Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices

Yanqi Zhang
Cornell University
Ithaca, NY, USA
yz2297@cornell.edu

Weizhe Hua
Cornell University
Ithaca, NY, USA
wh399@cornell.edu

Zhuangzhuang Zhou
Cornell University
Ithaca, NY, USA
zz586@cornell.edu

G. Edward Suh
Cornell University
Ithaca, NY, USA
suh@ece.cornell.edu

Christina Delimitrou
Cornell University
Ithaca, NY, USA
delimitrou@cornell.edu

ABSTRACT

Cloud applications are increasingly shifting from large monolithic services, to large numbers of loosely-coupled, specialized microservices. Despite their advantages in terms of facilitating development, deployment, modularity, and isolation, microservices complicate resource management, as dependencies between them introduce backpressure effects and cascading QoS violations.

We present Sinan, a data-driven cluster manager for interactive cloud microservices that is online and QoS-aware. Sinan leverages a set of scalable and validated machine learning models to determine the performance impact of dependencies between microservices, and allocate appropriate resources per tier in a way that preserves the end-to-end tail latency target. We evaluate Sinan both on dedicated local clusters and large-scale deployments on Google Compute Engine (GCE) across representative end-to-end applications built with microservices, such as social networks and hotel reservation sites. We show that Sinan always meets QoS, while also maintaining cluster utilization high, in contrast to prior work which leads to unpredictable performance or sacrifices resource efficiency. Furthermore, the techniques in Sinan are explainable, meaning that cloud operators can yield insights from the ML models on how to better deploy and design their applications to reduce unpredictable performance.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → *Planning and scheduling*.

KEYWORDS

Cloud computing, datacenter, quality of service, tail latency, microservices, cluster management, resource management, resource allocation, resource efficiency, machine learning for systems

1 INTRODUCTION

In recent years, cloud applications have progressively shifted from *monolithic* services to graphs with hundreds of single-purpose and loosely-coupled *microservices* [1, 6, 7, 24–26, 46, 47]. This shift is becoming increasingly pervasive, with large cloud providers, such as Amazon, Twitter, Netflix, and eBay having already adopted this application model [1, 6, 7].

Despite several advantages, such as modular and flexible development and rapid iteration, microservices also introduce new system challenges, especially in resource management, since the complex topologies of microservice dependencies exacerbate queueing effects, and introduce cascading Quality of Service (QoS) violations that are difficult to identify and correct in a timely manner [26, 53]. Current cluster managers are designed for monolithic applications or applications consisting of a few pipelined tiers, and are not expressive enough to capture the complexity of microservices [17, 19, 20, 22, 26, 31–33, 35, 36, 43, 44, 51]. Given that an increasing number of production cloud services, such as EBay, Netflix, Twitter, and Amazon, are now designed as microservices, addressing their resource management challenges is a pressing need [6, 7, 26].

We take a data-driven approach to tackle the complexity microservices introduce to resource management. Similar machine learning (ML)-driven approaches have been effective at solving resource management problems for large-scale systems in previous work [14, 16–18, 18, 19, 21, 42]. Unfortunately, these systems are not directly applicable to microservices, as they were designed for monolithic services, and hence do not account for the impact of dependencies between microservices on end-to-end performance.

We present Sinan, a scalable and QoS-aware resource manager for interactive cloud microservices. Instead of tasking the user or cloud operator with inferring the impact of dependencies between microservices, Sinan leverages a set of validated ML models to automatically determine the impact of per-tier resource allocations

on end-to-end performance, and assign appropriate resources to each tier.

Sinan first uses an efficient space exploration algorithm to examine the space of possible resource allocations, especially focusing on corner cases that introduce QoS violations. This yields a training dataset used to train two models: a Convolutional Neural Network (CNN) model for detailed short-term performance prediction, and a Boosted Trees model that evaluates the long-term performance evolution. The combination of the two models allows Sinan to both examine the near-future outcome of a resource allocation, and to account for the system's inertia in building up queues with higher accuracy than a single model examining both time windows. Sinan operates online, adjusting per-tier resources dynamically according to the service's runtime status and end-to-end QoS target. Finally, Sinan is implemented as a centralized resource manager with global visibility into the cluster and application state, and with per-node resource agents that track per-tier performance and resource utilization.

We evaluate Sinan using two end-to-end applications from Death-StarBench [26], built with interactive microservices: a social network and a hotel reservation site. We compare Sinan against both traditionally-employed empirical approaches, such as autoscaling [4], and previous research on multi-tier service scheduling based on queueing analysis, such as PowerChief [52]. We demonstrate that Sinan outperforms previous work both in terms of performance and resource efficiency, successfully meeting QoS for both applications under diverse load patterns. On the simpler hotel reservation application, Sinan saves *25.9%* on average, and up to *46.0%* of the amount of resources used by other QoS-meeting methods. On the more complex social network service, where abstracting application complexity is more essential, Sinan saves *59.0%* of resources on average, and up to *68.1%*, essentially accommodating twice the amount of requests per second, without the need for more resources. We also validate Sinan's scalability through large-scale experiments on approximately 100 container instances on Google Compute Engine (GCE), and demonstrate that the models deployed on the local cluster can be reused on GCE with only minor adjustments instead of retraining.

Finally, we demonstrate the explainability benefits of Sinan's models, delving into the insights they can provide for the design of large-scale systems. Specifically, we use an example of Redis's log synchronization, which Sinan helped identify as the source of unpredictable performance out of tens of dependent microservices to show that the system can offer practical and insightful solutions for clusters whose scale make previous empirical approaches impractical.

2 OVERVIEW

2.1 Problem Statement

Sinan aims to manage resources for complex, interactive microservices with tail latency QoS constraints in a scalable and resource-efficient manner. Graphs of dependent microservices typically include tens to hundreds of tiers, each with different resource requirements, scaled out and replicated for performance and reliability. Section 2.2 describes some motivating examples of such services

with diverse functionality used in this work; other similar examples can be found in [1, 6, 7, 46].

Most cluster managers focus on CPU and memory management [14, 42, 51]. Microservices are by design mostly stateless, hence their performance is defined by their CPU allocation. Given this, Sinan primarily focuses on allocating CPU resources to each tier [26], both at sub-core and multi-core granularity, leveraging Linux cgroups through the Docker API [2]. We also provision each tier with the maximum profiled memory usage to eliminate out of memory errors.

2.2 Motivating Applications

We use two end-to-end interactive applications from DeathStar-Bench [26]: a hotel reservation service, and a social network.

2.2.1 Hotel Reservation. The service is an online hotel reservation site, whose architecture is shown in Figure 1.
Functionality: The service supports searching for hotels using geolocation, placing reservations, and getting recommendations. It is implemented in Go, and tiers communicate over gRPC [5]. Data backends are implemented in memcached for in-memory caching, and MongoDB, for persistent storage. The database is populated with 80 hotels and 500 active users.
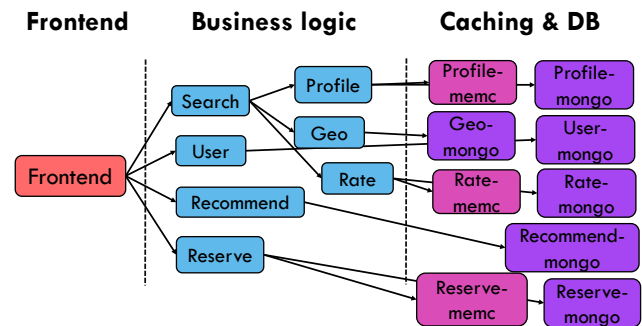

Figure 1: Hotel reservation microservice architecture [26]. Client requests first reach a front-end webserver, and, depending on the type of requests, are then directed to logic tiers implementing functionality for searching hotels, completing hotel reservations, and getting recommendations on available hotels. At the right-most of the figure, the requests reach the back-end databases, implemented both with in-memory caching tiers (memcached), and persistent databases (MongoDB).

2.2.2 Social Network. The end-to-end service implements a broadcast style social network with uni-directional follow relationships, shown in Figure 2. Inter-microservice messages use Apache Thrift RPCs [48].
Functionality: Users can create posts embedded with text, media, links, and tags to other users, which are then broadcasted to all their followers. The texts and images uploaded by users, specifically, go through image-filter (a CNN classifier) and text-filter services (an SVM classifier), and contents violating the service's ethics guidelines are rejected. Users can also read posts on their timelines. We use the Reed98 [41] social friendship network to populate the user

database. User activity follows the behavior of Twitter users reported in [30], and the distribution of post text length emulates Twitter's text length distribution [28].
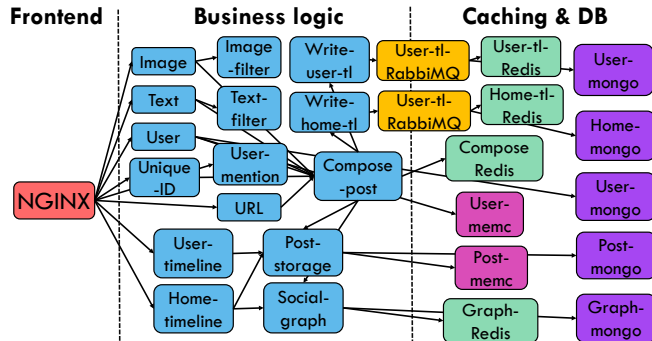

Figure 2: Social Network microservice architecture [26]. Client requests first reach Nginx, which works as frontend http servers. Then, depending on the type of user request, a number of logic, mid-tiers will be invoked to create a post, read a user's timeline and to follow/unfollow users. At the right-most of the figure, the requests reach the backend databases, implemented both with in-memory caching tiers (memcached and Redis), and persistent databases (MongoDB).

