DeepRest: Deep Resource Estimation for Interactive Microservices

Anonymous Author(s)
Submission Id: 120

Abstract

Interactive microservices expose API endpoints to be invoked by users. For such applications, precisely estimating the resources required to serve specific API traffic is challenging. This is because an API request can interact with different components and consume different resources for each component. The notion of API traffic is vital to application owners since the API endpoints often reflect business logic, e.g., a customer transaction. The existing systems that simply rely on the historical resource utilization are not API-aware and thus cannot estimate the resource requirement accurately. This paper presents DeepRest, a deep learning-driven resource estimation system. DeepRest formulates resource estimation as a function of API traffic and learns the causality between user interactions and resource utilization directly in a production environment. Our evaluation shows that DeepRest can estimate resource requirements with over 90% accuracy, even if the API traffic to be estimated has never been observed (e.g., 3× more users than ever or unseen traffic shape). We further apply resource estimation for application sanity checks. DeepRest identifies system anomalies by verifying whether the resource utilization is justifiable by how the application is being used. Our evaluation shows that DeepRest can successfully identify two major cybersecurity threats: ransomware and cryptojacking attacks.

1 Introduction

Microservices have caused a paradigm shift, with large content platforms and cloud providers, such as Netflix, Twitter, Uber, IBM, and Amazon, having migrated to this design [26, 29, 57, 58]. It divides monolithic applications into the graphs of multiple single-purpose components, interacting through RPC or REST interfaces to collectively provide a service. Figure 1 shows an example of a social network application. While this modular design allows agile development and resource flexibility where each component can be developed and scaled independently, it complicates resource management [28, 30, 31, 51, 70]. The application owner has to request resources for each component (a container or a pod) to ensure the application can serve the traffic from users. This is often assisted by resource estimation techniques to forecast *future* utilization such that the application owner can prepare and allocate such resources ahead of time to maintain QoS [51, 70]. Another role of resource estimation is to estimate the expected usage in the *past*. The application owner can compare the actual past usage with the expected consumption to identify system anomalies [19, 21].

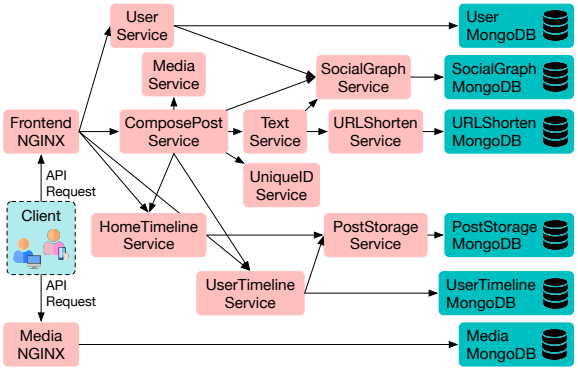


Figure 1. The flow of API requests across components in a social network application from DeathStarBench [29].

In this context, several solutions have emerged. They monitor the recurring resource consumption patterns of applications and forecast their future requirements [20, 35, 36, 40, 45]. However, such solutions still follow the traditional component-focused forecasting approach used for monolithic applications. Specifically, they lack any API awareness and cannot identify the resource footprints of specific APIs across the components. Since the user-facing APIs in an application often represent business logic (e.g., a `/purchaseProduct` API request refers to a consumer purchasing a product), tracking the impact of any change in the API traffic on resource consumption is vital to application owners. For instance, they may expect more traffic towards certain APIs during a holiday season and need to prepare in advance to meet the additional traffic demand. The resources can be scaled up to meet the performance requirements or scaled down to save cost. Yet, existing approaches are unable to answer such queries as the estimation problem has to be formulated as a function of API traffic. Moreover, tracking the resource footprint of APIs enables application owners to verify the sanity of applications. Even though the resource consumption appears to be consistent with the historical trends, any consumption that the corresponding API traffic cannot justify may represent an anomaly, such as bugs, cryptojacking, and crypto-ransomware attacks.

In spite of its benefits, identifying the resource footprint of an API on specific components can be challenging for several reasons. First, an API has its own flow that traverses different components in the application based on the underlying business logic. Second, multiple APIs can trigger the same component while consuming its resources differently. Third, an API may exhibit different consumption based on external factors, such as the content of a request. In this paper, we

address the above challenges by proposing DeepRest, a deep learning approach for resource estimation. DeepRest infers the causality between user activities on the application and resource consumption by tracking application traces and resource metrics. Our contributions are as follows.

- We propose a general-purpose solution to estimate resource consumption as a function of API traffic. That is, DeepRest is privacy-preserving and does not assume any knowledge of the application or its APIs.
- We develop a feature extractor to turn the unstructured traces collected across components into structured features critical to resource estimation. This allows DeepRest to be agnostic to the application's component structure and to be employed to any application without modification to DeepRest.
- We introduce a multi-expert neural design, where the dedicated expert for a resource in a component automatically discovers its dependency on both APIs and resources in other components. For instance, the disk usage of a component can be intensely dependent on the CPU consumption of another component for a specific API.

We use *Social Network* and *Hotel Reservation* applications from DeathStarBench [29], a microservice benchmark suite, to evaluate DeepRest. We identify three common use cases for application owners to query DeepRest for resource allocations, namely, API traffic with (i) unseen scales of users, (ii) unseen API compositions, and (iii) unseen traffic shapes, all violating the workload patterns observed in the past. In addition to resource estimation, our experiments on application sanity check demonstrate the use of DeepRest to detect two major cybersecurity threats. We identify ransomware and cryptojacking attacks by tracking the violation of causality between user activities and resource consumption. We envision that DeepRest can be deployed in on-premises clusters or a cloud as a service to serve any hosted application.

2 Related Work

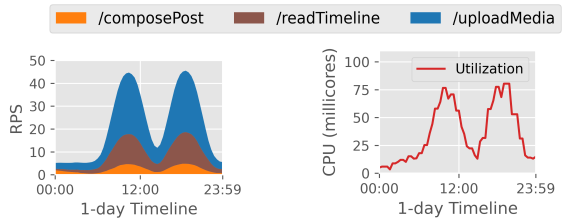
Auto-scaling has become an industry standard [2, 3, 5, 7] to relieve the pain of resource management by automating scaling decisions. The mechanisms can be categorized into two types. *Reactive approaches* continuously monitor the system and trigger a scaling action when a predefined condition is met [13]. For instance, Taherizadeh and Stankovski [60] generate dynamically changing thresholds per container such that it will be scaled when its resource utilization exceeds the threshold. MIRAS [68] and FIRM [51] use reinforcement learning to directly predict what scaling action should be taken. Similarly, ATOM [31] and Microscaler [69] use a combination of queuing theory and heuristics to find the best resource configuration of microservices. While some resources (e.g., CPU) may be scaled instantly if available, others may take time to request (e.g., storage type or additional capacity). Hence, the reaction time can be insufficient to avoid overloading the system. In this regard, DeepRest can assist in schedule-based autoscaling [1] so that the resources can be scaled prior to the surge in user requests. Moreover, knowing resource consumption in advance allows additional provisioning time, especially when the resources cannot be scaled up further and require application reconfiguration, e.g., by adding more instances or replicas.

Predictive techniques estimate resource consumption in advance. Verma et al. [63] use season-trend decomposition, while ARIMA [17] is another popular choice in using time-series analysis to help auto-scaling [46, 47, 54]. They use the historical resource utilization of a container or a virtual machine to forecast its consumption in the near future. MF-LSTM [61], ASFM [50], and HANSEL [66] improve the prediction accuracy of resource scaling using neural networks. The main weakness in these approaches is that they depend on recurring patterns in resource consumption, thus not suitable to predict unseen or occasional traffic patterns. Moreover, application owners may provision resources according to the usage trends, expressed in the form of API traffic. However, the existing approaches are unable to answer such queries, and it is difficult without knowing how user activities impact different components according to the application's business logic. Another predictive technique proposed by Zhou and Maas [72] explores the use of distributed traces to predict storage-related metrics. It assumes the application owner has injected expressive logs in traces and uses natural language processing tools to learn from them [71]. While it demonstrates the possibility of using information from the application layer to infer resource consumption, it is privacy-intrusive and heavily dependent on how the owner instruments the application.

Resource metrics reflect system health. Hence, a number of works have been proposed to detect anomalies in such metrics to assist cluster management [27, 52, 55, 64, 73]. They adopt a similar pipeline with, e.g., Seasonal (Hybrid) ESD [33] using robust statistics, RePAD [42] using deep neural networks, and DLA [55] using Markov models to estimate the expected utilization, and any measurement deviating significantly will be regarded as anomalies. The common weakness of the above approaches is the dependency on recurring patterns. Since the traffic to an application can change due to any benign reason (e.g., an event), those unseen trends/periodicity on metrics should not be identified as anomalous as long as they can be justified w.r.t. the traffic.

3 Background and Design

API-driven Microservices. DeepRest is a resource estimation system for API-driven microservices, which expose API endpoints for their users to invoke through, e.g., HTTP requests. Each API endpoint is single-purposed and often requires specific inputs accompanied with the request to complete the task. Once an API request is received, the entry component (e.g., an NGINX webserver) processes it based on



(a) API Traffic **(b)** Consumed CPU Resources

Figure 2. An example one-day API traffic and its consumed CPU resources in the FrontendNGINX webserver.

