GAMMA: Graph Neural Network-Based Multi-Bottleneck Localization for Microservices Applications

Gagan Somashekar
gsomashekar@cs.stonybrook.edu
PACE Lab, Stony Brook University
Stony Brook, NY, USA

Anurag Dutt
adutt@cs.stonybrook.edu
PACE Lab, Stony Brook University
Stony Brook, NY, USA

Mainak Adak
madak@cs.stonybrook.edu
PACE Lab, Stony Brook University
Stony Brook, NY, USA

Tania Lorido Botran
tbotran@roblox.com
Roblox
San Mateo, CA, USA

Anshul Gandhi
anshul@cs.stonybrook.edu
PACE Lab, Stony Brook University
Stony Brook, NY, USA

ABSTRACT

Microservices architecture is quickly replacing monolithic and multi-tier architectures as the implementation choice for large-scale web applications as it allows independent development, scalability, and maintenance. However, even with careful node scheduling and scaling, the microservices applications are still vulnerable to performance degradation due to unexpected (dependent or independent) events like anomalous node behavior, workload interference, or sudden spikes in requests or retries. These events can adversely affect the performance of one or more microservices (bottlenecks), degrading the overall application performance. To ensure a good customer experience and avoid revenue loss, it is crucial to detect and mitigate all bottlenecks swiftly.

This work introduces GAMMA, a novel, explainable graph learning model that integrates a mixture of experts to detect multiple bottlenecks. We evaluated GAMMA using a popular open-source benchmarking application deployed on Kubernetes under various practical bottleneck scenarios. Our experimental evaluation results show that GAMMA provides significantly better performance (46% higher F_1 score) than existing works that employ deep learning, machine learning, and statistical techniques, demonstrating its ability to detect multiple bottlenecks by learning complex interactions in a microservices architecture.

The dataset is made publicly available [49] for reproducibility and further research in the field.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; • **Computing methodologies → Neural networks**; • **Computer systems organization → Maintainability and maintenance**.

WWW '24, May 13–17, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0171-9/24/05…$15.00
https://doi.org/10.1145/3589334.3645665

KEYWORDS

microservices applications, bottlenecks, anomalies, graph neural network, dataset

ACM Reference Format:
Gagan Somashekar, Anurag Dutt, Mainak Adak, Tania Lorido Botran, and Anshul Gandhi. 2024. GAMMA: Graph Neural Network-Based Multi-Bottleneck Localization for Microservices Applications. In *Proceedings of the ACM Web Conference 2024 (WWW '24), May 13–17, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3589334.3645665

1 INTRODUCTION

Microservice architecture (MSA) is quickly becoming the choice of implementation for large-scale web applications owing to its modular nature [1, 9, 19, 24, 33, 34, 37, 54–56, 60]. Indeed, MSA is replacing monolithic and multi-tier application architectures as MSA designs applications as fine-grained, modular, and independent services called microservices, enabling independent development, scalability, and maintenance [19, 24, 33]. Even existing web and online gaming applications implemented using monolithic architecture are being transformed to MSA [1, 16, 21, 30, 36].

A critical problem for online web applications is performance management as it affects customer experience and revenue [10]. Among various aspects of performance management, detecting performance degradation and identifying the sources of performance degradation are crucial for providing a consistent user experience. We define *anomaly detection* as the process of detecting an application's performance degradation at the level of individual requests or over a time period, and *bottleneck localization* as the process of identifying which specific microservices are affecting the application's performance. Despite careful application design and proactive capacity planning, performance anomalies still happen due to unexpected surges in load or workload interference [24, 33, 56]. For that reason, bottleneck localization is a must. The microservices with degraded performance, i.e., *performance bottlenecks*, often arise due to resource saturation, resource contention, or microservice application misconfiguration [19, 20, 43, 51], and do not necessarily lead to errors or faults, making them difficult to detect.

Anomaly detection and bottleneck localization in MSA applications are challenging for various reasons. Firstly, bottlenecks can manifest in *different ways*, impacting one or more microservices, even propagating across microservices over time and sustaining even after the source of anomaly is mitigated [19, 23, 50].

This exacerbates bottleneck localization, multiplies the engineering hours needed to mitigate with time, and delays the restoration of applications' performance. Secondly, the effect of performance anomalies (e.g., host interference) differs across microservices. For example, two microservices hosted on the same node that is experiencing CPU saturation will react differently in terms of degradation depending on how compute-bound the microservices are. This necessitates solutions that learn the *unique characteristics* of the microservices. Thirdly, complex interactions among the microservices can complicate bottleneck localization. The dynamicity that arises from asynchronous calls, caching, queues, feature additions, deprecations, and design changes can further complicate these interactions. As such, the solution must utilize and learn from the *interactions* of these microservices. Lastly, the absence of publicly available datasets with metrics, traces, and logs containing multiple bottlenecks from various sources has hindered the ability of researchers to evaluate their methods for multi-bottleneck detection and localization [28].

The key challenge, and the focus of this paper, is the presence of ***multiple bottlenecks*** in MSA applications. Existing works, including those in recent editions of The Web Conference, have primarily focused on single bottlenecks and have ignored the practical case of multiple bottlenecks [34, 47, 58]. There are different ways in which multiple bottlenecks can arise in practice.

- Multiple, *independent* bottlenecks arise in one or more microservices. For example, a microservice responsible for logins could be bottlenecked due to a sudden spike in user logins, while simultaneously, another microservice could be bottlenecked due to resource contention at its host node. In such cases, all bottlenecks must be (independently) detected and mitigated.
- Multiple, *dependent* bottlenecks arise in one or more microservices, due to the same underlying problem. For example, if the underlying VM that hosts multiple microservices is under resource contention from different colocated VMs, then all microservices on this VM can experience performance degradation.
- Multiple, *cascading* bottlenecks appear in sequence in multiple microservices. For example, a database microservice that is experiencing workload interference can result in request queues building up in dependent microservices, causing their performance to degrade as well. Undetected, these bottlenecks can cascade to interacting microservices, increasing the number of bottlenecks over time. In such cases, it is important to first detect all bottlenecks, and then alleviate them to quickly revive the application performance.

Prior works in this space have mainly focused on anomaly detection [31, 58, 59] or providing solutions for single bottleneck localization that cannot be easily adapted for multiple bottleneck localization [18, 20, 22, 28, 34]. Even solutions that are capable (with some effort) of detecting multiple bottlenecks are not evaluated on traces or datasets with multiple bottlenecks [47]. FIRM [43] evaluates multi-bottleneck localization [43] but is not effective (Section 4) as it does not utilize the distributed traces to learn the complex interactions among microservices. To the best of our knowledge, *our solution is the first work specifically designed with multi-bottleneck anomaly detection and bottleneck localization for MSA in mind.*

This work introduces and evaluates GAMMA, a novel model to ***detect anomalies and multiple bottlenecks*** in web applications implemented using the microservices architecture. Specifically, GAMMA uses (a) an attention-based graph convolution network to *learn the complex interactions* between microservices, (b) a holistic multi-source end-to-end joint training framework to detect the presence of bottlenecks in an *explainable* manner, and (c) a mixture of experts to account for possibly *multiple bottlenecks* across microservices.

In designing and evaluating GAMMA, this work makes the following key contributions:

(1) We present the design of GAMMA, a holistic multi-source end-to-end joint training framework that learns complex interactions between microservices using an attention-based graph convolution trained over distributed traces of observable metrics (e.g., CPU and memory utilization), which are readily available in production systems [35]. Further, it uses a mixture of experts to learn the unique characteristics of microservices and account for possibly multiple bottlenecks across microservices.
(2) To evaluate GAMMA, we generate and open-source a dataset consisting of around 40 million request traces [49].
(3) We evaluate GAMMA against existing techniques on the above bottleneck dataset; we also extend a seminal prior work [20] created for localizing single bottlenecks to localize multiple bottlenecks.
(4) We perform a detailed ablation study to understand and explain the impact of telemetry on evaluation results.