2.3 Management Challenges & the Need for ML

Resource management in microservices faces four challenges.

1. Dependencies among tiers Resource management in microservices is additionally complicated by the fact that dependent microservices are not perfect pipelines, and hence can introduce backpressure effects that are hard to detect and prevent [26, 53]. These dependencies can be further exacerbated by the specific RPC and data store API implementation. Therefore, the resource scheduler should have a global view of the microservice graph and be able to anticipate the impact of dependencies on end-to-end performance.

2. System complexity Given that application behaviors change frequently, resource management decisions need to happen online. This means that the resource manager must traverse a space that includes all possible resource allocations per microservice in a practical manner. Prior empirical approaches use resource utilization [4], or latency measurements [11, 19, 32] to drive allocation decisions. Queueing approaches similarly characterize the system state using queue lengths [52]. Unfortunately these approaches cannot be directly employed in complex microservices with tens of dependent tiers. First, microservice dependencies mean that resource usage across tiers is codependent, so examining fluctuations in individual tiers can attribute poor performance to the wrong tier. Similarly, although queue lengths are accurate indicators of a microservice's system state, obtaining exact queue lengths is hard. First, queues exist across the system stack from the NIC and OS, to the network stack and application. Accurately tracking queue lengths requires application changes and heavy instrumentation, which can negatively impact performance and/or is not possible in public clouds.

Second, the application may include third-party software whose source code cannot be instrumented. Alternatively, expecting the user to express each tier's resource sensitivity is problematic, as users already face difficulties correctly reserving resources for simple, monolithic workloads, leading to well-documented underutilization [19, 39], and the impact of microservice dependencies is especially hard to assess, even for expert developers.

3. Delayed queueing effect Consider a queueing system with processing throughput T_o under a latency QoS target, like the one in Figure 3. T_o is a non-decreasing function of the amount of allocated resources R. For input load T_i, T_o should equal or slightly surpass T_i for the system to stably meet QoS, while using the minimum amount of resources R needed. Even when R is reduced, such that $T_o < T_i$, QoS will not be immediately violated, since queue accumulation takes time.

The converse is also true; by the time QoS is violated, the built-up queue takes a long time to drain, even if resources are upscaled immediately upon detecting the violation (red line). Multi-tier microservices are complex queueing systems with queues both across and within microservices. This delayed queueing effect highlights the need for ML to evaluate the long-term impact of resource allocations, and to proactively prevent the resource manager from re-
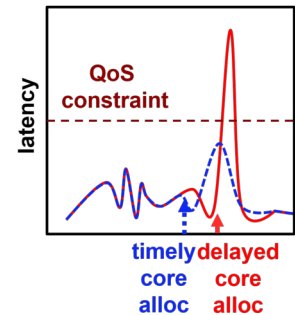

Figure 3: The figure showcases the delayed queueing effect in microservices; QoS violations that are not detected eagerly (blue line), become unavoidable (red), even if later action is taken.

ducing resources too aggressively, to avoid latency spikes with long recovery periods. To mitigate a QoS violation, the manager must increase resources proactively (blue line), otherwise the violation becomes unavoidable, even if more resources are allocated a posteriori.

4. Boundaries of resource allocation space Data collection or profiling are essential to the performance of any model. Given the large resource allocation space in microservices, it is essential for any resource manager to quickly identify the boundaries of that space that allow the service to meet its QoS, with the minimum resource amount [23], so that neither performance nor resource efficiency are sacrificed. Prior work often uses random exploration of the resource space [11, 19, 32] or uses prior system state as the training dataset [25]. Unfortunately, while these approaches work for simpler applications, in microservices they are prone to covariant shift. Random collection blindly explores the entire space, even though many of the explored points may never occur during the system's normal operation, and may not contain any points close to the resource boundary of the service. On the contrary, data from operation logs are biased towards regions that occur

frequently in practice but similarly may not include points close to the boundary, as cloud systems often overprovision resources to ensure that QoS is met. To reduce exploration overheads it is essential for a cluster manager to efficiently examine the necessary and sufficient number of points in the resource space that allow it to *just* meet QoS with the minimum resources.

2.4 Proposed Approach

These challenges suggest that empirical resource management, such as autoscaling [4] or queueing analysis-based approaches for multi-stage applications, such as PowerChief [52], are prone to unpredictable performance and/or resource inefficiencies. To tackle these challenges, we take a data-driven approach that abstracts away the complexity of microservices from the user, and leverages ML to identify the impact of dependencies on end-to-end performance, and make allocation decisions. We also design an efficient space exploration algorithm that explores the resource allocation space, especially boundary regions that may introduce QoS violations, for different application scenarios. Specifically, Sinan's ML models predict the end-to-end latency and the probability of a QoS violation for a resource configuration, given the system's state and history. The system uses these predictions to maximize resource efficiency, while meeting QoS.

At a high level, the workflow of Sinan is as follows: the data collection agent collects training data, using a carefully-designed algorithm which addresses Challenge 4 (efficiently exploring the resource space). With the collected data, Sinan trains two ML models: a convolution neural network (CNN) model and a boosted trees (BT) model. The CNN handles Challenges 1 and 2 (dependencies between tiers and navigating the system complexity), by predicting the end-to-end tail latency in the near future. The BT model addresses Challenge 3 (delayed queueing effect), by evaluating the probability for a QoS violation further into the future, to account for the system's inertia in building up queues. At runtime, Sinan infers the instantaneous tail latency and the probability for an upcoming QoS violation, and adjusts resources accordingly to satisfy the QoS constraint. If the application or underlying system change at any point in time, Sinan retrains the corresponding models to account for the impact of these changes on end-to-end performance.

3 MACHINE LEARNING MODELS

The objective of Sinan's ML models is to accurately predict the performance of the application given a certain resource allocation. The scheduler can then query the model with possible resource allocations for each microservice, and select the one that meets QoS with the least necessary resources.

A straightforward way to achieve this is designing an ML model that predicts the immediate end-to-end tail latency as a function of resource allocations and utilization, since QoS is defined in terms of latency, and comparing the predicted latency to the measured latency during deployment is straightforward. The caveat of this approach is the delayed queueing effect described in Sec. 2.3, whereby the impact of an allocation decision would only show up in performance later. As a resolution, we experimented with training a neural network (NN) to predict latency distributions over a future time window: for example, the latency for each second over the

next five seconds. However, we found that the prediction accuracy rapidly decreased the further into the future the NN tried to predict, as predictions were based only on the collected current and past metrics (resource utilization and latency), which were accurate enough for immediate-future predictions, but were insufficient to capture how dependencies between microservices would cause performance to evolve later on.

Considering the difficulty of predicting latency further into the future, we set an alternative goal: predict the latency of the immediate future, such that imminent QoS violations are identified quickly, but only predict the probability of experiencing a QoS violation later on, instead of the exact latency of each decision interval. This binary classification is a much more contained problem than detailed latency prediction, and still conveys enough information to the resource manager on performance events, e.g., QoS violations, that may require immediate action in the present.
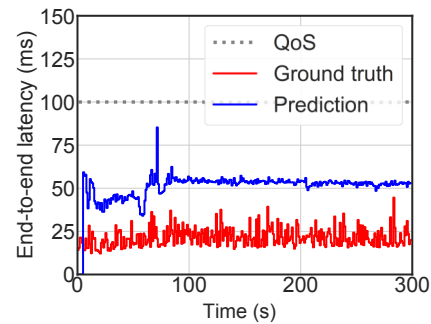

Figure 4: Multi-task NN overpredicts Social Network latency, due to the semantic gap between the QoS violation probability, a value between 0 and 1, and the latency, a value that is not strictly bounded.

An intuitive method for this are multi-task learning NNs that predict the latency of the next interval, and the QoS violation probability in the next few intervals. However, the multi-task NN considerably overpredicts tail latency and QoS violation probability, as shown in Figure 4. Note that the gap between prediction and ground truth does not indicate a constant difference, which could be easily learned by NNs with strong overfitting capabilities. We attribute the overestimation to interference caused by the semantic gap between the QoS violation probability, a value between 0 and 1, and the latency, a value that is not strictly bounded.

