# Proactive & Fine-Grained Monitoring for Microservice Call Chains in Cloud-Native Applications through Latency Distribution Prediction

**Hamza Hussain**

**A Thesis Submitted to the Faculty of Graduate Studies**

**In Partial Fulfillment of the Requirements**

**for the Degree of Master of Arts**

**Graduate Program in Information Systems and Technology**

**York University**

**Toronto, Ontario**

**October, 2024**

**Abstract**

Numerous algorithms have been proposed to predict performance metrics and detect anomalies in microservices-based cloud-native applications. Several performance prediction models use latency as an indicator of system performance, focusing on predicting mean end-to-end latencies of API requests. Such models often provide a single-point estimate of end-to-end latency across multiple APIs, limiting the insights drawn about system performance and restricting their usefulness for downstream tasks such as anomaly detection, root-cause analysis, and uncovering potential Service Level Agreement (SLA) violations. Conversely, anomaly detection models, while effective at identifying anomalous behavior, are reactive in nature and do not assess the performance implications of these anomalies, making it difficult to determine if they will lead to SLA violations. For instance, detecting high CPU usage does not indicate whether the performance degradation is significant enough to breach SLAs.

Modern cloud-native applications are distributed in nature and have their health monitored through multiple channels. In this study, we propose a singular novel approach that leverages multi-channel monitoring data for fine-grained performance analysis, proactive anomaly prediction, and root-cause analysis in microservices based applications. To this end, we employ Microservice Embeddings, Graph Neural Networks (GNN), and Gated Recurrent Units (GRU) to predict latency distribution, as opposed to a single latency value, for individual calls within a microservice call chain, as well as the distribution of end-to-end latency. Thus, our approach enables deeper insights into system performance and targeted diagnostics for anomalies. We use several benchmark datasets containing anomalies and show that our approach performs consistently across the latency spectrum while outperforming baseline latency prediction approaches by about 6%. Lastly, we show that our approach can be efficiently used to automate the process of trace-based anomaly prediction and perform root-cause analysis.

# Contents

# List of Tables

# List of Figures

**Declaration of Authorship**

I, Hamza Hussain, hereby declare that this thesis titled, "Proactive & Fine-Grained Performance Monitoring for Microservice Call Chains in Cloud-Native Applications through Latency Distribution Prediction", and the work presented herein are solely my own efforts. Parts of this thesis have formed the basis of a paper submitted to the 47th IEEE/ACM International Conference on Software Engineering (ICSE), Research Track, which is currently under review.

# 1 Introduction

A microservices system is typically a large-scale system composed of numerous instances, such as virtual machines or containers [1]. Each instance operates a small, specialized, and decoupled component of a larger software application. The correlations among these instances, including service invocations and resource contention, are often complex and dynamic [2], making monitoring and managing such a system challenging. Despite these complexities, the microservices architecture has gained significant popularity over the past decade due to its benefits for application development. These benefits include faster delivery, improved scalability, and greater autonomy [3]. The rise of cloud computing has further accelerated the adoption of microservices. This is because the cloud provides flexible resources and a pay-as-you-go model that aligns well with the decoupled nature of microservices.

However, relying on third-party cloud servers for data storage and processing requires careful monitoring of system reliability, stability, and performance to uphold performance-related Service Level Agreements (SLAs) [4]. This is because a typical microservices system may exhibit performance degradation due to resource contention across multiple instances [5]. Furthermore, anomalies, which are deviations from expected system behavior, can propagate among microservices, posing risks to system availability and integrity [6, 7]. These anomalies degrade system performance and lead to SLA breaches [8], which may also have significant financial repercussions for an organization. Traditionally, autoscaling has been used to ensure optimal performance and resource utilization in cloud applications [9]. However, traditional reactive autoscaling is not typically applicable in microservice-based applications due to highly variable and diverse workload patterns and complex interactions between microservices [9]. Hence, predictive performance estimation and anomaly detection have been extensively explored in literature to allow more responsive and efficient operation of microservices systems.

## 1.1 Performance Estimation in Microservices System

Performance estimation in microservices systems has already been explored in works related to distributed tracing [10, 11, 12], root cause analysis [13, 14, 15], and resource allocation [16, 17, 18, 19]. Latency, the time delay between a microservices system receiving an API request and completing its execution, is often used as a primary indicator of system performance [20, 21, 22] in these works. However, existing latency prediction models including Queuing Network Models, Regression and Machine learning models suffer from accuracy degradation outside narrow operational zones [23]. Whereas, in real systems, operational zones are changing frequently and there are large variations in latencies that can range from a few milliseconds to thousands of milliseconds across different APIs. Furthermore, current models often predict mean latency across all APIs [24, 25, 26], or focus on estimating end-to-end mean latency of an API [27]. This limits the insights that can be drawn about the overall system performance, thereby reducing the usefulness of these models for other downstream tasks such as anomaly detection, root-cause analysis and resource allocation.

## 1.2 Anomaly Detection in Microservices System

Several automated anomaly detection approaches have also been developed to improve reliability of microservices system. Techniques such as statistical methods, clustering, and rule-based systems have been historically employed for automated anomaly detection [28, 29, 30, 31, 32, 33, 34, 35]. Recently, several machine learning models have also been utilized to improve the accuracy of anomaly detection models [36, 29, 37, 38]. However, current anomaly detection models do not assess the impact of anomalies on the performance of a microservices system which limits the usefulness of these models. For instance, detecting high CPU usage doesn't indicate whether the performance

degradation is significant enough to breach SLAs. Thus, anomaly detection alone is not enough for a comprehensive analysis of the state/performance of a microservices system. Furthermore, current anomaly detection models, such as the ones proposed in [39, 40, 41], are reactive and focus on detecting anomalies rather than predicting them before they occur. Hence, a predictive approach is required so that measures can be taken pro-actively to improve reliability of microservices system.

## 1.3 Motivation

Due to the limitations of current performance prediction and anomaly detection models, a new consolidated framework is required to streamline the operations of a microservices system. This new framework should allow for a more accurate and fine-grained approach to automated performance prediction and provide helpful insights during the operation of a microservices system, including periods of anomalies. Thus, our motivation is to devise an approach that integrates performance prediction, pro-active anomaly prediction, and root-cause analysis in a single framework, allowing for comprehensive, fine-grained analysis of system state and performance.

## 1.4 Research Scope

Given that latency is the primary metric for performance measurement, benchmarking, and setting SLAs [20, 21, 22], in this study, we focus on predicting the latency distribution of individual API requests instead of a single mean latency value. Additionally, we predict the latency distribution of individual calls within the microservices call chain of an API request, alongside end-to-end latencies. Predicting the latency distribution, rather than a single latency value, provides valuable insights that facilitate tasks such as automated anomaly detection. Furthermore, predicting latency distribution at the level of individual calls within a microservices call chain enables targeted diagnostics

during anomalies and facilitate automating root-cause analysis. Thus, the scope of this research can be summarized as follows:

- **Granular latency distribution prediction** at the level of an API request and individual microservice calls involved in an API request, as a means for comprehensive performance monitoring of a microservices system.
- **Proactive anomaly detection** by utilizing predicted latency distributions to gauge the anomaly status of a trace.
- **Root-cause localization** by leveraging predicted latency distributions to identify the possible root cause of an anomalous trace.
- **Single Unified Framework** development for addressing the above tasks.

## 1.5 Approach & Contribution

We propose a novel approach that employs Microservice Embeddings, Graph Neural Networks (GNNs) and Gated Recurrent Units (GRUs) to forecast the latency distribution of individual microservice calls in diverse API requests based on current system parameters. To this end, we utilize traces, where all microservice invocations associated with the same API request constitute a single trace [40]. We represent each trace as an individual graph, where nodes represent microservices part of the trace, and each microservice call in the trace forms an edge in the graph. A specific API request is therefore represented as a span graph [27] in our framework. The GNN processes the graph, while the GRU unit processes the temporal ordering of the microservice calls (edges) in the graph. The predicted latency is then used by subsequent model layers in our framework to predict trace based anomalies and perform root-cause localization. Hence, we make the following contributions:

- We present a single unified framework for proactive performance prediction, anomaly prediction, and root-cause analysis in microservices based applications.
- We adopt the probabilistic approach for latency prediction, in contrast to single-point mean estimates, thereby capturing the latency distribution.
- We present *PerformanceLENS*: A new unified performance model using Microservice Embeddings, GNNs, and GRUs for accurate latency distribution prediction at a granular level of individual API requests.
- A novel approach to modeling temporal orderings of calls in a microservice call chain of an API request using GRU. This enables latency distribution prediction of individual calls part of the microservice call chain, on top of the end-to-end latency distribution.
- Evaluation of our approach across a broad spectrum of user requests. We devise a unified model to accommodate traces spanning a broad spectrum of latency regions. We empirically show that our model surpasses state-of-the-art performance prediction methods by around 6%.
- Experimentation with benchmark datasets for anomaly detection to enable detection of potential system issues and root-cause localization via latency distribution analysis.
- We empirically show that our framework can be used to match state-of-the art trace-based anomaly detection approaches, while simultaneously providing performance predictions.

## 1.6 Thesis Organization

The remainder of this thesis is organized as follows: We start by reviewing related works in the areas of latency prediction and anomaly detection in Chapter 2. In Chapter 3, we discuss the design considerations taken into account while formulating *PerformanceLENS* and formalize the problem definition for latency distribution prediction. We also provide details of the *PerformanceLENS* model for predicting latency distribution in the same chapter. We then discuss the integrated framework, incorporating *PerformanceLENS*, for trace-based anomaly prediction and root-cause localization in Chapter 4. We formulate research questions and present the details of our experiments in Chapter 5. Chapter 6 presents an in-depth analysis of the findings from our experiments, followed by a discussion of threats to validity that must be considered when interpreting the results. Lastly, we conclude with future research directions in Chapter 7.

# 2 Literature Review

In this chapter, we review state-of-the-art works on exploring performance prediction and anomaly detection in microservices systems and their limitations. We also individually discuss a few notable works in these areas that serve as baselines for our experiments.

## 2.1 Performance Prediction

Various models have been proposed in the literature for representing the microservices architecture [24, 25, 27, 26, 42, 43]. For instance, ATOM [43] and Kraken [26] use the Layered Queuing Network (LQN) and Variable Order Markov Model (VOMM) to predict the workload (i.e., call rate) for each microservice component and then estimate the overall application performance. These models assume that latency follows a simple function with respect to the workload and resource configuration [27]. However, these assumptions do not always hold and lead to high estimation errors for latency.

Consequently, another body of work investigates machine learning models to improve the accuracy of performance prediction [24, 25, 27, 44, 45]. Some works leverage CNN with boosted trees or LSTM to predict end-to-end latency under different workloads [44, 45]. However, these models ignore the dependency between microservices. Works leveraging GNNs overcome the limitation of modeling microservice dependencies [24, 27] and, therefore, typically perform better than traditional machine learning approaches.

## 2.2 Performance Prediction Baselines

Works that utilize GNNs to model data from microservices systems are closely related to our approach and, therefore, are suitable for comparison with our model. In particular, we discuss two such works—GRAF and PERT-GNN [24, 27]—which also serve as

baselines for our experiments.

