Sinan: Data-Driven Resource Management for Interactive Microservices

Yanqi Zhang*, Weizhe Hua*, Zhuangzhuang Zhou, Edward Suh, Christina Delimitrou
Cornell Univeristy
{yz2297, wh399, zz586, edward, delimitrou}@cornell.edu
*Equal contribution

***Abstract*—Cloud applications are increasingly shifting to interactive and loosely-coupled microservices. Despite their advantages, microservices complicate resource management, due to inter-tier dependencies. We present Sinan, a cluster manager for interactive microservices that leverages easily-obtainable tracing data instead of empirical decisions, to infer the impact of a resource allocation on on end-to-end performance, and allocate appropriate resources to each tier. In a preliminary evaluation of Sinan with an end-to-end social network built with microservices, we show that Sinan's data-driven approach, allows the service to always meet its QoS without sacrificing resource efficiency.**

I. Introduction

Over the past few years, the design of online interactive applications has shifted from *monolithic* services that encompass the entire functionality in a single binary, to *microservices* that divide application to a graph of tens or hundreds of single-purpose and loosely-coupled tiers [1, 2, 3, 9, 31, 32]. *Microservices* are appealing for several reasons, including modularity, flexible development and deployment, and high tolerance of software heterogeneity.

Despite their advantages, microservices also introduce new system challenges, primarily in resource management. The dependencies between microservices exacerbate queueing and introduce cascading QoS violations that are difficult to identify and correct in a timely manner [9, 35]. Current cluster managers that optimize for monolithic applications or applications consisting of few tiers are not expressive enough to capture the complexity of microservices [6, 14, 15, 19, 21, 22, 24, 29, 30, 34]. Furthermore, given that typical microservice deployments include tens to hundreds of unique tiers, exhaustively exploring the resource allocation space is prohibitively expensive [7, 16, 17].

Instead in this work we take a data-driven approach to microservices management. Previous work [5, 28] highlighted the potential of data-driven approaches to address resource scheduling for large-scale systems, but they do not directly apply to microservices.

We present our preliminary work on Sinan, an ML-based cluster manager for microservices that leverages the cloud's tracing data and a set of practical ML techniques to infer the impact of resource allocation on end-to-end performance, and assign appropriate resources to each application tier. Sinan leverages efficient action space pruning to reduce the overheads of exploration, and trains two models with the tracing data; a 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 account for the system's inertia in building up queues. Sinan operates online, adjusting per-tier resources dynamically according to the service's runtime status and end-to-end Quality of Service (QoS) target.

We evaluate Sinan using an end-to-end, microservices-based application that implements a social network [9], and compare it against traditional autoscaling approaches. We also validate the accuracy of Sinan's ML models, and show that QoS does not come at the price of resource inefficiency. Finally, we emphasize the need for explainable ML models, which provide design insights for large-scale systems, using an example of Redis's log synchronization, which Sinan identified as the source of unpredictable performance.

II. Overview

A. *Motivating Application*

We use the `Social Network` from DeathStarBench [9]. The service implements a broadcast-style social network with uni-directional follow relationships, and its architecture is shown in Fig. 1.

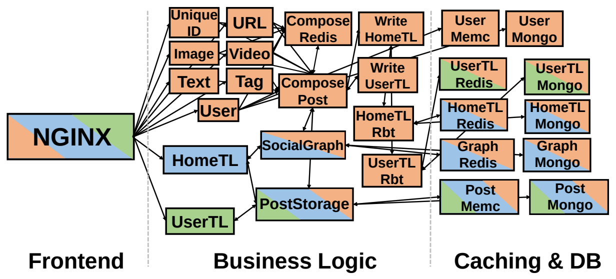

Fig. 1: Social Network: vertical lines separate different types of tiers (front-end, logic, backend), while colors show microservices for different request types. Blue: read own timeline, green: read user timeline, orange: compose post.

Functionality: Users (`client`) send requests over `http` to `NGINX` [23] front-end, which selects a specific downstream service to forward the request to. Users can create posts embedded with text, media, links, and tags to other users, which are then broadcasted to all their followers. Users can also read posts on their timelines and create new posts themselves.

