# Microservice Trace Generation with Large Language Models

**Donghyun Kim** [1] **Sriram Ravula** [1] **Alexandros G. Dimakis** [1] **Daehyeok Kim** [1] **Aditya Akella** [1]

## Abstract

Microservice architectures have emerged as a predominant framework for building distributed systems in large-scale enterprises due to their modular nature and scalability advantages. While characterizing microservice traces can help optimize and manage distributed systems with microservice architecture, using public microservice traces has limits such as their time-specific nature and incomplete data. As a result, generating synthetic traces using machine learning techniques is a promising alternative.

This paper presents a novel approach to generating microservice traces using Large Language Models (LLMs). Leveraging the power of LLMs to learn multiple tasks and to align with prompt instructions, we create multiple fine-tuning tasks from the trace dataset and fine-tune pre-trained LLMs. During trace generation, we introduce *Coarse-to-fine* generation scheme, which first produces high-level trace information and then uses this to generate more detailed, fine-grained traces. *Trace Oracle* component is employed to validate the LLM-generated traces, ensuring their accuracy and relevance. Our results with OpenLLaMA 7B model demonstrate the effectiveness of this method, with the fine-tuned model generating valid microservice traces with an accuracy of up to 88.7% and successfully adhering to the fine-tuning task instructions with a compliance rate of up to 93.2%.

## 1. Introduction

Microservice architectures have become the standard approach for constructing distributed systems within large-scale enterprises (Ferdman et al., 2012; Gan et al., 2019). This architectural style breaks down monolithic applications into smaller, intercommunicating software services. Such decomposition enhances the autonomy of various development teams, accelerates deployment rates, and allows for more precise scaling (Newman, 2021). However, the benefits of microservice architecture come at a cost; due to its complex dependencies within each other, characterizing applications requires an understanding of intricate interactions between microservices, which is crucial for optimizing and managing the system (Qiu et al., 2020; Bhardwaj et al., 2023).

Hence, microservice traces are key in simplifying this complexity, facilitating more efficient system maintenance and problem-solving. For instance, traces are pivotal in resource management tasks like autoscaling (Luo et al., 2022), where analyzing trace data aids in forecasting request patterns. This predictive ability is essential for timely resource allocation and deallocation, ensuring both efficiency and optimal performance. Additionally, microservice traces become indispensable to analyze the root cause of bottlenecks and errors (Ikram et al., 2022).

Accessing comprehensive microservice traces, however, poses significant challenges. While there are some public traces available, they often come with notable downsides. Firstly, they are typically obtained from a specific time range, which limits their applicability to different scenarios. Furthermore, these publicly available traces frequently suffer from missing or incomplete data, making them less reliable for thorough analysis. Another approach to obtaining microservice traces involves deploying applications of microservices independently. However, the sheer number of nodes necessary to deploy all applications can be prohibitively large (Luo et al., 2022; Huye et al., 2023), posing a significant barrier to the detailed study of microservice behaviors. These difficulties underscore the need for more accessible and versatile methods of obtaining microservice trace data.

The utility of synthetic traces (Yin et al., 2022; Bergsma et al., 2021) in this ecosystem cannot be overstated. Unlike natural trace data, synthetic traces offer an unlimited size, which is a significant advantage for extensive testing and analysis. They also enable the simulation of various conditions, such as stress-testing environments, that might be challenging to replicate in real-world scenarios. Furthermore, the deployment of applications in a microservice architecture often necessitates a substantial number of nodes. In contrast, generating synthetic traces requires consider-

[1]The University of Texas, Austin. Correspondence to: Aditya Akella <akella@cs.utexas.edu>.

ably fewer resources, presenting a more efficient alternative. A notable application of synthetic traces lies in infilling missing data, a common issue even in public datasets. By leveraging synthetic traces, gaps in collected data can be effectively addressed, enhancing the robustness of the dataset.