To address this, we designed a two-stage model: first, a CNN that predicts the end-to-end latency of the next timestep with high accuracy, and, second, a Boosted Trees (BT) model that estimates the probability for QoS violations further into the future, using the latent variable extracted by CNN. BT is generally less prone to overfitting than CNNs, since it has much fewer tunable hyperparameters than NNs; mainly the number of trees and tree depth. By using two separate models, Sinan is able to optimize each model for the respective objective, and avoid the overprediction issue of using a joint, expensive model for both tasks. We refer to the CNN model as the *short-term latency predictor*, and the BT model as the *long-term violation predictor*.

3.1 Latency Predictor

As discussed in Section 2.3, the CNN needs to account for both the dependencies across microservices, and the timeseries pattern of resource usage and application performance. Thus, both the application topology and the timeseries information are encoded in the input of the CNN. The input of the CNN includes the following three parts:

(1) an "image" (3D tensor) consisting of per-tier resource utilization within a past time window. The y-axis of the "image" corresponds to different microservices, with consecutive tiers in adjacent rows, the x-axis corresponds to the timeseries, with one timestep per column, and the z-axis (channels) corresponds to resource metrics of different tiers, including CPU usage, memory usage (resident set size and cache memory size) and network usage (number of received and sent packets), which are all retrieved from Docker's cgroup interface. Per-request tracing is not required.
(2) a matrix of the end-to-end latency distribution within the past time window, and
(3) the examined resource configuration for the next timestep, which is also encoded as a matrix.

In each convolutional (Conv) layer of the CNN, a convolutional kernel ($k \times k$ window) processes information of k adjacent tiers within a time window containing k timestamps. The first few Conv layers in the CNN can thus infer the dependencies of their adjacent tiers over a short time window, and later layers observe the entire graph, and learn interactions across all tiers within the entire time window of interest. The latent representations derived by the convolution layers are then post-processed together with the latency and resource configuration information, through concatenation and fully-connected (FC) layers to derive the latency predictions. In the remainder of this section, we first discuss the details of the network architecture, and then introduce a custom loss function that improves the prediction accuracy by focusing on the most important latency range.

As shown in Figure 5, the latency predictor takes as input the resource usage history (X_{RH}), the latency history (X_{LH}), and the resource allocation under consideration for the next timestep (X_{RC}), and predicts the end-to-end tail latencies (y_L) (95th to 99th percentiles) of the next timestep.

X_{RH} is a 3D tensor whose x-axis is the N tiers in the microservices graph, the y-axis is T timestamps ($T > 1$ accounts for the non-Markovian nature of microservice graph), and channels are F resource usage information related to CPU and memory. The set of necessary and sufficient resource metrics is narrowed down via feature selection. X_{RC} and X_{LH} are 2D matrices. For X_{RC}, the x-axis is the N tiers and the y-axis the CPU limit. For X_{RH}, the x-axis is T timestamps, and the y-axis are vectors of different latency percentiles (95th to 99th). The three inputs are individually processed with Conv and FC layers, and then concatenated to form the latent representation L_f, from which the predicted tail latencies L_f are derived with another FC layer.
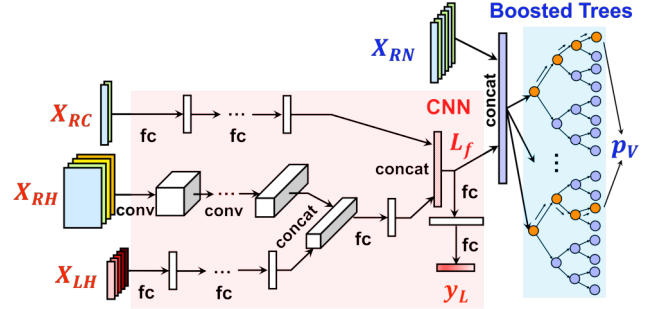

Figure 5: Sinan's hybrid model, consisting of a CNN and a Boosted Trees (BT) model. The CNN extracts the latent variable (L_f) and predicts the end-to-end latency (y_L). The BT take the latent variable and proposed resource allocation, and predicts the probability of a QoS violation (p_V).

The CNN minimizes the difference between predicted and actual latency, using the squared loss function below:

$$\mathcal{L}(X, \hat{y}, W) = \sum_{i}^{n} (\hat{y}_i - f_W(x_i))^2 \qquad (1)$$

where $f_W(\cdot)$ represents the forward function of the CNN, $\hat{y}$ is the ground truth, and n is the number of training samples. Given the spiking behavior of interactive microservices that leads to very high latency, the squared loss in Eq. 1 tends to overfit for training samples with large end-to-end latency, leading to latency overestimation in deployment. Since the latency predictor aims to find the best resource allocation within a tail latency QoS target, the loss should be biased towards training samples whose end-to-end latencies are $\leq QoS$. Therefore, we use a scaling function to scale both the predicted and actual end-to-end latency before applying the squared loss function. The scaling function ($\phi(\cdot)$) is:

$$\phi(x) = \begin{cases} x & x \leq t \\ t + \frac{x-t}{1+\alpha(x-t)} & x > t \end{cases} \qquad (2)$$

where the latency range is $(0, t)$, and the hyper-parameter α can be tuned for different decay effects. Figure 7 shows the scaling function with $t = 100$ and $\alpha = 0.005, 0.01, 0.02$. It is worth mentioning that scaling end-to-end latencies only mitigates overfitting of the predicted latency for the next decision interval, and does not improve predictions further into the future, as described above. We implement all CNN models using MxNet [13], and trained them with Stochastic Gradient Descent (SGD).

3.2 Violation Predictor

The violation predictor addresses the binary classification task of predicting whether a given allocation will cause a QoS violation further in the future, to filter out undesirable actions. Ensemble methods are good candidates as they are less prone to overfitting. We use Boosted Trees [34], which realizes an accurate non-linear model by combining a series of simple regression trees. It models the target as the sum of trees, each of which maps features to a score. The final prediction is determined by accumulating scores across all trees.

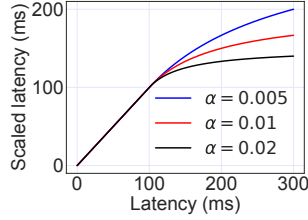| Param	Definition
k	future timesteps in BT
T	past timesteps in CNN&BT
N	application tiers
M	latency percentiles
F	resource statistics
R	allocated resources

Figure 6: ML model parameters.

Figure 7: Scale function $\phi(\cdot)$ with different k.

To further reduce the computational cost and memory footprint of Boosted Trees, we reuse the compact latent variable L_f extracted from the CNN as its input. Moreover, since the latent variable L_f is significantly smaller than X_{RC}, X_{RH}, and X_{LH} in dimensionality, using L_f as the input also makes the model more resistant to overfitting.

Boosted Trees also takes resource allocations as input. During inference, we simply use the same resource configuration for the next k timesteps to predict whether it will cause a QoS violation k steps in the future. As shown in Figure 5, each tree leaf represents either a violation or a non-violation with a continuous score. For a given example, we sum the scores for all chosen violation (s_V) and non-violation leaves (s_V) from each tree. The output of BT is the predicted probability of QoS violation (p_V), which can be calculated as $p_V = \frac{e^{s_V}}{e^{s_V} + e^{s_{NV}}}$. For the violation predictor we leverage XGBoost [12], a gradient tree boosting framework that improves scalability using sparsity-aware approximate split finding.

We first train the CNN and then BT using the extracted latent variable from the CNN. The CNN parameters (number of layers, channels per layer, weight decay etc.) and XGBoost (max tree depth) are selected based on the validation accuracy.

4 SYSTEM DESIGN

We first introduce Sinan's overall architecture, and then discuss the data collection process, which is crucial to the effectiveness of the ML models, and Sinan's online scheduler.

4.1 System Architecture

Sinan consists of three components: a centralized scheduler, distributed operators deployed on each server/VM, and a prediction service that hosts the ML models. Figure 8 shows an overview of Sinan's architecture.
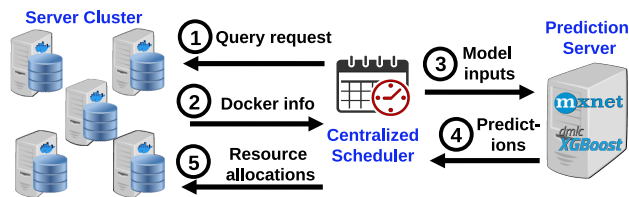

Figure 8: Sinan's system architecture. As user requests are being received, Sinan collects resource and performance metrics through Docker and Jaeger, inputs the collected metrics to the ML models, and uses the models' output to accordingly allocate resources for each tier. Allocation decisions are re-evaluated periodically online.

