CapPredictor: A Capacity Headroom Prediction Framework in Cloud

Ruoying Wang, Lei Zhang, Yang Yang, Yi Zhen, Bo Long, Tie Wang,
Vinoth Govindaraj, Todd Palino, Samir Tata, Viji Nair
LinkedIn Corporation, Mountain View, California, USA
{ruowang, lzhang1, yyang, yzhen, blong, tiewang, vgovindaraj, tpalino, stata, vijinair}@linkedin.com

Abstract

Nowadays, optimizing capacity is becoming a critical and hot topic for cloud computing. At LinkedIn, to understand and forecast service capacity needs, especially for stateful services, we developed an end-to-end machine learning framework, called CapPredictor. CapPredictor can predict capacity headroom of system[1] and workload[2] variables based on production requirement of service performance metrics[3]. Evaluation on production data demonstrates that CapPredictor is very effective.

1 Introduction

LinkedIn serves more than 660 million members on a global computing infrastructure through hundreds of internal services. With releases of new product features, infrastructure upgrades, and organic traffic growth, it is important to accurately predict the capacity needs of services and hence provide sufficient resources in a timely manner. The task is, however, extremely challenging due to the complex and dynamic nature of cloud computing.

There are two main categories of services: stateful and stateless. Stateless services only have stateless operations, in which the returned contents solely depend on inputs. Usually, such services are deployed on machines that could perform identical tasks. Stateful services, on the other hand, support stateful operations, of which the outputs depend on not only inputs but also internal states. Data services are typical stateful services, because replicas of data partitions are mixed and distributed in different machines of which each machine is a distinct entity and hence the operations will always be stateful.

[1]System variables are resource factors of the system such as cpu and/or memory usage, disk I/O, garbage collection (GC) count/time, JVM thread counts, and etc.

[2]Workload variables are the operation loads on the system such as read/write query count per second (QPS), read and write key count per second (KPS), read and write throughput, and etc.

[3]System performance metrics are the measurements used to quantify and monitor system behavior and performance, such as latency, errors etc.

Traditionally, one can leverage load tests by redirecting site traffic to a subset of machines, increasing the load on any single machine to estimate the capacity needs for stateless services. For example, LinkedIn has been using Redliner (Xia and Rao 2017) for a few years to identify capacity need for stateless services. However, we can not apply this kind of mechanism to stateful services directly, because it is futile to redirect traffic to machines where target data does not reside. Instead, one needs to duplicate site traffic in order to increase workload meaningfully, which is a difficult task. In addition, there are many workload and system variables could affect service performance, and, obviously, running tests on each variable can not scale.

In this paper, we propose CapPredictor, a generic machine learning framework that predict capacity headroom by extrapolating the relationship between service performance metrics and system/workload variables. CapPredictor differs from the typical machine learning model in that it does not predict the service performance within the domain of the observed system or workload variables, but predict what is likely to happen beyond observation (e.g., larger QPS reads that the server has not seen). To this end, CapPredictor models the data generating process using a smooth function which can be reasonably extrapolated. For easier understanding of the capacity needs, we present a single variable CapPredictor which extrapolates on one influencing variable, but the methodology can be extended to multiple variable easily and we leave it to future work.

The contributions of this paper can be summarized as follows:

1. To the best of our knowledge, this paper is the first study to understand capacity of stateful services in cloud environment.
2. CapPredictor is the first generic and end-to-end machine learning framework to predict capacity headroom of stateful services.

2 Related Work

DevOps (Kim et al. 2016) has been widely adopted as a method for facilitating continuous development and release of services. The ever-increasing scale and complexity of services pose significant challenges to engineers on building services efficiently. Under this scenario, a new term, AIOps (Lerner 2017), was proposed to address DevOps' challenges with artificial intelligence (AI), especially in cloud computing environment. Recent research and applications on AIOps have shown great potential in constraint detection, resource optimization, and capacity planning (Li and Dang 2019; Dang, Lin, and Huang 2019; Chen et al. 2019).