Our experimental evaluation results show that GAMMA provides an F_1 score of up to 0.92 and 0.89 for anomaly detection and bottleneck localization, respectively. GAMMA significantly exceeds the performance of prior works (3–4× improvement for anomaly detection and 46% improvement for bottleneck localization) based on deep learning, machine learning, and statistical techniques, demonstrating its ability to detect multiple bottlenecks by learning complex interactions in microservices architecture.

Our analysis reveals that the performance gap between GAMMA and other baselines increases with the increasing complexity of the evaluation scenario. While existing works perform reasonably well when there is a single source of anomaly, their performance drops when evaluated in scenarios consisting of multiple sources of anomaly, unlike GAMMA. Further, while existing works can perform better if they are separately trained on each source of anomaly, GAMMA provides consistently better performance despite not being trained separately on individual anomaly sources, making GAMMA easier to deploy in practice. Finally, we show that GAMMA can provide explainability with its bottleneck localization, thereby aiding the bottleneck mitigation task.

2 RELATED WORK

Related prior works can be broadly categorized into (a) anomaly detection works, and (b) bottleneck localization (or root cause analysis) works. Since there are numerous prior works in these general areas, we limit our discussion below to closely related works and refer readers to relevant surveys for further detail [48].

2.1 Anomaly Detection

DeepTraLog [59] uses a unified graph embedded with log events, called trace event graphs, to represent the complex interaction among microservices. It finds anomaly scores for each trace or request by training a gated graph neural network-based deep support vector data description model on the trace event graphs. TraceVAE [58] is an unsupervised anomaly detection model that uses a novel dual-variable graph variational autoencoder with Negative Log-Likelihood (NLL) as the anomaly score. TraceAnomaly [31] is an unsupervised anomaly detection system that uses novel trace representation and deep Bayesian networks with posterior flow. The model is trained offline periodically to learn normal patterns in traces and then classifies traces as anomalous when they deviate from these learned patterns.

2.2 Bottleneck Localization

Groot [55] is Ebay's graph-based framework for bottleneck localization in MSA applications. Groot constructs a causality graph with events that include anomalies in metrics, abnormal log statements, etc., as the nodes and causal links between these nodes are based on domain knowledge. However, Groot requires domain knowledge for creating links between nodes and additionally requires continuous human involvement to track changes to the causal links between nodes. CRISP [60] is Uber's tool for critical path analysis over traces from MSA applications which can be used for anomaly detection and bottleneck localization. The critical paths in MSA, however, are dynamic [43], requiring constant recomputation of critical paths. Murphy [22] is an automated performance diagnosis system that detects bottlenecks in complex enterprise environments by monitoring data to define associations between entities in an MSA application. However, Murphy uses a linear model that cannot capture the complexities in production microservices [22, 23].

FIRM [43] proposes a Support Vector Machine (SVM) model for detecting bottlenecks on the critical path in the call graph. The SVM model is trained on hand-crafted features that capture the per-critical-path and per-microservice performance variability. FIRM only considers latency as a feature and also ignores the structural information in the call graphs of the MSA application, limiting its ability to detect multiple bottlenecks (as we show in Section 4.4).

Seer [20] is an online performance debugging system that leverages deep learning to detect and mitigate bottlenecks in MSA. Seer uses a hybrid network of Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) networks to learn spatial and temporal patterns that lead to bottlenecks. However, analysis of Alibaba's production systems suggests that CNN-based approaches fail to characterize complex graph dynamics and do not apply to real-world applications; instead, the authors suggest using GNNs [33]. Our evaluation of Seer on a dataset consisting of multiple bottlenecks further substantiates this claim (Section 4.4).

ϵ-diagnosis [47] uses a threshold technique to detect anomalies and distance correlation [53] to compare metrics of anomalous traces and normal traces for localizing bottlenecks. The localization algorithm runs on each microservice without utilizing any structural information available through distributed tracing. As we show in Section 4.4, this and other drawbacks significantly impact the performance of ϵ-diagnosis in the case of multiple bottlenecks.

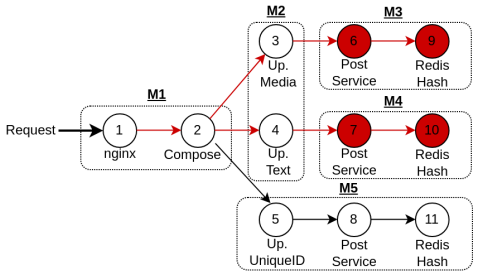


Figure 1: A simultaneous failure in machines M3 and M4 will affect the RPC calls in the invocation chain while other RPC calls in the call graph are not affected.

AutoMAP [34] relies on a heuristic algorithm using forward, self, and backward random walks on a graph representing the interaction between services to localize bottlenecks. Since it is a heuristic, AutoMAP may not be accurate and can suffer for large call graph sizes [6, 60]. B-MEG [50] is a two-staged graph-learning-based classifier that does anomaly detection and bottleneck localization in the first and second stages, respectively. However, B-MEG is only designed to detect single bottlenecks. Eadro [28] is a framework that uses traces, logs, and metrics along with multiple models to learn representations, which in turn are used to detect anomalies and localize bottlenecks jointly. The framework, owing to the series of models it uses, makes it difficult to interpret the results. MicroCU [25] is a framework that uses API logs and Granger causality to detect bottlenecks. Ablation studies on the importance of telemetry, traces, and logs in detecting bottlenecks reveals that logs provide the least information to detect bottlenecks [28]. Sage [18] uses a Causal Bayesian Network (CBN) to capture the dependencies between microservices. However, the assumption in Sage that the latency of non-leaf nodes in the call graph is determined by the wait time of its child nodes might not always hold (e.g., when a non-leaf child node is a message queue [33]). Moreover, Sage [18] can only work on call graph DAGs (no cycles), but call graphs in production systems have cycles [22, 33].

In summary, prior works provide solutions or evaluate their solutions ***only for single bottlenecks*** [18, 20, 28, 34, 43, 47, 50, 55] and do not fully utilize the rich telemetry and distributed tracing that is part of the MSA [25, 35, 43, 47]. Our work, described next, addresses this important gap by using a graph learning module to understand the complex interaction among microservices and a mixture of experts model to detect multiple bottlenecks effectively.

3 DESIGN OF GAMMA

Traditional bottleneck localization techniques (deep learning or heuristic-based) often operate in a linear or isolated manner, failing to capture the *dependencies and interactions* inherent in MSA [39]. Consider Figure 1, which shows a small subgraph of the entire social network call graph from DeathStarBench suite [19]. A simultaneous failure in Machines 3 and 4 will impact the corresponding *on-chain* RPC calls but will not affect the off-the-chain ones.

