Large Language Models are Realistic Microservice Trace Generators

Donghyun Kim
UT Austin
donghyun@utexas.edu

Sriram Ravula
UT Austin
sriram.ravula@utexas.edu

Taemin Ha
UT Austin
taemin.ha@utexas.edu

Alexandros G. Dimakis
UT Austin
dimakis@austin.utexas.edu

Daehyeok Kim
UT Austin
daehyeok@utexas.edu

Aditya Akella
UT Austin
akella@cs.utexas.edu

Abstract

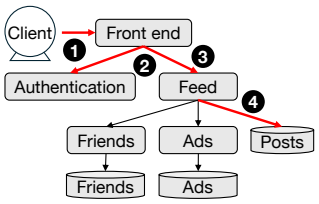
Computer system traces, defined as records of hardware or software events during system operations, are essential for understanding the behavior of complex systems and managing resources. However, obtaining real-world traces can be challenging due to the significant collection overheads in performance and privacy concerns that arise in proprietary systems. As a result, synthetic trace generation is considered a promising alternative to real-world traces. This paper proposes to train a large language model (LLM) to generate synthetic computer system traces, specifically microservice call graphs. To capture hierarchical structures and implicit constraints in traces, we fine-tune LLMs to generate each layer recursively, making call graph generation a sequence of easier steps. To allow the simulation of uncommon situations and further enforce learning constraints in traces, we apply additional instruction tuning steps to align our model with the desired trace features. Our evaluation results show that our model can generate diverse traces under various conditions and outperform existing methods in accuracy and validity. We also demonstrate the utility of our model with two downstream tasks that predict trace features and infill missing data for given partial traces.

1 Introduction

Computer system traces, which document hardware or software events during operations, are vital for analyzing complex systems and optimizing resource management. However, obtaining real-world traces is often hindered by privacy concerns and their general unavailability. As an alternative, synthetic traces provide limitless size and variety, offering significant advantages for testing and analysis, including the ability to simulate challenging conditions like stress-testing environments. While recent advances in generative machine learning, including LSTMs [35], GANs [7], and diffusion models [9, 38], have facilitated the creation of realistic synthetic traces, these methods typically generate only specific fields, such as request numbers or resource types [2], or are restricted to certain lengths, like network packets [14, 48].

We argue for the use of Large Language Models (LLMs), transformer-based [41] neural networks pre-trained autoregressively on large and diverse text datasets [4, 39], to generate synthetic traces. It has been shown that LLMs can be readily adapted to model a variety of domains besides natural language, such as protein sequences [34], code [32], and tabular data [3]. LLMs can produce outputs

Preprint. Under Review.



	EDGE ID	Src.	Dest.	Type	Start (ms)	Finish (ms)
1	0	Client	Front end	HTTP	0	24
2	0.1	Front end	Authentication	RPC	0	1
3	0.2	Front end	Feed	RPC	1	23
4	0.2.1	Feed	Posts	DB	6	15

Figure 1: A simple social network application consists of eight microservices [12]. Each user request triggers a sequence of microservice calls, forming a microservice call graph. The red lines represent the microservice call graph for a user request. Microservice call graphs are commonly logged in a tabular format, as shown in the figure on the right. Each row in the table represents a communication between two microservices, with the details of the communication logged as features in the columns.

that are well-aligned with user inputs in several flexible ways such as fine-tuning model weights [28, 43], and can generalize to new user inputs at inference [5, 33]. Thus, LLMs have the potential to generate synthetic traces that accurately model the structure of traces while following user prompts.

Despite their potential, using LLMs for synthetic trace generation presents significant challenges. Traces are often logged in a tabular format and follow an underlying data structure such as a graph, meaning it is non-trivial to represent valid traces as text sequences, which is the format best suitable for modern autoregressive LLMs. Moreover, there are often implicit constraints in trace data that rely on relationships between multiple trace features. For example, a software application's start time must be earlier than the start time of all the child processes it spawns; similarly, the parent application's end time must be later than the end time of its children.

In this paper, we show how general-purpose LLMs can be adapted to generate synthetic system traces. We focus specifically on *microservice call graphs*, a special class of trace with a directed acyclic graph (DAG) structure. One of our key innovations is to generate complete call graphs by recursively generating subgraphs, or *layers*. This approach allows the model to break down the complex task of reasoning about hierarchical graph structures and constraints into multiple easier tasks. We pre-train our model with the next token prediction on call graphs, then perform supervised fine-tuning to align the model output with user-requested attributes. During fine-tuning, we train the model to explicitly generate a series of intermediate reasoning steps. These steps appear between recursive layer generation calls and make the model perform simple arithmetic and logical calculations to check its own progress, improving the model's ability to adhere to structural and user-defined constraints.

We demonstrate the effectiveness of our approach by fine-tuning Llama-2 7B [39] with our method on microservice trace data and performing a series of evaluations. Our results demonstrate that the proposed recursive generation and intermediate reasoning steps improve the LLM's ability to produce valid outputs for both complex (i.e., deep and wide) and simple call graph structures. When compared to traces from a learned generative model baseline and a probabilistic model handcrafted by an expert, synthetic traces produced by our model more closely match the distribution of real traces. We further show that our fine-tuned model performs well when used for real-world downstream tasks.

We summarize our key contributions below:

- We introduce a novel method for using LLMs to create valid synthetic microservice traces. To ensure that the complex structural constraints of valid call graphs are respected, we *recursively generate* layers of subgraphs along with instructions for subsequent layers. We also train the LLM to describe the generated layers with *intermediate instructions*.
- We show that recursive generation and intermediate instructions improve the validity of synthetic traces. Also, synthetic traces generated by our model are more realistic regarding distribution similarities than those from a baseline generative model and a handcrafted expert model.
- Our model can generalize to unseen combinations of user-requested attributes at inference. Also, our model can be further trained to perform key downstream tasks such as infilling missing data.

2 Background

Microservice Call Graphs. In modern software architecture, an application is typically constructed as a constellation of multiple microservices [6, 25, 12], each with specific functionalities and dependencies on one another. When users interact with these applications, for instance, by sending

HTTP requests to web servers, a complex sequence of communications among these microservices is triggered. Thus, a user request induces a microservice *call graph*, which maps the control/data flow and dependencies among the microservices involved in fulfilling the user's request.

