PERT-GNN: Latency Prediction for Microservice-based Cloud-Native Applications via Graph Neural Networks

Da Sun Handason Tam
The Chinese University of Hong Kong
Hong Kong
tds019@ie.cuhk.edu.hk

Yang Liu
Shanghai University
China
yangliu_cs@shu.edu.cn

Huanle Xu*
University of Macau
Macau
huanlexu@um.edu.mo

Siyue Xie
The Chinese University of Hong Kong
Hong Kong
xs019@ie.cuhk.edu.hk

Wing Cheong Lau
The Chinese University of Hong Kong
Hong Kong
wclau@ie.cuhk.edu.hk

ABSTRACT

Cloud-native applications using microservice architectures are rapidly replacing traditional monolithic applications. To meet end-to-end QoS guarantees and enhance user experience, each component microservice must be provisioned with sufficient resources to handle incoming API calls. Accurately predicting the latency of microservices-based applications is critical for optimizing resource allocation, which turns out to be extremely challenging due to the complex dependencies between microservices and the inherent stochasticity. To tackle this problem, various predictors have been designed based on the Microservice Call Graph. However, Microservice Call Graphs do not take into account the API-specific information, cannot capture important temporal dependencies, and cannot scale to large-scale applications.

In this paper, we propose PERT-GNN, a generic graph neural network based framework to predict the end-to-end latency for microservice applications. PERT-GNN characterizes the interactions or dependency of component microservices observed from prior execution traces of the application using the Program Evaluation and Review Technique (PERT). We then construct a graph neural network based on the generated PERT Graphs, and formulate the latency prediction task as a supervised graph regression problem using the graph transformer method. PERT-GNN can capture the complex temporal causality of different microservice traces, thereby producing more accurate latency predictions for various applications. Evaluations based on datasets generated from common benchmarks and large-scale Alibaba microservice traces show that PERT-GNN can outperform other models by a large margin. In particular, PERT-GNN is able to predict the latency of microservice applications with less than 12% mean absolute percentage error.

*Corresponding author

KDD '23, August 6–10, 2023, Long Beach, CA, USA

ACM ISBN 979-8-4007-0103-0/23/08…$15.00
https://doi.org/10.1145/3580305.3599465

CCS CONCEPTS

• **Computing methodologies** → **Learning latent representations**; **Neural networks**.

KEYWORDS

delay prediction, microservices, cloud computing, graph neural networks, graph transformers, machine learning

ACM Reference Format:
Da Sun Handason Tam, Yang Liu, Huanle Xu, Siyue Xie, and Wing Cheong Lau. 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 (KDD '23), August 6–10, 2023, Long Beach, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3580305.3599465

1 INTRODUCTION

The Microservice architecture is an important software development framework in the cloud native area. It structures an application as a collection of small, independent, self-contained services that can be deployed and managed independently. Companies like Netflix, Twitter, Facebook, and Airbnb use this architecture to build their cloud-based applications [4, 5, 16, 47].

Compared to monolithic, microservice framework offers the advantage of flexible resource scaling. Figure 1 shows a social network application composed of multiple microservices forming a Microservice Call Graph (MCG). User clients send API call requests to the application through the API endpoints [11]. Microservices with dedicated functions are then invoked to fulfil specific user requests. When the application is under growing load, the service provider can locate and scale individual microservices experiencing heavy load instead of scaling the whole application.

Despite the flexibility, the microservice architecture brings a great challenge in providing end-to-end QoS guarantees [8] in production environments, as a single user request needs to be handled by hundreds of fine-grained microservices [35]. Today's production clusters typically overprovision resources to provide such a guarantee, which can easily lead to low overall resource utilization, e.g., the CPU utilization in Alibaba microservice clusters is as low as 20% [34]. Designing more efficient resource scaling solutions to meet QoS guarantees therefore becomes critical.

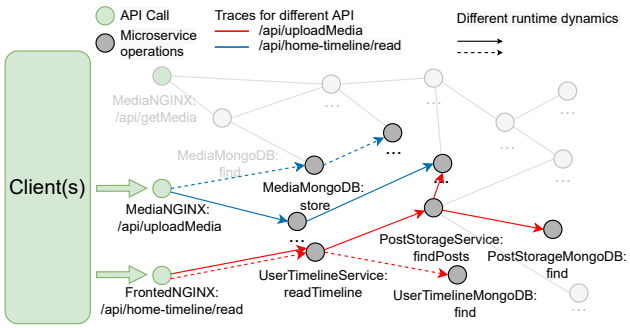
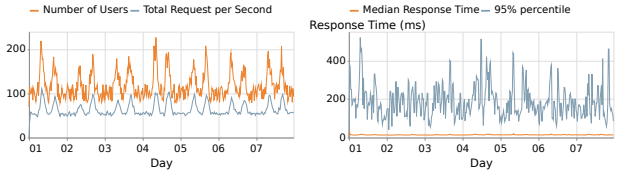


Figure 1: A microservice-based application handles multiple API calls from the clients. Each API call could have different runtime behaviors. A microservice operation can invoke other microservice operations in sequential or parallel.



(a) Number of users and requests. **(b) Response Times (ms).**

Figure 2: The 95th percentile end-to-end latency/ response time in (b) varies significantly as RPS (the number of requests per second) in (a) fluctuates over time.

There mainly exist two approaches of resource scaling for microservices: reactive and proactive. Reactive methods tune resources on-the-fly based on the performance feedback from the system [25, 56]. By contrast, proactive methods build predictive models to estimate the tail (e.g., 95th percentile) end-to-end latency of user requests under different resource configurations in advance and then choose the best configuration [42, 55]. While easy to implement, reactive solutions can easily cause QoS violations because obtaining the end-to-end feedback can be too late for scaling especially when the call chain is long. As a consequence, proactive scaling tends to be a more promising approach for microservices in recent years.

The ability to achieve efficient proactive scaling is highly dependent on accurate prediction of tail end-to-end latency, which however, remains quite challenging. First, the component microservices within the same application can form complex interactions between each other. The response of one microservice would highly affect its downstream microservices, which in turn would affect the end-to-end delay of the entire trace. As a consequence, failing to capture the complex dependencies between microservices would lead to inaccurate prediction. Second, the tail latency is not stable due to its inherent stochasticity because it accounts for the worst-case scenario [8]. As shown in Fig. 2, although the workload of a social network application changes periodically throughout the entire week (Fig. 2a), the median latency (response time) stays between 13ms and 17ms (Fig. 2b, the orange curve). As a comparison, the 95th percentile latency presents a significant fluctuation, varying between 60ms and 510ms (Fig. 2b, the blue curve).

In the literature, many efforts have been devoted to capturing the complex dependencies between microservices and their operations to predict the tail end-to-end latency [19, 38, 55]. In particular, these works mainly adopt deep neural networks to model the dependencies based on the whole MCG. However, there are three limitations with MCG-based predictors as explained below.

