RESCAPE: A Resource Estimation System for Microservices with Graph Neural Network and Profile Engine

Jinghao Wang*, Guangzu Wang†, Tianyu Wo*, Xu Wang*, Renyu Yang*
*School of Software, Beihang University, †School of Computer Science and Engineering, Beihang University
{wang_jinghao, wang_guangzu, woty, xuwang, renyuyang}@buaa.edu.cn

***Abstract*—Microservice architecture has become a prevalent paradigm for constructing scalable and flexible cloud-native applications by leveraging the abundant resources of the cloud. However, the topological complexity of microservices poses significant challenges to resource management frameworks that rely on container orchestration. It is paramount to optimize resource utilization within cloud computing clusters while reducing operational costs for service providers. To this end, we present RESCAPE, a framework designed to effectively predict the resource demands of variable microservice workloads. It is instrumental for downstream optimization tasks, particularly heterogeneous resource scheduling, aiming to enhance resource utilization and efficiency. Experiments, based on open-source microservice benchmarks such as DeathStarBench and HPC-AI500, demonstrate an average absolute percentage error (MAPE) of 7.9% when forecasting resource needs for the subsequent timestamp. This indicates an adequate precision for microservices' resource estimation.**

***Index Terms*—Microservice, Resource Estimation, GNN**

I. Introduction

In recent years, cloud computing has been increasingly utilized in various scenarios, such as the Internet of Things (IoT) and smart cities, due to the abundant resources it offers. Microservice architecture has also been introduced to address the growing complexity of program size and business logic in these contexts [1]. Concurrently, the demand for speed and agility in the software development industry has catalyzed the shift from monolithic to microservice architecture applications [2].

Microservices are lightweight, self-contained applications. As depicted in Figure 1, in a microservices architecture, the original monolithic application is decomposed into several lightweight, distributed services that operate collaboratively. Each service provides a specific function and operates independently from the others [3], allowing for the use of different programming languages and independent deployment through a container orchestration system.

Despite the convenience of building and deploying applications, the microservices architecture complicates resource management as each component requests resources independently in large-scale cloud computing clusters. Resource scheduling is typically augmented by predicting future demand to prepare for allocation, thereby preventing service Quality of Service (QoS) degradation. Meanwhile, communications

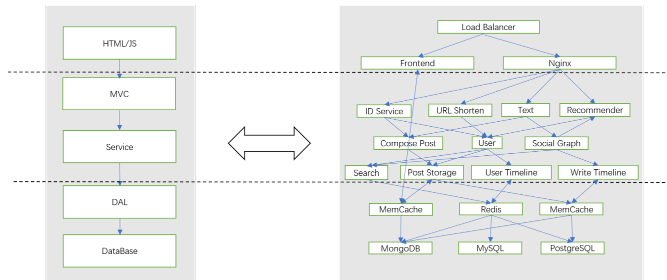


Figure 1: Monolithic vs. Microservices Architecture

between services via API endpoints in real applications introduce non-trivial topological dependencies and blocking effects [4], complicating resource estimation compared to traditional monolithic applications [5].

We present RESCAPE, a data-driven, deep learning prediction system for microservices resource estimation. It performs online profiling for runtime metrics and further extracts graphical features to adapt to variations in real workloads. The structured data is fed into a deep model offline to predict the resource requirements for each service. The framework can be implemented as an API Gateway function or a plugin for the container orchestration engine in production so that the demand for resources can be anticipated upon the arrival of a user request [6]. Then, downstream optimization tasks like resource scheduling can utilize the prediction to make better solutions, enhancing resource utilization and lowering service operating costs of a cloud computing cluster. We evaluated RESCAPE on the Deathstar [7] and HPC-AI500 [8] microservices benchmarks. Experimental results show that RESCAPE can support heterogeneous microservice resource metrics collection and prediction with a 7.9% average absolute percentage error in predicting the resources required at the next timestamp.

In general, we make the following contributions:

- We propose a system for predicting the microservices resource requirements through online profiling and offline training, which is generalizable with varying workloads taken into consideration. (Section III-C1);
- We develop an algorithm to extract microservices features into graph-structured data with profiling metrics and traces. (Section III-C2);

©2024 IEEE

- We design a GNN-RNN combined deep learning model, which not only perceives historical resource usage but is also aware of spatial relationships in the microservices topology. (Section III-C3).

II. Background and Motivation

A. Resource Prediction for Runtime Application

Compared to running a monolithic application on a single machine, resource scheduling and workload assignment are critical when operating a large number of workloads in clusters. Effective resource allocation in such environments can significantly increase overall resource utilization [9], [10]. In this context, resource prediction is an industry standard that aids scheduling by formulating a more concrete resource allocation problem.