In this paper, we propose a method to fine-tune pre-trained Large Language Models (LLMs) (Touvron et al., 2023; Brown et al., 2020) to generate synthetic microservice traces. Leveraging the power of LLMs to learn multiple tasks (Sanh et al., 2021; Chung et al., 2022), we create multiple fine-tuning tasks from the trace dataset by augmenting traces with instructions. In addition, we suggest *Coarse-to-fine* generation scheme that generates traces in multiple stages using the tasks learned during fine-tuning. Furthermore, *Trace Oracle* component complements the probabilistic nature of LLM inference by validating generated traces. We fine-tune OpenLLaMA-7B v2 model (Geng & Liu, 2023) with Alibaba microservice traces (Luo et al., 2022) and show the fine-tuned model can generate valid microservice traces by up to 88.7% and satisfy instructions of fine-tuning tasks well by up to 93.2%.

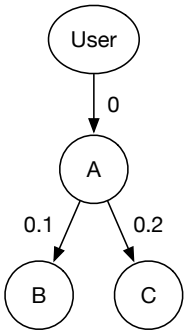
## 2. Background

### 2.1. Microservice Traces

In modern software architecture, an application is typically constructed as a constellation of multiple microservices (Gan et al., 2019; Luo et al., 2022; Huye et al., 2023), each with specific functionalities and dependencies on one another. When users interact with these applications, for instance by sending HTTP requests to web servers, it triggers a complex sequence of communications among these microservices. Each user request forms a Directed Acyclic Graph (DAG), aptly termed a microservice dependency graph, which maps the flow and dependencies of the microservices involved in fulfilling the user's request.

Figure 1 is an example of a microservice dependency graph with three microservices involved to process the user's request. The vertices of each graph correspond to microservices, while the edges correspond to API calls invoking the microservices. Each edge originates at the requesting service and terminates at the target service. Each graph is represented as a 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. database lookup), and relative start time.

Microservice traces, therefore, can be understood as a chronological series of such microservice dependency graphs. These traces encapsulate the interaction dynamics of all applications running on a given cluster over a period, offering a comprehensive view of the microservice landscape and their interdependencies. The analysis of microservice traces plays a pivotal role in understanding complex application architectures (Ikram et al., 2022) and optimizing cluster management (Qiu et al., 2020; Meng et al., 2023).




(a) Microservice dependency graph

**Arrived at**: 01H 20M 30S 040ms

**Application ID**: APP_12

**Edges:**

| API ID | Started (ms) | Source | Dest. | Type | Latency (ms) |
|---|---|---|---|---|---|
| 0 | 0 | USER | A | HTTP | 15 |
| 0.1 | 3 | A | B | DB | 2 |
| 0.2 | 5 | B | C | HTTP | 10 |

(b) Attributes of microservice dependency graph as tabular format

*Figure 1.* An example of microservice dependency graphs and attributes of communications between microservices.

### 2.2. Microservice Trace Generation Problem Definition

Our goal is to train a generative model for microservice traces using a dataset of directed microservice dependency graphs. We want that the model (1) generates outputs with correct structure, e.g. only valid DAGs; (2) takes optional conditioning information to simulate varying system environments in a controllable manner; and (3) generalizes to out-of-distribution graph structures and service conditions.

We turn to transformer-based architectures, as these show excellent performance in various domains including natural language (Brown et al., 2020; Touvron et al., 2023), time series data (Nie et al., 2023), climate modeling (Nguyen et al., 2023), and vision (Dosovitskiy et al., 2021; Caron et al., 2021; Oquab et al., 2023; He et al., 2022), as well as demonstrating the ability to model many domains at once (Reed et al., 2022). Since our dataset represents graphs using natural language descriptors, we can pose our problem as a sequence modeling task and leverage pre-trained, publicly available large language models (LLMs) as *foundation models* that we can fine-tune for our problem. It has been shown that LLMs can be adapted to generate structured data in new domains (Borisov et al., 2023), satisfying requirement (1). LLM outputs can be conditioned in a variety of arbitrary ways, including via natural language prompting (Ouyang et al., 2022) and structured input sequences (Borisov et al., 2023), meeting requirement (2). Finally, large transformer-based models pre-trained in a self-supervised manner (as with autoregressive generative LLMs) on extensive datasets demonstrate superior generalization to out-of-distribution tasks, satisfying requirement (3).