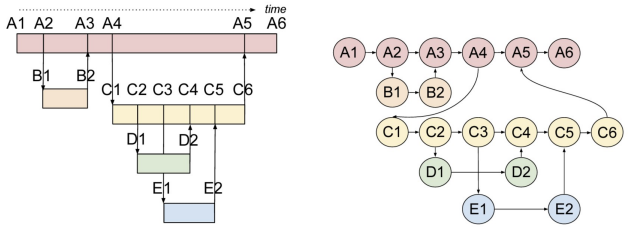
### 2.2.1 GRAF: Optimizing Resource Allocation through Latency Prediction

Park et al. introduced GRAF [24], a framework to optimize resource allocation in microservices systems to prevent SLA breaches. GRAF focuses on CPU allocation, leveraging latency prediction as an intermediate step to check for potential SLA breaches under the current CPU allocation values. GRAF then uses a second module to optimize CPU allocation by checking its impact on predicted latency. The latency prediction module in GRAF utilizes GNN. The graph is formulated based on APIs and their execution histories. Park et al. note that the execution chain of an API varies under varying load conditions. To account for this variability, they merge all traces in a certain time window, forming a graph wherein nodes represent microservices and edges represent the invocations between them (i.e., a Microservice Call Graph (MCG)). This allows the prediction of mean latency across all API requests. However, since a single request only invokes some paths and components across the MCG according to the business logic [27], the remaining components introduce noise in the model, limiting the accuracy of this approach for predicting the latency of individual requests. In addition, GRAF does not incorporate other metrics except CPU utilization, further limiting its predictive power.

### 2.2.2 PERT-GNN: Preserving Temporal Ordering of a Microservice Call Chain for Latency Prediction

Tam et al. introduced PERT-GNN in [27] to account for temporal dependencies between microservices for specific APIs. Inspired by the program evaluation and review technique (PERT) outlined in [46], which models the tasks necessary for project completion, Tam et al. introduced a new way of constructing graphs. Figure 1 demonstrates

the creation of a PERT graph from a span graph.




**(a) Span Data** **(b) The final PERT Graph.**

Figure 1: PERT Graph Construction

Tam et al. use CPU utilization and memory usage as node features in PERT graphs, where nodes represent the state of individual microservices represented through node features across different stages of a trace. PERT-GNN then aggregates different runtime behaviors of APIs to predict the mean end-to-end latency of specific APIs. However, constructing such a graph necessitates sampling metric values for each microservice whenever another microservice invokes it, it invokes another microservice, or it concludes the execution of a requisite task. Additionally, the process of constructing PERT graphs may prove impractical in live systems. For lengthy traces, the size of the PERT graph may increase dramatically, diminishing the benefits of preserving temporal ordering compared to the overhead required to construct such graphs in production environments. Lastly, due to the aggregation process, PERT graphs do not capture the nuances of individual API requests required for traced-based latency prediction.

## 2.3 Anomaly Detection

Several different methods have been devised to represent anomalous behavior in microservices applications. Some works focus on system-level anomalies, defining them as deviations from normal values of metrics measured across the entire system [4, 47, 48].

These anomalies often manifest in metrics such as the number of 500 errors observed during a specific time window. Another approach targets anomalies at the level of individual microservices, observing whether metrics like CPU usage exhibit abnormal behavior [1]. Lastly, trace-based approaches concentrate on individual traces, defining anomalies as latent errors within specific traces [40, 39, 49, 50, 51, 41].

Regardless of how anomalies are represented, due to the limitations of models relying on a single data source, such as metrics or traces, approaches like Anofusion [1] and Hades [47] focus on fusing multi-modal data such as logs and metrics for accurate anomaly detection. Anofusion works with logs, metrics, and traces data at the level of individual microservices, training separate models to detect deviations from the normal behavior of each microservice. In contrast, Hades utilizes logs and metrics at a system level, labeling the overall system behavior as either normal or anomalous.

To improve upon the performance of these works, another body of research focuses on using Graph Neural Networks (GNNs) combined with multi-modal data to enhance the accuracy of anomaly detection in microservices-based applications [48, 4]. These approaches construct microservice call graphs based on traces passing through the system within a specific time window to detect system-wide anomalies and pinpoint the root-cause microservice responsible for such anomalies.

However, trace-based anomaly detection approaches provide a more practical definition of anomalies by focusing on latent errors present in traces [40, 39, 49, 50, 51, 41]. These approaches often employ multiple models for both anomaly detection and root-cause localization [40, 39]. Initially, traces are analyzed, and a prediction model determines whether a trace exhibits a latent error. Subsequently, a second model is used to pinpoint the root-cause microservice of the detected errors.

## 2.4 Anomaly Prediction & Root-Cause Localization Baselines

Since our work focuses on performance analysis of traces, these trace-based approaches serve as baselines for our experiments on anomaly prediction and root-cause analysis. We discuss the two best-performing works in this area in more detail below.

### 2.4.1 MEPFL: Latent error prediction and fault localization for microservice applications by learning from system trace logs

Zhou et al. presented MEPFL, a supervised learning approach designed to detect latent errors in traces and identify their root causes in [39]. MEPFL employs a strategy that utilizes three models in tandem: the first model detects whether a trace exhibits latent errors, while the other two models identify the type of fault and the root cause of the fault. To achieve this, Zhou et al. define features at different levels, such as microservice instance, trace, or the microservice itself, for each microservice involved in the trace. These features are categorized into configuration, resource, instance, and interaction, and are collected from trace logs in real-time to detect system issues. Zhou et al. then test their method using a variety of machine learning models, including Random Forest and KNN. However, this approach is reactive, focusing on detecting latent errors after they occur rather than predicting them before they manifest in a request. It also requires extensive monitoring of all requests passing through the system in real-time for live error detection and root-cause analysis. Additionally, MEPFL does not address system performance analysis for potential SLA breaches.

### 2.4.2 TraceRCA: Practical Root Cause Localization for Microservice Systems via Trace Analysis

To overcome the limitations associated with supervised learning approaches like MEPFL, Li et al. introduced TraceRCA in [40], an unsupervised method for detecting trace-

based anomalies and identifying their root causes. TraceRCA calculates historical averages of features such as CPU usage, memory usage, and HTTP status for microservice invocations, and then determines if the current metric values deviate from these averages by more than a predefined threshold. When a deviation is detected, the invocation is marked as anomalous with respect to the specific feature. If an invocation is found to be abnormal in any feature, the entire trace associated with that invocation is flagged as abnormal. Once abnormal traces are identified, TraceRCA determines root-cause microservices based on the principle that the proportion of abnormal traces passing through a faulty microservice will be higher than through normal ones. Using this principle, TraceRCA calculates anomaly scores for each microservice and ranks them for root-cause localization. However, like MEPFL, TraceRCA is reactive, focusing on detecting latent errors after they occur rather than predicting them before they impact a request. Similarly, TraceRCA also requires real-time analysis of all traces passing through the system for live anomaly detection.

## 2.5 Advancing the State of Performance and Anomaly Prediction Models

We address the limitations of the aforementioned work and improve their performance in several ways. For performance prediction, we propose *PerformanceLENS* , that leverages an arbitrary number of metrics that may be available through monitoring channels of a microservices system in its training pipeline. Secondly, to overcome the limitations of MCGs, *PerformanceLENS* considers each trace as a separate graph, as proposed in DeepRest [52], and uses GNN to understand the dependencies between microservices part of an individual trace. *PerformanceLENS* also uses microservice embeddings to capture relationships between different microservices on a system level, encoding their identities to better understand individual APIs. Moreover, *PerformanceLENS* intro-

duces a novel and more practical method to capture the temporal dependencies in a microservices call chain using GRU. Leveraging the GRU also allows *PerformanceLENS* to introduce a novel feature for predicting latency distribution of all individual microservice calls (edges) involved in an API request, in addition to predicting the distribution of end-to-end latency. This allows for fine-grained analysis of system performance at trace and edge levels. Lastly, *PerformanceLENS* moves away from single-point estimates to predicting the distribution of latencies.

These capabilities uniquely position *PerformanceLENS* for use with anomalous data and for automating other downstream tasks such as anomaly detection and root-cause analysis. By integrating *PerformanceLENS* into a unified predictive framework, we move away from reactive detection to proactive prediction of trace-based anomalies. Our framework predicts the performance of a pre-configured trace before the system has received it, based on the system's current state inferred from current metric values. This prediction allows us to assess whether the trace will be anomalous and pinpoint the root cause in advance. This proactive approach, combined with *PerformanceLENS* 's novel features for performance prediction, allows our framework to advance the problem domain.

# 3 Performance Prediction

In this chapter we discuss different design considerations that must be taken into account while formulating a latency distribution prediction model. Subsequently, we formulate the problem of latency predictions based on these considerations and discuss the details of the performance prediction part of our framework. We start with the design considerations and then present the details of our proposed performance prediction model, *PerformanceLENS*, its components, and corresponding processing steps.

## 3.1 Design Considerations

To effectively design a framework for predicting the latency distribution of each microservice call involved in an API request and end-to-end latency distribution, we consider various factors while formulating *PerformanceLENS*. These considerations lead us to formulate the problem of predicting the latency distribution as a supervised graph regression problem. In this section we discuss each of these considerations in more detail:

1. **Underlying Conditional Probability Distribution.** Latency distribution can be predicted using two approaches: *The Parametric Approach:* This method assumes that the underlying latency distribution follows a specific parametric probability distribution. In this scenario, the model can be trained to predict the parameters of this distribution. The second method is the *Non-Parametric Approach*, which involves training the model to predict multiple points in the latency distribution, thereby avoiding any assumptions about the distribution itself. *PerformanceLENS* uses the latter approach.

2. **Granularity of Estimates.** To understand the shape of the underlying distribution, it is essential to predict points in the distribution at granular intervals.

There are multiple options available for the loss function for single-point estimates, including Mean Absolute Error (MAE), Mean Squared Error (MSE), and Mean Absolute Percentage Error (MAPE). A minimization problem involving each corresponds to predicting a different point in the distribution. However, it is challenging to interpret the exact position of these points on the latency distribution, except for MAE, which can be shown to predict the 50th quantile. In contrast, quantile loss, as proposed in literature [53], allows for predicting a specific quantile on the distribution. *PerformanceLENS* leverages multi-quantile loss, formulated using quantile loss, to train our model to predict an arbitrary number of quantiles on the latency distribution.

3. **Performance Variability.** In a microservices system, end-to-end latency can range from a few milliseconds to several thousand milliseconds across different APIs. The issue is further exacerbated when anomalous data are considered where observed latency spikes. Therefore, *PerformanceLENS* uses the log scale to compress the scale of predictions.

4. **Unstructured Data.** To predict the performance of an API request in a microservices system, it is essential to consider the microservices involved, their metrics at the time of prediction, and the order in which these microservices are invoked. This data is inherently unstructured due to the various microservices involved in API requests. Therefore, a unified model architecture is needed to effectively handle the unstructured microservices data while simultaneously accounting for relationships between microservices and the temporal ordering of microservice calls. *PerformanceLENS* utilizes microservice embeddings and a hybrid GNN/-GRU architecture to address this challenge. Microservice Embeddings capture the relationships between microservices at a system level, and the GNN unit pro-

cesses the unstructured microservices data to account for intra-trace dependencies between microservices. While the GRU unit processes the temporal aspect of the microservice call chain. Furthermore, the metric values used as predictor variables can have different measurement scales. Therefore, *PerformanceLENS* uses different transformation schemes to train the model components effectively.

### 3.1.1 Problem Formulation

Based on the above considerations, we formulate the problem of predicting a certain quantile latency as a supervised graph quantile regression problem, represented as follows:

$$\hat{y}_\tau := f_\tau(G, X_{t,G}) \tag{1}$$

where $G$ represents the graph associated with a single API request the microservices system receives. $G$ is constructed by using microservices involved in responding to the API request as the graph nodes and invocations between these microservices as edges. $X_{t,G}$ represents the metrics, or resource consumption, of all microservices part of $G$ at time $t$. Then, $\hat{y}_\tau$ is the predicted quantile ($\tau$) latency at time $t$ for the API after the function $f_\tau$ is applied to the graph and resource values. *PerformanceLENS* is then trained to simultaneously predict an arbitrary number of quantiles based on the above formulation.