Our work falls in the realm of capacity planning (Menasce and Almeida 2001). It is comprised of two sub-areas: (1) understanding how various workloads impact service performance; (2) forecasting workload needs (Zhuang et al. 2015). The former focuses on estimating the relationship between performance and other variables, while the latter tries to accurately project workload growth on a quarterly/yearly basis. For the second category, there exist some successful stories in industry. For example, Facebook has developed a customized suite of forecasting models, including Bayesian time series models and deep learning models, to forecast data center demands (Krishnamurthy and Kelkar 2018). Similar work has been done at Google and Amazon (Llamas 2016; Krazit 2017). The first category, which is the focus of this paper, has been relatively less explored. One use case is Uber's work (Boone 2018) on leveraging quantile regression to predict CPU utilization based on trips occurred. Our approach has two extensions compared to theirs. First, we relax the linear assumption of the underlying relationship function to only assume its smoothness; second, we extrapolate the relationship to predict beyond the observed workload and system variable range to facilitate immediate capacity planning.

Compared to the stateless services, the stateful services require some type of persistent storage that will survive service restarts, and the data query and operations may vary from service to service. There are some research done in this area to model the relationship between specific query semantics and their impact on the system performance. Li et al. (2012) modeled resource usage at the level of individual query operators, with different models and features for each operator type, and explicitly model the asymptotic behavior of each operator. Marcus and Papaemmanouil (2019) introduced the "plan-structured neural network" for query performance prediction, which reduced the need for human-crafted input features. Others have studied the system architectures that streamline query scheduling at scale (Xu, Cole, and Ting 2019; Liu et al. 2019; Sun and Li 2019). Our approach does not investigate the specific stateful services but rely on the key system and workload variables tracked within the system. Therefore, the framework is applicable for any stateful service. In this paper, we will focus on the capacity prediction of stateful services, but it is applicable to the simpler stateless services as well.

3 Proposed Approach

3.1 Overview

Here we measure the capacity of a stateful service by its headroom, the percentage increase in workload a service could hold while satisfying its Service Level Objective (SLO)[4]. Denote X_i an example workload/system variable, x_i its workload value, x_0 its current workload, H^{X_i} its headroom, and y the measure of service performance. We have

$$H^{X_i} = \frac{\max(x_i | y < \text{SLO}) - x_0}{x_0} \tag{1}$$

To estimate headroom at the service level (H^S), we first estimate the headroom for each system or workload variable (H^{X_i}). Then, the service level headroom would be that of the most constrained variable.

$$H^S = \min_{X_i \in \mathbf{X}} H^{X_i} \tag{2}$$

where $\mathbf{X}$ denotes the set of system and workload variables $\{X_1, X_2, \ldots, X_n\}$.

Figure 1 illustrates the architecture of our proposed end-to-end pipeline. It consists of 4 modules.

1. Data preprocessing module: Collect the historical time series of service performance and workload variables (usually at 1-minute or 5-minute granularity), and aggregate to the desired level, e.g., machine node or cluster level.
2. Variable selection module: Select among hundreds of tracking variables to include for estimation.
3. Headroom estimation and model selection module: Evaluate the performance of a set of model candidates to select the best model for final estimation.
4. Headroom insights module: Summarize headroom estimation and other insights into reports and dashboards.

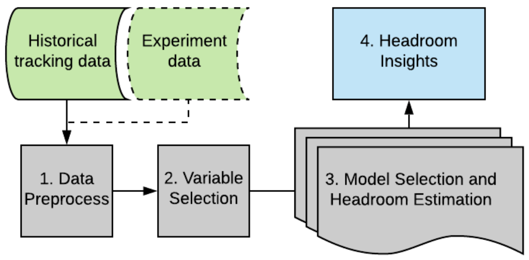

Figure 1: CapPredictor End-to-End Pipeline

3.2 Data Preprocess

In this module, users can define the level of aggregation that they care about. For some use cases, users might be interested in the 99th latency percentile across all machines, because breaking of one machine may not cause a serious issue. In such cases, the data preprocess module could aggregate data from the machine level to the service level, based on user defined aggregation methods. In other cases, breaking of one machine would cause problems on the whole service, therefore understanding the headroom at the machine level is critical.

[4]One way of defining SLOs is for latency at the 99th percentile. That is, an SLO of 10 milliseconds would mean that 99 out of 100 queries should return data within 10ms.