Figure 1 is an example of a social network application deployed with several microservices (8 in total). In the figure, the red arrows indicate communications between microservices involved in processing the user's request; these form a microservice call graph with four microservices. The vertices of the graph correspond to microservices (or the client), while the edges correspond to API calls invoking the microservices. Each edge originates at the requesting microservice and terminates at the target microservice. Note that some edges are not part of the call graph as the corresponding microservices are not invoked in processing this particular request.

Each call graph can be represented as a tabular log trace with a textual description of the features of each API call (i.e., edges), including the source and destination of the request, type of request (e.g., HTTP and RPC), and start/finish time. As call graphs have a *hierarchical structure*, microservices should appear in a specific order to maintain the parent-child relationships. Moreover, the start and end times of each call should be consistent with each other: (1) the start time of a microservice should be smaller than the finish time, and (2) the parent-child relationships should be honored, i.e., the parent's start time should be smaller than child's, and parent's finish time should be larger than the child's. The IDs within a call graph (dot-decimal numbers provided for each call) should also be hierarchically connected to form a DAG structure.

Synthetic Trace Generation using Machine Learning. Microservice traces encapsulate the interaction dynamics of all services running on a given cluster of machines over a period, so the analysis of the traces plays a pivotal role in improving the performance and reliability of services. Representative use cases include critical path analysis [50], anomaly detection [45], root cause analysis [13], and cluster management optimization [31, 26]. Unfortunately, access to such traces remains challenging due to business and privacy concerns.

Given the importance and limited availability of public computer system traces, including microservice traces, several recent studies have explored generative models for synthetic trace generation. [20, 48, 14] leverage GAN [7] and diffusion [9, 38] models to generate network packet traces, while [2] uses LSTMs [35] to generate virtual machine workloads. Even though the generative models have shown effectiveness in each domain, the methods can be used only for predicting specific fields or generating fixed-length traces. These methods do not apply to generating microservice call graphs because they cannot handle the variable lengths and hierarchical structures of the call graphs.

3 Problem Formulation

Our goal is to train a generative model to generate synthetic microservice traces, specifically call graphs. We want to allow end-users to simulate various scenarios by conditioning the model's output on user-requested attributes. Given the limitations of existing trace generation approaches, we turn to LLMs, which are transformer-based [41] models with billions of parameters. We initialize our model from a general-purpose LLM pre-trained on a large and diverse text dataset, as these models have shown effectiveness when adapted for specialized domains such as proteins [34], code [32], and tabular data [3]. LLMs accept variable-length inputs and can be conditioned in a variety of arbitrary manners, including natural language prompting [28] and structured input sequences [3].

Given a dataset of call graphs $\{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_N\}$, we want to learn the distribution $p(\mathbf{X})$ of the nodes and edges within call graphs. Pre-trained LLMs expect sequences of text as input, which the model's tokenizer $T(\cdot)$ decomposes as a sequence $[s_1, s_2, \ldots, s_\ell]$, where s_j is a discrete token from a fixed vocabulary $\mathcal{V}$ and sequence length ℓ can vary between inputs. Therefore, we need to encode each call graph $\mathbf{X}$ into a text-based representation $\mathbf{t}$. Our objective is to learn $p(\mathbf{t})$ with our model, which is equivalent to learning $p(\mathbf{X})$ given that we can recover a call graph from its text encoding.

Throughout this paper, we treat autoregressive language models, which factorize $p(\mathbf{t})$ as the product of conditional probabilities over token sequence $T(\mathbf{t}) = [s_1, s_2, \ldots, s_\ell]$:

$$p(\mathbf{t}) = \prod_{k=1}^{\ell} p(s_k | s_1, s_2, \ldots, s_{k-1}). \tag{1}$$

The model is trained to predict the next token given the sequence of preceding tokens. To generate call graphs based on user input, we condition the model distribution:

$$p(\mathbf{t}|\mathbf{c}) = \prod_{k=1}^{\ell} p(s_k|s_1, s_2, \ldots, s_{k-1}, s_1^c, s_2^c, \ldots, s_p^c). \tag{2}$$

Here, $\mathbf{c}$ is an additional text-based prompt from the user, tokenized as $T(\mathbf{c}) = [s_1^c, s_2^c, \ldots, s_p^c]$ where s_j^c is a token from the same fixed vocabulary $\mathcal{V}$.

4 Method

This section presents our approach for training LLMs to generate microservice call graphs. First, we describe how we encode call graphs, stored as tabular data, into a text format that can be tokenized and processed by the LLM. Then, we detail a novel approach to improve the model's generation of complex call graph structures. We decompose the generation task into multiple subgraph generation tasks and train the model to recursively generate subgraph instructions to condition its own output. Finally, we give an overview of our two-stage training process, which consists of pre-training to learn the data distribution, followed by instruction fine-tuning to improve user-controlled generation.

4.1 Encoding Call Graphs as Text

Before training our language model, call graphs must be encoded into text-based representations. As detailed in §2 and shown in Figure 1, microservice call graphs are initially in a tabular format, with rows representing communications between microservices and columns detailing features for each edge. Our encoding process is meticulously designed to preserve the integrity of the call graph's structure and the specific constraints dictated by edge features, following the method described by [3]. This method involves representing features in a natural language format, which leverages the LLM's pre-training on diverse datasets and requires minimal preprocessing. Additionally, we encode attributes of the overall call graph to serve as conditioning information for the model.

Edge Features. Tabular call graph $\mathbf{X}$ has m columns of features with textual names $\{f_1, f_2, \ldots, f_m\}$ and n rows of edges $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$. We denote the value of feature $j \in \{1, \ldots, m\}$ for edge $i \in \{1, \ldots, n\}$ as v_{ij}. We encode each edge $\mathbf{x}_i$ as a text sequence $\mathbf{t}_i = [t_{i1}, t_{i2}, \ldots, t_{im}]$, where t_{ij} is a description of the j-th feature with the format $t_{ij} = [\phi(f_j), v_{ij}]$. Here, $\phi(f)$ encodes feature name f into a text template with a subject-predicate structure to provide a natural language description of feature value v_{ij}. For instance, the encoding for edge 1 in Figure 1 would be `[Edge ID is 0, Source is Client, Destination is Front end, Type is HTTP, Communication starts at 0 ms, Communication finishes at 24 ms]`. We encode tabular call graph $\mathbf{X}$ to the equivalent text-based representation $\mathbf{t} = [\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n]$, formed as a sequence of the text-encoded edges $\mathbf{t}_i$. We note that the structure and constraints of the call graph only depend on the feature values and are invariant to the specific feature order. Therefore, during training, we randomly shuffle the order of the features within each edge as in [3] to remove any spurious associations that arise from position information.