Graph Neural Networks (GNNs) are ideally suited to model such intricacies in graphical data and to capture dependencies between nodes [20, 50]. GNNs are designed to naturally assimilate and process information from nodes and their respective neighborhoods

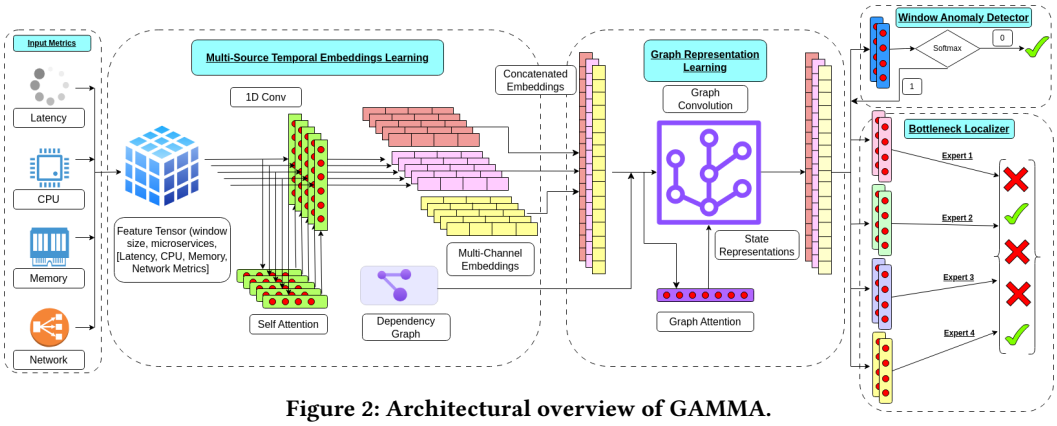


Figure 2: Architectural overview of GAMMA.

in a graph. GNNs can also handle the complexities of enterprise environments, especially cycles in the call graphs [57].

The key idea behind GAMMA is to understand patterns in call graphs using inputs from multiple system metrics and the graph dependency structure; this information can help identify the interconnections among microservices and guide system diagnosis. Figure 2 shows the architectural overview of GAMMA, which is broadly divided into 4 stages: Multi-Source Temporal Embeddings Learning, Graph-Representation Learning, Anomaly Classifier, and Bottleneck Localizer.

3.1 Multi-Source Temporal Embeddings Learning

Capturing temporal patterns helps to reveal the dynamic nature of system performance, highlighting fluctuations and evolving trends over time. Since bottlenecks may be induced due to episodic anomalies in the system, analyzing temporal chunks allows us to capture *correlations and sequences* across requests while also providing macroscopic trends in the system for the anomalous episodes [13, 59]. Multi-input temporal embeddings encapsulate the spatio-temporal behavior of a system, providing a comprehensive view of spatial relations and time-evolving patterns within a given window. Consider a call-graph with η microservices. For the microservices, we organize the system metrics into an η-dimensional time-series $\mathbb{M}^{\eta}$. The system metrics (e.g., RPC latency, CPU usage) act as our model features. We split the entire feature tensor $\mathbb{M}^{\eta}$ into windows of length τ, thus giving us window inputs of size $\mathbb{M}^{\eta x \tau}$. The parameter τ is trainable and is decided based on validation metrics during training. Analyzing windows as opposed to individual traces allows us to aggregate the temporal dynamics of the system.

The input tensor is processed using a Multivariate Temporal Convolution Attention Network, which is designed to recognize patterns over time. This network employs Dilated Causal Convolution (DCC), a method that efficiently captures relationships within and between features over time. DCC is highly scalable, and it has proven to be superior to traditional methods, like CNNs and LSTMs, especially when predicting future events based on past data. For a more detailed explanation of DCC, please refer to Appendix E.1.

3.2 Graph Representation Learning

In this stage, our goal is to understand the end-to-end status of the MSA application and provide a detailed overview of the entire system including the dependent interactions between the microservices themselves. This requires three key actions: (1) merging the multi-channel embeddings generated in the previous stage to get concatenated embeddings for each microservice; (2) incorporating the microservice call-graph and the concatenated embeddings to generate the dependency graph and microservice-level status representations (node representations) for the application; and (3) modeling this dependency graph. We begin by creating a directed graph from the call-graph that illustrates how microservices are interconnected. Next, we integrate the output embedding sourced from earlier stages into unified node representations, showcasing the status at the microservice-level. Information within this graph is then channeled through a GNN, enabling the understanding of neighboring interconnections and interactions.

3.2.1 Generating the Dependency Graph. The process of extracting a call graph from microservices traces can be systematically understood by visualizing microservices as nodes and their invocations as directed edges. A dependency graph $G = \{V, E\}$ can be derived from traces, where V denotes the set of nodes with $|V| = M$, with M being the total number of distinct microservices. E represents the set of edges; an edge $e_{a,b} = (v_a, v_b) \in E$ indicates a directed relationship from node v_a to node v_b, implying that the microservice associated with node v_b has made an invocation to that associated with node v_a at least once in recorded history.

Since it is essential for us to calculate temporal representations of microservices that capture both inter- and intra-feature correlations for our inputs, we *concatenate* our embeddings at an intermediate stage before we generate our dependency graph [26, 28]. Studies in cross-modal learning [27, 32, 38] hint that intermediate fusion tends to be more effective for processing temporal representations. Initially, we concatenate ($[\cdot\|\cdot]$) the representations of each microservice acquired from the prior phase, ensuring comprehensive data retention. The resulting tensor is then projected on a lower dimensional subspace by passing it through a fully-connected layer and subsequently passed through a Gated Linear Unit to fuse the representations while controlling for vanishing gradients and increasing resiliency to gradient forgetting [15]. The microservice

levels concatenated embeddings $\mathbb{O}^{\eta\times\epsilon}$ serve as node embeddings for the GNN with each node η_n having the embedding vector $\mathbb{O}^{\epsilon}_{\eta_n}$.

3.2.2 Graph-Attention Network. We employ the Graph Attention Network (GAT) [12], a specialized GNN variant that offers several advantages in the context of microservices [28]. Unlike traditional GNNs, GAT is capable of learning node and edge representations while dynamically assigning importance weights to neighboring nodes. This attention mechanism ensures that the network can *focus on more influential or anomalous microservices*, potentially acting as communication hubs or displaying abnormal behavior patterns. The local representation $\mathbb{O}^{\epsilon}_{\eta_n}$ encapsulates the feature set for individual nodes. The model digests this information and learns a holistic representation of the entire graph. For a more detailed explanation of Graph Attention, please refer to Appendix E.2.

3.3 Detection and Localization

In the final phase, GAMMA performs two functions: it predicts if a given observation window indicates an anomaly (anomaly detection), and if so, it discerns which microservices are the root cause (bottleneck localization). Contrary to traditional approaches [20, 43] which treat anomaly detection and bottleneck localization as separate functionalities, GAMMA adopts a holistic approach to leverage the knowledge of the inter-related functionalities.

Leveraging the earlier acquired representation $\mathbb{O}^{\zeta}$, an initial detector performs a binary assessment to ascertain the presence of any anomalies. If the outcome is negative, GAMMA directly presents the results. However, if an anomaly is detected, a subsequent localizer arranges the microservices in order of their likelihood to be the origin of the issue. This two-step mechanism, comprising the detector and the localizer, employs multiple experts comprising of connected neural networks followed by a binary classifier. Each microservice in the call-graph has a dedicated expert assigned to predict if the microservice is bottlenecked or not. Both these components, the detector and localizer, are trained in tandem with a shared goal. The model's primary focus is to curtail the total binary cross-entropy loss of the detector (λ_d) and localizer (λ_l). The joint loss function is given as:

$$\lambda_{total} = \alpha \cdot \lambda_l + \sum_{k\in\eta} \frac{(1-\alpha)}{\eta} \cdot \lambda_k, \tag{1}$$

where $\lambda_l = \sum_{k\in\eta} \lambda_k$; and α is a hyperparameter to tune the contribution of λ_l and λ_d towards the total loss. Should an anomaly be detected, GAMMA outputs a binary vector of 0s and 1s which predicts the bottleneck and non-bottlenecked microservices. We defer the details of hyperparameter tuning to Appendix C.