3.3 Variable Selection

There are hundreds of system and workload variables tracked for each service, many of which are highly correlated. For efficiency concerns as well as the ability to explain, it is helpful to reduce the dimension of the variable space if some variables are highly correlated or have limited impact on performance measures. This module provides two sets of results:

- Calculate the Pearson correlation between each variable and the performance variable, and only keep the top *k* influencing variables for investigation.
- Provide groups of highly correlated variables for domain experts to select from. Within each group, the impact of member variables on the performance measures could largely overlap.

3.4 Headroom Estimation and Model Selection

To estimate H^X, the first step is to predict performance changes based on x. We assume

$$y = F(\mathbf{X}) + \varepsilon, \tag{3}$$

where ε denotes some independent observation noise. Our goal is to estimate function $F(\cdot)$, which can be done using historical observed data such as $\{y_i, \mathbf{x}_i\}_{i=1}^N$. Note that, what's special here compared to regular regression problem is that we would like to estimate F beyond the observed domain of $\{\mathbf{x}_i\}_{i=1}^N$.

This is an extrapolation problem, which is in itself challenging since we need to predict beyond the data range we have seen. Machine learning models such as trees are not helpful here since they cannot extrapolate. A good candidate for extrapolation is a linear model, which assumes the form $F(\mathbf{X}) = \beta_0 + \sum_i \beta_i x_i$. Using Ordinary Least Squares (OLS) regression, we estimate the parameters $\beta = (\beta_0, .., \beta_i, ...)$'s through minimizing the squared errors such that $\hat{\beta} = \text{argmin}_\beta \sum_i (y_i - \beta_0 - \sum_i \beta_i x_i))^2$. β_i reflects the speed at which latency increases (decreases) as x_i increases. The higher the β_i's, the faster would y reach to SLO.

Under the linear model assumption, y is assumed to change with $\mathbf{x}$'s with constant speed. This is a rather strict assumption and is usually violated in reality. A natural extension is to assume $F(\mathbf{x}) = \beta_0 + \sum_i \beta_i x_i + \sum_i \gamma_i x_i^2$ to allow nonlinearity. But what if we don't want to specify upfront the parametric format of the model? The generalization is to only assume the smoothness of $F(\cdot)$, which can be estimated using Gaussian Process Regression (GPR) (Williams and Rasmussen 2006).

More specifically, a Gaussian Process defines a distribution over functions, which can be completely specified by its mean function $m(x)$ and covariance function (kernel) $k(x, x')$

$$F(\mathbf{x}) \sim GP(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{4}$$

A Gaussian process is like an infinite-dimensional multivariate Gaussian distribution, where any collection of data observations are jointly Gaussian distributed. We assume $m(\mathbf{x}) = 0$ and use the radial basis function (RBF) kernel to model covariance, which can be parametrized by $\eta = (\phi, \ell_1, ..., \ell_d)$:

$$k_\eta(\mathbf{x}, \mathbf{x}') = \phi^2 \exp\left(-\frac{1}{2}\sum_{i=1}^{d} \frac{(x_i - x_i')^2}{\ell_i^2}\right) \tag{5}$$

which controls the smoothness of the function F. Based on observed data, we can estimate $\hat{F}(\mathbf{x})$ and its confidence bound. The prediction uncertainty would increase with respect to extrapolation distance beyond the $\mathbf{x}$ domain, which can be reflected in larger confidence intervals.

Maximum Load from Confidence Interval No matter using linear or GPR estimation, $F(\cdot)$ only estimates the expected average of latency given a hypothetical x, whereas we usually care more about the more extreme cases, for example, the 90th or 95th percentile. We define the maximum load of variable x, x^*, to be a value such that the probability of y being smaller than SLO is larger than q.

$$P(y < \text{SLO}|x^*) > q \tag{6}$$

If $q = 0.975$, it means the probability of observing a y above the SLO value should be smaller than 2.5%. This is equivalent to requiring the upper bound of the 95% confidence interval of $\hat{y}$ given x^* to be equal to the SLO value. That is how we find x^*.