## 3.2 PerformanceLENS

*PerformanceLENS* is a performance prediction model designed to predict the latency distribution of individual microservice calls part of the microservice call chain of an API request, on top of end-to-end latency distribution, within a microservices system. The primary inputs to *PerformanceLENS* are metrics and trace data extracted from

the microservices system. The framework's output is the latency distribution given the current metric values of the microservices part of a single API request. The distribution is captured by predicting an arbitrary number of quantiles on the latency distribution. The traces extracted from the microservices system define the graph's topology in the *PerformanceLENS* model, where nodes represent individual microservices. The node features are constructed by concatenating the metrics values of the microservices with the microservice embeddings. The formed graph is then fed into the GNN, the output of which is used to construct edge embeddings and fed to the GRU in sequence. The output of the GRU is then read through the model read-out layer. Figure 2 provides a visual representation of the *PerformanceLENS* framework, which is discussed in more detail in the subsequent subsections.

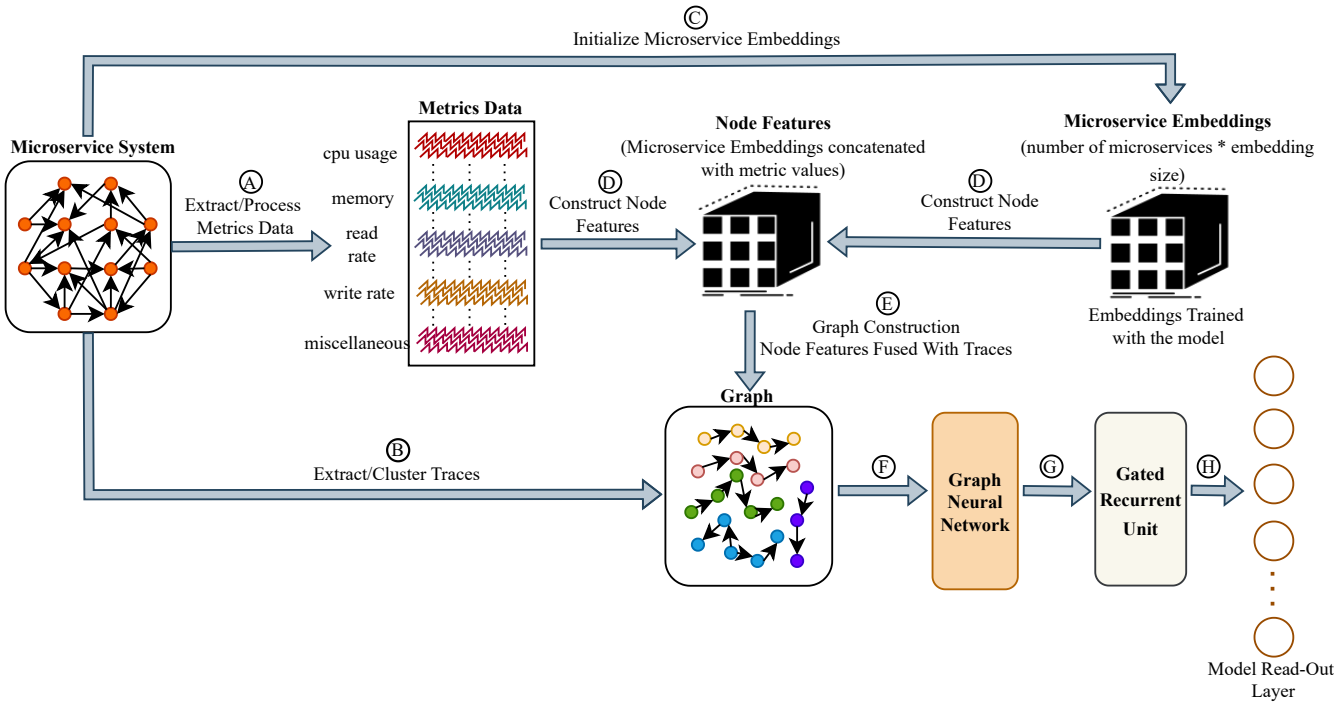



Figure 2: End-to-end illustration of *PerformanceLENS* model including data extraction, graph construction, and model layers

### 3.2.1 Metrics Data Processing

Metrics represent monitoring data collected from the system's monitoring pipeline, such as CPU usage, memory usage, etc. We employ multiple processing steps to extract the useful features from this data for training *PerformanceLENS* . In what follows, we discuss each data processing step.

- **Feature Engineering:** We create new features to represent the performance of a microservice relative to its historical average performance. For each metric present in the data, we calculate the value of the corresponding engineered feature $f$, for each individual microservice $m \in M$, where $M$ is the set of all microservices, as follows:

$$f = (x_{t,m} - \mu_m)/\sigma_m \tag{2}$$

  Here, $x_{t,m}$ represents the metric value at time $t$ of microservice $m \in M$ and the value of the new feature $f$ is the number of standard deviations metric value at time $t$ deviate from the historical mean $\mu_m$ of the metric for microservice $m$, where $\sigma_m$ represents the standard deviation of metric value for the respective microservice. This helps the model uncover deviations from the normal behavior of each microservice.

- **DataTransformation:** We also apply different transformation schemes to manage the variables' scale effectively. For the model input variables, we use the standard normal transformation, expressed as:

$$x_{std} = (x_i - \mu)/\sigma \tag{3}$$

  where $x_{std}$ represents the number of standard deviations a particular data point $x_i$ lies from the mean $\mu$ of the feature across all data points, and $\sigma$ represents

the standard deviation of the feature across all data points. To manage the wide spectrum of latency values ranging from less than one millisecond to thousands of milliseconds, we use the logarithmic scale, essentially reducing the scale of the target variable to an approximate range of -1 to 5. The transformed target variable is given by the equation:

$$y_{\text{target}} = \log_{10} y \tag{4}$$

### 3.2.2 Trace Clustering

Traces denote the sequence of microservice calls generated due to a single API request where each call is represented through pairs of microservices involved in the call. We cluster traces by hashing the list of microservice pairs that are part of a trace. Hashing allows the assigning of integers to traces based on the microservice pairs involved, and all traces resulting in the same integer after hashing can be considered part of the same cluster. This approach allows us to use historical performance values of an edge belonging to a trace cluster as edge features.

### 3.2.3 Graph Construction

An API request in a microservice system involves multiple microservices, each impacting the request's latency. Such requests can be represented as directed graphs, with microservices as nodes and each invocation as an edge between pairs of microservices directed from the upstream microservice to the downstream microservice. This representation allows accurate prediction of the time required to fulfill a request by considering the metric values of all involved microservices. The sequence in which these microservices are invoked is also incorporated in a directed graph. This is important as different requests may involve the same microservices but in different orders. We

use trace data to construct graphs and consider each request distinct. Traces form the graph's topology, and nodes represent the microservices part of the trace.

### 3.2.4 Microservice Embeddings

Bengio et al. introduced the concept of word embeddings to train language models in [54]. The key idea introduced was to use an embedding layer to obtain word representations to feed other network layers. Word embeddings can accurately capture semantic and syntactic relationships between words [55]. We adapt this concept of word embeddings in the context of a microservices system to capture relationships between microservices at a system level. We introduce an embedding layer on top of our GNN to accomplish this. While GNNs are useful in capturing relationships between microservices within a single trace, trained embeddings capture relationships between all microservices parts of the system. This also implicitly helps make the GNN aware of the API call that generated the trace through the identities of involved microservices. The microservice embeddings are initialized based on the number of microservices present in the system and trained with the rest of the model.

### 3.2.5 Node Features

The node features in *PerformanceLENS* graph consist of two parts. The first part includes normalized and engineered metrics data. The second part is derived from the microservice embedding of the respective microservice. These two components are concatenated to form the complete node features.

### 3.2.6 The Prediction Layers

The prediction layers in *PerformanceLENS* are composed of a single embedding layer on top of three graph convolution layers, followed by a fully connected layer to construct

edge embeddings. The edge embeddings are then processed sequentially by a GRU cell with single update and reset gates. We use the GeLU activation function for the graph convolutional layers and ReLU for the GRU gates. We train the model using Adam optimizer and multi-quantile loss function given by equations (13) and (14). Figure 3 provides a zoomed view of the *PerformanceLENS* model layesrs. Below, we discuss the GNN and GRU layers in more detail.

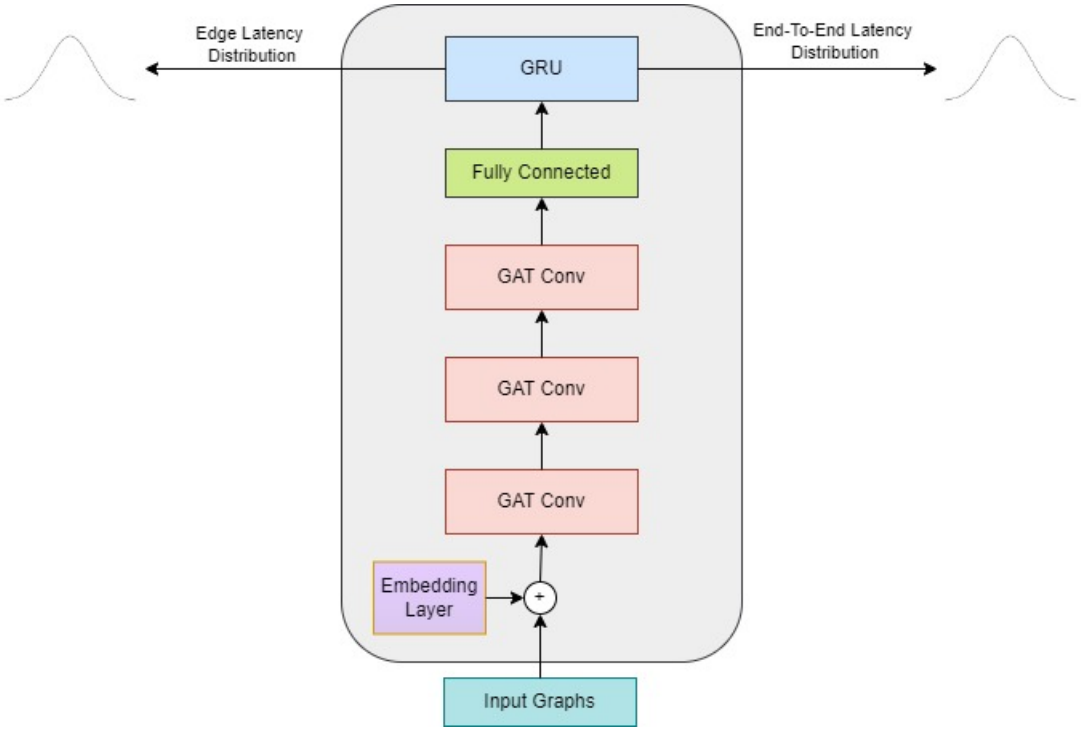



Figure 3: Model Layers for *PerformanceLENS*

- **Graph Neural Networks (GNN) layers:** *PerformanceLENS* employs a GNN with Graph Attention Networks (GAT) convolution layers, as proposed in [56], to process traces. GAT enables specifying different weights to different nodes [56]. This allows *PerformanceLENS* to understand the importance of each microservice in a trace for latency prediction based on the metric values of the involved microservices.

- **Gated Recurrent Units (GRU) layers:** To utilize the temporal ordering of calls in an API request, *PerformanceLENS* employs a Gated Recurrent Unit (GRU). The input to the GRU are edge embeddings, which are derived from the output of the GNN. We concatenate the node embeddings from the GNN output corresponding to the nodes in a particular edge to construct these edge embeddings. We then use the invocation times of the microservices to sequence the edges, feeding the edge invoked last into the GRU first. The GRU predicts the latency distribution of the relevant edge at each step. Since the latency of the current edge encompasses the latency of all previous edges, the final output of the GRU represents the end-to-end latency of the trace.

### 3.2.7 Model Read-out Layer

Models giving single-point estimates for latency typically employ a single neuron in the output layer of the neural network. Multiple neurons need to be added to the output layer to predict multiple points on the latency distribution, with each neuron predicting a specific point on the distribution. In *PerformanceLENS* , each neuron in the output layer predicts a specific quantile on the latency distribution. This approach allows the model to be trained to predict an arbitrary number of quantiles on the latency distribution. The number of quantiles to be predicted depends on the use case and the granularity required for estimating the latency distribution.

