FastPERT: Towards Fast Microservice Application Latency Prediction via Structural Inductive Bias over PERT Networks

Da Sun Handason Tam[1], Huanle Xu[2], Yang Liu[3], Siyue Xie[1], Wing Cheong Lau[1]

[1]The Chinese University of Hong Kong
[2]University of Macau
[3]Shanghai University
tds019@ie.cuhk.edu.hk, huanlexu@um.edu.mo, yangliu_cs@shu.edu.cn, xs019@ie.cuhk.edu.hk, wclau@ie.cuhk.edu.hk

Abstract

The recent surge in popularity of cloud-native applications using microservice architectures has led to a focus on accurate end-to-end latency prediction for proactive resource allocation. Existing models leverage Graph Transformers to Microservice Call Graphs or the Program Evaluation and Review Technique (PERT) graphs to capture complex temporal dependencies between microservices. However, these models incur a high computational cost during both training and inference phases. This paper introduces FastPERT, an efficient model for predicting end-to-end latency in microservice applications. FastPERT dissects an execution trace into several microservices tasks, using observations from prior execution traces of the application, akin to the PERT approach. Subsequently, a prediction model is constructed to estimate the completion time for each individual task. This information, coupled with the computational and structural inductive bias of the PERT graph, facilitates the efficient computation of the end-to-end latency of an execution trace. As a result, FastPERT can efficiently capture the complex temporal causality of different microservice tasks, leading to more accurate and robust latency predictions across a variety of applications.

An evaluation based on datasets generated from large-scale Alibaba microservice traces reveals that FastPERT significantly improves training and inference efficiency without compromising performance, demonstrating its potential as a superior solution for real-time end-to-end latency prediction in cloud-native microservice applications.

1 Introduction

Microservice architecture is becoming a pivotal software development framework within the realm of cloud-native applications. It decomposes an application as an assembly of small and self-contained services, which can be independently deployed and managed. Owing to these advantages, industry giants such as Netflix, Twitter, Facebook, and Airbnb have embraced this architecture for building their cloud-based applications (net 2019; Dragoni et al. 2017; Sriraman, Dhanotia, and Wenisch 2019; air 2017).

In many cases, microservices are deployed in distinct Docker containers and orchestrated by Kubernetes (Merkel et al. 2014; Kuberntes 2014). Client users send API call requests to the application via the API endpoints (Chow et al. 2022a). Microservices with dedicated functions are then invoked to fulfill these user requests. For example, a social networking application mighht have a `post/compose` API endpoint that enables users to create a post. This API endpoint could invoke the `postStorage` microservice to archive the post in the database and the `userTimeline` microservice to refresh the user's timeline.

Compared to a monolithic architecture, the modular nature of microservices facilitates easy scaling of the application. When the application faces an increasing load, the service provider can identify and scale individual microservices experiencing heavy traffic, rather than scaling the entire application. State-of-the-art resource scaling solutions for microservices primarily focus on proactive scaling (Zhang et al. 2021; Qiu et al. 2020; Chow et al. 2022b; Park et al. 2021a; Wang et al. 2022; Tam et al. 2023). In this approach, the objective is to predict the end-to-end latency of a microservices trace under various resource configurations in advance. It's important to note that a trace encompasses all microservices invoked from the initial API call requests to the final response through the API endpoints. The optimal configuration is then chosen to meet the service level objective (SLO).

The success of real-time proactive scaling hinges on efficient and accurate end-to-end latency prediction models. However, these goals often conflict, as accounting for intricate microservice interactions can compromise model lightweightness. Neglecting complex dependencies leads to inaccurate predictions (Zhang et al. 2021; Park et al. 2021b; Tam et al. 2023; Park et al. 2021b), while overly complex models, like PERT-GNN (Tam et al. 2023), which uses a Program Evaluation and Review Technique (PERT) graph (see sections 4.1 and 4.2 for details) and Graph Transformers to capture temporal dependencies between microservice tasks, become computationally demanding and challenging to deploy in real-time production environments.

In addition to grappling with the trade-off between effectiveness and efficiency, most prior works overlook the inherent structural and computational inductive biases of microservices traces. Specifically, the **structural inductive bias** refers to the fact that the relationships among microservices in a trace naturally form a Directed Acyclic Graph (DAG). This structure reveals the dependencies between consecutive microservices. The **computational inductive bias** pertains to the principle that the end-to-end latency of

a microservices trace is equivalent to the cumulative completion time of each microservice task along any path from the source to the sink. This principle implies a consistent time cost across different paths. However, these two crucial inductive biases are often neglected by existing methods when constructing their models, leading to suboptimal performance in end-to-end latency predictions.

To tackle the aforementioned challenges, we introduce FastPERT, a lightweight model for predicting end-to-end latency in microservice applications. Building on PERT-GNN, FastPERT leverages structural and computational inductive biases from microservices traces to accelerate training and inference without sacrificing performance. Unlike PERT-GNN, our two-stage approach first predicts the completion time for each microservice task (node) using a lightweight model, then captures temporal dependencies by averaging path completion times in the PERT graph. Experiments on Alibaba's real-world datasets show that FastPERT achieves comparable accuracy to PERT-GNN while significantly improving training and inference efficiency.