Inter-microservice messages use Thrift RPCs [33]. The service's backend uses `memcached` [8] for caching, and `MongoDB` [20] for persistently storing posts, user profiles, and media. Index information, such as user timeline indices, are stored in `Redis`. `RabbitMQ` [25] instances are added between business logic and `MongoDB` to make time-consuming database write operations asynchronous, and prevent them from blocking upstream connections. We use the `Reed98` [27] social friendship network to populate the user database. It is extracted from Facebook, and consists of 962 people and 18.1K edges, where an edge is a follow relationship. We model user activity according to the behavior of Twitter users reported in [13], where a user's posting activity positively correlates with the number of their followers. The distribution of post length emulates Twitter's text length distribution [11].

B. Management Challenges and the Need for ML

Microservices management faces three major challenges.

1. Prohibitively-large action space 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. Assuming N microservice tiers and a pool of C ($C \geq N$) homogeneous physical cores, each with F frequency levels, the size of the action space is $\binom{C-1}{N-1} \cdot N^F$. For example, given a cluster with 150 cores and assuming 10 frequency steps per tier, the resource allocation space size of the *Social Network* application is 7.78×10^{55}. Profiling all actions under different loads would require significant time and computation resources. As a result, efficient action space pruning methods and statistical tools with strong generalization capability are urgently needed for resource scheduling.

2. Delayed queueing effect Consider a queueing system with processing throughput T_o under a latency QoS target. 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 meet its QoS, and remain stable, while using the minimum amount of resources R needed. Even in the case where 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 QoS violation. Multi-tier microservices are complex queueing systems with queues both between and within microservices [9, 10]. This delayed queueing effect highlights the need for the ML model to evaluate the long-term effect of resource management actions, and proactively prevent the resource manager from reducing resources overly-aggressively to avoid latency spikes that introduce long recovery periods. To mitigate a QoS violation, the manager must increase resources proactively, otherwise the QoS violation becomes unavoidable, even if more resources are allocated a posteriori.

3. 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 [9, 10]. 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.

C. Proposed Approach

These challenges suggest that empirical resource management, such as autoscaling, is prone to unpredictable performance and/or resource inefficiencies. Sinan takes instead a data-driven, ML-based approach that automates resource management for microservices, leveraging a set of scalable and validated ML models, that allows high and predictable performance and resource utilization. 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 the application's QoS. Below, we first describe the ML models (Sec. III-A), and Sinan's system architecture (Sec. III).

III. Sinan

A. Machine Learning Models

We initially designed Sinan's ML model to only predict the end-to-end latency tail distribution using a CNN, such that the scheduler can query the model with potential resource allocations, and 1) choose the one that minimizes the required resources while meeting the end-to-end QoS, or 2) the one that minimizes end-to-end latency if there are multiple allocations satisfying QoS. However, this model experienced consistently high errors during deployment, due to the delayed queueing effect mentioned previously. Therefore, it is crucial for the model to also predict the long-term impact of resource allocations.

A straightforward fix is to use a multi-task CNN model that predicts latency distribution for the immediate future, and the QoS violation probability in the long term. This approach was also shown to frequently overpredict latency, due to the large gap between latency and probability values.

Sinan currently follows a hybrid approach, which uses a CNN to predict the end-to-end latency of the next decision interval, and a boosted trees (BT) model to anticipate QoS violations further into the future. We refer to the CNN model as the *short-term latency predictor*, and the BT model as the *long-term violation predictor*.

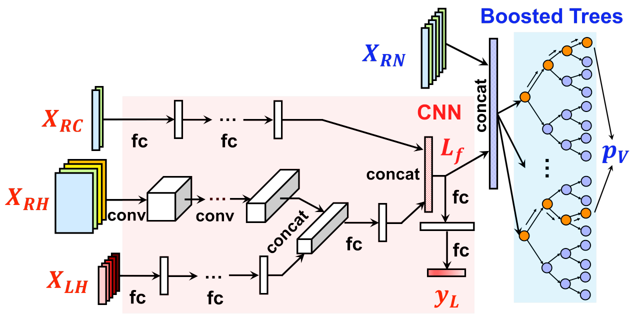

Fig. 2: Sinan's hybrid model, consisting of a CNN and a boosted trees model. The CNN predicts the end-to-end latency (y_L), and the boosted trees predicts the probability of a QoS violation (p_V).