We now formalize our problem. Given a dataset of mi-

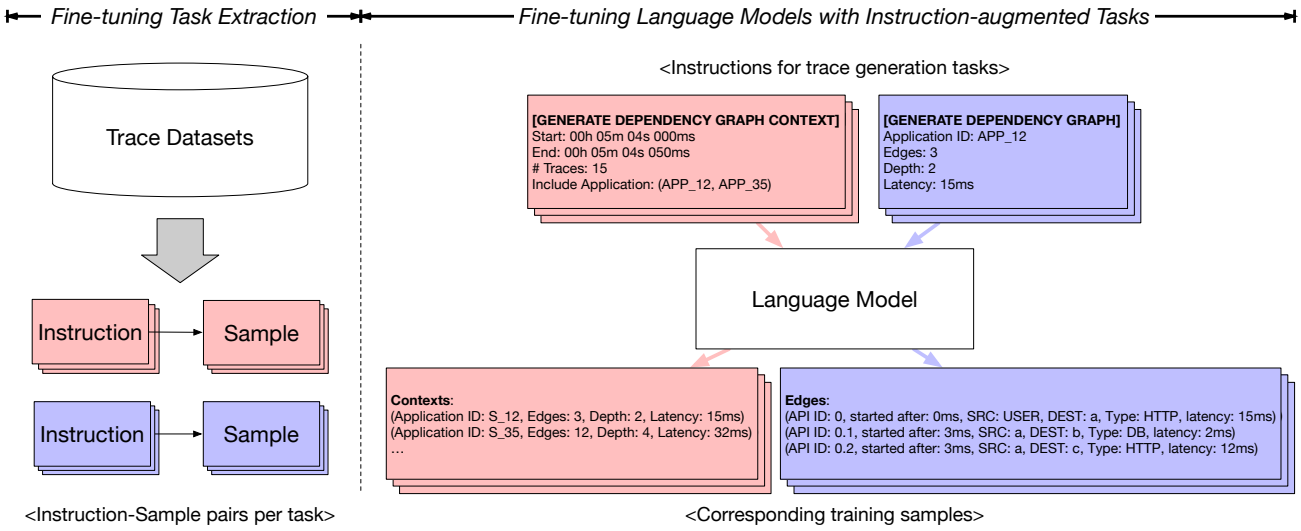



*Figure 2.* **Our proposed fine-tuning method overview.** We create fine-tuning tasks that consist of instruction and sample pairs from the trace dataset. Then, we fine-tune pre-trained large language models with the tasks extracted from the trace dataset.

croservice traces $\{x_1, x_2, \ldots, x_n\}$, we wish to learn the data distribution, $p(x)$. We approach the problem by decomposing each trace into a sequence of symbols, or tokens, $x := (s_1, s_2, \ldots, s_m)$, where sequence length $m$ varies depending on the trace $x$. We factorize the data distribution as the product of conditional probabilities over the decomposed sequence:

$$p(x) = \prod_{i=1}^{m} p(s_m | s_1, s_2, \ldots, s_{m-1}). \tag{1}$$

Using this representation of the distribution, we parameterize a generative model $p_\theta(x) := \prod_{i=1}^{t} p_\theta(s_t | s_1, s_2, \ldots, s_{t-1})$ (with weights $\theta$) over sequences representing microservice traces. We propose to train our model in an autoregressive, self-supervised manner to predict the next token in the sequence given the previous tokens.

## 3. Methodology

### 3.1. Trace Data into Multiple Fine-tuning Tasks

In our method, we dissect the challenge of microservice trace generation into a series of detailed fine-tuning tasks. This process involves extracting precise instructions from the training data traces, thereby enriching them with supplemental context. These enriched traces, articulated in natural language, serve as the foundation for fine-tuning pre-trained Large Language Models (LLMs). The integration of these instruction-augmented tasks ensures comprehensive learning of both the distribution and the structural intricacies of microservice traces.