In summary, our primary contributions are threefold:

1. We revisit the problem of end-to-end latency prediction in microservices traces and investigate the limitations of existing models, which suffer from slow training and inference efficiency.
2. We propose FastPERT, a lightweight framework design that leverages structural and computational inductive biases to predict end-to-end latency with efficient precomputation techniques, **without** relying on Graph Neural Networks (GNNs).
3. Through extensive experiments on a real-world Alibaba traces dataset, we demonstrate FastPERT's effectiveness and efficiency, outperforming PERT-GNN in training and inference speed while achieving comparable prediction accuracy without requiring a GPU.

2 Related Works

Microservice performance estimation has been explored in various works. Distributed tracing systems (Las-Casas et al. 2019; Chow et al. 2014; Huang and Zhu 2021; Weng et al. 2023a) focus on bias sampling towards underrepresented traces. Root cause analysis systems (Gan et al. 2019; Jayathilaka, Krintz, and Wolski 2017; Weng et al. 2021, 2023b) diagnose performance issues by comparing runtime performance to estimations. Resource allocation systems (Liu, Xu, and Lau 2020, 2022; Rzadca et al. 2020; Luo et al. 2023, 2022b) scale containers based on performance estimation. We categorize prior works into three groups:

Model-based approaches: Sophisticated models (Cohen et al. 2004; Gias, Casale, and Woodside 2019; Bhasi et al. 2021; Luo et al. 2023; Shi et al. 2023; Luo et al. 2022b) represent microservice architecture or performance with mathematical models such as Layered Queuing Network (LQN) and Variable Order Markov Model (VOMM). However, these approaches often lead to significant estimation errors due to assumptions about latency or resource usage.

Learning-based approaches: Machine learning approaches (Gan et al. 2019; Zhang et al. 2021; Yang et al. 2019; Shi et al. 2022) (e.g. CNN, LSTM) can also be used to design performance estimation models. In addition, *Firm*(Qiu et al. 2020) uses Reinforcement Learning to improve latency along the critical path and *Sage*(Gan et al. 2021) uses a Causal Bayesian Network and Graphical Variational Auto-Encoder to generate counterfactuals for QoS violations. *DeepRest*(Chow et al. 2022b) incorporates span graphs to include the call order information into the estimation model. While some consider component dependencies, they often neglect the order of invocations or temporal dependencies among microservice operations and fail to capture the runtime dynamics of the same API call.

Our approach combines structural and computational inductive bias without relying on GNNs. By decomposing execution traces into microservice tasks via the PERT graph, it individually estimates task completion times, providing accurate and fast latency predictions for diverse applications.

GNN-based approaches: Graph Neural Networks (GNNs) (Park et al. 2021a; Wang et al. 2022; Tam et al. 2023) process graph-structured data (e.g. MS Call Graph, Span Graph, Trace-PERT Graph) to handle the callings relationships between microservices. However, these approaches have significant training and inference overhead and require frequent retraining to ensure accuracy.

In comparison to PERT-GNN (Tam et al. 2023), our proposed FastPERT method has three key differences:

1. Unlike PERT-GNN, FastPERT leverages ground-truth delay information for each microservice task, enhancing prediction capability.
2. Unlike PERT-GNN, FastPERT is not a GNN model and it does not rely on computationally intensive graph transformers. It does not use neighborhood aggregation for learning node/graph representations. Instead, it leverages the key property of constant path latencies in Trace-PERT graphs and rely on the path latency aggregation schema to compute the end-to-end latency predictions, which is difficult for ML models to learn. This approach enables a highly scalable algorithm using vector operations (Section 5.1), significantly improving training efficiency while maintaining comparable prediction accuracy.
3. FastPERT allows for interpretable models, such as decision trees, to estimate task delays and explain end-to-end latency predictions, unlike the black-box nature of PERT-GNN's graph transformers.

A comprehensive comparison of our approach with existing methods can be found in Table 1.

3 Problem Formulation

Our research aims to predict the end-to-end latency of microservice applications based on API calls and recent resource utilization. We define a dataset $\mathcal{D} = \{(G^{(i)}, \mathbf{X}^{(i)}, \mathcal{R}^{(i)}, y^{(i)})_{i=1}^{n}\}$, where $G^{(i)}$ is a graph representing microservice relations, $\mathbf{X}^{(i)}$ are static node features (e.g., unique identifier, communication paradigms such as HTTP, RPC, Database, Memchached), $\mathcal{R}^{(i)}$ is the recent resource utilization (e.g., CPU, memory, response times), and $y^{(i)}$ is the ground-truth end-to-end latency. A full list of node features is detailed in the supplementary material.

Source	Tech.	MCG	Span Graph	PERT Graph	Est. Speed
Kraken s	M	✓	✗	✗	Fast
ATOM [ICDCS'19]	M	✓	✗	✗	Fast
Nodens [ATC'23]	M	✓	✓	✗	Fast
Seer [ASPLOS'19]	L	✗	✗	✗	Slow
Sage [ASPLOS'21]	L	✗	✗	✗	Slow
FIRM [OSDI'20]	L	✓	✗	✗	Slow
DeepRest[EuroSys'22]	L	✓	✓	✗	Slow
GRAF [CoNEXT'21]	GNN	✓	✗	✗	Slow
DeepScaling [SoCC'22]	GNN	✓	✗	✗	Slow
PERT-GNN [KDD'23]	GNN	✗	✗	✓	Slow
Our work	**L**	✗	✗	✓	Fast

Table 1: Related Works for Performance Estimation
"M" represents model-based approach;
"L" represents learning-based approach, except GNN;
"GNN" represents GNN-based approach;