(1) MCG is not **API-aware** and contains a lot of irrelevant information that can result in suboptimal prediction results. Given an API call, only a small subset of microservices in the MCG are involved. For example the `readTimeline` is irrelevant to the execution of the API call sent to `MediaNGINX`, as we can see in Figure 1. Without filtering the noise, the latency prediction will be highly inaccurate. Besides, for large applications, failing to capture the relevant microservices with API-awareness can easily incur large computation overhead.
(2) MCG is a static graph and does not capture the **temporal dynamics** among calls. For example, the `PostStorageService` microservice subsequently call two other microservices, but MCG does not show which one is invoked first. In addition, MCG does not distinguish whether these two other microservices are invoked in parallel or sequentially.
(3) MCG cannot identify **runtime dynamics**. Given the same API call, the runtime dynamics can be different depending on the input parameters. For example, the `/api/home-timeline/read` API call can be made with or without the `userId` parameter. This would affect whether the `UserTimelineMongoDB:find` microservice operation is called, which in turn would affect the end-to-end latency of the `/api/home-timeline/read` API call. However, MCG is unable to contain this information since it couples all runtime behaviors using a single static graph.

Due to these limitations, current predictors have low prediction accuracy, leading to QoS violations and wasted resources. To address this, we introduce PERT-GNN, a graph neural network (GNN) model that predicts the end-to-end latency of microservice-based applications based on API call and resource utilization at a given time. PERT-GNN is a generic model that works for any microservice-based application and only requires a graph with node features and/or graph features as input. In this work:

- Graphs (Span/PERT) are constructed from the traces, which are collected by distributed tracing tools e.g. Jaeger [1]. See Sec. 3 for details.
- Node features refer to resource utilizations (CPU/ RAM/ Disk/ Network loading) of EACH component microservice of an application. These features are periodically scraped by data telemetry tools e.g. Prometheus [6].
- Graph features refer to the type of the API-call, the timestamp of the trace, and a time-series of the number of requests/response time in history.

PERT-GNN is API-aware and can incorporate the temporal dependencies between microservices for a given API call. Our contributions are summarized as follows:

- To address limitations (1)-(2), we propose a new design based on the program Evaluation and Review Technique (PERT) to construct graphs that are API-aware and able to capture

the temporal dependencies between microservices. The construction of the graphs is data-driven, based on raw microservice trace data collected by distributed tracing tools such as Jaeger [1] without any manual intervention.
- To address limitation (3), we propose a general graph mixture approach that identifies different runtime dynamics for the same API call. This allows us to characterize the relationship between relevant microservices specific to different runtime dynamics within the same API call.
- We propose PERT-GNN, a transformer-based GNN that predicts different end-to-end latency percentiles for microservice applications by capturing structural and positional information of microservices and be trained end-to-end.
- Our model was evaulated on large-scale traces from a benchmark suite for cloud microservice [18] and real-world Alibaba traces [34]. Our model can predict end-to-end latency with less than 12% mean absolute percentage error, which is an improvement of 12% compared to existing predictors.

2 RELATED WORK

Nowadays, performance estimation has already been a fundamental problem for distributed tracing [13, 23, 30], root cause analysis [19, 24, 51], and resource allocation [31–33, 44]. A lot of elaborate models [9, 14, 20] have appeared to depict the microservice architecture. For example, *ATOM* [20] and *Kraken* [9] 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. They usually assume the latency or resource usage of each component microservice known beforehand or following a simple function with respect to the workload and resource configuration. These model-driven approaches do not require much training effort. However, their assumptions are too strong and the input is based on MCG, which can cause high estimation errors.

Machine learning approaches [19, 53, 55] are widely adopted to design accurate performance estimation models. In particular, *Sinan* [55] and *Seer* [19] directly use CNN combined with Boosted Trees and CNN with LSTM to predict the end-to-end latency under different workloads, respectively. These approaches treat the microservice application as a black-box and ignore the dependencies between microservice components (i.e. not API-aware). Such data-driven approaches require tremendous training data and are hard to deploy in large-scale production environments.

In this sense, a body of works [17, 33, 42, 44] combine the advantage of model-driven and data-driven approaches. They can consider MCGs while learning the complex mapping from the workload to the computation performance. For example, *Sage* [17] uses a Causal Bayesian Network (CBN) to capture the dependencies between components and trains a Graphical Variational Auto-Encoder (GVAE) to generate QoS-violation counterfactuals. *Firm* [42] defines the critical path and uses the Reinforcement Learning to improve the latency along the critical path. Meanwhile, a request only invokes some paths of components cross the MCG according to the business logic. Without focusing on such footprint of specific APIs may involve too much noise in the machine learning model. *DeepRest* [12] consider the span graphs, graphs containing the logic unit of requests, to enhance the accuracy of the estimation model. Although it is API-aware, it still ignores the temporal dependencies between microservice operations. It also fails to capture the runtime dynamics of the same API call since it couples all runtime behaviors of the same API call together.

Table 1: Related Works for Performance Estimation

Source	Tech.	MCG	Span Graph	Perf. Metric	Estimated Perf.
Kraken [SoCC'21]	M	✓	✗	LC	CCR:99.7%
ATOM [ICDCS'19]	M	✓	✗	TPS	MAPE:5.05%
Seer [ASPLOS'19]	D	✗	✗	LC	CCR:90%
MIRAS [ICDCS'19]	D	✗	✗	LC	MAPE:14.3%
Sifter [SoCC'19]	D	✗	✗	LC	RMSE:6.81ms
Sage [ASPLOS'21]	D	✗	✗	LC	CCR:94%
Sinan [ASPLOS'21]	D	✓	✗	LC	RMSE:25.9ms
DeepRest[EuroSys'22]	D	✓	✓	CPU	MAPE:11.9%
GRAF [CoNEXT'21]	D	✓	✗	LC	MAPE:26.9%
AutoPilot[EuroSys'20]	M	✓	✗	CPU	CCR:99.5%
FIRM [OSDI'20]	M	✓	✗	LC	CCR:93.5%
Our work	**M+D**	✓	✓	**LC**	**MAPE:11.8%, RMSE:1.658ms**

"LC" represents Latency; "TPS" represents Transaction per Second; "CPU" represents CPU utilization for processing the application; "M + D" represents mixture of model-driven and data-driven; "MAPE" represents Mean Absolute Percentage Error; "CCR" represents Correct Classification Rate;

Graph neural network (GNN) [45] is used to process graph-structured trace information. *GRAF* [39] predicts application latency using graph node embeddings and fully connected layers, but it doesn't consider API footprints. To address this, we propose PERT-GNN, which constructs a PERT graph from span graphs and uses GNN to provide API-aware latency prediction for microservice applications. For detailed comparison, see Table. 1.

3 GRAPH CONSTRUCTION FOR MICROSERVICE TRACES

Our work aims to predict the end-to-end latency of an API call for microservice-based applications. In general, the latency of a specific API call is the amount of time it takes from when a request is sent by the user to the time it takes for the response to be received. It is a challenging task as the latency depends on the complicated invocation dependencies among a plenty of microservices, while the run-time dynamics of different API call vary greatly. To capture the complex dependencies between microservices, previous works proposed to model the traces as a graph. In this section, we introduce different graph construction methods for representing microservice traces. We will first introduce MCG and Span Graph (Sec. 3.1 and Sec. 3.2), which are the most popular graph structures for representing microservice traces. Then we will propose to build PERT graphs (Sec. 3.3) to model the relationships among microservices, which resolve the limitations of MCG and Span graphs. Finally, since there is a one-to-many relationship between the API call and the Span/PERT graphs due to the dynamic runtime behavior of API calls, we

Table 2: Comparison across different graphs structures for trace representation

	MCG Graph (Sec. 3.1)	Span Graph (Sec. 3.2)	PERT Graph (Sec. 3.3)	Span + PERT (Sec. 4.1)
Directed	✓	✓	✓	✓
Acyclic	✗	✓	✓	✓
Tree	✗	✓	✗	✗
Scalability	✗	✓	✓	✓
Temporal orderings	✗	✗	✓	✓
Distinguish parallel/ sequential calls	✗	✗	✓	✓
API-aware	✗	✓	✓	✓
Run-time Dynamics	✗	✗	✗	✓
Work	[38]	[11], Ours	Ours	Ours

apply graph-based algorithms to identify different runtime variants of the Span/ PERT Graphs efficiently in section 4.1. We summarize the comparison of different graph structures in Table. 2.