4 EVALUATION

We now present our experimental evaluation results for GAMMA under various bottleneck scenarios. We also compare GAMMA's performance with that of recent works on bottleneck localization.

4.1 Experimental Setup

We evaluate GAMMA on a cluster of 17 VMs (4 vCPUs, 8GB memory) managed by Kubernetes. The VMs are synchronized via NTP for accurate measurements. The metrics (CPU, memory, network) are collected via Prometheus [44], while Jaeger [4] collects distributed traces. To generate a variety of bottlenecks, we use a CPU load generator [2] and *stress-ng* tool to generate interferences on one or more host VMs. This generates multi-bottlenecks of varying intensities and duration that may overlap in time.

We use the popular social networking benchmark from DeathStarBench [19] that consists of 28 microservices implementing several features of real-world social networking applications. The constituent microservices are Nginx, Memcached, MongoDB, Redis, as well as microservices that implement the logic of the application. The workload consists of *Compose* requests that create a post, *User* requests that read the timeline of other users, and *Home* requests that read the user's own timeline. We use wrk2 [7] to generate workloads of different intensities. We benchmark the application to find the peak load (800 requests per second, or RPS) beyond which it is unstable. We use different intensities in the range of 100–800 RPS. We deploy monitoring services like Prometheus and Jaeger on a separate VM to avoid unintended interference. We provide more details about the social networking application in Appendix A.

4.2 Dataset Creation

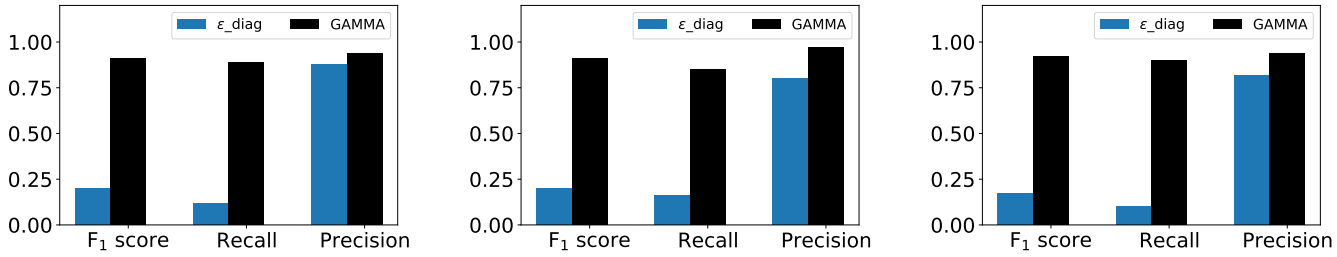
A key contribution of this work is constructing a dataset for research on anomaly detection and multi-bottleneck localization. Prior works have noted that existing public traces [42] on anomaly detection and bottleneck localization only contain single, severe bottlenecks that are not representative of real-world scenarios [50]. When such a bottleneck is introduced, the resulting latency increases by an order of magnitude (100×), making it trivial to detect that singe bottleneck using a simple grid search or threshold-based approaches.

To create a more realistic dataset that includes traces with *multiple bottlenecks* at *different intensities*, we carefully benchmarked the social networking application under different interference intensities and duration of interference. We chose intensities and duration values that degrade the application performance but do not cause any faults or errors that can be trivially detected. We induced interference on different VMs at different times and also simultaneously. A single VM could be induced with different types of interference (e.g., CPU and memory), resulting in the hosted microservices experiencing a mixture of interference patterns. The resulting dataset consists of around *40 million request traces* along with corresponding time series of CPU, memory, I/O, and network metrics. The dataset also includes application, VM, and Kubernetes logs. Appendix D provides additional information about the dataset. We have open-sourced the dataset [49] to facilitate further research in the area of performance management of microservices.

4.3 Metrics and Baselines

For evaluation of anomaly detection and bottleneck localization, we use the following performance metrics:

- *Recall* is the ratio of true positive predictions to the total number of positive data points. It measures how many of the positive data points were classified as positive by the model. A high recall is essential for MSA-based web application deployments as it is important to detect *all* anomalies and bottlenecks.



(a) Compose request type. **(b) User request type.** **(c) Home request type.**

Figure 3: F_1 score, Recall, and Precision for anomaly detection over the entire dataset.

- *Precision* is the ratio of true positive predictions to the total positive predictions. It measures how many of the data points that were classified positive by the model are actually positive. A high precision is desirable as it implies fewer engineer hours wasted investigating false positives.
- *F_1 score* is the harmonic mean of precision and recall. It is a metric that balances the trade-off between precision and recall.

We experimentally compare the performance of GAMMA with the following state-of-the-art baselines from recent works:

(1) *FIRM* is a framework that uses SVM and hand-crafted features to localize bottlenecks on the critical path. We use Scikit-learn library [41] to implement FIRM's SVM model.
(2) *ϵ-diagnosis* performs both anomaly detection and bottleneck localization. It uses a simple threshold scheme for anomaly detection and distance correlation for bottleneck localization. We use the dcor [46] library to implement the ϵ-diagnosis' localization module.
(3) *Seer* is an online bottleneck localization framework that uses CNN and LSTM to learn spatial and temporal features, respectively, to recognize patterns that lead to anomalies. We implement Seer using Pytorch [40].
(4) *Seer** is our modified version of Seer for multi-bottleneck localization which works by adapting softmax to individual binary classification for each microservice in the call-graph and replacing cross-entropy loss with hinge-loss [14].

To evaluate ϵ-diagnosis and FIRM on multi-bottleneck data, we run these baselines on all the microservices serially. Since the original Seer model cannot be directly applied for multi-bottleneck localization, we evaluate how well it localizes the most dominant bottlenecked microservice. We tune the hyperparameters of all baselines and present the best results in our evaluation.

4.4 Results

4.4.1 Aggregate results for anomaly detection. We start by evaluating GAMMA and the baselines using our entire dataset (with all resource bottleneck traces). Figure 3 shows the F_1 score, Recall, and Precision for anomaly detection using the entire trace dataset for GAMMA and ϵ-diagnosis. Note that ϵ-diagnosis is the only baseline among those considered that does anomaly detection. Starting with Figure 3a, which shows the results when analyzing Compose traces, we see that GAMMA provides significantly better results than ϵ-diagnosis. The F_1 score, Recall, and Precision values for GAMMA are 0.91, 0.89, and 0.94, respectively.

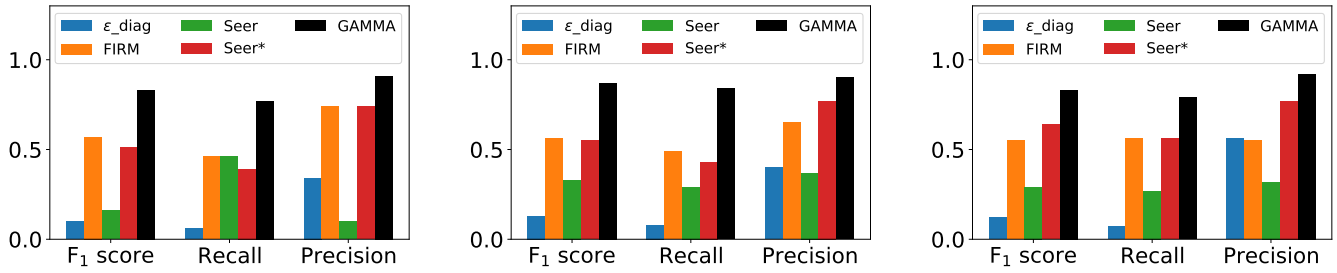
By contrast, the corresponding values for ϵ-diagnosis are lower by 78%, 87%, and 6%, respectively. We do observe that ϵ-diagnosis achieves reasonable Precision because of the low confidence threshold that its localizer uses, which ensures the quality of predictions. Recall, from Section 2, that ϵ-diagnosis does not leverage any structural information about the application, thus losing out on important information. Further, ϵ-diagnosis uses a static threshold to detect anomalies. While this threshold might work well for scenarios where only a single, severe performance bottleneck exists, this static threshold does not adapt to the more realistic case of multiple, different bottlenecks. In fact, when we evaluated ϵ-diagnosis for the simpler, pathological dataset where a single bottleneck exists that causes performance to degrade significantly [43], ϵ-diagnosis resulted in near-perfect F_1 scores. This underscores the difficulty in anomaly detection when multiple bottlenecks exist.