Sinan makes decisions periodically. In each 1s decision interval (consistent with the granularity at which QoS is defined), the centralized scheduler queries the distributed operators to obtain the CPU, memory, and network utilization of each tier in the previous interval. Resource usage is obtained from Docker's monitoring infrastructure, and only involves a few file reads, incurring negligible overheads. Aside from per-tier information, the scheduler also queries the API gateway to get user load statistics from the workload generator. The scheduler sends this data to the hybrid ML model, which is responsible for evaluating the impact of different resource allocations. Resource usage across replicas of the same tier are averaged before being used as inputs to the models. Based on the model's output, Sinan chooses an allocation vector that meets QoS using the least necessary resources, and communicates its decision to the per-node agents for enforcement.

Sinan focuses on compute resources, which are most impactful to microservice performance. Sinan explores sub-core allocations in addition to allocating multiple cores per microservice to avoid resource inefficiencies for non-resource demanding tiers, and enable denser colocation.

4.2 Resource Allocation Space Exploration

Representative training data is key to the accuracy of any ML model. Ideally, test data encountered during online deployment should follow the same distribution as the training dataset, so that covariate shift is avoided. Specifically for our problem, the training dataset needs to cover a sufficient spectrum of application behaviors that are likely to occur during online deployment. Because Sinan tries to meet QoS without sacrificing resource efficiency, it must efficiently explore the boundary of the resource allocation space, where points using the minimum amount of resources under QoS reside. We design the data collection algorithm as a multi-armed bandit process [27], where each tier is an independent arm, with the goal of maximizing the knowledge of the relationship between resources and end-to-end QoS.

The data collection algorithm approximates the running state of the application with a tuple (rps, lat_{cur}, lat_{diff}), where rps is the input requests per second, lat_{cur} is the current tail latency, and lat_{diff} is the tail latency difference from the previous interval, to capture the rate of consuming or accumulating queues. Every tier is considered as an arm that can be played independently, by adjusting its allocated resources. For each tier, we approximate the mapping between its resources and the end-to-end QoS as a Bernoulli distribution, with probability p of meeting the end-to-end QoS, and we define our information gain from assigning certain amount of resources to a tier, as the expected reduction of confidence interval of p for the corresponding Bernoulli distribution. At each step for every tier, we select the operation that maximizes the information gain, as shown in Eq. 3, where op_T^s is an action selected for tier T at running state s, n are the samples collected for the resulting resource assignment after applying op on tier T at state s, p is the previously-estimated probability of meeting QoS, and p_+ and p_- are the newly-estimated probabilities of meeting QoS, when the new sample meets or violates QoS respectively. Each operation's score is multiplied by a predefined coefficient C_{op} to encourage meeting QoS and reducing overprovisioning.

$$op_T^s = \arg\max_{op} C_{op} \cdot (\sqrt{\frac{p(1-p)}{n}} - p\sqrt{\frac{p_+(1-p_+)}{n+1}}$$
$$-(1-p)\sqrt{\frac{p_-(1-p_-)}{n+1}}) \quad (3)$$

By choosing operations that maximize Equation. 3, the data collection algorithm is incentivized to explore the boundary points that meet QoS with the minimum resource amount, since exploring allocations that definitely meet or violate QoS (with $p = 1$ or $p = 0$) has at most 0 information gain. Instead, the algorithm prioritizes exploring resource allocations whose impact on QoS is nondeterministic, like those with $p = 0.5$. It is also worth noting that the state encoding and information gain definition are simplified approximations of the actual system, with the sole purpose of containing the exploration process in the region of interest. Eventually, we rely on ML to extract the state representation that incorporates inter-tier dependencies in the microservice graph.

To prune the action space, Sinan enforces a few rules on both data collection and online scheduling. First, the scheduler is only allowed to select out of a predefined set of operations. Specifically in our setting, the operations include reducing or increasing the CPU allocation by 0.2 up to 1.0 CPU, and increasing or reducing the total CPU allocation of a service by 10% or 30%. These ratios are selected according to the AWS step scaling tutorial [4]; as long as the granularity of CPU allocations does not change, other resource ratios also work without retraining the model. Second, an upper limit on CPU utilization is enforced on each tier, to avoid overly aggressive resource downsizing that can lead to long queues and dropped requests. Third, when end-to-end tail latency exceeds the expected value, Sinan disables resource reclamations so that the system can recover as soon as possible. A subtle difference from online deployment is that the data collection algorithm explores resource allocations in the $[0, QoS + \alpha]$ tail latency region, where α is a small value compared to QoS. The extra α allows the data collection process to explore allocations that cause slight QoS violations without the pressure of reverting to states that meet QoS immediately, such that the ML models are aware of boundary cases, and avoid them in deployment. In our setting α is 20% of QoS empirically, to adequately explore the allocation space, without causing the tail latency distribution to deviate too much from values that would be seen in deployment. Collecting data exclusively when the system operates nominally, or randomly exploring the allocation space does not fulfill these requirements.

Figure 9 shows the latency distribution in the training dataset, and how the training and validation error of the model changes with respect to the latency range observed in the training dataset, for the Social Network application. In the second figure, the x-axis is the latency of samples in the training dataset, the left y-axis is the root mean squared error RMSE of the CNN, and the right y-axis represents the classification error rate of XGBoost. Each point's y-axis value is the model's training and validation error when trained only with data whose latency is smaller than the corresponding x-value. If the training dataset does not include any samples that violate QoS (500ms), both the CNN and XGBoost experience serious overfitting, greatly mispredicting latencies and QoS violations.

Figure 10 shows data collected using data collection mechanisms that do not curate the dataset's distribution. Specifically, we show
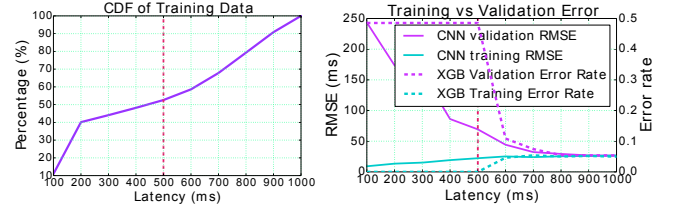

Figure 9: Training dataset latency distribution and ML training vs. validation error with respect to dataset latency range. The training dataset includes an approximately balanced set of samples between those that preserve and those that violate QoS. If the training dataset does not include any samples that violate QoS (500ms), both the CNN and XGBoost experience serious overfitting, greatly mispredicting latencies and QoS violations.

the prediction accuracy when the training dataset is collected when autoscaling is in place (a common resource management scheme in most clouds), and when resource allocations are explored randomly. As expected, when using autoscaling, the model does not see enough cases that violate QoS, and hence seriously underestimates latency and causes large spikes in tail latency, forcing the scheduler to use all available resources to prevent further violations. On the other hand, when the model is trained using random profiling, it constantly overestimates latency and prohibits any resource reduction, highlighting the importance of jointly designing the data collection algorithms and the ML models.

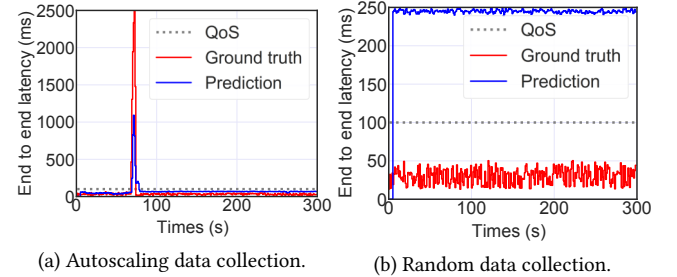(a) Autoscaling data collection.   (b) Random data collection.

Figure 10: Comparison of predicted and true latency with (a) autoscaling and (b) random data collection schemes. When using autoscaling, the model significantly underestimates latency due to insufficient training samples of QoS violations, and causes large spikes in tail latency, forcing the scheduler to use all available resources to prevent further violations. On the other hand, when the model is trained using random profiling, it constantly overestimates latency and prohibits any resource reduction, leading to resource overprovisioning.

Incremental and Transfer Learning: Incremental retraining can be applied to accommodate changes to the deployment strategy or microservice updates. In deployment, retraining can be triggered periodically in the background or when prediction accuracy drops below expected thresholds. In cases where the topology of the microservice graph is not impacted, such as hardware updates and

Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou

Table 1: Resource allocation actions in Sinan.

Category	Actions
Scale Down	Reduce CPU limit of 1 tier
Scale Down Batch	Reduce CPU limit of k least utilized tiers, $(1 < k \leq N)$
Hold	Keep current resource allocation
Scale Up	Increase CPU limit of 1 tier
Scale Up All	Increase CPU limit of all tiers
Scale Up Victim	Increase CPU limit of recent victim tiers, that are scaled down in previous t cycles

change of public cloud provider, transfer learning techniques such as fine tune can be used to train the ML models in the background with newly collected data. If the topology is changed, the CNN needs to be modified to account for removed and newly-added tiers.

Additional resources: Sinan can be extended to other system resources. Several resources, such as network bandwidth and memory capacity act like thresholds, below which performance degrades dramatically, e.g., network bandwidth [11], or the application experiences out of memory errors, and can be managed with much simpler models, like setting fixed thresholds for memory usage, or scaling proportionally with respect to user load for network bandwidth.