3.1 Microservice Call Graph

With insider knowledge of the microservice application structure, one can construct a graph where each node represents a microservice, and each edge represents the communication between two microservices. Figure 1 shows an example of a MCG for the Social Network Microservices application. MCG has several fundamental limitations, as explained in Sec. 1.

3.2 Span Graph

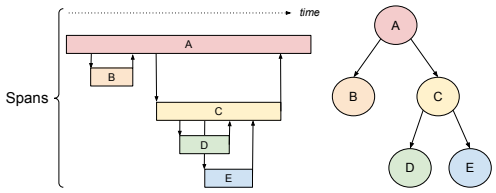
Due to the inability of the MCG to capture different invocation paths for different API calls, DeepRest [11] proposed to use the Span Graph to represent the microservice traces. Definitions are as follows:

Definition 3.1. A **span** represents a logical unit of work that has a microservice name, an operation name, the start time of the operation, the duration, and the parent Span. Spans may be nested and ordered to model causal relationships between microservice operations [1]. A **trace** is made up of one or more spans. Figure 3a shows a visualization of a trace.

Definition 3.2. A **Span Graph** is a directed rooted tree where nodes are the spans and edges represent the calling relationships. The Span Graph can be constructed directly from the trace where the edges are the set of (parent Span, child Span) pairs. Figure 3b shows an example of a Span Graph.

We denote the Span Graph of a trace as $S = (V_S, E_S)$ where V_S is the set of spans and E_S is the corresponding set of edges.

Every span has only one parent span, except the root span. The root span/ node is the user span which represents the process of the user sending and receiving an HTTP request. The end-to-end delay is the duration of the root node, which is our target for prediction. Although Span Graph can capture the calling hierarchy between microservices and eliminate irrelevant microservices, it



(a) Trace (b) Span Graph

Figure 3: A span represents a logical unit of work in Jaeger that has an operation name, the start time of the operation, and the duration. Spans may be nested and ordered to model causal relationships.

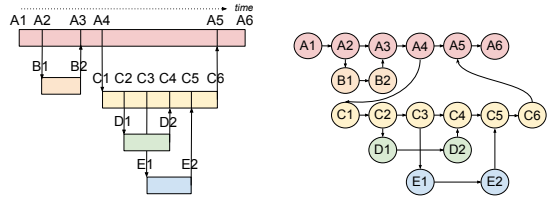
ignores the temporal orderings of the operations in the trace and cannot distinguish parallel and sequential calls in the graph. For example in Figure 3a, C calls D before calling E, but the Span Graph (Fig. 3b) cannot capture this information. In addition, considering the cases that A invokes C until B is finished, versus C invokes D and subsequently invokes E without waiting D to finish. The Span Graph cannot distinguish these two cases as they both have the same Span Graph structure (two spans connected by a common parent span). Consider an image retrieval system with three microservices: (A) search query, (B) images ranking, and (C) image filtering. For a user requesting the "top-K largest size monkey images among all images," the calling order should be A → C → B. For a request to "find all monkey images among top-K largest size images," the calling order should be A → B → C. The Span graph structure remains the same, with A as the parent and B and C as the leaves. However, it does not reflect the calling order. Since end-to-end latency is highly dependent on the calling order of microservices, failing to capture the temporal dynamics of the operations will lead to a suboptimal prediction of the response time.

3.3 PERT Graph

To address the limitations of the MCG and Span Graph, we propose to use the PERT Graph to represent the microservice traces, which captures the temporal dynamics of microservices call intrinsically. The idea is inspired by the program evaluation and review technique (PERT), a statistical tool for project management. PERT was created to evaluate and represent the tasks necessary to complete a certain project [48]. It is a technique for examining the tasks involved in finishing a given project, particularly the amount of time required to perform each work, and determining the time required for the whole project [48]. PERT analysis is based on Project Control's network diagram, a Directed Acyclic Graph (DAG), where the nodes represent events (or completed phase, or stages) and the arrow represents the activities necessary to reach the nodes.

Since PERT Control's network diagram is very useful in identifying the time required to complete the total project, we present a generalization of PERT to model microservice traces. Towards this end, we define a DAG where nodes are stages and edges are activities or work necessary to reach the next stages. Concretely, we define a PERT Graph as follows:

Definition 3.3. A **PERT Graph** is a connected directed acyclic graph (DAG) with a single source and a single sink. In a PERT Graph, nodes are the stages/ milestones and edges represent the



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

Figure 4: Illustration of the PERT Graph construction.

activities/ work/ tasks that must be performed to reach the next stages. In the context of microservice traces, a stage is the moment when a microservice invokes a call to or receives a response from another microservice. The activities will be internal business logic of the microservice. A PERT Graph can be constructed from the Span Data (See Algorithm 1 for details). The source node represents the start of the trace, i.e. the stage where a client sends an HTTP request. The sink node represents the end of the trace, i.e. the stage where the client receives the HTTP response.

PERT Graph divides a span into multiple stages to capture the order of requests and responses. This helps distinguish between sequential and parallel calls, which are represented differently.

3.3.1 Algorithm for Constructing PERT Graph. The algorithm of constructing a PERT Graph is given in Algorithm 1. It takes a Span Graph $S = (V_S, E_S)$ as input, and returns a PERT Graph $\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P})$ with respect to S where $v_{s,i} \in V_\mathcal{P}$ represents the i-th stage of span s, and a hashmap $timestamps$ where $timestamps.get(v_{s,i})$ represents the corresponding timestamp. In summary, we create the PERT Graph from a trace by first splitting each span into multiple stages (line 5 of Alg. 1), and then adding edges between the stages of the same span (line 6 of Alg. 1) and between the stages of different spans (Fig. 4b and line 9-26 of Alg. 1).

Let $|V_S|$ and $|E_S|$ be the number of vertices and edges in the span graph S respectively, Algorithm 1 has a time complexity of $O(|V_S|)$, since line 3 - 7 takes $O(|V_S|+|E_S|)$ time and line 9 - 26 takes $O(|E_S|)$ time. Since Span graph is a tree, $|E_S| = |V_S| - 1$, which implies Algorithm 1 takes $O(|V_S|)$ time. Furthermore, we demonstrate the correctness of Algorithm 1 in the following theorem.

Theorem 3.4. *Algorithm 1 always produces a DAG.*

Proof. First, the vertices of the PERT Graph sorted by their $timestamps$ are always in topological order. This is because the edges are added in temporal order in algorithm 1, that is for any edge (a, b) added to the PERT graph in algorithm 1, $timestamps.get(a) \leq timestamps.get(b)$. Since temporal order is a total order, that is, it satisfies the following properties for all timestamps $t1$, $t2$, and $t3$:

(1) $t1 \leq t1$,
(2) $t1 \leq t2$ and $t2 \leq t3 \implies t1 \leq t3$,
(3) $t1 \leq t2$ and $t2 \leq t1 \implies t1 = t2$, and
(4) $t1 \leq t2$ or $t2 \leq t1$.