Call Graph Attributes. Apart from individual edges, the overall call graph can also be described by attributes, including the maximum depth, the total number of edges, and the total communication latency. These attributes are useful for summarizing the complex interactions between edges, and can be fed to the model as a prompt (e.g., by an end-user) to condition call graph generation. Let call graph $\mathbf{X}$ have r attributes with names $\{a_1, a_2, \ldots, a_r\}$ and corresponding values $\{w_1, w_2, \ldots, w_r\}$. We encode the attributes as a text sequence $\mathbf{c} = [c_1, c_2, \ldots, c_r]$, where c_j is a description of the j-th attribute with the format $c_j = [a_j, \text{“ : ”}, w_j]$. See the *Conditions* shown in red in Figure 2 for a simplified example of text-encoded call graph attributes. Similar to the edge features, we randomly shuffle graph attributes during training. We additionally drop each attribute independently with probability p_{drop} to allow flexible prompting with arbitrary subsets of attributes.

4.2 Recursive Generation

To handle complex structures, we propose to break down the task of generating a call graph based on a prompt into a series of recursive *layer generation* tasks. Starting from the initial prompt $\mathbf{c}$, the

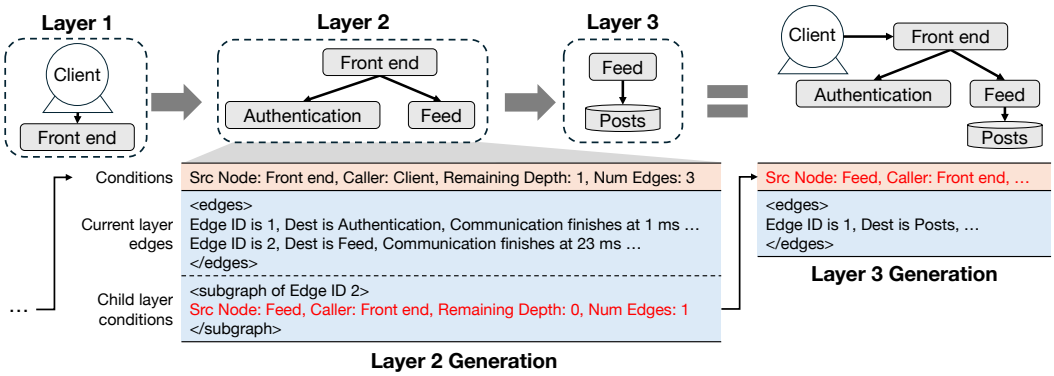


Figure 2: Overview of the recursive generation method with a simplified example. *Layer 2* uses conditions (e.g., source node, caller, the number of edges) generated by *Layer 1*. Using the conditions, the model generates two edges in *Layer 2* with destinations to *Authentication* and *Feed* respectively, and next layer conditions starting from the *Feed* microservice. The recursion continues until all the edges in *Layer 3* are generated.

task for each layer is to generate the edges originating from the *Start Node* specified in the prompt. We also generate a new prompt for the next layer based on the previous layer prompt and the edges generated in the current layer. This new prompt is then re-used to condition the model's output for the next layer. The recursive process continues until the requested attributes $\mathbf{c}$ are satisfied.

Formally, for an encoded call graph $\mathbf{t} = [\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n]$, we partition the edges $\mathbf{t}_i$ into a sequence of layers $[\mathbf{t}^1, \mathbf{t}^2, \ldots, \mathbf{t}^l]$, where $l \leq n$. Each layer is comprised of a sequence of edges that share the same parent (i.e., source) node, and no two edges are shared by layers. For call graph conditions $\mathbf{c}$ that describe $\mathbf{t}$, we introduce layer conditions $\mathbf{c}^j$, $j \in \{1, 2, \ldots, l+1\}$. Layer condition $\mathbf{c}^j$ encodes the attributes of the remaining portion of the call graph after the sequence of layers $[\mathbf{t}^1, \mathbf{t}^2, \ldots, \mathbf{t}^{j-1}]$ has been generated, and we define $\mathbf{c}^1 := \mathbf{c}$ and $\mathbf{c}^{l+1} := \emptyset$. We decompose the conditional call graph distribution as a chain of conditional layer distributions:

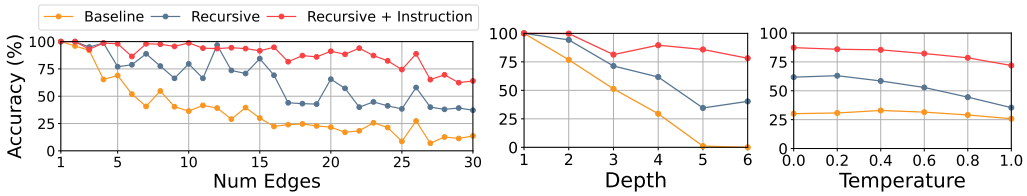
$$p(\mathbf{t}|\mathbf{c}) = \prod_{k=1}^{l} p(\mathbf{c}^{k+1}, \mathbf{t}^k | \mathbf{c}^k). \tag{3}$$

In other words, the model predicts call graphs from user prompts iteratively layer-by-layer. For layer k the model takes conditions $\mathbf{c}^k$ as input and produces the sequence of edges $\mathbf{t}^k$ followed by the conditions $\mathbf{c}^{k+1}$ of the next layer. The model-generated conditions $\mathbf{c}^{k+1}$ are then re-used as inputs to predict the next layer, $k+1$. Figure 2 illustrates an example of a recursively generated call graph.