Results are similar for User (Figure 3b) and Home (Figure 3c) requests, with GAMMA significantly outperforming ϵ-diagnosis and achieving high performance values. Specifically, in Figure 3b, GAMMA's F_1 score (0.91) is 355% higher than that of ϵ-diagnosis (0.20). Likewise, in Figure 3c, GAMMA's F_1 score (0.92) is 441% higher than that of ϵ-diagnosis (0.17). We note that User and Home requests have smaller call graphs than Compose. Additionally, Compose has asynchronous calls, caches, queues, and other complexities, that are inherent in MSA applications, making Compose a popular choice for analysis in prior works [28, 43]. While we experimented with all request types, due to lack of space, we will primarily focus on the complex Compose request type in our results.

4.4.2 Aggregate results for bottleneck localization. Figure 4 shows our results for the more challenging bottleneck localization task using the entire trace dataset for GAMMA and all baselines. *Across all request types, GAMMA outperforms all other baselines for all performance metrics.* In particular, GAMMA achieves a high F_1 score of 0.83–0.87 across Figures 4a–4c. Further, GAMMA also achieves a high Recall of 0.77–0.84 and a high Precision of 0.90–0.92 across all subfigures.

Starting with Figure 4a, we see that GAMMA outperforms all other baselines under all metrics. GAMMA achieves an F_1 score, Recall, and Precision of 0.83, 0.77, and 0.91, respectively. ϵ-diagnosis again performs poorly, with an F_1 score of only 0.1; this is due to the weaknesses of ϵ-diagnosis identified above which limit its accuracy for the multi-bottleneck scenario.

FIRM performs better than ϵ-diagnosis, but still only achieves an F_1 score of 0.57 compared to the 0.83 (46% higher) obtained by GAMMA. This is likely because FIRM does not leverage the



(a) Compose request type. (b) User request type. (c) Home request type.

Figure 4: F_1 score, Recall, and Precision for bottleneck localization over the entire dataset.

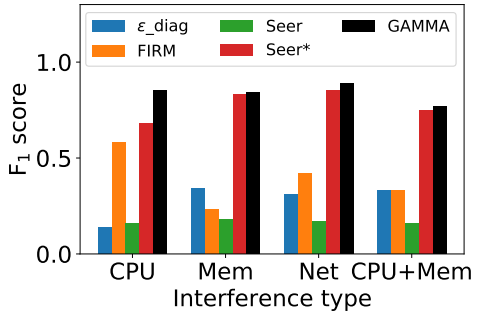


Figure 5: Bottleneck localization results by interference type.

structural information in the call graphs of the MSA application or resource metric timelines, unlike GAMMA. We note that FIRM can perform quite well if we only consider single bottleneck traces, again highlighting the challenge of dealing with multiple bottlenecks. When we evaluated FIRM for the simpler, pathological dataset where a single, severe bottleneck exists [43], FIRM resulted in a much higher F_1 score of 0.83 with a Recall and Precision of about 0.7 and 0.9, respectively.

Seer also performs poorly, with an F_1 score of only 0.16. This is to be expected, however, as the unmodified Seer only focuses on localizing one bottleneck. Since real-world request traces may contain multiple bottlenecks (e.g., our traces contain as many as 12 bottlenecks each), Seer's performance is limited. To account for this shortcoming, we extended Seer to Seer* by replacing softmax in the prediction layer with individual binary classification for every microservice in the call graph, and then replaced cross-entropy loss with hinge-loss [14], as discussed in Section 4.3. With this extension, Seer* performs better, with an F_1 score of 0.51. However, this is still significantly below GAMMA's F_1 score of 0.83. We believe this is because while Seer* does leverage multiple neural network models, it does not make use of GNNs, which are ideally suited to MSA application call graphs [33]. As we show later in Table 3, the GNN component of GAMMA is crucial for good performance.

The results for User (Figure 4b) and Home (Figure 4c) request types are qualitatively similar to that of Compose in Figure 4a, with GAMMA outperforming all other baselines for all metrics.

4.4.3 Results per bottleneck source. We now evaluate GAMMA and the baselines by separately training and testing over traces that contain bottlenecks from a specific source (CPU, Memory, Network, CPU+Memory). This will allow us to assess the performance under specific bottleneck types. Figure 5 shows the F_1 score for GAMMA and all baselines for bottleneck localization (under Compose request type) separated by the interference type that creates the bottleneck. (GAMMA continues to be significantly better than ϵ-diagnosis for anomaly detection so we omit those results.)

Across all subfigures, we see that GAMMA is always superior to the other baselines with an F_1 score of at least 0.77 and as much as 0.89 (for Network interference type). However, the performance of each baseline does differ across the subfigures. For example, FIRM performs much better when the bottlenecks are caused by CPU interference as opposed to other interference types. Seer* has the opposite behavior, with performance being close to that of GAMMA for non-CPU interference types, but worse for CPU interference. This suggests that specific baselines may perform better if they are separately trained for each source of bottleneck. However, this is tedious in practice. By contrast, GAMMA shows consistently good performance whether it is trained on each interference type (Figure 5) or more efficiently trained once on all interference types (Figure 4).

4.4.4 Overhead analysis. To compare the overhead of GAMMA and the baselines, we computed the average inference time across all traces for the combined tasks of anomaly detection and bottleneck localization, as applicable. Table 1 shows the overhead time in seconds for processing each window; for Seer*, which operates at the granularity of traces, we converted the times to per window by normalizing by the average number of traces in a window.

GAMMA	ϵ-diagnosis	FIRM	Seer*
3.87×10^{-5}	2.75×10^{-3}	1.30×10^{-6}	5.78×10^{-6}

Table 1: Average inference time (seconds) per window.

We see that GAMMA has a much lower overhead than ϵ-diagnosis, but is slower than FIRM and Seer*. Given the design of GAMMA, and its superior performance compared to FIRM and Seer*, we consider the larger inference time as a trade-off between performance and overhead. Regardless, we note that the overhead for GAMMA per 1s window is only about 38.7μs, representing a 0.004% overhead for each second of window length.

4.4.5 Explainability of GAMMA. A key advantage of GAMMA is its ability to not just localize bottlenecks, but also aid in identifying the source of the bottlenecks. Utilizing multi-source data-based approaches, such as GAMMA, offers a significant advantage over other approaches that only rely on latency traces to identify and localize bottlenecks. System metrics, such as CPU usage and network congestion, can offer crucial insights, such as trends, threshold breaches, and correlations, in detecting bottlenecks for large

MSA applications. GAMMA effectively integrates multiple system metrics with the microservice-dependency graph to understand cross-modal and temporal patterns for system interactions. To highlight this ability, we run GAMMA with different subsets of features. The intuition is that the feature whose omission causes a significant drop in performance is likely the source of the bottleneck.