Figure 2[5] illustrates the estimated and extrapolated mean and confidence interval from a GPR model. The underlying data would be introduced in Section 4. The blue dots indicate data we used for training the model $F(\cdot)$, and the green dots indicate the hold out set. The predicted average latency is represented by the orange line, with its 95% confidence interval in shaded orange. The horizontal dotted red line indicates the SLO. Its intersection with the confidence interval at the upper bound indicates the maximum load x^*. With the maximum load and the current max (or any projected future traffic), we can use equation (1) to calculate headroom.

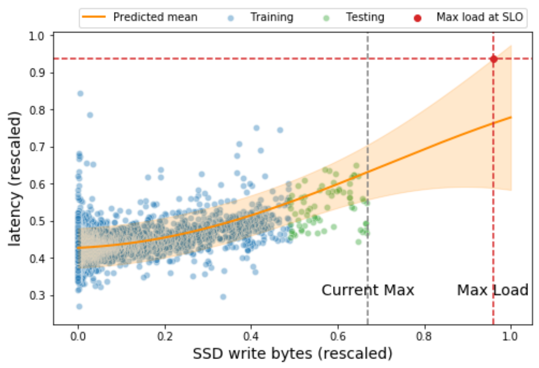

Figure 2: Determine Max Load from Confidence Interval

Deal with distribution Skewness Both OLS and GPR have the assumption that the residuals ε follow a normal distribution. Their prediction power, especially for the confidence interval, would drastically degrade if the dependent

[5]The axes are re-scaled to $(0, 1)$ for illustration purpose only.

variable y is highly skewed, which could be the case once the system is under some stress. Figure 3 shows such a case. For the hold out set (the green triangles), the 90th percentile (P90) is only 0.38, whereas P97.5 is as high as 0.66. The data is highly skewed. If we use GPR on the raw data, we will predict a very narrow 95% confidence interval centered at the mean, with the P97.5 estimated at 0.10.

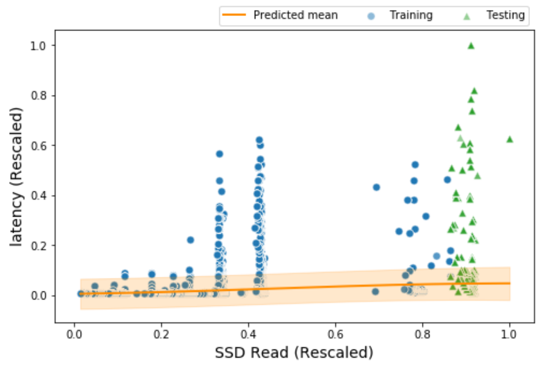

Figure 3: Confidence Interval Estimation on Skewed Data

To address this issue, we formulate a module called "rolling quantiles generator" which process the data in 2 steps:

1. Sample N data points from raw data. Rank the sample along x, take a rolling window of T and get the quantile q of the rolling window for each x value. Call this newly generated feature the "rolling quantile".
2. Repeat step 1 for B times. Pool the sampled data together.

After this module, we feed the "rolling quantiles" as y variables into the headroom estimation model. Figure 4 shows the 97.5th percentile estimation using the "rolling quantiles" method on the same data as in Figure 3. It is much closer to the true 97.5th percentile in the data.

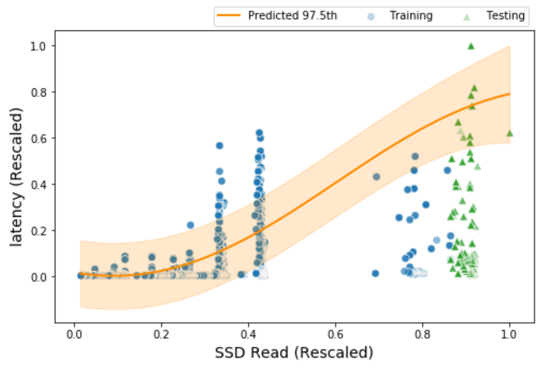

Figure 4: Rolling Quantile Estimation on Skewed Data

To decide whether to incur the "rolling quantiles generator", we calculate the skewness of y and use a threshold[6] to decide whether the data is skewed. This step is illustrated in the left panel of Figure 5.

