PANORAMA: Unveiling Latent Dependencies for Microservice Autoscaling via Meta-learning

Zhuangbin Chen
Sun Yat-sen University
Zhuhai, China
chenzhb36@mail.sysu.edu.cn

Hongjiang Feng
Sun Yat-sen University
Zhuhai, China
fenghj5@mail2.sysu.edu.cn

Yang Liu
Sun Yat-sen University
Zhuhai, China
liuy2355@mail2.sysu.edu.cn

Juzheng Zheng
Sun Yat-sen University
Zhuhai, China
zhengjzh8@mail2.sysu.edu.cn

Xiaoyu He*
Sun Yat-sen University
Zhuhai, China
hexy73@mail.sysu.edu.cn

Zibin Zheng
Sun Yat-sen University
Zhuhai, China
zhzibin@mail.sysu.edu.cn

Abstract

Existing autoscaling approaches primarily leverage *direct service calls* to model the propagation effects of scaling decisions across inter-service dependencies. However, the intricate and implicit service interactions (i.e., *latent dependencies*) often go beyond such direct invocations, rendering suboptimal scaling actions and poor adaptability. To address these limitations, we propose PANORAMA, a novel autoscaling framework that explicitly unveils and exploits indirect service interactions. In particular, we formalize two fundamental categories of latent dependencies, i.e., *resource interference dependencies* (RIDs), which emerge from shared infrastructure resources, and *execution blocking dependencies* (EBDs), which stem from temporal constraints in the execution logic of business workflows. PANORAMA employs a hybrid attention mechanism to model these multifaceted service dependencies alongside temporal dynamics. We also design a meta-learning paradigm to enable rapid adaptation to evolving workload patterns and dependency structures without extensive retraining. Evaluation across four representative microservice benchmarks shows that PANORAMA consistently outperforms state-of-the-art baselines, achieving up to 68% reduction in resource consumption while improving response time by 42%. This highlights the critical importance of explicitly modeling latent dependencies for effective microservice resource management.

CCS Concepts

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Software performance**.

Keywords

Microservices, Resource Management, Autoscaling, QoS

ACM Reference Format:
Zhuangbin Chen, Hongjiang Feng, Yang Liu, Juzheng Zheng, Xiaoyu He, and Zibin Zheng. 2026. PANORAMA: Unveiling Latent Dependencies for Microservice Autoscaling via Meta-learning. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837485

*Xiaoyu He is the corresponding author.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837485

1 Introduction

Microservice architectures enable independent scaling of individual services [6, 9, 37, 54]. However, this fine-grained scalability introduces significant complexity when managing resources in production environments. Rapid resource autoscaling therefore becomes essential for microservice applications to accommodate workload fluctuations. State-of-the-art methods [32, 39, 40, 57] leverage deep learning to capture intricate and evolving workload patterns. A key strategy used involves modeling spatial and temporal features between microservices through service dependency graphs. They recognize that effective autoscaling requires understanding both inter-service interactions across the system topology (spatial dimension) and their evolution over time (temporal dimension). By analyzing these dual aspects, existing methods can track workload propagation patterns through dependencies and preemptively mitigate bottlenecks before they degrade end-user performance.

Although *direct invocations* are the most representative type of service dependency, intricate inter-service relationships go considerably beyond explicit call graphs [43, 49, 50, 55]. Existing methods that rely on direct invocations to model the cascading effects of their scaling decisions may yield an incomplete and inaccurate view of system-wide dynamics. Recent work, e.g., DeepScaler [32], attempts to address this gap by adaptively learning an "affinity matrix" to uncover hidden statistical correlations among services. However, microservices influence each other through multiple distinct mechanisms, from infrastructure-level resource sharing to application-level workflow coordination. DeepScaler's expectation-maximization-based learning, seeded by the direct call graph, reduces these interactions to a single undifferentiated affinity matrix optimized for prediction error rather than dependency discovery. Thus, it cannot distinguish dependencies rooted in different system semantics and may miss interactions that do not manifest as statistical co-variation in service-level metrics. Our investigation (Sec. 4.2.3) confirms that DeepScaler fails to capture many critical latent dependencies that arise from these distinct mechanisms.

To overcome these limitations, we propose to explicitly model the latent dependencies, which aims to reveal service interaction

patterns beyond direct call graphs and provide more accurate dependency representations. Our framework systematically categorizes and models two distinct types of latent dependencies, each grounded in a specific mechanism of indirect service interaction.

- *Resource Interference Dependencies (RIDs):* At the *infrastructure layer*, RIDs emerge when microservices deployed on the same physical or virtual host compete for shared resources (e.g., CPU and memory), creating performance interdependencies that transcend direct invocations. While some approaches [13, 58] use system metrics to infer such interactions, they often suffer from high modeling complexity and imprecise results.
- *Execution Blocking Dependencies (EBDs):* At the *application layer*, EBDs occur when a microservice is delayed or impeded due to its reliance on another service to complete a prerequisite task within the business workflow. For example, a checkout service must complete payment before sending a purchase email (Sec. 2.2.3). Crucially, these blocking dependencies may exist even without direct calls, allowing bottlenecks in one service to delay others that depend on its output or state change.

While resource contention among co-located services and workflow-level execution ordering are individually recognized phenomena in cloud and distributed systems, existing autoscaling approaches have not systematically formalized them as explicit, mineable dependency types. We address this gap by directly incorporating these dependencies into the autoscaling framework and designing mechanism-specific algorithms to mine them.

Based on these latent dependencies, we design Panorama, a microservice autoscaling framework for more effective and adaptive resource management. Panorama introduces a hybrid attention mechanism that decouples attention computation into specialized heads focusing on distinct dependency types alongside temporal dynamics, capturing multifaceted service relationships. Moreover, we design a meta-learning paradigm for rapid adaptation to evolving workload patterns and dependency structures, avoiding the overhead of extensive retraining. This is achieved by identifying distinct adaptation tasks and incorporating spatial positional encodings and real-time dependency information as contextual elements. Through this integrated approach, Panorama enables coordinated and unified resource adjustments across the entire microservice application while balancing resource efficiency and service quality. Our evaluation shows that Panorama consistently outperforms state-of-the-art baselines, achieving up to 68% reduction in resource consumption while improving response time by 42%.

To sum up, this work makes the following major contributions:

- We identify and formalize two critical types of latent dependencies that remain largely unaddressed, i.e., RIDs and EBDs. They capture the complete spectrum of service interactions that influence resource requirements and performance propagation.
- We present Panorama, which employs a hybrid attention mechanism with a meta-learning strategy for accurate and adaptive autoscaling. By integrating latent dependencies with direct service invocations, Panorama delivers coordinated autoscaling decisions that balance service quality and resource efficiency, while adapting effectively to dynamic service environments.
- We implement Panorama on K8s and evaluate it across various benchmarks. The results demonstrate that Panorama achieves significant improvements over existing methods. Further studies confirm the benefits of latent dependencies toward accurate service autoscaling and Panorama's ability to discover them.

2 Background and Problem Formulation

2.1 Microservice Autoscaling

Autoscaling dynamically allocates compute resources to maintain service performance under changing workloads. It operates as a closed-loop control system via container orchestration platforms like Kubernetes (K8s). Autoscaling can be performed through *horizontal scaling*, which adjusts the number of service instances (replicas), or *vertical scaling*, which modifies the CPU/memory allocation of existing instances. Microservice complexity greatly impacts autoscaling effectiveness, presenting two key challenges.

- **Multifaceted service dependencies:** The multifaceted nature of service dependencies, which go beyond simple direct invocations, creates complex and often non-obvious performance relationships. This makes it difficult to anticipate how scaling one service might propagate and impact others. Thus, modeling such relationships requires capturing context-sensitive interactions while accounting for intricate service-state variations.
- **Dynamic environments:** The workload patterns and microservice characteristics are constantly evolving due to user behavior changes, code deployments, and configuration modifications. Achieving an optimal balance between QoS guarantees and resource efficiency requires sophisticated modeling and decision-making capabilities that account for both immediate performance needs and future demand forecasts [5, 6].

2.2 Heterogeneous Microservice Dependencies

Direct dependencies characterize explicit service interactions but represent only one facet of complex service relationships. We investigate other critical latent dependencies that significantly impact service performance and resource efficiency, but remain unaddressed. We illustrate different dependencies using an e-commerce application (partial) in Fig. 1, where the white and gray traces represent the *product recommendation* and *checkout* workflow, respectively.

2.2.1 Direct Invocation Dependencies (DIDs). DIDs represent explicit call relationships between microservices, which are relatively straightforward to identify through distributed tracing systems (e.g., Jaeger, Zipkin). The discrepancies in processing throughput or resource allocation among directly connected services may create bottlenecks that amplify latency and degrade end-to-end performance. Thus, many autoscaling approaches [28, 32, 48] incorporate these direct dependencies into their decision-making process, recognizing that effective resource provisioning must consider the entire request execution path rather than local service metrics.