Feature Omitted	CPU interference	Memory interference	CPU+Memory interference
None	0.851	0.844	0.771
CPU	0.693	0.805	0.714
Memory	0.789	0.573	0.600
Network	0.919	0.838	0.764
CPU & Mem	0.646	0.459	0.438

Table 2: Illustrating GAMMA's explainability by evaluating F_1 score when specific features are omitted from GAMMA.

The rows in Table 2 show the F_1 score when GAMMA is run with a specific subset of features for bottleneck localization. We consider three different bottleneck source scenarios, one per column: bottlenecks caused by (a) only CPU interference, (b) only Memory interference, and (c) CPU and Memory interference.

Starting with the only CPU interference scenario in the first column, we see that GAMMA's F_1 score drops from 0.851 to 0.693 when the CPU feature is omitted, but only drops to 0.789 when the Memory feature is omitted. When omitting the Network feature, the F_1 score actually increases to 0.919, suggesting that Network feature data may be hurting performance in this case. Overall, the results show that the CPU feature has a larger impact, suggesting that CPU saturation is the source of the bottleneck. We see similar trends across the other two columns in the table.

We acknowledge that the term "explainability" is broader than our focus, as it provides the reasoning behind a model's decision-making process. By design, GAMMA can provide the probability of confidence in its predictions since we essentially use binary classification in the last stage for both the anomaly detector and bottleneck localizer. For example, on average, the anomaly detector has a probability of prediction close to 0.72 and 0.84 for traces with memory and CPU interference, respectively. This probability metric can be obtained at the trace level for anomaly detection and bottleneck localization.

4.4.6 Ablation study for GAMMA. The GAMMA core model involves a few stages, as discussed in Section 3. Two specific stages of interest are the Graph Attention Network (which combines Graph Convolution with an attention mechanism allowing nodes to aggregate information from their most insightful neighbors) and Self Attention in context of causal convolution (which allows a temporal sequence to weigh the importance of its own past values). The former provides the ability to capture the interactions between microservices by leveraging the dependency structure. The latter ensures that the 1D-convolution operation, which is inherently local, is guided by a global understanding of the entire temporal sequence, ensuring a context-aware feature extraction.

To validate our design choices, we performed an ablation study by replacing the two specific stages of GAMMA with alternative ones. Table 3 shows our results over the entire dataset. Comparing

Stage Omitted	F_1 score	Recall	Precision
None	0.830	0.87	0.83
Graph Attention	0.669	0.69	0.65
Self Attention	0.695	0.71	0.68

Table 3: Ablation study to highlight the importance of specific stages of GAMMA.

row 1 (GAMMA, as-is) and row 2 (GAMMA with Graph Attention replaced with a standard fully-connected linear layer), we clearly see that the performance numbers drop significantly, indicating the importance of Graph Convolution in the design of GAMMA. Similarly, comparing row 1 with row 3 (GAMMA with Self Attention removed), we again see a drop in performance for all three metrics. This highlights the significance of self-attention in the design of GAMMA.

We find that while replacing GAT with a linear layer, the dependency-agnostic representations are not as helpful for localizing bottlenecks as GAMMA loses the ability to factor-in the neighborhood interactions in its inference. Removing self attention also has an adverse effect on the performance of GAMMA as the model becomes myopic, limited by the filter-size of the convolution layers, and loses its ability to hold long-term patterns.

5 CONCLUSION

Online web applications are increasingly adopting the microservices architecture (MSA). While modular and flexible, MSA applications have numerous microservices that interact with each other in complex ways, making it difficult to identify and pinpoint performance bottlenecks. Further, since multiple bottlenecks can arise independently or dependently for an MSA application, it is crucial to accurately detect and localize *all* performance bottlenecks.

This work focuses on the key gap in this problem space—the ability to detect ***multiple bottlenecks*** efficiently for MSA applications, a realistic use-case that has been mostly ignored by prior works. Our solution framework, GAMMA, learns complex interactions between microservices using graph neural networks and integrates this with a mixture of experts to enable multiple bottleneck localization. Evaluation results using the DeathStarBench Social Networking application highlight the superiority of GAMMA compared to several existing techniques. Further, our results show that GAMMA can be trained efficiently and performs well ***across bottleneck types***, unlike existing techniques. Finally, GAMMA's model lends itself to ***explainability***, making it practical for performance diagnosis.

We believe that GAMMA's graph-based model is inherently scalable due to the efficient handling of complex and large-scale interconnections between nodes (microservices) by GNNs. The modular design also enables independent scaling of resources. However, a thorough experimental evaluation is necessary to determine if any design changes are required, and we plan to pursue this in the future. Additionally, future work will involve experimental evaluations to assess how well GAMMA adapts to changing operational conditions in production environments.

ACKNOWLEDGMENT

This work was supported by NSF CNS grants 2106434 and 1750109.

REFERENCES

[1] [n. d.]. Building Microservices Driven by Performance–Roblox. https://medium.com/@acovarrubias_7488/building-microservices-driven-by-performance-b347ed1c48e3.

[2] [n. d.]. CPU Load Generator. https://github.com/molguin92/CPULoadGenerator.

[3] [n. d.]. Google-DeathStarBench. https://cloud.google.com/blog/products/management-tools/in-tests-cloud-profiler-adds-negligible-overhead.

[4] [n. d.]. Jaeger. https://www.jaegertracing.io/.

[5] [n. d.]. Microsoft-DeathStarBench. https://microsoft.github.io/VirtualClient/docs/workloads/deathstarbench/.

[6] [n. d.]. Uber's production Jaeger data. https://github.com/jaegertracing/jaeger-ui/issues/680.

[7] [n. d.]. wrk2 Workload Generator. https://github.com/giltene/wrk2.

[8] Randy Abernethy. 2018. *The Programmer's Guide to Apache Thrift.*

[9] Harold Aragon, Samuel Braganza, Edwin Boza, Jonathan Parrales, and Cristina Abad. 2019. Workload Characterization of a Software-as-a-Service Web Application Implemented with a Microservices Architecture. In *Companion Proceedings of The 2019 World Wide Web Conference* (San Francisco, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 746–750. https://doi.org/10.1145/3308560.3316466

[10] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024

[11] Daniel Beck, Gholamreza Haffari, and Trevor Cohn. 2018. Graph-to-Sequence Learning using Gated Graph Neural Networks. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Melbourne, Australia, 273–283. https://doi.org/10.18653/v1/P18-1026

[12] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How Attentive are Graph Attention Networks? arXiv:2105.14491 [cs.LG]

[13] Jian Chen, Fagui Liu, Jun Jiang, Guoxiang Zhong, Dishi Xu, Zhuanglun Tan, and Shangsong Shi. 2023. TraceGra: A trace-based anomaly detection for microservice using graph deep learning. *Computer Communications* 204 (2023), 109–117. https://doi.org/10.1016/j.comcom.2023.03.028

[14] Koby Crammer and Yoram Singer. 2002. On the Algorithmic Implementation of Multiclass Kernel-Based Vector Machines. *J. Mach. Learn. Res.* 2 (mar 2002), 265–292.