Intermediate Instructions. We find that the model often has trouble generating consistent and correct next layer conditions $\mathbf{c}^{k+1}$ based on the current layer edges $\mathbf{t}^k$ and conditions $\mathbf{c}^k$. For instance, the conditions will violate physical constraints by requesting a layer that has higher latency than the overall call graph. Inspired by recent work demonstrating that LLM ability improves when explicitly forced to reason step-by-step [44, 27], we propose including a series of natural language reasoning steps that reinforce the model's ability to adhere to constraints. For example, we include a step-by-step calculation to find the number of remaining edges based on the *Num Edges* attribute in $\mathbf{c}^k$ and the number of edges generated in $\mathbf{t}^k$. We include these *intermediate instructions* immediately before the next layer conditions $\mathbf{c}^{k+1}$. We give an example of these reasoning steps in §A.3.

4.3 Training

Pre-Training. We pre-train the model on text-encoded call graphs using next-token prediction. The purpose of this stage is to adapt the general-purpose LLM, which was previously trained to model text sequences mostly consisting of natural language, to the more specialized domain of microservice call graphs. We follow the recursive scheme outlined in §4.2, leaving out intermediate instructions to focus mainly on modeling call graph structure. The model learns to generate the layer conditions $\mathbf{c}^j$ followed by the layer edges $\mathbf{t}^j$ sequentially layer-by-layer for each training sample. A pre-training example is shown in §A.3.



(a) Accuracy by the number of edges. (b) Accuracy by depth. (c) Varying temperatures.

Figure 3: Call graph generation accuracy with varying (a) edges and (b) depth in prompt using greedy sampling. (c) shows the accuracy with varying temperature parameters. Accuracy measures the fraction of generated traces that are valid and follow the initial instructions. As shown, both recursive generation and intermediate instructions help to increase the validity of the synthetic traces.

Instruction Tuning. We perform supervised fine-tuning after pre-training to improve the model's ability to generate call graphs following user instructions. We again follow the recursive scheme from §4.2, this time including the intermediate instructions. Different from pre-training, the model does not predict the initial call graph attributes $\mathbf{c}$ (equivalent to the first layer conditions $\mathbf{c}^1$), which are now treated as a fixed prompt. The user can supply additional natural language instructions for the model, and in §5.3, we provide results for two types of additional instruction. We further supplement the instructions with additional prompts, which can be programmatically generated from a template based on the user-requested attributes, to aid the model reasoning's abilities, as detailed in §A.3.

5 Evaluation

We evaluate our synthetic trace generation method across four dimensions: structured reasoning, distribution similarity, instruction-following capabilities, and downstream task performance. We initialize our model from Llama-2 7B [39] and train with LoRA [10] on 1.36 million microservice call graph samples from the Alibaba v2022 dataset [25], corresponding to 1.1B tokens. We reserve 10% of these samples for validation. Instruction tuning datasets were created by randomly selecting 5% of the training graphs, reformatted for instruction tuning. The training lasted four epochs, using a temperature of 0.8 and top-K of 50 for trace generation, unless otherwise specified. Further details on data preprocessing and training hyperparameters are provided in Appendix A.

5.1 Structured Reasoning with Recursive Generation

This experiment demonstrates how recursive generation enhances LLMs' ability to accurately construct microservice call graphs. We evaluate the model by generating traces with specified `num_edges` and `depth`, comparing these to their corresponding requested values. A trace is deemed accurate if it correctly matches the specified `num_edges` and `depth` and adheres to all structural constraints, such as valid DAG formations and appropriate start/finish times for communications, detailed in Appendix B. We conducted 50 generations for each (`num_edges`, `depth`) pair across ranges of $1 \leq$ `num_edges` ≤ 30 and $1 \leq$ `depth` ≤ 6, assessing the accuracy of our model against the baseline.

Baseline. We compare the recursive generation method with a Llama-2 7B model fine-tuned on the tabular data format [3] of call graph traces. At the beginning of each training sample, we include `num_edges` and `depth` so that we can specify the desired configuration to the model. We keep the same training environments along with the number of training tokens. During generation, the baseline generates all the edges of a call graph in one sequence for each instruction.

Results. Figure 3a and Figure 3b present the accuracy of call graph generations across varying numbers of edges and depths. Generally, as complexity increases (i.e., more edges or greater depth), the baseline model's accuracy decreases significantly—dropping below 25% for edges greater than 15 and nearing zero for depths above four. In contrast, the recursive generation model maintains higher accuracies, approximately 30% and 35%, respectively. This improved performance is attributed to the model breaking down complex generation tasks into simpler, more manageable sub-tasks.

Figure 3c illustrates how accuracy varies with the temperature parameter during decoding. Both models show decreased performance as the temperature increases, but the recursive model consistently outperforms the baseline, maintaining about 10% higher accuracy even at a temperature of 1.

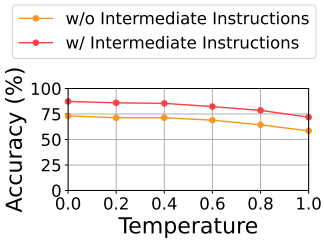
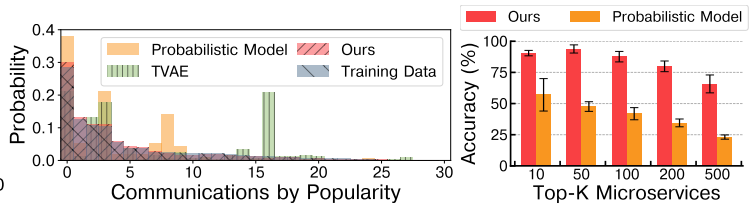


Figure 4: Impact of intermediate instructions on accuracy.



(a) Distribution of popular edges. (b) Heavy-hitter prediction.

Figure 5: Distribution similarity between real and synthetic traces.

Further, instruction tuning enhances model accuracy—from 23% to 36%—by directing the model to adhere to specific generation instructions, such as the number of edges per layer, which are outlined in §A.3. Notably, removing the intermediate instructions during instruction tuning results in an approximate 13% decrease in accuracy across all temperatures, as shown in Figure 4.

5.2 Similarity between Real and Synthetic Traces

To evaluate the quality of synthetic traces, we compare similarities between real traces from the training dataset and synthetic ones. We generate 50K call graph traces using prompts generated by the validation datasets and compare to the call graphs in the validation dataset.