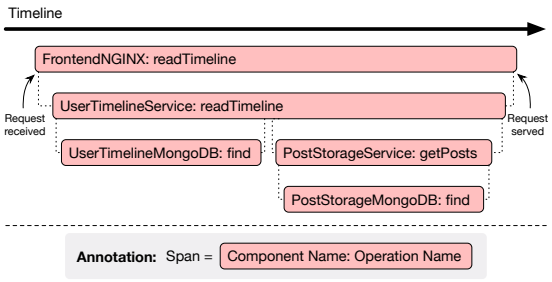


Figure 3. The execution diagram of a trace originated from a `/readTimeline` API request.

the implemented business logic and may trigger other components to serve the request collectively. The stream of API traffic to an application can be represented as a multivariate time-series, indicating how many Requests Per Second (RPS) are received for every exposed API endpoint. Throughout this paper, we use the social network application in Figure 1 as a motivating example. Figure 2a gives an example of one-day traffic to the social network with two peak-hour and three example APIs, showing at each time step, how many `/composePost` (orange), `/readTimeline` (brown), and `/uploadMedia` (blue) requests are received per second.

To perform an API-aware resource estimation, in addition to the resource consumption metrics (e.g., Figure 2b shows the CPU utilization of a component serving the traffic in Figure 2a collected by monitoring tools such as cAdvisor [4] and Prometheus [12]), we also need to track the application logic. Such observability can be provided by *distributed tracing*. Distributed tracing was originally designed to help application owners pinpoint the culprit component responsible for poor performance or a failure [56]. With tracing, every API request received by the application is recorded as a *trace*. Figure 3 visualizes an example trace using the format commonly adopted by off-the-shelf tracing tools (e.g., Jaeger [8]). Each operation performed by the application to serve an API request is represented as a *span*. For instance, the `/readTimeline` API request first triggers the FrontendNGINX component, creating the root span (top) in Figure 3. Then, it invokes another component, UserTimelineService, and spawns a child span, which subsequently communicates with the UserTimelineMongoDB to find the post IDs belonging to the target user's timeline and retrieves post contents from the PostStorageService querying PostStorageMongoDB. Given that the entire lifetime of an API request is encapsulated in these traces, harvesting knowledge from them allows DeepRest to infer how each API request interacts with the application and estimate its resource consumption.

DeepRest Design. DeepRest adheres to the following design principles:

- *Application-independence*: DeepRest should not assume any knowledge of the application components or its APIs. This allows DeepRest to serve any application deployed in a cluster without modification. The application only needs to include the desired libraries to enable monitoring and tracing. Such libraries are becoming standard today in microservice frameworks [11].
- *Privacy-preserving*: DeepRest should only rely on common resource metrics and distributed traces without requiring any application specific information (e.g., logs). All sensitive attributes (e.g., component and API details) need to be hashed before being ingested by DeepRest to minimize the risk of privacy leakage.
- *Unsupervised learning*: DeepRest should not require any manual intervention by the application owner to label the data. Also, it should not require separate preparation for training, e.g., with a custom workload, but has to learn directly from live user traffic.

During the application learning phase, DeepRest queries the telemetry server in the production environment to obtain distributed traces and past resource utilization of each component. They are used by DeepRest to learn which components are triggered and what resources are used by each API endpoint. Upon the completion of application learning, the application owner can perform the following two types of queries, as depicted in Figure 4. (1) Taking the expected API traffic as input, DeepRest allows the application owner to estimate the required resources to serve the specified API traffic. The API traffic is sent to DeepRest's trace synthesizer to produce synthetic traces, following the distribution obtained in the application learning phase. (2) Taking the real API traffic and traces as input, DeepRest allows the application owner to estimate how much resources should be consumed in the corresponding time frame. The application's sanity can be checked based on whether the resource utilization can be justified w.r.t. the API traffic. We demonstrate the use of sanity checks for identifying ransomware and cryptojacking attacks in Section 5. Either type of queries provides a sequence of traces (synthetic or real) to DeepRest's feature extractor to transform unstructured traces into structured features. Our API-aware deep resource estimator then predicts the expected utilization and the confidence interval for each resource in each component in the application.

4 DeepRest Methodology

4.1 Distributed Tracing Feature Extractor

Distributed traces offer invaluable information for DeepRest to understand how API requests interact with different

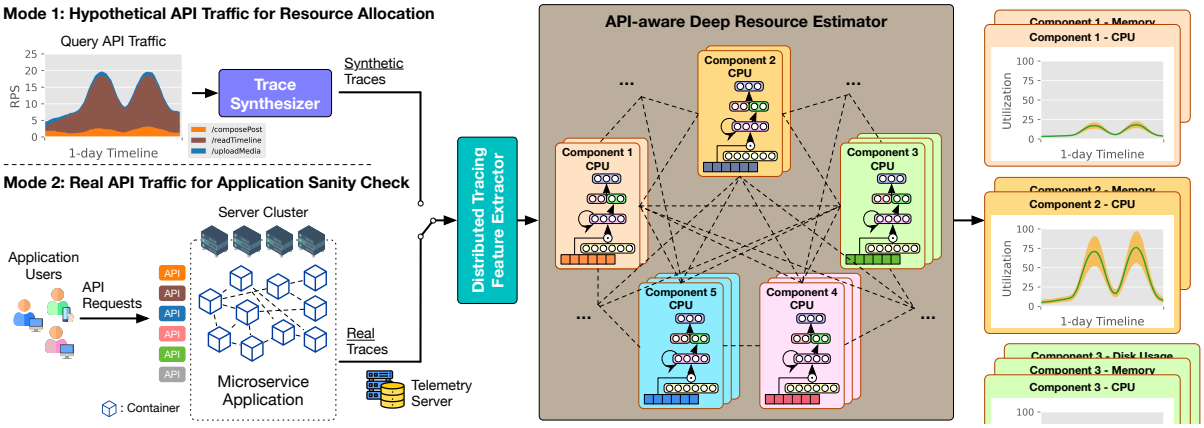


Figure 4. The end-to-end design of DeepRest with two modes in the query phase. The synthetic/real traces are sent to the feature extractor to transform them into structured features for the API-aware deep resource estimator to predict utilization.

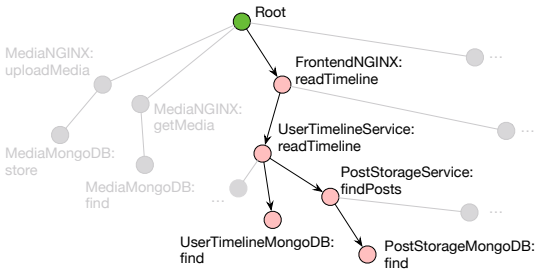


Figure 5. The execution topology graph highlighted with the invocation path of the example trace in Figure 3.

components. However, by default, they are represented as execution diagrams of spans (recall Figure 3). Depending on the payload of the API request triggering different business logic, the number of spans varies from trace to trace. This imposes challenges for discovering knowledge using machine learning techniques because structured (vectorized) inputs are required [32]. Thus, feature engineering needs to be done for DeepRest to digest such unstructured data.

While expressive logs may be found in spans as the application owner can insert them for debugging purposes (e.g., SQL statements can be associated with the spans created in MySQL components), DeepRest considers only the execution topology. This is in stark contrast to existing works mining the logs in traces with natural language processing techniques [71, 72], which can be privacy-intrusive and application-dependent. Given that each span must be associated with the component name (e.g., PostStorageService) and the operation name (e.g., findPosts), we can construct an execution topology graph where each node is a (component, operation) pair found in those traces for application learning. A trace can then be represented as a directed invocation path in the graph, as shown in Figure 5, where the highlighted path is equivalent to the example trace in Figure 3.

The intuition of DeepRest feature engineering is that the utilization of a resource in a component is related to how many times the component is triggered *conditioned* on the business logic. Such logic can be inferred from the invocation path. Taking the disk usage of the MediaMongoDB component as an example, if the invocation path is:

"Root → MediaFrontend:uploadMedia → MediaMongoDB:**store**"

one can expect the disk usage to increase. However, if the invocation path triggering the *same* MediaMongoDB is:

"Root → MediaFrontend:getMedia → MediaMongoDB:**find**"

we should not expect changes to disk usage. Note that this is an overly simplified example. In practice, we hash the component and operation names to avoid privacy leakage, especially when DeepRest is deployed as a service. Also, the cause of disk usage is rather intuitive, but most resources (e.g., CPU) are non-trivial. DeepRest constructs a feature space that covers all possible invocation paths from the root to every node in the execution topology graph such that the DNN estimator can discover which ones are relevant for the prediction task and exploit them for estimation.

Algorithm 1 shows the feature space construction process, where the number of entries in the returned path-to-feature map $\boldsymbol{M}$ is the dimensionality of the feature space. Resource utilization is measured as the average consumption over a time window (e.g., 5 seconds). We partition the collected traces accordingly and transform the t-th partition $\mathcal{T}_t$ into the feature vector $\boldsymbol{x}_t$ using Algorithm 2. The time-series of feature vectors $\{\boldsymbol{x}_1, ...\boldsymbol{x}_T\}$ is sent to our API-aware deep resource estimator to predict the utilization time-series $\{\hat{\boldsymbol{y}}_1^{c,r}, ..., \hat{\boldsymbol{y}}_T^{c,r}\}$ for all component c's and their resource r's.