As shown in Fig. 2, the latency predictor takes as input the resource usage history (X_{RH}), the latency history (X_{LH}), and the potential resource configuration (X_{RC}) for the next timestep, and predicts the end-to-end tail latency distribution (y_L) (95^{th} to 99^{th} percentiles) of the next timestep. X_{RH} is formed as a 3D tensor whose x-axis is N tiers in the microservices graph, 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. X_{RC} and X_{LH} are 2D matrices. For X_{RC}, x-axis is N tiers and y-axis is core number and cpu frequency. For X_{RH}, x-axis is T timestamps, and y-axises are latency tail distribution (95^{th} to 99^{th} percentiles). The three inputs are firstly individually processed with convolution (Conv) and fully connected (FC) layers and then concatenated to form the latent representation L_f, from which the predicted latency distribution L_f is derived with another FC layer. The loss function of the CNN is shown below:

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

where $f_W(\cdot)$ represents the forward function of the CNN network, $\hat{y}$ is the ground truth, and n is the number of training samples. The CNN is implemented with MxNet [4], and trained with Stochastic Gradient Descent (SGD).

The violation predictor addresses a binary classification task of whether a given allocation will cause a QoS violation further in the future, to filter out undesirable resource options. Ensemble methods are good candidates for this, as they usually perform well in classification tasks, and are robust to overfitting. We use XGBoost [18], 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 the features to a score. The final prediction is the accumulation of scores across all trees. We use the compact latent variable L_f, extracted from the CNN as the input to the BT, to reduce the computation cost. 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 robust to overfitting.

B. *Online Scheduler*

Sinan consists of three components: a centralized scheduler, distributed operators deployed on each server, and a performance forecaster hosting the ML models.

Sinan makes decisions periodically, once every 1s, consistent with the granularity at which QoS is defined. The centralized scheduler queries the distributed operators to obtain the CPU, memory, network, and I/O usage information of each tier in the previous interval through Docker's cgroup stats API. Aside from per-tier information, the scheduler also queries the API gateway to get statistics of user load (implemented via workload generator for simplicity in our experiments). Using the model's output, the scheduler chooses one allocation that is beneficial to QoS and resource-efficient, i.e., uses the least resources needed to meet QoS, and sends its decision to the per-node agents for enforcement.

1) Data Collection: Representative training data is key to the accuracy of ML models. Ideally, the training and testing data should follow similar distributions, to avoid covariate shift, which means that the training dataset needs to cover a sufficient spectrum of application behaviors. Meanwhile, because of the impractical size of resource allocation space, Sinan's data collection agent is only able to cover a limited fraction of the entire space within the permitted time and computation budgets. As a result, we design the data collection agent to follow two principles. Firstly, we quantize the minimum amount of resources by which Sinan can adjust an allocation, to reduce the size of the explored resource space. Secondly, we enforce the data collection agent to only explore allocations in the $[0, QoS + \alpha]$ tail latency region, where α is a small value compared to QoS, so that the trained model is able to learn the behavior of resource allocations which initiate QoS violations without biasing the collected distribution severely towards values greater than QoS. We have also compared Sinan's data collection scheme against collecting data randomly and when a resource autoscaler is in place, and showed that Sinan consistently explores a larger and more useful region of the resource space, improving the accuracy of the ML models.

2) Resource allocation: Online scheduling in Sinan currently involves has two phases: core allocation and power management. In core allocation, the scheduler minimizes the number of cores until no further reduction is considered feasible by the ML model. Then the scheduler enters the power management phase and gradually reduces frequency. After the two phases are complete, the scheduler keeps the resulting allocation, and increases resources when required by the ML model. The scheduler also has a safety mechanism for cases where the ML model fails to correctly predict a QoS violation. Whenever the number of missed QoS violations exceeds that threshold, the scheduler trusts the model less, and is more conservative when reclaiming resources. In practice, Sinan never had to lower its trust to the ML model.