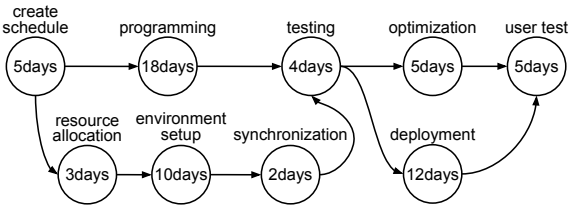


Figure 1: Project-PERT graph

In this paper, we use the Trace-PERT Graph for $G^{(i)}$, which is discussed in section 4.2. Each graph $G^{(i)}$ may differ due to varying microservice dependencies. We formulate our task as a **supervised graph regression problem**:

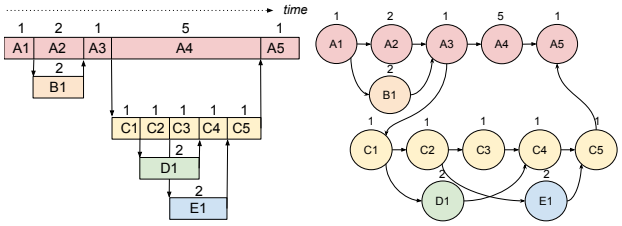
$$\hat{y}^{(i)} := f_\theta(G^{(i)}, \mathbf{X}^{(i)}, \mathcal{R}^{(i)}) \in \mathbb{R} \quad \forall i \in \{1, \ldots, n\} \quad (1)$$

where f_θ is a learnable function that predicts the end-to-end latency. For simplicity, we will omit the superscript i and assume $\hat{y}$ pertains to a single trace.

4 Graph Construction for Microservice Traces

4.1 Project-PERT Graph

PERT (Program Evaluation and Review Technique) is a statistical tool for project management that involves breaking down a project into tasks, estimating task duration, and determining dependencies between tasks (Vance 1963). It examines the time required to complete each task and determines the total project duration. A Project-PERT graph represents the tasks and their dependencies as a Directed Acyclic Graph (DAG), where nodes represent tasks and edges represent dependencies. Figure 1 shows an example of a Project-PERT graph, with task durations inside each node. In this example, the edge from programming to testing indicates that testing cannot start until programming is completed. In a Project-PERT graph, the total project duration is the sum of activity delays along the path with the longest delay from source to sink. Given activity completion times, the end-to-end project duration can be computed by running the longest path algorithm on the PERT network. The pseudocode for computing end-to-end latency using topological sort and dynamic programming is included in the supplementary material.



(a) A Trace (b) The Trace-PERT Graph

Figure 2: A trace consists of 5 microservices and the corresponding Trace-PERT Graph

4.2 Trace-PERT Graph

We introduce the Trace-PERT Graph, a variant of the PERT Graph suitable for microservice traces. A Trace-PERT Graph is a Directed Acyclic Graph (DAG) where nodes represent tasks and directed edges represent task dependencies. Our definition differs slightly from Tam et al. (2023) in that we represent microservice tasks as nodes instead of edges, resulting in a smaller graph size and more straightforward derivation of end-to-end latency estimates.

A Trace-PERT Graph can be automatically constructed from trace data (See Figure 2b) where the trace can be collected by distributed tracing tools such as Jaeger (jae 2015). The source node represents the start of the trace (client sending an HTTP request), while the sink node represents the end of the trace (client receiving the HTTP response).

In a Trace-PERT Graph, there are three types of edges: (1) a microservice sending a request to another microservice (e.g. A1 to B1), (2) a microservice sending a response back to another microservice (e.g. D1 to C4), and (3) a dependency between tasks within a microservice (e.g. C2 to C3). The duration of a node represents the time between two consecutive task dependencies within the microservice. For instance, the duration of node A4 is equivalent to the time between when microservice A sends a request to microservice C and when microservice A receives a response from microservice C.

Notably, **in a Trace-PERT Graph, all paths from the source to the sink have the same duration due to the inherent computational inductive bias.** This is because not all nodes represent unique independent tasks. Some nodes are solely waiting for a response from another microservice. For example, in Figure 2b, all paths starting from node A1 and ending at node A5 have the same duration of 10ms.

This property differentiates a Trace-PERT graph from common Project-PERT graphs, where nodes represent inde-

Algorithm 1: FastPERT: Compute the end-to-end latency estimate of a trace given the delay estimates of each microservice task.

Input: A Trace-PERT Graph G with (directed) adjacency matrix $\mathbf{A}$, the node features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$, the source node src, the sink node snk, the longest path length between the source and the sink ℓ, differentiable function that maps node features into a node delay estimate $f : \mathbb{R}^d \mapsto \mathbb{R}$
Output: The expected end-to-end latency of the trace.

1: Initialize the node delay estimates $\hat{\mathbf{d}} \in \mathbb{R}^{|\mathcal{V}|}$ with $\hat{\mathbf{d}}_v = f(\mathbf{x}_v) \quad \forall v \in \mathcal{V}$
2: Compute $S = I + A + A^2 + \ldots + A^\ell$
3: $n\_paths \leftarrow S_{src,snk}$ ▷ The number of paths from src to snk
4: $sum\_of\_delay \leftarrow (S_{src,:} \odot S^T_{:,snk}) \cdot \hat{\mathbf{d}}$ ▷ The sum of end-to-end delays over all paths from src to snk
5: **return** $\frac{sum\_of\_delay}{n\_paths}$

pendent activities and path durations can vary.