4.2 API-aware Deep Resource Estimator

We adopt a multi-expert design in our API-aware deep resource estimator. For each resource in each component, we build a dedicated DNN expert, as depicted in Figure 6. The swarm of experts has an API-aware neural design and are allowed to communicate with each other to exploit the strong dependencies across components and resources in microservices [29]. Let $F^{c,r}$ be the expert dedicated to estimating the

Algorithm 1 DeepRest feature space construction

Input: $\mathcal{T}$: the invocation paths of traces collected in the application learning phase
Output: $\mathcal{M}$: the path-to-feature map
procedure CONSTRUCT-FEATURE-SPACE($\mathcal{T}$)
  $\mathcal{M} \leftarrow$ HASHMAP()
  for each trace $\mathcal{T}_i$ **in** $\mathcal{T}$ **do**
    $\mathcal{M} \leftarrow$ TRAVERSE-CONSTRUCT($\mathcal{M}, \mathcal{T}_i.Root$, LIST())
  return $\mathcal{M}$
procedure TRAVERSE-CONSTRUCT($\mathcal{M}$, node, prefix)
  prefix.*append*(node.*ID*)
  if prefix **not in** $\mathcal{M}$ **then**
    $\mathcal{M}$[prefix] $\leftarrow \mathcal{M}.length$
  for each child **in** node.*children* **do**
    $\mathcal{M} \leftarrow$ TRAVERSE-CONSTRUCT($\mathcal{M}$, child, prefix)
  return $\mathcal{M}$

Algorithm 2 DeepRest feature extraction

Input: $\mathcal{T}_t$: the invocation paths of traces collected at the t-th time window; $\mathcal{M}$: the path-to-feature map
Output: $\boldsymbol{x}_t$: the feature vector at the t-th time window
procedure EXTRACT-FEATURE($\mathcal{T}_t, \mathcal{M}$)
  $\boldsymbol{x}_t \leftarrow$ ZEROVECTOR(size = $\mathcal{M}.length$)
  for each trace $\mathcal{T}_i$ **in** $\mathcal{T}_t$ **do**
    $\boldsymbol{x}_t \leftarrow$ TRAVERSE-EXTRACT($\boldsymbol{x}_t, \mathcal{T}_i.Root$, LIST())
  return $\boldsymbol{x}_t$
procedure TRAVERSE-EXTRACT($\boldsymbol{x}_t$, node, prefix)
  prefix.*append*(node.*ID*)
  $\boldsymbol{x}_t$[$\mathcal{M}$[prefix]] $\leftarrow \boldsymbol{x}_t$[$\mathcal{M}$[prefix]] + 1
  for each child **in** node.*children* **do**
    $\boldsymbol{x}_t \leftarrow$ TRAVERSE-EXTRACT($\boldsymbol{x}_t$, child, prefix)
  return $\boldsymbol{x}_t$

resource r in component c. It takes the time-series of feature vectors $\{\boldsymbol{x}_1, ...\boldsymbol{x}_T\}$ from the DeepRest feature extractor as input and returns the estimated utilization $\hat{\boldsymbol{y}}_1^{c,r}, ..., \hat{\boldsymbol{y}}_T^{c,r} = F^{c,r}(\boldsymbol{x}_1, ...\boldsymbol{x}_T)$ within the same time frame.

API-aware Neural Design. Recall from Section 4.1 that each feature corresponds to one possible invocation path originated from an API request (e.g., the path in Figure 5 comes from `/readTimeline`). To facilitate the learning, we explicitly instruct the DNN expert to discover which APIs (their invocation paths) are relevant to the resource it is responsible for estimating. We introduce an API-aware mask $\boldsymbol{m}^{c,r}$ that is a learnable weight vector to mask the input features at each time step:

$$\tilde{\boldsymbol{x}}_t = \sigma(\boldsymbol{m}^{c,r}) \odot \boldsymbol{x}_t, \tag{1}$$

where $\sigma(\cdot)$ is the sigmoid function and $\odot$ is the Hadamard product. The mask $\boldsymbol{m}^{c,r}$ is fine-tuned during the optimization process to only amplify those features that are relevant

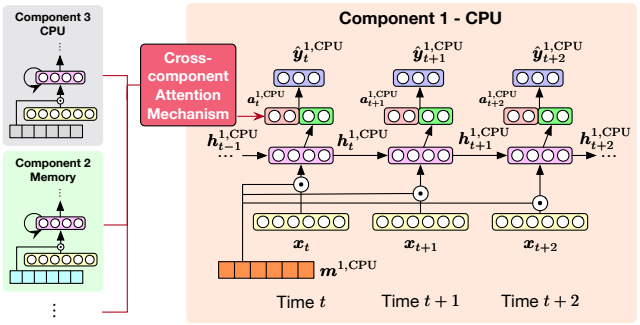


Figure 6. The API-aware DNN experts in DeepRest.

and useful to boost the estimation accuracy. We show in Section 6 that interpreting these learned masks enables various interesting use cases in addition to resource estimation.

Recurrent Architecture. Resource estimation is inherently a time-series prediction problem. First, the utilization of a resource at the current time step is not only conditioned on what requests have been received at the same time step but also in the past because of possibly queuing effects [70]. Also, the duration of the time frame can vary from query to query (e.g., the application owner may want to estimate one day of traffic for the first query and only 30 minutes for the second query, etc.). One cannot simply use feedforward neural networks for estimation, which (1) do not consider the temporal factor and (2) require a fixed-length input. In light of these properties, we incorporate a recurrent structure into each DNN expert using Gated Recurrent Units (GRUs) [22] to allow information propagation over time and estimating variable-length time-series. In particular, at time step t, we send the corresponding masked feature vector $\tilde{\boldsymbol{x}}_t$ (see Equation 1) as input and compute the hidden states $\boldsymbol{h}_t^{c,r}$:

$$\begin{aligned}
z_t^{c,r} &= \sigma(\boldsymbol{W}_z^{c,r}\tilde{\boldsymbol{x}}_t + \boldsymbol{U}_z^{c,r}\boldsymbol{h}_{t-1}^{c,r} + \boldsymbol{b}_z^{c,r}) \\
k_t^{c,r} &= \sigma(\boldsymbol{W}_k^{c,r}\tilde{\boldsymbol{x}}_t + \boldsymbol{U}_k^{c,r}\boldsymbol{h}_{t-1}^{c,r} + \boldsymbol{b}_k^{c,r}) \\
\tilde{\boldsymbol{h}}_t^{c,r} &= \tanh(\boldsymbol{W}_h^{c,r}\tilde{\boldsymbol{x}}_t + \boldsymbol{U}_h^{c,r}(k_t^{c,r} \odot \boldsymbol{h}_{t-1}^{c,r}) + \boldsymbol{b}_h^{c,r}) \\
\boldsymbol{h}_t^{c,r} &= z_t^{c,r} \odot \boldsymbol{h}_{t-1}^{c,r} + (1 - z_t^{c,r}) \odot \tilde{\boldsymbol{h}}_t^{c,r}
\end{aligned} \tag{2}$$

where $\boldsymbol{W}^{c,r}$'s, $\boldsymbol{U}^{c,r}$'s, and $\boldsymbol{b}^{c,r}$'s are trainable model weights, including the learning to control how the memory should be updated (i.e., $z_t^{c,r}$) or reset (i.e., $k_t^{c,r}$) to construct the hidden states $\boldsymbol{h}_t^{c,r}$, which will be subsequently used to produce the estimated utilization *and* fed to the next time step $t+1$. With this design, estimation at time t is conditioned on the preceding inputs through the hidden states extracted in the past, and Equation 2 can be repeated until all $\boldsymbol{x}_t$'s are processed.

Cross-component Attention Mechanism. Microservices are correlated [29]. For instance, the increase in CPU of the ComposePostService component can imply the increase in the disk usage of the PostStorageMongoDB component as a new social media post will be stored. Such strong dependencies can be valuable, but identifying them manually to

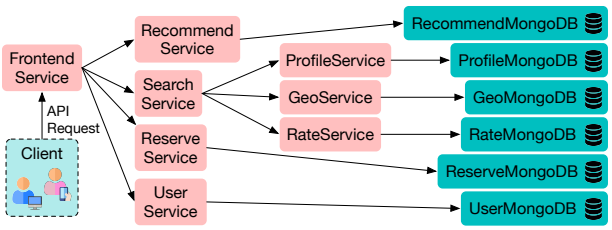


Figure 7. The hotel reservation application.

optimize the architectural design of DeepRest can be infeasible in practice. We hence develop a data-driven approach through the concept of attention mechanism from neural machine translation [15, 62], allowing each expert to optionally communicate with others. In particular, we construct the attention vector $\boldsymbol{a}_t^{c,r}$ at time t as follows:

$$\boldsymbol{a}_t^{c,r} = \sum_{(c',r') \in \boldsymbol{C} \times \boldsymbol{\mathcal{R}} \setminus \{(c,r)\}} \alpha_{c',r'}^{c,r} \boldsymbol{h}_t^{c',r'}, \tag{3}$$

where we abuse the notation $\boldsymbol{C} \times \boldsymbol{\mathcal{R}}$ to represent the set of all component-resource pairs, and $\alpha_{c',r'}^{c,r}$ is the trainable weight controlling how much information from the expert $F^{c',r'}$ should be taken by the expert $F^{c,r}$. The attention vector is then concatenated with the hidden states from Equation 2 and sent to the fully connected layer parameterized by $\boldsymbol{V}^{c,r}$ to obtain the final estimation at time t:

$$\hat{\boldsymbol{y}}_t^{c,r} = \boldsymbol{V}^{c,r}(\boldsymbol{a}_t^{c,r} || \boldsymbol{h}_t^{c,r}), \tag{4}$$

which is a three-dimensional vector representing (1) the expected utilization, (2) the lower limit, and (3) the upper limit of the confidence interval at time t.