IV. Evaluation

A. Methodology

We use a local cluster with 150 physical cores in total for data collection and online deployment. Each microservice runs in a Docker container. We collected 192,031 samples, and split them by 9:1 as training and testing set.

B. Sinan's Accuracy and Speed

We first compare the CNN in Sinan against a multilayer perceptron (MLP), and a long short-term memory (LSTM) network. Sinan's CNN achieves the lowest RMSE (9.5ms vs. 19.6ms for MLP and 13.1ms for LSTM), while also having the smallest model size (264KB). Although the CNN's speed is slightly slower than the LSTM (6.7ms/batch vs. 3.6ms/batch for LSTM), its inference latency is within 1% of the decision interval (1s), which does not delay online decisions. In terms of the BT model, validation accuracy is higher than 93%, with 3.1% false positives, and 3.9% false negatives. In all cases, Sinan runs on a single NVidia Titan XP GPU with average utilization below 2%.

C. Online Deployment

We now evaluate Sinan's ability to meet QoS during online deployment. We compare Sinan against two autoscaling policies. AS_Opt is configured according to [12], which reduces cores and frequency when the CPU utilization of a tier drops below 30% and 40% respectively, and increases cores when utilization exceeds 70%. AS_Cons is more conservative, and optimizes for QoS. It uses 20% and 30% CPU utilization, to downsize cores and frequency respectively, and 50% to upscale cores. For each service, we run 10 experiments with constant load from 10% to 100% of the max QPS, and a diurnal load pattern, where load starts from 10%, gradually rises to peak QPS, and then decreases back to 10%.

At near-saturation load, differences between schedulers are small because of the limited size of our cpu pool. The difference becomes more apparent at low loads, where Sinan reduces tail latency and latency variability considerably. In contrast, tail latency varies widely for the two autoscalers, and especially for AS_Opt. The violations in AS_Opt are caused by not upscaling `NGINX`, whose utilization did not exceed the upscale threshold. The difference is more dramatic for the diurnal load, where AS_Opt violates QoS by more than an order of magnitude.

Note that Sinan's tail latency reduction also comes with significant resource savings. Even when compared to AS_Cons, Sinan reduces the active cores by **16.3%** on average, and up to **29.1%**. Sinan also reduces the average frequency of active cores by **37.2%** on average, and up to **57.47%**.

Fig. 3 shows the detailed results of Sinan's resource allocation over time for the diurnal pattern. Sinan is able to

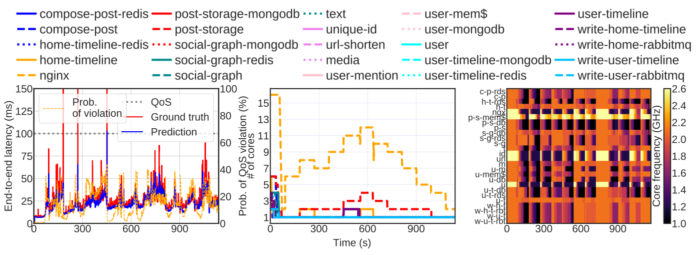

Fig. 3: Latency and resources under a diurnal load.

dynamically adjust resources to handle the fluctuating load, and the predicted tail latency closely follows the ground truth.

D. Explainable ML

For users to trust ML used in systems, 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 of understanding this are threefold: 1) debugging the ML models; 2) identifying and fixing performance issues; 3) filtering out insignificant features to reduce the model size and speed up inference.