Specifically, we generate several sets $\mathcal{C}_i = \{(c, y)_j\}_{j=1}^{n_i}$ of structured prompts $c$ and their responses $y$ using our original microservice trace dataset. Each set $\mathcal{C}_i$ corresponds to a different type of instruction, with the goal of creating a model suited for diverse and possibly unseen generation tasks at inference time. Noting that the responses are sequences of tokens $y = (s_1, s_2, \ldots, s_m)$, we factorize the target data distribution for a given prompt $c$ as

$$p(y|c) = \prod_{i=1}^{m} p(s_m | c, s_1, s_2, \ldots, s_{m-1}). \tag{2}$$

We denote the prompt with the single letter $c$ for convenience, though $c$ is a sequence of tokens as well. We parameterize our generative model given prompt $c$ as $p_\theta(x|c) := \prod_{i=1}^{t} p_\theta(s_t | c, s_1, s_2, \ldots, s_{t-1})$. We fine-tune our pre-trained model weights $\theta$ over the examples in each instruction set $\mathcal{C}_i$ to predict the next token of target response $y$ given the prompt $c$. Since we create each instruction-response task with no manual human input (besides defining the prompt format), this is a weakly supervised learning objective.

**Task types and instruction templates.** We define several tasks for microservice trace generation:

- Dependency graph generation: given prompt $c$ with graph-level information (e.g. depth and number of edges), generate the edges of the corresponding graph as a microservice trace. Note that this corresponds to the trace generation task we describe in Section 2.2 when $c$ is blank, meaning that our prompt-based tuning method is a generalization of "vanilla" autoregressive training.

- Dependency graph context generation: given $c$ with information about a system's activity at a specific time interval, generate graph-level information about the microservice dependency graphs created during that interval.

We extract prompts from traces for each task to finetune LLMs with instructions. Table 1 shows the manually designed templates of instructions and training data. Instructions are generated by concatenating those instructions, where the values are extracted from training data. For generalizability, we randomly shuffle the order of attributes and drop some attributes in each instruction. We illustrate our approach in Figure 2.

**Benefits of separating into multiple tasks.** The development of prompt-guided trace generation addresses critical inefficiencies inherent in trace data. Predominantly, trace data are characterized by high skewness, resulting in substantial redundancy. In the case of Alibaba microservice traces (Luo et al., 2022), the traces feature small dependency graphs (e.g., 1 or 2 edges), resulting in 55% being duplicates of those in the remaining 45%. This redundancy not only bloats datasets but also risks overfitting machine learning models to specific trace structures, thus impairing their generalizability. Conventionally, the naive removal of redundant traces has been practiced; however, this approach disrupts the natural distribution of training data. A sophisticated method that eliminates redundancy while preserving the underlying data distribution is essential.

Furthermore, the efficacy of this approach is enhanced when considering the constraints imposed by sequence length limits in the transformer modules. A simple approach would be to fit a variety of trace structures within a single sequence, but this would sacrifice the ability to make long-term predictions. Prompt-guided trace generation circumvents this by extracting key distribution-related information from the traces and converting them into concise instructions. This technique allows for the inclusion and learning of a broader spectrum of trace information during the fine-tuning process of models, thereby optimizing their learning capacity and efficiency.

### 3.2. Trace Generation with Fine-tuned Models

**Coarse-to-fine Trace Generation** We generate microservice traces through *Coarse-to-fine* scheme, where traces are generated in multiple stages from coarse-grained trace contexts to fine-grained dependency graph structures. For each step, we leverage the tasks defined in Section 3.1. The defined tasks for trace generation have close relationships with each other so that the output of a task can be leveraged as an instruction for other tasks.

We generate synthetic traces $x$ using system-level information $c_1$ as a conditioning signal. Since the desired graph properties are poorly specified in the conditioning signal, it is challenging for our model to generate sharp and diverse samples. To improve the quality of generated samples, we propose a two-stage sampling process. First, we use $c_1$ to generate graph-level context information $c_2$ using the model. The context information $c_2$ is then used as a conditioning signal to generate the final output traces $x$. This corresponds to factorizing the desired conditional distribution as $p(x|c_1) = p(x|c_2)p(c_2|c_1)$. By introducing the latent variable $c_2$ as an intermediate generation step, we allow the model to sharpen and add detail to an initial coarse output.

