TUNA: Tuning Unstable and Noisy Cloud Applications

Johannes Freischuetz
freischuetz@cs.wisc.edu
University of Wisconsin – Madison
Madison, Wisconsin, USA

Konstantinos Kanellis
kkanellis@cs.wisc.edu
University of Wisconsin – Madison
Madison, Wisconsin, USA

Brian Kroth
bpkroth@microsoft.com
Microsoft Gray Systems Lab
Madison, Wisconsin, USA

Shivaram Venkataraman
shivaram@cs.wisc.edu
University of Wisconsin – Madison
Madison, Wisconsin, USA

Abstract

Autotuning plays a pivotal role in optimizing the performance of systems, particularly in large-scale cloud deployments. One of the main challenges in performing autotuning in the cloud arises from performance variability. We first investigate the extent to which noise slows autotuning and find that as little as 5% noise can lead to a 2.5x slowdown in converging to the best-performing configuration. We measure the magnitude of noise in cloud computing settings and find that while some components (CPU, disk) have almost no performance variability, there are still sources of significant variability (caches, memory). Furthermore, variability leads to autotuning finding *unstable* configurations. As many as 63.3% of the configurations selected as "best" during tuning can have their performance degrade by 30% or more when deployed. Using this as motivation, we propose a novel approach to improve the efficiency of autotuning systems by (a) detecting and removing outlier configurations and (b) using ML-based approaches to provide a more stable *true* signal of de-noised experiment results to the optimizer. The resulting system, TUNA (Tuning Unstable and Noisy Cloud Applications) enables faster convergence and robust configurations. Tuning PostgreSQL running `mssales`, an enterprise production workload, we find that TUNA can lead to 1.88x lower running time on average with 2.58x lower standard deviation compared to traditional sampling methodologies.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Computing methodologies → Machine learning**.

Keywords: Bayesian Optimization, Autotuning, Performance Tuning, Noise Reduction, Performance Variability

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3717480

ACM Reference Format:
Johannes Freischuetz, Konstantinos Kanellis, Brian Kroth, and Shivaram Venkataraman. 2025. TUNA: Tuning Unstable and Noisy Cloud Applications. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3689031.3717480

1 Introduction

Software tuning is important for many different types of systems. Before the rise in popularity of machine learning (ML), heuristics and search-based approaches for tuning were used in different areas including databases [3, 22, 77], key-value stores [15], VM sizing [95], compilers [83], distributed systems [40], and memory systems[41]. With the rise in popularity of applying ML to systems problems, several recent works have developed autotuning frameworks for databases (e.g., OtterTune [88], CDBTune [99], QTune [57]), file systems [12] and analytics frameworks [94].

Most state-of-the-art tuners follow a similar design for autotuning as shown in Figure 1. The autotuning process consists of a short loop: first, an *optimizer* (e.g., Bayesian or Gaussian Process optimizer etc.) suggests a new configuration (config) to evaluate for a System-under-Test (SuT) running a specific workload. Typically, an initialization set, a set of randomly selected configurations, is used to bootstrap the model. After the initialization set, a metric such as Expected Improvement or Thompson sampling [82] is used to generate the new suggestions [36]. Each suggested config is evaluated (sometimes called "profiling") for a fixed period to measure its performance. The results are then cataloged, along with all prior evaluations, and then finally returned to the optimizer to improve future suggestions. Once a given stopping criteria is met (e.g., total tuning time or number of configs), the best config is selected from the catalog.

Although ML-based solutions have been shown to achieve significant *peak* performance improvements (e.g., 2-5x [88] performance improvement in database systems, 20-25% [12] performance improvement in storage systems, and 15-20% [61] improvement in cluster scheduling), existing work has not studied if tuning produces configs with stable and predictable performance in the cloud (i.e., performance remains

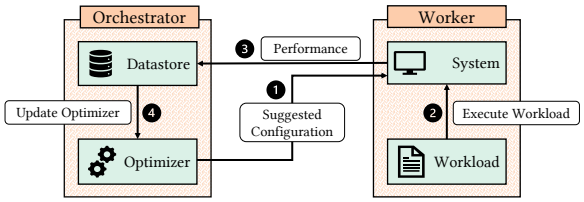


Figure 1. Modern auto-tuning system design [42, 98, 99]

the same when deployed on similar hardware). This conflicts with service level agreement (SLA) [52] requirements, which demand stable performance on customer workloads.

We find that there are two main challenges in finding stable, well-performing configs: first from platform level *performance variability* during tuning, and second from *unstable behavior* of the SuT. Given that various levels of the hardware and software stack have been shown to have variable performance [10, 24, 44, 62, 79, 80], when a config suggested by the optimizer is evaluated, the measured performance of the SuT can be noisy. We find that even having a small amount of measurement noise (5%) can lead to a significant slowdown (2.5×) in finding good performing configs (§3). Additionally, we find that the software systems can exhibit unstable behavior with specific configurations; typically this can be attributed to configurations changing which code paths are taken at runtime, leading to unexpected performance. We first expand on these findings by discussing our benchmark of cloud computing offerings to measure the amount of variability caused by hardware components, virtualization, and noisy neighbors and follow that with a case study on how unstable behavior can affect autotuning.

To measure variability in the cloud, we run a 68 week-long study, sampling across over 43500 VMs on Microsoft Azure. While there have been prior cloud measurement studies (listed in Table 1), these studies are either too old [25, 37, 75] and do not capture recent changes to the cloud, or do not sample a large set of VMs to understand the distribution of performance across a region [19, 25, 54, 62, 75, 76, 87]. To the best of our knowledge, this is the longest, and largest cloud study conducted and we find, contrary to prior work, very consistent CPU and disk performance. The variability of these components using the Coefficient of Variation (CoV) (the standard deviation normalized by the mean), is less than 1.0%. This is significantly lower than the results reported for bare-metal machines (9.0% for disk [62] and 12.8% for CPU [4]). We still find that some components have significant variance such as memory, CPU cache, and OS-related operations (4.92%, 8.82%, 14.39% coefficient of variation, respectively). Given that significant variability still exists in the cloud, this motivates the question: *How does platform performance variability affect ML-based tuning of software systems that use these components?*

To explore this, we run a case study tuning PostgreSQL [81] on a well-known workload, `TPC-C` [56], in the cloud. We use `SMAC` [36], a popular Bayesian Optimizer. We chose PostgreSQL as our SuT, as, in addition to disk and CPU, PostgreSQL also heavily uses cache, memory, and the OS - the components which we found have high variance in the cloud. Surprisingly, we find that 39.0% *of the configs seen during tuning are "unstable"*; i.e., TPC-C throughput varies substantially, with up to 101.3% CoV. Worse, many of the configs that performed best during tuning, when deployed to a new VM, experienced up to 76.1% performance degradation compared with that during tuning. However, not all best-seen configs experienced such degradation, and there exist configs that are stable and yield high performance.

Motivated by the above findings, we develop TUNA, a new sampling methodology that aims to find stable, but still high-performing, configs. TUNA primarily changes how we measure SuT performance in the autotuning design, allowing TUNA to directly integrate with existing optimizers (e.g. `SMAC` [36], `botorch` [11], etc.) and generalize to any SuT. We use two main insights in TUNA: (a) the only way to quickly detect unstable configs is by sampling across a representative cluster to capture the variance across nodes, and (b) component-level metrics can mitigate noise from samples.

TUNA integrates these insights through three main components. First, we use *multi-fidelity sampling* to evaluate configs at various "budgets", with higher budgets corresponding to sampling from more nodes. Secondly, we build an outlier detector and an aggregation policy using these additional collected samples that, together, filter out unstable configs. Finally, we design a *Noise Adjuster* model that aims to remove the influence that platform variability has on the optimizer using component-level metrics (e.g., CPU, Memory stats etc.). Together these techniques help autotuning frameworks achieve fast cost-effective convergence to a stable config that is both performant and robust.

We evaluate TUNA under various scenarios to show that it can generalize well to different scenarios. These include three SuT (PostgreSQL , Redis , and NGINX), six workloads (`TPC-C`, `epinions`, `TPC-H`, `YCSB-C`, `mssales` [84], an internal Microsoft production workload, and a Wikipedia serving workload), two cloud regions, two distinct VM SKUs, and two optimizers (Bayesian Optimization, Gaussian Processes). We compare TUNA against the existing state-of-the-art approach to tuning with a single machine. Every system and workload evaluated improved when using TUNA, either in terms of better-achieved performance, reduced variability, or both. For example, `mssales` [84], tuned using TUNA has 46.8% lower latency and 61.2% lower standard deviation than a system tuned with traditional sampling techniques.

Open Source. To benefit the community we release the cloud measurement *dataset* [1] and *source code* [2] for TUNA.

[1] https://aka.ms/mlos/tuna-eurosys-dataset
[2] https://aka.ms/mlos/tuna-eurosys-artifacts

2 Background

Performance Variability. Systems performance variability comes in a variety of forms (e.g., inherent software non-determinism, subtle hardware differences, interference from other "noisy neighbor" workloads on shared infrastructure such as in cloud environments, etc.) and has been studied for many decades in many applications of system design under different names (variability, interference, noise, cloud weather, etc.) [10, 62, 79, 80, 104]. Recent work has studied hardware variability that can lead to gray failures [32, 35, 96]. Many types of systems attempt to be agnostic to absolute system performance or have solutions that do not need config tuning and evaluation. For example, systems like MapReduce schedule duplicate work on additional machines to avoid stragglers [20]. Other systems [16, 18, 69] attempt to more directly address interference caused by sharing infrastructure. Similar concepts have also been deployed in the hypervisor [7] and in the HPC community [19, 34].

In the context of software testing, there has been work to detect outliers caused by performance regression in the cloud rather than the system design. Some works are as simple as taking more samples across machines [50], or better test orderings [1]. Other works have integrated metrics into outlier detection [13, 92], however, there is no work we are aware of that goes beyond using metrics for detection.