This implies that the vertices of the PERT Graph sorted by their $timestamps$ are in total order. Therefore, there exists a topological ordering of the PERT Graph generated by algorithm 1. Since a topological ordering is possible if and only if the graph is a DAG, the PERT Graph generated by Algorithm 1 is always a DAG. This completes the proof of Theorem 3.4. □

Algorithm 1 PERT Graph construction

Input: A Span Graph $S = (V_S, E_S)$, the start time and the end time of each span $s \in V_S$.
Output: A PERT Graph $\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P})$ with respect to S where $v_{s,i} \in V_\mathcal{P}$ represents the i-th stage of span s, and a hashmap $timestamps$ where $timestamps.get(v_{s,i})$ represents the corresponding timestamp.

1: Initialize $V_\mathcal{P} = \emptyset$ and $E_\mathcal{P} = \emptyset$.
2: Initialize $n\_stages[]$ to be an empty array of size $|V_S|$
3: **for** $s \in V_S$ **do**
4: $\quad n\_stages[s] \leftarrow v.outdegree() * 2 + 2$
5: $\quad V_\mathcal{P} \leftarrow V_\mathcal{P} \cup \{v_{s,i} | i \in [1, n\_stages[s]]\}$ ▷ Create copies of node s and add them to the PERT Graph
6: $\quad E_\mathcal{P} \leftarrow E_\mathcal{P} \cup \{(v_{s,i}, v_{s,i+1}) | i \in [1, n\_stages[s] - 1]\}$ ▷ Add edges between the copies of node s to the PERT Graph
7: **end for**
8: Initialize an empty hashmap $timestamps$.
9: **for** $s \in V_S$ **do**
10: $\quad start\_times \leftarrow \{(c.start\_time, c) | c \in s.out\_neighbors()\}$
11: $\quad end\_times \leftarrow \{(c.end\_time, c) | c \in s.out\_neighbors()\}$
12: $\quad out\_neighbors\_times \leftarrow start\_times \cup end\_times$
13: $\quad$ i $\leftarrow 2$
14: $\quad$ **for** $(t, c) \in sort(out\_neighbors\_time, key = time)$ **do**
15: $\quad\quad$ **if** $t == c.start\_time$ **then**
16: $\quad\quad\quad E_\mathcal{P} \leftarrow E_\mathcal{P} \cup \{(v_{s,i}, v_{c,1})\}$
17: $\quad\quad\quad timestamps.insert(v_{s,i}, t)$
18: $\quad\quad\quad timestamps.insert(v_{c,1}, t)$
19: $\quad\quad$ **else if** $t == c.end\_time()$ **then**
20: $\quad\quad\quad E_\mathcal{P} \leftarrow E_\mathcal{P} \cup \{(v_{c,n\_stages[c]}, v_{s,i})\}$
21: $\quad\quad\quad timestamps.insert(v_{c,n\_stages[c]}, t)$
22: $\quad\quad\quad timestamps.insert(v_{s,i}, t)$
23: $\quad\quad$ **end if**
24: $\quad\quad$ i $\leftarrow i + 1$
25: $\quad$ **end for**
26: **end for**
27: **return** $(V_\mathcal{P}, E_\mathcal{P}), timestamps$

4 SUPERVISED GRAPH REGRESSION WITH GRAPH NEURAL NETWORKS

As mentioned in Sec. 1, we aim to predict the end-to-end latency of microservice traces as a function of the API call and the resource utilization for all microservices at a particular timestamp. That is, let MS be the set of all microservices and $\mathcal{R}_t^{(ms)}$ be the resource utilization of a microservice $ms \in MS$ at any given time t:

$$\mathcal{R}_t := \{\mathcal{R}_t^{(ms)} \in \mathbb{R}^{d_r} \mid ms \in MS\} \tag{1}$$

where d_r is the number of resources. Let $G_\mathcal{A} = (V_\mathcal{A}, E_\mathcal{A})$ be the Span/PERT graph associated with the API $\mathcal{A}$ with $V_\mathcal{A}$ being the set of vertices and $E_\mathcal{A}$ being the set of edges. Let $\mathcal{R}_{t,G_\mathcal{A}}$ be the resource utilization of all microservices at time t restricted to graph $G_\mathcal{A}$, i.e.:

$$\mathcal{R}_{t,G_\mathcal{A}} = \{\mathcal{R}_t^{(ms)} \in \mathbb{R}^{d_r} \mid ms \in V_\mathcal{A}\} \tag{2}$$

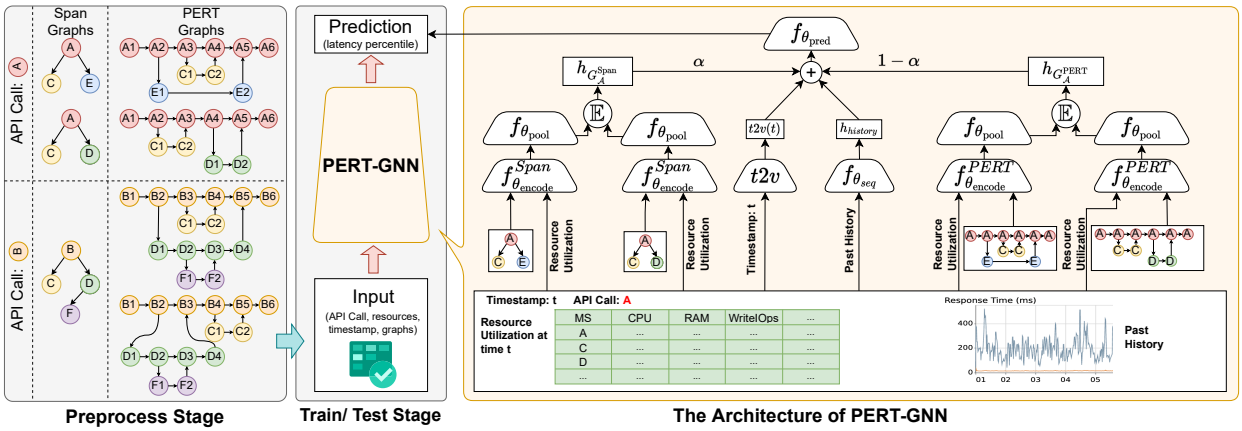


Figure 5: The Design Overview of PERT-GNN

$\mathcal{R}_{t,G_\mathcal{A}}$ is also the set of node features that are related to $\mathcal{A}$. Since we can use different Span/ PERT graphs to represent different API calls to capture microservice operations dependencies [1], our problem can be formulated as a **supervised graph regression problem**:

$$\hat{y}_{\mathcal{A},\mathcal{R}_t} := f_\theta(G_\mathcal{A}, \mathcal{R}_{t,G_\mathcal{A}}) \in \mathbb{R} \tag{3}$$

An overview of our proposed PERT-GNN framework is shown in Fig. 5. In PERT-GNN, f_θ is a composition of a node encoder $f_{\theta_{\text{encode}}}$, a graph pooling operator $f_{\theta_{\text{pool}}}$ and a graph predictor $f_{\theta_{\text{pred}}}$ as follows:

$$\hat{y}_{\mathcal{A},\mathcal{R}_t} = (f_{\theta_{\text{pred}}} \circ f_{\theta_{\text{pool}}} \circ f_{\theta_{\text{encode}}})(G_\mathcal{A}, \mathcal{R}_{t,G_\mathcal{A}}) \tag{4}$$