[15] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. 2016. Language Modeling with Gated Convolutional Networks. *CoRR* abs/1612.08083 (2016). arXiv:1612.08083 http://arxiv.org/abs/1612.08083

[16] Sinan Eski and Feza Buzluca. 2018. An Automatic Extraction Approach: Transition to Microservices Architecture from Monolithic Application. In *Proceedings of the 19th International Conference on Agile Software Development: Companion* (Porto, Portugal) *(XP '18)*. Association for Computing Machinery, New York, NY, USA, Article 25, 6 pages. https://doi.org/10.1145/3234152.3234195

[17] Huan Fu, Mingming Gong, Chaohui Wang, and Dacheng Tao. 2018. MoE-SPNet: A mixture-of-experts scene parsing network. *Pattern Recognition* 84 (2018), 226–236. https://doi.org/10.1016/j.patcog.2018.07.020

[18] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 135–151. https://doi.org/10.1145/3445814.3446700

[19] 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. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[20] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 19–33. https://doi.org/10.1145/3297858.3304004

[21] Jean-Philippe Gouigoux and Dalila Tamzalit. 2017. From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 62–65. https://doi.org/10.1109/ICSAW.2017.35

[22] Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, Brighten Godfrey, and Sujata Banerjee. 2023. Murphy: Performance Diagnosis of Distributed Cloud Applications. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 438–451. https://doi.org/10.1145/3603269.3604877

[23] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 73–90. https://www.usenix.org/conference/osdi22/presentation/huang-lexiang

[24] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. 2023. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 419–432. https://www.usenix.org/conference/atc23/presentation/huye

[25] Xinrui Jiang, Yicheng Pan, Meng Ma, and Ping Wang. 2023. Look Deep into the Microservice System Anomaly through Very Sparse Logs. In *Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23)*. Association for Computing Machinery, New York, NY, USA, 2970–2978. https://doi.org/10.1145/3543507.3583338

[26] Hamid Reza Vaezi Joze, Amirreza Shaban, Michael L. Iuzzolino, and Kazuhito Koishida. 2020. MMTM: Multimodal Transfer Module for CNN Fusion. arXiv:1911.08670 [cs.CV]

[27] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-Scale Video Classification with Convolutional Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 1725–1732. https://doi.org/10.1109/CVPR.2014.223

[28] Cheryl Lee, Tianyi Yang, Zhuangbin Chen, Yuxin Su, and Michael R. Lyu. 2023. Eadro: An End-to-End Troubleshooting Framework for Microservices on Multi-Source Data. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1750–1762. https://doi.org/10.1109/ICSE48619.2023.00150

[29] Xinjie Li and Huijuan Xu. 2023. MEID: mixture-of-experts with internal distillation for long-tailed video recognition. In *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence (AAAI'23/IAAI'23/EAAI'23)*. AAAI Press, Article 161, 9 pages. https://doi.org/10.1609/aaai.v37i2.25230

[30] Bo Liu, Jingliu Xiong, Qiurong Ren, Shmuel Tyszberowicz, and Zheng Yang. 2022. Log2MS: a framework for automated refactoring monolith into microservices using execution logs. In *2022 IEEE International Conference on Web Services (ICWS)*. 391–396. https://doi.org/10.1109/ICWS55610.2022.00065

[31] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, and Dan Pei. 2020. Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 48–58. https://doi.org/10.1109/ISSRE5003.2020.00014

[32] W. Liu, Wei-Long Zheng, and Bao-Liang Lu. 2016. Emotion Recognition Using Multimodal Deep Learning. In *International Conference on Neural Information Processing*. https://api.semanticscholar.org/CorpusID:7767769

[33] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 412–426. https://doi.org/10.1145/3472883.3487003

[34] Meng Ma, Jingmin Xu, Yuan Wang, Pengfei Chen, Zonghua Zhang, and Ping Wang. 2020. AutoMAP: Diagnose Your Microservice-Based Web Applications Automatically. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 246–258. https://doi.org/10.1145/3366423.3380111

[35] Jonathan Mace. 2017. *End-to-End Tracing: Adoption and Use Cases.* Survey. Brown University.

[36] Genc Mazlami, Jürgen Cito, and Philipp Leitner. 2017. Extraction of Microservices from Monolithic Software Architectures. In *2017 IEEE International Conference on Web Services (ICWS)*. 524–531. https://doi.org/10.1109/ICWS.2017.61

[37] Franck Michel, Catherine Faron-Zucker, Olivier Corby, and Fabien Gandon. 2019. Enabling Automatic Discovery and Querying of Web APIs at Web Scale Using Linked Data Standards. In *Companion Proceedings of The 2019 World Wide Web Conference* (San Francisco, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 883–892. https://doi.org/10.1145/3308560.3317073

[38] Micah M. Murray, Antonia Thelen, Silvio Ionta, and Mark T. Wallace. 2019. Contributions of Intraindividual and Interindividual Differences to Multisensory Processes. *J. Cognitive Neuroscience* 31, 3 (mar 2019), 360–376. https://doi.org/10.1162/jocn_a_01246

[39] Hoa Xuan Nguyen, Shaoshu Zhu, and Mingming Liu. 2022. A Survey on Graph Neural Networks for Microservice-Based Cloud Applications. *Sensors* 22, 23 (2022). https://doi.org/10.3390/s22239492

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith

Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[41] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[42] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar Iyer. 2020. Pre-processed Tracing Data for Popular Microservice Benchmarks. https://databank.illinois.edu/datasets/IDB-6738796. Online.

[43] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[44] Bjorn Rabenstein and Julius Volz. 2015. Prometheus: A Next-Generation Monitoring System (Talk). USENIX Association, Dublin.

[45] Elahe Rahimian, Golara Javadi, Frederick Tung, and Gabriel Oliveira. 2023. DynaShare: Task and Instance Conditioned Parameter Sharing for Multi-Task Learning. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 4535–4543. https://doi.org/10.1109/CVPRW59228.2023.00477

[46] Carlos Ramos-Carreño and José L. Torrecilla. 2023. dcor: Distance correlation and energy statistics in Python. *SoftwareX* 22 (2 2023). https://doi.org/10.1016/j.softx.2023.101326

[47] Huasong Shan, Yuan Chen, Haifeng Liu, Yunpeng Zhang, Xiao Xiao, Xiaofeng He, Min Li, and Wei Ding. 2019. ϵ-Diagnosis: Unsupervised and Real-Time Diagnosis of Small- Window Long-Tail Latency in Large-Scale Microservice Platforms. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 3215–3222. https://doi.org/10.1145/3308558.3313653

[48] Jacopo Soldani and Antonio Brogi. 2022. Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey. *ACM Comput. Surv.* 55, 3, Article 59 (feb 2022), 39 pages. https://doi.org/10.1145/3501297

[49] Gagan Somashekar, Anurag Dutt, Mainak Adak, Tania Lorido Botran, and Anshul Gandhi. 2024. Microservices Bottleneck Detection Dataset. https://doi.org/10.34740/KAGGLE/DSV/7638732

[50] Gagan Somashekar, Anurag Dutt, Rohith Vaddavalli, Sai Bhargav Varanasi, and Anshul Gandhi. 2022. B-MEG: Bottlenecked-Microservices Extraction Using Graph Neural Networks. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering* (Bejing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 7–11. https://doi.org/10.1145/3491204.3527494

[51] G. Somashekar, A. Suresh, S. Tyagi, V. Dhyani, K. Donkada, A. Pradhan, and A. Gandhi. 2022. Reducing the Tail Latency of Microservices Applications via Optimal Configuration Tuning. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE Computer Society, Los Alamitos, CA, USA, 111–120. https://doi.org/10.1109/ACSOS55765.2022.00029

[52] Ximeng Sun, Rameswar Panda, Rogerio Feris, and Kate Saenko. 2020. Adashare: Learning what to share for efficient deep multi-task learning. *Advances in Neural Information Processing Systems* 33 (2020).