4.3 Quantile Regression Optimization

We have described how DeepRest employs a swarm of DNN experts to estimate resource utilization. During the application learning phase, we exploit the traces collected in the past by the telemetry server to make estimations and compare them with the resource utilization collected during the same period to guide the iterative fine-tuning of trainable parameters. Rather than giving a single-point estimation at each time step, DeepRest formulates a quantile regression problem to also estimate the δ-confidence interval. Given the auxiliary quantile loss [38]:

$$\boldsymbol{Q}(\Delta|\delta) = \begin{cases} \delta\Delta & \text{if } \Delta \geq 0 \\ (\delta - 1)\Delta & \text{if } \Delta < 0, \end{cases} \tag{5}$$

the optimization function of DeepRest with input time-series $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_T\}$ and ground truth utilization $\{y_1^{c,r}, ..., y_T^{c,r}\}$ for each resource r in component c is formulated as follows:

$$\mathcal{L}(\boldsymbol{\theta}|\delta) = \frac{1}{T} \sum_{t=1}^{T} \sum_{(c,r) \in \boldsymbol{C} \times \boldsymbol{\mathcal{R}}} \Big[\boldsymbol{Q}(\hat{y}_{t,\text{exp}}^{c,r} - y_t^{c,r}|0.5) + \boldsymbol{Q}(\hat{y}_{t,\text{low}}^{c,r} - y_t^{c,r}|\frac{1-\delta}{2}) + \boldsymbol{Q}(\hat{y}_{t,\text{up}}^{c,r} - y_t^{c,r}|\delta + \frac{1-\delta}{2}) \Big], \tag{6}$$

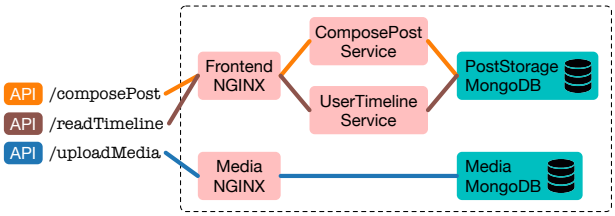


Figure 8. Three representative APIs and their simplified invocation relationships with six components in the social network application.

where $\boldsymbol{\theta}$ denotes all trainable parameters in DeepRest, $\hat{y}_{t,\text{exp}}^{c,r}$ is the expected utilization, $\hat{y}_{t,\text{low}}^{c,r}$ and $\hat{y}_{t,\text{up}}^{c,r}$ are the lower limit and upper limit of the confidence interval respectively. We use an optimizer to iteratively fine-tune $\boldsymbol{\theta}$, minimizing the above loss function until convergence.

4.4 Trace Synthesizer

DeepRest allows the application owner to submit API traffic that the application is expected to serve in the future. This type of queries does not provide traces to DeepRest as the API traffic is yet to be served. We introduce a trace synthesizer by observing the traces captured in the application learning phase. For each API, we find all traces it triggered and estimate the probability distribution of invocation paths conditioned on the API, i.e., Prob($\mathcal{P}$|API). Once DeepRest receives the query API traffic, which, e.g., specifies N requests of `/readTimeline` at a particular time step, we can synthesize the invocation paths by sampling from Prob($\mathcal{P}$|API = `/readTimeline`) N times. Then, DeepRest can convert the query API traffic into a sequence of invocation paths (traces) for downstream modules to operate.

5 Experimental Evaluation

5.1 Experiment Setup

Microservice Applications. We evaluate DeepRest on two microservice benchmarks - a *Social Network* and a *Hotel Reservation* applications from DeathStarBench [29]. The social network comprises 23 stateless and 6 stateful components (see Figure 1) interacting with each other to provide users functionalities to publish, read, and react to social media posts through 11 API endpoints. The hotel reservation system (see Figure 7 for a simplified architecture) has 12 stateless and 6 stateful components with 4 API endpoints for searching, getting recommendations, and reserving hotels. These applications cover a wide range of workflow patterns and uses various programming languages (e.g., Go, C++, and Lua). While the following experiments are conducted using the entire application, for the discussion, we focus on three representative APIs and six components in the social network. The API invocation and component relationships of the application are illustrated in Figure 8.

Workload Generation. We generate workloads based on real-world behaviors using Locust [10]. The social network graph and post contents are imported from real-world

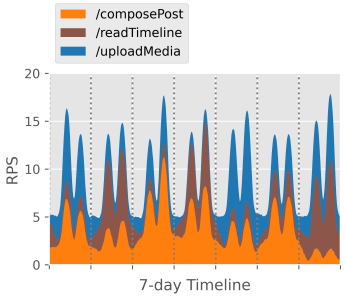
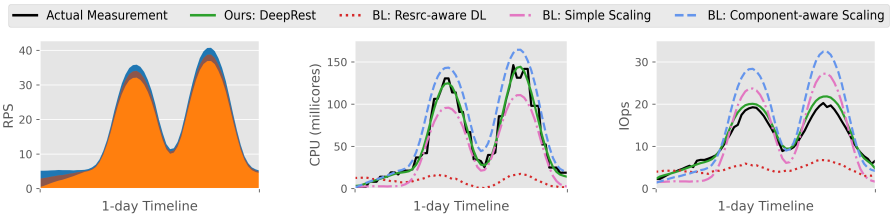
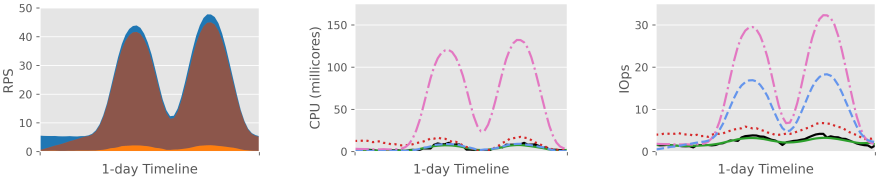


Figure 9. An example 7-day API traffic during the application learning phase with three APIs: `/composePost`, `/readTimeline`, and `/uploadMedia`. Each day has two peak-hour (e.g., lunchtime and late evening) to match real-world social network behaviors.



(a) Query API Traffic **(b)** ComposePostService - CPU **(c)** PostStorageMongoDB - IOps

Figure 10. Estimation given `/composePost`-intensive API Traffic



(a) Query API Traffic **(b)** ComposePostService - CPU **(c)** PostStorageMongoDB - IOps

Figure 11. Estimation given `/readTimeline`-intensive API Traffic

datasets from Facebook to resemble the realistic interactions between users [53]. The photos are drawn from the INRIA dataset, having pictures of people with various resolutions [25]. For the hotel reservation system, we follow the same setting as in [29]. Our generator simulates one-day traffic in five minutes. By default, we follow real-world phenomenon to simulate two peak-hour per day (e.g., lunchtime and late evening). API requests are sent according to real-world distributions with variations from day to day to mimic non-deterministic properties in practice [41].

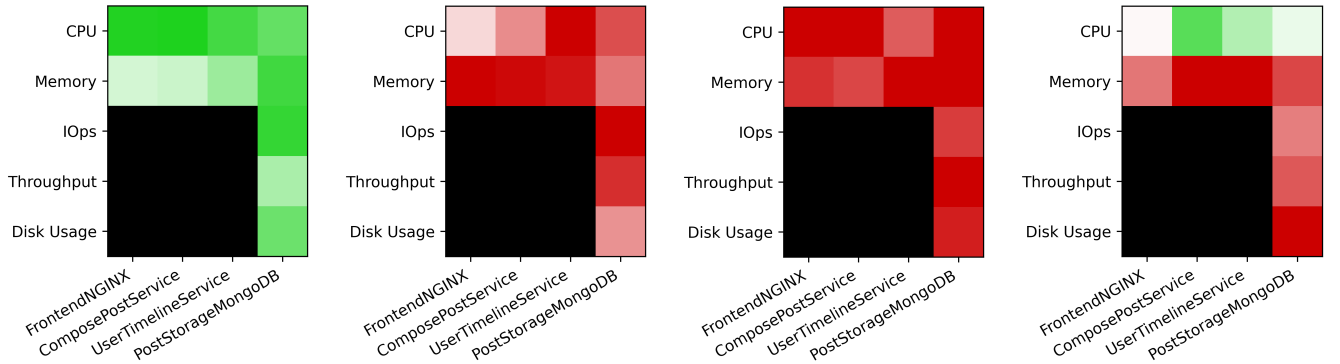
System Setup and Hyperparameters. We deploy all microservices in separate Docker containers orchestrated by Kubernetes [9]. We install the most commonly-used telemetry tools, including Jaeger [8] for distributed tracing and Prometheus [12] for resource monitoring. Our prototype considers CPU and memory utilization in all components, and also write IOps, write throughput, and disk usage in stateful components. Hence, our experiments cover 76 resources in 29 components for social network and 54 resources in 18 components for hotel reservation. The scrape interval is set to be 5 seconds, and all other configurations are set to be their default value. DeepRest is implemented using PyTorch [49]. We use the same hyperparameter setting to train two instances of DeepRest, one for each application. The DNN experts have an identical neural architecture: one GRU layer with 128 hidden units and the cross-component attention mechanism with 128 hidden units. We collect seven days of data for application learning and train DeepRest with 30 epochs and a batch size of 32. Stochastic gradient descent is used as the optimizer with a learning rate of 0.001. The experiments are conducted on Intel Core i7-9700K CPU x 8 with 32 GiB of memory and a GeForce RTX 2080 SUPER GPU running Ubuntu 18.04.3 LTS.