We argue that while further systems advancements in isolation, detection, and scheduling are useful and important, interference will continue to be an issue for systems developers and operators, particularly those working with shared infrastructure, and hence any system that intends to run in the cloud needs to be capable of addressing increased noise.

ML-based Tuning. There have been many works that perform ML-based tuning of data systems [42, 51, 57, 85, 98, 99]. These approaches can be either online or offline, and both are sensitive to noise in the environment.

Online approaches use an agent-based approach to change a smaller set of parameters that can be tuned without interrupting the system (e.g., restart). The policies employed are typically more conservative in order to avoid incorrect configs which may impact the liveness of the system and as such require more tooling and are more challenging to explain and support [102]. Moreover, since workloads are not replayable, the learning process is more complicated.

Offline tuning, in contrast, allows for a greater degree of exploration. Given a workload, the configuration space is searched out-of-band on a set of test machines and later periodically applied to the production systems as workloads change. Since these events can be scheduled and monitored, much like a normal user config change, they are easier to deploy and reason about, and are generally the preferred initial approach by both support teams and customers. For this reason, we primarily focus on offline tuning in this paper, but many of our lessons apply to both contexts.

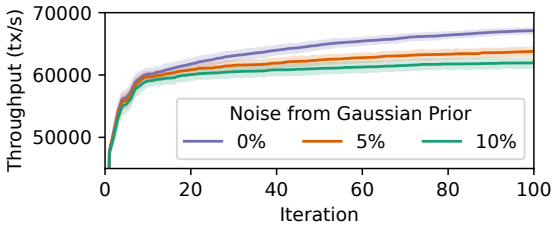


Figure 2. Optimizer Rate of Convergence for epinions workload, running on PostgreSQL 16.1 at various levels of noise.

In either case, the two primary performance considerations for an autotuner are (a) its ability to *discover* an optimal config, and (b) its rate of *convergence*. Since the global optimal is typically not known unless one performs an exhaustive (and expensive) grid search, ML-based techniques are used to improve the search rate towards the "best found" config.

Many of the most successful systems use some variant of Bayesian Optimization (BO) [42, 88, 98] for their optimizer, as it can often find the optimum in fewer evaluations. Reinforcement learning (RL) [57, 99] has also been used for this task, yet RL often requires an initial training phase which can be time-consuming. CDBTune focuses on parallelizing training in a cloud environment, however, it does not address the noise that is inherent in the cloud [99]. We argue that for these systems to be reliable in production, the tuning system needs to account for the variance across machines, changes in collocated VMs, and the noise of the cloud environment.

Prior studies have proposed ML algorithms that are resilient to noisy data [8, 28, 55]. There are theoretical studies that characterize, and devise ways to handle constant Gaussian noise in BO [28, 55], which make strong assumptions about the shape of the noise and do not apply to the non-parametric noise that we observe in the cloud. Additionally, such methodology has never been applied to tuning systems.

3 Motivation

Existing approaches for tuning systems have assumed that tuning environments represent eventual deployment environments. However, it has been reported that even on bare-metal (i.e., isolated) nodes, performance variability can be as high as 29.2% CoV for network, and 16.0% CoV for memory [10, 62, 79]. The situation can be even worse in cloud environments, where performance variability due to shared infrastructure has been reported to be even higher [2, 24, 68]. In this section, we investigate how performance variability affects ML-based tuners regarding convergence rates and final configs. We use a series of experiments understand: (i) the impact of performance variability (noise) on tuner convergence, (ii) the magnitude of noise in the cloud, and (iii) the impacts of cloud noise on best-performing configs.

3.1 Impact of Interference on Tuner Convergence

We start by investigating the impact of noise on tuner convergence. We employ PostgreSQL v16.1 as an example SuT

we wish to optimize, and we run the `epinions` [70] workload. More details about the workload are given in §6.1. We use `SMAC` [58] as our optimizer, as done in prior state-of-the-art work [42], and maximize throughput as our objective. Here, we employ isolated `c220g5` bare-metal nodes from CloudLab [23], which do not experience any noise from virtualization or noisy neighbors. To understand the impact of noise on convergence, we synthetically simulate various levels of noise by reporting noise adjusted using a Gaussian distribution prior. In particular, if P represents the measured performance, σ represents our chosen CoV, then $P^* = P \times \Delta$ is the value we report back to the tuner, where $\Delta \sim \mathcal{N}(1, \sigma^2)$. We explore three such noise levels (i.e., sampling noise): 0%, 5%, and 10% We execute 100 tuning runs, each with a different initialization set, for each level of noise; for each run, we evaluate 100 samples (i.e., configs).

Figure 2 depicts the performance of the best config found so far by the tuner at a given iteration. The solid line is the mean performance, and the shaded region is the 99% confidence interval. We observe that the tuner convergence rate rapidly degrades when the noise in the prior is increased from 0% to 5%. The tuner using a noise level of 0% converges to the same noise-free performance in 40 iterations as the 5% noise case in 100 iterations. The slow-down ratio of these numbers is called *time-to-optimal* performance [42], and we find it to be 2.50x. Increasing noise to 10% further hinders convergence, and gives a *time-to-optimal* ratio of 4.35x as compared to no noise. Thus, given that a small amount of *synthetic* sampling noise can have such a significant impact on the tuner convergence, and hence its cost, we must investigate the magnitude of noise in the cloud to determine how much impact it will have on tuning systems.

♦ Increased sampling noise during tuning significantly slows down convergence, increasing tuning cost.

3.2 Quantifying Cloud Noise

Performance variability in the cloud has been studied in multiple prior works (shown in Table 1); however, we find that these prior studies are limited in scale and duration, or outdated, and do not report results on OS-related variability. We also note that while some studies (e.g., Figiela et al. [27]) do sample many instances, they use serverless nodes, which have less consistent performance properties than VMs [59], as they have weaker isolation.

We thus conduct a study to investigate the magnitude of variability in today's cloud caused explicitly by platform and hardware variability. We ran a longitudinal study for 480 days on Microsoft Azure, starting May 28th, 2023, and ending on September 19th, 2024. We conducted our study using 40 microbenchmarks, and 13 end-to-end application benchmarks, resulting in a total of 92 metrics. We investigate `Standard_D8_v5` [39] VMs general-purpose VM in

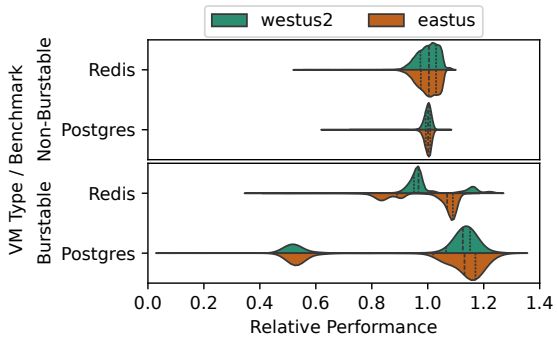


Figure 3. The variance of PostgreSQL and Redis benchmarks. Relative performance is relative to the mean performance seen within a region and VM SKU type. Higher is better.

two regions: `westus2`, and `eastus` and across two types of lifespans: short-running, and long-running VMs. Additionally, we briefly discuss `Standard_B8ms` [90] VMs in the same regions and with the same lifespans. We isolated our benchmarks to use `SSDv2` [72] disks in Azure where applicable. Long-running VMs ran for the entire duration of the study (68 weeks). Short-running VMs were deployed, ran the benchmarks, recorded the results, and were immediately deprovisioned. This allows us to sample across many physical nodes in each region, collecting different types of cloud weather as compared to long-running VMs, which we find seldom migrate. There were three long-running VMs of each type in each region, and a total of *over* $43k$ short-lived VMs approximately evenly distributed between the two regions and types of VM. During the study, we collected *over* $7M$ data points approximately evenly distributed between the two regions, the two VM lifespans, and the two VM types.

First, we compare the end-to-end performance of long-lived burstable VMs to non-burstable VMs running two full systems, PostgreSQL and Redis , in Figure 3. For PostgreSQL, we run a read-write workload from `pgbench` [33] with a dataset significantly larger than memory. For Redis , we run a write-heavy workload from `redis-benchmark` [74].

Immediately, it is clear that there is not only significantly higher variability in burstable VMs but also a bimodal distribution. There are two key differences between these two types of VMs. First, burstable VMs are oversubscribed with weaker isolation as compared to non-burstable VMs. This causes a wider distribution of performance. Secondly, and more problematic for tuning, bursting credit depletion causes extreme performance bimodality. For example, when there are sufficient disk credits for the PostgreSQL workload, we see performance in the upper end of the distribution, and when they are depleted, there is a greater than 50% degradation. This is not only problematic because of the loss of performance, but it also makes it difficult for the optimizer to distinguish between credit depletion and unstable configurations (§ 3.2.1). For this reason, for the remainder of the paper,

Paper	Year	Duration	Samples	Instances	Platform	Disk	Memory	CPU	Network	OS
Schad et al. [75]	2010	4 weeks	6 k	4	A	✓	✓	✓	✓	✗
Iosup et al. [37]	2011	52 weeks	250 k	N/A[a]	A, G	✗	✗	✓	✗	✗
Farley et al. [25]	2012	2 weeks	59k	40	A	✓	✓	✓	✓	✗
Leitner and Cito [54]	2016	4 weeks	54k	82	A, M, G, I	✗	✓	✓	✗	✗
Maricq et al. [62]	2018	46 weeks	900 k	835	CL	✓	✓	✗	✓	✗
Figiela et al. [27]	2018	22 weeks	730 k	13,723[a]	A, M, G, I	✗	✗	✓	✗	✗
Scheuner and Leitner [76]	2018	4 weeks	63 k	244	A	✓	✓	✓	✓	✗
Uta et al. [87]	2020	3 weeks	1000 k	1	A, G, H	✗	✗	✗	✓	✗
De Sensi et al. [19]	2022	N/A	516 k	2	A, M, G, O	✗	✗	✗	✓	✓
This Work	**2024**	**68 weeks**	**7037 k**	**43,641**	M	✓	✓	✓	✗	✓

[a] These studies were conducted on serverless nodes

Table 1. Comparison of our work to prior benchmarking studies. For platform A indicates Amazon AWS, M indicates Microsoft Azure, G indicates Google GCP, I indicates IBM Cloud, O indicates Oracle OCI, CL indicates CloudLab, H indicates HPCCloud.