5 Proposed Method: FastPERT

In this section, we first present the FastPERT latency prediction algorithm, which generates the end-to-end latency prediction based on the PERT graph and the resource utilization of each microservice (refer to section 5.1 for details). Subsequently, in section 5.3, we outline the backward propagation algorithm used for training the model. The comprehensive framework of the proposed FastPERT is depicted in Figure 3.

5.1 End-to-end latency prediction algorithm

In this section, we detail the forward propagation algorithm, assuming that the model parameters have already been provided. FastPERT consists of two steps: (1) estimating the delay of each microservice task, and (2) computing the end-to-end latency estimate for a trace.

Step 1: Delay Estimation We employ a model $f_\theta : \mathbb{R}^{(d+d_r)} \mapsto \mathbb{R}$ to estimate microservice task delays (node-level delays), where the input is a concatenated vector of static node features and resource utilization. Unlike previous methods, our approach allows flexibility in choosing f_θ, requiring only that it be optimizable based on the loss function's gradient with respect to its output. We utilize a gradient boosted tree (GBT) as f_θ in this study. Although GBTs are not inherently differentiable, they can still be employed since their updates primarily depend on the loss function's gradient (discussed in Section 5.3). The delay estimate for each node v is given by:

$$\hat{\mathbf{d}}_v = f_\theta(\mathbf{x}_v || \mathcal{R}_v) \tag{2}$$

where $||$ is the concatenation operator and f_θ is shared across all nodes to minimize model complexity.

Step 2: End-to-end latency estimation For the second step, we predict the end-to-end latency of the trace using task delay estimates $\hat{\mathbf{d}}$. As discussed in section 4.2, in a Trace-PERT graph, adhering to the computational inductive bias, all source-sink paths share the same duration, capturing structural dependencies between microservices. To estimate end-to-end latency, selecting one source-sink path and summing its task delays can lead to accumulated errors and substantial variance due to path sensitivity. Instead, we propose averaging latency estimates across all source-sink paths, effectively combining predictions via ensemble learning.

Enumerating all possible paths in a Trace-PERT graph to estimate average latency can be computationally demanding due to the exponential number of paths with increasing graph size. To address this, we propose an efficient method that avoids enumerating all paths, significantly speeding up the model. Let A be the directed adjacency matrix of the graph, where $A_{uv} = 1$ if there's an edge from u to v, and 0 otherwise. Since the graph is a DAG, we can use powers of A to encode the number of paths between nodes. Specifically, A^k_{uv} is the number of paths from node u to node v that can be reached in exactly k steps. Consider the matrix

$$S = I + A + A^2 + \ldots + A^\ell = \sum_{i=0}^{\ell} A^i \tag{3}$$

, where ℓ is the longest path length from source to sink (ℓ is finite since a Trace-PERT is acyclic). Then, S_{uv} represents the number of paths from node u to node v reachable in ℓ or fewer steps. For the source node (src) and sink node (snk), $S_{src,snk}$ gives the total number of source-sink paths.

We can efficiently estimate the average end-to-end delay from source (src) to sink (snk) as $\hat{y} = T/S_{src,snk}$, where T is the sum of delay estimates over all src-snk paths. We compute T using the following proposition:

Proposition 1.

$$T = (S_{src,:} \odot S^T_{:,snk})\hat{\mathbf{d}} \tag{4}$$

where $\odot$ denotes element-wise product. Notably, the vector $S_{src,:} \odot S^T_{:,snk}$ can be precomputed, enabling efficient end-to-end latency prediction via a single vector operation.

Proof. Let $\mathcal{P}_{src,snk}$ be the set of paths from the source node to the sink node. Then,

$$T = \sum_{P \in \mathcal{P}_{src,snk}} \sum_{v \in P} \hat{\mathbf{d}}_v \tag{5}$$

$$= \sum_{P \in \mathcal{P}_{src,snk}} \sum_{v \in V} \hat{\mathbf{d}}_v \mathbb{1}_{\{v \in P\}} \tag{6}$$

$$= \sum_{v \in V} \hat{\mathbf{d}}_v \left(\sum_{P \in \mathcal{P}_{src,snk}} \mathbb{1}_{\{v \in P\}} \right) \tag{7}$$

$$= \sum_{v \in V} \hat{\mathbf{d}}_v \left(S_{src,v} \cdot S_{v,snk} \right) \tag{8}$$

$$= (S_{src,:} \odot S^T_{:,snk})\hat{\mathbf{d}} \tag{9}$$

where $\mathbb{1}_{\{v \in P\}}$ is an indicator variable denoting whether node v is in path P. The inner sum in equation 7 counts the number of source-sink paths passing through node v, which is equivalent to the product of the number of source-v paths and the number of v-sink paths, as shown in equation 8. □

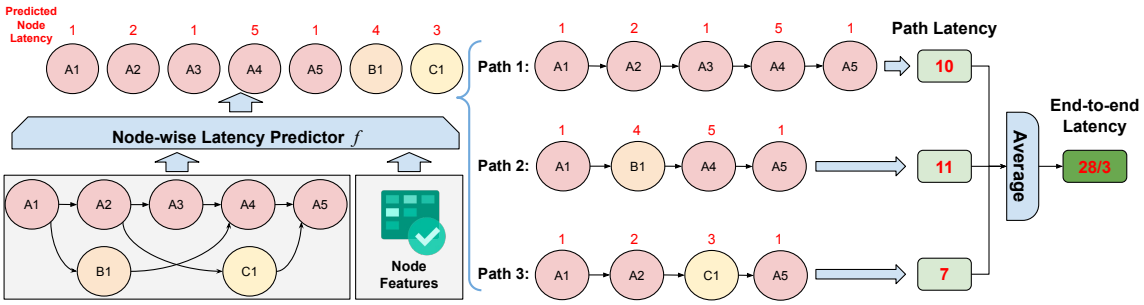
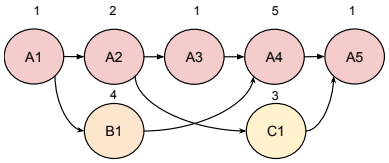