Comparison Baselines. We have implemented three competitive baselines: (i) *resource-aware deep learning* trains a neural network for each component-resource pair taking the last day of resource utilization as input to predict the next-day utilization, which is a typical periodicity-based approach using only information from the resource layer. (ii) *simple scaling* scales all resources in all components by the *same* factor according to how many more or fewer API requests will be received by the application w.r.t. the past. With this baseline, the application can be scaled according to the usage of the API endpoints without relying on application traces. (iii) *component-aware scaling* uses distributed traces to learn a scaling factor for each component according to how many more or fewer invocations it will expect w.r.t. the past. This approach scales all resources in a component by the same factor. While the baseline is aware of the API flow and the components' resource utilization, it need not learn correlation between them.

5.2 Estimation Accuracy Analysis

To highlight the advantages of DeepRest, we begin with qualitative analysis on estimating the CPU utilization in the ComposePostService and the write IOps in the PostStorage-MongoDB given two query API traffic patterns in the social network application. Each query sends one-day traffic to DeepRest for resource estimation. Figure 9 shows the 7-day API traffic we used for application learning to train DeepRest.

`/composePost`-dominated Traffic. Figure 10a shows the one-day query API traffic with two peak-hour similar to the application learning phase, but it expects much more requests, and yet the additional ones are primarily `/composePost` (the orange region). Figure 10b and Figure 10c compare different techniques with the actual measurements of the CPU utilization and the write IOps respectively. The actual measurements are collected by running the query traffic in the application and are used as the ground truth in this experiment: an accurate resource estimator should produce a curve close to the actual measurements (the black curve). Resource-aware deep learning (i.e., resrc-aware DL with the red curve)



(a) Ours: DeepRest **(b)** BL: Resrc-aware DL **(c)** BL: Simple Scaling **(d)** BL: Component-aware Scaling

Figure 12. The estimation quality heatmaps on four components (columns) and five resources (rows) in social network. The green color represents accurate prediction while the red color indicates inaccurate estimation. IOps, throughput, and disk usage are inapplicable in stateless components and marked in black. DeepRest offers the most stable and accurate estimation compared with all other approaches.

performs the worst because it does not consider the number of requests to be served in the one-day period. Compared with Figure 9, the query traffic has 2× more requests than in the past, and those additional requests consume much more CPU and incur much more IOps. Differently, with DeepRest (green curve) giving the most precise estimation, simple scaling (pink curve) and component-aware scaling (blue curve) can also capture the burst in CPU and IOps incurred by the increased number of requests. The results validate the importance of bridging application and resource layers.

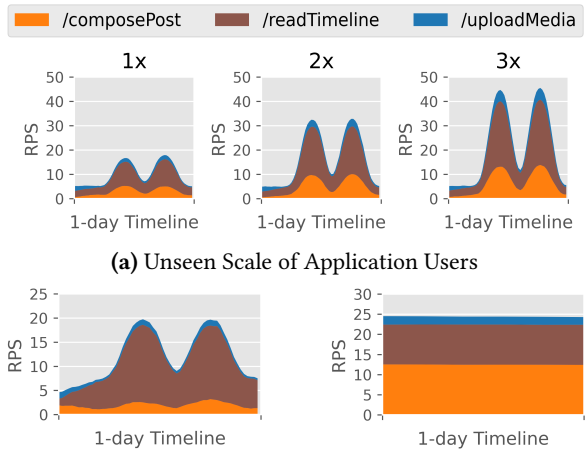
`/readTimeline`-dominated Traffic. Figure 11 shows the same set of experiments with a different query API traffic, dominated by `/readTimeline` (the brown region in Figure 11a). Interesting observations can be obtained by contrasting the results with Figure 10. First, even though the total number of requests in both scenarios are alike, the actual measurements on the CPU utilization in the ComposePostService do not increase similarly (the black curve in Figure 11b). This is because from Figure 8, the `/readTimeline` API does not invoke the ComposePostService. However, simple scaling still mistakenly estimates a high CPU utilization since it cannot know which components will be triggered, and all resources in the application will be scaled in the same way. Component-aware scaling addresses this flaw by utilizing distributed traces, and hence it can offer a better CPU estimation in this scenario. Focusing on the write IOps estimation in Figure 11c, our program analysis reveals that `/readTimeline` does not incur any write operations on the PostStorageMongoDB. It resonates with the actual measurements where the IOps does not increase wildly as in Figure 10c. Due to the same reason, simple scaling produces an inaccurate estimation. But interestingly, component-aware scaling overestimates by a large margin. This is because even though it knows this component will be busy serving the query API traffic, it does not know which resource(s) will be utilized, and all resources in the component will be scaled in the same way. Finally, DeepRest can again deliver high-quality estimation on both resources because it is designed to learn the resource footprint of each API. Since `/readTimeline` has a low correlation with the ComposePostService and the write IOps in the PostStorageMongoDB, it expects a low utilization for the given query API traffic.

These observations are not limited to the two resources discussed above but can be consistently obtained in other resources and components as highlighted in Figure 12, where the green cells refer to accurate estimation while the red cells mean the opposite. Given that DeepRest outperforms competitive baselines, we delve into detailed analysis on two use cases: resource allocation and application sanity check.

5.3 Resource Allocation

The application owner can use outputs from DeepRest to allocate resources ahead of time, which is particularly useful for resources that cannot be assigned instantly. One immediate question is: how accurate can DeepRest be in answering queries that submit API traffic different from what the application has been serving. We categorize three types of API traffic that are unseen in the application learning phase based on common business scenarios and use a popular metric in time-series estimation: mean absolute percentage error to quantify the quality of the resource allocation plans generated by different estimation algorithms. The evaluation metric reflects how much resources will be under/over-provisioned on average at a time step. Even though DeepRest can take queries of any duration (e.g., 1-hour, 1-day, 3-day, etc.), we consider one-day traffic for simplicity. Each type of query is repeated nine times with minor variations in the maximum number of application users and the composition of APIs to record the worst-case performance.

Unseen Scale of Application Users. The application owner may expect more users than ever and attempt to estimate how much resources to prepare for such growing popularity of their application (or simply a burst in users due to a special weekend sale in online shops). Figure 13a



(a) Unseen Scale of Application Users

(b) Unseen API Composition **(c)** Unseen API Traffic Shape

Figure 13. Example one-day query API traffic of the three types of queries in business scenarios.

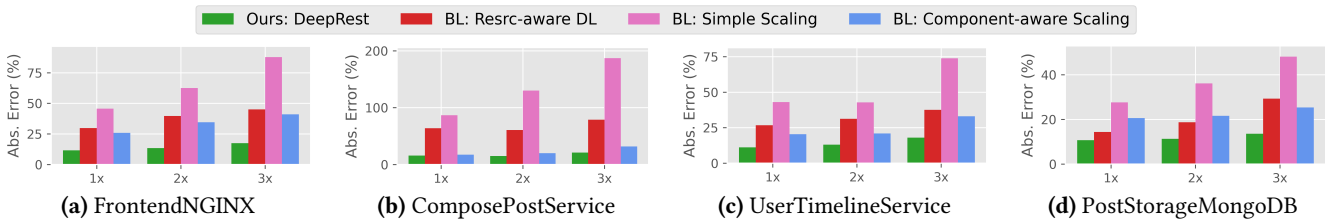
shows example queries of three cases: 1× (same scale), 2×, and 3× more users than the application learning phase (recall Figure 9). Figure 14 compares four algorithms with a focus on CPU allocation to four components: (a) FrontendNGINX, (b) ComposePostService, (c) UserTimelineService, and (d) PostStorageMongoDB. While we can observe a larger scale of users leads to a higher error in allocation, DeepRest consistently outperforms other approaches by a large margin. To demonstrate the applicability of DeepRest on different applications, Figure 17 highlights the results of estimating the CPU utilization of the FrontendService in the hotel reservation system (see Figure 7). With at most 200 concurrent users at the application learning phase, we query DeepRest to estimate how much CPUs are needed to serve 3× more users. The results indicate that DeepRest still offers the most accurate estimation, but both simple scaling and component-aware scaling lead to significant overestimation. This is because little errors can be magnified when a large number of users are expected. Instead, DeepRest relies on the universal approximation property of neural networks [34] to robustly learn the mapping from API traffic to resource utilization.