[53] Gábor J. Székely, Maria L. Rizzo, and Nail K. Bakirov. 2007. Measuring and testing dependence by correlation of distances. *The Annals of Statistics* 35, 6 (2007), 2769 – 2794. https://doi.org/10.1214/009053607000000505

[54] Nicolas Viennot, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh. 2015. Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 21, 16 pages. https://doi.org/10.1145/2741948.2741975

[55] Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. 2022. Groot: An Event-Graph-Based Approach for Root Cause Analysis in Industrial Settings. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 419–429. https://doi.org/10.1109/ASE51524.2021.9678708

[56] Yingying Wen, Guanjie Cheng, Shuiguang Deng, and Jianwei Yin. 2022. Characterizing and synthesizing the workflow structure of microservices in ByteDance Cloud. *Journal of Software: Evolution and Process* 34, 8 (2022), e2467. https://doi.org/10.1002/smr.2467 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2467

[57] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

[58] Zhe Xie, Haowen Xu, Wenxiao Chen, Wanxue Li, Huai Jiang, Liangfei Su, Hanzhang Wang, and Dan Pei. 2023. Unsupervised Anomaly Detection on Microservice Traces through Graph VAE. In *Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23)*. Association for Computing Machinery, New York, NY, USA, 2874–2884. https://doi.org/10.1145/3543507.3583215

[59] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 623–634. https://doi.org/10.1145/3510003.3510180

[60] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 655–672. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

A BENCHMARKING APPLICATION

The social networking application and its three request types are representative of microservices applications as they capture the intricacies of microservices architecture. The social networking application is widely used by prior works as a representative microservices application workload, including those on bottleneck localization [20, 43] and in characterization studies by cloud providers such as Microsoft [5] and Google [3]. The application services both read (home and user) and write requests (compose). The application has Nginx, MongoDB, Memcached, Redis, RabbitMQ, synchronous and asynchronous calls, and logic servers implemented using the industry-standard Thrift framework [8]. The call graphs of these three requests are different. The call graph of compose, home, and user requests consists of 28, 7, and 9 nodes, respectively. The nodes and the edges of these graphs vary significantly, too. There is also variance in system characteristics. For example, the compose request call graph has queueing and asynchronous requests, whereas the user request has multiple levels of caching.

B GAMMA IMPLEMENTATION DETAILS

The anomaly detector is a binary classifier consisting of a 2-layer feed-forward and a softmax. The 2-layer feed-forward network processes and refines the input representations. The refined representations are passed to the softmax layer, which provides the likelihood of traces being anomalous. The motivation for choosing this design for the anomaly detector is recently published works for multi-task learning, such as AdaShare [52] and DynaShare [45], which use a hard share framework for multi-task learning. The bottleneck localizer is a mixture of expert (MoE) binary classifiers with one binary classifier for each microservice. The binary classifier consists of a 2-layer feed-forward network and a softmax layer. The input to each classifier is the same intermediate representations, but each expert is trained to capture the unique characteristics of the assigned microservice. The choice of a mixture of experts for this stage is motivated/influenced by recent works on MoE, such as SPNet [17] and MEID [29], which use an MoE setup for a multi-label classification task. The dimensions of the networks in both the anomaly detector and the bottleneck localizer depend on the number of nodes (microservices) in the call graph.

C GAMMA HYPERPARAMETERS

α is the hyperparameter that decides the contribution of the losses of anomaly detection and bottleneck localizer in the joint loss function. A higher value of α would imply the model focuses on making fewer errors during anomaly detection as the penalty for such errors is high. On the other hand, a lower value of α would mean the model focuses on making fewer errors during the bottleneck localization.

For all three call graphs (compose, user, home), we start with α =0.5, providing equal weights for both tasks. We change α in steps of 0.05 and observe the training and validation loss. We observe that values 0.2, 0.25, and 0.35 provide the best results for compose, user, and home requests. This shows that penalizing errors for the harder task of localizing bottlenecks provides the best overall result.

The parameter τ represents the window size the input vector is split into. Ideally, an optimal window size would be one that is (a) large enough to have enough data points in them to capture temporal patterns since this allows us to see relative changes in latency and resource usage in the window, and (b) small enough that the data points in the window are not too noisy and are relatively cohesive. Hence, there is scope for tuning the window size. We looked at the validation loss for separate models trained for window sizes starting from 0.5 seconds to 5 seconds, at 0.5 seconds intervals. We found that a window size of 1 second gave the lowest validation loss.

D DATASET

The train, validation, and test split was 70%, 10%, and 20%, respectively, for all the graphs (compose, home, and user). So, overall, around 28M, 4M, and 8M traces were used for training, validation, and testing, respectively.

The proportion of normal and anomalous traces in the dataset is 63% and 37% (the proportion of normal traces in prior works ranges from 2% to 50%). The percentage distribution of the number of bottlenecks (N) among all the windows is given in Table 4.

N	Proportion (%)
0	63
1	5
2	3
3	5
4	4
5	13
6	3
10	1
12	3

Table 4: Distribution of bottlenecks in the dataset.

E GAMMA DESIGN

In this section, we describe some of the components of GAMMA.

E.1 Dilated Causal Convolution

The dilated causal convolution for a feature vector $\mathbb{M}^{\eta x \tau}$ is

$$\mathbb{O}(\eta, k, q) = \sum_{\eta} \sum_{\tau+\delta \cdot s=q} \mathbb{M}(\eta, \tau) \times Weight(k, s), \tag{2}$$

where $\mathbb{M}(\eta, \tau)$ is the input tensor, $\mathbb{O}(\eta, k, q)$ are the multi-channel output embeddings, $k \times q$ represent the Convolution filters, δ is the expansion factor, and $Weight(k, s)$ is the filter size for η output channels. Self-attention is then applied on the input embedding tensor $\mathbb{M}^{\eta x \tau}$, as:

$$Attn(\mathbb{M}) = softmax\left(\frac{W_q\mathbb{M} \cdot (W_k\mathbb{O})^T}{\sqrt{d}} W_v\mathbb{M}\right), \tag{3}$$

where W_q, W_k and, W_v are trainable hyperparameters and d is an empirical scaling factor. This phase outputs multi-channel embeddings with latent representation $\mathbb{O}^{\eta \times k \times q}$.

E.2 Graph Attention

Dynamic edge weights, integral to the attention mechanism, are formulated as per Equation (4), ensuring an understanding of microservice interactions.

$$\omega_{a,b} = \frac{exp(LeakyReLU(v^T[W\mathbb{O}_a^\epsilon || W\mathbb{O}_b^\epsilon]))}{\sum_{k \in \mathbb{N}_a} exp(LeakyReLU(v^T[W\mathbb{O}_a^\epsilon || W\mathbb{O}_k^\epsilon]))}, \tag{4}$$

where $\omega_{a,b}$ is the computed weight of edge $\vec{e}_{a,b}$, $\mathbb{N}_a$ is the set of neighbor nodes for node a; $\mathbb{O}_a^\epsilon$ is the intermediate node representation of node a; $W \in \mathbb{R}^{E^G \times E}$ and $v \in \mathbb{R}^{E^G}$ are trainable parameters. E^G is the shape of the output representation. The impact of all the neighboring nodes b on node a is calculated as follows:

$$\hat{O}_a^\epsilon = ReLU(\sum_{b \in \mathbb{N}_a} \omega_{a,b} W\mathbb{O}_b^\epsilon) \tag{5}$$

Global Attention Pooling [11] is then performed on the node representations to generate dependency-aware embeddings $\mathbb{O}^\zeta$.