Given $(G_\mathcal{A}, \mathcal{R}_{t,G_\mathcal{A}})$, we first generate a node embedding $\mathbf{h}_v \in \mathbb{R}^{d_{out}}$ for each node $v \in V$ with an encoder $f_{\theta_{\text{encode}}}$, which captures the structural information of the graph. These node embeddings can be stacked to form a matrix $\mathbf{H}_{G_\mathcal{A}} \in \mathbb{R}^{|V| \times d_{out}}$. The encoder $f_{\theta_{\text{encode}}} : \mathcal{G} \to \mathbb{R}^{N \times d_{out}}$ can be a graph neural networks (GNNs) [21, 27, 50] or a graph transformer [10]. Then, we use a graph pooling operator $f_{\theta_{\text{pool}}} : \mathbb{R}^{N \times d_{out}} \to \mathbb{R}^{d_{out}}$ to aggregate the node embeddings to generate a graph embedding $\mathbf{h}_{G_\mathcal{A}} \in \mathbb{R}^{d_{out}}$. $f_{\theta_{\text{pool}}}$ can be a permutation invariant pooling function such as max pooling, mean pooling, or attention pooling [52, 54]. Finally, we use a graph predictor $f_{\theta_{\text{pred}}} : \mathbb{R}^{d_{out}} \to \mathbb{R}$ to predict the target value $\hat{y}_{\mathcal{A},\mathcal{R}_t}$ from the graph embedding $\mathbf{h}_{G_\mathcal{A}}$. $f_{\theta_{\text{pred}}}$ can be a simple linear function or a multi-layer perceptron (MLP).

In this work, apart from the resource utilization $\mathcal{R}_{t,G_\mathcal{A}}$, we also include the microservice name, the operation name and other structural features such as node depth as the node features (See Sec. 5.1 for more details.). We use structure-aware graph transformer (SAT) [10] as the graph encoder $f_{\theta_{\text{encode}}}$ since it achieves state-of-the-art performance on graph prediction benchmarks. Detailed description of SAT can be found in the Appendix A. We use mean pooling for $f_{\theta_{\text{pool}}}$ and a 2-layer MLP for $f_{\theta_{\text{pred}}}$.

[1] Caveat: We assume here that there is only one Span/ PERT graph for each API call. We will discuss how to handle the general case in Sec. 4.1. In practice, it is very likely that there are multiple Span/ PERT graphs for the same API call due to different runtime behaviors (as mentioned in Sec. 1.)

4.1 Graph Mixtures: Characterizing different runtime behaviors of the same API call

Since we are predicting the latency based on that API call but not the actual trace, and we want to capture the dependencies between microservice operations, we need to construct a graph that is API-specific but not trace-specific. In other words, we need to construct a graph for each API call to capture different API-specific dependencies. However, there could be different runtime dynamics for the same API call. That is, there is no one-to-one correspondence between the entry operation name and the span graph topologies. An intuitive solution is to form a static graph by taking the union of all Span Graphs of the same API call. But this method couples all runtime behaviors and therefore fails to isolate and capture different invocation patterns.

To resolve this challenge, for a given API call $\mathcal{A}$, we interpret $\mathrm{G}_\mathcal{A}$ as a random variable that takes values in the set of Span (or PERT) graphs associated with the API call $\mathcal{A}$. Let $\mathcal{G}_\mathcal{A}$ be the set of all Span/PERT graphs where the operation name of the root node (the API call) is $\mathcal{A}$, we compute the graph representation $\mathbf{h}_{\mathrm{G}_\mathcal{A}} \in \mathbb{R}^{d_{out}}$ by taking the conditional expectation of the graph representation given $\mathcal{A}$ as follows:

$$\mathbf{h}_{\mathrm{G}_\mathcal{A}} := \mathbb{E}_{\mathrm{G}|\mathcal{A}}[f_{\theta_{\text{pool}}} \circ f_{\theta_{\text{encode}}}(\mathrm{G}, \mathcal{R}_{t,G})|\mathcal{A}] \tag{5}$$

$$= \sum_{g \in \mathcal{G}_\mathcal{A}} p(g|\mathcal{A}) f_{\theta_{\text{pool}}} \circ f_{\theta_{\text{encode}}}(g, \mathcal{R}_{t,g}) \tag{6}$$

With $\mathcal{G}_\mathcal{A}^{\text{Span}}$ and $\mathcal{G}_\mathcal{A}^{\text{PERT}}$ being the collection of Span graphs and PERT graphs, we can generate their graph representation $\mathbf{h}_{\mathrm{G}_\mathcal{A}^{\text{Span}}}$ and $\mathbf{h}_{\mathrm{G}_\mathcal{A}^{\text{PERT}}}$ using Eq. 5 respectively, with different parameters. We can compute the final graph representation matrix $\mathbf{h}_{\mathrm{G}_\mathcal{A}}$ by taking the weighted average of the two matrices with a trade-off parameter α as follows:

$$\mathbf{h}_{\mathrm{G}_\mathcal{A}} = \alpha \mathbf{h}_{\mathrm{G}_\mathcal{A}^{\text{Span}}} + (1-\alpha)\mathbf{h}_{\mathrm{G}_\mathcal{A}^{\text{PERT}}}. \tag{7}$$

4.2 Time2Vec

As mentioned in sec. 4, we include the timestamp of the request. For predicting the service response time at a particular time, it should

be useful to know if it is at peak-hours or not. Since peak-hours and off-peak hours are periodic in real-world traffic (e.g. peak-hour in lunchtime and late evening occurs everyday) [29], we use a Time2Vec [36] to capture the periodicity of the traffic. Specifically, given a timestamp $t \in \mathbb{R}$, we use the Time2Vec to generate a vector in $\mathbb{R}^{d_{\text{out}}}$ as follows:

$$\mathbf{t2v}(t)[i] = \begin{cases} \omega_i t + \phi_i & \text{if } i = 0 \\ \sin(\omega_i t + \phi_i) & \text{if } 1 \leq i \leq d_{\text{out}} - 1 \end{cases} \tag{8}$$

where ω_is and ϕ_is are trainable parameters.

In our experiments, the granularity of t is an hour. Our ablation studies in Sec. 5.5.3 show that the Time2Vec is effective for predicting the service response time. We also find that in most runs, at least one of the learned ω_i takes the value of $0.26 \approx \frac{2\pi}{24}$, indicating that the traffic repeats every 24 hours, which captures the daily periodic patterns.

4.3 Processing Graph features for Predictions

Apart from the timestamp, we also include a time series of the past service response as the graph-level features to the model. To capture the trend of this time series, we apply a sequence model $f_{\theta_{\text{seq}}}$ to the time series, to obtain a vector representation $\mathbf{h}_{\text{history}} \in \mathbb{R}^{d_{out}}$ of the time series, as shown in Fig. 5. $f_{\theta_{\text{seq}}}$ can be any differentiable sequence model such as LSTM [22] or Transformer [49]. Finally, we combine the **t2v** vector, the graph embeddings $\mathbf{h}_{G_{\mathcal{A}}}$, and the time-series embedding $\mathbf{h}_{\text{history}}$ to generate the latency prediction as follows:

$$\hat{y}_{\mathcal{A},\mathcal{R}_t} = f_{\theta_{\text{pred}}}(\mathbf{h}_{G_{\mathcal{A}}} + \mathbf{t2v}(t) + \mathbf{h}_{\text{history}}) \in \mathbb{R}. \tag{9}$$

4.4 Quantile Regression Loss

Standard regression model minimizes the mean squared error (MSE) between the predicted and the ground truth. The method of least squares estimates the conditional mean of the delay given the entry microservice and the resource utilization. However, end-to-end delay is commonly modeled in terms of quantiles, such as the 95th percentile. The tail behavior is usually more important because some user studies have found out that people favor a slightly slower system to one with unstable response time [8].