2.2.2 Resource Interference Dependencies (RIDs). RIDs represent implicit performance interference between co-located services that compete for shared hardware resources, even without direct communication [24, 39, 50]. Although container mechanisms such as cgroups, namespaces, and CPU/memory limits provide OS-level isolation, they cannot fully eliminate contention for shared resources such as last-level cache, memory bandwidth, and node-level I/O [24, 50]. Even finer-grained controls (e.g., CPU pinning or cache

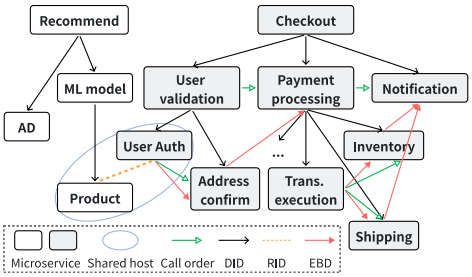

Figure 1: Heterogeneous Microservice Dependencies

partitioning) cannot fully prevent cross-core interference on shared hardware [13, 58]. RIDs are especially common in cloud because K8s placement mechanisms such as node affinity and anti-affinity are often soft controls rather than hard guarantees. Scaling operations, node failures, or resource pressure may force unrelated services to share the same node, creating "noisy neighbor" effects [39, 50] that can degrade performance and mislead autoscaling decisions.

2.2.3 Execution Blocking Dependencies (EBDs). EBDs represent implicit performance constraints in hierarchical execution flows and fixed call sequences of microservice workflows. Unlike DIDs, EBDs occur when downstream services are throttled because earlier services in the request lifecycle must complete before the workflow can proceed, even without direct invocations. For the checkout process in Fig. 1, the *checkout* service orchestrates a sequential workflow wherein *user validation* must precede *payment processing*. The *user validation* phase itself involves *user authentication* followed by *address confirmation*. If *address confirmation* experiences bottlenecks, the entire downstream execution chain (e.g., *payment processing*) is blocked regardless of resource availability. A similar blocking dependency exists between the *payment processing* and *notification* service (Sec. 3.3.3). Conventional methods fail to detect such hidden bottlenecks explicitly, as they only analyze call graphs, not end-to-end execution semantics. We formalize these temporal constraints by modeling all potential blocking relationships, revealing how resource shortages in seemingly unrelated services can cascade through the system via execution ordering requirements.

These execution blocking patterns stem from common microservice design choices [22, 30, 38], including synchronous communications (e.g., HTTP, RPC), strict workflow prerequisite tasks, and distributed transaction coordination. Additionally, contention for finite logical resources (e.g., database connection pools) or strict ordering requirements in event-driven pipelines frequently forces downstream services into temporary waiting states.

2.3 Problem Formulation

We model a microservice application as a directed graph of multiple services, denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, where $\mathcal{V} = \{v_1, v_2, \ldots, v_N\}$ is a finite set of nodes representing the services with $|\mathcal{V}| = N$ and $\mathcal{A} = \{\mathcal{A}^{(did)}, \mathcal{A}^{(rid)}, \mathcal{A}^{(ebd)}\}$ is a collection of adjacency matrices. Each matrix $\mathcal{A}^{(*)} \in \{0,1\}^{N\times N}$ represents a distinct dependency type (i.e., DID, RID, or EBD) between services and $A^{(*)}_{ij} = 1$ if a dependency exists between v_i and v_j. At each time step t, we collect a set of L performance metrics for each service $v_i \in \mathcal{V}$ (e.g., CPU utilization, instance count, request rate). These metrics are derived from per-instance measurements, aggregated across all running instances of a service (e.g., summed for request count).

Let $x^t_{k,i} \in \mathbb{R}$ denote the value of the k-th metric for service v_i at time t, where $k \in \{1, \ldots, L\}$. The vector of metrics for service v_i at time t is $x^t_i = (x^t_{1,i}, x^t_{2,i}, \ldots, x^t_{L,i})^T \in \mathbb{R}^L$. The state of the entire application at time t is represented by the collection of metrics for all services, denoted by $\mathcal{X}_t = (x^t_1, x^t_2, \ldots, x^t_N)^T \in \mathbb{R}^{N\times L}$. To capture the temporal dynamics and provide context for prediction, we consider a historical window of τ time steps. The input data available at time t for predicting the future resource requirements is the sequence of historical metric observations $\mathcal{X}_{t-\tau+1:t} = (\mathcal{X}_{t-\tau+1}, \ldots, \mathcal{X}_{t-1}, \mathcal{X}_t)$, which can be viewed as a tensor in $\mathbb{R}^{N\times L\times\tau}$.

The core problem is to determine the appropriate number of instance replicas for each service $v_i \in \mathcal{V}$ at the next time step $t+1$. This demand allocation relies on historical metric data $\mathcal{X}_{t-\tau+1:t}$ and multifaceted dependencies $\mathcal{A}$. Let $y^{t+1}_i \in \mathbb{R}_{\geq 0}$ be the predicted resource demand for service v_i at time $t+1$. The output vector for all services is denoted by $\mathcal{Y}^{t+1} = (y^{t+1}_1, y^{t+1}_2, \ldots, y^{t+1}_N)^T \in \mathbb{R}^N_{\geq 0}$. The formulation operates at the service level where individual instances do not evolve independently (i.e., horizontal scaling), which is a common setting in existing studies [4, 32, 48, 52]. The objective is to determine $\mathcal{Y}^{t+1}$ to simultaneously address two competing goals:

- *QoS Assurance:* Guarantee that the application meets its Service Level Agreements (SLAs), primarily concerning the end-to-end response time. This implies allocating enough instances to handle workload surges and maintain low latency.
- *Resource Utilization Optimization:* Achieve high utilization of the allocated computational resources across the microservice application. This aims to minimize resource waste and operational costs by avoiding over-provisioning.

Although both scaling dimensions can address this problem, we focus on horizontal scaling for practical considerations. First, vertical scaling in K8s often lacks support for "in-place" resource updates [3, 26], as modifying CPU/memory limits typically requires a container restart, disrupting request processing. Although K8s v1.35 recently graduated in-place Pod resource resize to GA [17], it still has known limitations such as restricted support for certain resource policies and potential race conditions between the kubelet and scheduler. Second, vertical scaling introduces scheduling challenges [41], as increasing resource limits may cause instances to no longer fit on their node, triggering rescheduling and cascading evictions. More importantly, our core contribution is explicitly modeling heterogeneous service dependencies to decide *which* services to scale and *by how much*, which is independent of *how* the scaling is executed. The dependency-aware model forecasts per-service resource demand, which can be fulfilled either by adding replicas (horizontal) or by increasing per-instance allocations (vertical). We adopt horizontal scaling as a mature and operationally stable platform for validating our dependency modeling contribution.

3 Methodology

3.1 Overview

In this section, we present Panorama for microservice autoscaling. It follows the well-established MAPE (Monitor-Analyze-Plan-Execute) pipeline commonly used in existing approaches [32]. Fig. 2

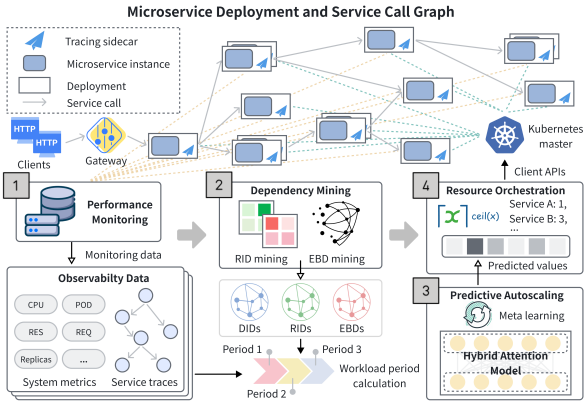

Figure 2: Overview of the Panorama System Architecture

illustrates this architecture: *performance monitoring* (Monitor) periodically collects workload-related metrics and microservice traces; *dependency mining* (Analyze) uncovers fine-grained heterogeneous service dependencies from the collected data; *predictive autoscaling* (Plan) forecasts per-service resource demands informed by these dependencies; and *resource orchestration* (Execute) converts the predictions into scaling actions to optimize resource allocation and service performance. These four phases form a closed control loop: each execution changes the system state, which the next monitoring cycle observes, enabling Panorama to continuously adapt to evolving workloads and dependencies.

3.2 Performance Monitoring

System metrics provide real-time visibility into service performance, workload patterns, and resource utilization, which are essential for autoscaling decisions. However, fine-grained service-level monitoring presents challenges in practice, as cloud platforms may lack native support. Traditional application-level instrumentation typically demands specialized developer expertise and can lead to intrusive code modifications or tight coupling with core microservice logic.

To enable transparent data collection, we deploy Envoy sidecar proxies into each microservice pod via the Istio service mesh, creating a dedicated data plane for observability. A *monitoring* component retrieves metrics (e.g., CPU utilization, memory usage, request latency, and request rate) from the sidecar proxies through Prometheus-compatible endpoints. A *tracing* component leverages Jaeger (which is seamlessly integrated with Istio's data plane) to collect traces by intercepting inter-service communications at the proxy level. Both metrics and distributed traces are then forwarded to a database backend for storage and subsequent processing. Crucially, this transparency means that data collection requires no code changes or developer involvement. All telemetry is gathered automatically by Envoy sidecars. Panorama passively observes production traffic rather than prescribing workloads, so it naturally captures runtime variations such as request-rate fluctuations, shifting call patterns, and sudden traffic spikes.

3.3 Dependency Mining

Using the collected telemetry, this phase constructs a holistic service interaction graph with different types of dependencies, i.e., DIDs, RIDs, and EBDs. Given the intricate and evolving nature of service workloads, these dependencies are mined in real time to ensure that accurate and timely dependencies are available for autoscaling.

3.3.1 Direct Invocation Dependencies. DIDs are derived by parsing distributed traces from monitoring phase to reconstruct call graphs. Each trace, representing an end-to-end request flow [10, 56], is composed of spans annotated with service identifiers, timestamps, etc. These spans record the operations of individual services throughout the lifecycle of a request. A DID is established from a parent service to a child service if a parent span initiates a call to a child span.

