

[98] Y. Sun, B. Shi, M. Mao, M. Ma, S. Xia, S. Zhang, and D. Pei, "Art: A unified unsupervised framework for incident management in microservice systems," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, October 2024.

[99] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang, "Perfscope: Practical online server performance bug inference in production cloud computing infrastructures," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, p. 113.

[100] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, p. 115126.

[101] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized "zero-queue" datacenter network," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, p. 307318.

[102] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, p. 127138.

[103] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, p. 379391.

[104] S. Chen, Y. Jiang, C. Delimitrou, and J. F. Martnez, "Pimcloud: Qos-aware resource management of latency-critical applications in clouds with processing-in-memory," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, April 2022, pp. 1086–1099.

[105] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, p. 107120.

[106] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 193–206.

[107] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim, "EyeQ: Practical network performance isolation at the edge," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 297–311.

[108] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, p. 607618, jun 2013.

[109] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microservices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: ACM, 2019.

[110] L. Chen, S. Luo, C. Lin, Z. Mo, H. Xu, K. Ye, and C. Xu, "Derm: Sla-aware resource

management for highly dynamic microservices," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 424–436.

[111] D. Saxena and A. K. Singh, "A proactive autoscaling and energy-efficient vm allocation framework using online multi-resource neural network for cloud data center," *Neurocomputing*, vol. 426, pp. 248–264, 2021.

[112] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell, "Hipster: Hybrid task manager for latency-critical cloud workloads," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 409–420.

[113] wrk2: An HTTP benchmarking tool based on wrk, https://github.com/giltene/wrk2, 2024, accessed: 2024-05-25.

[114] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 450–462.

[115] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan, "Prepare: Predictive performance anomaly prevention for virtualized cloud systems," in *2012 IEEE 32nd International Conference on Distributed Computing Systems (ICDCS '12)*, June 2012, pp. 285–294.

[116] I. K. Kim, W. Wang, Y. Qi, and M. Humphrey, "Forecasting cloud application workloads with cloudinsight for predictive resource management," *IEEE Transactions on Cloud Computing*, vol. 10, no. 3, pp. 1848–1863, 2022.

[117] J. Kumar and A. K. Singh, "Workload prediction in cloud using artificial neural network and adaptive differential evolution," *Future Generation Computer Systems*, vol. 81, pp. 41–52, 2018.

[118] A. Aznavouridis, K. Tsakos, and E. G. Petrakis, "Micro-service placement policies for cost optimization in kubernetes," in *International Conference on Advanced Information Networking and Applications*. Springer, 2022, pp. 409–420.

[119] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th International Conference on Architectural*

Support for Programming Languages and Operating Systems, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, p. 127144.

[120] C. Delimitrou, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, p. 7788.

[121] J. Liu, S. Zhang, and Q. Wang, "Conadapter: Reinforcement learning-based fast concurrency adaptation for microservices in cloud," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC '23. New York, NY, USA: ACM, 2023, p. 427442.

[122] M. Xu, C. Song, S. Ilager, S. S. Gill, J. Zhao, K. Ye, and C. Xu, "CoScal: Multifaceted scaling of microservices with reinforcement learning," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 3995–4009, 2022.

[123] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: ACM, 2021, p. 412426.

[124] F. Zhang, Y. Chen, H. Lu, and Y. Huang, "Network-aware reliability modeling and optimization for microservice placement," *IEEE Transactions on Network and Service Management*, vol. 22, no. 4, pp. 3705–3720, 2025.

[125] M. Xu, Q. Zhou, H. Wu, W. Lin, K. Ye, and C. Xu, "Pdma: Probabilistic service migration approach for delay-aware and mobility-aware mobile edge computing," *Software: Practice and Experience*, vol. 52, no. 2, pp. 394–414, 2022.

[126] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, no. 3, pp. 939–951, 2021.

[127] S. Pallewatta, V. Kostakos, and R. Buyya, "Reliability-aware proactive placement

of microservices-based iot applications in fog computing environments," *IEEE Transactions on Mobile Computing*, pp. 1–16, 2024.

[128] ——, "Placement of microservices-based iot applications in fog computing: A taxonomy and future directions," *ACM Computing Surveys*, vol. 55, no. 14s, pp. 1–43, 2023.

[129] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 598–610.

[130] D. Cheng, X. Zhou, Z. Ding, Y. Wang, and M. Ji, "Heterogeneity aware workload management in distributed sustainable datacenters," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 30, no. 2, pp. 375–387, Feb 2019.

[131] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 1–18.

[132] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online job scheduling in distributed machine learning clusters," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. IEEE Press, 2018, p. 495503.

[133] Z. Bian, S. Li, W. Wang, and Y. You, "Online evolutionary batch size orchestration for scheduling deep learning workloads in gpu clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: ACM, 2021.

[134] L. Cao and P. Sharma, "Co-locating containerized workloads using service mesh telemetry," in *Proceedings of the 17th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2021, pp. 168–181.

[135] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environ-

ments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, p. 2350, Jan. 2011.

[136] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, p. 153167.

[137] L. Biggio, T. Bendinelli, C. Kulkarni, and O. Fink, "Ageing-aware battery discharge prediction with deep learning," *Applied Energy*, vol. 346, p. 121229, 2023.

[138] H. Tian, Y. Zheng, and W. Wang, "Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, USA: ACM, 2019, p. 139151.

[139] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 485–497.

[140] Istio: Performance and Scalability, https://istio.io/latest/docs/ops/deployment/performance-and-scalability/, 2024, accessed: 2024-05-25.

