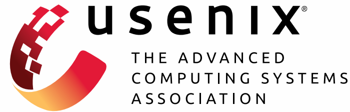


Metis: Robustly Tuning Tail Latencies of Cloud Systems

Zhao Lucis Li, *USTC;* Chieh-Jan Mike Liang, *Microsoft Research;* Wenjia He, *USTC;* Lianjie Zhu, Wenjun Dai, and Jin Jiang, *Microsoft Bing Ads;* Guangzhong Sun, *USTC*

https://www.usenix.org/conference/atc18/presentation/li-zhao

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Metis: Robustly Optimizing Tail Latencies of Cloud Systems

Zhao Lucis Li[⋆‡] Chieh-Jan Mike Liang[‡] Wenjia He[⋆‡] Lianjie Zhu[○] Wenjun Dai[○]
Jin Jiang[○] Guangzhong Sun[⋆]
[⋆]*USTC* [‡]*Microsoft Research* [○]*Microsoft Bing Ads*

Abstract

Tuning configurations is essential for operating modern cloud systems, but the difficulty arises from the cloud system's diverse workloads, large system scale, and vast parameter space. Building on previous space exploration efforts of searching for the optimal system configuration, we argue that cloud systems introduce challenges to the robustness of auto-tuning. First, performance metrics such as tail latencies can be sensitive to non-trivial noises. Second, while treating target systems as a black box promotes applicability, it complicates the goal of balancing exploitation and exploration. To this end, Metis is an auto-tuning service used by several Microsoft services, and it implements customized Bayesian optimization to robustly improve auto-tuning: (1) diagnostic models to find potential data outliers for re-sampling, and (2) a mixture of acquisition functions to balance exploitation, exploration and re-sampling. This paper uses Bing Ads key-value store clusters as the running example – compared to weeks of manual tuning by human experts, production results show that Metis reduces the overall tuning time by 98.41%, while reducing the 99-percentile latency by another 3.43%.

1 Introduction

For many web-scale cloud systems, main evaluation metric is tail latencies (e.g., 99 and 99.9-percentile latencies) of serving a request [9]. While tail latencies seem rare, the probability of a user request experiencing the tail latency in an end-to-end system can be high, especially that most web-scale applications employ a multi-stage architecture. Imagine a web-scale application with a five-stage processing pipeline, the probability of a request encountering 99-percentile latency at least is ∼5%. Furthermore, tail latencies can be more than 10 times higher, as compared to the average latency [9].

Recently, advances in machine learning have spawned strong interests in automating system customizations, where auto-tuning system configuration of parameters is a popular scenario [2, 3, 20]. Notably, cloud systems exhibit two motivating characteristics for auto-tuning. First, the overhead of operating cloud systems is increasingly larger, due to the increasingly more dynamic and variable system workloads, and the scale of modern cloud systems. In many read-intensive web applications such as news sites and advertisement networks, data queries are tied to end-user personal interests and current contexts, which can exhibit temporal dynamics. Furthermore, even within one cloud system, there can be several components that individually impose different data requirement and handle different types of meta data and user data. Second, as cloud systems become more complicated, both design space and parameter space grow significantly. The optimal system configuration is beyond what human operators can efficiently and effectively reason about, and the cost to naïvely benchmark all possible system configurations is exorbitant.

Bayesian optimization (BO) with Gaussian process (GP) has emerged as a powerful black-box optimization framework for system customizations [2, 3, 19]. Mathematically, we model the configuration-vs-performance space by regressing over data points already collected, i.e., system configurations benchmarked. And, this regression model allows us to estimate the global optimum, or the best-performing system configuration. While collecting more data points can improve the regression model, benchmarking one system configuration of parameters can be time-consuming due to system warm-up and stabilization. Fortunately, BO offers a way to iteratively build up the training data by suggesting system configurations to benchmark, with the goal to maximally

Chieh-Jan Mike Liang is the corresponding author. This work was done when Zhao Lucis Li and Wenjia He were interns at Microsoft Research.

improve the regression model accuracy.

1.1 Contributions

In this paper, we report the design, implementation, and deployment of *Metis*, an auto-tuning service used by several Microsoft services for robust system tuning. While Metis is inspired by previous efforts to leverage BO to train the GP regression model, we address the following factors to improve the robustness of optimizing system customizations.

First, since the GP regression model is trained with data points from benchmarking some system configurations, how well these data points capture the configuration-vs-performance space's global optimum determines the auto-tuning effectiveness. At the same time, we should avoid unnecessarily over-sampling the space, as system benchmarking can be resource-intensive and time-consuming. To this end, at each iteration, BO's strategy for picking the next system configuration to benchmark should balance *exploitation* (i.e., regions with high probability of containing optimum) and *exploration* (i.e., regions with high uncertainty of containing optimum). Specifically, inadequate exploration reduces the chance of moving away from a local optimum, and inadequate exploitation impacts the efficiency of identifying the global optimum. In contrast to related efforts that rely on simple-to-implement strategies [2, 3], Metis strongly decouples exploitation and exploration, so that it can independently evaluate each action's expected improvement and anticipated regret.

Second, the theory behind BO and GP mostly assumes the data collection is reasonably noise-free, or susceptible to only the Gaussian noise. Unfortunately, many system performance metrics (e.g., tail latencies) are sensitive to non-Gaussian or unstructured noise sources in the wild, e.g., CPU scheduling and OS updates. In contrast to related efforts [2, 3], not only does Metis consider exploitation and exploration, but it also weighs the benefit of *re-sampling* existing data points. Key enabler is the diagnostic model that Metis creates to identify potential outliers.

To demonstrate the usefulness of Metis for real-world system black-box optimizations, this paper showcases one of our production deployments as the running example – optimizing tail latencies of Microsoft Bing Ads key-value (KV) stores. We present measurements and experiences from two KV clusters that handle ∼4.14 billion and ∼3.18 billion key-value queries per day, respectively. Compared to weeks of manual tuning by human experts, production results show that Metis reduces the overall tuning time by 98.41%, while reducing the 99-percentile latency by another 3.43%.

2 Background and Motivation

As a black-box optimization service, Metis tunes the configuration of several Microsoft services and networking infrastructure. This section describes one deployment as the running example in this paper – Microsoft Bing Ads key-value (KV) store cluster, *BingKV*. Then, we motivate optimally customizing cloud systems.