3.3.2 Resource Interference Dependencies. Identifying RIDs requires mapping logical services to their physical deployment topology to uncover potential resource contention among co-located instances. Based on K8s, we establish this topology through a two-step extraction process. First, we construct *service-to-instance* mappings (e.g., $UserService \rightarrow [Pod\text{-}A, Pod\text{-}B, Pod\text{-}C]$) by querying label selectors from *Deployment* and *Service* metadata. Subsequently, we build *instance-to-node* mappings (e.g., $[Pod\text{-}A, Pod\text{-}B, Pod\text{-}C] \rightarrow Node\text{-}X$) by inspecting the `spec.nodeName` attribute of each Pod. To maintain an accurate topology view during dynamic workload redistribution, we utilize the K8s Watch API to track real-time Pod lifecycle events. Instances co-residing on the same node are flagged as potential RIDs based on these mappings. However, simple co-location does not guarantee actual interference due to native isolation mechanisms like cgroups and CPU pinning. We address this discrepancy by dynamically validating and quantifying these structural dependencies. Specifically, we analyze their runtime workload characteristics by computing an attention score between co-located services using their real-time performance metrics (see Sec. 3.4.2).

3.3.3 Execution Blocking Dependencies. The mining of EBDs employs a two-stage process. First, we infer *call order* relationships between services, which refer to the execution sequence of sibling microservices (i.e., services invoked by the same parent). Based on the call orders, we then identify deeper sequential constraints across various workflow execution paths, which constitute the EBDs.

Call order construction. We mine runtime traces to construct dynamic call orders, which is essentially a timestamp-based graph construction problem [11, 33, 50]. For each span of service v_k, we identify its child spans representing the sibling services (say v_i and v_j) and sort them by start timestamps. In this sorted sequence, a pairwise call order $v_i \rightarrow v_j$ is considered if, for two chronologically adjacent spans s_i and s_j (corresponding to v_i and v_j, respectively), they satisfy $s_j.start\_time \geq s_i.end\_time - \delta$. The small tolerance $\delta \geq 0$ accounts for potential clock skew between nodes. Within each parent service context, all such pairwise call orders are transitively merged to form *call order paths* of the parent v_k. For instance, merging $User\ validation \rightarrow Payment\ processing$ and $Payment\ processing \rightarrow Notification$ yields the path $User\ validation \rightarrow Payment\ processing \rightarrow Notification$. This merging process is recursively applied until no more call orders can be merged.

EBD mining. Given a service v_p that precedes v_s within the same parent context, we identify the child services at the tail of v_p's call order paths, whose completion is a prerequisite for v_s to initiate. Each tail service is recursively traversed to explore its own tail services, forming a dependency chain that terminates at leaf services (those issuing no further calls) of the invocation graph rooted at v_p.

An EBD is established from these tail leaf services to the direct successor of v_p, i.e., v_s, in the inferred call order. Consider the *Checkout* service in Fig. 1, where *Payment processing* precedes *Notification* in the call order. The path $[\cdots \rightarrow Trans.\,execution \rightarrow Shipping]$ represents one of the call order paths initiated by *Payment processing*. In this sequence, *Shipping* functions as the tail service. The same applies to *Inventory*, as *Payment processing* calls them concurrently. Since both *Shipping* and *Inventory* are tail services in distinct call order paths from *Payment processing*, they maintain an EBD with their common successor, *Notification*. This logic extends to a finer granularity within the *Payment processing* workflow itself, where *Trans. execution* precedes both *Inventory* and *Shipping*. As *Trans. execution* is a leaf node, it is identified as its own tail service. This establishes an EBD from *Trans. execution* to its successors, indicating that their execution is blocked until the transaction completes.

The above mining processes may introduce inaccuracies. *False positives* can arise from clock skew or network jitter in EBD mining (mitigated by δ), transient co-location without actual contention for RIDs (addressed by attention-based validation in Sec. 3.4.2), or stale traces for DIDs (mitigated by aggregation over a sufficient observation window). These spurious edges may lead to resource misallocation driven by non-existent relationships. Our hybrid attention mechanism provides a principled defense: since attentions are computed from real-time monitoring data within per-type heads, false dependencies lacking consistent correlations receive near-zero weights and are excluded from scaling decisions (Sec. 3.4.2). *False negatives* occur when certain execution paths remain unexercised during the observation window, leaving some dependencies undetected. In this case, Panorama falls back to utilizing DIDs alongside each service's temporal patterns. This is precisely the methodology that existing methods [19, 32, 48] employ. Panorama's additional value stems from RIDs and EBDs when correctly detected.

3.4 Predictive Autoscaling

After mining service dependencies, we introduce our model for dynamic workload forecasting and resource provisioning, as shown in Fig. 3. The architecture extends the principles of the Transformer model [45] to leverage heterogeneous service dependencies and handle multivariate time series data defined on a graph structure. The model is trained and adapted using a meta-learning strategy.

3.4.1 Input Embedding. The input metric data $\mathcal{X}_{t-\tau+1:t} \in \mathbb{R}^{N\times L\times \tau}$ are mapped into a D-dimensional embedding space. To enrich this initial representation with microservice contextual information, we incorporate both temporal and spatial positional encodings.

- **Temporal Positional Encoding (TPE).** To encode the sequential order within the historical window, a learnable positional representation $TPE_{t'} \in \mathbb{R}^D$ corresponding to time step $t' \in [t-\tau+1, t]$ is injected. This encoding explicitly captures the chronological position of each observation sequence.
- **Spatial Positional Encoding (SPE).** To capture the dynamic topological features of microservices, we introduce a spatial positional encoding $SPE_i \in \mathbb{R}^D$ for each service v_i. It leverages the spectral properties of the invocation graph formed by the DIDs at time t, reflecting the microservices' real-time interaction patterns and traffic flow. Let $\mathcal{A}_t$ denote the adjacency matrix representing DIDs and $\mathcal{D}_t$ represent its diagonal degree matrix where $(\mathcal{D}_t)_{ii} = \sum_j (\mathcal{A}_t)_{ij}$. The Symmetric Normalized Laplacian $L_{sym,t}$ is computed using the identity matrix I as:

$$L_{sym,t} = I - \mathcal{D}_t^{-1/2}\mathcal{A}_t\mathcal{D}_t^{-1/2} \tag{1}$$

We extract the dynamic features $\phi_i \in \mathbb{R}^k$ from the eigenvectors associated with the k smallest non-trivial eigenvalues of $L_{sym,t}$. The final encoding SPE_i is produced via a linear layer:

$$SPE_i = \mathcal{W}_s\phi_i + b_s \tag{2}$$

where $\mathcal{W}_s \in \mathbb{R}^{D\times k}$ and $b_s \in \mathbb{R}^D$ constitute learnable weight and bias parameters. This spectral approach effectively embeds services into a continuous latent space where structural proximity and topological pathways are structurally preserved. In particular, SPE is constructed solely from the DID graph, excluding RIDs and EBDs. The invocation topology captured by DIDs evolves relatively slow, so its Laplacian eigendecomposition provides a stable spatial reference frame across time steps. In contrast, RIDs shift with pod rescheduling and EBDs vary with workload intensity and request mix. Including these volatile edges would cause positional encodings to drift between consecutive steps, undermining SPE's stability as a structural anchor. The hybrid attention mechanism (Sec. 3.4.2), which recomputes dependency-specific attention scores at each time step, is better suited for capturing these dynamic dependencies.

Let $e_i^{t'} \in \mathbb{R}^D$ denote the foundational D-dimensional representation of the raw features $x_i^{t'}$ after linear projection. The final input embedding for service v_i at time step t' integrates the temporal and spatial insights through element-wise addition:

$$h_i^{t'(0)} = e_i^{t'} + TPE_{t'} + SPE_i \tag{3}$$

These aggregated embeddings form the primary input tensor $\mathcal{H}^{(0)} \in \mathbb{R}^{N\times D\times\tau}$ for the subsequent encoder layers.

3.4.2 Encoder Layers. Panorama's prediction model uses K identical encoder layers. Each layer takes the output of the previous layer $\mathcal{H}^{(l-1)}$ and produces $\mathcal{H}^{(l)}$ via a Hybrid Dependency-Temporal Self-Attention mechanism and a position-wise Feed-Forward Network (FFN), interspersed with normalization and residual connections.