We sample the quantiles to be estimated from a standard normal distribution. This is done as we expect the latency distribution to resemble the normal distribution around the median, with most of the probability density concentrated near the median. As an example, approximately 40 percent of the probability density lies within 0.5 standard deviations on either side of the median of standard normal distribution. Figure 4 illustrates these high-density regions of a normal distribution.

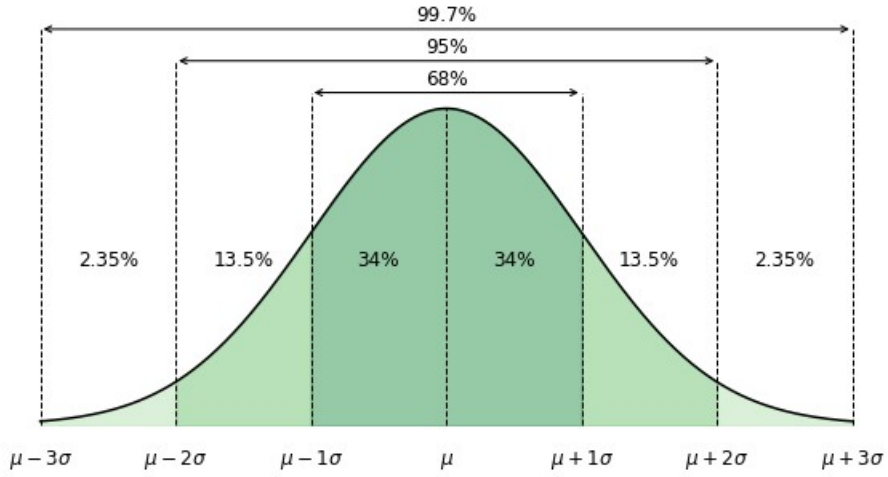



Figure 4: High Density Regions of Standard Normal Distribution

To capture this high-density region more accurately, we predict more quantiles from this area than from the tail ends of the standard normal distribution, thereby biasing our model towards more accurate predictions in this region. Table 1 provides details of the specific quantiles that we predict. It is to be noted that this does not translate to a normality assumption for the probability distribution of latencies.

| **Standard Deviations** | **Quantiles (%)** | | **Region Density** |
|---|---|---|---|
| | **Lower Quantiles** | **Upper Quantiles** | |
| **3.000** | 0.13 | 99.87% | Low |
| **2.500** | 0.62 | 99.38 | Low |
| **2.000** | 2.28 | 97.72 | Low |
| **1.500** | 6.68 | 93.32 | Low |
| **1.000** | 15.87 | 84.13 | Low |
| **0.750** | 22.66 | 77.34 | Low |
| **0.500** | 30.85 | 69.15 | High |
| **0.375** | 35.39 | 64.62 | High |
| **0.250** | 40.13 | 59.87 | High |
| **0.125** | 45.03 | 54.98 | High |
| **0.000** | 50.00 | 50.00 | - |

Table 1: Sampled Points from Standard Normal Distribution

### 3.2.8 Penalizing Distribution Estimates

Given the output of a single neuron in the read-out layer as $\hat{y}_\tau$, we use quantile regression as formulated in (1), to model the output of the neuron. For a single neuron output layer, then the choice of learning target is the minimization of quantile loss, proposed in [53], and represented by the equations below:

$$\sum_{i=0}^{n} \tau \cdot |y_i - \hat{y}_i| \qquad \text{for} \qquad y_i \geq \hat{y}_i, \tag{5}$$

$$\sum_{i=0}^{n} (1-\tau) \cdot |y_i - \hat{y}_i| \qquad \text{for} \qquad y_i < \hat{y}_i, \tag{6}$$

It can be shown that quantile loss can be used to predict conditional quantile [53] of the distribution given the API request and resource utilization. For brevity, we keep the proof to a high level in this thesis; a more rigorous mathematical proof can be found in [53]. As a first step, we rewrite the quantile loss, introduced in (5) and (6), as represented in the following equations:

$$\sum_{i=0}^{n} \tau \cdot (y_i) - \tau \cdot (\hat{y}_i) \qquad \text{for} \qquad y_i \geq \hat{y}_i, \tag{7}$$

$$\sum_{i=0}^{n} (1-\tau) \cdot \hat{y}_i - (1-\tau) \cdot y_i \qquad \text{for} \qquad y_i < \hat{y}_i, \tag{8}$$

Taking derivative of (7) and (8) with respect to $\hat{y}_i$ we get the following:

$$\sum_{i=0}^{n} -\tau \qquad \text{for} \qquad y_i \geq \hat{y}_i, \tag{9}$$

$$\sum_{i=0}^{n} (1-\tau) \qquad \text{for} \qquad y_i < \hat{y}_i, \tag{10}$$

Equating the (9) and (10) to zero and assuming that $\alpha$ represents the number of ob-

servations satisfying the constraint $y_i < \hat{y}_i$ out of $n$ total observations, we have the following:

$$-\tau \cdot (n - \alpha) + (1 - \tau) \cdot \alpha = 0 \tag{11}$$

$$\tau = \frac{\alpha}{n} \tag{12}$$

Equation (12) proves that in order to minimize quantile loss for a particular quantile $\tau$, the proportion of observations lower than the predicted value should be equivalent to $\tau$. Thus, a single neuron can be used to predict a particular point in the latency distribution. For our experiments, we introduce multiple neurons in the readout layer of the model, with each neuron predicting a particular quantile. We then penalize each neuron using the quantile loss based on the corresponding quantile that the neuron is supposed to predict. We leverage multi-quantile loss, formulated from quantile loss, as the overall loss function of the model. The formula for multi-quantile loss is as follows, where $n$ represents the number of data instances in a batch of inputs, and $k$ represents the number of quantiles being predicted by the model, as provided in Table 1.

$$\sum_{i=0}^{n} \sum_{j=0}^{k} \tau_j \cdot |y_i - \hat{y}_{ij}| \qquad \text{for} \qquad y_i \geq \hat{y}_{ij}, \tag{13}$$

$$\sum_{i=0}^{n} \sum_{j=0}^{k} (1 - \tau_j) \cdot |y_i - \hat{y}_{ij}| \qquad \text{for} \qquad y_i < \hat{y}_{ij}, \tag{14}$$

$$\tag{15}$$

# 4 Anomaly Prediction & Root Cause Localization

In this chapter we discuss how we integrate anomaly prediction and root-cause analysis modules with *PerformanceLENS* to create a single unified framework capable of predicting performance of traces, predict trace-based anomalies, and root-cause localization.

## 4.1 Anomaly Prediction

Since our focus is on predicting trace-based anomalies, and each trace constitutes a single graph in our approach, the task of anomaly prediction is inherently a graph-level task. Therefore, the anomaly prediction model should incorporate the end-to-end latency distribution output from *PerformanceLENS* alongside other trace-based features. Figure 5 illustrates the architecture for integrating *PerformanceLENS* with the anomaly prediction model within our framework. The inputs to the anomaly prediction model include the end-to-end latency distribution output of the trace from *PerformanceLENS* and the graph embedding of the trace. The graph embedding is constructed by applying a pooling operator to the node embeddings of the trace, which are the outputs of the last graph convolutional layer in *PerformanceLENS* .

The output of the anomaly prediction model consists of anomaly scores for each trace. These anomaly scores are penalized using cross-entropy loss, as defined by equation (16), where $y_i$ represents the true label for the trace, $\hat{y_{ia}}$ represents the score indicating the trace is anomalous, and $\hat{y_{in}}$ represents the score indicating the trace is normal.

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log(\frac{\hat{y_{ia}}}{\hat{y_{ia}} + \hat{y_{in}}}) \tag{16}$$

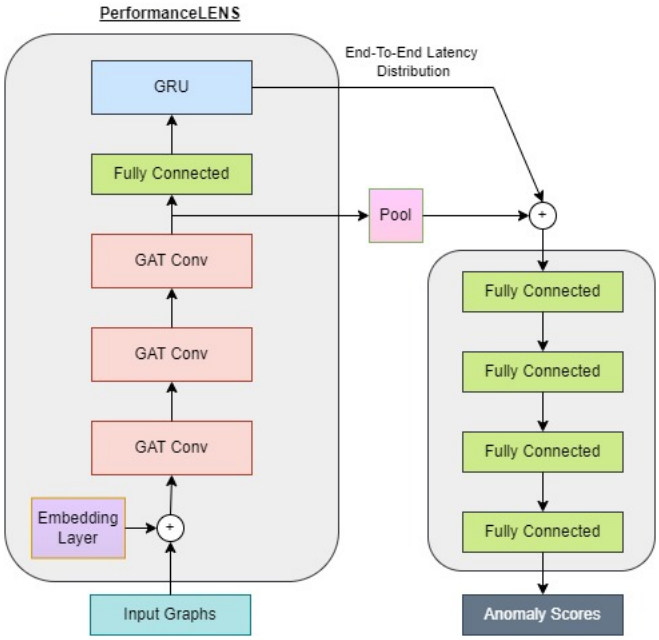



Figure 5: End-to-end illustration for integrating *PerformanceLENS* with anomaly prediction model.

## 4.2 Root-Cause Localization

The task of root-cause localization is a node-level task that requires incorporating the latency distribution information of individual edges within a trace to accurately infer the root-cause node of an anomaly. Additionally, it is crucial to capture information related to the individual nodes that are part of the trace. These relationships are best captured using a GNN; therefore, we construct a new graph for root-cause localization. The topology of this graph is based on the trace being processed by *PerformanceLENS* , with node features defined by the node embeddings output by the last graph convolutional layer. Edge features are then defined as the latency distribution of individual edges produced by *PerformanceLENS* . Figure 6 illustrates the architecture for integrating a root-cause localization model with *PerformanceLENS* and the anomaly prediction model.

The newly constructed graph is then passed through multiple graph convolutional lay-

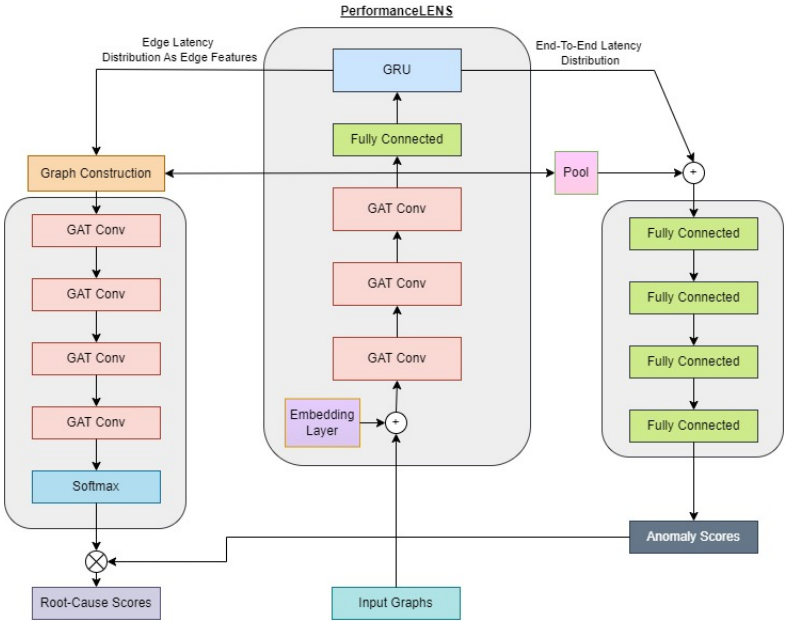



Figure 6: End-to-end illustration for integrating *PerformanceLENS* with anomaly prediction and root-cause localization models.

ers. Finally, a softmax operation, as defined in equation (17), is applied by grouping the nodes in each graph separately. In this equation, $K$ represents the number of nodes (microservices) in an individual trace, $\hat{z}_i$ and $\hat{z}_j$ represent the predicted scores for individual nodes within the trace, and $\sigma(\hat{\mathbf{z}})_i$ represents the score for the specific node in question after the softmax operation is applied. This approach allows for ranking nodes within a specific trace based on their scores for being the root cause of the detected fault.

$$\sigma(\hat{\mathbf{z}})_i = \frac{e^{\hat{z}_i}}{\sum_{j=1}^{K} e^{\hat{z}_j}} \tag{17}$$

As a final step, we use the anomaly prediction model as a supervisor for the root-cause localization model. Specifically, if the anomaly scores do not indicate an anomaly for a given trace, we set the root-cause scores for all the nodes in that trace to zero.