One common approach to predicting runtime performance involves static code analysis, which includes parsing source code or binary compilation files and constructing internal representations such as Abstract Syntax Trees (ASTs) [11] and Directed Acyclic Graphs (DAGs) [12] to identify critical paths and instruction components. This method then predicts performance metrics based on the program's control flow and architectural parameters of the target machine. Another approach relies on dynamic runtime metrics to forecast resource usage. This involves a system with integrated workload profiling, data collection and processing, model training, and prediction [4], [13], [14]. It deploys workloads in a cluster, extracts performance metrics such as response time, throughput, and resource utilization from application logs and hardware monitoring, and designs a model to learn from historical resource utilization data through feature engineering [15]. The system operates on the cluster management node to assist with resource allocation decisions.

B. MicroService Resource Estimation

Resource estimation becomes more complex with microservices due to the increased requirements for guaranteed Quality of Service (QoS) [15], which necessitates finer-grained resource management, typically at the service pod level. Additionally, microservices communicate with each other to respond to fluctuating request flows, introducing call dependencies and network effects.

Static resource estimation methods are limited in this context because microservices have loosely coupled codebases, and the number of target machines can be vast. The predominant approach for estimating microservice runtime resource usage has shifted towards dynamic analysis systems with data-driven models. However, these systems face challenges as workloads vary significantly, and a single deep model may not generalize well across all application scenarios. Moreover, many current deep models for resource estimation fail to consider a crucial feature: the topology of microservice calls.

C. Motivation

Our design of RESCAPE is rooted in the existing dynamic analysis approach, which we have expanded to construct a versatile pipeline for resource estimation of microservice workloads. The system is capable of adapting to diverse workloads by profiling and gathering data at the pod level, without requiring prior knowledge of the microservices' implementation specifics. Concurrently, it employs Deep Neural Networks (DNNs) for prediction, alleviating the need for sophisticated feature engineering in previous work and addressing the microservice topology within the deep model training and prediction processes.Designed for easy integration into existing container orchestration systems, RESCAPE enables real-time resource estimation and allocation, enhancing the efficiency and cost-effectiveness of microservice deployments.

III. Overview of RESCAPE

A. Key Idea

RESCAPE is a resource estimation system designed to enable rapid adaptation to any microservice workloads. This is primarily achieved by implementing an online profiling and offline data processing pipeline to train a deep model for prediction. RESCAPE constructs a deep model that integrates the Graph Attention Mechanism (GAT) into traditional Recurrent Neural Networks (RNNs) for better capturing the topological features between microservices within a time series prediction framework.

B. RESCAPE *Architecture*

Figure 2 provides a high-level overview of the RESCAPE architecture, inspired by Seer[13]. RESCAPE operates by deploying microservices workloads onto a container orchestration engine. It then initiates simulated clients to generate requests directed at the exposed endpoints of the microservices. Utilizing a distributed tracing system, RESCAPE constructs the call graph topology of the microservices. Concurrently, it gathers time series data from the monitoring metrics server for offline training of the deep learning model. In the production environment, the model operates as a service within the container orchestration engine, ready to receive user requests and perform online inference in real-time.

C. *Key Techniques of* RESCAPE

1) Workload Profiling: As discussed in Section II, our work targets microservices deployed in container orchestration engines, and we note the variability that real workloads have. Therefore, the rapid deployment of new workloads is essential for online profiling.

RESCAPE assumes that the code repository for a specific workload has been provisioned, and the Dockerfile for image building is available, which is necessary for running any workload on a container orchestration system. To achieve this, we allocate a custom resource in Kubernetes, which orchestrates the creation and management of Kaniko[16] pods for image building and pushing. We then deploy the workload by submitting a YAML file to the Kubernetes API Server. The API Server receives the request and stores the resource contents in etcd[17], a distributed key-value store for reliable data storage in distributed systems. After storing the resource