In light of this, rather than giving a point estimate of the end-to-end delay (e.g. the conditional mean), we propose to use quantile regression [28] to predict the conditional quantiles of the end-to-end delay given the API call and the resource utilization for all microservices at a particular time. Let $\hat{y}$ be the predicted end-to-end delay and y be the corresponding ground truth (We omit the subscript $\mathcal{A}$ and $\mathcal{R}_t$ for simplicity.), the quantile regression loss [28] for τ-th quantile is defined as:

$$\mathcal{L}_\tau(\hat{y}, y) = \tau \max(y - \hat{y}, 0) + (1 - \tau) \max(\hat{y} - y, 0) \tag{10}$$

$$= \begin{cases} \tau(y - \hat{y}) & \text{if } y > \hat{y} \\ (1 - \tau)(\hat{y} - y) & \text{if } y \leq \hat{y} \end{cases} \tag{11}$$

Note that when $\tau = 0.5$, the quantile regression loss is equivalent to half of the Mean Absolute Error (MAE) loss. It can be shown that the minimizer of the quantile regression loss is the conditional τ-th quantile of the end-to-end delay given the entry microservice and the resource utilization [28]. The quantile regression loss is a convex function of the predicted delay. Since the quantile loss is not differentiable when $\hat{y} = y$, we use subgradient descent to optimize the quantile regression loss.

5 EXPERIMENT

In this section, we present the experimental results to demonstrate the effectiveness of PERT-GNN. We first describe the experimental setup, including the datasets, the evaluation metrics, and the baselines. Then, we describe the results of our proposed method and compare it with the baselines. We further conduct ablation studies to analyze the effectiveness of the proposed components.

5.1 Data Processing

We evaluate our proposed method on a Social Network microservice application using simulated workloads from DeathStarBench [18] [2] and real-world traces from Alibaba [34] [3]. The experiments are conducted on a server with Intel Core AMD EPYC 7532 32-Core Processor CPU ×2 with 1 TB of memory and GeForce RTX 3090 SUPER GPU x 8 running Ubuntu 20.04.3 LTS.

5.1.1 DeathStarBench Social Network application. The Social Network application consists of 29 components and 76 resources. We preprocess the dataset according to [11] by generate the workloads with Locust [3]. The loading pattern follows real-world traffic pattern [29] with two peaks per day, and we simulate 7 days of workloads for training and evaluation. The social network graph and post contents are based on real-world datasets from Facebook and photos are drawn from the INRIA dataset [15] to resemble the realistic interactions between users [43] All microservices are deployed in separate Docker containers [37] orchestrated by Kubernetes [2]. Span data is collected with Jaeger [1].

For the node features, we include the microservice name, the operation name and the most recent CPU and memory utilization, write IOps and write throughput for all components. We use Prometheus [6] to collect these utilization metrics, with a scrape interval of 5 seconds. For Span Graph, we also include the node depth (i.e., the number of hops from the entry microservice) as node features. For PERT Graph, we extract both the minimum and maximum node depth (i.e. the shortest and the longest path length between the root and the nodes) as node features.

For graph features, we include API-call id, and create a multivariate time series of past number of requests and the response time of the entry microservice. We divide the history into k time windows of u seconds each, where k and u are hyperparameters. For each time window, we extract the timestamp and compute the count, mean, min, max, and standard deviation of the response time.

5.1.2 Alibaba Microservice applications. Alibaba's real-world traces (collected in 2021) [34] were obtained from over ten thousand bare-metal nodes on production clusters for 12 hours, with a 0.5% sampling rate. The dataset includes 20 million traces (user requests), spanning over 20 thousands microservices within 10 clusters.

Node features include the microservice name, the communication paradigms (e.g. HTTP, Remote procedure call, Database,

[2]The application is available at https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork

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

Table 3: Test set quantile regression results. The best-performing model is highlighted with boldface.

		Social Network			Alibaba		
		$\tau = 0.5$		$\tau = 0.95$	$\tau = 0.5$		$\tau = 0.95$
Model	Input Graph	MAPE	MAE (ms)	Quantile Loss	MAPE	MAE (ms)	Quantile Loss
Linear	-	18.27%	2.547	0.6564	44.38%	2.046	6.2496
Random Forest	-	18.85%	2.596	0.6731	42.40%	3.866	1.9174
Decision Tree	-	18.65%	2.577	0.6693	70.15%	6.054	2.7274
Gradient Boosting	-	18.25%	2.546	0.6560	12.57%	1.407	0.6234
MLP	-	18.92%	2.521	0.7038	16.76%	1.755	2.5460
GRAF	MCG	21.3% [a]	-	-	OOM	OOM	OOM
PERT-GNN ($\alpha = 1$)	Span Mixture	12.84%	1.683	0.2310	12.04%	1.391	0.5865
PERT-GNN ($\alpha = 0$)	PERT Mixture	**11.81%**	**1.658**	**0.2184**	**11.47%**	**1.275**	**0.5781**

[a] We report the lowest MAPE among different latency regions shown in [39].

Memcached), the interface name, the node depth (see Sec. 5.1.1), and the CPU and memory utilization of the microservices. Since only 1300+ nodes are selected for recording the CPU/ RAM utilization per 30 seconds, node features are missing for most of the microservices. We selected 1.75 million traces (9.37%) for training and testing, each with over 60% of microservices having node features (i.e. CPU/RAM utilization). To address missing features, we added a missing value indicator. Out of 1.75 million traces, there are 65 different API calls. As mentioned in Sec. 4.1, a single API call can have numerous runtime behaviors. On average, there are 722 different runtime behaviors per API call with some API calls having up to 6061 different runtime behaviors.

API call id is the only graph features in this dataset. The multivariate time series of past number of requests and the response time are not available in the dataset. Refer to [34] for more details.

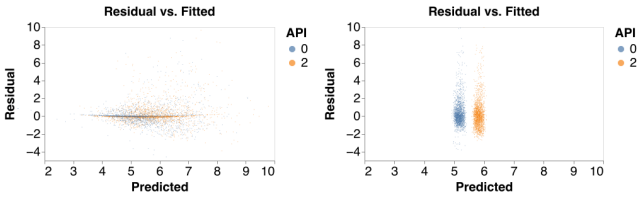
5.2 Hyperparameter Settings

We adapt the code from Chen et al. [10] to implement SAT model for our applications. [4] We use the Random Walk Embedding as the Absolute embedding [46]. The number of transformer layers of the SAT is selected from 1-4. The embedding dimension is selected from 16-256. The dropout rate is selected from $\{0, 0.2, 0.4, 0.6, 0.8\}$. The number of layers are selected from 1-3. We use the mean pooling as the pooling operator $f_{\theta_{\text{pool}}}$. We train PERT-GNN with 50 epochs and a batch size of 256. We use Adam [26] as the optimizer for the model and the learning rate is selected from the range of $[0.0002, 0.005]$.