4.3 Online Scheduler

During deployment, the scheduler evaluates resource allocations using the ML models, and selects appropriate allocations that meet the end-to-end QoS without overprovisioning.

Evaluating all potential resource allocations online would be prohibitively expensive, especially for complex microservice topologies. Instead, the scheduler evaluates a subset of allocations following the set of heuristics shown in Table 1. For scaling down operations, the scheduler evaluates reducing CPU allocations of single tiers, and batches of tiers, e.g., scaling down the k tiers with lowest cpu utilization, $1 < k \leq N$, N being the number of tiers in the microservice graph. When scaling up is needed, the scheduler examines the impact of scaling up single tiers, all tiers, or the set of tiers that were scaled down in the past t decision intervals, $1 < t < T$ with T chosen empirically. Finally, the scheduler also evaluates the impact of maintaining the current resource assignment.

The scheduler first excludes operations whose predicted tail latency is higher than $QoS - RMSE_{valid}$. Then it uses the predicted violation probability to filter out risky operations, with two user-defined thresholds, p_d and p_u ($p_d < p_u$). These thresholds are similar to those used in autoscaling, where the lower threshold triggers scaling down and the higher threshold scaling up; the region between the two thresholds denotes stable operation, where the current resource assignment is kept. Specifically, when the violation probability of holding the current assignment is smaller than p_u, the operation is considered acceptable. Similarly, if there exists a scale down operation with violation probability lower than p_d, the scale down operation is also considered acceptable. When

the violation probability of the hold operation is larger than p_u, only scaling up operations with violation probabilities smaller than p_u are acceptable; if no such actions exist, all tiers are scaled up to their max amount. We set p_u such that the validation study's false negatives are no greater than 1% to eliminate QoS violations, and p_d to a value smaller than p_u that favors stable resource allocations, so that resources do not fluctuate too frequently unless there are significant fluctuations in utilization and/or user demand. Among all acceptable operations, the scheduler selects the one requiring the least resources.

The scheduler also has a safety mechanism for cases where the ML model's predicted latency or QoS violation probability deviate significantly from the ground truth. If a mispredicted QoS violation occurs, Sinan immediately upscales the resources of all tiers. Additionally, given a trust threshold for the model, whenever the number of latency prediction errors or missed QoS violations exceeds the thresholds, the scheduler reduces its trust in the model, and becomes more conservative when reclaiming resources. In practice, Sinan never had to lower its trust to the ML model.

5 EVALUATION

We first evaluate Sinan's accuracy, and training and inference time, and compare it to other ML approaches. Second, we deploy Sinan on our local cluster, and compare it against autoscaling [4], a widely-deployed empirical technique to manage resources in production clouds, and PowerChief [52], a resource manager for multi-stage applications that uses queueing analysis. Third, we show the incremental retraining overheads of Sinan. Fourth, we evaluate Sinan's scalability on a large-scale Google Compute Engine (GCE) cluster. Finally, we discuss how interpretable ML can improve the management of cloud systems.

5.1 Methodology

Benchmarks: We use the Hotel Reservation and Social Network benchmarks described in Section 2.2. QoS targets are set with respect to 99% end-to-end latency, 200ms for `Hotel Reservation`, and 500ms for `Social Network`.

Deployment: Services are deployed with Docker Swarm, with one microservices per container for deployment ease. *Locust* [3] is used as the workload generator for all experiments.

Local cluster: The cluster has four 80-core servers, with 256GB of RAM each. We collected 31302 and 58499 samples for Hotel Reservation and Social Network respectively, using our data collection process, and split them into training and validation sets with a 9:1 ratio, after random shuffling. The data collection agent runs for 16 hours and 8.7 hours for Social Network and Hotel Reservation respectively, and collecting more training samples do not further improve accuracy.

GCE cluster: We use 93 container instances on Google Compute Engine (GCE) to run Social Network, with several replicas per microservice tier. 5900 extra training samples are collected on GCE for the transfer learning.

5.2 Sinan's Accuracy and Speed

Table 2 compares the short-term ML model in Sinan (CNN) against a multilayer perceptron (MLP), and a long short-term memory

Table 2: RMSE, model size, and performance for three NNs — Batch size is 2048. Initial learning rates for MLP, LSTM, and CNN are 0.0001, 0.0005, and 0.001, respectively. All models are trained with a single NVidia Titan Xp.

Apps	Models	Train &Val. RMSE (ms)		Model size (KB)	Train & Inference speed (ms/batch)	
Hotel Reservation	MLP	17.8	18.9	1433	1.9	3.7
	LSTM	17.7	18.1	384	1.3	3.2
	CNN	14.2	14.7	68	4.5	3.5
Social Network	MLP	32.3	34.4	4300	6.4	5.9
	LSTM	29.3	30.7	404	4.5	5.6
	CNN	25.9	26.4	144	16.0	5.7

Table 3: The accuracy, number of trees, and total training time of Boosted Trees using a single NVidia Titan Xp.

Apps	Train & Val. accuracy (%)		Val. false pos. & neg.		# of trees	Total train time (s)
Hotel Reservation	94.4	94.1	3.2	3.1	229	2.3
Social Network	95.5	94.6	3.4	2.0	239	6.5

(LSTM) network, which is traditionally geared towards timeseries predictions. We rearrange the system history X_{RH} to be a 2D tensor with shape $T \times (F * N)$, and a 1D vector with shape $T * F * N$ for the LSTM and MLP models, respectively. To configure each network's parameters, we increase the number of fully-connected, LSTM, and convolutional layers, as well as the number of channels in each layer for the MLP, LSTM, and Sinan (CNN), until accuracy levels off. Sinan's CNN achieves the lowest RMSE, with the smallest model size. Although the CNN is slightly slower than the LSTM, its inference latency is within 1% of the decision interval (1s), which does not delay online decisions.

Table 3 shows a similar validation study for the Boosted Trees model. Specifically, we quantify the accuracy of anticipating a QoS violation over the next 5 intervals (5s), and the number of trees needed for each application. For both applications, the validation accuracy is higher than 94%, demonstrating BT's effectiveness in predicting the performance evolution in the near future. Sinan always runs on a single NVidia Titan XP GPU with average utilization below 2%.

5.3 Performance and Resource Efficiency

We now evaluate Sinan's ability to reduce resource consumption while meeting QoS on the local cluster. We compare Sinan against autoscaling and PowerChief [52]. We experimented with two autoscaling policies: AutoScaleOpt is configured according to [4], which increases resources by 10% and 30% when utilization is within [60%, 70%) and [70%, 100%] respectively, and reduces resources by 10% and 30% when utilization is within [30%, 40%) and [0%, 30%). AutoScaleCons is more conservative and optimizes for QoS, using thresholds tuned for the examined applications. It increases resources by 10% and 30% when utilization is within [30%, 50%)
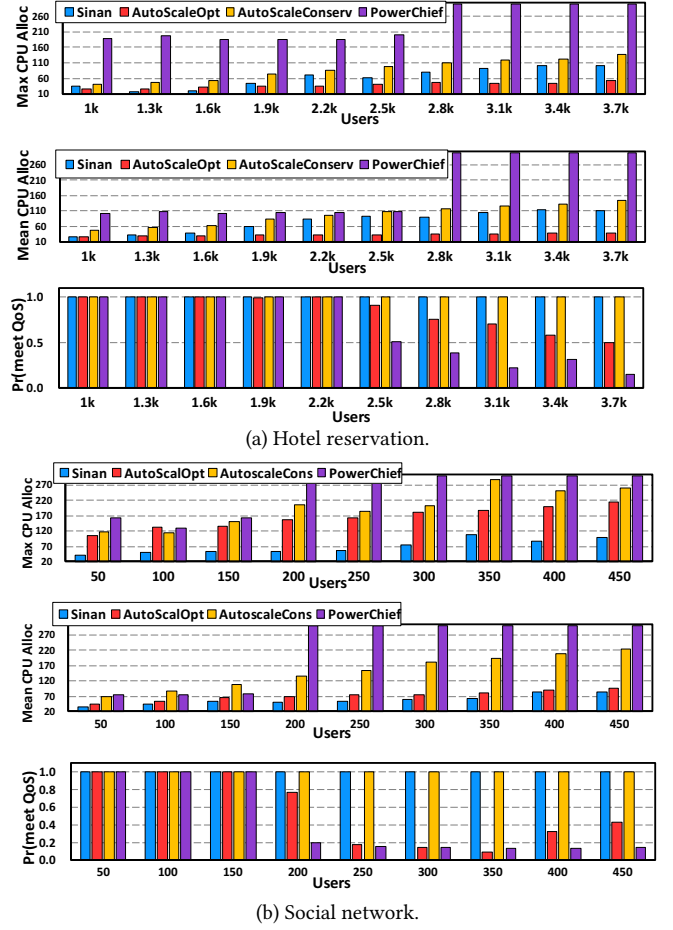

(a) Hotel reservation.

(b) Social network.