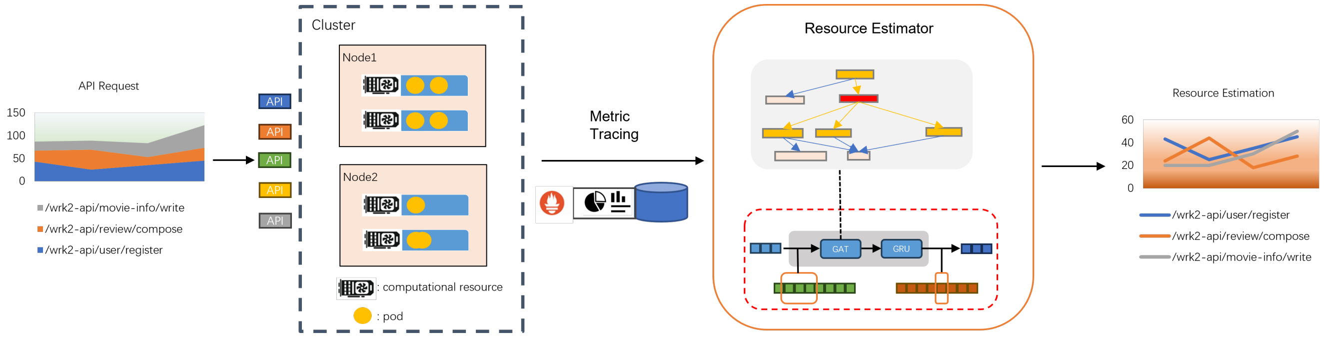


Figure 2: RESCAPE Architecture

contents, the API Server creates the desired number of pod instances and schedules them onto specific nodes for execution. Once the binding is successful, the image is pulled from the repository, and each service operates within a container.

The simulated user clients receive lists of exposed microservices endpoints and employ a randomized level of concurrency to send requests over a specified period. Monitoring metrics data are periodically extracted using Prometheus[18] and stored in a time-series database. At the end of the request simulation, we fetch the performance metrics data for each microservice through PromQL query and aggregation operations within the database.

2) Microservice Call Graph Modelling: The topology of a microservice application, denoted as G, can be described using a set of nodes V, representing the microservices, and edges E, representing the communication between them.

$$G = (V, E) \tag{1}$$

A microservice application consists of a number n of microservices, each of which can be represented by a unique identity string. Let the set S of strings be composed of these identity strings. This set S can be mapped to the domain of natural numbers $\mathbb{N}$ through a function f, thereby defining the set of microservice nodes V.

$$\begin{aligned} S &\xrightarrow{f} V \\ V &= \{i \mid i \in \mathbb{N}\} \end{aligned} \tag{2}$$

The CPU and GPU resource utilization vector of the aforementioned microservice application is defined as the concatenation of sequentially spliced resource utilization values c_t^i and g_t^i for each microservice i at timestamp t, indexed by a natural number, where n is the cardinality of the set of microservice nodes V.

$$\begin{aligned} c_t^G &= \{c_t^1, c_t^2, ..., c_t^i, ..., c_t^n\} \\ g_t^G &= \{g_t^1, g_t^2, ..., g_t^i, ..., g_t^n\} \end{aligned} \tag{3}$$

The topology of a microservice application is derived from the chain of invocation relationships along the traces. When a service request (Request) arrives at the microservice's service gateway (API Gateway), the gateway distributes the request

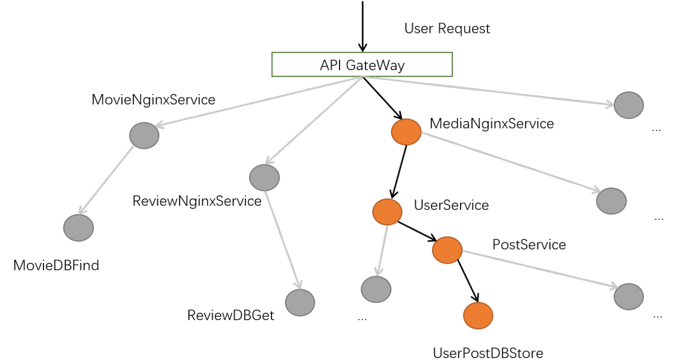


Figure 3: Media Microservice Trace

to the downstream service, which goes on to request other services.

.

As shown in Figure 3, using a portion of a social media microservice application as an example, when a user posts a personal tweet on social media, the service request arrives at the service gateway. The downstream services (MediaNginxService) first obtain user ID data from the user information service (UserService), then package the data and send it to the push service (PostService), and finally store the user's post information in the database (UserPostDBStore). The chain of calls is formed and marked by orange highlighted nodes.

For microservices that engage in upstream and downstream calling relationships, an ordered pair representing a call from an upstream to a downstream microservice is added to the set of edges E, denoted as $calls$, to signify the presence of a calling relationship.

$$E = \{(i, j) \mid i, j \in V \wedge i \ calls \ j\} \tag{4}$$

Given the sequential nature of microservice invocations, it is evident that the variation in resource utilization among microservices is not only influenced by their own historical resource utilization patterns, but also by the patterns exhibited by the microservices that are invoked before and after them. Therefore, it is crucial to incorporate the graph structure information of the microservices into the design of the deep learning model for resource prediction.

3) GAT-RNN Combined Deep Estimation: The task of resource prediction involves learning a mapping function f that maps the historical resource usage sequence $W_{history}$ of a microservice onto its future resource usage sequence W_{future}.

$$W_{history} \xrightarrow{f} W_{future} \tag{5}$$

where $W_{history}$ and W_{future} are sets of resource utilization vectors characterizing microservice applications with a certain topology at a series of timestamps.

$$\begin{aligned} W_{history} &= \{X^G_{t-T+1}, X^G_{t-T+2}, \ldots, X^G_{t-1}, X^G_t\} \\ W_{future} &= \{X^G_{t+1}, X^G_{t+2}, \ldots, X^G_{t+T'-1}, X^G_{t+T'}\} \\ X^G_t &= \{c^G_t, g^G_t\} \end{aligned} \tag{6}$$

Let X^G_t denote the resource utilization vector of the microservice application with topology G at time t. The vector c^G_t represents the CPU resource utilization, while g^G_t signifies the GPU resource utilization. T represents the length of the historical time series used for prediction, and T' denotes the length of the predicted future time series.