Hybrid Dependency-Temporal Self-Attention. The core of each encoder layer is a self-attention mechanism that decouples the standard global attention into independent heads: one for temporal attention and three for the attentions of different dependency types. This enables Panorama to capture the complex semantic relationships across these four dimensions. For a dependency attention head $\mathcal{A}^{(\kappa)}$ focusing on dependency type κ ($\kappa \in \{did, rid, ebd\}$), the input $\mathcal{H} \in \mathbb{R}^{N\times D\times\tau}$ is projected into Queries ($\mathcal{Q}^{(\kappa,h)}$), Keys ($\mathcal{K}^{(\kappa,h)}$), and Values ($\mathcal{V}^{(\kappa,h)}$) for head h. Attention scores between nodes i and j at time t' are computed as:

$$\alpha_{i,j}^{t'(\kappa,h)} = \frac{\mathcal{Q}_i^{t'(\kappa,h)} \cdot \mathcal{K}_j^{t'(\kappa,h)}}{\sqrt{D_h}} \tag{4}$$

where D_h is the dimension of a single head. A dependency-based masking is applied such that $\alpha_{i,j}^{t'(\kappa,h)} = -\infty$ if $A_{ij}^{(\kappa)}$ indicates no dependency. The masked scores are normalized via softmax to obtain attention weights $\hat{\alpha}^{(\kappa,h)}$. This mechanism enables a service to disregard unrelated services and concentrate on dependent ones. Thus, the model can use the discrete dependency matrix to quantify

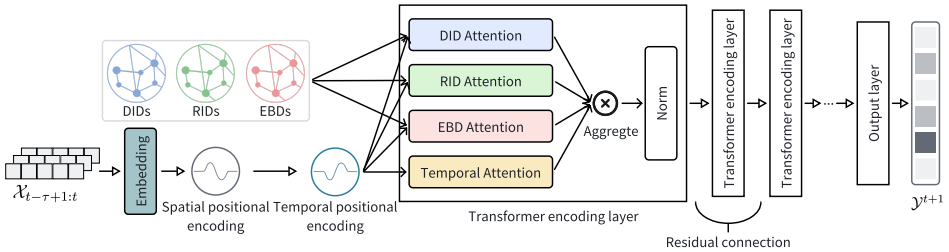

Figure 3: The Architecture of Panorama's Prediction Model

the actual strength of these dependencies, which is represented by the attention scores α. These scores facilitates the enhancement of the model's capacity for effective and informed resource management decisions, underpinned by dynamic quantitative dependency information. The output for this head is the weighted sum of values:

$$O_i^{t'(\kappa,h)} = \sum_{j=1}^{N} \hat{\alpha}_{i,j}^{t'(\kappa,h)} V_j^{t'(\kappa,h)} \tag{5}$$

Temporal attention heads operate similarly but compute attention scores across the τ dimension for each node i. The outputs from all heads are concatenated and linearly transformed.

False Positive Filtering via Learned Attention. The hybrid attention mechanism inherently defends against false positives from the mining stage. Spurious dependencies in $A^{(\kappa)}$ may pass the structural mask, but their dot-product scores remain low (compared to genuine dependencies) due to absent consistent temporal or performance correlations, yielding near-zero attention weights $\hat{\alpha}^{(\kappa,h)}$ after softmax normalization. Per-type attention heads further isolate false positives to prevent cross-type contamination. During training, the L1 loss (Eq. 6) penalizes attention to non-causal dependencies, reinforcing this suppression. Together, structural masking, data-driven attention weighting, per-type head isolation, and training-time optimization effectively neutralize false positives.

Feed-Forward Network and Residual Connections. Following the attention mechanism, each encoder layer includes a position-wise FFN. To mitigate vanishing gradients and enhance training stability, residual connections [18] are integrated with layer normalization around both sub-layers. Additionally, stochastic depth [21] is employed as a regularization technique to prevent overfitting and enhance the model's generalization performance.

3.4.3 Multi-scale Feature Aggregation and Output. The outputs of all encoder layers are aggregated via residual connections to capture multi-scale features. The final aggregated representation is processed by a series of linear layers to produce the predicted resource demand for each service at time $t+1$. We optimize our model by minimizing the L1 loss between the predicted demand vector $\mathcal{Y}^{t+1} \in \mathbb{R}_{\geq 0}^{N}$ and the ground-truth demand vector $\mathcal{Y}_{\text{true}}^{t+1} \in \mathbb{R}_{\geq 0}^{N}$. The loss function is defined as:

$$\mathcal{L}(\mathcal{Y}^{t+1}, \mathcal{Y}_{\text{true}}^{t+1}) = |\mathcal{Y}^{t+1} - \mathcal{Y}_{\text{true}}^{t+1}| \tag{6}$$

The ground-truth demand vector $\mathcal{Y}_{\text{true}}^{t+1}$ is derived retrospectively using the standard K8s HPA scaling formula [25]: $y_{i,\text{true}}^{t+1} = \lceil r_i^t \cdot c_i^t \,/\, c_{\text{target}} \rceil$, where r_i^t is the current replica count of service v_i, c_i^t is its observed average CPU utilization, and c_{target} is the target utilization threshold (set to 70% in this work). This heuristic is widely used in production systems and prior autoscaling research [32, 46]. Because optimal resource demand cannot be measured directly in a live system, we adopt this formula as the ground-truth label. It naturally captures both scaling directions: when $c_i^t < c_{\text{target}}$, the service is under-utilized and the formula yields $y_{i,\text{true}}^{t+1} < r_i^t$, signaling that fewer replicas suffice; when $c_i^t > c_{\text{target}}$, it yields $y_{i,\text{true}}^{t+1} > r_i^t$, signaling that more replicas are needed. This frames demand prediction as a well-defined optimization goal, computing the ideal replica count that maintains the target utilization given the observed load. In the resource orchestration phase, the prediction vector, $\mathcal{Y}^{t+1} \in \mathbb{R}_{\geq 0}^{N}$, directly serves as the forecasted demand for all services.

3.4.4 Meta-learning for Model Adaptation. Microservice environments are inherently dynamic and uncertain. Continuous shifts in workload patterns and evolving dependencies can cause distributional drift from training data, degrading prediction accuracy. Traditional approaches typically address this issue by retraining the model on newly observed data, which can incur prohibitive time and maintenance costs. To equip Panorama with robust adaptability, we design a meta-learning paradigm based on Model-Agnostic Meta-Learning (MAML) [12, 51, 53], where we introduce the concept of *workload periods* as the core units for task division. A new workload period, i.e., a meta-learning task, is identified when the computed dissimilarity between consecutive segments of workloads (DTW) or dependencies (Jaccard similarity) exceeds a predefined threshold λ. λ is set to the 95th percentile of the dissimilarities observed across consecutive segments during the training phase, serving as a breakpoint between normal within-period variation and genuine distributional shift. Each task encapsulates a particular operational scenario defined by request rate fluctuations, service interaction patterns, and underlying resource contention. In each workload period, the data collected are used to perform an adaptation step, fine-tuning the predictive model for the current operational context.

Unlike conventional fine-tuning or full retraining, MAML meta-trains parameters across diverse tasks, yielding base weights suited for rapid adaptation. Upon detecting drift, Panorama adapts with only a few gradient steps on limited recent data, starting from near task-optimal parameters. Two contextual elements provide task- specific information: spatial positional encoding (Sec. 3.4.1), capturing current invocation patterns, and real-time dependencies, integrated as structural priors through the attention mechanism.

3.5 Resource Orchestration

The final phase executes resource provisioning decisions. A key challenge is to prevent system oscillations, where overly frequent resource adjustments can trigger unnecessary scale-up and scale-down events, leading to instability and resource waste. Thus, we

implement two smoothing mechanisms. First, we establish a minimum time interval η (six minutes) between consecutive scaling operations. Second, we make scaling decisions by averaging predicted value over a future time window to smooth out minor fluctuations.

An *online scheduling* component takes the predicted resource demands, incorporates potential static policies or constraints (e.g., min/max instance limits, deployment quotas), and adjusts per-service replica counts via K8s client APIs by modifying the corresponding Deployment objects, ensuring uniform resource specifications across all instances. This orchestration guarantees uniform allocation across replicas while dynamically adjusting aggregate processing capacity. Upon receiving the updated replica targets, the K8s control plane automatically provisions or terminates Pods across available worker nodes. This continuous adjustment reconfigures the runtime topology and compute capacity of the targeted services. These infrastructural modifications close the autonomous autoscaling loop. The resulting shifts in system state and operational metrics are subsequently captured by the performance monitoring phase to drive the next optimization cycle.

4 Experimental Evaluation

In this section, we conduct experiments to evaluate the performance of Panorama, aiming to answer the following questions:

- **RQ1:** How effective is Panorama in autoscaling?
- **RQ2:** How adaptive is Panorama in autoscaling?
- **RQ3:** Are latent dependencies useful for autoscaling?
- **RQ4:** What is the performance overhead of Panorama?

4.1 Experimental Settings

4.1.1 Benchmark Microservices. The evaluation is conducted using four representative microservice benchmarks that exhibit characteristics of interactive and responsive applications. They use a variety of programming languages and frameworks (e.g., C#, Java, Python), which are widely used in microservice-related research [10, 27, 55] and provide diverse structural and workload complexities.

- *Online Boutique* [15] represents a cloud-native e-commerce prototype designed to showcase K8s and gRPC interactions. It models standard online retail operations such as merchandise browsing, cart management, and checkout workflows.
- *Social Network* [14] features unidirectional follow relations and multiple services communicating via Apache Thrift RPCs. It supports complex user interactions like multimedia posts and timeline reads. The resulting complex service chains and diverse workloads make it highly suitable for scalability analysis.
- *Hotel Reservation* [14] implements a hotel reservation service built using Go and gRPC. It simulates user interactions involving transactional operations, e.g., hotel querying, recommendation. It involves interactions between multiple services to fulfill these user requests, making it a suitable testbed for autoscaling.
- *Train Ticket* [59] is a railway ticketing application comprising 41 microservices. These services are responsible for a wide array of functionalities, including user authentication, ticket booking, payment processing, and notification. Its extreme scale and functional density create an exceptionally challenging environment for system-wide optimization algorithms.

4.1.2 Workload Generation. To ensure our evaluation reflects realistic production conditions, we model workload intensity using a public microservice trace dataset from Alibaba [30], which provide minute-level aggregate request rates per service interface. As shown in Fig. 4, the full workload spans 6.25 hours with 4,500 data points. We employ Locust [29], an asynchronous workload generator, to replay the traffic profile. Each data point, representing a target requests-per-second (rps) value, is used by Locust to govern the request generation for a five-second interval. At each second, Locust randomly invokes various service APIs based on the target rps. The profile is split into a 5-hour training phase and a subsequent 1.25-hour evaluation phase. The training traffic exhibits clear periodic patterns and wide rps fluctuations, including sudden spikes and sharp drops that stress the autoscaler's responsiveness. The evaluation traffic begins with a relatively stable period of minor fluctuations, followed by a distinct upward trend in request rate. This workload profile thus presents a challenging scenario for assessing the adaptive capabilities of autoscaling mechanisms.