Figure 11: The mean and max CPU allocation, and the probability of meeting QoS for Sinan, Autoscaling, and Power-Chief.

and [50%, 100%], and reduces resources by 10% when utilization is within [0%, 10%). PowerChief is implemented as in [52], and estimates the queue length and queueing time ahead of each tier using network traces obtained through Docker.

For each service, we run 9 experiments with an increasing number of emulated users sending requests under a Poisson distribution with 1 RPS mean arrival rate. Figure 11 shows the mean and max CPU allocation, and the probability of meeting QoS across all studied mechanisms, where CPU allocation is the aggregate number of CPUs assigned to all tiers averaged over time, the max CPU allocation is the max of the aggregate CPU allocation over time, and the probability of meeting QoS is the fraction of execution time when end-to-end QoS is met.

For Hotel Reservation, only Sinan and AutoScaleCons meet QoS at all times, with Sinan additionally reducing CPU usage by 25.9% on average, and up to 46.0%. AutoScaleOpt only meets QoS at low loads, when the number of users is no greater than 1900. At 2200 users, AutoScaleOpt starts to violate QoS by 0.7%, and the probability of meeting QoS drops to 90.3% at 2800 users, and less than 80% beyond
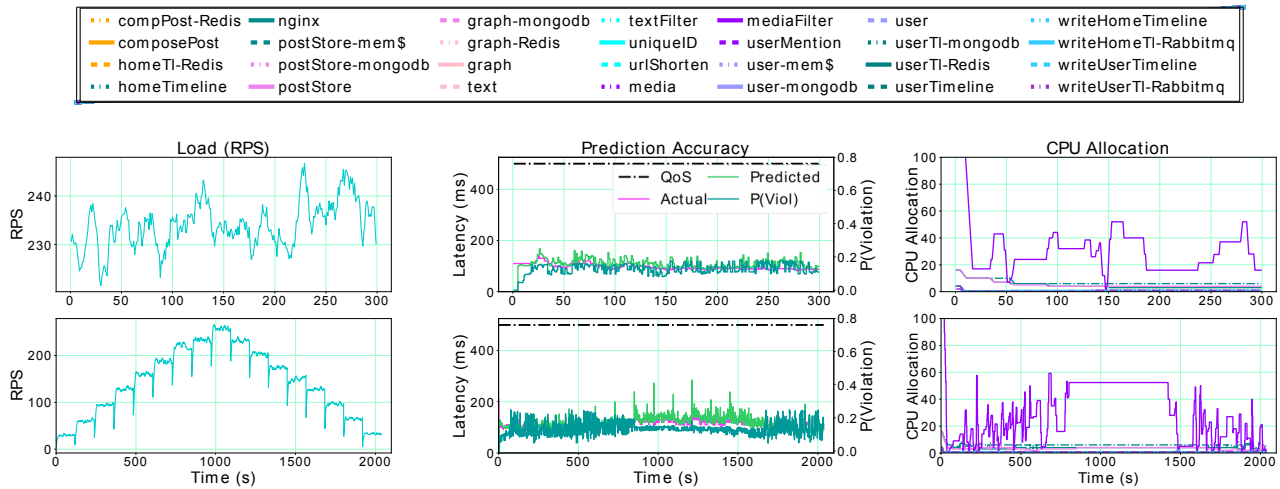
Figure 12: (Top) RPS, latency, and allocated resources per tier with Sinan for Social Network with 250 users. (Bottom) RPC, latency, and allocated resources per tier with diurnal load. For both scenarios, Sinan's predicted latency closely follows the end-to-end measured latency, avoiding QoS violations and excessive overprovisioning, while allocated resources per tier take into account the impact of microservice dependencies on end-to-end performance.

3000 users. Similarly, PowerChief meets QoS for fewer than 2500 users, however the probability of meeting QoS drops to 50.8% at 2800 users, and never exceeds 40% beyond 3000 users. AutoScaleOpt uses 53% the amount of resources Sinan requires on average, at the price of performance unpredictability, and PowerChief uses 2.57× more resources than Sinan despite violating QoS.

For the more complicated Social Network, Sinan's performance benefits are more pronounced. Once again, only Sinan and AutoScaleCons meet QoS across loads, while Sinan also reduces CPU usage on average by 59.0% and up to 68.1%. Both AutoScaleOpt and PowerChief only meet QoS for fewer than 150 users, despite using on average 1.26× and up to 3.75× the resources Sinan needs. For higher loads, PowerChief's QoS meeting probability is at most 20% above 150 users, and AutoscaleOpt's QoS meeting probability starts at 76.3% for 200 users, and decreases to 8.7% for 350 users.

By reducing both the average and max CPU allocation, Sinan can yield more resources to colocated tasks, improving the machine's effective utilization [11, 19, 32, 33]. There are three reasons why PowerChief cannot reduce resources similarly and leads to QoS violations. First, as discussed in Sec. 2.3, the complex topology of microservices means that the tier with the longest igress queue, which PowerChief signals as the source of performance issues, is not necessarily the culprit but a symptom. Second, in interactive applications, queueing takes place across the system stack, including the NIC, OS kernel, network processing, and application, making precise queueing time estimations challenging, especially when tracing uses sampling. Finally, the stricter latency targets of microservices, compared to traditional cloud services, indicate that small fluctuations in queueing time can result in major QoS violations due to imperfect pipelining across tiers causing backpressure to amplify across the system.

Figure 12 shows the detailed results for Social Network, for 300 concurrent users under a diurnal load. The three columns each show

requests per second (RPS), predicted latency vs. real latency and predicted QoS violation probability, and the realtime CPU allocation. As shown, Sinan's tail latency prediction closely follows the ground truth, and is able to react rapidly to fluctuations in the input load.

5.4 Incremental Retraining

We show the incremental retraining overheads of Sinan's ML models in three different deployment scenarios with the Social Network applications: switching to new server platforms (from the local cluster to a GCE cluster), changing the number of replicas (scale out factor) for all microservices except the backend databases (to avoid data migration overheads), and modifying the application design by introducing encryption in post messages uploaded by users (posts are encrypted with AES [15] before being stored in the databases). Instead of retraining the ML models from scratch, we use the previously-trained models on the local cluster, and fine-tune them using a small amount of newly-collected data, with the initial learning rate λ being 1×10^{-5}, $\frac{1}{100}$ of the original λ value, in order to preserve the learnt weights in the original model and constrain the new solution derived by the SGD algorithm to be in a nearby region of the original one. The results are shown in Figure 13, in which the y-axis is the RMSE and the x-axis is the number of newly-collected training samples (unit being 1000). The RMSE values with zero new training samples correspond to the original model's accuracy on the newly collected training and validation set. In all three scenarios the training and validation RMSE converge, showing that incremental retraining in Sinan achieves high accuracy, without the overhead of retraining the entire model from scratch.

In terms of new server platforms and different replica numbers, the original model already achieve a RMSE of 33.23ms and 33.1ms correspondingly, showing the generalizability of selected input features. The RMSE of original model, when directly applied to the modified application, is higher compared to the two other cases,
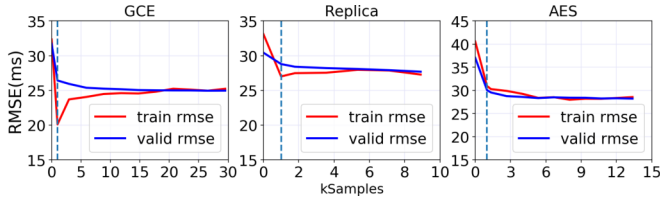
Figure 13: Training & validation RMSE of Fine-tunned CNNs with different amounts of samples.
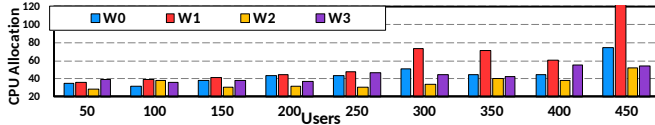

Figure 14: Comparison of the average CPU allocation of four request mixes for Social Network on GCE.

Figure 15: 99th percentile latency distribution for four workload types of Social Network on GCE, managed by Sinan.

reaching 40.56ms. In all of the three cases, the validation RMSE is siginificantly reduced with 1000 newly collected training samples (shown by the dotted lines in each figure), which translates to 16.7 minutes of profiling time. The case of GCE, different replica number and modified application stabilize with 5900 samples (1.6 hours of profiling), 1800 samples (0.5 hour of profiling) and 5300 samples (1.5 hours of profiling), and achieve training vs. validation RMSE of 24.8ms vs. 25.2ms, 27.5ms vs. 28.2ms, and 28.4ms vs. 28.3ms correspondingly.

5.5 Sinan's Scalability

We now show Sinan's scalability on GCE running Social Network. We use the fine-tuned model described in Section 5.4. Apart from the CNN, XGBoost achieves training and validation accuracy of 96.1% and 95.0%. The model's size and speed remain unchanged, since they share the same architecture with the local cluster models.