Unseen API Compositions. The application owner may expect the change of user behaviors in using the application due to, e.g., holidays. One example is an online shop that can be dominated by API requests browsing products before Thanksgiving, and the users begin to purchase items through the corresponding API when the sale starts. Figure 13b gives an example query with API composition: 10% of `/composePost`, 85% of `/readTimeline`, and 5% of `/uploadMedia`, which has never been observed during the application learning phase. Figure 15 compares two settings: API traffic with compositions that have been or have not been observed in the application learning phase. We observe a similar trend in allocation quality in both settings, where DeepRest always offers the most accurate plan, followed by the component-aware scaling and resrc-aware DL, and simple scaling leads to the most significant error.

Unseen API Traffic Shapes. While we have been using two peak-hour per day to be the traffic shape, the application owner may, e.g., expand the customer base from one timezone to multiple ones. The aggregated traffic can become flat. This serves as the third business scenario where the application owner attempts to estimate how much resources to prepare if the traffic shape is different from the past. Figure 13c gives an example of API traffic with a flat shape in contrast to the spiky shapes during application learning. Figure 16 compares two settings: (1) the application learning phase has a traffic shape of two peak-hour per day, but the query traffic is flat, and (2) the reverse. We can consistently observe the accurate and stable performance of DeepRest. Figure 18 explains such an observation by two examples with "2-peak/day→Flat": (a) allocating CPU in the ComposePostService and (b) estimating write IOps in the PostStorageMongoDB. Since resrc-aware DL assumes the future utilization always follows a similar pattern as in the past (i.e., the application learning phase), it still returns two peak-hour even though the submitted query traffic is flat (see Figure 13c). The connection to the application layer addresses this problem, as demonstrated by simple scaling and component-aware scaling. While they can output utilization in a flat shape, the magnitude can still be far from the actual measurements collected to be the ground truth. For "Flat→2-peak/day" in Figure 16, resrc-aware DL performs the worst in FrontendNGINX, ComposePostService, and UserTimelineService, while component-aware scaling has the largest error in PostStorageMongoDB, showing their instability even if they are not the worst technique in other settings.

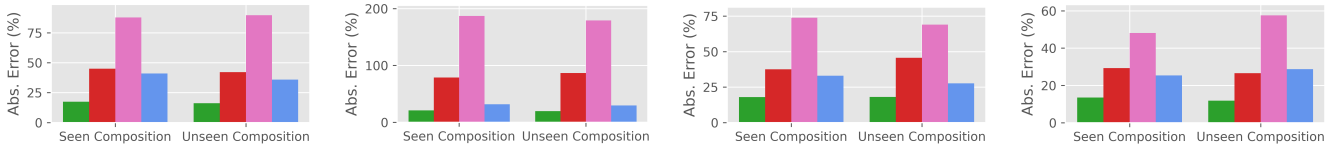
Trace Synthesizer. In this use case, the application has not received the query API traffic yet. DeepRest uses a trace synthesizer to generate synthetic traces for its feature extractor to prepare for model inputs. The high-quality estimation by DeepRest can be partially attributed to the synthesizer because it can produce traces that closely resemble the ones we will collect using distributed tracing tools if the query traffic is sent to the application. We measure the percentage accuracy of the synthesized traces by comparing them with the ground truth traces captured by running the query for evaluation purposes. Table 1 indicates that our trace synthesizer can reach an accuracy of over 91% in all six settings of three common business scenarios.

Takeaway Messages. For resource estimation system, it is essential to understand the relationship between the specific APIs, the components and their resources, especially when the query API traffic pattern differs from the one the application has been serving in the past. Only considering the historical utilization patterns (i.e., with resrc-aware DL) can result in inaccurate provisioning. Second, identifying the dependency between APIs and components (e.g., with



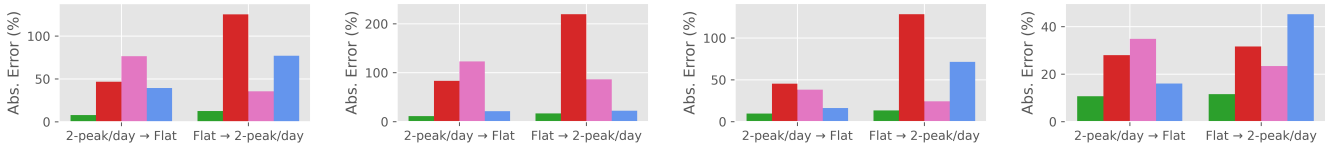
(a) FrontendNGINX **(b)** ComposePostService **(c)** UserTimelineService **(d)** PostStorageMongoDB

Figure 14. Estimating CPU utilization given query API traffic with unseen scales of application users.



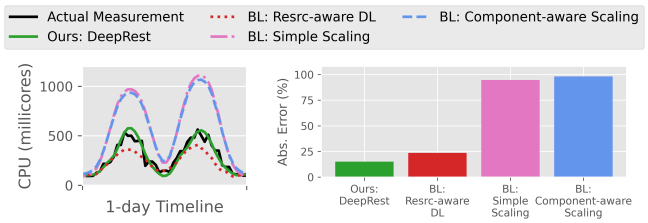
(a) FrontendNGINX **(b)** ComposePostService **(c)** UserTimelineService **(d)** PostStorageMongoDB

Figure 15. Estimating CPU utilization given query API traffic with unseen API compositions.



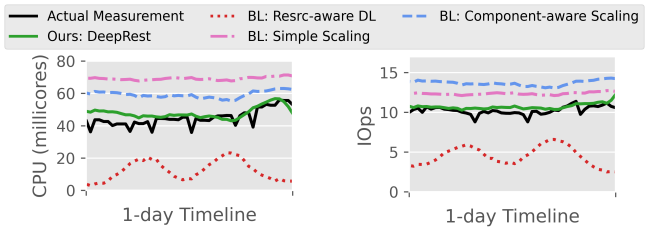
(a) FrontendNGINX **(b)** ComposePostService **(c)** UserTimelineService **(d)** PostStorageMongoDB

Figure 16. Estimating CPU utilization given query API traffic with unseen traffic shapes.



(a) FrontendService - CPU **(b)** Absolute percentage error

Figure 17. The estimation of the CPU utilization of the FrontendService in the hotel reservation system having 3× more users than ever.



(a) ComposePostService - CPU **(b)** PostStorageMongoDB - IOps

Figure 18. The estimation of two resources given the query API traffic in Figure 13c.

component-aware baseline) is not enough for accurate estimation. For example, 2× increase in the overall traffic can lead to 4× increase in the consumption of a certain resource in a component. The estimation system needs to build the knowledge of how each API consumes resource(s) on each component and estimate based on the composition of APIs in the traffic and their resource footprints.

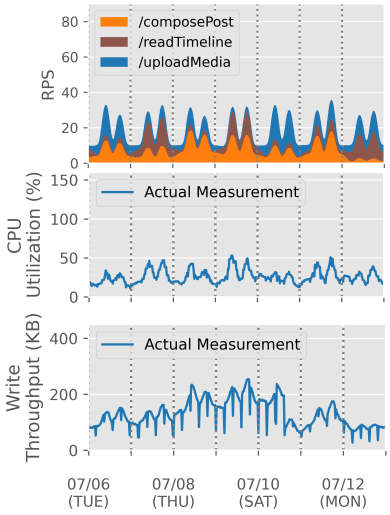
Query Scenario		Synthesis Quality (%)
Unseen Scale	1×	91.03
	2×	92.75
	3×	93.54
Unseen API Composition		91.41
Unseen Shape	2-peak/day → flat	93.22
	flat → 2-peak/day	92.24

Table 1. DeepRest synthesizes traces with over 91% accuracy.

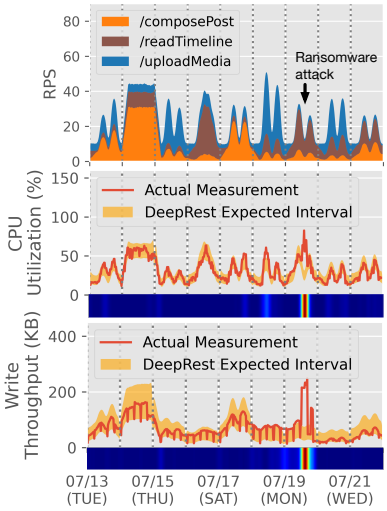
5.4 Application Sanity Check

DeepRest offers a unique opportunity to verify whether the utilized resources are justifiable by how the application is being used. We name this verification process as *application sanity check*. For sanity check, we feed the *real* API traffic and their traces received in the production environment to DeepRest, which it uses to estimate the expected resource utilization for each component. By observing the deviation between the DeepRest-expected metrics and the actual metrics, the application owner can identify potential anomalies [19]. We demonstrate the use of DeepRest in detecting two major cybersecurity threats: ransomware attacks and cryptojacking attacks [18].

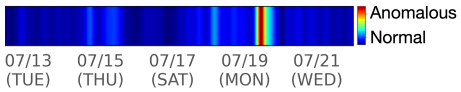
Identifying Ransomware Attacks. Consider an attacker launches ransomware attacks on the PostStorageMongoDB to encrypt post contents and ask for ransoms [6]. Like the



(a) Application Learning Phase



(b) Sanity Checking Phase



Anomalous Event #1

Time: 07/19 (MON) 12:00 - 13:30
Component: PostStorageMongoDB
Throughput: 210.2% higher than expected
CPU: 163.4% higher than expected
IOps: 31.8% higher than expected
Memory: 21.7% higher than expected
Component: FrontendNGINX
CPU: 21.1% lower than expected

(c) DeepRest examines the anomaly score of each resource in each component and combines them as an ensemble to boost the accuracy and produce the overall anomaly score (top) for generating the interpretable alert (bottom) to be sent to the application owner.