2.1 Running Example: *BingKV*

Individual stages of the ads query processing pipeline, e.g., selection and ranking, host separate KV store clusters. Within the cluster, each KV server is responsible for one non-overlapping dataset partition. Therefore, each server can experience different workload, as defined by the frequency distribution and the size distribution of top queried key-value objects. Manually tuning each server is difficult as a cluster can have thousands of servers.

The configuration space consists of parameters of the local caching mechanism, which decides what KV objects should be cached in the in-memory cache or served from the persistent data store. Main evaluation metric is tail latencies of serving key queries after the in-memory cache is full. We describe these parameters and their typical value ranges next.

First, both the recency and frequency distribution are well-known foundation for cache eviction. BingKV supports `NumCacheLevels` (1 - 10) levels of LRU (Least Recently Used) or LFU (Least Frequently Used) based caches – a cached object can move up a level if it has been queried `CachePromotionThreshold` (1 - 1,000) times. Since cached objects at higher levels have been queried more than those at lower levels, locality principle implies that they should not be easily evicted. Therefore, we allow `NumInevictableLevels` (0 - 9) levels to be specified as being inevictable.

Second, many cache designs incorporate a shadow buffer that stores the key only, instead of the entire key-value object. BingKV implements a shadow buffer with a capacity of `ShadowCapacity` (1 - 10) MB, to hold keys with an object size larger than `AdmissionThreshold` (1 - 1,000) bytes. Then, keys in the shadow buffer are moved to the cache only if they have been queried more than `ShadowPromotionFreq` (1 - 1,000) times.

Finally, `CacheCapacity` (1 - 10,000) specifies the cache capacity (excluding the shadow buffer) in MB, and the dataset partition on each server is divided into `NumShards` (1 - 64) shards.

2.2 Impacts of Suboptimal Configurations

To illustrate impacts of suboptimal configurations, we empirically measure BingKV's 99-percentile latency un-

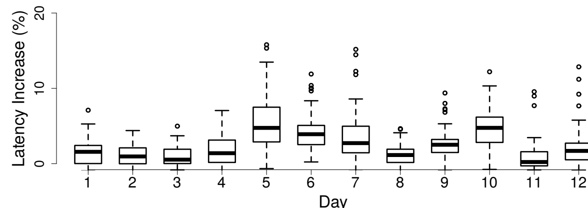


Figure 1: Motivating example – suboptimal configurations can impact the system performance. We use BingKV as the running example, and compare the tail latency increase with respect to the optimal system configuration.

der different parameter configurations, and quantify the tail latency increase with respect to the best-performing system configuration. Experiments use two 12-day pre-recorded production workload traces of BingKV, $Prod_Trace_1$ and $Prod_Trace_2$, and replay these workload traces to benchmark system configurations.

Suboptimal configurations can happen when system tuning does not consider machine-specific setup. To illustrate, we vary the cache capacity between 32 MB and 512 MB – we find the best-performing configuration (e.g., `AdmissionThreshold` ranging from 1 to 1,000) for each capacity, and then we compare the latency of the best-performing configuration to 100 configurations uniformly sampled from the parameter space. Results show that the average 99-percentile latency increase can range from 15.34% (for 256-MB cache) to 22.74% (for 512-MB cache), with more than 34.14% increase in the worst case. For $Prod_Trace_2$, the average 99-percentile latency increase can range from 10.21% (for 32-MB cache) to 13.89% (for 64-MB cache), with more than 13% increase in the worst case.

Suboptimal configurations can also happen when system tuning does not consider temporal dynamics. For each day of $Prod_Trace_1$, we compare the best-performing configuration to that of the other 11 days. Figure 1 shows that the average 99-percentile latency increase can be as high as more than 5.58% (day 5).

2.3 Strawman Solutions

Manual tuning. Manual tuning can leverage administrators' knowledge and prior experience about the target system. Unfortunately, as modern systems get larger and more complicated, reasoning the system behavior in a high-dimensional configuration space becomes increasingly difficult. Furthermore, real-world workload can have dynamics across machines and time, and manual tuning does not scale. To illustrate this argument, our experience shows that manually tuning a subset of BingKV can take weeks. And, in some cases, it is not certain whether the manual tuned configuration is indeed the best-performing one.

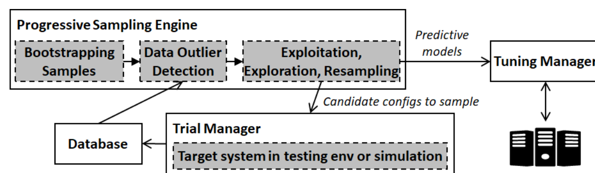


Figure 2: Architecture of Metis service.

Naïve Bayesian optimization. OtterTune [2] and CherryPick [3] demonstrated the potential of Bayesian optimization in adaptively finding the best-performing configuration for databases and data analytics systems, respectively. Both approaches adopt common BO selection strategies such as Expected-Improvement (EI). While these strategies are easy-to-implement, they do not consider factors impacting the robustness of tuning system customizations: the balance of exploitation and exploration [14], and data outliers.

3 Our Approach

Given a workload of a system, Metis has the objective of predicting the best-performing configuration by selectively exploring the configuration-vs-performance space. Through system benchmarking, Metis can collect data points that describe system inputs (e.g., system workload and parameter values) and outputs (e.g., performance metrics of interests) for training its model. When Metis does not change its prediction of the best-performing configuration with additional data points, we say the model has converged. Maximizing the prediction accuracy and minimizing the model convergence time are two evaluation metrics for Metis. The former relates to the auto-tuning correctness, and the latter relates to the service scalability.

This section first outlines the usage flow of Metis. Then, we formulate auto-tuning as an optimization problem, and highlight practical challenges to motivate customizations proposed in subsequent sections.

3.1 Architectural Overview

Figure 2 shows the architecture of Metis containing components for model training and system tuning.

Model training. The Progressive Sampling Engine solves the optimization problem of robustly constructing the predictive regression model, which models the configuration-vs-performance space. Each model corresponds to one performance metric, and the model dimensionality depends on the number of parameters.