Each root-cause score is then penalized using Binary Cross Entropy (BCE) loss, as

defined in equation (18), where $z_i$ represents the actual root-cause label for the node.

$$\text{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\left[z_i \cdot \log(\hat{\sigma}(z_i)) + (1 - z_i) \cdot \log(1 - \hat{\sigma}(z_i))\right] \tag{18}$$

Finally, we train the three models together by treating the sum of the respective loss functions as the overall loss function for the model.

# 5 Experimental Validation

We conduct several experiments to evaluate multiple features of our proposed framework. We begin this chapter by formulating research questions based on which we design our experiments. In addition, we discuss the experimental setup, the datasets, and the evaluation metrics used for these experiments.

## 5.1 Research Questions

We conduct experiments to evaluate our framework based on the following Research Questions (RQ):

***RQ1: How effective is PerformanceLENS in predicting latency distribution at a granular level of individual API requests?***

Since single-point estimates alone are not sufficient to fully understand system performance, *PerformanceLENS* introduces a novel features to allow predicting latency distribution. In our experiments, we use several metrics, such as the coverage probability and quantile loss, to help us validate the reliability of predicted latency distribution.

***RQ2: How does PerformanceLENS compares to other baseline approaches for end-to-end latency prediction?***

To evaluate the effectiveness of *PerformanceLENS* in predicting end-to-end latency distribution of traces, we train *PerformanceLENS* with end-to-end latency data on multiple datasets. This allows us to compare *PerformanceLENS* against baseline methods providing a single-point estimate for end-to-end latencies such as the Multi-Layer Perceptron (MLP) model, GRAF [24], and PERT-GNN [27]. We use the lowest MAPE and MAE values obtained from the predicted quantiles by *PerformanceLENS* falling in the high-density regions specified in Table 1 for such a comparison.

***RQ3: How well can PerformanceLENS predict latency distribution of individual calls (edges) in a microservice call chain of an API request?***
*PerformanceLENS* employs a more practical approach for preserving temporal orderings as discussed in chapter 2 & 3. A byproduct of this approach is greater explainability, which helps us predict the latencies of individual microservice calls (edges) involved in an API request. We conduct a separate experiment to evaluate the performance of *PerformanceLENS* for this novel feature and report MAPE and MAE figures for different latency regions.

***RQ4: What insights can be drawn from the output latency distribution of PerformanceLENS during periods of anomalies?*** A key objective of our framework is to proactively monitor the system and provide valuable insights during anomalies by assessing their impact on system performance. To validate this aspect, we evaluate the performance of *PerformanceLENS* using datasets containing several types of anomalies leading to degradation in system performance. We check if the output latency distribution of *PerformanceLENS* have enough distinguishing features between normal and anomalous traces.

***RQ5: How effectively can the latency distributions predicted by PerformanceLENS be used by other Machine Learning models to automate the process of trace-based anomaly prediction?***
We check if the output of *PerformanceLENS* can be effectively used to automate the process of trace-based anomaly prediction by machine learning models. Hence, we feed the predicted latency distribution by a trained *PerformanceLENS* model to multiple machine learning models, such as Random Forest, KNNs, etc, to predict trace anomalies. We then report the accuracy, recall, and F1 figures for predicting trace anomalies. This is done to gather support for building a single framework for performance analysis and anomaly prediction.

***RQ6: How effective is an integrated framework based on PerformanceLENS for comprehensive performance analysis, anomaly prediction and root-cause localization for traces***

We train the integrated framework proposed in chapter 4 to predict performance of traces, trace-based anomalies and their root causes at the same time. We then report several metrics to compare our approach to several baseline trace-based anomaly prediction and root-cause localization approaches.

## 5.2 Datasets

We conduct our experiments on three benchmark datasets widely used in literature. We use the Alibaba dataset to compare *PerformanceLENS* with state-of-the-art latency prediction approaches proposed in [24] and [27]. This is because the Alibaba dataset does not contain anomalous data, and is used for experiments in [24] and [25], making it suitable for such a comparison. In addition, we use two benchmark datasets used for anomaly detection in literature: Train Ticket and MicroSS. Both datasets collected traces during periods when the underlying system operated with anomalies and contain traces with latencies spanning from 0.5 milliseconds to 20,000 milliseconds. We provide a summary of these datasets in Table 2 and discuss more details about these datasets below:

| **Dataset** | **Total Traces** | **Training Traces** | **Test Traces** |
|---|---|---|---|
| Alibaba | 100,000 | 80,000 | 20,000 |
| Train Ticket | 230,000 | 184,000 | 46,000 |
| MicroSS | 700,000 | 560,000 | 140,000 |

Table 2: Summary of Data Used In Experiments

### 5.2.1 Alibaba Dataset

The Alibaba dataset contains real-world traces collected in 2021 [57] from 10 production clusters during a 12-hour period. The dataset spans 20 thousand microservices. However, the dataset only records CPU and memory utilization of 1300+ microservices [27]. Since the dataset lacks other metric values and does not contain anomalous instances, it is not ideal for fully utilizing the performance of *PerformanceLENS* . Nonetheless, it is a good dataset for comparison with other baseline approaches. To remain consistent in our comparison, we use the same traces used in [27] for our experiments on this dataset.

### 5.2.2 MicroSS Dataset

MicroSS is another widely used dataset in the literature for anomaly detection. It contains 10 microservices, including MySQL and Redis database nodes. Multiple instances of the same microservices are deployed on 5 distinct machines, and multiple types of faults, such as system hang-ups, process crashes, and system failures like login issues, missing files, and access denials, are injected to create this data [4]. The dataset has been successfully used in [1] and [48] for anomaly detection and root-cause localization and provides metrics at the Docker container level for each microservice instance, making it suitable for our experiments.

### 5.2.3 Train Ticket Dataset

The Train Ticket application, made available by authors of [58], consists of 41 microservices and is frequently used in literature for root-cause identification and localization. We use the data generated from this application and published with [40] for our experiments. The dataset contains faults of varying types, such as application bugs, CPU exhaustion, and network congestion. In addition to having anomalous data, the dataset

provides metric values for CPU usage, memory usage, network, and disk-level metrics. We also use the train ticket dataset for the experiments we run for anomaly prediction and root-cause localization. Table 3 provides details of normal vs anomalous traces present in the train ticket dataset.

| **Total Traces** | **Normal Traces** | **Anomalous Traces** |
|---|---|---|
| 230,000 | 206,759 | 23,241 |

Table 3: Summary of Train Ticket Dataset

## 5.3 Experiments Configuration

We randomly divide our datasets into training and testing sets. We set aside 80 percent of traces for training and kept the remaining 20 percent hidden from the model for testing purposes. The exact number of traces used for training and testing in each dataset can be found in Table 2.

For reporting purposes, we divide the test data into four latency regions shown in [24] and report various evaluation metrics, discussed in the next section, for each latency region to understand the performance of *PerformanceLENS* across the latency spectrum. For our experiments, in total, we predict 21 quantiles on the latency distribution selected based on the discussions in chapter 3. Table 1 details the specific quantiles we predict on the latency distribution.

We also validate the effectiveness of the GRU block in understanding the temporal orderings by training *PerformanceLENS* in two distinct configurations: *PerformanceLENS-GNN* (without the GRU block) and *PerformanceLENS-Hybrid* (GRU block included) in all the experiments that we conduct.

Lastly, we conduct two sets of experiment for anomaly prediction and root-cause localization. In the first experiment, we train traditional ML models on the outputs of

a trained *PerformanceLENS* model for trace-based anomaly prediction. This is done to verify that there is enough information in the predicted latency distribution for ML models to predict anomalies. In the second experiment we train the framework proposed in chapter 4 as a whole to predict trace's latency distribution, whether it exhibits an anomaly, and identify the root cause if an anomaly is predicted.

## 5.4 Evaluation Metrics

We use a variety of metrics to validate both the novel features of *PerformanceLENS* as well as to compare the results with state-of-the-art approaches. We mention each of the metrics below.

**1. Quantile Loss:** As shown in chapter 3, quantile loss, given in equations (5) and (6), measures how accurately the model predicts a certain quantile ($\tau$). While it is hard to interpret quantile loss for a given quantile ($\tau$), as the ground truth value of the quantile is unknown, lower values indicate a better prediction model. We use quantile loss and other metrics to gauge how well the model predicts latency distribution. Specifically, we apply quantile loss across multiple quantiles predicted by *PerformanceLENS* to ensure that the loss values remain consistently low throughout the distribution.

**2. Coverage Probability:** For latency distribution prediction, coverage probability measures the proportion of times the predicted latency interval contains the observed latency value. Although the exact quantile corresponding to the observed value is unknown, the observed value falls within the latency distribution. Therefore, coverage probability, when used in tandem with quantile loss, allows assessment of the reliability of the predicted latency distribution. The coverage probability is calculated based on the equation (19), where $\tau_{i,l}$ and $\tau_{i,h}$ represent the lowest and highest quantile predicted

by our model for data point $i$ and $y_i$ represents the observed latency value.

$$\frac{1}{n} \cdot \sum_{i=0}^{n} 1(\tau_{i,l} < y_i < \tau_{i,h}) \tag{19}$$

**3. Mean Absolute Error (MAE):** MAE, represented in equation (20), is typically used to measure the accuracy of models providing single-point estimates of latency.

$$\frac{\sum_{i=0}^{n} |y_i - \hat{y}_i|}{n} \tag{20}$$

However, acceptable MAE values vary across different latency regions, and using an average MAE value for the whole dataset may bias the value towards either high or low latency values. For this reason, MAE is calculated for different latency regions separately. Since our model predicts latency distribution instead of single-point estimates, we report the lowest MAE figures obtained from the high-density regions shown in Table 1 to gauge the accuracy of our predictions.

**4. Mean Absolute Percentage Error (MAPE):** To allow better interpretability of the results we use MAPE, as represented in equation (21), along with MAE.

$$\frac{1}{n} \cdot \sum_{i=0}^{n} |\frac{y_i - \hat{y}_i}{y_i}| \tag{21}$$

This overcomes any bias that may be present in the reported MAE values. This also allows us to compare the performance of *PerformanceLENS* with other baseline approaches providing single-point latency estimates.

**5. R2-Score:** We use the R2 score, given by equation (22), to understand the proportion of variance in the observed latency value that the predictions can explain.

$$1 - \frac{\sum_{i=0}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{n}(y_i - \bar{y}_i)^2} \tag{22}$$

We calculate R2 for each quantile predicted by *PerformanceLENS* in the high-density regions specified in Table 1. This allows us to ascertain that flattening of predictions does not occur and that predicted quantiles capture relevant variations in data.