To further test Sinan's robustness to workload changes, we experimented with four workloads for Social Network, by varying request types. Some requests, like ComposePost involve the majority of microservices, and hence are more resource intensive, while others, like ReadUserTimeline involve a much smaller number of tiers, and are easier to allocate resources for. We vary the ratio of ComposePost:ReadHomeTimeline:ReadUserTimeline requests; the ratios of the $W0$, $W1$, $W2$ and $W3$ workloads are 5:80:15, 10:80:10, 1:90:9, and 5:70:25, where $W0$ has the same ratio as the training set. The ratios are representative of different social media engagement scenarios [41]. The average CPU allocation and tail latency distribution are shown in Figure 14 and Figure 15. Sinan always meets QoS, adjusting resources accordingly. $W1$ requires the max compute resources (170 vCPUs for 450 users), because of the highest number of ComposePost requests, which trigger compute-intensive ML microservices.

5.6 Explainable ML

For users to trust ML, it is important to interpret its output with respect to the system it manages, instead of treating ML as a black box. We are specifically interested in understanding what makes
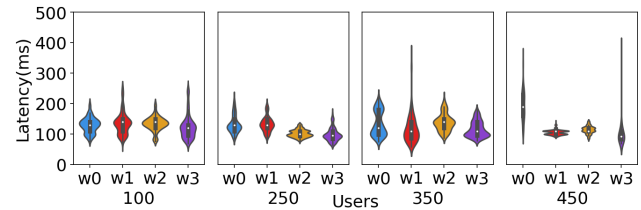
some features in the model more important than others. The benefits are threefold: 1) debugging the models; 2) identifying and fixing performance issues; 3) filtering out spurious features to reduce model size and speed up inference.

5.6.1 Interpretability Methods. For the CNN model, we adopt the widely-used ML interpretability approach LIME [40]. LIME interprets NNs by identifying their key input features which contribute most to predictions. Given an input X, LIME perturbs X to obtain a set of artificial samples which are close to X in the feature space. Then, LIME classifies the perturbed samples with the NN, and uses the labeled data to fit a linear regression model. Given that linear regression is easy to interpret, LIME uses it to identify important features based on the regression parameters. Since we are mainly interested in understanding the culprit of the QoS violations, we choose samples X from the timesteps where QoS violations occur. We perturb the features of a given tier or resource by multiplying that feature with different constants. For example, to study the importance of MongoDB, we multiply its utilization history with two constants 0.5 and 0.7, and generate multiple perturbed samples. Then, we construct a dataset with all perturbed and original data to train the linear regression model. Last, we rank the importance of each feature by summing the value of their associated weights.

5.6.2 Interpreting the CNN. We used LIME to correct performance issues in Social Network [26], where tail latency experienced periods of spikes and instability despite the low load, as shown by the red line in Figure 16. Manual debugging is cumbersome, as it requires delving into each tier, and potentially combinations of tiers to identify the root cause. Instead, we leverage explainable ML to filter the search space. First, we identify the top-5 most important tiers; the results are shown in the w/ Sync part of Table 4. We find that the most important tier for the model's prediction is social-graph Redis, instead of tiers with heavy CPU utilization, like nginx.

We then examine the importance of each resource metric for Redis, and find that the most meaningful resources are cache and resident working set size, which correspond to data from disk cached in memory, and non-cached memory, including stacks and heaps. Using these hints, we check the memory configuration and statistics of Redis, and identify that it is set to record logs in persistent storage every minute. For each operation, Redis forks a new process and copies all written memory to disk; during this it stops serving requests.

Table 4: Top-5 most critical tiers and resources for QoS with/without log synchronization in Social Network — SGrf and WUsr are social graph and write user, respectively.

	Tiers	SGrf Redis	post storage	WUsr timeline	SGrf MongoDB	SGrf
w/ Sync	Weights	5109.9	1609.8	1503.1	849.7	482.7
	Resource utilization	cache memory	RSS	# of cores	CPU utilization	received packets
	Weights	15181.9	1576.1	658.5	322.7	20.0
w/o Sync	Tiers	WUsr timeline	WUsr rabbitmq	SGrf MongoDB	SGrf	SGrf Redis
	Weights	3948.6	3601.6	1794.0	600.9	451.7
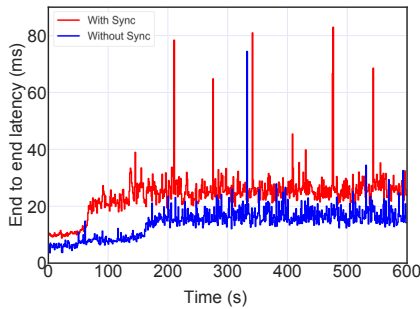
Figure 16: Tail latency for the Social Network application when Redis's logging is enabled (red) and disabled (blue). Sinan identified Redis as the source of unpredictable performance, and additionally determined the resources that were being saturated, pointing to the issue being in Redis's logging functionality. Disabling logging significantly improved performance, which is also reflected in that tier's importance, as far as meeting QoS is concerned, being reduced.

Disabling the log persistence eliminated most of the latency spikes, as shown by the blue line in Figure 16. We further analyze feature importance in the model trained with data from the modified Social Network, and find that the importance of social-graph Redis is significantly reduced, as shown in the w/o Sync part of Table 4, in agreement with our observation that the service's tail latency is no longer sensitive to that tier.

6 RELATED WORK

We now review related work on microservices, cloud management, and the use of machine learning in cloud systems.

Microservices: The emergence of microservices has prompted recent work to study their characteristics and system implications [24, 26, 38]. DeathstarBench [26] and uSuite [46] are two representative microservice benchmark suites. DeathStarBench includes several end-to-end applications built with microservices, and explores the system implications of microservices in terms of server design, network and OS overheads, cluster management, programming frameworks, and tail at scale effects. uSuite also introduces a number of multi-tier applications built with microservices and studies their performance and resource characteristics. Urgaonkar et al. [50] introduced analytical modeling to multi-tier applications,

which accurately captured the impact of aspects like concurrency limits and caching policies. The takeaway of all these studies is that, despite their benefits, microservices change several assumptions current cloud infrastructures are designed with, introducing new system challenges both in hardware and software.

In terms of resource management, Wechat [53] manages microservices with overload control, by matching the throughput of the upstream and downstream services; PowerChief [52] dynamically power boosts bottleneck services in multi-phase applications, and Suresh et al. [47] leverage overload control and adopt deadline-based scheduling to improve tail latency in multi-tier workloads. Finally, Sriraman et al. [45] present an autotuning framework for microservice concurrency, and show the impact of threading decisions on application performance and responsiveness.

Cloud resource management: The prevalence of cloud computing has motivated many cluster management designs. Quasar [18, 19], Mesos [29], Torque [49], and Omega [43] all target resource allocation in large, multi-tenant clusters. Quasar [19] is a QoS-aware cluster manager that leverages machine learning to identify the resource preferences of new, unknown applications, and allocate resources in a way that meets their performance requirements without sacrificing resource efficiency. Mesos [29] is a two-level scheduler that makes resource offers to different tenants, while Omega [43] uses a shared-state approach to scale to larger clusters. More recently, PARTIES [11] leveraged the intuition that resources are fungible to co-locate multiple interactive services on a server, using resource partitioning. Autoscaling [37] is the industry standard for elastically scaling allocations based on utilization [8–10]. While all these systems improve the performance and/or resource efficiency of the cloud infrastructures they manage, they are designed for monolithic applications, or services with a few tiers, and cannot be directly applied to microservices.

ML in cloud systems: There has been growing interest in leveraging ML to tackle system problems, especially resource management. Quasar leverages collaborative filtering to identify appropriate resource allocations for unknown jobs. Autopilot [42] uses an ensemble of models to infer efficient CPU and memory job configurations. Resource central [14] characterizes VM instance behavior and trains a set of ML models offline, which accurately predict CPU utilization, deployment size, lifetime, etc. using random forests and boosting trees. Finally, Seer [25] presented a performance debugging system for microservices, which leverages deep learning to identify patterns of common performance issues and to help locate and resolve them.

7 CONCLUSION

We have presented Sinan, a scalable and QoS-aware resource manager for interactive microservices. Sinan highlights the challenges of managing complex microservices, and leverages a set of validated ML models to infer the impact allocations have on end-to-end tail latency. Sinan operates online and adjusts its decisions to account for application changes. We have evaluated Sinan both on local clusters and public clouds GCE) across different microservices, and showed that it meets QoS without sacrificing resource efficiency. Sinan highlights the importance of automated, data-driven approaches that manage the cloud's complexity in a practical way.

A ARTIFACT APPENDIX

A.1 Abstract

This appendix contains the information needed to reproduce the main experiments in Sinan, which include the results for the online resource management deployment. **The results reproduced during the artifact evaluation process used the large-scale google compute (GCP) version of the artifact**, due to the experimental setting for the local version being difficult to reproduce on a different cluster equipment and topology.