5.3 Baselines

We compare PERT-GNN with 6 baselines: Linear Regression (LR), Random Forest (RF), Decision Tree (DT), Gradient Boosting Decision Tree (GBDT), Multi-Layer Perceptron (MLP), GRAF [39]. For LR, RF, DT, GBDT, and MLP, the input features are the concatenation of the features of all microservices (i.e. the most recent utilization) and the graph features. For RF and DT, we use Scikit-garden implementation [7] with default hyperparameters. For LR and GBDT, we use Scikit-learn implementation [41] with default hyperparameters. For MLP, we implement it using PyTorch [40]

[4] Implementation code is available at https://github.com/handasontam/PERT-GNN-KDD23.git.



(a) PERT-GNN ($\tau = 0.5$) **(b) Linear ($\tau = 0.5$)**

Figure 6: Residual vs. fitted for two selected API-call in the Social Network application. The linear regressor has a larger error and fails to identify different runtime behaviors of the same API call. Best viewed in color.

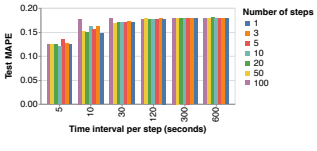


Figure 7: Performance of PERT-GNN across different time interval and number of steps. Best viewed in color.

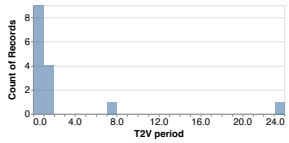


Figure 8: Histogram of the period learned by Time2Vec.

where the number of layers is selected from $2 - 3$, the number of hidden units is selected from $16 - 256$, and the learning rate is also selected from $0.0002 - 0.005$. DeepRest's [11] results cannot be directly comparable to ours since DeepRest focuses on the time-series resource estimation problem. Additionally, DeepRest builds a graph by only merging the Span graphs together, therefore we expect that if we apply its prediction model to estimate latency, the performance will be worse than PERT-GNN with $\alpha > 0$.

5.4 End-to-End Results

Table 3 shows the results of the experiments where we compare the performance of PERT-GNN with other machine learning models. We trained the models with $\tau = 0.5$ and $\tau = 0.95$ and report the test results with the best validation performance. PERT-GNN outperforms the other models in terms of MAPE and MAE for median estimate, as shown in Table 3. In the Social Network application,

PERT-GNN with $\alpha = 0$ outperforms all other models, indicating PERT Graph is more effective than Span Graph in capturing temporal dependencies. Alibaba trace dataset has more microservices and API calls than the Social Network application (20000+ vs. 29 and 65 vs. 3, respectively). All baseline models fail to provide accurate predictions in the Alibaba traces dataset. This is because they are not API-aware and cannot identify the few relevant microservices from a large set of microservices for each API call. In the worst case, if we include all relations between microservices (MCG), training becomes infeasible due to out-of-memory (OOM). Thus, an API-aware model is necessary to handle large-scale traces. Similarly, PERT-GNN with PERT mixture (i.e., $\alpha = 0$) also outperforms all other models in the Alibaba traces dataset. We display residual plots for two API calls in the Social Network Application's median estimate in Fig. 6a and Fig. 6b for PERT-GNN and Linear Regressor respectively. Compared to PERT-GNN, the linear regressor predicts the latency of the trace with a large error and gives almost the same prediction for the same API, indicating that it cannot capture different runtime behaviors of the same API.

Unlike median estimate, it is difficult to evaluate the performance of 95th percentile estimate. This is because we cannot observe the true 95th percentile value of any trace, i.e. there is no ground truth to compare with. Thus, we cannot use the MAPE or MAE to evaluate the performance of 95th percentile estimate. Since the minimizer of the quantile loss with $\tau = 0.95$ is the 95th conditional percentile, the model with the lowest quantile loss is the best model for 95th percentile estimate. Quantile loss is hard to interpret but we can still use it to compare the performance of different models. Table 3 shows that PERT-GNN outperforms the baselines in terms of quantile loss. Similar to the median estimate, PERT-GNN with $\alpha = 0$ outperforms the other models, which indicates that the PERT Graph incorporates important information for capturing tail behaviors.

5.5 Ablation Studies

In this section, we conduct ablation studies to analyze the effectiveness of the proposed components. We seek to answer the following questions:

(1) How does the use of the PERT graph affect the performance of the model?
(2) How does the number of steps and the time interval for each step for the historic affect the performance of the proposed method?
(3) Can **t2v** generate a good embedding that captures the trend or the periodicity of the time series?

5.5.1 Influence of using PERT Graph. To evaluate the impact of using PERT graphs, we compared PERT-GNN with PERT mixture to PERT-GNN with Span mixture while holding other hyperparameters fixed. As shown in Table 3, PERT-GNN with PERT mixture ($\alpha = 0$) outperforms PERT-GNN with Span mixture ($\alpha = 1$) in all metrics. We also attempted to use a weighted sum of the two graphs by selecting $\alpha \in (0, 1)$, but the performance was typically similar to PERT-GNN with $\alpha = 0$. This supports our intuition that the PERT graph provides additional temporal ordering information that is not captured by the Span graph, and adding the Span graph does not improve the performance since all the information in the Span graph is already captured by the PERT graph.

5.5.2 Influence of the number of time steps and time interval. To evaluate the influence of the number of time steps and the time interval for the time series of the past history of the number of requests and response time, we conduct experiments with different combinations of the number of time steps (ranging from 1 to 100) and the time interval (ranging from 5 to 600 seconds) while holding other hyperparameters fixed. Figure. 7 shows the performance of the proposed method with different combinations of the number of time steps and the time interval. As shown in the figure, smaller time interval leads to better performance. This is expected because the short-term trend of the time series gives more information about whether there is traffic congestion at the moment. On the other hand, we observe that the performance of the proposed method is not sensitive to the number of time steps across different time intervals. Surprisingly, setting the number of steps to 1 does not hurt the performance of the proposed method, and for the time interval of 10s and 30s, the model is significantly degraded when the number of steps is set to 100. This is because large step sizes introduce a lot of noise to the time series, which makes it difficult for the model to learn the short-term trends of the time series. For time intervals larger than 30s, the performance of the proposed method saturates, which again indicates that long-term trends are not as important as short-term trends for the prediction of latency.

5.5.3 t2v. Figure. 8 shows the histogram of the period learned by **t2v** for the different runs. Looking at the learned frequency for the sine functions across several runs, we observed that in many runs, we find that in most runs, one of the learned ω_i takes the value of $0.26 \approx \frac{2\pi}{24}$, indicating that **t2v** learns a periodic traffic pattern that repeats every 24 hours.

6 CONCLUSION AND FUTURE WORKS

We present PERT-GNN, a graph neural network model to predict the end-to-end latency for microservice applications. With the use of PERT Graph, the temporal orderings of microservice operations can be preserved and the run-time dynamics within or across different API-call can be decoupled and incorporated into the graph learning model. Experiments and ablation studies suggest that the proposed model is capable of producing accurate predictions while being computationally efficient. Future directions include designing a better graph neural networks model that exploits the DAG properties of a PERT Graph, and building a proactive resource allocation system based on PERT-GNN.

ACKNOWLEDGEMENTS

We sincerely thank the anonymous KDD'23 reviewers for their insightful suggestions. This work is supported in part by National Natural Science Foundation of China under Grant No.62202284b, the Shanghai Pujiang Program (22PJ1404000), the Innovation and Technology Commission (ITS/244/16), the CUHK MobiTeC R&D Fund, the Science and Technology Development Fund of Macau (0024/2022/A1), the Multi-Year Research Grant of University of Macau (MYRG2022-00119-FST), the Start-up Research Grant of University of Macau (SRG2021-00004-FST).