Users start by providing the Progressive Sampling Engine their requirements (e.g., parameters and metric of interest) and workload traces (e.g., production or synthesized traces). The engine bootstraps BO by picking a set of system configurations as bootstrapping trials, for the Trial Manager to benchmark. Benchmarking can happen in simulators or real machines. Then, using performance metrics collected as the training data points, the engine picks subsequent system configuration of parameters to benchmark. Ideally, each iteration should improve the regression model's estimation of the space's global optimum. We formulate this iterative process as an optimization problem in the next subsection (c.f. §3.2).

When a stopping criterion is met, the Progressive Sampling Engine stops collecting more data points, and it outputs the GP regression model trained so far. Stopping criteria can include training time budget and so on. As we show in §6, the system benchmarking time dominates training, rather than the model computation time.

System tuning. The Tuning Manager periodically receives status reports from individual servers of the target system. These status reports contain current workload characterizations and logged performance metrics. If performance degradation is detected (e.g., a significant increase in the tail latency), the Tuning Manager uses the workload characterization to select the nearest regression model. We use DNNs to classify workloads, which minimizes the burden of weighing workload features for classification.

3.2 Optimization Problem Formulation

Formally, for a workload w, we want to find the k-dimensional configuration c^* (representing k system parameters), which has the highest probability of being the best-performing configuration, c_w. The *overall problem objective* is as follows:

$$c^* = \underset{c \in configs}{\text{argmax}} \; P(c = c_w \mid w)$$

Given the k-dimensional parameter space can be too large for exhaustive searches, Metis opts the regression model to predict with limited amount of data points collected. While more training data will help reducing the regression uncertainty, the *training objective* should also consider the training overhead:

$$\begin{aligned} \text{minimize} \sum_{w \in workloads} & (confidence_interval(c_w)) \\ \text{subject to} \sum T(c) & \leq T_{budget} \end{aligned}$$

$T(c)$ denotes the time necessary to sample a configuration c, and T_{budget} denotes the maximum amount of time cost allocated for model training.

Predictive regression model formulation. Regression allows Metis to model the expected configuration-vs-performance space, with data points already collected from benchmarking some configurations. The regression model captures the conditional distribution of a target performance metric given a system configuration. We pick Gaussian process (GP) [13] as the model, which extends multivariate Gaussian distributions to infinite dimensionality, such that observations for an unsampled data point are assumed to follow a multivariate Gaussian distribution. In other words, assuming a stochastic function f where every input x has an output $f(x)$, each $f(x)$ is defined as a mean μ and a standard deviation σ of a Gaussian distribution.

GP exhibits a number of desirable properties. First, it does not assume a certain mathematical relationship between model inputs and outputs, e.g., the linear relationship. Second, for any x, GP can return the expected value of $f(x)$ and uncertainty (i.e., standard deviation).

Optimization framework. A proven approach to train the GP model is *Bayesian optimization* (BO) [17]. At each iteration, based on the current GP model, BO selects a system configuration to benchmark next to further train the GP model. The selected configuration is expected to maximally improve the accuracy predicting the global optimum of the configuration-vs-performance space. The logic behind selecting the next trial is implemented by BO's acquisition function, and its design traditionally aims to balance exploitation (i.e., sampling regions with high probability of containing optimum) and exploration (i.e., sampling regions with high uncertainty). The model converges when BO believes that the GP model has sufficiently captured the configuration-vs-performance space global optimum, or the best-performing system configuration in our case.

BO conceptually realizes a form of progressive sampling, and it is suitable for scenarios where collecting system performance metrics of a single trial is resource-intensive or time-consuming (e.g., waiting for a system to warm up and stabilize).

3.3 Practical Challenges of Robustness

Running BO with GP requires the following practical considerations for robust system tuning.

Sampling configuration-vs-performance space without strong assumptions on the target system behavior. Given that system behavior can be difficult to be properly reasoned, Metis tries to learn the configuration-vs-performance space from selective sampling. In other words, based on system configurations already benchmarked, Metis selects the next system configuration to benchmark, which is expected to maximally improve the

regression accuracy. To this end, we believe that existing efforts leave the following two gaps to fill.

First, balancing exploitation and exploration is still a non-trivial problem. Many proposed acquisition functions evaluate the potential improvement of a trial [19], and the community has shown their limitations in balancing exploitation and exploration [4, 17]. Expected Improvement (EI) is a widely popular choice of acquisition functions. It improves upon Maximum Probability of Improvement, by estimating the best-case improvement with both the mean and the uncertainty around a sampling point. However, Ryzhov et al. [14] showed that EI allocates only O(log n) samples to regions that are expected to be suboptimal, where n is the total number of trials. In other words, n needs to be significantly large for EI to balance exploitation and exploration.

Second, bootstrapping trials refer to the set of sampling points to initialize BO. Since BO decides the next trial with the GP model trained with past trial data points, prior data samples can influence how BO expects the posterior expectation to be. The main requirement for selecting bootstrapping trials should be exploration. Random sampling is a simple approach to pick bootstrapping trials, but it requires a large amount of sampling points to ensure coverage [12]. This is not ideal for time-consuming trials, where some systems need time to stabilize (e.g., system cache warm-up).

Handling non-Gaussian data noise. Most theoretical work on BO with GP assumes the trial data collected are noise-free (i.e., perfectly reproducible experiments), or susceptible to only the Gaussian noise [17]. Unfortunately, many system metrics such as tail latencies are sensitive to a wide range of noise sources in the real world, ranging from background daemons, local resource sharing, networking variability, etc. This is different from reproducible metrics such as classification accuracy. Since real-world data noise sources can be heterogeneous and non-trivial to model, the auto-tuning service should consider the benefits of removing potential outliers.

4 Improving Auto-Tuning Robustness

Given the challenges discussed in §3.3, we now discuss our customizations to Bayesian optimization to improve tuning robustness.

4.1 Bootstrapping Trials

To select sampling points to bootstrap Bayesian optimization for a given workload, Metis exercises Latin Hypercube Sampling (LHS) [12]. LHS is a type of stratified sampling. According to some assumed probability distribution, LHS divides the range of each of M parameters into I equally probable intervals, and randomly selects only *one* single data point from each interval. Since each interval of each parameter is selected only once, the number of bootstrapping trials chosen by LHS is exactly I, regardless of M. Furthermore, the maximum number of possible combinations for bootstrapping trials is only $(\prod_{i=0}^{I-1} I-i)^{M-1} = (I!)^{M-1}$.