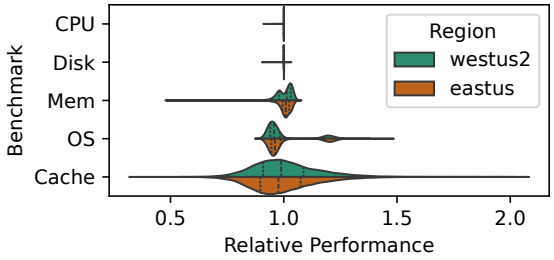


Figure 4. The variance of benchmarks targeting CPU, Disk, Memory, the OS, and CPU cache. Relative performance is relative to the mean performance seen. Higher is better.

we choose to use non-bursting VMs. We leave accounting for burstable VM credits during tuning as future work.

♦ Burstable VMs are not suitable for autotuning applications without accounting for bursting credit depletion.

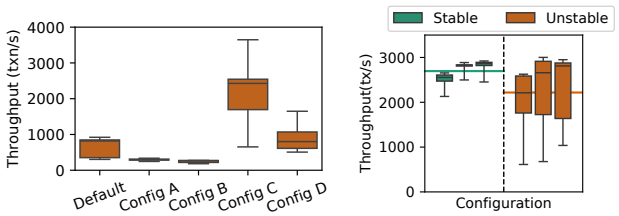
We focus on the results of 5 different resource microbenchmarks running on non-burstable VMs: prime verification using `sysbench` [48] to test CPU performance, random writes using `fio` [9] with the Linux Async IO engine for disk, Max Bandwidth with a 1:1 read/write ratio using Intel's Memory Latency Checker [91] for memory, thread creation time using `OSBench` [29] for the OS, and the `stress-ng` [45] cache micro-benchmark. While these are not the only benchmarks we ran for each of these components, they are each representative of other benchmarks that targeted their respective components. The results of these microbenchmarks are shown in Figure 4, and we present two main takeaways.

First, the performance characteristics of some components (e.g., disks and CPUs) have significantly lower variability than the results reported in prior works (Table 1), with the highest CoV for any disk-isolated benchmark or any CPU-isolated benchmark being 0.36%, and 0.17%, respectively. This decreased variability can likely be attributed to changes in the offerings provided by the platform. Prior studies have shown many different CPUs all within a single VM SKU, including a mix of Intel and AMD CPUs [25]. Modern VM SKUs, on the other hand, have much tighter SLAs and have separated these CPUs into different SKUs [6, 30, 39, 64]. In all of our experiments, we found that every VM reported using the same underlying CPU type. Additionally, there has also been work at the platform level to fairly share and isolate CPU cycles [101]. The virtual disk service also benefits from platform-level improvements. In Azure, we found that the SSDv2 Managed Disk offering [72] has significantly lower variability than storage shown in prior work. Note that Azure is not the only platform to provide these types of virtual disks. GCP and AWS both provide similar offerings[5, 31]

♦ Platform level advancements have made CPU and Disk cloud-based components much more stable than results shown in prior studies.

However, platform-level improvements have not solved all the variability problems that exist in the cloud. Memory, OS, and Cache still have high variability with CoVs of 4.92%, 9.82%, and 14.39% respectively. Some of the variance can be explained by design decisions that cloud providers have made. Memory and cache bandwidth are often not restricted, which allows heavy interference impacts. There has been work to more fairly share these resources at a platform level [65], and a hardware level [89, 97], however, these have either not been deployed or have not completely resolved the issue. OS operations are another problem. Many OS-related operations require a `VMEXIT`, which can incur a heavy overhead due to CPU overbooking and modern side-channel and other security mitigations that systems must employ [67]. While recent proposals have introduced some hardware support [86, 103], we find that they do not fully mitigate variability in virtualized systems. Given that there is variability in cloud-based systems, it raises the question: *how much does this variability impact tuning?*

♦ Memory and Cache still experience high levels of variance, we also see the OS as a previously unmeasured source of variance.



(a) Throughput for the 5 configurations of the initialization set evaluated on the same 30 nodes.

(b) A subset of best-performing configs. when deployed on new nodes.

Figure 5. The performance of initialization and of best-performing configurations (picked by the optimizer) during training (left) and when deployed on new nodes (right).

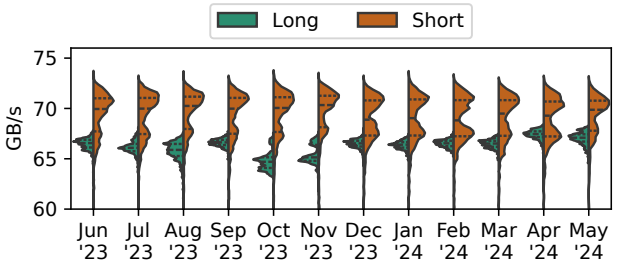


Figure 6. Performance of a memory bandwidth benchmark partitioned by month. The long-running VM is a single representative VM that ran for the entire duration, while the short-running is a set of 10,632 VMs, both in `westus2`.

3.2.1 Unstable Configurations. To study the impacts of tuning the cloud, we ran 30 tuning runs, each with the same initialization set, targeting PostgreSQL 16.1 running `TPC-C` [56] for 50 iterations. We evenly distributed these 30 runs across 30 identically configured `D8s_v5` [39] VMs using SSDv2 Data Disks in the same region. The question we wish to answer with this experiment is: *are modern tuning processes resilient to the performance variance seen in §3.1, and in turn always able to generate a usable configuration?*

First, we notice that the configurations in the initialization set have vastly different stability characteristics. Of the 10 total configurations in the initialization set, we present the 5 configs that do not immediately crash the system in Figure 5a. Config C is particularly concerning as it either performs extremely well or extremely poorly, depending on which machine it was run on, in a difficult-to-predict manner. We term these configurations *Unstable*.

Next, we explore the ability of the best-performing configurations to be deployed on a VM distinct from the one used during tuning. We term the ability to maintain performance as the *transferability* of a config. To explore *transferability*, we deployed the best config found from each optimization run, to a set of 10 new VMs with the same platform setup, and evaluated the performance of the optimized config on the new VM. We continue to find vastly different variability between these "best" configs, with some configs being unstable and others being stable. We can quite easily divide these configs into the two aforementioned categories and show three of each in Figure 5b. We find from this divide that 13 of the 30 transferred configs are unstable. Furthermore, some of these configs, including two of those shown, can have their performance degraded by over 70%, and have CoVs of up to 36.3%. This is significantly higher than what can be attributed to simply noisy neighbor effects. The highest CoV for any benchmark PostgreSQL benchmark in § 3.1 was 7.23%. In §4, we design a heuristic to implement this classification.

We thoroughly investigated system performance metrics, such as storage throughput and CPU performance via micro-benchmarks. These revealed (1) no obvious correlations and (2) that the new machines were performing within the small bounds expected to be caused by platform variance and noisy neighbors. We eventually found the root cause for this performance degradation to be a difference in the query plan selected by the DBMS at run time.

For `TPC-C`, when the query planner generates candidate plans, the top two candidate query plans for the `JOIN` query are predicted to have almost the same execution time. However, in reality, one plan takes two orders of magnitude longer. Small changes in the underlying component performance can tip the balance on which plan is selected. Machines that performed well always selected the high-performing plan, while machines that performed poorly occasionally picked the poor plan due to minor differences in predictions in the cost models. We found that the knobs that impact if a config may be *Unstable* include `enable_bitmapscan`, `enable_hashjoin`, `enable_indexscan`, `enable_nestloop`, but the exact combinations are inconsistent across configs. We also found unstable configs in other systems such as Redis and NGINX , but omit their details given space constraints.

A lack of awareness of *Unstable* configs during performance sampling can lead to *Unstable* configs being promoted during the tuning process. Without mitigation, this can cause the best apparent config to be *Unstable* and can cause significant performance degradation when deployed.

> ♦ The existence of *Unstable* configs and their impact on tuning, motivates the need for methods to *collect* representative samples, and *detect* anomalous configs

4 Design

The substantial impact of noise on tuner performance and the existence of *unstable* configs (§3) motivates a new system that mitigates the observed noise and produces configs that transfer across different nodes. Ideally, such a tuning design (i) should be agnostic to the system that is being tuned, (ii) should not require any changes to the underlying optimizer, and (iii) should avoid hurting optimizer performance (e.g., increased time to convergence or degradeyd config quality).

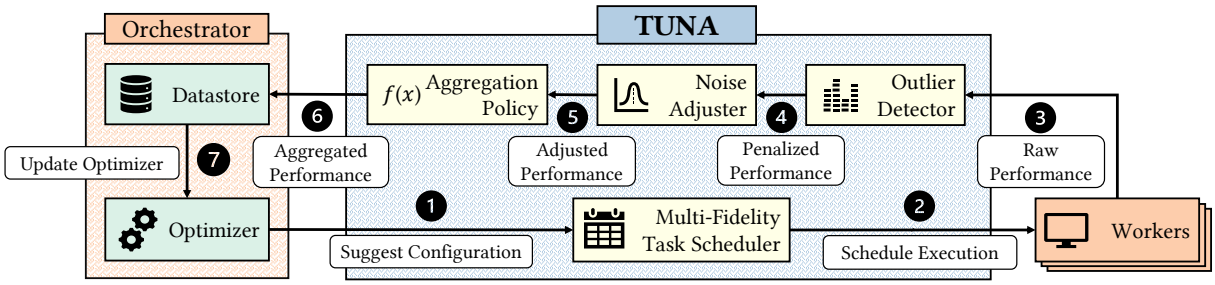


Figure 7. Design for TUNA with our contributions highlighted lime and blue.