[141] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 193–206.

[142] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang, "PerfIso: Performance isolation for commercial Latency-Sensitive services," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 519–532.

[143] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[144] J. Zhao, H. Cui, J. Xue, and X. Feng, "Predicting cross-core performance interference on multicore processors with regression analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, p. 14431456, may 2016.

[145] C. Jiang, Y. Qiu, W. Shi, Z. Ge, J. Wang, S. Chen, C. Crin, Z. Ren, G. Xu, and J. Lin, "Characterizing co-located workloads in alibaba cloud datacenters," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2381–2397, 2022.

[146] A. Radovanovi, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, S. Talukdar, E. Mullen, K. Smith, M. Cottman, and W. Cirne, "Carbon-aware computing for datacenters," *IEEE Transactions on Power Systems*, vol. 38, no. 2, pp. 1270–1280, 2023.

[147] T. B. Hewage, S. Ilager, M. Rodriguez, and R. Buyya, "Carbon-aware resource management in latency-sensitive cloud computing environments: A taxonomy and review," *ACM Computing Surveys*, 2025 (Under Review).

[148] D. Ding, X. Fan, Y. Zhao, K. Kang, Q. Yin, and J. Zeng, "Q-learning based dynamic task scheduling for energy-efficient cloud computing," *Future Generation Computer Systems*, vol. 108, pp. 361–371, 2020.

[149] R. Bhagwan, R. Kumar, C. S. Maddila, and A. A. Philip, "Orca: Differential bug localization in Large-Scale services," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 493–509.

[150] W. Chen, C. Lu, K. Ye, Y. Wang, and C.-Z. Xu, "Rptcn: Resource prediction for high-dynamic workloads in clouds based on deep learning," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 59–69.

[151] F. Farahnakian, P. Liljeberg, and J. Plosila, "Lircup: Linear regression based cpu usage prediction algorithm for live migration of virtual machines in data centers," in *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, 2013, pp. 357–364.

[152] J. Gao, H. Wang, and H. Shen, "Machine learning based workload prediction in cloud computing," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, 2020, pp. 1–9.

[153] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *arXiv preprint arXiv:1803.01271*, 2018.

[154] J. Bi, H. Yuan, and M. Zhou, "Temporal prediction of multiapplication consolidated workloads in distributed clouds," *IEEE Transactions on Automation Science and Engineering*, vol. 16, no. 4, pp. 1763–1773, 2019.

[155] N. Doulamis, A. Doulamis, A. Litke, A. Panagakis, T. Varvarigou, and E. Varvarigos, "Adjusted fair scheduling and non-linear workload prediction for qos guarantees in grid computing," *Computer Communications*, vol. 30, no. 3, pp. 499–515, 2007, special Issue: Emerging Middleware for Next Generation Networks.

[156] G. K. Shyam and S. S. Manvi, "Virtual resource prediction in cloud environment: A bayesian approach," *Journal of Network and Computer Applications*, vol. 65, pp. 144–154, 2016.

[157] F. Qiu, B. Zhang, and J. Guo, "A deep learning approach for vm workload prediction in the cloud," in *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2016, pp. 319–324.

[158] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium*

on Operating Systems Principles, ser. SOSP '17. New York, NY, USA: ACM, 2017, p. 153167.

[159] J. Bi, S. Li, H. Yuan, and M. Zhou, "Integrated deep learning method for workload and resource prediction in cloud systems," *Neurocomputing*, vol. 424, pp. 35–48, 2021.

[160] M. E. Karim, M. M. S. Maswood, S. Das, and A. G. Alharbi, "Bhyprec: A novel bi-lstm based hybrid recurrent neural network model to predict the cpu workload of cloud virtual machine," *IEEE Access*, vol. 9, pp. 131 476–131 495, 2021.

[161] D. Janardhanan and E. Barrett, "Cpu workload forecasting of machines in data centers using lstm recurrent neural networks and arima models," in *Proceedings of the 2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2017, pp. 55–60.

[162] K. Cetinski and M. B. Juric, "Ame-wpc: Advanced model for efficient workload prediction in the cloud," *Journal of Network and Computer Applications*, vol. 55, pp. 191–201, 2015.

[163] A. A. Bankole and S. A. Ajila, "Cloud client prediction models for cloud resource provisioning in a multitier web application environment," in *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, 2013, pp. 156–161.

[164] A. K. Singh, D. Saxena, J. Kumar, and V. Gupta, "A quantum approach towards the adaptive prediction of cloud workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 12, pp. 2893–2905, 2021.

[165] B. N. Oreshkin, D. Carpov, N. Chapados, and Y. Bengio, "Meta-learning framework with applications to zero-shot time-series forecasting," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 10, 2021, pp. 9242–9250.

[166] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the "micro" back in microservice," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 645–650.

[167] S. Schmid, C. Avin, C. Scheideler, M. Borokhovich, B. Haeupler, and Z. Lotker, "Splaynet: towards locally self-adjusting networks," *IEEE/ACM Trans. Netw.*, vol. 24, no. 3, p. 14211433, Jun. 2016.

[168] L. Leonini, É. Rivière, and P. Felber, "SPLAY: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)," in *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*. Boston, MA: USENIX Association, Apr. 2009.

[169] CNCF Community Infrastructure Lab (CIL): An on-demand infrastructure resource for CNCF developers., https://istio.io/latest/docs/ops/deployment/performance-and-scalability/, 2024, accessed: 2024-05-25.