Figure 19. DeepRest (a) takes the API traffic and the corresponding resource utilization to learn the application and (b) estimate the expected utilization interval of each resource for sanity check. If the resource consumption cannot be justified by how the application was being used, (c) interpretable alerts can be generated to notify the application owner for further examination.

resource allocation, we use seven days of data from the production environment in the past for application learning (e.g., from 07/06 to 07/13). Figure 19a shows the API traffic (1st row) and gives the utilization of two example resources in the corresponding 7-day period (2nd row for CPU utilization and 3rd row for write throughput in the PostStorageMongoDB). We then send the real API traffic received during the period where the application owner suspects unwanted activities to DeepRest to conduct resource estimation. Figure 19b shows the 9-day API traffic from 07/13 to 07/22 (1st row), the CPU utilization (2nd row), and write throughput (3rd row), where the yellow region indicates the DeepRest-expected interval with $\delta = 0.90$. Given the observation from Figure 19a that the application was likely to have two peak-hour per day, manual inspection on purely the resource utilization of, e.g., CPU (the red curve in 2nd row) can result in three suspicious dates: 07/14 with constantly high utilization, 07/16 with only one peak-hour, and 07/19 also with only one peak-hour. However, our DeepRest-expected interval reveals that constantly high utilization is expected on 07/14, and exactly one peak-hour should be found on 07/16. Only the burst on 07/19 is not justified, which is correct as we launched the attack that day. As shown below each figure, we can quantify this deviation by L_2 distance and visualize it as a 1D heatmap. The red color indicates the time points where the deviation from DeepRest's expectation is significant and hence anomalous. We can further enhance the trust by triangulating with other components and resources as an ensemble to generate interpretable events [21] as shown in Figure 19c to be followed up by the application owner (e.g., selecting a recovery point for disaster recovery).

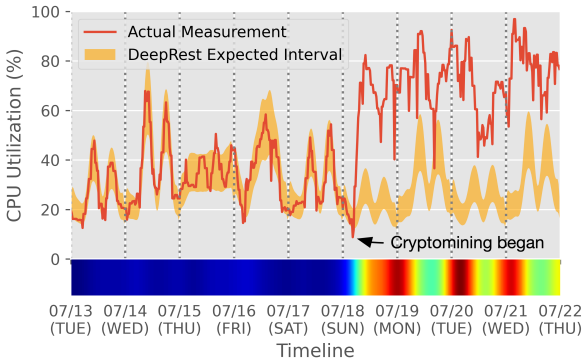


Figure 20. Using DeepRest to conduct application sanity checks to identify cryptojacking.

Identifying Cryptojacking Attacks. We also launch a cryptojacking attack by installing a process in the application to steal the resources for cryptomining. Figure 20 shows the CPU utilization of the PostStorageMongoDB, where the cryptomining began on 07/18. The actual measurements (red curve) indicate an increased CPU utilization. Still, because the application may serve more users from that day, we need DeepRest to estimate the expected interval to verify the hypothesis. Indeed, the anomaly score represented as the 1D heatmap confirms that according to the API traffic received in the corresponding period, we should expect comparatively low utilization with two peak-hour per day (yellow region).

Takeaway Messages. Violating the periodicity in utilization (e.g., from two peak-hour per day to consistently high utilization) is not always anomalous as long as it can be justified by how the application is being used w.r.t. the API traffic. Since DeepRest provides a unique insight on the causality between application and resource layers, it can be used in

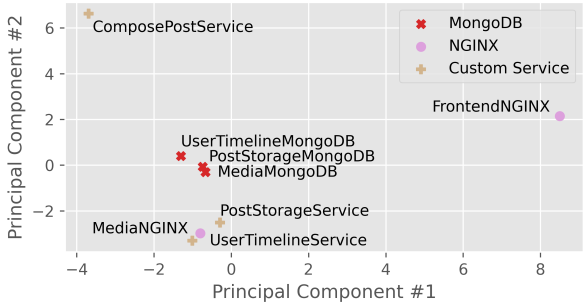


Figure 21. The DNN experts trained to estimate utilization in MongoDBs (red crosses) are similar to each other (forming a cluster).

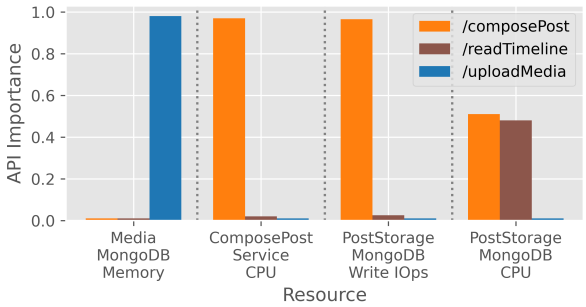


Figure 22. DeepRest neural design reveals the dependencies between resources and API endpoints.

conjunction with dedicated tools to boost the robust detection of cyberattacks [21, 23, 37] or other unwanted incidents such as memory leakage due to software bugs [16, 43, 44]. Note that eventual confirmation of an anomaly may require further analysis using specific malware scanners. However, DeepRest delivers low-overhead real-time notifications to the application owners for further investigation.

6 Discussion

Transfer Learning. We visualize the application-independent part of the DNN experts using principal component analysis (PCA), projecting the learned parameters in GRU cells onto a 2D space in Figure 21. Experts responsible for MongoDBs form a cluster, even though they are trained for different components with various roles in the social network. The similarity between those models implies that they learn to remember/forget in a similar way. Neural network training is to search for parameters iteratively. This phenomenon sheds light on initializing model parameters with pre-trained models [48] as, according to the substantial evidence from transfer learning on computer vision tasks [24, 39], convergence can be accelerated from strategically selected initial parameters, and accuracy can be improved. Such a transfer of knowledge does not only limit to the same application when it has a new component or to adapt to new behaviors over time [59] but possibly other applications [67].

Interpreting DeepRest Models. Recall in Section 4 that we introduce a trainable API-aware mask $\boldsymbol{m}$ in each expert to learn which APIs need to be emphasized for estimating the respective resource. Such a mask is valuable since it reveals which APIs are influential on a particular resource in a component. It can enable various additional use cases. For example, the application owner can identify which APIs can perform suboptimally without impacting user experience based on their domain knowledge and locate those resources only affecting them. This allows them to optimize the resource cost when necessary. Figure 22 gives four example resources by visualizing their learned API-aware mask with normalization: memory in the MediaMongoDB is affected only by `/uploadMedia`, both CPU in the ComposePostService and write IOps in the PostStorageMongoDB are influenced only by `/composePost`, and CPU in the PostStorageMongoDB is correlated to both `/composePost` and `/readTimeline`. Even though such information may be obtained from static program analysis [65], gathering the source code of all components is difficult to execute in practice. In contrast, DeepRest provides it as a byproduct by interpreting the neural network parameters trained in a data-driven manner.

Scalability. While DeepRest uses advanced deep learning techniques, it is scalable to large applications. The tracing overhead is as low as 2.6% on the 99th percentile latency [28]. In terms of the application scale, each DeepRest expert has a size of 801.5 kB, and the training time per expert is 5.4 seconds. The inference time to estimate one day of resource utilization is 1.589 milliseconds per expert. Hence, it can scale to thousands of components and offer fast inference in real-time to, e.g., provide proactive application sanity checks.

Future Work. The application owner may not need to allocate resources in serverless environments, but such a responsibility is delegated to the cloud provider. We are interested in extending DeepRest to optimize the infrastructure in serverless computing [14]. Also, DeepRest currently focuses on non-read operations as caching imposes learning challenges, which we also observe in memory consumption, leading to suboptimal estimation accuracy (2nd row in Figure 12a). We are interested in exploring approaches to enhance DeepRest in capturing such behaviors.

7 Conclusions

We have presented DeepRest, a deep resource estimation algorithm for API-driven microservices. It does not assume any internal knowledge of the application and can learn directly from resource metrics and application traces readily available in production cloud systems. DeepRest eliminates the dependency on recurring patterns and provides accurate estimation even for API traffic with unseen trends. Learning the causality between the application and resource layers also allows DeepRest to offer application sanity checks, which verify whether the utilization in the production environment is justifiable by how the application is being used w.r.t. the API traffic and identify potential anomalies such as ransomware attacks and cryptojacking.

References

[1] [n. d.]. At your service! With schedule-based autoscaling, VMs are at the ready. https://cloud.google.com/blog/products/compute/introducing-schedule-based-autoscaling-for-compute-engine. [Online; Accessed 2021/09/20].

[2] [n. d.]. AWS Auto Scaling Documentation. https://docs.aws.amazon.com/autoscaling/index.html. [Online; Accessed 2021/09/20].

[3] [n. d.]. Azure Autoscale. https://azure.microsoft.com/en-us/features/autoscale/. [Online; Accessed 2021/09/20].

[4] [n. d.]. cAdvisor - Analyzes resource usage and performance characteristics of running containers. https://github.com/google/cadvisor. [Online; Accessed 2021/09/20].