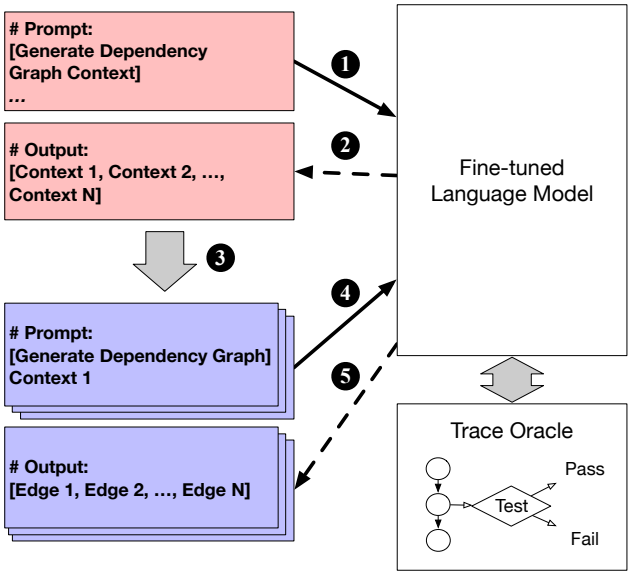



*Figure 3.* Trace generation workflow using the fine-tuned language model.

The generation workflow is illustrated in Figure 3. First, the model generates dependency graph contexts (❶, ❷). The generated contexts are broken down into multiple instructions for dependency graph generation tasks (❸), and those instructions are passed to the model to generate the dependency graphs (❹, ❺).

**Trace Validation using Trace Oracle** Since there are no guarantees that LLMs are always generating the right outputs, we introduce the concept of *Trace Oracle* to ensure the validity of generated traces during generation. This *Trace Oracle* evaluates the generated traces, ensuring they not only conform to pre-defined constraints but also incorporate structures akin to those in the training data. The validation process involves checking that the generated traces form a connected Directed Acyclic Graph (DAG). Furthermore, *Trace Oracle* can be used to only include edges or subgraphs present in the training data. This methodology guarantees that the generated traces are both structurally sound and

| Dependency Graph Generation | | |
|---|---|---|
| Attribute | Template | Description |
| Application ID | `ID:<string>` | Application ID |
| Number of Edges | `num edges:<integer>` | Number of API calls in the graph |
| Maximum Depth | `max depth:<integer>` | Maximum depth of the graph |
| Request Latency | `latency(ms):<integer>` | Time taken to finish the user request |
| **Dependency Graph Context Generation** | | |
| Attribute | Template | Description |
| Number of Traces | `num traces:<integer>` | Number of dependency graphs to generate |
| Start Time | `start:<datetime>` | Time of the first dependency graph in the window |
| End Time | `end:<datetime>` | Time of the last dependency graph in the window |
| Application IDs | `include:<string><string>,...,<string>` | A list of application IDs that should be included in the window |

*Table 1.* Templates of attributes included in instructions for each task.

reflective of the learned patterns from the training dataset. Whenever the generated traces do not satisfy conditions, *Trace Oracle* sends the request to the model again and retries generation.

## 4. Evaluation

**Dataset** We use Alibaba microservice traces (Luo et al., 2022) as our training data. We obtain 19 million microservice dependency graphs from the first 5 hours of public traces after filtering data with missing or incomplete information. The data are obtained from more than 10K applications total with 17K microservices. Also, after extracting fine-tuning tasks, we remove redundant dependency graphs from training data. We use 2% and 1% of the trace dataset as validation and test data, respectively.

**Training Details** We select pre-trained Open LLaMA-7B v2 model (Geng & Liu, 2023) as our fine-tuning target. For fine-tuning, we leverage LoRA technique (Hu et al., 2022) (rank=8, alpha=16, dropout=0.1) adapting only attention weights as suggested in the original paper. We fine-tune the pre-trained model for 1 epoch of training data with batch size 64 using AdamW (Loshchilov & Hutter, 2017) optimizer with maximum learning rate 3e-4.

### 4.1. Trace Validity

We first report the quality of generated dependency graphs in terms of validity. We break down validity conditions into two aspects, structural and semantic validity. In the case of structural validity, we check whether the generated outputs form connected DAGs with proper attributes in each field. We define a generated dependency graph as semantically valid when all edges can be found in the training data. An example case of semantically invalid outputs can include edges connecting two different applications.