Evaluation We now discuss how we select out the best model. As illustrated in Figures 2-4, when training, we can hold out the proportion of data with higher x values. On the testing domain of x, we calculate recall and precision based on the overlap ratio between the predicted distribution range and the actual range of the performance variable.

$$\text{Recall} = \frac{\sum_j I(y_j \in \hat{C} \cap y \in C)}{\sum_j I(y_j \in C)} \tag{7}$$

$$\text{Precision} = \frac{\sum_j I(y_j \in C \cap y \in \hat{C})}{\sum_j I(y_j \in \hat{C})} \tag{8}$$

$\hat{C}$ is the estimated distribution range of $\hat{y}$ on the testing set, and C is the actual distribution range of y. For example, let τ denote percentile, then $C = [\tau_{25}, \tau_{75}]$ if we are looking at the range between the 25th and 75th percentile.

Finally, we can combine recall and precision into the F1 score to get our final evaluation metric for model selection, and the final model to be used would be the one with the highest F1 score on the hold out set.

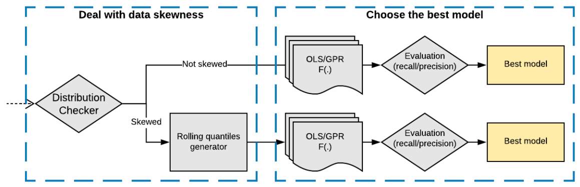

Figure 5: Inside Headroom Estimation and Model Selection

3.5 Headroom Insights

We have now generated the headroom numbers for variables **X**. To better understand the numbers, there are two follow-up questions.

First, what we cannot explain? The residual ε from equation (3) contains all the information that cannot yet be explained. Looking at Figure 4, we want to understand given an x value, why latency has such high dispersion? This problem is no longer an extrapolation task so we can resort to more complicated non-parametric models such as trees, to help us understand the effects coming from any additional workload/system variables given the level of the extrapolation variable x.

Second, we want to distinguish the most influencing variable from the workload variable we care about. Often in reality, the workload variable that we care most about is not necessarily the variable that is most correlated with latency. In this case, we need one more step to link the variable we care about to the variables that matter most.

For example, we have found that for some key-value stores, solid state drive (SSD) I/O is driving read latency: We can always identify SSD write or read as the most important

[6]The default skewness threshold is set at 1 as the rule of thumb. Users are free to tune it.

system variables on read latency, but not the read QPS itself. After comparing results from different data stores, we found that what matters is whether the underlying data has enough distinct keys which requires intensive SSD I/O operations. This understanding helps us more accurately provide headroom numbers given different use case characteristics.

4 Case Study

In this section, we illustrate a case study on one data service at LinkedIn, Venice. Venice (Yan 2017) is a distributed key-value data serving platform. It has two tiers when serving data: the router layer distributes queries to the server layer, which contains storage nodes that hold the corresponding partition of data, and returns the stored values.

The router and server variables can be collected at 5-minute intervals. Table 1 shows a list of the important workload and system variables that are tracked.

Variable	Explanation
Latency	99 percentile among all queries within 5 min * We focus on *read latency* throughout
Read QPS	Single get or batch get queries per second
Read KPS	Keys queried per second
Throughput	Including inbound and outbound
SSD I/O	SSD write/read in bytes
GC	Garbage collection related variables
JVM threads	JVM thread count
Memory	Percentage of free memory

Table 1: Important workload/system variables for Venice

We show results for two data sets. The first one, which we call ***production data***, are collected for a Venice production cluster in July 2019. As mentioned in Section 1, it is hard to do load tests for stateful services. Therefore other than the hold out testing data set, there is very little we can do on production to validate our model. To better validate and stress the model under different scenarios, we resort to the experiment cluster, which is comprised of a set of machines used for testing. We try to mimic real production traffic and system settings on this cluster, and generate synthetic SSD I/O and data queries. In particular, we pushed SSD read to be much higher than what we see in production, while holding SSD write on par with production[7]. We also put read KPS to levels much higher than production. The data collected from this experiment setting is called ***experiment data***.

4.1 Results

Figure 6 shows the feature selection module for the ***production data***. Through simple correlation, we see that data push related variables affect read latency the most. Therefore we illustrate the headroom estimation for the top influencing variable, SSD Write Bytes, in Figure 2. Figure 7 shows our fit of the relationship between read latency and read KPS, which is a direct measure of traffic directed to Venice.