[5] [n. d.]. Google Cloud - Load balancing and scaling. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling. [Online; Accessed 2021/09/20].

[6] [n. d.]. Hacker ransoms 23k MongoDB databases and threatens to contact GDPR authorities. https://www.zdnet.com/article/hacker-ransoms-23k-mongodb-databases-and-threatens-to-contact-gdpr-authorities/. [Online; Accessed 2021/09/20].

[7] [n. d.]. IBM Cloud - Auto scale. https://cloud.ibm.com/docs/virtual-servers?topic=virtual-servers-about-auto-scale. [Online; Accessed 2021/09/20].

[8] [n. d.]. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/. [Online; Accessed 2021/09/20].

[9] [n. d.]. Kubernetes. https://kubernetes.io. [Online; Accessed 2021/09/20].

[10] [n. d.]. Locust - a modern load testing framework. https://locust.io/. [Online; Accessed 2021/09/20].

[11] [n. d.]. OpenTelemetry: An observability framework for cloud-native software. https://opentelemetry.io. [Online; Accessed 2021/09/20].

[12] [n. d.]. Prometheus - Monitoring system and time series database. https://prometheus.io/. [Online; Accessed 2021/09/20].

[13] Fahd Al-Haidari, M Sqalli, and Khaled Salah. 2013. Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, Vol. 2. IEEE, 256–261.

[14] Malay Bag, Alekh Jindal, and Hiren Patel. 2020. Towards Plan-aware Resource Allocation in Serverless Query Processing. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*.

[16] Jasmin Bogatinovski and Sasho Nedelkoski. 2020. Multi-source anomaly detection in distributed it systems. In *International Conference on Service-Oriented Computing*. Springer, 201–213.

[17] George EP Box and David A Pierce. 1970. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American statistical Association* 65, 332 (1970), 1509–1526.

[18] Domhnall Carlin, Jonah Burgess, Philip O'Kane, and Sakir Sezer. 2019. You could be mine (d): the rise of cryptojacking. *IEEE Security & Privacy* 18, 2 (2019), 16–22.

[19] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys (CSUR)* 41, 3 (2009), 1–58.

[20] Zheyi Chen, Jia Hu, Geyong Min, Albert Y Zomaya, and Tarek El-Ghazawi. 2019. Towards accurate prediction for high-dimensional and highly-variable cloud workloads with deep learning. *IEEE Transactions on Parallel and Distributed Systems* 31, 4 (2019), 923–934.

[21] Ka-Ho Chow, Umesh Deshpande, Sangeetha Seshadri, and Ling Liu. 2021. SRA: Smart Recovery Advisor for Cyber Attacks. In *Proceedings of the 2021 International Conference on Management of Data*. 2691–2695.

[22] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS Deep Learning and Representation Learning Workshop*.

[23] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. 2016. ShieldFS: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 336–347.

[24] Yin Cui, Yang Song, Chen Sun, Andrew Howard, and Serge Belongie. 2018. Large scale fine-grained categorization and domain-specific transfer learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4109–4118.

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

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

[27] Qingfeng Du, Tiandi Xie, and Yu He. 2018. Anomaly detection and diagnosis for container-based microservices with performance monitoring. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 560–572.

[28] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 135–151.

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

[30] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*. 19–33.

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

[32] Jiawei Han, Jian Pei, and Micheline Kamber. 2011. *Data mining: concepts and techniques*. Elsevier.

[33] Jordan Hochenbaum, Owen S Vallis, and Arun Kejariwal. 2017. Automatic anomaly detection in the cloud via statistical learning. *arXiv preprint arXiv:1704.07706* (2017).

[34] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5 (1989), 359–366.

[35] Waheed Iqbal, Josep Lluis Berral, David Carrera, et al. 2020. Adaptive sliding windows for improved estimation of data center resource utilization. *Future Generation Computer Systems* 104 (2020), 212–224.

[36] Waheed Iqbal, Josep Lluis Berral, Abdelkarim Erradi, David Carrera, et al. 2019. Adaptive prediction models for data center resources utilization estimation. *IEEE Transactions on Network and Service Management* 16, 4 (2019), 1681–1693.

[37] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. 2016. {UNVEIL}: A large-scale, automated approach to detecting ransomware. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 757–772.

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

[39] Simon Kornblith, Jonathon Shlens, and Quoc V Le. 2019. Do better imagenet models transfer better?. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2661–2671.

[40] Jitendra Kumar and Ashutosh Kumar Singh. 2020. Decomposition based cloud resource demand prediction using extreme learning machines. *Journal of Network and Systems Management* 28, 4 (2020), 1775–1793.

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

[42] Ming-Chang Lee, Jia-Chun Lin, and Ernst Gunnar Gran. 2020. RePAD: real-time proactive anomaly detection for time series. *arXiv preprint arXiv:2001.08922* (2020).

[43] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, et al. 2021. Practical Root Cause Localization for Microservice Systems via Trace Analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 1–10.

[44] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. 2020. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 48–58.

[45] Karl Mason, Martin Duggan, Enda Barrett, Jim Duggan, and Enda Howley. 2018. Predicting host CPU utilization in the cloud using evolutionary neural networks. *Future Generation Computer Systems* 86 (2018), 162–173.

[46] Yang Meng, Ruonan Rao, Xin Zhang, and Pei Hong. 2016. CRUPA: A container resource utilization prediction algorithm for auto-scaling based on time series analysis. In *2016 International conference on progress in informatics and computing (PIC)*. IEEE, 468–472.

[47] Valter Rogério Messias, Julio Cezar Estrella, Ricardo Ehlers, Marcos José Santana, Regina Carlucci Santana, and Stephan Reiff-Marganiec. 2016. Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure. *Neural Computing and Applications* 27, 8 (2016), 2383–2406.

[48] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.

[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[50] Issaret Prachitmutita, Wachirawit Aittinonmongkol, Nasoret Pojjanasuksakul, Montri Supattatham, and Praisan Padungweang. 2018. Auto-scaling microservices on IaaS under SLA with cost-effective framework. In *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*. IEEE, 583–588.

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

[52] Rajsimman Ravichandiran, Hadi Bannazadeh, and Alberto Leon-Garcia. 2018. Anomaly detection using resource behaviour analysis for autoscaling systems. In *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, 192–196.

[53] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

[54] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 500–507.

[55] Areeg Samir and Claus Pahl. 2019. Dla: Detecting and localizing anomalies in containerized microservice architectures using markov models. In *2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 205–213.

[56] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

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

[58] Akshitha Sriraman and Thomas F Wenisch. 2018. μ suite: a benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–12.

[59] Yu Sun, Ke Tang, Zexuan Zhu, and Xin Yao. 2018. Concept drift adaptation by exploiting historical knowledge. *IEEE transactions on neural networks and learning systems* 29, 10 (2018), 4822–4832.

[60] Salman Taherizadeh and Vlado Stankovski. 2019. Dynamic multi-level auto-scaling rules for containerized applications. *Comput. J.* 62, 2 (2019), 174–197.

[61] Nhuan Tran, Thang Nguyen, Binh Minh Nguyen, and Giang Nguyen. 2018. A multivariate fuzzy time series resource forecast model for clouds using LSTM and data correlation analysis. *Procedia Computer Science* 126 (2018), 636–645.

[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[63] Manish Verma, GR Gangadharan, Nanjangud C Narendra, Ravi Vadlamani, Vidyadhar Inamdar, Lakshmi Ramachandran, Rodrigo N Calheiros, and Rajkumar Buyya. 2016. Dynamic resource demand prediction and allocation in multi-tenant service clouds. *Concurrency and Computation: Practice and Experience* 28, 17 (2016), 4429–4442.

[64] Lingzhi Wang, Nengwen Zhao, Junjie Chen, Pinnong Li, Wenchi Zhang, and Kaixin Sui. 2020. Root-cause metric location for microservice systems via log anomaly detection. In *2020 IEEE International Conference on Web Services (ICWS)*. IEEE, 142–150.

[65] Zhongxing Xu, Ted Kremenek, and Jian Zhang. 2010. A memory model for static analysis of C programs. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 535–548.

[66] Ming Yan, XiaoMeng Liang, ZhiHui Lu, Jie Wu, and Wei Zhang. 2021. HANSEL: Adaptive horizontal scaling of microservices using Bi-LSTM. *Applied Soft Computing* 105 (2021), 107216.

[67] Xi Yan, David Acuna, and Sanja Fidler. 2020. Neural data server: A large-scale search engine for transfer learning data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3893–3902.

[68] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. 2019. MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 122–132.

[69] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, 68–75.

[70] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware resource management for cloud microservices. In *Proceedings of the 26th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems. 167–181.

[71] Giulio Zhou and Martin Maas. 2019. Multi-task learning for storage systems. In *Proc. ML Syst. Workshop*.

[72] Giulio Zhou and Martin Maas. 2021. Learning on Distributed Traces for Data Center Storage Systems. *Proceedings of Machine Learning and Systems* 3 (2021).

[73] Zheng Zhu, Rongbin Gu, ChenLing Pan, Youwei Li, Bei Zhu, and Jing Li. 2019. CPU and network traffic anomaly detection method for cloud data center. In *Proceedings of the International Conference on Advanced Information Science and System*. 1–7.