To this end, we present our new tuner design, TUNA, which achieves the above goals by introducing four key components: *(i) Multi-fidelity sampling, (ii) Unstable configuration detection, (iii) A model for sample noise mitigation, (iv) sample aggregation.* TUNA refrains from making any changes to the underlying optimizer or SuT, but rather focuses on changing what data is collected by intelligently distributing samples across machines, augmenting results with system metrics, and unstable configuration detection. Similar to prior works [42, 57, 98], TUNA optimizes a chosen target metric for a static workload. A high-level overview of how the above components can be fit into an existing tuning setup can be seen in Figure 7. In the next subsections, we describe these components in more detail.

4.1 Multi-Fidelity Sampling

Performance variability across different cloud nodes necessitates evaluating a config on multiple nodes. Figure 6 depicts the performance of a memory benchmark running on a single long-lived VM for 68 weeks, versus running the same benchmark on many different short-lived VMs over the same period. We observe that while there are performance changes for our long-lived VM over time, the time scale on which these happen is too long and they do *not* capture the performance variance across the broader cluster on which the config could be deployed.

The naive way for the optimizer to gain more confidence is to evaluate each configuration on many nodes, yet doing so can be prohibitively expensive. To this end, we leverage *multi-fidelity* sampling. The key idea of multi-fidelity sampling is to associate a "budget" for each config suggested by the optimizer. Configs initially begin with a limited budget. Promising configs are then promoted, reevaluating them with increased budget, increasing the *confidence* that this config is indeed better-performing, without spending "too much" on poor configs.

Multi-fidelity sampling provides a principled way to parallelize config evaluations across multiple nodes and filter bad-performing configs quickly. We associate the multi-fidelity budget with the number of nodes on which a config is evaluated. We implement this policy using Successive Halving [38], a well-known multi-fidelity policy. This enables us to collect additional samples of individual configs across several machines to evaluate their robustness to noise in the deployment environment. In particular, we start by evaluating a config on a single VM, followed by a small set of VMs (e.g., 3). As long as it keeps performing well, we evaluate it on an increasingly larger set of nodes until eventually, we run it on the entire cluster. Otherwise, we quickly discard the config to give the evaluation budget to more promising ones, which have not yet been run on as many nodes.

Our multi-fidelity tuning methodology comes with two important benefits concerning optimization robustness. First, the resulting best-performing configs are more likely to be transferable to a new deployment VM. This stems from the fact that these configs were already evaluated on multiple nodes, providing us with a more representative performance distribution. Second, given that we now have a distribution of samples, we can use these to detect *unstable* configs. We elaborate on how we do this, in the next subsection.

4.2 Unstable Configuration Detection

We recall (from §3.2.1) that we observed a wide gap in stability between stable and unstable configs. Given the existence of this wide gap, we introduce a simple heuristic, which, given a set of gathered samples (i.e., performance) for a config, decides whether it is stable or not (i.e., outlier).

Our heuristic should make sure that when configs are classified as stable, they are also *transferable*. The classification decision is made using the performance observed when evaluating the configuration on multiple nodes. To this end, the heuristic should only consider whether an outlier performance value exists, i.e., neither the degradation size nor frequency of outliers are important. For example, a configuration that resulted in a single outlier when run in our tuning cluster does not imply being inherently more reliable than a configuration that had two extreme outlier performance numbers; both should be classified as unstable.

One might try to use, as heuristics, the standard deviation or CoV to classify unstable configs. Yet, both options are unsuitable. The former is sensitive to the absolute performance numbers, thus requiring manual tuning for each target system and workload. The latter is inherently biased towards the ratio of outliers to non-outlier measurements, making it hard to reason about the magnitude of performance degradation. To this end, we choose *relative range* as our target

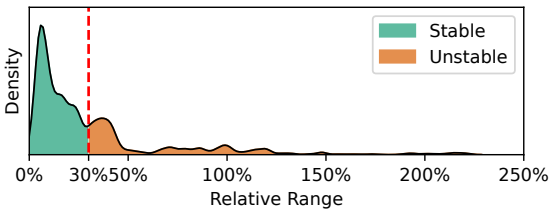


Figure 8. Sensitivity analysis of 1000 configurations seen during tuning. Using a red dashed line, we specify our detection threshold at 30% between the first and second peaks.

heuristic for the outlier detector. This metric does not require tuning and is not biased towards the incidence of outliers.

$$\text{Relative Range} = \frac{max(x) - min(x)}{\mathbb{E}(x)}$$

Using the above heuristic and a fixed threshold of 30%, classify a configuration as stable or not. If we detect an unstable configuration, we inject a penalty score to prevent the optimizer from searching this region again. We implemented this penalty as halving the reported performance as in prior work [88], though other policies are also possible.

We justify our detection threshold by analyzing relative ranges from 1000 configurations seen during tuning, all of which were run on 10 nodes. The results of this experiment can be seen in Figure 8. We pick a simple threshold value of 30%, as this sits in the trough between the first and second peaks of relative ranges. While there is still significant density in the second peak, we are not concerned about a false positive, as we can always find another well-performing stable configuration. However, a false negative can be disastrous when deployed, particularly those with extreme ranges, and thus we opt for a slightly conservative value. This all said the exact choice of this value is not particularly important as long as we safely remove extremely unstable configurations at the far end of the spectrum (right side of the graph), and believe any value between 15% and 30% is reasonable.

4.3 Sampling Noise Modeling

Aside from detecting unstable configurations, we also wish to mitigate the inherent cloud noise from our measurements to improve the convergence rate (§3.1). Specifically, given the set of noisy application performance measurements, we wish to predict the mean *noise-less* performance value to give the optimizer a more stable signal suitable for learning. To do this we make use of guest OS resource usage metrics. Our main observation is that since configs *and* noise change the end-to-end application performance, they also impact the use of system resources in the VM guest OS and can thus be used as an additional signal. To this end, we propose building a predictive model that given an input noisy performance measurement and a set of system metrics can ultimately predict the noise-less performance. This is possible as we evaluate a config on multiple nodes via multi-fidelity tuning,

Algorithm 1 Model Training

Input: C ▷ The configuration evaluated
Input: W ▷ A set of workers
Input: M_{CW} ▷ Metrics of Configuration run on Worker
Input: P_{CW} ▷ Performance of Configuration run on Worker
1: $X \leftarrow \{M_{cw} \cup \text{OneHot}(w)\} \quad \forall c \in C, \forall w \in W$
2: $y \leftarrow \frac{P_{cw}}{\mathbb{E}[P_{cw}|C=c]} - 1 \quad \forall c \in C, \forall w \in W$ ▷ Percent Error
3: Model = RandomForestRegressor ∘ Standardize
4: Model.fit(X, y)
Output: Model

Algorithm 2 Model Inference

Input: Model ▷ The model used to predict performance
Input: M_{cw} ▷ Metric for a configuration on a worker
Input: w ▷ Worker ID
Input: P_{cw} ▷ Performance of configuration run on worker
Input: O_c ▷ If configuration has outliers
1: $s \leftarrow$ Model.predict$(M_{cw} \cup \text{OneHot}(w))$
2: $m(s, p, o) := \begin{cases} p & o = True \\ \frac{p}{s+1} & o = False \end{cases}$ ▷ Apply Model
Output: $\hat{P}_{cw} \leftarrow m(s, P_{cw}, O_c)$

giving us additional data that we can leverage for training our predictive model. Note that while such predictive approaches have been utilized for performing outlier detection [60, 93], they have not been used in the context of mean performance prediction or for systems tuning.

There are a few important design decisions we make for our noise adjuster model. First, and most importantly, we do not bring any data from any prior tuning run, *i.e., the model is initialized randomly every time we begin a new tuning run.* We do this to make comparisons more general to new workloads, similar to how the surrogate model in the optimizer is not updated with prior data. We leave transfer learning from prior observations to warm up the model for future work.

We use system metrics and a one-hot encoding of the machine ID collected from the configurations with the most samples within a run as training data to build our model. We do not make any effort to select a subset of metrics (e.g., by using a pre-processing technique), but rather provide all available metrics from `psutil` [71] to our model. For our target metric, we use performance relative to the mean performance of a given configuration (Line 4 of Algorithm 1). This allows our model to learn which metrics are important rather than requiring us to hand-select them per SuT.

Secondly, we require three properties from our model. (i) The model needs to be able to generalize well over unseen data as we cannot expect configurations to take the same code paths. (ii) The model must be able to select important input features (i.e., `psutil` metrics), from a large feature space, without sacrificing predictive performance as we do not wish to encode this information. (iii) And finally, the model must be able be able to be trained on a small amount

of data, as optimization is a cold-start process. We choose to use a random forest regressor as our predictive model as it has been shown to satisfy all of these properties [78].

Finally, we only use data from configs that have been run at the highest budget. This is the most reliable data for training input to model as it has the highest chance of catching and removing unstable configs, which can negatively impact our model accuracy. While this limits the amount of training data, we find that the model still converges quickly to produce more accurate predictions. Additionally, as training random forest regressors is cheap, we simply rebuild the entire model every time there is an added data point. A description of this entire training process can be seen in Algorithm 1.

We then use this model to predict the true mean performance for every sample that we collect from stable configs. If we detect an unstable configuration, we bypass the model as these fall outside of our training data, and will already be heavily penalized by the outlier detector. The inference phase of this model can be described in Algorithm 2. We simply collect the metrics and the machine ID that the configuration was run on and adjust the score we report to the next stage of our sampling process by the performance prediction.