REFERENCES

[1] 2022. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/

[2] 2022. Kubernetes, an open-source system for automating deployment, scaling, and management of containerized applications. https://kubernetes.io/

[3] 2022. Locust: An open source load testing tool. https://locust.io/. Accessed: 2022-09-27.

[4] 2022. Microservices - The Airbnb Tech Blog. https://medium.com/airbnb-engineering/tagged/microservices

[5] 2022. Microservices - The Netflix Tech Blog. https://netflixtechblog.com/tagged/microservices

[6] 2022. Prometheus, a systems and service monitoring system. https://prometheus.io/

[7] 2023. Scikit-garden. https://scikit-garden.github.io/.

[8] Betsy Beyer, Niall Richard Murphy, David K Rensin, Kent Kawahara, and Stephen Thorne. 2018. *The site reliability workbook: Practical ways to implement SRE.* " O'Reilly Media, Inc.".

[9] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*. 153–167.

[10] Dexiong Chen, Leslie O'Bray, and Karsten Borgwardt. 2022. Structure-aware transformer for graph representation learning. In *International Conference on Machine Learning*. PMLR, 3469–3489.

[11] Ka-Ho Chow, Umesh Deshpande, Sangeetha Seshadri, and Ling Liu. 2022. DeepRest: deep resource estimation for interactive microservices. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 181–198.

[12] Ka-Ho Chow, Umesh Deshpande, Sangeetha Seshadri, and Ling Liu. 2022. DeepRest: deep resource estimation for interactive microservices. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 181–198.

[13] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 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.

[14] Ira Cohen, Jeffrey S Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. 2004. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control.. In *OSDI*, Vol. 4. 16–16.

[15] Navneet Dalal and Bill Triggs. 2005. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, Vol. 1. Ieee, 886–893.

[16] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering* (2017), 195–216.

[17] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 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.

[18] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[19] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 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.

[20] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1994–2004.

[21] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NIPS*. 1024–1034.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[23] Lexiang Huang and Timothy Zhu. 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.

[24] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 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.

[25] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, and Jason Mars. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of EuroSys*.

[26] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[27] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[28] Roger Koenker and Kevin F Hallock. 2001. Quantile regression. *Journal of economic perspectives* 15, 4 (2001), 143–156.

[29] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.

[30] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*. 312–324.

[31] Yang Liu, Huanle Xu, and Wing Cheong Lau. 2020. Accordia: Adaptive Cloud Configuration Optimization for Recurring Data-Intensive Applications. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 831–841.

[32] Yang Liu, Huanle Xu, and Wing Cheong Lau. 2022. Online Resource Optimization for Elastic Stream Processing with Regret Guarantee. In *Proceedings of the 51st International Conference on Parallel Processing (ICPP '22)*.

[33] Shutian Luo, Xu HL, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 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*.

[34] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*. 412–426.

[35] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 2022. The Power of Prediction: Microservice Auto Scaling via Workload Learning. In *Proceedings of the ACM Symposium on Cloud Computing*.

[36] Seyed Mehran Kazemi, Rishab Goel, Sepehr Eghbali, Janahan Ramanan, Jaspreet Sahota, Sanjay Thakur, Stella Wu, Cathal Smyth, Pascal Poupart, and Marcus Brubaker. 2019. Time2Vec: Learning a Vector Representation of Time. *arXiv preprint arXiv:1907.05321* (2019).

[37] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 239, 2 (2014), 2.

[38] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: a graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 154–167.

[39] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: a graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 154–167.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *NIPS*. 8024–8035.

[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[42] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. {FIRM}: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 805–825.

[43] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Twenty-ninth AAAI conference on artificial intelligence*.

[44] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[45] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[46] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-Attention with Relative Position Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, Marilyn A. Walker, Heng Ji, and Amanda Stent (Eds.). Association for Computational Linguistics, 464–468. https://doi.org/10.18653/v1/n18-2074

[47] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. 513–526.

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

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NIPS*. 5998–6008.
[50] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).
[51] Lingmei Weng, Peng Huang, Jason Nieh, and Junfeng Yang. 2021. Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 193–207.
[52] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *ICLR*.
[53] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. 2019. MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*. IEEE, 122–132.
[54] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. 2018. Hierarchical graph representation learning with differentiable pooling. In *NIPS*. 4800–4810.
[55] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 167–181.
[56] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of EuroSys*.

A STRUCTURE-AWARE GRAPH TRANSFORMER

In this paper, we use structure-aware graph transformer (SAT) [10] as the graph encoder $f_{\theta_{\text{encode}}}$ since it achieves state-of-the-art performance on graph prediction benchmarks, and it is able to capture the structural information of the graph, which is important in our application. We will briefly introduce SAT in this section. SAT first uses a k-subgraph GNN extractor to extract local structure of a node. Let $\text{GNN}^{(k)}(v, G)$ be an arbitrary GNN with k layers applied to node v in graph G. Let $\mathbf{h}'_v \in \mathbb{R}^{d_{out}}$ be the embeddings of node v computed by $\text{GNN}^{(k)}_\theta$, that is:

$$\mathbf{h}'_v := \text{GNN}^{(k)}_\theta(v, G) \in \mathbb{R}^{d_{out}}. \tag{12}$$

Let $\mathbf{h}^{\text{struc}}_v$ be the local structural embedding of node v, which is computed by summing the embedding of all nodes within the k-hop neighborhood of v:

$$\mathbf{h}^{\text{struc}}_v := \sum_{u \in \mathcal{N}_k(v)} \mathbf{h}'_u \in \mathbb{R}^{d_{out}}. \tag{13}$$

SAT then define an asymmetric exponential kernel function κ to compute the structural similarity between two nodes v and u in graph G via:

$$\kappa(u, v) := \exp\left(\langle \mathbf{W_Q}\mathbf{h}^{\text{struc}}_u, \mathbf{W_K}\mathbf{h}^{\text{struc}}_v \rangle / \sqrt{d_{out}}\right) \in \mathbb{R}, \tag{14}$$

where $\langle \cdot, \cdot \rangle$ is the dot product, $\mathbf{W_Q}, \mathbf{W_K} \in \mathbb{R}^{d_{out} \times d_{out}}$ are trainable parameters, and d_{out} is the dimension of the structural embedding.

Having defined the structural similarity between two nodes, SAT then computes the self attention [10, 49] of node v via:

$$\text{SA-attn}(v) := \sum_{u \in V} \frac{\kappa(v, u)}{\sum_{w \in V} \kappa(v, w)} (\mathbf{W_V} x_u) \in \mathbb{R}^{d_{out}}, \tag{15}$$

Finally, we apply a skip-connection and a multi-layer perceptron to the output of the self-attention to obtain the final embedding of node v:

$$\mathbf{h}_v = \text{ReLU}((\mathbf{x}_v + \text{SA-attn}(v))W_1)W_2 \in \mathbb{R}^{d_{out}}. \tag{16}$$

In practice, we use multi-head attention [49], where we apply multiple Eq. 15 with different $\mathbf{W_Q}, \mathbf{W_K}, \mathbf{W_V}$, and θ, and concatenate the results, followed by a projection. This allows the model to attend to different structural information. Equation (16) is a layer of structure-aware graph transformer (SAT) [10]. One can stack multiple layers of SAT to obtain a deeper SAT.