**6. Precision:** We use precision scores to judge the reliability of a positive anomaly prediction by our framework. Precision, given be (23), measures the proportion of true positive results to overall number of positive results. This makes precision a good indicator of the trustworthiness of a positive prediction.

$$\frac{TruePositives}{TruePositives + FalsePositives} \tag{23}$$

**7. Recall:** Recall is an indicator of how good a model is uncovering actual anomalies. Recall, given by (24), measures the proportion of true positive results to the number of actual positive samples present in the dataset. Thus, a model with a higher recall score correctly uncovers all positive samples in the given data.

$$\frac{TruePositives}{TruePositives + FalseNegatives} \tag{24}$$

**8. F1-Score:** Precision and Recall scores target different properties of a model and should be used together to better understand how well a model performs on a classification task. Usually, there is a tradeoff between precision and recall and F1-score assist in finding the best point on the tradeoff line. F1-score, given by (25), incorporates both precision and recall score for finding the best model.

$$2 \times \frac{Precision \times Recall}{Precision + recall} \tag{25}$$

**9. Top-k Accuracy:** Top-k accuracy is a metric commonly used in retrieval and recommendation tasks [4]. It measures how often the true item appears in the top k

items of the ranked list [4]. For the purpose of our experiments, we use it to measure the reliability of root-cause recommendations. Top-k accuracy can be calculated using equation (26), where $RC_i$ is the true root cause microservice and $RC_{is}[k]$ is the set of top-k predicted microservices, for an anomalous sample $i$.

$$A@k = \frac{1}{N}\sum_{i=0}^{n} \begin{cases} 1 & \text{if } RC_i \in RC_{is}[k] \\ 0 & \text{otherwise} \end{cases} \tag{26}$$

**10. Mean First Rank (MFR):** Another commonly used metric to gauge the accuracy of a ranked list is Mean First Rank (MFR). MFR, as the name suggests, measures the average rank of the first correct item in a ranked list. MFR can be calculated using equation (27), where $i$ represents a particular data sample, and $j$ represents the index of the first correct item in the sample $i$.

$$MFR = \frac{1}{N}\sum_{i=0}^{n} \text{rank}_{ij} \tag{27}$$

# 6 Results and Discussions

In this chapter, we present a detailed discussion of the experimental results and their findings. For each research question (RQ), we begin by summarizing the key takeaways, followed by an in-depth analysis of the results.

## 6.1 RQ1

*We observe that PerformanceLENS consistently obtains low quantile loss values for different $\tau$ values, indicating its effectiveness in predicting various points on the latency distribution. Moreover, high coverage probability values indicate a highly reliable and accurate predicted distribution interval.*

We provide a summary of model performance, using quantile loss and coverage probability, for latency distribution prediction in Table 4. These results indicate that *PerformanceLENS* can effectively monitor several SLAs set against different proportions of user requests for a given API. By using the output of *PerformanceLENS* for quantiles corresponding to SLAs, such as 75%, 90%, etc., we can monitor if there is a risk of the predicted quantile latency exceeding the latency threshold of an API governed by an SLA.

| | | Coverage | Quantile Loss | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Dataset** | **Model** | **Probability** | $\tau$ | **0.0228** | **0.1587** | **0.3085** | **0.4013** | **0.5000** | **0.5987** | **0.6915** | **0.8413** | **0.9772** |
| Train Ticket | PERF-GNN | 99.53% | | 0.0095 | 0.0466 | 0.0756 | 0.0888 | 0.0992 | 0.1044 | 0.1036 | 0.0869 | 0.0266 |
| Train Ticket | PERF-Hybrid | 99.65% | | 0.0088 | 0.0419 | 0.0668 | 0.0777 | 0.0859 | 0.0897 | 0.0886 | 0.0726 | 0.0235 |
| MicroSS | PERF-GNN | 99.77% | | 0.0029 | 0.0118 | 0.0178 | 0.0204 | 0.0223 | 0.0235 | 0.0233 | 0.0199 | 0.0068 |
| MicroSS | PERF-Hybrid | 99.76% | | 0.0024 | 0.0104 | 0.0160 | 0.0186 | 0.0207 | 0.0218 | 0.0212 | 0.0176 | 0.0059 |
| Alibaba | PERF-GNN | 99.53% | | 0.0056 | 0.0294 | 0.0489 | 0.0585 | 0.0674 | 0.0747 | 0.0788 | 0.0617 | 0.0164 |
| Alibaba | PERF-Hybrid | 99.01% | | 0.0055 | 0.0286 | 0.0480 | 0.0575 | 0.0655 | 0.0718 | 0.0730 | 0.0568 | 0.0149 |

Table 4: Latency Distribution Prediction Results - - *PERF* represents *PerformanceLENS*

## 6.2 RQ2

*PerformanceLENS efficiently explains variations in latencies and maintains consistent performance across all latency regions, even on anomalous datasets. In parallel, PerformanceLENS outperforms baselines, scoring lower MAPE values across all latency regions shown in [24].*

We use the MAPE and MAE figures to evaluate the performance of *PerformanceLENS* for end-to-end latency predictions and comparison with baseline approaches. Table 6 provides comprehensive results for end-to-end latency predictions on three benchmark datasets we use in our experiments, while Table 5 compares the results with baseline approaches. The benefit of preserving the temporal ordering through the GRU block can be seen in the results of *PerformanceLENS-Hybrid*, which outperforms *PerformanceLENS-GNN* across all metrics. The improvement over baseline approaches is achieved because *PerformanceLENS* manages to preserve the individuality of each request by treating each trace as a separate graph. Unlike PERT-GNN, which aggregates traces belonging to a specific API to make PERT Graphs API aware. In contrast, we use microservice embeddings for this purpose while simultaneously maintaining the individuality of each request. Furthermore, using the GRU unit to preserves temporal dependencies in a trace further improves the results of *PerformanceLENS* .

| | **Latency Regions** | | | |
|---|---|---|---|---|
| **Model** | ***0-50ms*** | ***50-100ms*** | ***100-1000ms*** | ***Overall*** |
| MLP | 16.76% | - | - | - |
| PERT-GNN [27] | 11.47% | - | - | - |
| GRAF [24] | 21.30% | 27.1% | 31.9% | - |
| **PERF-GNN** | **6.97%** | **8.78%** | **12.57%** | **10.83%** |
| **PERF-Hybrid** | **6.80%** | **5.68%** | **9.79%** | **9.75%** |

Table 5: Comparison of Model Performances on End-to-End Latencies Using MAPE (Alibaba Dataset) - - *PERF* represents *PerformanceLENS*

| | | Latency Regions | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | ***0-50ms*** | | ***50-100ms*** | | ***100-1000ms*** | | ***Overall*** | | |
| **Dataset** | **Model** | MAPE | MAE (ms) | MAPE | MAE (ms) | MAPE | MAE(ms) | MAPE | MAE (ms) | R2 Score |
| Train Ticket | PERF-GNN | 15.75% | 4.529 | 21.42% | 63.631 | 16.79% | 239.029 | 18.09% | 171.692 | 0.7689 |
| Train Ticket | PERF-Hybrid | 15.65% | 4.503 | 20.96% | 57.362 | 14.05% | 187.674 | 16.10% | 141.489 | 0.7866 |
| MicroSS | PERF-GNN | - | - | - | - | 1.25% | 69.424 | 1.48% | 141.024 | 0.8110 |
| MicroSS | PERF-Hybrid | - | - | - | - | 1.05% | 48.876 | 1.33% | 130.235 | 0.8311 |
| Alibaba | PERF-GNN | 6.97% | 1.886 | 8.78% | 52.144 | 12.57% | 104.279 | 10.83% | 9.315 | 0.2374 |
| Alibaba | PERF-Hybrid | 6.80% | 1.853 | 5.68% | 22.3619 | 9.79% | 94.4056 | 9.75% | 8.570 | 0.3004 |

Table 6: Regression Results for End-to-End Latency Predictions- *PERF* represents *PerformanceLENS*

## 6.3 RQ3

*PerformanceLENS can maintain consistent performance levels for edge latencies as it does for end-to-end latencies across all latency regions, achieving consistently low MAPE and MAE scores.*

Table 7 provides detailed performance metrics on edge latency predictions, showcasing *PerformanceLENS* 's ability to accurately predict latency distribution at a granular level. Due to several sub-millisecond latencies present in the Train Ticket dataset for the first edge in the microservice call chain, we report the MAE score separately for this region. This is because MAPE is inappropriate for latencies below 1 millisecond; even a small prediction error can lead to high MAPE values, skewing the overall figures.

| | Latency Regions | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ***<1ms*** | ***1-50ms*** | | ***50-100ms*** | | ***100-1000ms*** | | ***Overall*** | | |
| **Dataset** | MAE (ms) | MAPE | MAE (ms) | MAPE | MAE (ms) | MAPE | MAE(ms) | MAPE | MAE (ms) | R2 Score |
| Train Ticket | 0.2556 | 11.46% | 2.949 | 18.56% | 45.008 | 13.29% | 159.801 | 12.81% | 60.961 | 0.8520 |
| MicroSS | - | 13.96% | 2.902 | 7.12% | 16.901 | 2.09% | 51.389 | 3.84% | 64.166 | 0.9699 |

Table 7: Regression Results for Edge Latency Predictions

## 6.4 RQ4

*The results indicate that the output of PerformanceLENS contains enough distinguishing features to differentiate between normal and anomalous traces providing basis for automating the task of trace-based anomaly detection.*

A key objective of *PerformanceLENS* is to proactively monitor the system and identify potential SLA breaches, which are more likely during anomalies. We validate the performance of *PerformanceLENS* under anomalous conditions using MicroSS and Train Ticket datasets. We use the residual vs fitted plot (Figure 7) for the 50th quantile to visualize how model performance varies with latency and validate if our model is biased by the presence of anomalous instances. The plot does not reveal any bias but we observe heteroscedasticity, with residual values decreasing as the fitted values increase. This is expected as we use logarithmic scale for predictions, which compresses the scale of higher latency values resulting in smaller residuals as the predicted latency increases.

Since the model is not biased by the inclusion of anomalous data, we look at the prediction of *PerformanceLENS* on normal and anomalous traces separately to check if there are any distinguishing feature between predicted latency distribution of normal and anomalous traces. Table 8 compares the predictions for normal vs anomalous data.

| | **Normal Data** | **Anomalous Data** |
|---|---|---|
| Avg. Predicted Interval Length | 2.3509 | 3.8720 |
| Avg. Prediction ($\tau = 0.5$) | 0.8520 | 1.5989 |

Table 8: Predictions on Normal vs Anomalous Data (Log Scale)

We observe that, on average, the predicted interval length for anomalous data is much greater than for normal data. Similarly, the median prediction for anomalous data is

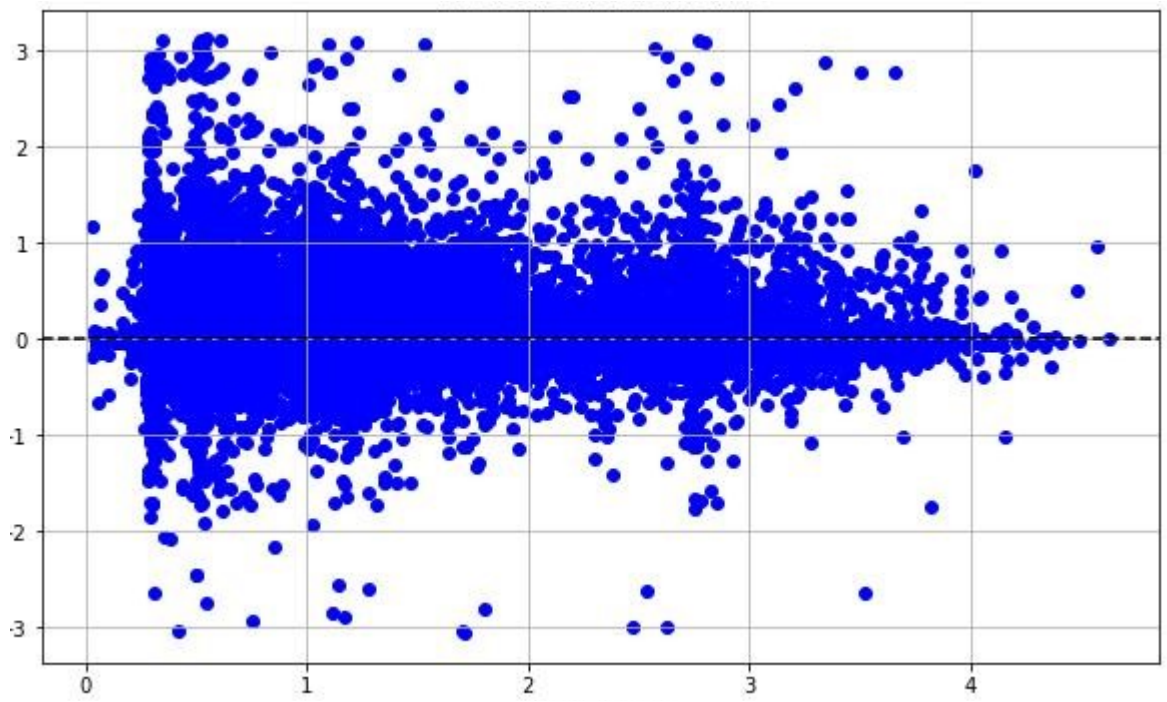



Figure 7: Residual vs Fitted plot for $\tau = 0.5$

significantly higher when compared to normal data. This is visualized by plotting the predicted interval length of the latency distribution on a box and whisker plot in figure 8. This serves as an encouraging base to feed the output of *PerformanceLENS* to traditional machine learning models for automating the task of anomaly prediction.