Figure 3: The overall framework of FastPERT. The model follows a two-stage workflow: 1. A microservices trace is transformed into a PERT graph, where each node corresponds to a microservice task. A node-wise latency predictor is subsequently applied to estimate the latency on each node. 2. With node latency estimates, FastPERT derives the total latency of all possible paths from the source to the sink. The final prediction is then obtained by averaging over all path latencies.



(a) A simple TRACE-PERT Graph with source node A1 and sink node A5. The number on top of each node represents the delay estimate of the node.

	A1	A2	A3	A4	A5	B1	C1
A1	0	1	0	0	0	1	0
A2	0	0	1	0	0	0	1
A3	0	0	0	1	0	0	0
A4	0	0	0	0	1	0	0
A5	0	0	0	0	0	0	0
B1	0	0	0	1	0	0	0
C1	0	0	0	0	1	0	0

(b) Adjacency Matrix A

	A1	A2	A3	A4	A5	B1	C1
A1	1	1	1	2	3	1	1
A2	0	1	1	1	2	0	1
A3	0	0	1	1	1	0	0
A4	0	0	0	1	1	0	0
A5	0	0	0	0	1	0	0
B1	0	0	0	1	1	1	0
C1	0	0	0	0	1	0	1

(c) $S = I + A + A^2 + A^3 + A^4$

Figure 4: Illustration of Algorithm 1 on a Trace-PERT graph. The average end-to-end latency estimate over all paths from the source node to the sink node is given by $\hat{y} = \frac{(S_{src,:} \odot S^T_{:,snk}) \cdot \hat{\mathbf{d}}}{S_{src,snk}} = \frac{28}{3}$.

Note that the vector $S_{src,:} \odot S^T_{:,snk}$ can be precomputed, as it does not depend on the parameters of the model, the expected end-to-end latency prediction can be efficiently simplified with a vector-operation.

5.2 An illustrative example

We summarize the key steps of FastPERT in Algorithm 1. An illustrative example of computing the end-to-end latency estimate for a Trace-PERT graph using our algorithm is provided in Figure 4.

Consider a simple Trace-PERT graph with source node A1 and sink node A5, as shown in Fig. 4a. The delay estimates for each node are given by $\hat{\mathbf{d}}^T = [1, 2, 1, 5, 1, 4, 3]$. The adjacency matrix A of the graph is presented in Fig. 4b. There are three paths from the source to the sink with a maximum path length of 4. Using our algorithm, we can derive $S = I + A + A^2 + A^3 + A^4$, as depicted in Fig. 4c. Specifically, we obtain $S_{src,:} = [1, 1, 1, 2, 3, 1, 1]$ (row A1 of matrix S) and $S_{:,snk} = [3, 2, 1, 1, 1, 1, 1]^T$ (column A5 of matrix S). Consequently, we compute $S_{src,:} \odot S^T_{:,snk} = [3, 2, 1, 2, 3, 1, 1]$ and the end-to-end latency estimate is:

$$\hat{y} = \frac{(S_{src,:} \odot S^T_{:,snk})\hat{\mathbf{d}}}{S_{src,snk}} = \frac{28}{3}. \tag{10}$$

5.3 Learning the model parameters

We describe the backward propagation algorithm for learning the model f. The loss function for FastPERT comprises two components: node-level microservice task delay estimates and graph-level end-to-end latency estimate.

For a given Trace-PERT graph G with actual end-to-end latency y, the loss function is defined as:

$$\mathcal{L} = (\hat{y} - y)^2 + \lambda \cdot \sum_{v \in V} (\hat{\mathbf{d}}_v - \mathbf{d}_v)^2 \tag{11}$$

where λ is a hyperparameter that balances the node-level and graph-level losses. The second term acts as a regularization term, encouraging delay estimates to align with actual delays. The gradient of the loss function with respect to the output of f can be calculated using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{d}}_v} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \hat{\mathbf{d}}_v} + \lambda \frac{\partial (\hat{\mathbf{d}}_v - \mathbf{d}_v)^2}{\partial \hat{\mathbf{d}}_v} \tag{12}$$

$$= 2(\hat{y} - y) \frac{S_{src,v} S_{v,snk}}{S_{src,snk}} + 2\lambda(\hat{\mathbf{d}}_v - \mathbf{d}_v) \tag{13}$$

This regularization term embodies multi-task learning, enhancing the model's generalization by leveraging shared information across tasks (Caruana 1997).

5.4 Inference with FastPERT

During inference, for proactive resource allocation, the underlying Trace-PERT graph is not predetermined. We are only provided with the API call type $\mathcal{A}$ and the recent resource utilization of each microservice $\mathcal{R}$. Let $\mathcal{G}_{\mathcal{A}}$ represent

the set of all Trace-PERT graphs associated with the API call $\mathcal{A}$. Consequently, the end-to-end latency prediction is defined as:

$$\hat{y} = f_\theta(\mathcal{A}, \mathcal{R}) = \sum_{G \in \mathcal{G}_\mathcal{A}} p(G|\mathcal{A}) * f_\theta(G, \mathbf{X}, \mathcal{R}) \quad (14)$$

In our implementation, we approximate $p(G|\mathcal{A})$ by the frequency of occurrence of G in $\mathcal{G}_\mathcal{A}$ in the training set.