We adopt a widely-used ML interpretability approach called LIME [26]. LIME interprets NNs by identifying key input features which contribute most to predictions. Given an input X, LIME perturbs X to obtain a set of artificial samples, close to X in the feature space. Then, LIME labels the perturbed samples by classifying them with the NN, and uses the labeled data to fit a linear regression model, and uses it to identify important features based on the regression parameters. Since we are mainly interested in understanding the culprit of QoS violations, we choose input samples X close to when QoS violations occur, and apply LIME. We perturb resource usage statistics, and construct a dataset with all perturbed and original data to train a linear regression model. Lastly, we rank the importance of each feature.

We applied LIME to diagnose performance issues in cases where tail latency experienced spikes despite the low load. First, 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 cached in memory and non-cached memory for a process, including stacks and heaps. Using these hints, we check the memory-related configuration and runtime statistics of social-graph Redis, and find that it is configured to record logging data in persistent storage every minute. For each persistence operation, Redis forks a new process and copies all written memory to disk; during that time it stops serving user requests. Disabling logging resulted in most latency spikes being eliminated. Re-applying LIME to the modified Social Network showed that the importance of Redis had significantly dropped, in agreement with our observation that tail latency is no longer sensitive to it.

V. Conclusion

We have presented our early work on Sinan, an ML-driven, online resource manager for interactive microservices. Sinan highlights the challenges of managing complex microservices, and leverages a set of scalable and validated ML models to reduce resource usage while meeting end-to-end Quality of Service.

Acknowledgements

We sincerely thank Daniel Sanchez and the anonymous reviewers for their feedback on earlier versions of this manuscript. This work was in part supported by NSF grants NeTS CSR-1704742, CCF-1846046, and a John and Norma Balen Sesquisentennial Faculty Fellowship.

References

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

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

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

[4] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[5] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. 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*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 153–167. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132772

[6] C. Delimitrou and C. 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.

[7] ——, "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.

[8] B. Fitzpatrick, "Distributed caching with memcached," in *Linux Journal, Volume 2004, Issue 124, 2004*.

[9] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "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*. ACM, 2019, pp. 3–18.

[10] Y. Gan, Y. Zhang, K. Hu, Y. He, M. Pancholi, D. Cheng, and C. Delimitrou, "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 (ASPLOS)*, April 2019.

[11] K. Gligorić, A. Anderson, and R. 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.

[12] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 427–444.

[13] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. AcM, 2010, pp. 591–600.

[14] C.-C. Lin, P. Liu, and J.-J. 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. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2011.94

[15] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. 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.

[16] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, 2015.

[17] J. Mars and L. Tang, "Whare-map: heterogeneity in "homogeneous" warehouse-scale computers," in *Proceedings of ISCA*. Tel-Aviv, Israel, 2013.

[18] L. Mason, J. Baxter, P. Bartlett, and M. Frean, "Boosting algorithms as gradient descent," in *Proceedings of the 12th International Conference on Neural Information Processing Systems*, ser. NIPS'99. Cambridge, MA, USA: MIT Press, 1999, pp. 512–518. [Online]. Available: http://dl.acm.org/citation.cfm?id=3009657.3009730

[19] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011, pp. 319–330.

[20] "Mongodb," https://www.mongodb.com.

[21] R. Nathuji, C. Isci, and E. Gorbatov, "Exploiting platform heterogeneity for power efficient data centers," in *Proceedings of ICAC*. Jacksonville, FL, 2007.

[22] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *Proceedings of EuroSys*. Paris,France, 2010.

[23] "Nginx," https://www.nginx.com.

[24] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proceedings of SOSP*. Farminton, PA, 2013.

[25] "Rabbitmq," https://www.rabbitmq.com.

[26] M. T. Ribeiro, S. Singh, and C. 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*, 2016, pp. 1135–1144.

[27] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[28] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[29] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of EuroSys*. Prague, Czech Republic, 2013.

[30] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of SOCC*. Cascais, Portugal, 2011.

[31] A. Sriraman and T. F. Wenisch, "usuite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 1–12.

[32] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu, "Distributed resource management across process boundaries," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 611–623.

[33] "Apache thrift," https://thrift.apache.org.

[34] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at Google with Borg,” in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[35] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, “Overload control for scaling wechat microservices,” in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2018, pp. 149–161.