LHS offers several desirable properties for bootstrapping Bayesian optimization. First, it has been shown that, compared to random sampling, stratified sampling can reduce the sample size required to achieve the same conclusion [12]. Second, compared to another well-known stratified sampling technique, Monte Carlo, LHS allows one to obtain a stable output with less samples [7]. And, while quasi-Monte Carlo can be an alternative approach, its output is a low discrepancy sequence which is not random, but uniformly deterministic [6]. Third, as LHS does not control the sampling of combinations of dimensions, the number of samples picked LHS is agnostic to the dimensionality of the parameter space.

A well-known concern of LHS is that it may exhibit a higher memory consumption, in order to keep track of parameter intervals that have already been sampled. However, we note that only a few sampling points are necessary to bootstrap Metis, and our current running system sets I to be 5.

4.2 Customized Acquisition Function

After obtaining bootstrapping sampling trials with LHS, Metis then runs Bayesian optimization to iteratively train the Gaussian process model. At each iteration, BO outputs the system configuration that is expected to offer the most improvement to the GP model, in terms of predicting the best-performing system configuration. This output represents the system configuration that Metis should sample in the next iteration of training. To produce this output, Metis customizes the acquisition function to balance three goals: *exploitation*, *exploration*, and *re-sampling*.

Our customized acquisition function works as a mixture of three separate sub-acquisition functions (c.f. §4.2.1) corresponding to each goal. At each iteration of BO, based on the GP model constructed so far, individual sub-acquisition functions compute the next system configuration to sample to maximize their own goal. Then, the acquisition function evaluates these candidates in terms of the improvement of the expected best-performing system configuration, and selects the candidate with the highest gain to sample next (c.f. §4.2.2).

Compared to related efforts, our acquisition function exhibits two differences. First, by introducing re-sampling as a possible action, we allow Metis to re-

sample a trial whose previous results might be susceptible to non-Gaussian noise. Second, Metis decouples all three actions and runs separate acquisition functions for each action. This decoupling allows Metis to independently evaluate the potential information gain and regret aversion from taking each action in the next iteration.

4.2.1 Sub-acquisition functions

Metis decouples exploitation, exploration, and re-sampling into three separate sub-acquisition functions. Given past trials, each sub-acquisition function outputs a system configuration to maximize its own goal. These system configurations become candidates for the acquisition function.

Exploration. This sub-acquisition function looks for the sampling point whose expected observation would have with the highest uncertainty. Formally, in the GP regression model, this sampling point would exhibit the largest confidence interval.

Exploitation. This sub-acquisition function looks for the sampling point whose expected observation would most likely be the system optimum. While one simple approach is to consider the absolute observation of a sampling point as expected by the GP regression model, its accuracy can be impacted by GP's uncertainty of that sampling point. Therefore, Metis takes an approach inspired by TPE [5], and tries to estimate the probability of a sampling point giving near-optimum observation.

The exploitation sub-acquisition function works as follows. It first separates the past trials into two groups: the first group has best observations, and the second group has the rest. Then, we construct two Gaussian mixture models (GMM) to describe each group. With these two GMMs, we can evaluate the likelihood of a system configuration, c, being in either group, i.e., $P_1(c)$ and $P_2(c)$. The ratio of these two likelihood, or $\frac{P_1(c)}{P_2(c)}$, would give us a score of how likely a sampling point is in the first group, instead of second group.

Re-sampling. This sub-acquisition function looks for outliers in past trials, and suggest trials that should be re-sampled (i.e., system configurations that should be re-benchmarked).

The re-sampling sub-acquisition function works by creating the diagnostic model. To examine a data point, the diagnostic model quantifies the difference between the measured value and the expected value. To do so, the diagnostic model is a GP regression model trained without the examined data point. Then, the diagnostic model can calculate the expected value and 95% confidence interval. If the measured value falls outside the confidence interval, it is probable that the examined data point could be an outlier. The sub-acquisition function repeats the evaluation for all past trials, and it selects the trial that is farthest from the expected confidence interval.

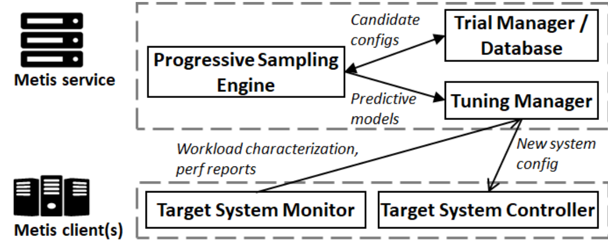


Figure 3: Components of Metis service implementation.

4.2.2 Evaluation of Candidates

Taking sub-acquisition function outputs as candidate configurations, our acquisition function computes their information gain with respect to how the prediction of optimal configuration changes. Conceptually, for each candidate, the GP model bounds the expected observation with a confidence interval. And, the upper and lower bound of this interval can help us estimate the potential information gain (if we were to actually select the corresponding candidate for benchmark).

The acquisition function starts by predicting the currently best-performing system configuration, c_{cur_best}, and this step searches for the sampling point with the lowest expected observation in the GP regression model. Then, to evaluate a candidate, $c_{candidate}$, we add its lower-bound confidence interval (i.e., best-case) to the GP regression model, and predict the would-be best-performing system configuration, c_{new_best}. The different between the expected observation of c_{cur_best} and c_{new_best} is the improvement, and 0 if the improvement is negative. Then, we repeat the process for upper-bound confidence interval (i.e., worst-case).

Finally, the acquisition function outputs the $c_{candidate}$ with the highest sum of improvement calculated with the lower-bound and the upper-bound confidence interval.

5 Implementation

Figure 3 illustrates the Metis system components implemented. Our current implementation is in Python, and this language choice allows us to use the popular Scikit-learn library for Gaussian process regression [16]. We choose the summation of Matern (3/2) and white noise as the covariance kernel [19].

Separation of logic and execution. Metis consists of a centralized web service and client stubs that sit on servers hosting the target system. Network communications happen over TCP, and the message format is JSON. Conceptually, the client stub hooks up to the target system,

and it abstracts away system-specific interfaces and execution from the web services. While an alternative implementation is to place both the logic and execution in a local service on each server, the separation provides several benefits. First, the computation-intensive logic of Metis does not contend for resources on production servers. Second, being centralized, Metis has the global view of all status reports from all servers. This allows Metis to continuously improve the predictive model with new data points.