Figure 4: Production Traffic from Alibaba's Traces

4.1.3 Evaluation Metrics. To validate the performance of different autoscalers, we use *instance count* and *response time* as the evaluation metrics, which represent the fundamental trade-off between resource cost and service performance. Instance count directly reflects resource consumption and cost, while response time is the most critical QoS/SLA indicator of service performance that impacts user experience. For robust and reliable comparison, we evaluate methods based on the *average instance count (AIC)* and the *average response time (ART)* observed over the experimental period.

Besides the trade-off between performance and cost, understanding the efficiency of resource usage and compliance with SLA constraints is equally important. Thus, we also measure *CPU utilization* of service instances and *p95 response time* during experiments. A higher average CPU utilization (without degrading performance) generally indicates more efficient resource usage and helps avoid resource over-provisioning. The p95 response time refers to the 95th percentile of response time among all requests. As a high-end latency metric, it captures system tail performance and is less susceptible to outliers compared to the maximum response time.

4.1.4 Baseline Methods. We compare Panorama's performance against several state-of-the-art methods, which represent diverse autoscaling paradigms.

- *Horizontal Pod Autoscaling (HPA)* [25] is the built-in autoscaler of K8s and a widely-used rule-based approach. It monitors per-service CPU utilization and adjusts the replica count to maintain a target threshold. While simple and robust, HPA operates on a per-service basis and cannot capture inter-service dependencies or anticipate workload propagation.
- *DeepScaler* [32] is an autoscaling approach based on deep learning. It uses spatiotemporal graph neural networks and adaptive graph learning to capture service dependencies and accurately

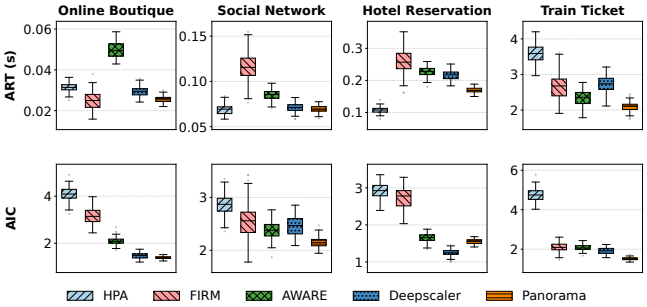

Figure 5: Autoscaling Performance of Different Approaches

estimate resource needs for simultaneous scaling actions. It represents a state-of-the-art approach leveraging dependency-aware modeling for microservice resource management.

- *FIRM* [39] is a reinforcement learning (RL)-based approach that learns to adjust resources based on feedback. It finds services with abnormal response times via anomaly detection and adjusts multiple resources for the service through RL iterations, aiming to learn effective scaling policies from experience.
- *AWARE* [40] is another RL-based approach designed for cloud systems. It features rapid online learning mechanisms alongside a safe bootstrapping strategy to guarantee stable performance during both initial exploration and online serving phases.

4.1.5 Experimental Configuration. We conduct experiments on a 12-node K8s cluster. Each node is a VM with 8 dedicated vCPUs, 32 GB memory, and Ubuntu 22.04 LTS, distributed evenly across 6 physical hosts. Services are deployed as standard K8s Deployments with default topologies, monitored via Prometheus for metrics and Jaeger for distributed tracing, and operate under the K8s "burstable" QoS class with standard cgroup isolation. We avoid strict hardware partitioning to simulate realistic shared production environments. This setting intentionally preserves the "noisy neighbor" contention to validate Panorama's explicit modeling of resource interference.

4.2 Experimental Results

4.2.1 The Effectiveness of Panorama (RQ1). We evaluate all approaches across the benchmarks and measure service ART and AIC. As Fig. 5 shows, Panorama consistently achieves the best overall performance. On Online Boutique, it delivers the lowest ART (0.025 s) and AIC (1.38), using significantly fewer resources than all baselines (e.g., 4.12 AIC for HPA). On Social Network, Panorama yields an ART (0.069 s) comparable to HPA (0.069 s) while maintaining the lowest AIC (2.16 vs. 2.87 for HPA). For Hotel Reservation, HPA obtains the lowest ART (0.107 s) via aggressive over-provisioning (2.92 AIC); Panorama balances this trade-off with 0.170 s ART and 1.56 AIC. Train Ticket, the most challenging benchmark, highlights Panorama's advantages most clearly, reducing ART from 3.57 s for HPA to 2.08 s and AIC from 4.74 to 1.51. To assess stability, we repeat each experiment with three random seeds. The boxplots display the resulting distributions, showing median, interquartile range, and outliers. Panorama exhibits the smallest variability across all benchmarks, with ART standard deviation ranging from 0.002 to 0.115 s (vs. 0.286 s for HPA and 0.220 s for DeepScaler on Train Ticket) and AIC standard deviation from 0.063 to 0.095 (vs. 0.332 for HPA). Panorama achieves the best median

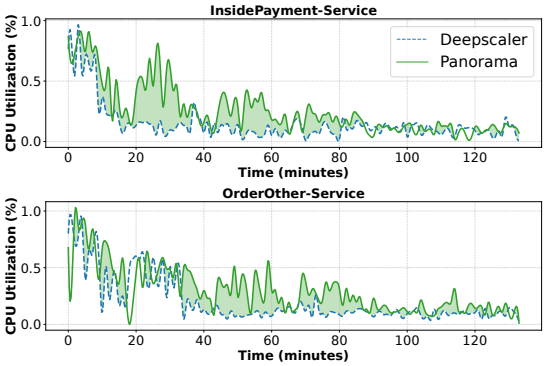

Figure 6: CPU Utilization of Panorama and DeepScaler

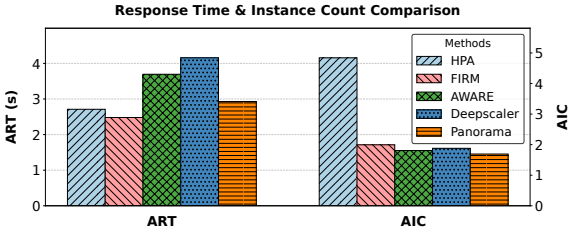

Figure 7: Adaptability of Different Approaches

ART on all benchmarks and the lowest median AIC on three, with narrow interquartile ranges confirming stable, reproducible gains.

We also calculate CPU utilization to evaluate resource efficiency. As Fig. 6 shows for two core Train Ticket services, Panorama maintains higher utilization than the strongest baseline DeepScaler for most of the observation period, and consistently outperforms DeepScaler on over 70% of Train Ticket's microservices, demonstrating superior resource efficiency. This shows its superior resource efficiency in effectively leveraging allocated resources. Beyond CPU utilization, we also measure the p95 response time of microservices, as shown in Fig. 8. For the selected core business services of Train Ticket, our method achieves lower p95 response times compared to all baselines. This enhanced tail-latency performance can be attributed to our method's dynamic dependency awareness, which enables timely and precise scale-out of critical services in response to sudden load spikes, thereby effectively mitigating SLA violations.

4.2.2 The Adaptability of Panorama (RQ2). We evaluate all methods in a dynamic and changing setting, where the workloads are not included in the training data. We select Train Ticket as our benchmark due to its close resemblance to real-world microservice deployments. For the workload, we select a set of complex request scenarios designed to stress the system's inter-service communication. This involves filtering for workload requests that flow through three or more distinct services to focus on non-trivial execution paths. Thus, the workload used represents a distinct workload period (Sec. 3.4.4) that significantly deviates from the historical data.

As shown in Fig. 7, while HPA achieves marginally better ART than Panorama, its AIC is approximately three times higher than that of Panorama. This stems from HPA's reactive, per-service scaling mechanism, which triggers aggressive scale-outs based solely on local utilization thresholds without considering cross-service dependencies or overall cluster efficiency. Among other baselines, FIRM attains the lowest ART at higher instance cost,

Table 1: Comparison of Different Adaptation Strategies

Model Variant	ART (s)	AIC
Panorama (MAML)	2.89	1.69
Panorama (Finetuning)	2.96	2.27
Panorama (No adaptation)	3.28	2.42

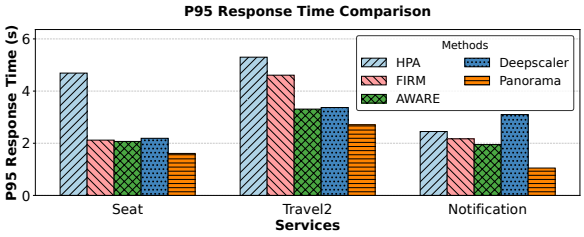

Figure 8: P95 Response Time of Different Approaches

while Panorama strikes the best balance with competitive ART and the fewest instances. RL-based methods (FIRM, AWARE) benefit from their online learning and adaptation capacity, whereas DeepScaler yields the highest ART, struggling with rapid workload shifts. Panorama leverages RIDs and EBDs that capture the deep operational logic between services. When combined with meta learning, Panorama is able to effectively process and manage complex workloads within highly dynamic microservice environments.