In this context, Recurrent Neural Networks (RNNs) [19] are commonly utilized to address problems where both inputs and outputs are consecutive sequences, with the sequences exhibiting variable lengths, such as those encountered in time series analysis.

However, it is known that RNNs have a limited ability to retain information over extended periods and struggle with processing extremely long input sequences [20]. This limitation arises because the output at each time step is contingent upon the input from the preceding time step, which can lead to a loss of information as the sequence length increases.

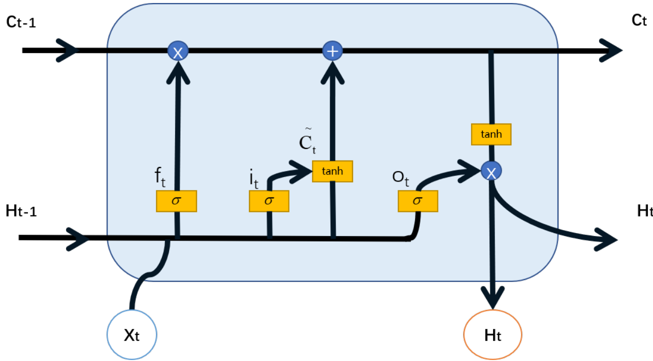


Figure 4: LSTM Neural Cell

Therefore, we adopt variants of RNNs, such as the Long Short-Term Memory Network (LSTM) [20] and the Gated Recurrent Unit (GRU) [21], to address the issue of long-term dependencies in input sequences. Unlike standard RNNs, which have a very simple structure within each neural module (or block), LSTMs and GRUs offer more sophisticated mechanisms for memory retention. As depicted in Figure 4, the LSTM design incorporates "neural cells" and intracellular functional structures that are inspired by real-life tasks, resulting in a more complex structure within the blocks. The GRU simplifies the neural architecture by eliminating the separate cell state and merging the forgetting and input gates. However, both models facilitate the preservation and flow of information across a greater number of timesteps compared to standard RNNs.

$$\begin{aligned} f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) \\ i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= tanh(W_C[h_{t-1}, x_t] + b_C) \\ C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \\ o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\ h_t &= o_t * tanh(C_t) \end{aligned} \tag{7}$$

Assuming that the time series of microservice resource usage is predicted using RNN variant models, the input vector is constructed by aggregating resource utilization data across all microservices. This process effectively treats the recurrent network's propagation as a series of linear regressions, enabling global resource utilization at each moment to contribute to the computation of the next hidden state.

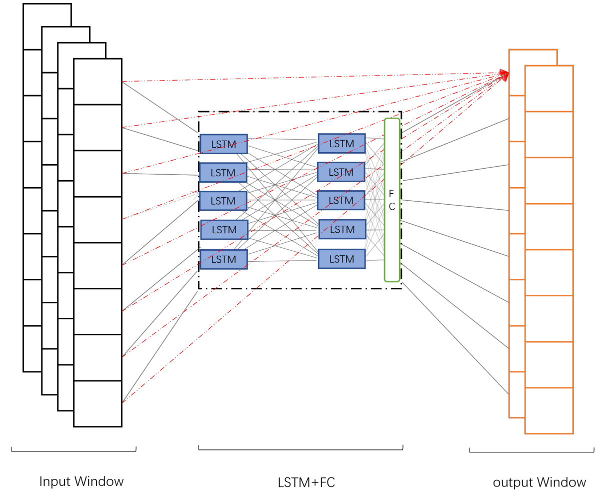


Figure 5: Tensors Propagation Abstract

As depicted in Figure 5, the solid gray lines depict the actual propagation of vectors, whereas the solid red lines represent the scenario where all global microservice features contribute to computing the predicted resource utilization of a particular microservice.

However, the correlation of resource utilization among microservices is primarily driven by upstream and downstream invocations, rather than global interactions. Extracting and utilizing the rich correlational information encoded in the microservices' topological graph structure is crucial for accurate hidden state computation and propagation. Incorporating an attention mechanism during the processing of resource utilization vectors allows the model to focus on relevant upstream and downstream microservices, thereby enhancing prediction accuracy.