Metis web service. The web service trains one predictive regression model for each workload trace. And, these traces can be recorded in the production environment, or synthesized with tools such as YCSB [8]. Trial Manager replays each workload trace in simulators or real servers, and each replay represents one trial benchmarking one selected system configuration. Then, the training dataset consists of system configurations and corresponding performance benchmarks. In the case of BingKV, we built a simulator wrapping the production KV code binaries, to run trials.

From status reports uploaded by client stubs, if Tuning Manager detects that a server is experiencing a performance degradation above some user-specified thresholds (e.g., tail latencies have increased by more than 10% in the last six hours), it computes a new configuration and sends a command to the server's client stub.

Metis client stubs. The client stub deals with system-specific interfaces, ***(1)*** to periodically upload recent system performance measurements and workload characterization, and ***(2)*** to execute configuration change commands from the web service.

Most web-scale cloud systems already have an extensive logging mechanism for continuous performance monitoring, and Metis client stubs periodically retrieve relevant performance measurements from the same logging mechanism. While different systems have different workload characterization features, these features are either already available in the logging mechanism, or require additional code instrumentation. In the case of BingKV, our workload features are the size and query frequency of the top queried KV objects, and the incoming query traffic rate.

Upon receiving a configuration change command, Metis does not try to aggressively re-configure servers. Instead, it employs a guard time following a re-configuration to allow the target system to warm up and stabilize. Other than time durations, this guard time can also be values of system states, e.g., cache occupancy.

	Selection of Configs	Modeling	DO
Random	Random		No
iTuned [10]	LHS + EI	BO w/ GP	No
CherryPick [3]	Random + EI	BO w/ GP	No
TPE [5]	Random + EI	GMM	No
Metis	LHS + Customized	BO w/ GP	Yes

Table 1: This table highlights differences among comparison baselines and Metis. The "DO" column shows whether data outlier removal is considered.

6 Evaluation

Our major results include – ***(1)*** compared to baselines, Metis picks better configurations 84.67% of times in the presence of low data noises. ***(2)*** In the presence of data outliers, Metis has 56.77% more chance of picking better configuration. ***(3)*** Metis has a faster convergence time in searching for the best-performing configuration.

6.1 Methodology

We use the Bing Ads KV store (c.f. §2) as the target system. Our testbed machines have two 2.1 GHz CPUs (with 16 cores), 16 GB RAM, and a 256 GB SSD. These machines host the target system binaries, and simulate configurable key-value query arrival rates (or queries per second) for the given workload trace. We log performance metrics with asynchronous I/Os to minimize any artifact introduced by logging.

Datasets and workload traces. We obtained two 12-day key-value query traces, $Prod_Trace_1$ and $Prod_Trace_2$, from two Bing Ads KV store clusters, $BingKV_1$ and $BingKV_2$. These two real-world traces exhibit following characteristics – the average size of top 500 frequently queried KV objects can have a difference up to 9.49% from one week to another, and that of $Prod_Trace_1$ is approximately two-times larger than $Prod_Trace_2$.

For more controlled experiments, we also synthesized three workload traces containing $\sim$0.5 million query requests. Our workload generation tool is based on the Yahoo! Cloud Serving Benchmark (YCSB) [8]. However, while YCSB considers the key-value frequency distribution, it does not consider the key-value size distribution. To fill this gap, given a list of unique keys from YCSB (sorted in descending order of frequency), we assign key-value sizes according to some distributions. For $Synth_Trace_1$, $Synth_Trace_2$ and $Synth_Trace_3$, we fix their size distribution to be Zipf, and we generate keys following the frequency distribution of Zipf, normal, and linear. Doing so allows us to examine different size distributions.

Comparison baselines. In addition to random search,

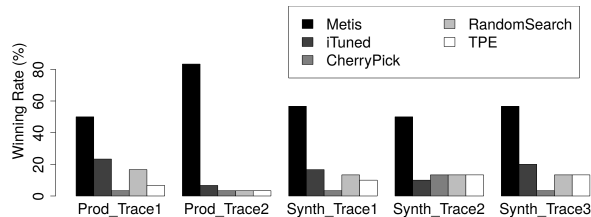


Figure 4: Chance that each approach picks the best configuration. Each experiment allows each approach to run 25 trials, and we repeat the experiment 30 times.

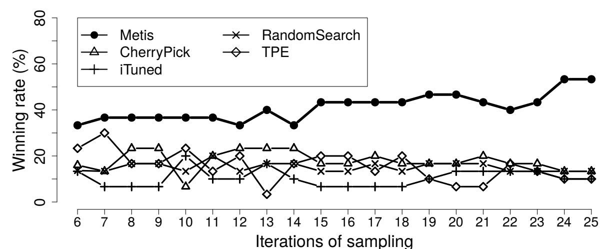


Figure 5: Chance that each approach picks the best configuration for $Prod_Trace_1$, while changing the number of sampling points allowed.

we take state-of-the-art BO-driven approaches including iTuned [10], CherryPick [3], and TPE [5]. We configure these comparison baselines with the recommended setting, e.g., the Matern(5/2) kernel for CherryPick. At each iteration, these approaches output the configuration they expect to have lowest tail latency. Table 1 lists their differences. For the ground truth, we brute-force all possible configurations for $Prod_Trace_1$.

6.2 Effectiveness of Metis

This section quantifies the likelihood that the system configuration selected by Metis outperforms those of other approaches. This relates to the auto-tuning correctness. We also discuss factors including time budgets (i.e., number of trials allowed) and data outliers.

Metis has a higher chance in picking system configurations that outperforms those of other approaches. Assuming a fixed time budget, we allow each approach to run 25 trials. We then rank approaches by the 99-percentile latency of their expectedly best-performing system configurations. We clean-install the machine to ensure minimal noise in measurements, and evaluate the case of data outliers later in this section. Experiments are repeated 30 times, to compensate for the randomness in some approaches. For random search, we take the best of all 25 trials for evaluation. We note that TPE requires at least 20 bootstrapping trials (from random sampling), and we allocate five bootstrapping trials to CherryPick, iTuned and Metis.

Figure 4 summarizes results when system benchmarks, or training data, are relatively noise-free. It shows that Metis has a higher chance of picking the winning configuration, i.e., the system configuration that outperforms those selected by comparison baselines. Depending on the synthesized workload trace, Metis can outperform 48.97% (for $Synth_Trace_2$) to 84.67% (for $Prod_Trace_2$) of times.