Baselines. We compare the following synthetic trace generation methods:

- **Probabilistic model**: Probabilistic model based microservice call graph generators by Alibaba [24]. The model is designed to follow the random distribution of different statistics, such as communication types and the number of children per depth, and does not generate any time-related fields (e.g., communication finish time).
- **TVAE [46]**: Tabular data generative model using VAE [7]. We choose TVAE as a baseline since the model typically performs better in generating tabular data than other GAN [7] models as shown in [3]. Since tabular data cannot be used to generate traces, we use the baseline only to compare distributions of generated edge attributes. Also, to limit the training data size, we randomly choose 100k training samples from the trace dataset and use SDV [30] to train.

We omit similarity comparison results of the baseline from §5.1 since it shows similar results with our method as both use the same LLM architecture to learn distributions of call graph attributes.

Distribution of Popular Calls. Realistic synthetic traces should mirror real-world communication patterns. To assess this, we analyze the distribution of calls, defined by the attributes `(Source, Destination, Communication type)`. Figure 4a illustrates the distributions of the 100 most popular calls generated by our method and the baselines, limited to the top 30 due to space constraints.

The KL divergence for traces generated by our model is 0.16, indicating close similarity to the training data, whereas the probabilistic model's divergence is significantly higher at 3.84, due to its random selection processes. TVAE shows an intermediate divergence of 0.74, which is better than the probabilistic model but still less accurate than our method in capturing popular call distributions.

Heavy-hitter Prediction. The capability to generate heavy-hitter microservices—defined as top-K microservices triggered in a sequence of call graphs—is critical for tasks such as resource optimization and anomaly detection in microservice management. In this experiment, we select 1K traces from the validation dataset and create instructions consisting of a service ID and call graph attributes such as depth and the number of edges. These instructions guide the synthetic trace generation for both the baseline and our models. We evaluate the accuracy by comparing the top-K microservices between the synthetic and validation traces over 20 runs.

Figure 4b illustrates the accuracy for varying K values, ranging from 10 to 500. Our method demonstrates robust performance, maintaining over 90% accuracy for $K \leq 50$ and 65% at K=500. In contrast, the probabilistic model starts at 59% accuracy for K=10 and declines to 23% at K=500, showcasing our method's capability to capture and replicate heavy-hitter dynamics in synthetic traces.

Table 1: Instruction-following accuracy (%).

Prompt Type	w/ inst.	w/o inst.
High latency	66.9	20.2
Uncommon communication	82.2	30.2
Combined	59.3	40.3

Table 2: Downstream task accuracy (%).

Task	Llama-2 7B	Ours
Prediction	60.6	76.8
Infilling an attribute	41.0	70.9
Infilling an edge	24.3	66.2

5.3 Instruction-following Capability

Enabling users to specify desired characteristics of synthetic data is crucial for trace generators. We assess our instruction-tuned model's capacity to reflect user-requested attributes in the generated traces accurately. We evaluate the model's ability to produce call graphs featuring specific attributes (high latency and uncommon communications). Additionally, we explore the model's performance when prompted with a combination of these attributes not present in the training data.

When constructing the instruction tuning training datasets, we embed specific instructions to guide the generation of call graphs:

- **High Latency:** Instructions specify that call graphs should exhibit latencies above the 90th percentile (p90) of the training dataset's latency distribution, which varies by service. For example, the instruction might read: `Build a call graph with high latency`.
- **Uncommon Communications:** Instructions indicate that the call graph layer should include a communication occurring in less than 10% of the training data. An instruction example is: `Include an edge from (SRC) to (DEST) with (TYPE) communication type`.

We intentionally avoid combining these specific instructions in training samples to test the model's response to novel instruction combinations during inference.

Results. Table 1 presents the instruction-following accuracy for high latency and uncommon communication. We assessed this by filtering 1K validation instructions to see how many generated call graphs met the defined criteria (e.g., exceeding p90 latency). We also compared these results against outputs generated without specific instructions to evaluate the impact of tailored prompts on model performance.

Additionally, we examine the model's performance when both instructions are combined in prompts, a scenario not covered in the training data. The model's ability to satisfy both conditions simultaneously, despite not being explicitly trained to do so, is detailed in the last row of Table 1. Higher accuracy in scenarios without specific instructions often results from inherent biases in attributes like service ID or the number of edges, which may align with the desired user outcomes.

5.4 Downstream Tasks

We extend our evaluation beyond generating synthetic traces, demonstrating the utility of our model in performing downstream tasks related to microservice traces. We focus on scenarios where partial information from distributed environment traces is available, emphasizing the challenges posed by incomplete data. This section compares our fine-tuned model with the standard Llama-2 7B, which lacks specific training on call graph data, to highlight the importance of domain-specific training.

Predicting Uncommon Communication Patterns. A key task is predicting uncommon communication patterns based on the first 10 lines of a trace. We train the original Llama model and our adapted model for this binary classification task on 15K samples. Each sample's prompt comprised the first 10 edges of a real trace, with binary labels indicating the presence of uncommon communication patterns in the subsequent trace sections.

Results, detailed in the first row of Table 2, indicate that the original Llama model achieves only 60.6% accuracy, suggesting insufficient training for recognizing uncommon patterns. In contrast, our model achieves 76.8% accuracy, demonstrating its enhanced capability to interpret and predict based on partial trace data.

Infilling Missing Data. Missing data is common in large-scale trace logging, such as in Alibaba's microservice call graphs, where 67% of traces contain missing values [11]. This task focuses on fine-tuning our model to accurately infill missing data in microservice call graphs, considering partial

information. Specifically, we conduct two separate experiments on infilling (1) a missing attribute and (2) a missing call connecting two layers.

In the first experiment, we construct a training dataset with 1.2K questions, each containing a sequence of edges with one attribute marked as `[MISSING]`. The missing value is the unknown ground truth for prediction, so these are multi-class classification problems. Attributes targeted include communication type (e.g., HTTP, RPC) or destination microservice. We evaluate the model on a 6K-sample test dataset, where our model demonstrated over 70% accuracy in predicting the correct attributes, significantly outperforming the baseline Llama-2 model's accuracy by about 30% as reported in the second row of Table 2.