Figure 4 shows the percentage of valid dependency graphs out of 10K generations varying the temperature parameter (Ackley et al., 1985) during the decoding phase. We use

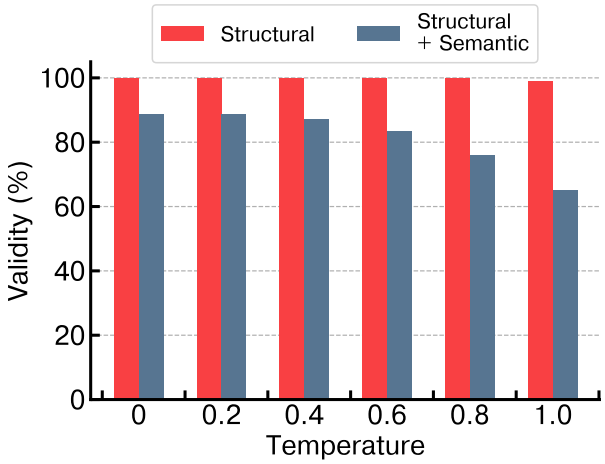



*Figure 4.* Validity of generated microservice dependency graphs varying temperature.

prompts from the test dataset to generate dependency graphs. When only considering the structural validity, the fine-tuned LLM can generate at least 99.0% valid dependency graphs. When including semantic validity, the validity drops from 88.7% to 65.2% by increasing the temperature parameter.

### 4.2. Accuracy in Following Instructions

To see whether the model learned fine-tuning tasks, we evaluate whether the model follows instructions in prompts. We compare values in instructions with the generated outputs and count the number of generations that satisfy all conditions in the instructions. For instance, in the case of the *Number of Traces* attribute, we count the number of traces in the generated output and compare it with the value in the instruction.

Figure 5 reports the accuracy of 1K generation queries from dependency graph context generation tasks. The accuracy ranges from 71.8% to 93.2% showing that the fine-tuned model generally follows instructions well. Most of the failure cases are from the *Application IDs* condition, where all the application IDs in instructions should be found in generated traces.

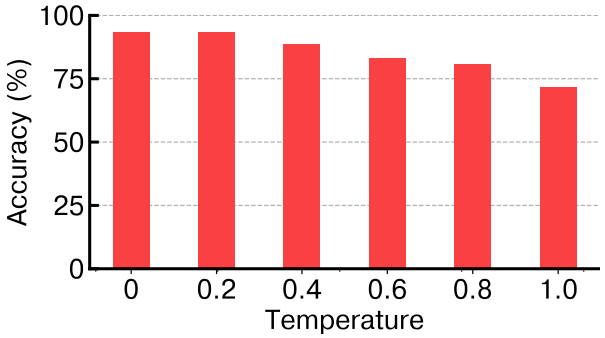



*Figure 5.* The percentage of generations that satisfy all conditions specified in prompts varying temperature.

## 5. Conclusion

In this paper, we propose a fine-tuning method to use pre-trained LLMs for microservice trace generations. Leveraging the power of LLMs to learn multiple tasks, we create multiple fine-tuning tasks from the trace dataset by augmenting traces with instructions. During generation, *Coarse-to-fine* trace generation scheme allows generating traces in multiple stages using outputs from context generation tasks as instructions for dependency graph generation tasks. To complement the probabilistic nature of LLM inference, *Trace Oracle* checks the generated trace validity and retries invalid cases. Using our fine-tuning and generation schemes, we show that the fine-tuned LLaMA-7B model can generate valid dependency graphs by up to 88.7% and follow conditions specified in the instructions by up to 93.2%.

## References

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. A learning algorithm for boltzmann machines. *Cogn. Sci.*, 9:147–169, 1985.

Bergsma, S., Zeyl, T., Senderovich, A., and Beck, J. C. Generating complex, realistic cloud workloads using recurrent neural networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pp. 376–391, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483590.