Furthermore, taking a closer look into cases where Metis failed to win, we note that Metis 99-percentile latency is within 1% of the winner. In addition, Metis observes the lowest difference of 99-percentile latency to the optimal, on average. This difference is 0.44% for Metis, which is 67.16% lower than CherryPick.

Metis has a faster convergence time than other approaches. Regardless of the parameter tuning approach, the effectiveness in finding the optimal configuration should ideally increase with the number of data points already sampled, or configurations benchmarked. One natural question is how Metis converges to the optimal system configuration, as compared to other approaches. To this end, we try to answer two questions: ***(1)*** at each iteration, what is the likelihood that Metis selects the system configuration that outperforms those of other approaches? ***(2)*** how fast does Metis converge to the optimal system configuration?

Figure 5 shows that the chance for Metis to pick the winning configurations is high at all iteration. We repeat this experiment 30 times, due to the randomness in some approaches. This result is desirable for auto-tuning under limited time budget, as system benchmarks can consume a long time. In addition, the figure shows that, as the number of sampled data points increase, Metis is able to better model the configuration-vs-performance space, which increases the likelihood of picking the winning configuration. Furthermore, looking at configurations that CherryPick and iTuned pick, we observe that they lean towards exploitation. On the other hand, Metis independently evaluates the potential information gain from exploitation and exploration.

Following Figure 5, Figure 6 illustrates how each approach converges to the global optimum of the configuration-vs-performance space. The figure calculates the average 99-percentile latency of selected configurations from 30 repeated experiments. Compared to other approaches, Metis has a faster convergence time, as it is able to benchmark system configurations that would iteratively improve modeling the configuration-vs-performance space. We note that the effectiveness of TPE significantly improves after 20 iterations, as its Gaussian Mixture Models require many training data. Fi-

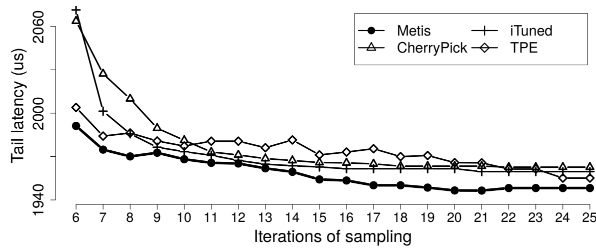


Figure 6: The search path visualizes the tail latency of system configurations that BO-driven approaches select at each iteration.

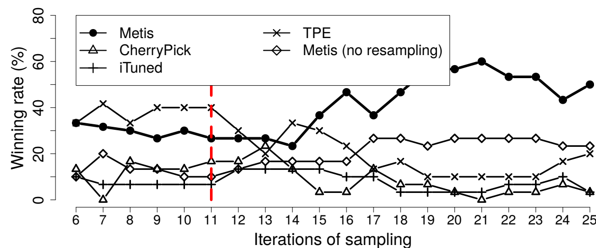


Figure 7: Chance that each approach picks the winning configuration for $Prod_Trace_1$. The test machine has intermittent background activities that result in data outliers. The red line marks when Metis has sufficient data points for re-sampling.

nally, we also note that Figure 6 suggests that iTuned and CherryPick have a higher chance of exploiting local optimum, and thus their acquisition function would require more iterations to find the global optimum.

Metis maintains high chance of picking the winning configuration in the presence of data outliers. Building on the discussion so far, we now demonstrate the robustness of Metis to data outliers. We repeat the previous experiment where we allow each approach to iteratively select 25 trials to benchmark. The machine is a production Bing Ads server, which runs intermittent background activities for software updates, monitoring, and periodic database updates. Resulting noises in the measurement can not reasonably modeled by normal distribution. Unlike comparison baselines, Metis proactively looks for potential data outliers after a sufficient amount of data points is collected, or 10 in our case.

Figure 7 illustrates that Metis has a higher chance of picking the optimal configuration in the presence of data outliers. The red line shows when Metis stops marking all sampled data points as potential data outliers (due to insufficient data collected). The effectiveness of Metis improves after the red line. The figure also shows the effectiveness of Metis without re-sampling, and removing outliers improves Metis's winning rate by 33.33%.

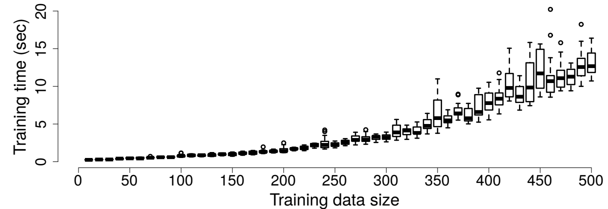


Figure 8: Impact of training data size on the training time of GP models.

6.3 Overhead of Metis

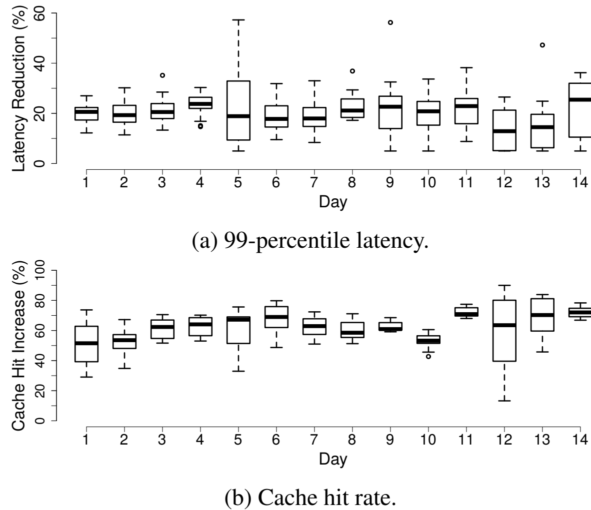
It is known that benchmarking configurations can be time-consuming due to system warm-up and stabilization. Another potential source of overhead is the predictive regression model – both in training and configuration selection. To quantify this overhead, we randomly generate the training data for individual experiment runs, and repeat each experiment 50 times.

Model training time. This is the time for Metis to fit a Gaussian process model over all sampled data points. We note that the community has shown that training the Gaussian process model can exhibit a complexity of $O(N^3 + N^2D)$ [21], where N and D are the number and the dimensionality of training data points, respectively. This suggests that the number of training data points largely determines the training time. Our goal is to understand whether the training complexity will limit Metis's practicality in the real-world. Figure 8 shows that the training time increases with the number of sampled data points (for a training data set with a dimension of 20, and parameter values range between 1 and 1,000). When the number of sampled data points increases to 500, the training time reaches 12.33 seconds. However, this training time is still practical in real-world deployments.