4.4 Sample Aggregation

Finally, we need to generate a single value for the optimizer, and hence we need an aggregation policy. While it may seem intuitive to use median or mean as an aggregation policy, these both have the potential to ignore outliers. Additionally, we find that during tuning, when unstable configurations performed poorly (e.g. Config C in Figure 5a), the final configurations were not unstable as seen in Figure 5b. For this reason, we select `min` as our aggregation policy, as it correctly penalizes unstable configurations, and also optimizes for the worst case. While it is still possible for this aggregation policy to have unpredictable performance above the worst case, the outlier detector (§ 4.2) bounds this uncertainty to 30%. Future work may explore more complicated parametric aggregation policies that balance optimizing for reduced variance *and* increased performance simultaneously.

5 Implementation

We implement all the TUNA components in Python. By default, we use `SMAC` [58] as our optimizer as it supports multi-fidelity sampling natively and has been shown to perform better than other optimizers in the context of system configuration tuning [100] We show, however, that TUNA also supports other optimizers in §6.6. To manage our SuT we use Nautilus [43], a system that manages instantiation and cleanup for benchmarking a given workload [49]. For our evaluation, we run our sampling method for the same amount of time as a traditional single-machine methodology. The rest of this section focuses on some specific implementation decisions used in our evaluation.

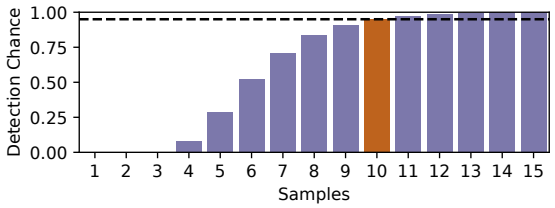


Figure 9. The chance of detecting all unstable configurations in a tuning run based on the number of nodes it was run on. This result is based on the data collected in Section 3.2.1.

5.1 Cluster Size

One important decision is choosing how to manage the nodes that are required for Multi-Fidelity Sampling (§4.1). One option would be to provision a new node for every sample, however this has high overheads from spinup and spindown. Instead, we use a fixed cluster. Determining the size of the cluster is a trade-off between confidence in detecting an unstable configuration as unstable and sample efficiency. While increasing the number of nodes in the system will increase the parallelism, at the same time it will lead to higher costs and increased delays in obtaining samples with the maximum budget. To combat this, we choose to make the cluster as small as possible to give us 95% confidence that we will detect all unstable configurations based on the unstable configurations described in § 3.2.1. We used this data to calculate the average chance of detecting a known unstable configuration when sampling from a given number of nodes. We then use this to find the odds of all unstable configurations being found with a given cluster size during an entire tuning run. The results are shown in Figure 9 and show that a cluster size of 10 is large enough to detect unstable configurations with 95% confidence. Thus, we set the maximum budget as 10 in our implementation.

Given a fixed cluster size, we need to decide how to place sampling tasks across nodes. In our implementation, we reuse old samples taken at a lower budget when we need to sample at a higher budget. This means that we can decrease the cost of sampling at a higher budget, however, we must ensure that the new samples are not performed on the same nodes as the prior runs to ensure the detection guarantees. This is done by maintaining a queue of pending configurations that need to be run, where configurations are placed when an eligible node becomes available. For example, if the optimizer suggests a config at a budget of 3 that was already run at a budget of one on node 1, the two additional required runs will wait until a different node is free.

5.2 Example Pipeline

Figure 10 shows a concrete example of an iteration through the pipeline as follows: The optimizer suggests a config with a budget of 3, which we have previously run with a budget of 1. We find that the config was previously run on worker 1, with a throughput of 500 TPS. Next, we schedule this config

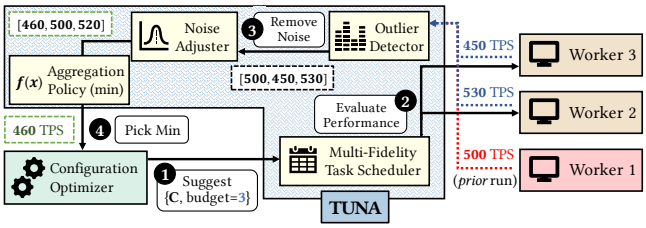


Figure 10. Example Pipeline for TUNA

to run on *two* workers other than worker 1. We then receive the results, of 450 TPS, and 530 TPS. We can then feed all three values (including the prior value) to the outlier detector, which will find that it is a stable config as the relative range (16.2%) is below 30%. Next, as the config is stable, the noise adjuster model adjusts these values to 460, 500, and 520 TPS. Note that if this is the first iteration, we skip the noise adjuster model, as we have not yet seen any training data. Finally, we take the minimum and report 460 TPS back to the optimizer as a result.

6 Evaluation

In this section, we evaluate TUNA's ability to effectively find high-performance configurations, as well as reduce variance and prevent unstable configurations. This means we focus on testing in a way that a real production system would be used: running a tuning methodology offline and then deploying it onto a set of new systems to measure the distribution of performance that a production system could experience. Specifically, we run the best configuration found during tuning on 10 new systems and report the results.

For all our evaluations, during tuning and testing, we use a cluster of 10 worker nodes to measure config performance and 1 orchestrator node, totaling 11 nodes. Unless stated otherwise, all worker nodes use the `D8s_v5` VM SKU with an SSDv2 data disk attached. All systems evaluated are mounted exclusively on this disk. For our optimizer, we use SMAC with a random forest surrogate model, except in §6.6, where we use a *Gaussian Process* optimizer as in OtterTune [98]. This is the same setup that we used in §3.2.1. The orchestrator node is another `D8s_v5` VM without a data disk attached, as the performance variability of the optimizer does not impact the suggestions created by the optimizer.

For our experiments, we compare against the sampling technique used in prior state-of-the-art works [42, 57, 98, 99]: a single-node sequentially evaluating suggested configurations, without any repeated samples. We will refer to this as *traditional sampling*. Both traditional sampling and TUNA are evaluated for 8 hours. Additionally, we compare TUNA and traditional sampling with the default (untuned) configuration. To compare these three, we compare the performance and standard deviation of performance during deployment, not tuning. Transactional Workloads (OLTP) and workloads minimizing latency are evaluated for 5 minutes (as in prior work). Analytical workloads (OLAP) target minimizing runtime and have a workload-dependent runtime.

6.1 Generalizability Across Workloads

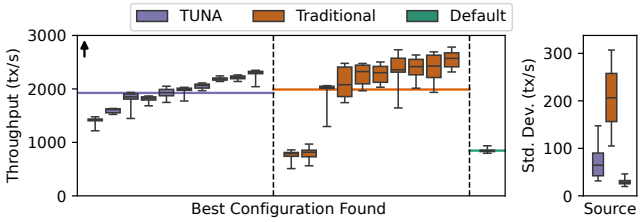
First, we fix the SuT, and vary the workload to show that TUNA can generalize across workloads. For our first system, we choose PostgreSQL 16.1 as DBMS autotuning has been a hot area of work, and has already been demonstrated to be a good candidate for autotuning . For workloads, we select TPC-C [56], and TPC-H [66] as in prior work [98], and additionally evaluate epinions [70], and `mssales`, a production OLAP workload from Microsoft.

OLTP workloads. First, for evaluating OLTP workloads, we start with `TPC-C`, as seen in Figure 11a. There is one `JOIN` query, and otherwise the rest of the transactions consist of simple single table queries. For `TPC-C`, TUNA achieves 1925 TPS on average, a 127% improvement over the default, while the traditional sampling setup achieves 1989 TPS on average, a 134% improvement over the default. While traditional sampling finds slightly higher performance on average, looking towards the low-performing traditional sampling configurations, two runs perform worse on average than the default. These two configurations were selected because they performed well (1744 TPS and 2040 TPS) during training. This is one of the major risks of traditional sampling on single nodes. Additionally, we see that traditional sampling is significantly more likely to find unstable configurations. Configurations found using TUNA have an average standard deviation of 69.0 TPS, whereas configurations found using traditional sampling methods have an average standard deviation of 205.7 TPS, a 198.1% increase.

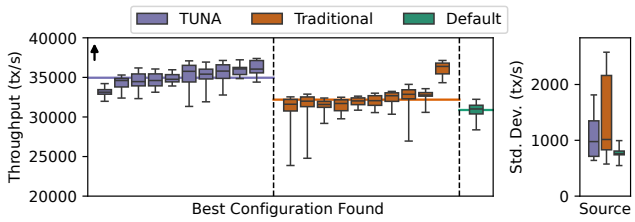
Next, we evaluate `epinions`, a transactional workload that represents a blog website. The queries in `epinions` are simpler than those in `TPC-C`, but `Epinions` experiences many of the same problems as TPC-C. In general, we find that TUNA achieves 34957 TPS on average, a 13.2% improvement over the default, while the traditional sampling setup achieves 32189 TPS on average, a 4.2% improvement over the default. Convergence to the optimal configuration is much slower than for TPC-C when there is variability, as seen in § 3.1. This workload allows us to show the benefits of the noise adjuster model in converging to a higher performance level than in traditional sampling.

Additionally, we find that 3 configurations learned from traditional sampling are unstable, with a standard deviation greater than 2000. Once again, these configurations all performed above 37,000 TPS during tuning, but the instability was missed by only sampling from one node. While there is one configuration from TUNA that had higher variability, the CoV was only 5.15%, and in the worst case, the performance seen was still 1.46% better than the mean performance of the default configuration.