To isolate the contribution of meta learning, we conduct an ablation comparing three adaptation strategies under the same dynamic workload: (1) full Panorama with MAML-based adaptation, (2) Panorama with standard fine-tuning from training-phase weights rather than a meta-learned initialization, and (3) Panorama without any adaptation, applying the training-phase model directly to the unseen workload. As Table 1 shows, the no-adaptation variant exhibits the worst ART, as it cannot adjust to the distributional shifts in test workloads. Standard fine-tuning improves but converges slowly and requires more data to reach comparable accuracy. MAML-based Panorama achieves the best ART with a much lower AIC, as its meta-learned initialization enables rapid task-specific updates with few gradient steps on limited recent data. This confirms that meta learning delivers distinct, substantial adaptability gains beyond dependency modeling and attention mechanisms alone.

4.2.3 The Usefulness of Latent Dependencies (RQ3). To validate the utility of latent dependencies, we conduct an ablation study on Train Ticket, which features the most intricate dependencies. To assess the impact of the RIDs and EBDs, we configure different variants of Panorama that omit them. As shown in Table 2, "Panorama w/o RID & EBD" achieves the best AIC (1.35), while the full Panorama uses 6.7% more instances (1.44). However, this marginal increase yields a substantial performance improvement: without RIDs and EBDs, ART increases from 2.08 s to 3.14 s, a 51% degradation. This occurs because this variant can only react to superficial scaling needs of services with explicit call dependencies, overlooking other indirectly affected services that possess latent dependencies. This leads to bottlenecks in these unaddressed services, ultimately resulting in prolonged service response times and compromised service quality. Other experiments with "Panorama w/o RID" and "Panorama w/o EBD" reveal similar performance degradation patterns across both metrics.

We further show the usefulness of RIDs and EBDs via a case study on proactive bottleneck resolution. We focus on the *RoutePlan* service of Train Ticket, which orchestrates intensive travel planning computations and frequently becomes a bottleneck during traffic surges. To eliminate the noise from mixed requests and trigger a bottleneck at *RoutePlan*, we apply a dedicated, upward-trending workload consisting exclusively of itinerary queries. We exclude HPA due to its limitations in accurately considering resource utilization and aggressive over-provisioning. Besides *RoutePlan*'s ART, we also calculate its average response time of requests (ARTR) to measure the system's end-to-end performance. The results are shown in Table 3. Particularly, the response times are higher than typical steady-state values since the system is intentionally initialized in a resource-constrained state. The reported metrics include the initial period of queue accumulation prior to autoscaler intervention. Once scaled, the system stabilizes at normal millisecond-level latency.

Panorama consistently outperforms all baselines in both metrics. During workload surge, Panorama uses RIDs and EBDs to identify *RoutePlan* as a latent bottleneck, proactively scaling out *RoutePlan* and other related services. Conversely, the baseline methods lack this awareness and allocate additional instances to other services that exhibit more direct load signals, such as the upstream services that invoke *RoutePlan*. This leaves the root bottleneck unresolved. Furthermore, the increased throughput from the scaled upstream services overwhelms the unscaled *RoutePlan*, which severely exacerbates the bottleneck and degrades response times.

Having demonstrated the critical impact of dependencies, we now evaluate the dependency mining capabilities of Panorama against the leading method DeepScaler, which infers latent correlations rather than explicitly modeling RIDs and EBDs. Fig. 9 shows dependency heatmaps for both methods on Train Ticket, with each cell $(i,\ j)$ quantifying the dependency strength between service i and j. The comparison reveals a clear contrast in both the completeness and interpretability of the discovered dependencies. DeepScaler identifies DIDs and indirectly infers some RIDs from correlated metrics, but largely misses EBDs, which are rooted in workflow semantics and require execution-order understanding beyond metric correlation. In contrast, Panorama's explicit mining process yields a far more comprehensive and semantically rich dependency graph, which allows it to proactively anticipate and mitigate performance degradation. Furthermore, these dependency insights provide actionable diagnostics for system optimization. For example, there are some dense columns in Panorama's RID heatmap, which identify services that act as resource hotspots. This enables operators to perform targeted interventions, such as adjusting resource allocations or re-scheduling services.

To quantitatively compare dependency discovery, we evaluate binary *existence* (whether ground-truth edge is detected) rather than dependency *strength*, because precise strength depends on runtime factors such as instantaneous service load. We manually establish ground-truth dependencies. DID ground truth is derived from each application's source code call graph. EBD ground truth is obtained by analyzing workflow logic to identify execution ordering constraints. We do not evaluate RIDs because co-location does not reliably imply resource interference; whether contention actually occurs depends on workload intensity and cgroup configurations, making ground truth difficult to establish. Across all

Table 2: Performance of Panorama Variants

Model Variant	ART (s)	AIC
Panorama	2.08	1.44
Panorama w/o RID	2.95	1.43
Panorama w/o EBD	3.04	1.44
Panorama w/o RID & EBD	3.14	1.35

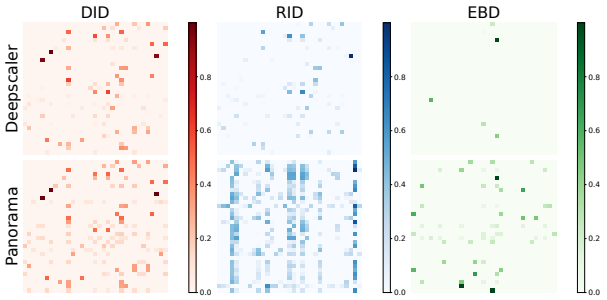

Figure 9: Dependency Strength Comparison

benchmarks, Panorama detects all DIDs and >95% of EBDs via its mechanism-specific mining of traces and workflow semantics. DeepScaler reports a comparable number of DIDs but virtually no EBDs, since its EM-based affinity refinement optimizes for prediction error rather than dependency fidelity and cannot recover workflow-level execution ordering from metric correlation alone.

4.2.4 The Overhead of Panorama (RQ4). Panorama's overhead is a critical factor for practical deployment. We measure CPU usage (in millicores), memory consumption, and execution time across its four core operations: *performance monitoring*, *dependency mining*, *meta-learning*, and *resource orchestration*. Table 4 shows the results. Persistent background performance monitoring via Prometheus and Jaeger consumes around 558 millicores of CPU and 936 MB of memory. Dependency mining periodically uses telemetry data to extract RIDs and EBDs, requiring 1,148 millicores of CPU and 668 MB of memory for about 21.5 seconds per run. For meta-learning, initial offline meta-training on five hours of data (partitioned into 12 workload periods) takes roughly four hours as a one-time cost. Upon detecting a distributional shift at runtime, Panorama adapts the meta-learned base model via a few gradient steps on approximately 24 minutes of recent data. This adaptation consumes 970 millicores of CPU and 4,355 MB of memory and completes in roughly 76.5 seconds. Finally, resource orchestration performs lightweight model inference at each scaling interval, utilizing 1,711 millicores of CPU and 300 MB of memory. The observed 20-second orchestration time primarily results from our prototype serially issuing K8s API calls for Train Ticket's 41 microservices. A production deployment would parallelize these calls to substantially reduce latency. Our method demonstrates its practical significance by incurring modest overhead, making it suitable for production deployment.

To quantify whether performance gains justify the operational cost of Panorama, we compute an *efficiency ratio*: cluster-wide CPU savings from reduced replicas divided by Panorama's own CPU consumption. On Train Ticket (41 microservices), Panorama requires around 3 fewer replicas per service than HPA (Fig. 5). At a per-instance CPU request of 100 millicores, this translates to cluster-wide savings of $\sim 3 \times 41 \times 100 = 12{,}300$ millicores. Panorama consumes 558 millicores for the continuously running monitoring layer, plus $(1148 * 21.5 + 970 * 76.5 + 1711 * 20)/360 = \sim 370$ millicores amortized from the six-minutes periodic dependency mining and orchestration operations, for a total overhead of roughly 928 millicores. This yields an efficiency ratio of $12{,}300/928 > 13{:}1$. That is, for every unit of CPU that Panorama consumes, it saves over an order of magnitude more across the managed cluster.

Table 3: A Case Study on the RoutePlan Service

Method	ART (s)	ARTR (s)
Panorama	88.33	160.66
FIRM	114.87	183.60
AWARE	123.59	175.95
DeepScaler	120.11	167.51

Table 4: The Operational Overhead of Panorama

One-shot Operation	Overhead		
	CPU (millicores)	Mem (MB)	Time (s)
Performance Monitoring	558 ± 56	936 ± 18	+∞
Dependency Mining	1148 ± 14	668 ± 7	21.5 ± 1.0
Meta Learning	970 ± 56	4355 ± 31	76.5 ± 2.4
Resource Orchestration	1711 ± 30	300 ± 10	20 ± 1.2

4.3 Parameter Sensitivity

In this section, we evaluate the sensitivity of Panorama against three key hyperparameters.

The historical window size τ controls the temporal context for resource demand prediction (Sec. 2.3). We evaluate $\tau \in \{6, 12, 24\}$ across all benchmarks. As shown in Fig. 10, $\tau = 12$ yields the best trade-off: $\tau = 6$ provides too few steps to capture workload trends, degrading ART, while $\tau = 24$ introduces noise from stale observations. In all cases, the variation remains modest, demonstrating that Panorama is not overly sensitive to τ. This robustness stems from the temporal attention heads, which learn to down-weight irrelevant observations, and the meta-learning component, which adjusts the model's temporal focus when workload patterns shift. We thus apply $\tau = 12$ across all benchmarks without per-application tuning. Although characteristics such as request chain length and traffic variability could in principle favor different window sizes, our results indicate that these mechanisms provide sufficient flexibility.