Bhardwaj, R., Kandasamy, K., Biswal, A., Guo, W., Hindman, B., Gonzalez, J., Jordan, M., and Stoica, I. Cilantro: Performance-Aware resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 623–643, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2.

Borisov, V., Sessler, K., Leemann, T., Pawelczyk, M., and Kasneci, G. Language models are realistic tabular data generators. In *The Eleventh International Conference on Learning Representations*, 2023.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020.

Caron, M., Touvron, H., Misra, I., Jégou, H., Mairal, J., Bojanowski, P., and Joulin, A. Emerging properties in self-supervised vision transformers. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2021.

Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, E., Wang, X., Dehghani, M., Brahma, S., et al. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*, 2022.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pp. 37–48, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450307598. doi: 10.1145/2150976.2150982.

Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., Lin, R., Liu, Z., Padilla, J., and Delimitrou, C. 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, pp. 3–18, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304013.

Geng, X. and Liu, H. Openllama: An open reproduction of llama, May 2023. URL https://github.com/openlm-research/open_llama.

He, K., Chen, X., Xie, S., Li, Y., Dollár, P., and Girshick, R. Masked autoencoders are scalable vision learners. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 15979–15988, 2022. doi: 10.1109/CVPR52688.2022.01553.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Huye, D., Shkuro, Y., and Sambasivan, R. R. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 419–432, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-35-9.

Ikram, A., Chakraborty, S., Mitra, S., Saini, S., Bagchi, S., and Kocaoglu, M. Root cause analysis of failures in microservices through causal discovery. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 31158–31170. Curran Associates, Inc., 2022.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Luo, S., Xu, H., Ye, K., Xu, G., Zhang, L., Yang, G., and Xu, C. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pp. 355–369, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394147. doi: 10.1145/3542929.3563477.

Meng, C., Song, S., Tong, H., Pan, M., and Yu, Y. 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)*, pp. 53–65, Los Alamitos, CA, USA, sep 2023. IEEE Computer Society. doi: 10.1109/ASE56229.2023.00038.

Newman, S. *Building microservices: designing finegrained systems*. O'Reilly Media, Inc., 2021.

Nguyen, T., Brandstetter, J., Kapoor, A., Gupta, J. K., and Grover, A. Climax: A foundation model for weather and climate. In *International Conference on Machine Learning*, 2023.

Nie, Y., Nguyen, N. H., Sinthong, P., and Kalagnanam, J. A time series is worth 64 words: Long-term forecasting with transformers. In *The Eleventh International Conference on Learning Representations*, 2023.

Oquab, M., Darcet, T., Moutakanni, T., Vo, H. V., Szafraniec, M., Khalidov, V., Fernandez, P., Haziza, D., Massa, F., El-Nouby, A., Howes, R., Huang, P.-Y., Xu, H., Sharma, V., Li, S.-W., Galuba, W., Rabbat, M., Assran, M., Ballas, N., Synnaeve, G., Misra, I., Jegou, H., Mairal, J., Labatut, P., Joulin, A., and Bojanowski, P. Dinov2: Learning robust visual features without supervision, 2023.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L. E., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., and Lowe, R. J. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155, 2022.

Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., and Iyer, R. K. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 805–825. USENIX Association, November 2020. ISBN 978-1-939133-19-9.

Reed, S., Zolna, K., Parisotto, E., Colmenarejo, S. G., Novikov, A., Barth-maron, G., Giménez, M., Sulsky, Y., Kay, J., Springenberg, J. T., Eccles, T., Bruce, J., Razavi, A., Edwards, A., Heess, N., Chen, Y., Hadsell, R., Vinyals, O., Bordbar, M., and de Freitas, N. A generalist agent. *Transactions on Machine Learning Research*, 2022. ISSN 2835-8856. Featured Certification, Outstanding Certification.

Sanh, V., Webson, A., Raffel, C., Bach, S. H., Sutawika, L., Alyafeai, Z., Chaffin, A., Stiegler, A., Scao, T. L., Raja, A., et al. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*, 2021.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Yin, Y., Lin, Z., Jin, M., Fanti, G., and Sekar, V. Practical gan-based synthetic ip header trace generation using netshare. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pp. 458–472, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394208. doi: 10.1145/3544216.3544251.