The second experiment's dataset comprises 1K samples, each representing a pair of parent and child layers with a missing connecting edge tagged as `[MISSING]`. After training, we test both models on 5K test cases to generate the correct edge, ensuring the finish time matched or exceeded the start time. The last row of Table 2 shows that while the original Llama model scored only 24% accuracy, our model maintained a high accuracy of 66%, underscoring its robustness in more complex tasks.

These experiments demonstrate that the capabilities of our trace pre-trained model enable it to effectively handle infilling tasks with additional fine-tuning, even when facing complex data scenarios.

6 Related Work

Adapting LLMs for Specific Domains. Pre-trained LLMs are increasingly adapted for specialized domains due to their vast, diverse training datasets, which enable broad generalization capabilities. Examples include fine-tuning LLMs for programming [32], quantitative reasoning [17], healthcare [37], and semiconductor manufacturing [22]. Our paper is the first effort to apply this approach to the domain of computer system traces, which includes data with specific structures and constraints and focuses on generating synthetic data through fine-tuning.

Making Language Models Follow Instructions. Recent advancements have focused on enhancing LLMs' ability to follow instructions through prompting and prompt-tuning [19, 36, 15, 44] and instruction tuning [28, 43, 5, 33]. These two sets of methods are relevant to our setting since they augment powerful pre-trained LLMs (with or without updating the model weights) to improve their performance on new tasks. Unlike [27], which employs transformer models for task-specific performance enhancements, our approach seeks to refine output expressiveness within set prompts, aiming for greater fidelity in synthetic data production.

Multi-step Reasoning with LLMs. Iterating with LLMs over multiple steps is an effective strategy to solve complex problems. For instance, [47] suggests a tree-of-thoughts reasoning technique to solve problems by decomposing into smaller thoughts and letting language models explore diverse reasoning paths over different thoughts. To leverage the external knowledge outside of LLMs, [29, 1] propose to trigger tools during LLM inferences, while [40] interleaves retrieval with LLM inferences to solve multi-step question-answering. Multi-step reasoning is also useful to handle long-context scenarios by summarizing iteratively [42] and diving into subproblems [16]. In contrast to the above approaches, our work learns to generate prompts with specific structures and constraints for generating subsequent layers.

7 Limitations

While the recursive generation method shows advantages over generating an entire call graph trace at once in terms of correctly generating call graph structures, one of the major drawbacks is that previously generated edges are dropped, since we generate a call graph in a sequence of multiple layers, where only conditioning information from the previous layer is passed into the prompts of the next layer. While dropping previously generated results does not affect the outputs much in the case of microservice call graph generation (since microservices in direct neighborhoods influence each other the most [49]), we believe that providing past information, such as previous layers and even a time series of call graph traces, would be helpful. Furthermore, our method uses manually constructed instruction templates, which may lead to suboptimal generation quality, as we are not using the full potential of language models pre-trained with trillions of tokens [39]. Following the

methods of [21, 8, 18], we believe that diversifying instructions using LLM-generated output is a potential method to improve the ability of LLMs to follow user intentions.

8 Conclusion

This paper presents a training method for pre-trained LLMs tailored for generating microservice trace graphs through a recursive call graph generation scheme complemented by instruction tuning with intermediate instructions. Our model demonstrates superior performance in generating accurate and valid call graphs and shows better distribution similarity compared to baseline models. Our evaluation results highlight the effectiveness of instruction tuning in refining the generation of call graphs according to user-specified features, and reveal the potential for using our model in various downstream tasks, such as prediction and data infilling, by further training the model. While this paper focuses primarily on microservice call graphs, our approach holds promise for broader applicability to other types of computer system traces with similar structural characteristics.

References

[1] Renat Aksitov, Sobhan Miryoosefi, Zonglin Li, Daliang Li, Sheila Babayan, Kavya Kopparapu, Zachary Fisher, Ruiqi Guo, Sushant Prakash, Pranesh Srinivasan, et al. Rest meets react: Self-improvement for multi-step reasoning llm agent. *arXiv preprint arXiv:2312.10003*, 2023.

[2] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J. Christopher Beck. Generating complex, realistic cloud workloads using recurrent neural networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 376–391, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Vadim Borisov, Kathrin Sessler, Tobias Leemann, Martin Pawelczyk, and Gjergji Kasneci. Language models are realistic tabular data generators. In *The Eleventh International Conference on Learning Representations*, 2023.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[5] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

[6] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 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*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[8] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

[9] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[10] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[11] Darby Huye, Lan Liu, and Raja R. Sambasivan. Systemizing and mitigating topological inconsistencies in alibaba's microservice call-graph datasets. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24, page 276–285, New York, NY, USA, 2024. Association for Computing Machinery.

[12] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, Boston, MA, July 2023. USENIX Association.

[13] Azam Ikram, Sarthak Chakraborty, Subrata Mitra, Shiv Saini, Saurabh Bagchi, and Murat Kocaoglu. Root cause analysis of failures in microservices through causal discovery. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 31158–31170. Curran Associates, Inc., 2022.

[14] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Paul Schmitt, Francesco Bronzino, and Nick Feamster. Generative, high-fidelity network traces. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, page 131–138, New York, NY, USA, 2023. Association for Computing Machinery.

[15] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *ArXiv*, abs/2205.11916, 2022.

[16] Soochan Lee and Gunhee Kim. Recursion of thought: A divide-and-conquer approach to multi-context reasoning with language models. *arXiv preprint arXiv:2306.06891*, 2023.

[17] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857, 2022.

[18] Ming Li, Lichang Chen, Jiuhai Chen, Shwai He, Jiuxiang Gu, and Tianyi Zhou. Selective reflection-tuning: Student-selected data recycling for llm instruction-tuning. *arXiv preprint arXiv:2402.10110*, 2024.

[19] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online, August 2021. Association for Computational Linguistics.

[20] Zinan Lin, Alankar Jain, Chen Wang, Giulia Fanti, and Vyas Sekar. Using gans for sharing networked time series data: Challenges, initial promise, and open questions. In *Proceedings of the ACM Internet Measurement Conference*, pages 464–483, 2020.

[21] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36, 2024.

[22] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176*, 2023.

[23] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[24] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 412–426, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 355–369, New York, NY, USA, 2022. Association for Computing Machinery.