6 Experiment

In this section, we present the experimental results to demonstrate the effectiveness of FastPERT. We first describe the experimental setup, including the datasets, the evaluation metrics, and the baselines. Subsequently, we present the results FastPERT and compare it with the baselines.

6.1 Datasets

We evaluate our proposed method on real-world traces from Alibaba (Luo et al. 2021) [1]. Node features are listed in the supplementary material. The experiments are conducted on a server equipped with two Intel Core AMD EPYC 7532 32-Core Processor CPU with 1 TB of memory and an Nvidia GeForce RTX 3090 GPU. It's worth noting that GPU is not used in the training of FastPERT, but it is employed in the training process of other models for comparison purposes.

Alibaba Microservice Applications We utilized two real-world trace datasets from Alibaba, collected in 2021 and 2022, respectively. The 2021 dataset, characterized byLuo et al. (2021), comprises 20 million traces spanning over 20 thousand microservices across 10 clusters. After preprocessing (see supplementary material for details), we obtained 1.75 million traces with 65 distinct API calls.

The 2022 dataset, released by Alibaba(Luo et al. 2022a), contains detailed runtime metrics of nearly 20 thousand microservices over a span of 13 days and incorporates additional information, such as the service ID within the call graph and extra runtime metrics like the average response time and normalized read/write call rates to the RPC/Memcached/Database/Message Queue. Regrettably, these additional metrics are not recorded for most microservices. We selected the initial 2 days of the trace and, after preprocessing, obtained 1.50 million traces with 1084 distinct API calls. This dataset poses a greater challenge due to more varied runtime behaviors, larger variance in end-to-end latency.

We partitioned both datasets into training (75%), validation (10%), and test sets (15%). Additional preprocessing steps were performed to ensure data quality and consistency; see supplementary material for details.

6.2 Baseline Models

We compare FastPERT with three baseline models:

- **GBDT**: A gradient boosted tree model that predicts end-to-end latency directly from microservice resource utilization, without considering the PERT graph. (Trained on 64 CPU cores)
- **PERT-GNN**: A graph neural network model that predicts end-to-end latency using the PERT graph and microservice resource utilization. (Details in Tam et al. (2023); trained on Nvidia GeForce RTX 3090)
- **ProjectPERT**: A variant of FastPERT that uses the same node predictor but the long path algorithm (see Supplementary Material for details) is used to compute end-to-end latency estimates. (Trained on 64 CPU cores)

For FastPERT, we tune the hyperparameter λ over the range [0, 1] and report the Test Mean Absolute Percentage Error (MAPE) and Test Mean Absolute Error (MAE) based on the best performance on the validation set. We do not compare FastPERT with FIRM (Qiu et al. 2020), SAGE (Gan et al. 2021) and GRAF (Park et al. 2021a) as they focus on different aspects: FIRM optimizes resource allocation using reinforcement learning; SAGE identifies root causes of QoS violations post-occurrence. In contrast, FastPERT aims to proactively predict end-to-end latency. In addition, the lack of publicly available source code for GRAF also makes fair and extended comparisons with FastPERT difficult.

6.3 Results

Table 2 presents the experimental results on the test set, comparing PERT-GNN with other machine learning models. We observe that FastPERT outperforms GBDT in terms of Mean Absolute Percentage Error (MAPE) and Mean Absolute Error (MAE), as it leverages API-awareness and structural relationships between microservices. Although PERT-GNN achieves similar performance to FastPERT, it requires significantly more computational resources: FastPERT is orders of magnitude faster, does not necessitate a GPU for training, and predicts 1.5 million traces within 10 seconds using CPUs, whereas PERT-GNN takes over 10 minutes with a GPU. This disparity stems from FastPERT's simpler approach with $O(n)$ complexity, compared to PERT-GNN's complex graph transformers with $O(n^2)$ complexity.

To demonstrate the statistical significance of the improvement of FastPERT over GBDT, we evaluated both models on the same 5-fold cross-validation split of the Alibaba 2022 dataset. We used a one-sided paired t-test to determine whether the average performance, measured by Mean Absolute Percentage Error (MAPE) and Mean Absolute Error (MAE), differed significantly between the models. For MAPE, the null hypothesis was rejected with a p-value of 0.0196 and a t-statistic of -3.017. For MAE, the null hypothesis was rejected with a p-value of 0.0000137 and a t-statistic of -21.570.

ProjectPERT underperforms compared to PERT-GNN and FastPERT due to significant overestimation of end-to-end latency. Since ProjectPERT selects the paths with the largest delay estimations, the path with larger overestimation is more likely to be chosen. Such overestimation originates from limited model capacity or data noise, resulting in imprecise estimation of the delay of microservice tasks at the node level. This results in a substantial overestimation of the end-to-end latency estimate and the issue exacerbates when the graph is large, as the overestimation can accumulate along an extended path. In contrast, FastPERT mitigates this by aver-

[1]The dataset is available at https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021