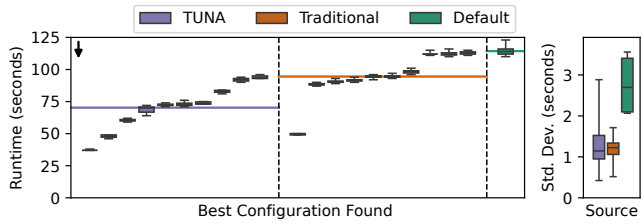
OLAP workloads. We next move on to OLAP workloads where we try to minimize runtime. We start by examining



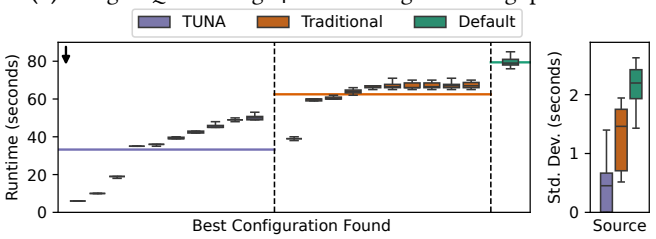
(a) PostgreSQL running TPC-C. Higher Throughput is better.



(b) PostgreSQL running epinions. Higher throughput is better.



(c) PostgreSQL running TPC-H. Lower Runtime is better.



(d) PostgreSQL running mssales. Lower Runtime is better.

Figure 11. Results of applying tuned PostgreSQL configs with several workloads to new systems. The horizontal lines are mean throughput (or runtime). In all cases, TUNA either improves performance, reduces variability, or both.

TPC-H in Figure 11c, an analytical workload with many, relatively easy, JOINs [53]. Here, we optimize for workload completion time, a realistic metric for analytical workloads. For TUNA, on average we can complete the workload in 70.3 seconds, a 38.6% improvement over the default, while the traditional sampling setup achieves 94.5 TPS on average, a 17.3% improvement over the default. In relative terms TUNA, on average, finds configurations that are 38.4% faster on average with 53.4% lower standard deviation compared to the default. The standard deviation of TUNA and traditional sampling methodologies were found to be: 1.3 and 1.2 respectively – both extremely low values. We believe that this in large part is simply because there are no unstable configurations seen that were optimal for this workload, and the variability simply comes from the platform. Overall, we can show that TUNA can converge to a faster configuration without significant changes to stability.

Finally, we present mssales, a production analytical workload, with many complex JOIN , to show that our techniques are not limited to synthetic workloads. As seen in Figure 11d, TUNA, on average, finds a configuration that completes the workload in 33.2 seconds, with a standard deviation of 0.49 seconds. Traditional sampling, on the other hand, on average, finds a configuration that completes the workload in 62.5 seconds, with a standard deviation of 1.26 seconds. In relative terms, TUNA finds a configuration that is 47.8% faster with 61.2% lower standard deviation, however, it is important to note that the standard deviation is again quite small for both systems. Overall, this result is quite impressive as the default configuration, on average, takes 79.4 seconds. This means that the single-node system finds only marginal improvements, where TUNA can find significant improvements, sometimes as high as 2.39x speedup.

6.2 Generalizability Across Regions

Next, to show that TUNA generalizes across regions with different variability characteristics, we repeat the evaluations in the centralus azure region. We continue to tune PostgreSQL 16.1, and run TPC-C. We report the results of this experiment in Figure 12, Notably, the variability in this region is higher than the results shown in Figure 11a in the sense that there are fewer high-performing machines, as shown by the large gaps between the upper quartile and the maximum throughput in the boxplots, which did not exist in the prior experiments. Empirically, we find TUNA achieves 2321 TPS on average, a 669% improvement over the default, while the traditional sampling setup achieves 2239 TPS on average, a 642% improvement over the default. Even with the higher variability, TUNA still mitigates the variability better than traditional sampling, with the average standard deviation of TUNA being 113.0 TPS, and traditional sampling being 267.7 TPS, a 57.8% improvement.

6.3 Generalizability Across Hardware

To show TUNA generalizes across platforms, we repeat the evaluation on CloudLab [23], a research platform providing bare-metal nodes. We continue to tune PostgreSQL 16.1, and run TPC-C on c220g5 nodes. As seen in Figure 13, TUNA achieves 5756 TPS on average, a 19.1x improvement over the default, while the traditional sampling setup achieves 5380 TPS on average, a 17.8x improvement over the default. Part of the reason for this extreme performance improvement is that the default configuration does not fully utilize the system memory. We find that traditional tuning finds many configurations with 7.71x higher standard deviation than TUNA. Also, we can see that 8 out of 10 of these configurations found using traditional sampling are unstable, while all

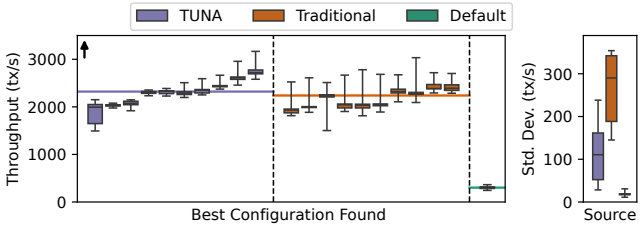


Figure 12. Results of applying tuned PostgreSQL configs running TPC-C in a new region. Higher throughput is better.

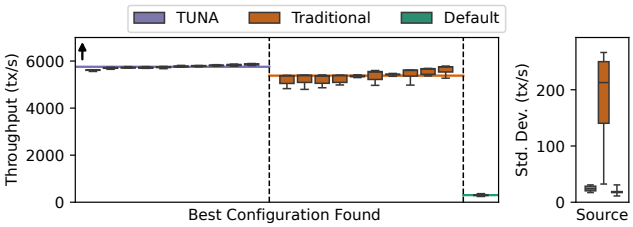


Figure 13. Results of tuned PostgreSQL configs running TPC-C on bare-metal CloudLab Nodes. Higher is better.

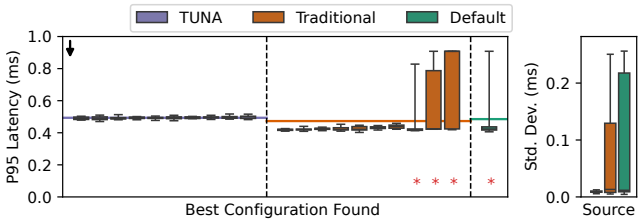


Figure 14. Redis running tuned YCSB-C workload configs. Lower is better. A red asterisk indicates that a node crashed.

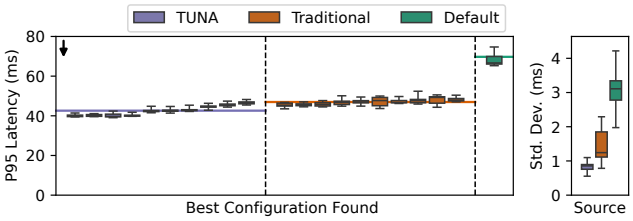


Figure 15. Results of applying tuned NGINX configs serving the top 500 Wikipedia pages. Lower is better.

of the configurations found using TUNA are exceptionally stable and, on average, perform 7.0% better than those found using traditional sampling.

6.4 Generalizability Across Systems

To show that our system generalizes across different scenarios, we next perform experiments across two new SuT: Redis , and NGINX . Neither of these two systems natively supports any of the workloads that we tested in the previous section, so we also introduce new workloads.

For Redis , we use YCSB-C [17], a well-known read-only workload with a Zipfian distribution. Here, we target 95th percentile latency as Redis is often chosen to minimize latency rather than maximize throughput. With this system, 3 configurations found using traditional sampling crashed Redis 30% of the time on average. Furthermore, the default configuration crashed Redis 8% of the time. These system crashes were caused by an out of memory exception, caused by an overly aggressive configuration. For these crashed runs, we follow methodology from [88], and replace them with the worst P95 latency seen (.908*ms*) on the default configuration as a conservative penalty, rather than ∞, mostly for visualization purposes.

With this data-cleaning step, we find that TUNA has 1.7% higher latency than the default, and 4.3% higher latency than traditional sampling on average. However, in this case the main benefit of TUNA is that *no configurations crashed*. This results in TUNA, on average, having 27.5% lower standard deviation than the default, and 86.8% lower standard deviation than traditional sampling.

For NGINX , we use a custom workload serving Wikipedia. We serve the Top 500 Wikipedia pages in 2023, including all media content, in the same distribution as these pages were accessed over this same period. The reported latency is the latency to serve the entire page, including all We target 95th percentile latency, as web servers often target minimizing high percentile latencies. As seen in Figure 15, on average, the 95th percentile latency of the workload with TUNA is 42.6 ms, a 38.9% improvement over the default, while the traditional sampling setup achieves 46.6 ms, a 32.7% improvement over the default. We also find that the standard deviations for TUNA and traditional sampling are 0.82 ms and 1.46 ms, respectively. TUNA has a standard deviation that is 63.3% lower than when using traditional sampling.

6.5 Equal Cost Comparisons

Previously, throughout the evaluation, we compared TUNA against traditional sampling in the setting where both approaches take equal time. An alternative to this is tuning with equal cost. This means that each sampling policy should use an equivalent number of samples. There are two main ways that an equal-cost budget could be implemented. First, one could simply extend the traditional sampling technique, as described at the start of §6, to run for more iterations, to match the number of samples used by TUNA. Alternatively, one could use a cluster for tuning, and run each configuration on every node in the cluster (equivalent to max budget in TUNA), and aggregate the result for the optimizer. To compare TUNA against both these equal-cost baselines, we again tune PostgreSQL running TPC-C on Azure.

6.5.1 Extended Traditional Sampling. Simply extending traditional sampling exacerbates the problem of instability previously seen (Figure 11a). Examining the results shown in Figure 16, we can see that while the maximum performance is higher, the variability also becomes even higher. We can see that in this case, because of the extreme performance degradation seen on nodes during deployment

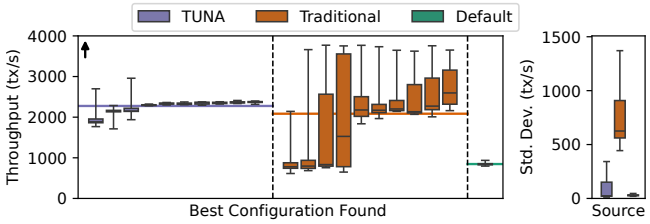


Figure 16. Results of applying tuned PostgreSQL configs running TPC-C comparing TUNA and traditional sampling when budgets (i.e., costs) are both 500. Higher is better.