For the workload period threshold λ (Sec. 3.4.4), we vary the calibration percentile over $q \in \{90, 95, 99\}$, recomputing the threshold from the same training-phase dissimilarity distribution for each q and repeating the RQ2 evaluation. Lower percentiles produce more, shorter periods and hence more frequent adaptation, whereas higher percentiles merge more adjacent segments and can delay adaptation. Across the three settings, both ART and AIC vary by less than 5% (with $q = 95$ achieving the best results), confirming that Panorama is robust to the threshold choice within this range.

We also assess sensitivity to the scaling interval η (Sec. 3.5) by varying it over $\{3, 6, 12, 24\}$ minutes on Train Ticket. A shorter interval enables more frequent adjustments, allowing the autoscaler to react promptly to workload changes, but at the cost of increased

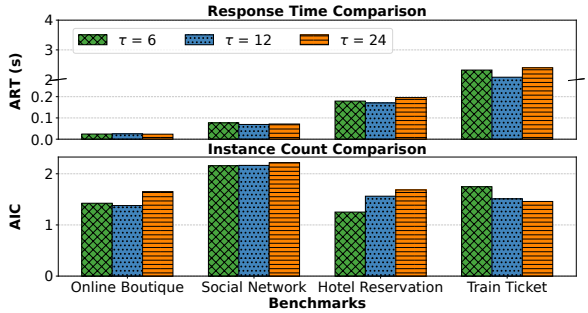

Figure 10: Sensitivity Analysis of Window Length τ

resource overhead from more frequent scaling actions. A longer interval reduces operational overhead but delays the system's response to rapid workload shifts. ART is nearly identical at 3 and 6 minutes (2.08 s), but AIC drops 17% from 1.82 to 1.51 at 6 minutes. Beyond 6 minutes, ART degrades noticeably (16% higher at 12 minutes, 36% higher at 24 minutes), while AIC decreases only marginally (1.45 at 12 minutes, 1.40 at 24 minutes). The 6-minute interval thus strikes the optimal balance between responsiveness and stability, avoiding oscillatory behavior.

4.4 Threats to Validity

The primary threat to *internal* validity lies in the implementation of our approach and the baselines. To mitigate this, we utilize author-provided implementations for the baselines and perform code reviews and testing of our own approach and experimental scripts.

Threats to *external* validity concern the generalizability of our findings. We mitigate this by using diverse microservice applications with varying scales and architectures. The method's adaptability is further tested with different workloads on Train Ticket. We also broaden our comparison by including both traditional rule-based and learning-based baselines. Moreover, our evaluation incorporates resource efficiency metrics like CPU utilization and P95 response time, providing a more comprehensive evaluation.

Threats to *construct* validity relate to whether our evaluation metrics accurately reflect the effectiveness of the approach. To alleviate it, we conduct a case study for fine-grained analysis and use visualization to show the effectiveness of the latent dependencies.

5 Related Work

5.1 Rule-based and Model-based Autoscaling

Rule-based autoscaling approaches, exemplified by K8s HPA [25] and commercial services like AWS Auto Scaling [1], Azure Autoscale [2], and Google Cloud Load Balancing [16], operate through predefined threshold-based mechanisms. While simple and widely adopted, these methods are reactive, treat services in isolation, and often struggle with the dynamic nature of microservice workloads.

Model-based approaches advance beyond thresholds by establishing mathematical relationships between workload and resource requirements, including queuing theory models [23, 44], time series models like ARMA [35], and workload prediction methods such as CloudScale [42] (FFT), AGILE [34] (wavelet transforms), and Autopilot [41] (exponential smoothing). However, these approaches often rely on strong assumptions about workload distributions and typically do not capture inter-service dependencies.

5.2 Deep Learning-based Autoscaling

Recent studies leverage deep learning techniques for more sophisticated, dependency-aware autoscaling. A prominent line of research utilizes Graph Neural Networks (GNNs) to model spatiotemporal relationships between microservices [19, 32]. For instance, DeepScaler [32] employs a spatiotemporal GNN with adaptive graph learning, iteratively refining an "affinity matrix" from the direct invocation graph to capture latent statistical correlations. Other GNN-based frameworks integrate system state into spatio-temporal GNNs [31] or focus on proactive allocation for tail latency SLOs [36]. While these methods move beyond static call graphs, the dependencies they learn are often black-box correlations lacking clear physical or logical semantics, making root cause diagnosis difficult.

Another category captures system dynamics without an explicit graph structure. For example, Sinan [57] treats the collective service state as an "image" and uses a Convolutional Neural Network (CNN) to learn mappings from resource allocations to QoS. RL-based approaches, such as FIRM [39], AWARE [40], and MTS-PPO [20], learn scaling policies from environmental feedback. They are powerful but treat the system as a black box, with dependency understanding implicitly encoded in the learned policy or value function.

While these approaches have improved autoscaling, their dependency modeling relies on opaque correlations seeded by direct invocations [32, 47], without distinguishing between fundamentally different interaction mechanisms. For instance, DeepScaler's EM-based affinity refinement optimizes for prediction error rather than dependency fidelity, so the resulting matrix conflates genuine dependencies with spurious correlations and cannot distinguish their underlying mechanisms. Panorama advances this by moving from implicit correlation learning to explicit dependency modeling. Our core contribution is the systematic formulation and mining of two latent dependency categories: resource interference dependencies, grounded in the physical deployment topology, and execution blocking dependencies, grounded in logical workflow semantics.

6 Conclusion

We present Panorama, a microservice autoscaling framework that addresses fundamental limitations in current approaches by explicitly modeling latent dependencies beyond direct service invocations. Our key contribution lies in the systematic formulation and integration of resource interference dependencies (RIDs) and execution blocking dependencies (EBDs), which capture critical yet previously invisible performance interactions in microservices. Panorama employs hybrid attention mechanisms and meta-learning adaptation, operating as a closed-loop control system, enabling more accurate workload forecasting and proactive resource allocations. Extensive evaluation validates the effectiveness of Panorama, which shows superior performance compared to state-of-the-art baselines.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62402536), the Key Core and Pivotal Technology Research Project of the Hengqin Guangdong-Macao In-Depth Cooperation Zone under Grant 2430004045040, and the Guangdong Basic and Applied Basic Research Foundation under Grant 2026A1515011904 and 2025A1515010427.

Data Availability

The implementation and data of Panorama are available on Zenodo [7] and GitHub [8].

References

[1] AWS. 2022. AWS Auto Scaling Documentation. Retrieved May, 2025 from https://aws.amazon.com/autoscaling/

[2] Azure. 2022. Azure Autoscale. Retrieved May, 2025 from https://docs.microsoft.com/en-us/azure/azure-monitor/autoscale/

[3] Luciano Baresi, Davide Yi Xian Hu, Giovanni Quattrocchi, and Luca Terracciano. 2021. KOSMOS: Vertical and Horizontal Resource Autoscaling for Kubernetes. In *Service-Oriented Computing - 19th International Conference, ICSOC 2021, Virtual Event, November 22-25, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13121)*, Hakim Hacid, Odej Kao, Massimo Mecella, Naouel Moha, and Hye-young Paik (Eds.). Springer, 821–829. doi:10.1007/978-3-030-91431-8_59

[4] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. 2022. Cypress: input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, Ada Gavrilovska, Deniz Altinbüken, and Carsten Binnig (Eds.). ACM, 257–272. doi:10.1145/3542929.3563464

[5] Tao Chen and Rami Bahsoon. 2017. Self-Adaptive and Online QoS Modeling for Cloud-Based Software Services. *IEEE Trans. Software Eng.* 43, 5 (2017), 453–475. doi:10.1109/TSE.2016.2608826

[6] Tao Chen, Rami Bahsoon, and Xin Yao. 2018. A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems. *ACM Comput. Surv.* 51, 3 (2018), 61:1–61:40. doi:10.1145/3190507

[7] Zhuangbin Chen, Hongjiang Feng, Yang Liu, Juzheng Zheng, Xiaoyu He, and Zibin Zheng. 2026. *Panorama.* doi:10.5281/zenodo.21404801

[8] Zhuangbin Chen, Hongjiang Feng, Yang Liu, Juzheng Zheng, Xiaoyu He, and Zibin Zheng. 2026. *Panorama.* Retrieved August 03, 2026 from https://github.com/OpsPAI/Panorama

[9] Zhuangbin Chen, Yu Kang, Liqun Li, Xu Zhang, Hongyu Zhang, Hui Xu, Yangfan Zhou, Li Yang, Jeffrey Sun, Zhangwei Xu, Yingnong Dang, Feng Gao, Pu Zhao, Bo Qiao, Qingwei Lin, Dongmei Zhang, and Michael R. Lyu. 2020. Towards intelligent incident management: why we need it and how we make it. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1487–1497. doi:10.1145/3368089.3417055

[10] Zhuangbin Chen, Junsong Pu, and Zibin Zheng. 2025. Tracezip: Efficient Distributed Tracing via Trace Compression. *CoRR* abs/2502.06318 (2025). arXiv:2502.06318 doi:10.48550/ARXIV.2502.06318

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1285–1298. doi:10.1145/3133956.3134015

[12] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 1126–1135. http://proceedings.mlr.press/v70/finn17a.html

[13] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 135–151. doi:10.1145/3445814.3446700

[14] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Cloud and IoT Microservices. *CoRR* abs/1905.11055 (2019). arXiv:1905.11055 http://arxiv.org/abs/1905.11055

[15] Google. [n. d.]. GitHub - GoogleCloudPlatform/microservices-demo: Sample cloud-first application with 10 microservices showcasing Kubernetes, Istio, and gRPC. — github.com. https://github.com/GoogleCloudPlatform/microservices-demo.