To effectively process the graph-structured information of microservices, we introduce the Graph Attention Network

(GAT) [22]. GAT is composed of multiple Graph Attention Layers (GALs), each designed to enhance the expressiveness of the node representations by applying a weight matrix W to each node. Given the input feature values of the nodes as $\vec{h}$, where N is the number of nodes and F is the dimensionality of the node features, after propagating through the graph attention layer, a new feature vector $\vec{h'}$ is produced. This vector retains the same number of nodes N, but the dimensionality of the node features is transformed to $F\prime$.

$$\vec{h} = \{\vec{h_1}, \ \ \vec{h_2}, \dots, \vec{h_N}\}, \ \vec{h_i} \ \in \ R^F \\ \vec{h'} = \{\vec{h'}_1, \ \ \vec{h'}_2, ..., \vec{h'}_N\}, \ \vec{h'}_i \ \in \ R^{F'} \tag{8}$$

eij represents the importance of node i to node j. Theoretically the weights from any node in the graph to the central node can be computed, but to simplify the computation, GAT restricts the nodes to the node itself and its one-hop neighbors.

$$e_{ij} = LeakyReLu(a^T[W\vec{hi}, W\vec{hj}]) \\ \alpha_{ij} = softmax(e_{ij}) = \frac{exp(e_{ij})}{\sum_{j \in N_i} exp(e_{ij})} \\ \vec{h'}_i = \sigma(\sum_{j \in N_i} \alpha_{ij}\vec{h_j}) \tag{9}$$

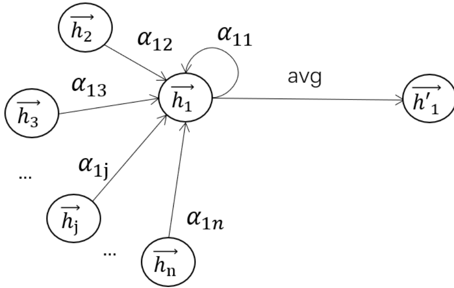


Figure 6: Graph Attention Network

To improve the generalization ability of the attention mechanism, GAT can use a multi-head attention layer consisting of K groups of mutually independent single attention layers. The formula of the final attention layer is shown below, where α_{ij}^k denotes the weight coefficient computed by the kth group of attention mechanisms and W^k is the weight coefficient of the kth module.

$$\vec{h'}_i = \sigma(\frac{1}{K}\sum_{k=1}^{K}\sum_{j \in N_i} \alpha_{ij}^k W^k \vec{h_j}) \tag{10}$$

As we describe microservices as nodes and callups between upstream and downstream microservices as edges for the time series prediction task, the time series processing of each microservice remains independent when using RNN models. However, there is an execution relationship between the microservices; thus, the time series information of the neighbor nodes in the graph is valuable for predicting its own sequence. Consequently, GAT aggregates features from

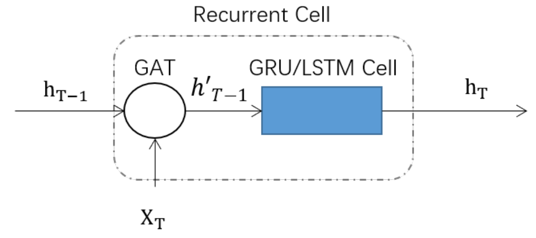


Figure 7: GAT-RNN cell

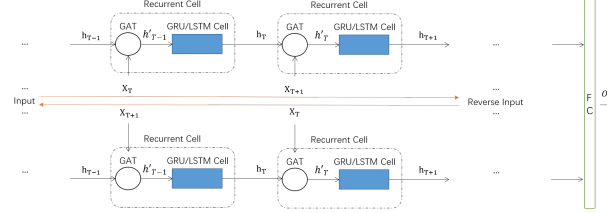


Figure 8: GAT-RNN Network

neighboring nodes to the central node, learning feature representations using information from the local graph.

As depicted in Figure 7, we integrate GAT into the recurrent basic block (Recurrent Cell) of the RNN to form a new basic block (GRU/LSTM Cell). This enables the model to learn the time series information of each microservice node while considering the graph structure among the microservices. For each basic block, the hidden state h_{T-1} of the previous timestamp is forwarded into the GAT as a feature vector. Based on the graph structure information G of the microservice workloads, the input feature vector is processed with the multi-head attention mechanism to obtain the enhanced feature vector h'_{T-1} with graph structure information. The enhanced feature vector is then input into the GRU/LSTM Cell to continue propagation or for resource prediction.

As shown in Fig. 8, we stack two layer of recurrent cell to construct a bi-directional recurrent neural network, with the same set of time series data respectively entering two layers of Recurrent Cell forward and reversely. The hidden state of the last cell in the two layers are concatenated to input into a fully connected layer, which outputs the final prediction result.