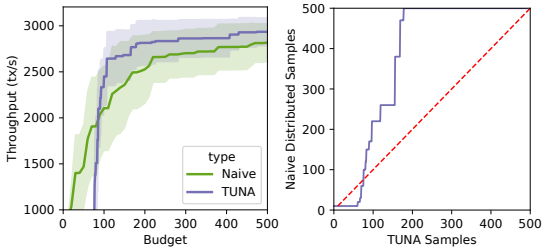


Figure 17. Results of tuning PostgreSQL running TPC-C comparing TUNA to naive distributed implementation. Plot on the left shows convergence rates in terms of absolute performance. Plot on the right shows convergence gain comparisons where above the dashed line implies TUNA converges faster, and below the line, naive distributed is better.

using traditional sampling, TUNA's average performance is now 9.2% higher than in traditional tuning with 87.8% lower standard deviation.

6.5.2 Naive Distributed Sampling. The alternative of simply running every configuration on every node causes extremely slow convergence. As an aggregation policy is now required to report a single score to the optimizer, we will use the same aggregation policy as TUNA, `min`. To compare convergence rates, we compare the number of samples TUNA takes to achieve the same performance as a naive distributed implementation. Initially, we see that TUNA converges slower than the Naive Distributed implementation. This is because both systems only compare configurations run at the maximum budget. The multi-fidelity sampling technique used in TUNA does not evaluate at higher budgets until sufficient configurations have been evaluated at a lower budget. Once TUNA begins to evaluate configurations at the maximum budget, at around 100 samples, TUNA quickly jumps ahead of the performance of the naive distributed implementation. The final result ends with TUNA achieving the same performance within 206 samples on average, or in other words, TUNA converges 2.47x faster. If TUNA is allowed to continue running until 500 samples, it will achieve 4.1% higher performance. This is not higher in large part because there are significant diminishing returns to tuning for

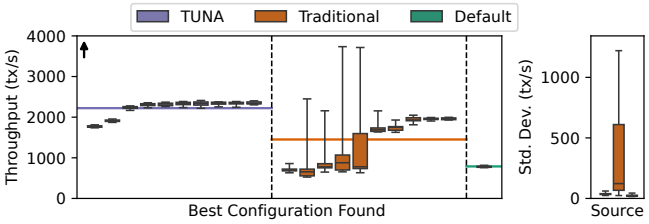
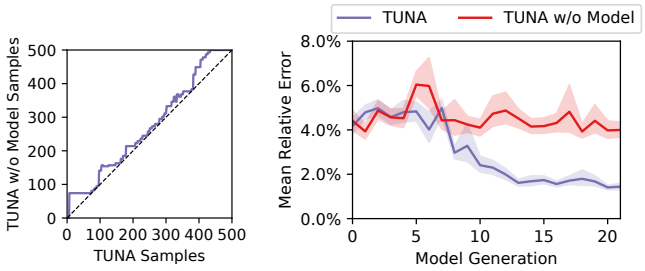


Figure 18. Results of applying tuned PostgreSQL configs running TPC-C with a GP optimizer. Higher is better.



(a) TUNA convergence gains, with and without the noise adjuster model. Being above dashed line means TUNA is better.

(b) Comparison of relative error between reported values with and without our noise adjuster model. We show that our model can remove 67.3% of the noise.

Figure 19. Convergence comparisons for 100 tuning runs on TUNA and TUNA without the noise adjuster model.

longer. Prior work has discussed intelligent stopping criteria, however, this is orthogonal to our approach [26].

6.6 Component Analysis

Optimizer. To show the generality of TUNA, we replace the optimizer used in both TUNA and traditional techniques. We tune PostgreSQL 16.1, and run TPC-C, as seen in Figure 18. On average, TUNA achieves 53.1% higher performance with 89.5% lower standard deviation than traditional sampling.

Noise Adjuster Model. To evaluate the performance of our noise adjuster model, we run an ablation study comparing TUNA against itself with the noise adjuster model removed. We tune PostgreSQL running `epinions` with the same setup as described in § 6.1. We run each system for 100 tuning runs, and report the results in Figure 19.

First, we examine the convergence rates between the two systems in Figure 19a. This figure shows the number of iterations it takes for TUNA to converge to the same performance as TUNA without the noise adjuster model. We can see immediately that the full TUNA system is always able to converge faster than when it lacks the noise adjuster model, as the performance is always above the diagonal. On average, the model allows TUNA to converge 13.3% faster.

This improvement in performance can be attributed to the the scores we report to the optimizer being closer to the ground truth and hence more consistent. We compare against our ground truth value, i.e. runs with the max budget, (as

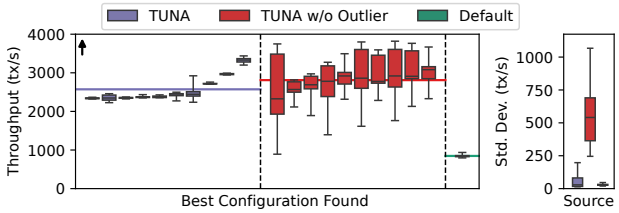


Figure 20. Comparison of TUNA with and without the Outlier Detector component running TPC-C on PostgreSQL 16.1. Although configs found using TUNA have a slightly lower mean performance by 8.5%, their variability is substantially slower by 91% on average.

described in §4.3). It is important to note that the inference step happens before the training step in our pipeline, so we do not leak any information to the noise adjuster model during this process. The results of this experiment can be seen in Figure 19b. We can see that at the beginning of the run, while the model has very few samples to learn from, it is not able to make significant improvements over a system without the model. By the halfway mark and beyond, the error in the system without the model is 4.87%, whereas, on the system with the model, it is 1.99%, a 53.34% relative reduction in error. On average, for the duration of the entire run, the model reduces the error by 35.8% including the initial runs in which the model has not seen much data. From midpoint of the run onwards, the model reduces the error by 59.2% on average. This reduction in error allows the optimizer to have a more accurate representation of the ground truth performance for a sample.

Impact of Outlier Detector. To evaluate the performance of the outlier detector, we compare our full system, TUNA, against itself with the outlier detector removed. The result of this experiment is shown in Figure 20. We find that TUNA finds performance values of 2572 TPS, with an average standard deviation of 54.8 TPS. The system with the outlier detection components removed finds configurations that, on average, have 2810 TPS and an average standard deviation of 550.8 TPS. Although it is not surprising that the optimizer is able to find configurations with higher performance, as this is its only objective, it is significantly limited by the fact that many of these configurations are extremely unstable. Configurations found with TUNA have 10.1x lower variability than a system without these components. From these results, we believe that this shows the necessity of *unstable* configuration aware sampling.

7 Future Work

We see four main areas of future work to further improve this sampling methodology. First, we did not constrain how much our noise model (§ 4.3) can adjust the result. While we did not experience any issues during evaluation, or testing, it is foreseeable that there exists a pattern of data that could cause unpredictable results from the model. For a production setting, it may be wise to include guardrails limiting how much the model can adjust the score for the optimizer. Building these guardrails would, however, require additional knowledge about the variability of the system.

An alternative approach to the outlier detector would be to have a scaling penalty based on the relative range seen, rather than a penalty for crossing a threshold. This is something that has been studied in economics [14, 46, 47, 63, 73], but for the context of tuning, a hyperparameter for penalty is required. This is something that we wished to avoid.

Our noise adjuster model only utilizes data for predictions from within a run which can delay its effectiveness. Alternatively, we could transfer data from prior runs to warm-start the model. This has technical challenges, particularly when using a different set of machines or workloads for warm-start information. This is in many ways similar to the transfer of the surrogate model in the context of autotuning.

Finally, we do not address burstable or serverless nodes, in large part because it is difficult to differentiate between an unstable configuration and a credit depletion. We ignore this as without platform introspection it is very difficult to determine the root cause. Additionally, a system that is targeting burstable VMs should generate a configuration that can perform well for both a VM with and without credits.

8 Conclusion

In this paper, we explored ideas around the impacts of performance noise on cloud system autotuning , and discovered two main problems. If one does not account for performance variability during tuning, convergence rates to an optimal configuration are slower and more costly, and configurations may be unstable during deployment. We further motivate our system by investigating the magnitude of noise in the cloud, and that while some components have virtually no noise, others have much higher noise than bare metal components.

We solve these problems through three main mechanisms: a multi-fidelity sampling process, an outlier detector, and a simple model built from component-level metrics to mitigate noise during sampling. We then integrate this into a state-of-the-art tuning setup and evaluate it across 6 workloads, 3 systems, 2 deployment regions, and 2 hardware setups, finding that TUNA reduces variability and improves the performance of tuned systems.

Acknowledgments

We would like to thank our anonymous reviewers and our shepherd, Thomas Pasquier. We would also like to thank the Microsoft Gray Systems Lab for providing Azure cloud credits and for early feedback from several members, including Carlo Curino, Yuanyuan Tian, and Andreas Mueller. This material is based upon work supported by the National Science Foundation under Grant No. 2326576.