[16] Google. 2022. Google Cloud Load Balancing and Scaling. Retrieved May, 2025 from https://cloud.google.com/load-balancing/docs/autoscaling

[17] Natasha Sarkar (Google). 2025. Kubernetes 1.35: In-Place Pod Resize Graduates to Stable. Retrieved February, 2026 from https://kubernetes.io/blog/2025/12/19/kubernetes-v1-35-in-place-pod-resize-ga/

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. doi:10.1109/CVPR.2016.90

[19] Kan Hu, Linfeng Wen, Minxian Xu, and Kejiang Ye. 2024. MSARS: A Meta-Learning and Reinforcement Learning Framework for SLO Resource Allocation and Adaptive Scaling for Microservices. In *IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2024, Kaifeng, China, October 30 - Nov. 2, 2024*. IEEE, 590–599. doi:10.1109/ISPA63168.2024.00081

[20] Yi Hu, Liangbo Hou, Junhui Hu, Mingyuan Ren, Menglan Hu, Chao Cai, and Kai Peng. 2025. Time-Varying Microservice Orchestration With Routing for Dynamic Call Graphs via Multi-Scale Deep Reinforcement Learning. *IEEE Trans. Serv. Comput.* 18, 5 (2025), 3276–3291. doi:10.1109/TSC.2025.3597631

[21] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q. Weinberger. 2016. Deep Networks with Stochastic Depth. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 9908)*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer, 646–661. doi:10.1007/978-3-319-46493-0_39

[22] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. 2023. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 419–432. https://www.usenix.org/conference/atc23/presentation/huye

[23] Dejun Jiang, Guillaume Pierre, and Chi-Hung Chi. 2010. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti (Eds.). ACM, 471–480. doi:10.1145/1772690.1772739

[24] Rouven Krebs. 2015. *Performance Isolation in Multi-Tenant Applications.* Ph. D. Dissertation. Karlsruhe Institute of Technology. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000049479

[25] Kubernetes. 2025. Horizontal Pod Autoscaling. Retrieved May, 2025 from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[26] Kubernetes. 2025. Vertical Pod Autoscaling. Retrieved February, 2026 from https://kubernetes.io/docs/concepts/workloads/autoscaling/vertical-pod-autoscale/

[27] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. 2022. Enjoy your observability: an industrial survey of microservice tracing and analysis. *Empir. Softw. Eng.* 27, 1 (2022), 25. doi:10.1007/S10664-021-10063-9

[28] Jianshu Liu, Shungeng Zhang, and Qingyang Wang. 2023. μConAdapter: Reinforcement Learning-based Fast Concurrency Adaptation for Microservices in Cloud. In *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*. ACM, 427–442. doi:10.1145/3620678.3624980

[29] Locust. [n. d.]. Locust.io — locust.io. https://locust.io/.

[30] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, 412–426. doi:10.1145/3472883.3487003

[31] Yang Luo, Mohan Gao, Zhemeng Yu, Haoyuan Ge, Xiaofeng Gao, Tengwei Cai, and Guihai Chen. 2024. Integrating System State into Spatio Temporal Graph Neural Network for Microservice Workload Prediction. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*, Ricardo Baeza-Yates and Francesco Bonchi (Eds.). ACM, 5521–5531. doi:10.1145/3637528.3671508

[32] Chunyang Meng, Shijie Song, Haogang Tong, Maolin Pan, and Yang Yu. 2023. DeepScaler: Holistic Autoscaling for Microservices Based on Spatiotemporal GNN with Adaptive Graph Learning. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 53–65. doi:10.1109/ASE56229.2023.00038

[33] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi Banerjee Dasgupta, and Subhrajit Bhattacharya. 2016. Anomaly Detection Using Program Control Flow Graph Mining From Execution Logs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 215–224. doi:10.1145/2939672.2939712

[34] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. 2013. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*, Jeffrey O. Kephart, Calton Pu, and Xiaoyun Zhu (Eds.). USENIX Association, 69–82. https://www.usenix.org/conference/icac13/technical-sessions/presentation/nguyen

[35] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. 2009. Automated control

of multiple virtualized resources. In *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, Wolfgang Schröder-Preikschat, John Wilkes, and Rebecca Isaacs (Eds.). ACM, 13–26. doi:10.1145/1519065.1519068

[36] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: a graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021*, Georg Carle and Jörg Ott (Eds.). ACM, 154–167. doi:10.1145/3485983.3494866

[37] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2024. Graph Neural Network-Based SLO-Aware Proactive Resource Autoscaling Framework for Microservices. *IEEE/ACM Trans. Netw.* 32, 4 (2024), 3331–3346. doi:10.1109/TNET.2024.3393427

[38] Xin Peng, Chenxi Zhang, Zhongyuan Zhao, Akasaka Isami, Xiaofeng Guo, and Yunna Cui. 2022. Trace analysis based microservice architecture measurement. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1589–1599. doi:10.1145/3540250.3558951

[39] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[40] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T. Kalbarczyk, Tamer Basar, and Ravishankar K. Iyer. 2023. AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 387–402. https://www.usenix.org/conference/atc23/presentation/qiu-haoran

[41] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 16:1–16:16. doi:10.1145/3342195.3387524

[42] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, Jeffrey S. Chase and Amr El Abbadi (Eds.). ACM, 5. doi:10.1145/2038916.2038921

[43] Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, and Minyi Guo. 2023. Nodens: Enabling Resource Efficient and Fast QoS Recovery of Dynamic Microservice Applications in Datacenters. In *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 403–417. https://www.usenix.org/conference/atc23/presentation/shi

[44] Jingwan Tong, Mingchang Wei, Maolin Pan, and Yang Yu. 2021. A Holistic Auto-Scaling Algorithm for Multi-Service Applications Based on Balanced Queuing Network. In *2021 IEEE International Conference on Web Services, ICWS 2021, Chicago, IL, USA, September 5-10, 2021*, Carl K. Chang, Ernesto Daminai, Jing Fan, Parisa Ghodous, Michael Maximilien, Zhongjie Wang, Robert Ward, and Jia Zhang (Eds.). IEEE, 531–540. doi:10.1109/ICWS53863.2021.00074

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[46] Dinh Dai Vu, Minh-Ngoc Tran, and Young-Han Kim. 2022. Predictive Hybrid Autoscaling for Containerized Applications. *IEEE Access* 10 (2022), 109768–109778. doi:10.1109/ACCESS.2022.3214985

[47] Hongbo Wang, Bo Deng, Jinyu Zhang, Siqi Wang, and Hong Zhu. 5555. A Dynamic Microservice Scheduling Model with Elastic Scalability and Adaptive Awareness . *IEEE Transactions on Cloud Computing* 01 (April 5555), 1–18. doi:10.1109/TCC.2026.3689123

[48] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, K. K. Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X. Liu. 2022. DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*, Ada Gavrilovska, Deniz Altinbüken, and Carsten Binnig (Eds.). ACM, 16–30. doi:10.1145/3542929.3563469

[49] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of Performance Issues in Microservices. In *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 1–9. doi:10.1109/NOMS47738.2020.9110353

[50] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. 2019. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 395–420. https://www.usenix.org/conference/nsdi19/presentation/wu

[51] Yichen Wu, Long-Kai Huang, Renzhen Wang, Deyu Meng, and Ying Wei. 2024. Meta Continual Learning Revisited: Implicitly Enhancing Online Hessian Approximation via Variance Reduction. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=TpD2aG1h0D

[52] Siqiao Xue, Chao Qu, Xiaoming Shi, Cong Liao, Shiyi Zhu, Xiaoyu Tan, Lintao Ma, Shiyu Wang, Shijun Wang, Yun Hu, Lei Lei, Yangfei Zheng, Jianguo Li, and James Zhang. 2022. A Meta Reinforcement Learning Approach for Predictive Autoscaling in the Cloud. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 4290–4299. doi:10.1145/3534678.3539063

[53] Xiao Yang, Sijia Li, Ke Wu, Zhijun Wang, Yuqi Tang, and Yueting Li. 2026. Adaptive Anomaly Detection in Microservice Systems via Meta-Learning. In *Proceedings of the 2026 International Conference on Artificial Intelligence and Control (CAIC '26)*. Association for Computing Machinery, New York, NY, USA, 653–660. doi:10.1145/3807246.3807348

[54] Guangba Yu, Pengfei Chen, Yufeng Li, Hongyang Chen, Xiaoyun Li, and Zibin Zheng. 2023. Nezha: Interpretable Fine-Grained Root Causes Analysis for Microservices on Multi-modal Observability Data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 553–565. doi:10.1145/3611643.3616249

[55] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 623–634. doi:10.1145/3510003.3510180

[56] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. 2023. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, Mahesh Balakrishnan and Manya Ghobadi (Eds.). USENIX Association, 321–339. https://www.usenix.org/conference/nsdi23/presentation/zhang-lei

[57] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 167–181. doi:10.1145/3445814.3446693

[58] Laiping Zhao, Yanan Yang, Yiming Li, Xian Zhou, and Keqiu Li. 2021. Understanding, predicting and scheduling serverless workloads under partial interference. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 22. doi:10.1145/3458817.3476215

[59] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Software Eng.* 47, 2 (2021), 243–260. doi:10.1109/TSE.2018.2887384

Received 2026-03-26; accepted 2026-06-18