IV. Evaluation Setup

A. Clusters

We build RESCAPE on our cloud computing clusters based on Kubernetes 1.22, which consists of 4 worker nodes connected through an 800 Gbps network. Each worker node runs the Ubuntu 22.04 operating system, has an 80-core Intel Xeon Platinum 8163 CPU at 2.50GHz, 4 x 32GB NVIDIA Tesla V100 GPUs, and 256GB of DDR4 RAM. We deploy the necessary plugins and dependencies, such as Prometheus, DCGM-Exporter, and Jaeger Operator, for collecting runtime metrics and distributed tracing.

B. Data

We evaluate RESCAPE using the open-source microservice benchmarks DeathStarBench and HPC AI500 [8], which encompass services that require heterogeneous computational resources. We deploy the selected benchmark assembly by submitting a YAML file to the API Server via kubectl. The constructed clients send requests with a randomized concurrency level (80-220) every minute for 12 hours to the API endpoints exposed by the microservices. After all the requests are processed, the performance metrics data for each microservice is collected in the format shown in Table I.

Table I: MS_Metric

Metric Name	Value	Description
$timestamp$	0	Sampled timestamps
ms_name	99f2e7b501f50db9	Raw id of the ms
ms_id	0	Numbered id of the ms
$cpu_utilization$	0.12992	CPU utilization for ms
$gpu_utilization$	0	GPU utilization for ms

The timestamps range from 0 to 1440 (12 hours of data collected, sampled at 30-second intervals). During the collection process, the string value ms_name of each original microservice is mapped to a natural number ms_id, enabling the sequential numbering of microservices.

To analyze the call dependencies between microservices, we query and aggregate the microservice traces to obtain the structural data of each microservice call graph, as shown in Table II.

Table II: MS_CallGraph

Metric Name	Value	Description
$timestamp$	163	Sampled timestamps
$trace_id$	015101cd1591939	Callup raw id
um	35114acfb54c54fb	Upstream ms raw id
dm	b65fdc9bfef6b497	Downstream ms raw id
um_id	5	Numbered id of the um
dm_id	29	Numbered id of the dm

We constructed the edge set of microservices based on the upstream and downstream microservice names (um, dm). Using the data from the microservice node set and edge set, we revised the topological graph structure.

C. Models

We compare the following models for the resource estimation task in RESCAPE:

FC is a baseline regression model used for resource estimation, similar to some related works.

GRU & LSTM are RNN-based models. This test group indicates whether RNN models can perceive the characteristics of the microservice resource utilization time-series data, and which RNN class model is more capable of handling long-term dependencies.

GAT-GRU & GAT-LSTM is our design to introduce the GAT into RNN models to process the hidden state. It is expected that the combined models perform better estimation based on the perception of graphical call-up information.

We use a consistent hyperparameter setting for the models, with 4 hidden layers and 48 hidden units, to avoid affecting the prediction accuracy due to the depth of the network. Meanwhile, the size of the hidden layers is set to a lower value to avoid overfitting and speed up the training process. The GAT combined with the RNN models has identical parameters of 4 multi-heads. Additionally, we introduce a sliding window for the input and output sequences. A sequence of resource utilization for the next 3 timestamps is predicted using a sequence of 10 historical timestamps as input to the model. The windows of the history and future are shifted by one simultaneously, continuing to extract inputs in the history window to predict the resource utilization in the future window. This strategy effectively enriches the predicted historical information, delays overfitting during training, and enhances the generalization ability of the model.

D. Metrics

We use three different metrics to assess the difference between the actual and predicted values of resource utilization. The actual value of resource utilization is denoted as $\mathbf{y} = \{y_1, y_2, \ldots, y_n\}$, and the predicted value of resource utilization is expressed as $\hat{\mathbf{y}} = \{\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n\}$, where n is the number of samples.

a) Mean Absolute Error (MAE): represents the average of the absolute errors between the predicted values and the true values. This metric provides an intuitive measure of the error magnitude.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\widehat{y_i} - y_i| \tag{11}$$

b) Mean Absolute Percentage Error (MAPE): indicates the average of the absolute errors of the predicted values from the true values as a percentage. This metric is sensitive to relative errors and is suitable for variables with large magnitudes of differences.

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}|\frac{\widehat{y_i} - y_i}{y_i}| * 100\% \tag{12}$$

c) Mean Squared Error (MSE): represents the average of the squared errors between the predicted values and the true values. Due to the squaring of errors, this metric is suitable for normalized data values and is particularly sensitive to larger errors.