Configuration selection time. This is the time for Metis to predict a sampling point that is expected to maximize the given acquisition function. It has been shown that the time complexity for prediction with the Gaussian process model is $O(N^2 + ND)$ [21], where N and D are the number and the dimensionality of training data points, respectively. When the training data size increases to 500, predicting the observation of an unsampled point takes $\sim$10.25 msec. This time suggests that the predictive model is feasible for real-world usages.

7 Case Study: BingKV

With a year of Metis operational experience, there are observations and lessons learned from adopting autotuning in Microsoft services. This section presents measurements and experiences from our running ex-



(a) 99-percentile latency.

(b) Cache hit rate.

Figure 9: Performance comparison between the previous LRU-based data store and the Metis-tuned BingKV, based on 14-day hourly performance logs.

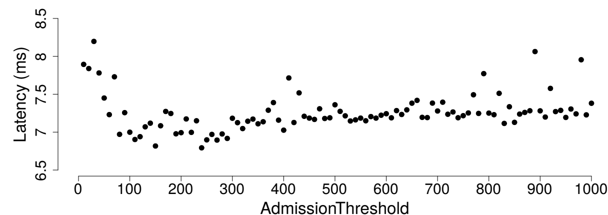
ample, Bing Ads KV store clusters (c.f. §2.1). The KV store evolved from LRU, expert-tuned BingKV, to Metis-tuned BingKV. We study measurements from two KV store clusters – $BingKV_1$ (∼700 servers handling ∼4.14 billion key queries per day) and $BingKV_2$ (∼1,700 servers handling ∼3.18 billion key queries per day). Each server of clusters has an in-memory cache capacity of 512 MB, and an SSD as the persistent data store.

Metis-tuned BingKV reduces the 99-percentile query lookup latency by an average of 20.4%, as compared to LRU. We start by comparing Metis-based KV store with the previously LRU-based KV store, under Bing Ads $BingKV_1$ production workload. The comparison is based on 14-day hourly logs of the 99-percentile query lookup latency and the cache hit rate (CHR). Figure 9 presents results for $BingKV_1$ – Figure 9a shows that Metis helps to reduce the 99-percentile latency by 20.4% on average (with a standard deviation of 8.4), and Figure 9b shows a CHR improvement of 60.6% (with a standard deviation of 11.7). This translates to a 22.76% reduction in disk I/O reads.

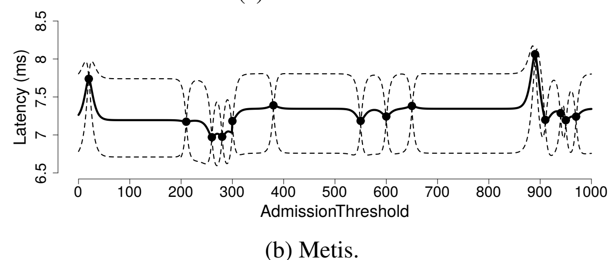
Metis-tuned BingKV reports a 3.43% lower 99-percentile latency, as compared to expert-tuned BingKV. Our human expert is one of the lead programmers for Bing Ads key-value store, with years of experience operating the system. As it is infeasible for the human expert to continuously tune the KV store, this subsection focuses on picking a single best-performing configuration. We note that the human expert did not have any time budget limitations, and manual tuning took about four weeks in a testing environment. The next subsection delves into the system tuning time comparison.

Perf metrics	**Differences**	
	$BingKV_1$	$BingKV_2$
99-percentile latency	-2.99%	-3.43%
Cache hit rate	2.43%	0.49%

Table 2: Performance comparison of expert-tuned and Metis-tuned BingKV-based data store, under 14-day Bing Ads workloads. Metis-tuned configurations outperform expert-tuned configurations, while reducing the tuning time from weeks to hours.



(a) Brute-force.



(b) Metis.

Figure 10: On average, Metis reduces the configuration tuning time by 98.41% for BingKV. For illustration, this figure shows a one-dimensional case of sampling points selected by Metis, as compared to brute-force.

Table 2 shows that Metis-tuning outperforms human-tuning under two weeks of production traffic: ***(1)*** the Metis-tuned configuration reports a 2.99% and 3.43% lower 99-percentile latency for $BingKV_1$ and $BingKV_2$, respectively. ***(2)*** Metis-tuned configuration reports a 2.43% and 0.49% higher CHR for $BingKV_1$ and $BingKV_2$, respectively.

Metis reduces the overall tuning time by 98.41%, as compared to manual tuning by human experts. While humans are typically guided by intuition based on their knowledge of the system, much of the manual tuning process mostly resembles the random search. In fact, the problem exacerbates as the dimensionality of the parameter space increases, especially in the case where parameters have dependencies. For reference, in one scenario deployment, manual-tuning took about 3 weeks, while Metis-tuning took about 8 hours (including sampling, modeling and prediction).

To illustrate how predictive modeling reduces the tuning time, Figure 10 shows that Gaussian process regres-

sion for a one-dimensional case. GP regression is able to estimate the impact of `ShadowAdmissionSize` without sampling the entire parameter space. We note that dotted lines in Figure 10b mark the 95% confidence interval for each point, which can guide both the space exploration and the stopping criteria. Finally, we note that GP regression operations are negligible as compared to system benchmarking – training typically takes ∼1.02 sec, and predicting the expected value of an unsampled data points takes ∼0.53 msec.

8 Related Work

Black-box parameter tuning services. Google Vizier [11] is an ongoing research project, and it supplies core capabilities to optimize hyper-parameters of machine learning models across Google. Similar to Google Vizier, SigOpt [18] is a startup that tunes hyper-parameters of ML models, but little technical details have been disclosed. Google Slicer [1] is a sharding service that dynamically generates the optimal resource assignment, but its goals are not parameter space exploration and sampling. Like Metis, BestConfig [23] uses stratified sampling. However, it heavily leans towards exploitation, as it assumes a high probability of finding better-performing configurations around the currently best-performing one.

In contrast, Metis focuses on providing a robust auto-tuning algorithm for cloud systems, and addresses challenges that systems introduce to the optimization problem. Metis has been used to tune parameters of Microsoft services and networking infrastructure.