A.2 Artifact Checklist (meta-information)

- **Algorithm:** An algorithm for generating training data that is used by Sinan to determine per-microservices resource allocations.
- **Program:** Modified version of the DeathstarBench suite; focusing on the Social Network and Hotel Reservation applications.
- **Model:** Included in the artifact.
- **Data set:** Scripts for generating the training dataset are included with the artifact.
- **Run-time environment:** 1. Google Cloud Platform (GCP) experiments: Google Cloud SDK is required. 2. Local experiments: Ubuntu 18.04, docker 19.03, docker-ce 20.10, MXnet and XGBoost are required.
- **Hardware:** Local experiments: a cluster with at least two servers, each with 88 cores (Intel E5-2660 v3@2.60GHz) for service deployment, and a GPU server (a NVidia Titan XP GPU) for model training and online inference. For the purpose of reproducing the paper's results, a cluster with at least an equal amount of resources to those specified above is recommended to avoid cross-tier interference.
- **Execution:** 1. Google Cloud Platform (GCP) experiments: deployment experiments take approximately 4 hours for the provided applications. 2. Local experiments: data collection takes 16 hours for Social Network and 9 hours for Hotel Reservation. Deployment experiments take around 4 hours for each application.
- **Metrics:** CPU usage and end-to-end tail latencies, collected periodically over the execution's duration.
- **Output:** Execution logs of the system's performance and resource utilization, including the CPU usage and tail latencies previously mentioned. Log processing scripts are included in the artifact.

- **Experiments:** Short cut scripts for reproducing results are included in the artifact.
- **Publicly available:** Yes
- **Archived:** 10.5281/zenodo.4537132

A.3 Description

A.3.1 How to Access. Please visit https://github.com/zyqCSL/sinan-gcp for the software implementation needed for the Google Cloud Platform (GCP) experiments, and https://github.com/zyqCSL/sinan-local for the software implementation needed for the local experiments.

A.3.2 Hardware Dependencies. For local experiments, we provide a configuration using two servers each with 88 cores (Intel E5-2660 v3@2.60GHz) to deploy the applications Sinan is evaluated with. A GPU server (a NVidia Titan XP GPU) is also required for model training and online inference. In order to reproduce the results locally, a cluster with no less CPU resources than the ones described above is recommended. If fewer resources are availale, you will need to lower the input load using the provided workload generators, and scale down each microservice's container accordingly.

A.3.3 Software Dependencies. For experiments on Google Cloud Platform (GCP), the Google Cloud SDK is required. For local experiments, Ubuntu 18.04, docker 19.03, docker-ce 20.10, MXnet and XGBoost are required.

A.3.4 Datasets. The scripts to generate the datasets used for training Sinan are included in the artifact. Information on how to use the scripts is included in the artifact's README file.

A.3.5 Models. The models are included in the submitted repo. Please check the README of the artifact for more information.

A.4 Installation

For the Google Cloud Platform (GCP) experiments, Google Cloud SDK is required. Please follow the guidelines in https://cloud.google.com/sdk/docs/how-to to install it. Initialization and software dependencies are automatically installed using the VM setup scripts provided in the artifact. For the local experiments, please use servers with Ubuntu 18.04, docker 19.03 and docker-ce 20.10 installed. Sinan's inference engine also requires MXNet and XGBoost installed. For interested readers, we refer you to the README of the local version of the artifact for more details.

A.5 Experiment Workflow

The workflow first generates the training dataset used by Sinan to infer per-tier resource allocations. It then deploys the target application on the local or Google Compute Platform (GCP) cluster, and launches Sinan's inference engine on the GPU server, which continuously determines the probabilities for QoS violations, and the required resource allocations per microservice. Please check the README of the artifact for more details on the experiment workflow.

A.6 Evaluation and Expected Results

The expected results are execution logs of the system, which includes CPU usage and end-to-end tail latencies of the managed

applications, collected periodically over time. The data processing scripts are included in the artifact, and are responsible for computing the average CPU usage and tail latency distribution throughout the duration of an experiment. They also generate the figures showing the CPU usage and end-to-end tail latency over time. Expected results are shown in the paper. Specifically, for Google Cloud Platform (GCP) experiments, please check Figure 14 and Figure 15 for the average CPU measurements and the tail latency distributions. For the local experiments, please check the data labeled with "Sinan" in Figure 11. Results for the diurnal pattern can be compared against Figure 12 in the paper.

REFERENCES

[1] Decomposing twitter: Adventures in service-oriented architecture. https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture.

[2] Docker containers. https://www.docker.com/.

[3] Locust. https://locust.io/.

[4] Step and simple scaling policies for amazon ec2 auto scaling. https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-simple-step.html.

[5] Why grpc? https://grpc.io/.

[6] The evolution of microservices. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, 2016.

[7] Microservices workshop: Why, what, and how to get there. http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference.

[8] Autoscale. https://cwiki.apache.org/cloudstack/autoscaling.html.

[9] Aws autoscaling. http://aws.amazon.com/autoscaling/.

[10] Jeffrey Chase, Darrell Anderson, Prachi Thakar, Amin Vahdat, and Ronald Doyle. Managing energy and server resources in hosting centers. In *Proceedings of SOSP*. Banff, CA, 2001.

[11] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120. ACM, 2019.

[12] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[14] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2017.

[15] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[16] Christina Delimitrou, Nick Bambos, and Christos Kozyrakis. QoS-Aware Admission Control in Heterogeneous Datacenters. In *Proceedings of the International Conference of Autonomic Computing (ICAC)*. San Jose, CA, USA, 2013.

[17] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, USA, 2013.

[18] Christina Delimitrou and Christos Kozyrakis. Quality-of-Service-Aware Scheduling in Heterogeneous Datacenters with Paragon. In *IEEE Micro Special Issue on Top Picks from the Computer Architecture Conferences*. May/June 2014.

[19] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014.

[20] Christina Delimitrou and Christos Kozyrakis. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems. In *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.

[21] Christina Delimitrou and Christos Kozyrakis. Bolt: I Know What You Did Last Summer... In The Cloud. In *Proc. of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[22] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)*, August 2015.

[23] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[24] Yu Gan and Christina Delimitrou. The Architectural Implications of Cloud Microservices. In *Computer Architecture Letters (CAL), vol.17, iss. 2*, Jul-Dec 2018.

[25] Yu Gan, Meghna Pancholi, Dailun Cheng, Siyuan Hu, Yuan He, and Christina Delimitrou. Seer: leveraging big data to navigate the complexity of cloud debugging. In *Proceedings of the 10th USENIX Conference on Hot Topics in Cloud Computing*, pages 13–13. USENIX Association, 2018.

[26] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Yuan He, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18. ACM, 2019.

[27] John Gittins, Kevin Glazebrook, and Richard Weber. *Multi-armed bandit allocation indices*. John Wiley & Sons, 2011.

[28] Kristina Gligorić, Ashton Anderson, and Robert West. How constraints affect content: The case of twitter's switch from 140 to 280 characters. In *Twelfth International AAAI Conference on Web and Social Media*, 2018.

[29] Ben Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI*. Boston, MA, 2011.

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

[31] Ching-Chi Lin, Pangfeng Liu, and Jan-Jan Wu. Energy-aware virtual machine dynamic provision and scheduling for cloud computing. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing (CLOUD)*. Washington, DC, USA, 2011.

[32] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. Minneapolis, MN, 2014.

[33] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, 2015.

[34] Llew Mason, Jonathan Baxter, Peter Bartlett, and Marcus Frean. Boosting algorithms as gradient descent. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 512–518, Cambridge, MA, USA, 1999. MIT Press.

[35] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 319–330, 2011.

[36] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of SOSP*. Farminton, PA, 2013.

[37] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 51(4):73, 2018.

[38] Joy Rahman and Palden Lama. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *IEEE International Conference on Cloud Engineering, IC2E 2019, Prague, Czech Republic, June 24-27, 2019*, pages 200–210. IEEE, 2019.

[39] Charles Reiss, Alexey Tumanov, Gregory Ganger, Randy Katz, and Michael Kozych. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of SOCC*. 2012.

[40] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144, 2016.

[41] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[42] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[43] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of EuroSys*. Prague, Czech Republic, 2013.

[44] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of SOCC*. Cascais, Portugal, 2011.

[45] Akshitha Sriraman and Thomas F. Wenisch. μtune: Auto-tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX

Association.

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

[47] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 611–623. ACM, 2017.

[48] Apache thrift. https://thrift.apache.org.

[49] Torque resource manager. http://www.adaptivecomputing.com/products/open-source/torque/.

[50] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. *SIGMETRICS Perform. Eval. Rev.*, 33(1):291–302, June 2005.

[51] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[52] Hailong Yang, Quan Chen, Moeiz Riaz, Zhongzhi Luan, Lingjia Tang, and Jason Mars. Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 133–146, New York, NY, USA, 2017. Association for Computing Machinery.

[53] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161. ACM, 2018.