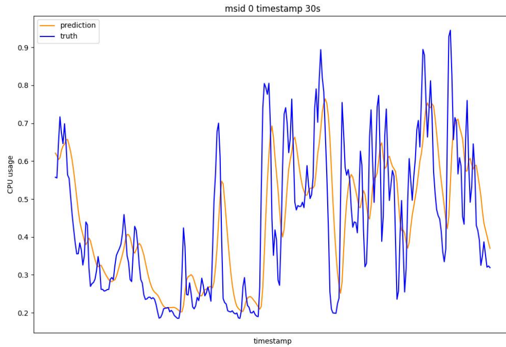
$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\widehat{y_i} - y_i)^2 \tag{13}$$

V. Experimental Results

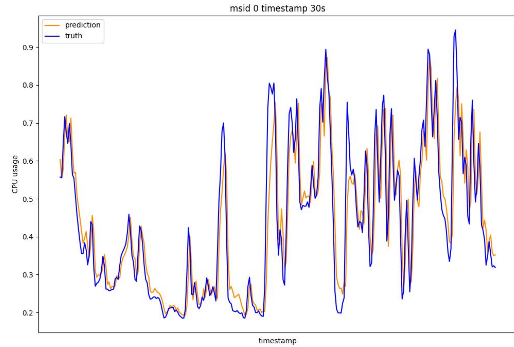
We train the models for 50 epochs with a batch size of 48 on an Intel Core i7-8750H CPU @ 2.20GHz and an NVIDIA GTX2060 GPU, running Ubuntu 22.04 LTS. The

Table III: Results of Experiment

Timestamps	MAE			MAPE			MSE		
	30s	60s	90s	30s	60s	90s	30s	60s	90s
FC	10.110	9.6000	9.8800	132.03	101.60	82.450	1.6500	1.5900	1.7200
GRU	**5.9130**	**6.7920**	**7.3490**	**15.717**	**18.626**	**21.625**	**0.6750**	**0.8860**	**1.0280**
LSTM	6.5940	7.1440	7.5770	22.235	22.260	21.543	0.8040	0.9420	1.0570
GAT-GRU	**3.9030**	**5.1600**	**5.9530**	**7.9420**	**11.700**	**15.049**	**0.3320**	**0.5530**	**0.7190**
GAT-LSTM	4.1320	5.2830	6.0460	15.831	16.297	17.038	0.3580	0.5680	0.7290



(a) GRU



(b) GAT-GRU

Figure 9: Comparison of the Truth and Prediction

Adam optimizer is used with a base learning rate of 0.001. After the training process is completed, the models (FC, GRU, LSTM, GAT-GRU, and GAT-LSTM) are evaluated on the designated test set, with the results for each metric shown in Table III.

The experimental results clearly demonstrate that the FC model exhibits the poorest performance across all evaluation metrics. Transitioning from the FC model to the RNN models results in a noticeable improvement in prediction accuracy. To illustrate this, the prediction and actual CPU utilization curves for the microservice with msid 0, using the GRU model, are plotted as shown in Figure 9a.

Introducing the GAT model to process the hidden states of the RNN models further enhances prediction accuracy. This indicates that the graph attention mechanism effectively extracts graph structure information from microservices, aiding in future resource utilization predictions. The prediction and actual CPU utilization curves for the microservice with msid 0 using the GAT-GRU model are shown in Figure 9b.

Comparative analysis between the two RNN models reveals that GRU provides better predictions for subsequent timestamps, achieving a mean absolute percentage error (MAPE) of 7.9%. Additionally, GRU has fewer parameters compared to LSTM, making GAT-GRU a more suitable model for microservice resource estimation.

VI. Related Work

The runtime resource estimation for programs typically employs two dominant techniques for building prediction models: static code analysis and dynamic runtime performance analysis. The former extracts features from the static source code, while the latter, which RESCAPE adopts, examines dynamic runtime performance metrics.

Dynamic analysis primarily involves several steps: profiling workloads, collecting and processing data, and training and evaluating models. This approach is supported by the work of J. Rahman and P. Lama [15]. Using the SockShop benchmark on Kubernetes [23], collecting metrics data by sending simulated service requests to various microservices. However, their work was limited to constructing simple regression and DNN models to predict end-to-end latency for microservices, failing to account for the topological features of microservices. Moreover, generalizing these prediction models to other microservice workloads in production environments is challenging.

In response, other studies have integrated the entire process into a unified system and introduced deep models to better address the spatial and temporal characteristics of microservice runtime metrics data.

Seer [13] utilizes distributed tracing to incorporate spatial features of microservices into input metric sequences. Leveraging this tracing for graph information and prediction with a CNN-RNN model shows promise. However, passive tracing collection may lead to incomplete graphs, and the sequential presentation of topology could lack full context due to small CNN filters. In contrast, RESCAPE actively queries all exposed microservices APIs during workload profiling to complete the topology and utilizes graph neural networks, which are well-suited for processing graph-structured data.

DeepRest[14] build the microservice resource estimation as a function of api gateway, and fuse the callup graph with a feature extractor that labels every possible execution path.

It introduce a feasible algorithm for capture graph structure information. However, constructing such an extensive feature space is unnecessary when only a subset of execution paths is relevant for API microservice endpoints. In this regard, RESCAPE builds the graph based on actual flow paths and dynamically scales the feature space according to the nodes and edges using graph neural networks.