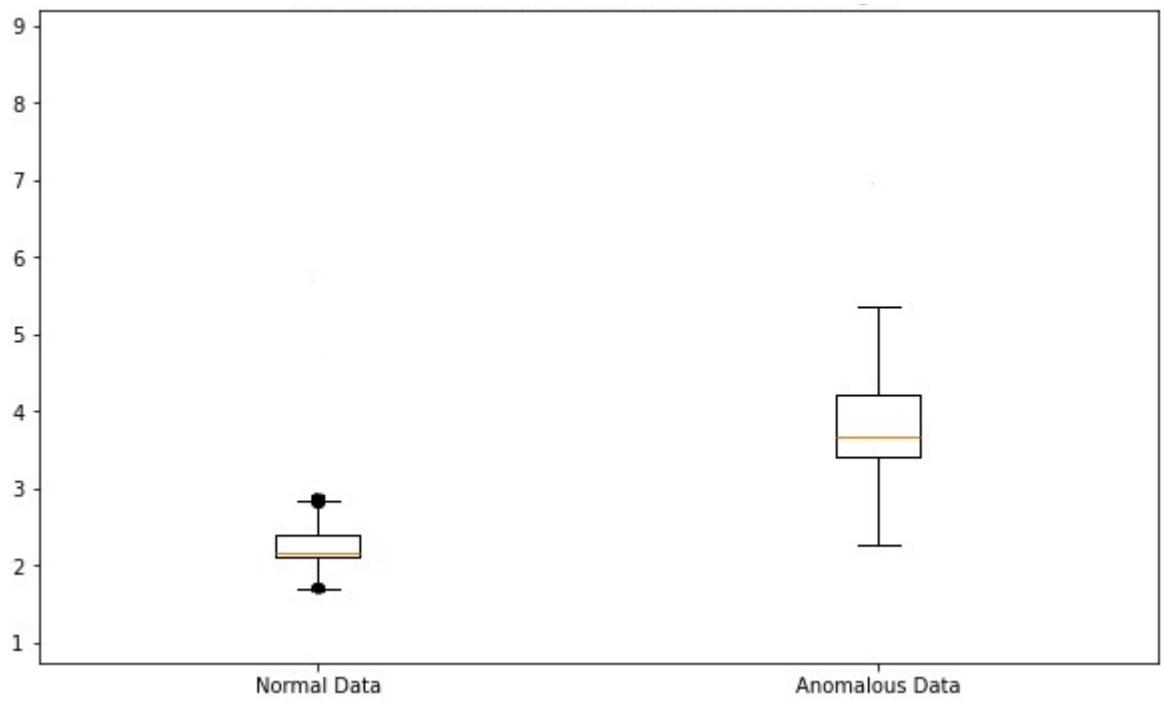



Figure 8: Box And Whisker Plot for Predicted Interval Length
Normal vs Anomalous Traces

## 6.5 RQ5

*The results prove that PerformanceLENS can be used effectively to automate the process of trace-based anomaly prediction, allowing PerformanceLENS to perform both performance prediction and anomaly prediction simultaneously at a marginal additional cost. Furthermore, this is a basis for automating other tasks, such as root-cause analysis.*

We use the following models: Gradient Boosting, Decision Trees, Random Forest, and KNNs to predict trace anomalies using the output of trained *PerformanceLENS* model. The training for the models are done after the training for *PerformanceLENS* has been completed. Table 9 provides the results of this experiment.

| ML Model | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| Gradient Boosting | 0.967 | 0.892 | 0.799 | 0.838 |
| Decision Trees | 0.963 | 0.833 | 0.849 | 0.841 |
| Random Forest | 0.976 | 0.921 | 0.856 | 0.885 |
| KNN | 0.976 | 0.907 | 0.873 | 0.889 |

Table 9: Automated Anomaly Prediction Results
(Train Ticket Dataset)

All models tested can achieve accuracy of more than 96% and with good precision and recall values in a heavily unbalanced dataset, with KNN achieving the best results. Since these additional models are easy to train and deploy, they come at a marginal additional cost, making *PerformanceLENS* a system that can perform performance prediction and anomaly detection in parallel. These insights can be combined with *PerformanceLENS* 's ability to predict latency distribution at the edge level for root-cause analysis. The edge with an abnormal latency distribution can be used as an indicator for further investigation. Figure 9 demonstrates how this can be achieved by highlighting nodes part of edges with abnormal latency distribution predictions in red. The graph can then be traced back to the anomalous node, enabling targeted diagnostics and resolution efforts.

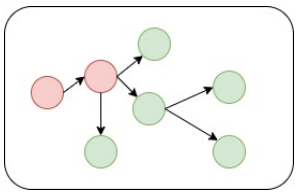


Figure 9: Root Cause Analysis using Edge Latency Distributions of a trace. Nodes with abnormal latency distribution prediction are highlighted in red.

## 6.6 RQ6

*The results show that our unified framework based on PerformanceLENS matches the performance of state-of-the-art trace-based anomaly detection methods and surpasses most baseline approaches in root-cause localization, with the exception of a few. This comparison highlights the effectiveness of our framework in predicting anomalies before the system receives anomalous traces, as opposed to state-of-the-art models that detect anomalies only after the system has processed the anomalous traces.*

We report the results of our unified framework for trace-based anomaly detection and root-cause localization along with the results of other state-of-the art approaches in Table 10 and Table 11 respectively. Specifically, we compare our framework with 7 other approaches proposed in [39], [40], [51], [50], [49] and [41]. Table 10 details the model performance for trace based anomaly detection and compares it with MEPFL and TraceRCA, two other best performing approaches.

| Model | Precision | Recall | F1 |
|---|---|---|---|
| **Proposed Framework** | **0.937** | **0.920** | **0.928** |
| MEPFL [39] | 0.865 | 0.993 | 0.924 |
| TraceRCA [40] | 0.801 | 0.750 | 0.775 |

Table 10: Comparison of proposed framework in proactive anomaly prediction with state-of-art approaches in reactive anomaly detection (Train Ticket Dataset)

As evident, our approach performs better than both TraceRCA and MEPFL on F1 score for trace-based anomaly detection. Although, the performance is similar to MEPFL, our approach predicts failure before they occur and without the knowledge of trace based features available to MEPFL after traces are executed.

We observe similar encouraging performance of our framework for root-cause localization. Table 11 provides comparison of our model for root-cause localization on top-k

accuracies for k=1,2, and 3. Our approach performs better than most baseline approaches, except for TraceRCA and MEPFL. This result is expected, as both TraceRCA and MEPFL perform root-cause localization after the system has received anomalous traces. Since anomalies can propagate both upstream and downstream within a trace [40], having access to all traces passing through the system within a given time window, allows MEPFL and TraceRCA to better identify anomaly propagation patterns. This results in better root-cause localization results. However, this reactive nature places them in the category of detection approaches rather than predictive approaches. In contrast, our framework operates on a predictive basis, assuming the system receives a particular trace without actually processing it. It predicts the trace's latency distribution, whether it will exhibit an anomaly, and identifies the root cause based solely on the current system metrics. This proactive approach means that the our framework can not have knowledge of traces passing through a system, as the system has not yet processed those traces, making it challenging to infer the propagation patterns of anomalies, thus making it difficult to pinpoint the root-cause microservice.

| **Model** | **A@1** | **A@2** | **A@3** |
|---|---|---|---|
| MEPFL [39] | 0.92 | 0.97 | 0.97 |
| TraceRCA [40] | 0.82 | 0.91 | 0.95 |
| **Proposed Framework** | **0.80** | **0.89** | **0.94** |
| RCSF [49] | 0.52 | 0.86 | 0.93 |
| Random Walk [50] | 0.51 | 0.86 | 0.94 |
| Microscope [51] | 0.56 | 0.62 | 0.70 |
| TraceAnomaly [41] | 0.49 | 0.59 | 0.63 |

Table 11: Comparison of proposed framework in proactive root-cause localization with state-of-art approaches in reactive root-cause localization (Train Ticket Dataset)

To further solidify our findings, we utilize MFR on root-cause prediction results and compare it with state-of-the art models. Table 12 provides the results of our experiments using MFR.

| Model | MFR |
|---|---|
| MEPFL [39] | 1.37 |
| TraceRCA [40] | 1.50 |
| **Proposed Framework** | **1.54** |
| RCSF [49] | 1.77 |
| Random Walk [50] | 2.26 |
| Microscope [51] | 3.55 |
| TraceAnomaly [41] | 4.58 |

Table 12: Comparison of proposed framework in proactive root-cause localization with state-of-art approaches in reactive root-cause localization using MFR (Train Ticket Dataset)

Again, our framework outperforms most existing approaches and closely matches the performance of TraceRCA. Our framework is only outperformed by MEPFL, however, as discussed earlier this is expected due to the proactive approach of prediction rather than detection.

Our approach also has a distinct advantage over state-of-the-art approaches when it comes to root-cause localization due to its focus on individual traces. By concentrating on a single trace, the set of ranked microservices includes only those that belong to that trace. This narrowed focus allows for quicker improvements in top-k accuracy as the value of k increases, thereby significantly reducing the delta with MEPFL and TraceRCA at higher k-values. While MEPFL and TraceRCA can better detect propagation patterns by having information on all traces passing through a system, this broader scope also introduces noise into the data. Consequently, these algorithms must consider all microservices involved in the traces, leading to a much larger set of potential root causes. We quantify these findings by providing the delta of our proposed framework's top-k accuracy against MEPFL and TraceRCA in Table 13.

| Model | A@1 | A@2 | A@3 |
|---|---|---|---|
| MEPFL [39] | -0.12 | -0.08 | -0.03 |
| TraceRCA [40] | -0.02 | -0.02 | -0.01 |

Table 13: Top-K accuracy delta of Proposed Framework against MEPFL and TraceRCA (Train Ticket Dataset)

## 6.7 Threats To Validity

Although the results of our framework are encouraging, we have identified a few threats to validity that warrant careful consideration when interpreting these results.

Firstly, we use datasets with anomalies and evaluate the performance of *Performance-LENS* in predicting latency during anomalous time periods. Even with the inclusion of anomalous data, on average, the performance of the model does not deteriorate. However, it is to be noted that during certain anomalies, complete system breakdown can occur, rendering latency inconsequential. We do not include such anomalies in our experiments. In such cases, the predicted latency is irrelevant, and the output of *PerformanceLENS* should be interpreted carefully to determine whether it indicates a system anomaly.