Dataset	Model	MAPE	MAE (ms)	Training Time (minutes) †	Inference Time ‡	Hardware
Ali-2021	GBDT	$18.69\% \pm 0.38\%$	2.12 ± 0.010	3.6 ± 0.8	<10 secs	CPU
	PERT-GNN	$11.57\% \pm 0.82\%$	1.60 ± 0.022	631.1 ± 181.7	10+ mins	GPU
	ProjectPERT	$13.73\% \pm 0.89\%$	1.68 ± 0.032	19.4 ± 3.2	<10 secs	CPU
	FastPERT	$11.59\% \pm 0.53\%$	1.61 ± 0.016	6.5 ± 3.0	<10 secs	CPU
Ali-2022	GBDT	$17.29\% \pm 0.41\%$	0.87 ± 0.013	4.4 ± 1.2	<10 secs	CPU
	PERT-GNN	$12.84\% \pm 0.89\%$	0.63 ± 0.074	696.0 ± 170.7	10+ mins	GPU
	ProjectPERT	$16.17\% \pm 0.80\%$	0.80 ± 0.070	21.2 ± 5.7	<10 secs	CPU
	FastPERT	$11.36\% \pm 0.51\%$	0.62 ± 0.017	7.4 ± 4.2	<10 secs	CPU

Table 2: Results for the end-to-end latency prediction on the Alibaba datasets (2021 and 2022)
† Training time is calculated based on training with one set of hyperparameters. ‡ Average inference time over 1.5 million traces.

Aggregation Scheme	MAPE	MAE (ms)
Entering-Service-Path Scheme	14.44%	0.6579
Longest-Delay-Path Scheme	16.17%	0.8017
All-Paths-Averaging Scheme	**11.36%**	**0.6178**

Table 3: Performance of FastPERT with different aggregation methods

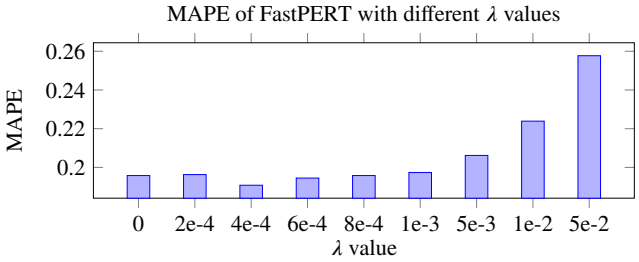


Figure 5: MAPE of FastPERT with different λ values. Lower MAPE values indicate better performance.

aging latency estimates across all source-sink paths, effectively employing an ensemble learning approach that combines multiple predictions.

7 Ablation Studies

In this section, we carry out ablation studies to scrutinize the influence of various components of FastPERT on the model's performance. We aim to address the following queries: (1) The effect of averaging paths for end-to-end latency estimation, (2) The sensitivity of the hyperparameter λ within FastPERT's loss function.

7.1 The impact of end-to-end latency estimate via paths averaging

We evaluate FastPERT against three aggregation schemes: (a) Entering-Service-Path Scheme, which selects the path associated with the entering microservice (e,g., the A1 → A2 → A3 → A4 → A5 path in Fig. 2b); (b) Longest-Delay-Path Scheme, which chooses the path with the longest delay estimate (i.e. the ProjectPERT); and (c) All-Paths-Averaging Scheme (i.e., FastPERT), which computes the average latency estimates of all paths from source to sink.

The results in Table 3 show that the All-Paths-Averaging Scheme outperforms the other two schemes. This superiority stems from their limitations: the Entering-Service-Path Scheme neglects resource utilization for other relevant microservices, while the Longest-Delay-Path Scheme is prone to overestimation in large graphs. In contrast, averaging latencies of all paths offers two advantages. Firstly, it acts as an ensemble method that improves predictive performance by combining predictions from different paths. Secondly, it enables efficient calculation via a weighted sum operation, eliminating the need for sampling as discussed in Section 5.1.

7.2 Sensitivity of λ

We investigate the sensitivity of FastPERT's hyperparameter λ, which regulates the node-level delay estimates to match ground truth delays (Section 5.3). Varying λ from 0 to 1, we evaluate FastPERT's performance on the Alibaba test set (2022) and present the results in Figure 5. We observe that increasing λ values lead to higher Training MAPE, mitigating model complexity. However, excessive λ values can cause underfitting, while diminutive values can cause overfitting, both resulting in higher Test MAPE. Our experiments set $\lambda = 4e-4$, achieving the lowest Validation and Test MAPE.

8 Conclusion and Future Works

We introduce FastPERT, a fast improvement of PERT-GNN. The pivotal components of FastPERT encompass predicting the delay of each microservice task with a parameter-shared model and harnessing the computational inductive bias of the PERT graph to compute the end-to-end latency estimate of the trace by averaging the end-to-end latency estimates of all paths from the source to the sink. This averaging approach can be perceived as an ensemble learning method, and it can be implemented efficiently with a vector operation. Experiments substantiate that the proposed method is orders of magnitude faster than PERT-GNN and achieves comparable performance. Future works include extending FastPERT to handle quantile estimation for tail latency prediction.

Acknowledgements

We sincerely thank the anonymous AAAI'25 reviewers for their insightful suggestions. This work is supported in part by the Science and Technology Development Fund of Macau (0024/2022/A1, 0071/2023/ITP2), the National Natural Science Foundation of China under Grant #62202284b and in part by Shanghai Pujiang Program under Grant #22PJ1404000.

References

2015. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/. Accessed: 2024-08-01.

2017. Microservices - The Airbnb Tech Blog. https://medium.com/airbnb-engineering/tagged/microservices/. Accessed: 2024-08-01.