From Figure 2, we see a clear trend between SSD write and latency (read). For read KPS, although there is an upward trend at the very top of the observed workload, the confidence interval is rather wide. Overall, there does not seem to be a clear pattern between read latency and read traffic. This seemingly surprising finding turns out to be consistent with the service constraint. Venice is data push heavy (SSD write bytes being 10 times that of read), and during each batch push, there would be high GC which blocks data read, causing read latency to increase (Liu 2018).

[7]In production, SSD write in bytes is 10 times that of SSD read.

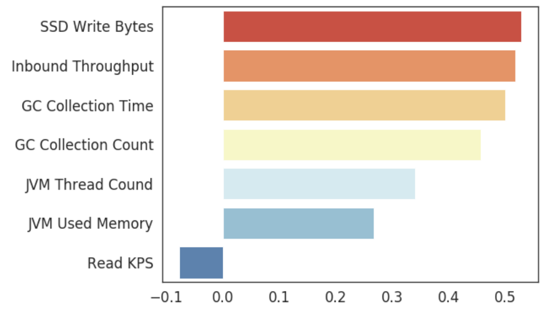

Figure 6: Feature Report for Production Data

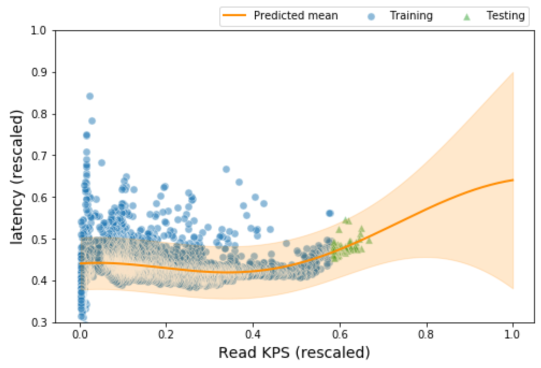

Figure 7: Latency v.s. Read KPS on Production Data

Variable	Recall	Precision
Production Data		
Read KPS	1.00	0.29
SSD Write	0.92	0.92
Experiment Data		
Read KPS	1.00	1.00
SSD Read	0.95	0.98

Table 2: Evaluation Metrics

Table 2 shows the evaluation metrics (see definitions in Section 3.4). We look at recall and precision in terms of the overlap ratio of the 2.5 - 97.5 distribution range. For read KPS, since the latency interval of the test data can all be covered by the predicted interval, recall is 1. However, precision is very low, echoing a wide predicted range, which indicates high uncertainty from the extrapolation. For SSD write, on the other hand, we see high recall and precision.

On the experiment cluster, we deliberately stress the

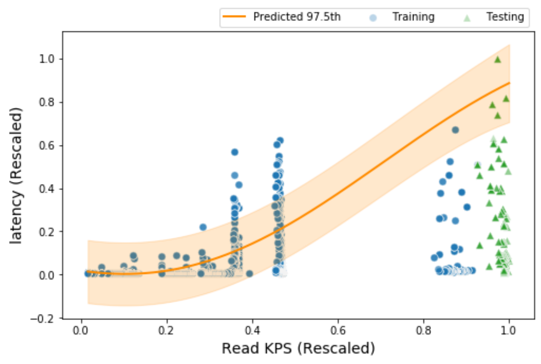

Figure 8: Latency v.s. Read KPS on Experiment Data

server nodes with SSD read. The feature report now does rank SSD read related variables as the top features. And as shown in Figures 8 and 4, both SSD Read and Read KPS now affect latency significantly. Looking at Table 2, we also see both high precision and recall[8].

5 Conclusion and Future Work

In this paper, we propose CapPredictor, an generic and end-to-end machine learning framework that predicts performance based capacity headroom for stateful services in cloud. CapPredictor has been applied to several stateful data services at LinkedIn, such as Venice(Yan 2017), Pinot(Im et al. 2018), and Espresso(Auradkar 2015). In addition to headroom prediction, CapPredictor also provides insights into the key variables that affect service performance. For example, we found that key-value stores are usually SSD I/O bound, whereas SQL (or no-SQL) based stores are usually CPU bound. Thus, understanding how read/write traffic affect SSD I/O or CPU computation is important for providing further actionable suggestions.