Parameter tuning with Bayesian optimization. Metis builds on previous efforts that demonstrate the potential of applying Bayesian optimization and Gaussian process to auto-tuning. iTuned [10] uses Latin Hypercube Sampling (LHS) to sample the parameter space, and model the parameter space with Gaussian process models. OtterTune [2] uses a combination of supervised and unsupervised machine learning methods to reduce the parameter dimension, characterize observed workloads, and recommend configurations. CherryPick [3] follows a similar approach to BO and GP in selecting the best-performing cloud configuration for a given machine learning workload.

TPE [5] uses Bayesian optimization with Gaussian mixture model, instead of Gaussian process. It is suitable for cases where there are some dependencies among parameters. However, since TPE trains with a subset of the training data, it needs a larger amount of data to be collected for training effectively. Smart Hill-Climbing [22] improves Latin Hypercube Sampling, but the GP-based approach has been shown to outperform [10].

Metis introduces customizations to the framework of BO with GP. These include the diagnostic model to find potential data outliers for re-sampling, and a mixture of acquisition functions to balance exploitation, exploration and re-sampling.

9 Discussion

We now discuss limitations concerning the applicability of Metis to systems in general.

Support of different system parameter types. Some types of system parameters can be non-trivial to model with regression models – First, categorical parameters take on one of a fixed number of non-integer values such as boolean. Since categorical parameters are not continuous in nature, it can be difficult to model the relationship among possible values. While categorical parameters are out of scope for this paper, our current implementation conceptually treats each categorical value as a new target system. Second, some systems have multi-step parameters, where one single configuration requires the system to go through a specific sequence of value changes for one or more parameters. Metis does not currently support multi-step parameters.

Costs of changing system configurations. Applying configuration changes can incur costs for some systems. First, server reboots might be necessary after a configuration change, thus service interruptions. To handle this case, system administrators can decide to push a configuration change only if the new configuration is predicted to offer a certain level of performance improvement (e.g., 10% latency reduction). Administrators can also bound the cost of reconfiguration, e.g., by performing reconfigurations gradually over time, or by bounding the parameter space exploration by the distance from the current running configuration. Second, mis-predictions can result in system performance degradation. Fortunately, Gaussian Process offers two ways to gain insights regarding uncertainties – GP offers a confidence interval for each prediction, and a log-marginal likelihood score [15] to quantify the model fitness with respect to the training dataset.

10 Conclusion

This paper reports the design, implementation, and deployment of Metis, an auto-tuning service used by several Microsoft services for robust system tuning. We demonstrate the effectiveness of Metis with controlled experiments and real-world system workloads. Furthermore, real-world deployments show that Metis can significantly reduce the system tuning time.

References

[1] ADYA, A., MYERS, D., HOWELL, J., ELSON, J., MEEK, C., KHEMANI, V., FULGER, S., GU, P., BHUVANAGIRI, L., HUNTER, J., PEON, R., KAI, L., SHRAER, A., MERCHANT, A., AND LEV-ARI, K. Slicer: Auto-Sharding for Datacenter Applications. In *OSDI* (2016), USENIX.

[2] AKEN, D. V., PAVLO, A., GORDON, G. J., AND ZHANG, B. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD* (2017), ACM.

[3] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *NSDI* (2017), USENIX.

[4] AZIMI, J. Bayesian Optimization (BO).

[5] BERGSTRA, J., BARDENET, R., BENGIO, Y., AND KEGL, B. Algorithms for Hyper-Parameter Optimization. In *NIPS* (2011).

[6] CAFLISCH, R. E. Monte Carlo and Quasi-Monte Carlo Methods. In *Acta Numerica* (1988).

[7] CHU, L., CURSI, E. S. D., HAMI, A. E., AND EID, M. Reliability Based Optimization with Metaheuristic Algorithms and Latin Hypercube Sampling Based Surrogate Models. In *Applied and Computational Mathematics* (2015).

[8] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *SoCC* (2010), ACM.

[9] DEAN, J., AND BARROSO, L. A. The Tail at Scale. In *Communications of the ACM* (2013), ACM.

[10] DUAN, S., THUMMALA, V., AND BABU, S. Tuning Database Configuration Parameters with iTuned. In *VLDB* (2009), ACM.

[11] GOLOVIN, D., SOLNIK, B., MOITRA, S., KOCHANSKI, G., KARRO, J., AND SCULLEY, D. Google Vizier: A Service for Black-Box Optimization. In *KDD* (2017), ACM.

[12] MCKAY, M. D., BECKMAN, R. J., AND CONOVER, W. J. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. In *American Statistical Association and American Society for Quality* (2000).

[13] RASMUSSEN, C. E., AND WILLIAMS, C. K. I. *Gaussian Processes for Machine Learning*. the MIT Press.

[14] RYZHOV, I. O. On the Covergence Rates of Expected Improvement Methods. In *Operations Research* (2014).

[15] SCHIRRU, A., PAMPURI, S., NICOLAO, G. D., AND MCLOONE, S. Efficient Marginal Likelihood Computation for Gaussian Process Regression. In *arXiv:1110.6546* (2011).

[16] SCIKIT-LEARN. GaussianProcessRegressor. `http://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessRegressor.html`.

[17] SHAHRIARI, B., SWERSKY, K., WANG, Z., ADAMS, R. P., AND DE FREITAS, N. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE* (2016).

[18] SIGOPT. SigOpt. `http://sigopt.com/`.

[19] SNOEK, J., LAROCHELLE, H., AND ADAMS, R. P. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS* (2012).

[20] VENKATARAMAN, S., YANG, Z., FRANKLIN, M., RECHT, B., AND STOICA, I. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytic. In *NSDI* (2016), USENIX.

[21] WILSON, A. G., HU, Z., SALAKHUTDINOV, R., AND XING, E. P. Deep Kernel Learning. In *AISTATS* (2016).

[22] XI, B., LIU, Z., RAGHAVACHARI, M., XIA, C. H., AND ZHANG, L. A Smart Hill-Climbing Algorithm for Application Server Configuration. In *WWW* (2004).

[23] ZHU, Y., LIU, J., GUO, M., BAO, Y., MA, W., LIU, Z., SONG, K., AND YANG, Y. BestConfig: Tapping the Performance Potential of Systems via Automatic Configuration Tuning. In *SoCC* (2017), ACM.