[26] C. Meng, S. Song, H. Tong, M. Pan, and Y. Yu. Deepscaler: Holistic autoscaling for microservices based on spatiotemporal gnn with adaptive graph learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 53–65, Los Alamitos, CA, USA, sep 2023. IEEE Computer Society.

[27] Maxwell Nye, Anders Andreassen, Guy Gur-Ari, Henryk Witold Michalewski, Jacob Austin, David Bieber, David Martin Dohan, Aitor Lewkowycz, Maarten Paul Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2021. https://arxiv.org/abs/2112.00114.

[28] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155, 2022.

[29] Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

[30] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. The synthetic data vault. In *IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 399–410, Oct 2016.

[31] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[32] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[33] Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*, 2021.

[34] Junhong Shen, Neil Tenenholtz, James Brian Hall, David Alvarez-Melis, and Nicolo Fusi. Tag-llm: Repurposing general-purpose llms for specialized domains, 2024.

[35] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[36] Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *arXiv preprint arXiv:2010.15980*, 2020.

[37] Karan Singhal, Shekoofeh Azizi, Tao Tu, S Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Tanwani, Heather Cole-Lewis, Stephen Pfohl, et al. Large language models encode clinical knowledge. *arXiv preprint arXiv:2212.13138*, 2022.

[38] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations*, 2021.

[39] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[40] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. *arXiv preprint arXiv:2212.10509*, 2022.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[42] Qingyue Wang, Liang Ding, Yanan Cao, Zhiliang Tian, Shi Wang, Dacheng Tao, and Li Guo. Recursively summarizing enables long-term dialogue memory in large language models. *arXiv preprint arXiv:2308.15022*, 2023.

[43] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

[44] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[45] Zhe Xie, Haowen Xu, Wenxiao Chen, Wanxue Li, Huai Jiang, Liangfei Su, Hanzhang Wang, and Dan Pei. Unsupervised anomaly detection on microservice traces through graph vae. In *Proceedings of the ACM Web Conference 2023*, WWW '23, page 2874–2884, New York, NY, USA, 2023. Association for Computing Machinery.

[46] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Modeling tabular data using conditional gan. *Advances in neural information processing systems*, 32, 2019.

[47] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[48] Yucheng Yin, Zinan Lin, Minhao Jin, Giulia Fanti, and Vyas Sekar. Practical gan-based synthetic ip header trace generation using netshare. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 458–472, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Y. Zhang, Z. Zhou, S. Elnikety, and C. Delimitrou. Ursa: Lightweight resource management for cloud-native microservices. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 954–969, Los Alamitos, CA, USA, mar 2024. IEEE Computer Society.

[50] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, Carlsbad, CA, July 2022. USENIX Association.

A Training Details

A.1 Training Setup

We train all models with 4× A100 80GB GPUs in our cluster with the hyperparameters described in Table 3. We apply LoRA ([10]) adapters to query and key projection matrices of attention layers with $rank = 8$, $alpha = 16$, and $dropout = 0.1$. For the downstream task training in §5.4, we freeze the backbone model and only train the last classification layer for the prediction task. For the infilling downstream task, we use LoRA adapters with the same configuration as mentioned earlier.

A.2 Training Data Preprocessing

From the Alibaba microservice v2022 traces [25], we use the first-hour call graph traces as our training data. The traces are collections of API calls, where each API call includes communication information between the two microservices. Note that the dataset anonymizes the service and microservice names. Service ID is a nine-digit number starting with the prefix "S_" instead of using a real service name (e.g., social network), and microservice is a five-digit number starting with the prefix "MS_". We construct call graphs using the `trace ID` field (i.e., API calls with the same trace ID belong to one call graph). When constructing call graphs, we remove calls with missing information (e.g., destination microservice IDs are unknown) and remove call graphs that are not connected (e.g., missing edges). To remove redundancy, we filter out call graphs that have the same structure and fields (e.g., service ID, latency) for all API calls. The distributions of training data after removing redundancy are shown in Figure 6.

A.3 Training Data Examples

From the call graph traces, we create text-based representations of call graphs as described in §4.1. First of all, Figure 7 is a training data example of converting a call graph into a tabular data format, which is the baseline in §5.1. At the beginning, we include high-level information about the call graph including service ID, the number of edges, and depth of the call graph. Each line inside the `<edges>` block corresponds to a call in a call graph. 6 fields exist for each call including the edge ID, source/destination microservices, communication type, and communication start/finish time.

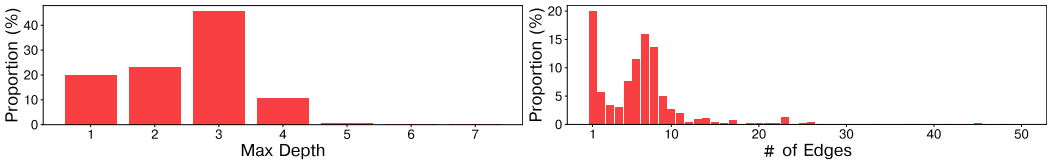
Figure 8 shows an example training data sample for recursive generation as described in §4.2. Each sample consists of a sequence of layers, where each layer includes the edges and the conditions for the next layers. At the beginning of each layer, we provide high-level information to explain connections with the previous layers (e.g., `start_node`, `caller`), structure in the call graph (e.g., `remaining_depth`, `num_edges`, `start_edge_id`), and time-related information (e.g., `latency`, `start_communication_at`). Note that the number of fields in each edge is reduced from 6 to 5 since the edges share the same start node. Also, the edge ID field is an integer, not a dot-decimal number. For each next layer, the condition is described in each `<subgraph>` block starting with the edge ID to be extended.

Figure 9 is an example of instruction-tuning data. The instruction starts with a system prompt followed by conditions as in Figure 8. We further explain the condition in natural language formats along with user-requested features as studied in §5.3. In the output section, we include Chain-of-Thought scratchpads at the end of `<edges>` block and at the beginning of `<subgraph>` blocks, which elaborate on the number of edges to generate and constraints of subgraph conditions. For instance, the scratchpad includes descriptions regarding the depth requirement to let LLMs understand better that the depth field should be decreased by 1 from the current layer's depth.