In the near future, we plan to extend CapPredictor in the following aspects. First, tunable hardware configurations could be added as system variables, enabling configuration recommendations under over- or under-provision capacity settings. Second, multi-variable smooth function could be utlized in CapPredictor and hence to improve headroom prediction. We would also like to use anomaly detection techniques to detect and explain the anomalous signals which are quite common in production data.

References

Auradkar, A. 2015. LinkedIn's Hot New Distributed Document Store.

Boone, R. 2018. "Capacity Prediction" instead of "Capacity Planning": How Uber Uses ML to Accurately Forecast Resource Utilization.

Chen, Y.; Yang, X.; Lin, Q.; Zhang, H.; Gao, F.; Xu, Z.; Dang, Y.; Zhang, D.; Dong, H.; Xu, Y.; et al. 2019. Outage Prediction and Diagnosis for Cloud Service Systems. In *The World Wide Web Conference*, 2659–2665. ACM.

Dang, Y.; Lin, Q.; and Huang, P. 2019. AIOps: real-world challenges and research innovations. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, 4–5. IEEE Press.

Im, J.-F.; Gopalakrishna, K.; Subramaniam, S.; Shrivastava, M.; Tumbde, A.; Jiang, X.; Dai, J.; Lee, S.; Pawar, N.; Li, J.; et al. 2018. Pinot: Realtime OLAP for 530 Million Users. In *Proceedings of the 2018 International Conference on Management of Data*, 583–594. ACM.

Kim, G.; Humble, J.; Debois, P.; and Willis, J. 2016. *The DevOps Handbook:: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution.

Krazit, T. 2017. How Amazon Web Services Uses Machine Learning to Make Capacity Planning Decisions.

Krishnamurthy, R., and Kelkar, A. 2018. Building Data Science Teams to Have an Impact at Scale.

Lerner, A. 2017. AIOps Platforms.

Li, Z., and Dang, Y. 2019. AIOps: Challenges and Experiences in Azure.

Li, J.; König, A. C.; Narasayya, V.; and Chaudhuri, S. 2012. Robust Estimation of Resource Consumption for SQL Queries Using Statistical Techniques. *Proceedings of the VLDB Endowment* 5(11):1555–1566.

Liu, Z.; Nath, A. K.; Ding, X.; Fu, H.; Khan, M. M.; and Yu, W. 2019. Multivariate Modeling and Two-Level Scheduling of Analytic Queries. *Parallel Computing* 85:66–78.

Liu, G. 2018. Venice Performance Optimization.

Llamas, R. M. 2016. Capacity Planning at Scale. Dublin: USENIX Association.

Marcus, R., and Papaemmanouil, O. 2019. Plan-structured Deep Neural Network Models for Query Performance Prediction. *arXiv preprint arXiv:1902.00132*.

Menasce, D. A., and Almeida, V. 2001. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR.

Sun, J., and Li, G. 2019. An End-to-End Learning-based Cost Estimator. *arXiv preprint arXiv:1906.02560*.

Williams, C. K., and Rasmussen, C. E. 2006. *Gaussian Processes for Machine Learning*, volume 2. MIT press Cambridge, MA.

Xia, S., and Rao, A. 2017. Redliner: How LinkedIn Determines the Capacity Limits of Its Services.

Xu, L.; Cole, R. L.; and Ting, D. 2019. Learning to Optimize Federated Queries. In *aiDM@ SIGMOD*, 2–1.

Yan, Y. 2017. Building Venice with Apache Helix.

Zhuang, Z.; Ramachandra, H.; Tran, C.; Subramaniam, S.; Botev, C.; Xiong, C.; and Sridharan, B. 2015. Capacity Planning and Headroom Analysis for Taming Database Replication Latency: Experiences with Linkedin Internet Traffic. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 39–50. ACM.

[8]Note that in Figures 8 and 4, the orange lines illustrate the fit of the upper bound of the distribution range. Thus the orange region does not correspond to the estimated distribution range.