Secondly, our framework uses individual traces to formulate graphs. It is expected that in a real-world setting, the path an API request takes is already known, allowing specific traces to be monitored after the model is trained. However, it is known that a single API can exhibit different runtime behaviors [27]. Therefore, in order to effectively monitor a single API, multiple traces will need to be monitored, and their results analyzed in aggregate to ensure that the system continues to uphold SLAs.

# 7 Conclusion

In this research, we presented *PerformanceLENS* , an approach that utilizes microservices embeddings along with the hybrid GNN/GRU architecture to model the performance of a microservices application by predicting the latency distribution of each individual microservice call involved in an API request. Each individual API call forms a trace and constitutes a single graph in our approach. We utilized metrics from microservices along with microservice embeddings to construct node features, where each node represents an individual microservice and each edge represents an interaction between two microservices. The output of the GNN is used to construct edge embeddings, which are then forwarded to a GRU unit. We trained *PerformanceLENS* using multi-quantile loss to predict an arbitrary number of quantiles, thereby estimating the latency distribution given the current system parameters. Our results show that *PerformanceLENS* outperforms baseline models across all latency regions on end-to-end latencies and consistently achieves low quantile loss values and high coverage probability for the predicted latency distribution.

The novel features introduced by *PerformanceLENS* makes it suitable not only for detecting potential SLA breaches but also for anomaly prediction and root cause analysis. To support this position, we leveraged traditional ML models to accurately predict trace-based anomalies based on the latency distribution output of *PerformanceLENS* model. We then demonstrated how a unified framework, based on *PerformanceLENS* model, can be designed for predicting performance, anomalies, and their root-causes. We empirically show that our framework manages to outperform state-of-art reactive anomaly detection approaches while predicting trace-based anomalies in advanced. Similarly, we show effectiveness of our framework in localizing the root-causes of the predicted anomalies.

Future directions include using the output of *PerformanceLENS* automate other down-

stream tasks such as resource allocation and integrating the proposed framework with a live system.

## 7.1 Integration With Live Systems

Once trained, our framework can be integrated into a live microservices system to provide comprehensive system monitoring. This integration depends on live metrics data obtained from the system's monitoring channels at regular intervals. In real-time operation, the incoming metrics data forms part of the node features for the GNN, just as in the training stage, while the other part is derived from the trained microservice embeddings. It is expected that the system operators will configure the traces they want monitored by specifying the microservices part of the traces. The constructed node features are then fused with the pre-configured traces to predict the performance for those traces based on the current metric values.

The output of *PerformanceLENS* can then be analyzed against predefined SLAs by monitoring specific quantiles from the latency distribution predictions to detect potential SLA breaches. Additionally, the framework can proactively predict whether a trace will be anomalous and identify the specific node within the trace that is likely responsible for the anomaly.

# Bibliography


[1] C. Zhao, M. Ma, Z. Zhong, S. Zhang, Z. Tan, X. Xiong, L. Yu, J. Feng, Y. Sun, Y. Zhang, *et al.*, "Robust multimodal failure detection for microservice systems," *arXiv preprint arXiv:2305.18985*, 2023.

[2] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li, "Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments," in *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021* (J. Leskovec, M. Grobelnik, M. Najork, J. Tang, and L. Zia, eds.), pp. 3087–3098, ACM / IW3C2, 2021.

[3] P. Jamshidi, C. Pahl, N. C. Mendoncca, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[4] R. Rouf, M. Rasolroveicy, M. Litoiu, S. Nagar, P. Mohapatra, P. Gupta, and I. Watts, "Instantops: A joint approach to system failure prediction and root cause identification in microserivces cloud-native applications," in *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24, (New York, NY, USA), p. 119–129, Association for Computing Machinery, 2024.

[5] T. F. da Silva Pinheiro, P. Pereira, B. Silva, *et al.*, "A performance modeling framework for microservices-based cloud infrastructures," *Journal of Supercomputing*, vol. 79, pp. 7762–7803, 2023.

[6] V. Christiansen, M. Haddara, and M. Langseth, "Factors affecting cloud erp adoption decisions in organizations," *Procedia Computer Science*, vol. 196, pp. 255–262, 2022.

[7] A. Khayer, N. Jahan, M. N. Hossain, and M. Y. Hossain, "The adoption of cloud computing in small and medium enterprises: a developing country perspective," *VINE Journal of Information and Knowledge Management Systems*, vol. 51, no. 1, pp. 64–91, 2021.

[8] C. Sauvanaud *et al.*, "Anomaly detection and diagnosis for cloud services: practical experiments and lessons learned," *Journal of Systems and Software*, vol. 139, pp. 84–106, 2018.

[9] S. Xie, J. Wang, B. Li, Z. Zhang, D. Li, and P. C. Hung, "Pbscaler: A bottleneck-aware autoscaling framework for microservice-based applications," *IEEE Transactions on Services Computing*, 2024.

[10] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 217–231, 2014.

[11] L. Huang and T. Zhu, "tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 76–91, 2021.

[12] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable sampling for distributed traces, without feature engineering," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 312–324, 2019.

[13] 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*, pp. 19–33, 2019.

[14] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web*, pp. 469–478, 2017.

[15] L. Weng, P. Huang, J. Nieh, and J. Yang, "Argus: Debugging performance issues in modern desktop applications with annotated causal tracing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 193–207, 2021.

[16] Y. Liu, H. Xu, and W. C. Lau, "Accordia: Adaptive cloud configuration optimization for recurring data-intensive applications," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 831–841, IEEE Computer Society, 2020.

[17] Y. Liu, H. Xu, and W. C. Lau, "Online resource optimization for elastic stream processing with regret guarantee," in *Proceedings of the 51st International Conference on Parallel Processing (ICPP '22)*, 2022.

[18] S. Luo, X. HL, K. Ye, G. Xu, L. Zhang, J. He, G. Yang, and C. Xu, "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*, 2023.

[19] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.

[20] C. Leung and J. Schormans, "Measurement-based end to end latency performance prediction for sla verification," *Journal of Systems and Software*, vol. 74, no. 3, pp. 243–254, 2005.

[21] B. Sun, B. Hall, H. Wang, D. W. Zhang, and K. Ding, "Benchmarking private cloud performance with user-centric metrics," in *2014 IEEE International Conference on Cloud Engineering*, pp. 311–318, 2014.

[22] C. Vazquez, R. Krishnan, and E. John, "Cloud computing benchmarking: a survey," in *Proceedings of the international conference on grid, cloud, and cluster computing (GCC)*, p. 1, The Steering Committee of The World Congress in Computer Science, Computer . . . , 2014.

[23] Y. Rouf, J. Mukherjee, and M. Litoiu, "Towards a robust on-line performance model identification for change impact prediction," in *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 68–78, 2023.

[24] J. Park, B. Choi, C. Lee, and D. Han, "Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices," in *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21)*, p. 14, 2021.

[25] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 200–210, IEEE, 2019.

[26] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in

serverless platforms," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 153–167, 2021.

[27] D. S. H. Tam, Y. Liu, H. Xu, S. Xie, and W. C. Lau, "Pert-gnn: Latency prediction for microservice-based cloud-native applications via graph neural networks," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '23)*, p. 11, 2023.

[28] R. Ahad, E. Chan, and A. Santos, "Toward autonomic cloud: Automatic anomaly detection and resolution," in *2015 International conference on cloud and autonomic computing*, pp. 200–203, IEEE, 2015.

[29] S. Garg, K. Kaur, N. Kumar, S. Batra, and M. S. Obaidat, "Hyclass: Hybrid classification model for anomaly detection in cloud environment," in *2018*, pp. 1–7, 2018.

[30] T. Hagemann and K. Katsarou, "A systematic review on anomaly detection for cloud computing environments," in *2020 3rd Artificial Intelligence and Cloud Computing Conference*, pp. 83–96, 2020.

[31] J. Hochenbaum, O. S. Vallis, and A. Kejariwal, "Automatic anomaly detection in the cloud via statistical learning," *arXiv preprint arXiv:1704.07706*, 2017.

[32] M. S. Islam, W. Pourmajidi, L. Zhang, J. Steinbacher, T. Erwin, and A. Miranskyy, "Anomaly detection in a large-scale cloud platform," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 150–159, IEEE, 2021.

[33] J. Mukherjee, A. Baluta, M. Litoiu, and D. Krishnamurthy, "Rad: Detecting performance anomalies in cloud-based web services," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pp. 493–501, IEEE, 2020.

[34] N. Pandeeswari and G. Kumar, "Anomaly detection system in cloud environment using fuzzy clustering based ann," *Mobile Networks and Applications*, vol. 21, pp. 494–505, 2016.

[35] J. Soldani and A. Brogi, "Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey," *ACM Computing Surveys (CSUR)*, vol. 55, no. 3, pp. 1–39, 2022.

[36] L. Girish and S. K. Rao, "Anomaly detection in cloud environment using artificial intelligence techniques," *Computing*, pp. 1–14, 2021.

[37] M. Litoiu, I. Watts, and J. Wigglesworth, "The 13th cascon workshop on cloud computing: engineering aiops," in *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*, pp. 280–281, 2021.

[38] D. Pop, "Machine learning and cloud computing: Survey of distributed and saas solutions," *arXiv preprint arXiv:1603.08767*, 2016.

[39] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 683–694, Association for Computing Machinery, 2019.

[40] Z. Li, J. Chen, R. Jiao, N. Zhao, Z. Wang, S. Zhang, *et al.*, "Practical root cause localization for microservice systems via trace analysis," in *IEEE/ACM International Symposium on Quality of Service (IWQoS) 2021*, IEEE, 2021.

[41] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, *et al.*, "Unsupervised detection of microservice trace anomalies through

service-level deep bayesian networks," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pp. 48–58, IEEE, 2020.

[42] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating instrumentation data to system states: A building block for automated diagnosis and control.," in *OSDI*, vol. 4, pp. 16–16, 2004.

[43] A. U. Gias, G. Casale, and M. Woodside, "Atom: Model-driven autoscaling for microservices," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1994–2004, IEEE, 2019.

[44] 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*, pp. 19–33, 2019.

[45] 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*, pp. 167–181, 2021.

[46] M. M. Vance, *PERT and CPM: a selected bibliography (Volume Committee of Planning Librarians. Exchange bibliography, no. 53)*, vol. 41. Harvard Business Review, 1963.

[47] C. Lee, T. Yang, Z. Chen, Y. Su, Y. Yang, and M. R. Lyu, "Heterogeneous anomaly detection for software systems via semi-supervised cross-modal attention," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1724–1736, IEEE, 2023.

[48] S. Zhang, P. Jin, Z. Lin, Y. Sun, B. Zhang, S. Xia, Z. Li, Z. Zhong, M. Ma, W. Jin, *et al.*, "Robust failure diagnosis of microservice system through multimodal data," *arXiv preprint arXiv:2302.10512*, 2023.

[49] K. Wang, C. Fung, C. Ding, P. Pei, S. Huang, Z. Luan, and D. Qian, "A methodology for root-cause analysis in component based systems," in *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*, pp. 243–248, IEEE, 2015.

[50] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 93–104, 2013.

[51] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16*, pp. 3–20, Springer, 2018.

[52] K.-H. Chow, U. Deshpande, S. Seshadri, and L. Liu, "Deeprest: deep resource estimation for interactive microservices," in *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 181–198, 2022.

[53] R. Koenker and G. Bassett, "Regression quantiles," *Econometrica*, vol. 46, no. 1, pp. 33–50, 1978.

[54] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, p. 1137–1155, mar 2003.

[55] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language*

*Technologies* (L. Vanderwende, H. Daumé III, and K. Kirchhoff, eds.), (Atlanta, Georgia), pp. 746–751, Association for Computational Linguistics, June 2013.

[56] V. P, C. G, C. A, R. A, L. P, and B. Y, "Graph attention networks," *stat*, vol. 1050, no. 20, pp. 10–48550, 2017.

[57] 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*, pp. 412–426, 2021.

[58] 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.