As described in §4.1, we drop each call graph attribute randomly with probability p_{drop}. We set p_{drop} to 0.9 for all attributes except for the service ID field, which is always kept ($p_{drop} = 1$), to ensure minimal conditioning.

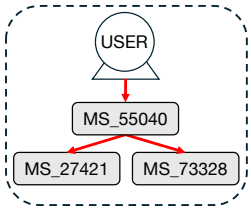
B Constraints in Call Graph Layers

In this section, we describe constraints to be met for each generated call graph layer to be correct. First of all, the generation results are considered invalid if the output does not have the valid format with `<edges>` and `<subgraph>` tags.



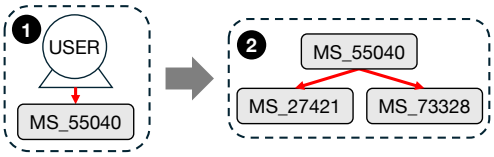
(a) Distribution by call graph depth. (b) Distribution by the number of edges.

Figure 6: Training data distribution after preprocessing steps.



[GENERATE GRAPH]
num_edges:3/id:S_058367691/max_depth:2
<edges>
(source is USER,communication finishes at 2 milliseconds,communication starts at 0 milliseconds,type is rpc,destination is MS_55040,edge_id is 0)
(communication starts at 1 milliseconds, source is MS_55040,communication finishes at 1 milliseconds,destination is MS_27421,edge_id is 1,type is db)
(communication finishes at 2 milliseconds,type is db,communication starts at 1 milliseconds,destination is MS_73328, source is MS_55040 ,edge_id is 2)
</edges>

Figure 7: A training data sample of a call graph with 3 edges represented in tabular format.



[GENERATE GRAPH]
num_current_edges:1/num_subgraphs:1/num_edges:3/start_communication_at:0/id:S_058367691/remaining_depth:1/
start_edge_id:0/start_node:USER
<layer>
<edges>
(communication finishes at 2 milliseconds,communication starts at 0 milliseconds,type is rpc,destination is MS_55040,edge_id is 0)
</edges>
<subgraph of edge_id 0>
latency:2/start_node:MS_55040/start_edge_id:1/num_subgraphs:0/num_current_edges:2/id:S_058367691/num_edges:2/
remaining_depth:0/start_communication_at:1/caller:USER
</subgraph>
</layer>

start_edge_id:1/id:S_058367691/latency:2/num_subgraphs:0/num_edges:2/remaining_depth:0/start_communication_at:1/
num_current_edges:2/start_node:MS_55040/caller:USER
<layer>
<edges>
(communication starts at 1 milliseconds,communication finishes at 1 milliseconds,destination is MS_27421,edge_id is 1,type is db)
(communication finishes at 2 milliseconds,type is db,communication starts at 1 milliseconds,destination is MS_73328,edge_id is 2)
</edges>
</layer>

Figure 8: A training data sample of a call graph with 3 edges for recursive generation.

Instruction

```
You are a trace generator that creates traces based on given requirements.
Requirements:
start_communication_at:0/start_node:USER/remaining_depth:2/num_current_edges:1/num_edges:4/latency:12/id:S_032647104
Conditions:
In each edge, communication start time should NOT be greater than latency 12 milliseconds
Generate subgraph instructions if necessary
the first start_communication_at should be requirement's start_communication_at 0
Also, communication should finish before latency 12 milliseconds
copy caller from requirement's start_node:USER
generate 1 edges following num_current_edges
```

Output

```
<layer>
<edges>
(edge_id is 0,type is http,communication starts at 0 milliseconds,destination is MS_57649,communication finishes at 12 milliseconds)
</edges>
num generated edges = the last edge id - the first edge id + 1 = 0 - 0 + 1 = 1
1 edges generated out of num_edges:4
num_remaining_edges = num_edges:4 - generated:1 = 3

generate subgraphs of edge:0
Subgraph constraints:
remaining_depth should be the requirement's remaining_depth:2 - 1 = 1
num_edges <= num_remaining_edges:3
copy start_node from edge 0 destination: MS_57649
<subgraph of edge_id 0>
remaining_depth:1/start_edge_id:1/num_edges:3/id:S_032647104/latency:12/num_subgraphs:1/num_current_edges:2/
start_node:MS_57649/start_communication_at:1/caller:USER
</subgraph>
now, num_remaining_edges is 3 - 3 = 0
finish generation
</layer>
```

Figure 9: A training data sample of a call graph layer for instruction-tuning.

Table 3: Training setup and hyperparameters.

Model	Hyperparameter	Value
Pre-Training & Instruction Tuning	Optimizer	AdamW ([23])
	Learning rate	3e-4 with cosine scheduler
	Batch size	64
	Gradient clipping	1.0
Downstream Task Fine-tuning	Optimizer	AdamW
	Learning rate	1e-4 with cosine scheduler
	Batch size	2
	Gradient clipping	1.0

Edges. For each edge, we check the following conditions. First of all, each edge should include the 5 fields: edge ID, destination, communication type, and communication start/finish time. Secondly, we check whether the right number of edges are generated as described in the condition. Third, the communication start time should be equal to or greater than the communication start time described in the condition, and should not be greater than the communication finish time of the edge. Lastly, the communication finish time should be equal to or less than the latency field in the condition.

Next Layer Conditions. For the next layer conditions, we first check whether the next layer conditions should be generated or not. If the remaining depth field in the instruction is 0 or the number of edges that need to be generated is 0, no `<subgraph>` blocks should be generated.

Then, we check the validity of each field in the next layer conditions. First of all, the edge ID inside the `<subgraph>` block should be found in the edges generated in the current layer. For the depth, the remaining depth field should be less than the remaining depth of the instruction. Additionally, at least one of the resulting subgraphs must have a depth that is reduced by one compared to the original graph. For the `start node` and `caller` fields, they should be copied from the destination from the parent edge and the start node from the instructions, respectively. Lastly, we check the latency and communication start time by comparing the values to those of the parent edge. The latency of a

child layer should not be greater than the communication finish time of the parent edge. Also, the communication start time of a child layer should not be less than the communication start time of the parent edge.

After generating both edges and the next conditions, we check if the sum of the number of edges matches the number of edges in the instruction.