2019. Microservices - The Netflix Tech Blog. https://netflixtechblog.com/tagged/microservices/. Accessed: 2024-08-01.

Bhasi, V. M.; Gunasekaran, J. R.; Thinakaran, P.; Mishra, C. S.; Kandemir, M. T.; and Das, C. 2021. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, 153–167.

Caruana, R. 1997. Multitask learning. *Machine learning*, 28: 41–75.

Chow, K.-H.; Deshpande, U.; Seshadri, S.; and Liu, L. 2022a. DeepRest: deep resource estimation for interactive microservices. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 181–198.

Chow, K.-H.; Deshpande, U.; Seshadri, S.; and Liu, L. 2022b. DeepRest: deep resource estimation for interactive microservices. In *Proceedings of the Seventeenth European Conference on Computer Systems*, 181–198.

Chow, M.; Meisner, D.; Flinn, J.; Peek, D.; and Wenisch, T. F. 2014. 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)*, 217–231.

Cohen, I.; Chase, J. S.; Goldszmidt, M.; Kelly, T.; and Symons, J. 2004. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*, volume 4, 16–16.

Dragoni, N.; Giallorenzo, S.; Lafuente, A. L.; Mazzara, M.; Montesi, F.; Mustafin, R.; and Safina, L. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, 195–216.

Gan, Y.; Liang, M.; Dev, S.; Lo, D.; and Delimitrou, C. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 135–151.

Gan, Y.; Zhang, Y.; Hu, K.; Cheng, D.; He, Y.; Pancholi, M.; and Delimitrou, C. 2019. 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*, 19–33.

Gias, A. U.; Casale, G.; and Woodside, M. 2019. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 1994–2004. IEEE.

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

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

Kuberntes. 2014. Kubernetes. https://kubernetes.io.

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

Liu, Y.; Xu, H.; and Lau, W. C. 2020. Accordia: Adaptive Cloud Configuration Optimization for Recurring Data-Intensive Applications. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 831–841. IEEE Computer Society.

Liu, Y.; Xu, H.; and Lau, W. C. 2022. Online Resource Optimization for Elastic Stream Processing with Regret Guarantee. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP '22)*.

Luo, S.; HL, X.; Ye, K.; Xu, G.; Zhang, L.; He, J.; Yang, G.; and Xu, C. 2023. 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*.

Luo, S.; Xu, H.; Lu, C.; Ye, K.; Xu, G.; Zhang, L.; Ding, Y.; He, J.; and Xu, C. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, 412–426.

Luo, S.; Xu, H.; Ye, K.; Xu, G.; Zhang, L.; He, J.; Yang, G.; and Xu, C. 2022a. The Power of Prediction: Microservice Auto Scaling via Workload Learning. In *Proceedings of the ACM Symposium on Cloud Computing*.

Luo, S.; Xu, H.; Ye, K.; Xu, G.; Zhang, L.; Yang, G.; and Xu, C. 2022b. The power of prediction: Microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, 355–369.

Merkel, D.; et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 239(2): 2.

Park, J.; Choi, B.; Lee, C.; and Han, D. 2021a. GRAF: a graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, 154–167.

Park, J.; Choi, B.; Lee, C.; and Han, D. 2021b. GRAF: a graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, 154–167.

Qiu, H.; Banerjee, S. S.; Jha, S.; Kalbarczyk, Z. T.; and Iyer, R. K. 2020. {FIRM}: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 805–825.

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

Shi, J.; Wang, J.; Fu, K.; Chen, Q.; Zeng, D.; and Guo, M. 2022. QoS-awareness of Microservices with Excessive Loads via Inter-Datacenter Scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 324–334. IEEE.

Shi, J.; Zhang, H.; Tong, Z.; Chen, Q.; Fu, K.; and Guo, M. 2023. Nodens: Enabling Resource Efficient and Fast {QoS} Recovery of Dynamic Microservice Applications in Datacenters. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 403–417.

Sriraman, A.; Dhanotia, A.; and Wenisch, T. F. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, 513–526.

Tam, D. S. H.; Liu, Y.; Xu, H.; Xie, S.; and Lau, W. C. 2023. 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*, 2155–2165.

Vance, M. M. 1963. PERT and CPM: a selected bibliography (Volume Committee of Planning Librarians. Exchange bibliography, no. 53) online. *Harvard Business Review*, 41(5): 98–108.

Wang, Z.; Zhu, S.; Li, J.; Jiang, W.; Ramakrishnan, K.; Zheng, Y.; Yan, M.; Zhang, X.; and Liu, A. X. 2022. DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems. In *Proceedings of the 13th Symposium on Cloud Computing*, 16–30.

Weng, L.; Hu, Y.; Huang, P.; Nieh, J.; and Yang, J. 2023a. Effective Performance Issue Diagnosis with Value-Assisted Cost Profiling. In *Proceedings of the Eighteenth European Conference on Computer Systems*, 1–17.

Weng, L.; Hu, Y.; Huang, P.; Nieh, J.; and Yang, J. 2023b. Effective Performance Issue Diagnosis with Value-Assisted Cost Profiling. In *Proceedings of the Eighteenth European Conference on Computer Systems*, 1–17.

Weng, L.; Huang, P.; Nieh, J.; and Yang, J. 2021. Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 193–207.

Yang, Z.; Nguyen, P.; Jin, H.; and Nahrstedt, K. 2019. MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*, 122–132. IEEE.

Zhang, Y.; Hua, W.; Zhou, Z.; Suh, G. E.; and Delimitrou, C. 2021. 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*, 167–181.