Nodens [4] utilizes bandwidth occupation to construct the microservice topology, allowing for refined modeling of blocking effects and queuing workload distribution forecasts via regression. Though it enhances metric monitoring, manual feature engineering is still required for microservice call graph dependencies. In contrast, RESCAPE automates this feature extraction and learning from aspects like queue occupancy and blocking effects using a neural network.

VII. Conclusion

We present RESCAPE, a robust data-driven system for precise microservice resource estimation. It adapts to variable workloads using online profiling and offline training, leveraging temporal and graphical profiling metrics to enhance prediction accuracy. RESCAPE can be deployed as an API Gateway function or as a container orchestration engine plugin, optimizing resource allocation in real time.

Our contributions include integrating a predictive system for microservice resource requirements, leveraging online profiling and offline training to adapt to varying workloads. We introduce algorithms to convert microservice features into structured data, improving prediction accuracy. Additionally, we present a GNN-RNN model that analyzes historical usage and spatial relationships between services, enhancing resource management in cloud ecosystems. Evaluated using the Deathstar and HPC-AI500 benchmarks, RESCAPE achieved a 7.9% average absolute percentage error in forecasting resource needs. These findings demonstrate the system's potential to significantly improve resource utilization and reduce operational costs in cloud computing clusters.

Acknowledgment

This work is supported in part by National Key R&D Program of China (2022YFB4502003), by the Fundamental Research Funds for the Central Universities (501QYJC2023121001). For any correspondence, please refer to Renyu Yang (renyuyang@buaa.edu.cn).

References

[1] L. Chen, Y. Xu, Z. Lu, J. Wu, K. Gai, P. C. K. Hung, and M. Qiu, "Iot microservice deployment in edge-cloud hybrid environment using reinforcement learning," *IEEE Internet of Things Journal*, vol. 8, pp. 12 610–12 622, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:226579924

[2] X. He, T. Wang, L. Liu, J. Li, Z. Su, Y. jun Guo, Z. Tu, H. Xu, and Z. Wang, "Rescureservice: A benchmark microservice system for the research of mobile edge and cloud computing," *ArXiv*, vol. abs/2212.11758, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:254974305

[3] P. Jamshidi, C. Pahl, N. das Chagas Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Softw.*, vol. 35, pp. 24–35, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:25437582

[4] J. Shi, H. Zhang, Z. Tong, Q. Chen, K. Fu, and M. Guo, "Nodens: Enabling resource efficient and fast qos recovery of dynamic microservice applications in datacenters," in *USENIX Annual Technical Conference*, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259859065

[5] L. Bao, C. Q. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, pp. 2114–2129, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:86671914

[6] X. He, Z. Tu, X. Xu, and Z. Wang, "Re-deploying microservices in edge and cloud environment for the optimization of user-perceived service quality," in *International Conference on Service Oriented Computing*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:204945669

[7] DeathStarBench. https://github.com/delimitrou/DeathStarBench.

[8] HPC AI500. https://www.benchcouncil.org/aibench/hpcai500/index.html.

[9] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *USENIX Annual Technical Conference*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:58014231

[10] G. Fan, L. Chen, H. Yu, and W. Qi, "Multi-objective optimization of container-based microservice scheduling in edge computing," *Comput. Sci. Inf. Syst.*, vol. 18, pp. 23–42, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:229654000

[11] K. Meng and B. Norris, "Mira: A framework for static performance analysis," *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 103–113, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:3440922

[12] V. K and M. Purnaprajna, "Performance estimation on heterogeneous systems: Making the most of static analysis," *2020 7th International Conference on Signal Processing and Integrated Networks (SPIN)*, pp. 435–440, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:216043125

[13] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:102347800

[14] K.-H. Chow, U. Deshpande, S. Seshadri, and L. Liu, "Deeprest: deep resource estimation for interactive microservices," *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247765661

[15] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 200–210, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:96438130

[16] kaniko. https://github.com/GoogleContainerTools/kaniko.

[17] etcd. https://etcd.io.

[18] Prometheus. https://prometheus.io/.

[19] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei, "Robust anomaly detection for multivariate time series through stochastic recurrent neural network," *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:196175745

[20] X. Shi, Z. Chen, H. Wang, D. Y. Yeung, W.-K. Wong, and W. chun Woo, "Convolutional lstm network: A machine learning approach for precipitation nowcasting," in *Neural Information Processing Systems*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:6352419

[21] J. Chung, Çaglar Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *ArXiv*, vol. abs/1412.3555, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:5201925

[22] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio', and Y. Bengio, "Graph attention networks," *ArXiv*, vol. abs/1710.10903, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:3292002

[23] SockShop. https://microservices-demo.github.io.