References

[1] Ali Abedi and Tim Brecht. 2017. Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 287–292. https://doi.org/10.1145/3030207.3030229

[2] Ali Abedi and Tim Brecht. 2017. Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 287–292. https://doi.org/10.1145/3030207.3030229

[3] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft SQL server 2005: demo. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 930–932. https://doi.org/10.1145/1066157.1066292

[4] Jose Nelson Amaral, Edson Borin, Dylan R. Ashley, Caian Benedicto, Elliot Colp, Joao Henrique Stange Hoffmam, Marcus Karpoff, Erick Ochoa, Morgan Redshaw, and Raphael Ernani Rodrigues. 2018. The Alberta Workloads for the SPEC CPU 2017 Benchmark Suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 159–168. https://doi.org/10.1109/ISPASS.2018.00029

[5] Amazon 2024. Amazon EBS volume types. https://docs.aws.amazon.com/ebs/latest/userguide/ebs-volume-types.html

[6] Amazon 2024. Amazon EC2 Instance types. https://aws.amazon.com/ec2/instance-types/

[7] Esmail Asyabi, Mohsen Sharifi, and Azer Bestavros. 2018. ppXen: A hypervisor CPU scheduler for mitigating performance variability in virtualized clouds. *Future Generation Computer Systems* 83 (2018), 75–84. https://doi.org/10.1016/j.future.2018.01.015

[8] Abhinav Atla, Rahul Tada, Victor Sheng, and Naveen Singireddy. 2011. Sensitivity of different machine learning algorithms to noise. *J. Comput. Sci. Coll.* 26, 5 (may 2011), 96–103. https://dl.acm.org/doi/abs/10.5555/1961574.1961594

[9] Jens Axboe and Vincent Fum. 2021. Fio. https://github.com/axboe/fio

[10] S. Balakrishnan, Ravi Rajwar, M. Upton, and K. Lai. 2005. The impact of performance asymmetry in emerging multicore architectures. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, Madison, WI, 506–517. https://doi.org/10.1109/ISCA.2005.51

[11] Maximilian Balandat, Brian Karrer, Daniel Jiang, Samuel Daulton, Ben Letham, Andrew G Wilson, and Eytan Bakshy. 2020. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 21524–21538. https://proceedings.neurips.cc/paper_files/paper/2020/file/f5b1b89d98b7286673128a5fb112cb9a-Paper.pdf

[12] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. 2018. Towards Better Understanding of Black-box Auto-Tuning: A Comparative Analysis for Storage Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 893–907. https://dl.acm.org/doi/10.5555/3277355.3277441

[13] Giuliano Casale and Mirco Tribastone. 2013. Modelling exogenous variability in cloud deployments. *SIGMETRICS Perform. Eval. Rev.* 40, 4 (April 2013), 73–82. https://doi.org/10.1145/2479942.2479951

[14] Tun-Jen Chang, Sang-Chin Yang, and Kuang-Jung Chang. 2009. Portfolio optimization problems in different risk measures using genetic algorithm. *Expert Systems with Applications* 36, 7 (2009), 10529–10537. https://doi.org/10.1016/j.eswa.2009.02.062

[15] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proc. VLDB Endow.* 15, 1 (Sept. 2021), 112–126. https://doi.org/10.14778/3485450.3485461

[16] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3297858.3304005

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[18] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[19] Daniele De Sensi, Tiziano De Matteis, Konstantin Taranov, Salvatore Di Girolamo, Tobias Rahn, and Torsten Hoefler. 2022. Noise in the Clouds: Influence of Network Performance Variability on Application Scalability. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 3, Article 49 (Dec. 2022), 27 pages. https://doi.org/10.1145/3570609

[20] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150. https://doi.org/10.1145/1327452.1327492

[21] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[22] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 1246–1257. https://doi.org/10.14778/1687627.1687767

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14. https://dl.acm.org/doi/10.5555/3358807.3358809

[24] Jamie Ericson, Masoud Mohammadian, and Fabiana Santana. 2017. Analysis of Performance Variability in Public Cloud Computing. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, San Diego, CA, USA, 308–314. https://doi.org/10.1109/IRI.2017.47

[25] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. 2012. More for Your Money: Exploiting Performance Heterogeneity in Public Clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 20, 14 pages. https://doi.org/10.1145/2391229.2391249

[26] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. 2020. To Tune or Not to Tune? In Search of Optimal Configurations for Data Analytics. In *Proceedings of the 26th*

ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 2494–2504. https://doi.org/10.1145/3394486.3403299

[27] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. 2018. Performance evaluation of heterogeneous cloud functions. *Concurrency and Computation: Practice and Experience* 30, 23 (2018). https://doi.org/10.1002/cpe.4792

[28] Peter I Frazier. 2018. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811* (2018). https://doi.org/10.48550/arXiv.1807.02811

[29] Marcus Geelnard. 2023. osbench. https://gitlab.com/mbitsnbites/osbench

[30] Google 2024. CoreMark scores of VMs by family. https://cloud.google.com/compute/docs/coremark-scores-of-vm-instances

[31] Google 2024. Storage options. https://cloud.google.com/compute/docs/disks

[32] Jim Gray. 1986. Why do computers stop and what can be done about it?. In *Symposium on reliability in distributed software and database systems*. Los Angeles, CA, USA, 3–12. https://doi.org/10.1109/MC.1985.1662717

[33] PostgreSQL Global Development Group. 2022. pgbench. https://www.postgresql.org/docs/15/pgbench.html

[34] Abhishek Gupta, Paolo Faraboschi, Filippo Gioachin, Laxmikant V. Kale, Richard Kaufmann, Bu-Sung Lee, Verdi March, Dejan Milojicic, and Chun Hui Suen. 2016. Evaluating and Improving the Performance and Scheduling of HPC Applications in Cloud. *IEEE Transactions on Cloud Computing* 4, 3 (2016), 307–321. https://doi.org/10.1109/TCC.2014.2339858

[35] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 150–155. https://doi.org/10.1145/3102980.3103005

[36] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization*, Carlos A. Coello Coello (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 507–523. https://doi.org/10.1007/978-3-642-25566-3_40.

[37] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. 2011. On the Performance Variability of Production Cloud Services. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, Newport Beach, CA, USA, 104–113. https://doi.org/10.1109/CCGrid.2011.22

[38] Kevin Jamieson and Ameet Talwalkar. 2015. Non-stochastic Best Arm Identification and Hyperparameter Optimization. arXiv:1502.07943 [cs.LG] https://doi.org/10.48550/arXiv.1502.07943

[39] Andy Jia, Micah McKittrick, Styli Tsinaroglou, and Joel Pelley. 2023. DV5 and DSV5-Series - Azure Virtual Machines. https://learn.microsoft.com/en-us/azure/virtual-machines/dv5-dsv5-series

[40] Shrinivas B. Joshi. 2012. Apache hadoop performance-tuning methodologies and best practices. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering* (Boston, Massachusetts, USA) *(ICPE '12)*. Association for Computing Machinery, New York, NY, USA, 241–242. https://doi.org/10.1145/2188286.2188323

[41] Shrinivas B. Joshi. 2012. Apache hadoop performance-tuning methodologies and best practices. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering* (Boston, Massachusetts, USA) *(ICPE '12)*. Association for Computing Machinery, New York, NY, USA, 241–242. https://doi.org/10.1145/2188286.2188323

[42] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: sample-efficient DBMS configuration tuning. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2953–2965. https://doi.org/10.14778/3551793.3551844

[43] Konstantinos Kanellis, Johannes Freischuetz, and Shivaram Venkataraman. 2024. Nautilus: A Benchmarking Platform for DBMS Knob Tuning. In *Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning*. 72–76.

[44] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. 2008. Characterization of storage workload traces from production Windows Servers. In *2008 IEEE International Symposium on Workload Characterization*. 119–128. https://doi.org/10.1109/IISWC.2008.4636097

[45] Colin Ian King. 2023. stress-ng. https://wiki.ubuntu.com/Kernel/Reference/stress-ng.

[46] Hiroshi Konno, Hayato Waki, and Atsushi Yuuki. 2002. Portfolio optimization under lower partial risk measures. *Asia-Pacific Financial Markets* 9 (2002), 127–140. https://doi.org/10.1023/A:1022238119491

[47] Hiroshi Konno and Hiroaki Yamazaki. 1991. Mean-absolute deviation portfolio optimization model and its applications to Tokyo stock market. *Management science* 37, 5 (1991), 519–531. https://doi.org/10.1287/mnsc.37.5.519

[48] Alexey Kopytov. 2020. sysbench. https://github.com/akopytov/sysbench

[49] Brian Kroth, Sergiy Matusevych, Rana Alotaibi, Yiwen Zhu, Anja Gruenheid, and Yuanyuan Tian. 2024. MLOS in Action: Bridging the Gap Between Experimentation and Auto-Tuning in the Cloud. *Proc. VLDB Endow.* 17, 12 (2024). https://doi.org/doi:10.14778/3685800.3685852

[50] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software microbenchmarking in the cloud. How bad is it really? *Empirical Software Engineering* 24, 4 (2019), 2469–2508. https://doi.org/10.1007/s10664-019-09681-1

[51] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2023. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. arXiv:2311.03157 [cs.DB] https://doi.org/10.14778/3659437.3659449

[52] A. Leff, J.T. Rayfield, and D.M. Dias. 2003. Service-level agreements and commercial grids. *IEEE Internet Computing* 7, 4 (2003), 44–50. https://doi.org/10.1109/MIC.2003.1215659

[53] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[54] Philipp Leitner and Jürgen Cito. 2016. Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.* 16, 3, Article 15 (apr 2016), 23 pages. https://doi.org/10.1145/2885497

[55] Benjamin Letham, Brian Karrer, Guilherme Ottoni, and Eytan Bakshy. 2019. Constrained Bayesian Optimization with Noisy Experiments. *Bayesian Analysis* 14, 2 (2019), 495 – 519. https://doi.org/10.1214/18-BA1110 https://doi.org/10.1214/18-BA1110.

[56] Scott T Leutenegger and Daniel Dias. 1993. A modeling study of the TPC-C benchmark. *ACM Sigmod Record* 22, 2 (1993), 22–31. http://doi.org/10.1145/170036.170042

[57] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[58] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. 2022. SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization. *Journal of Machine Learning Research* 23, 54 (2022), 1–9. http://jmlr.org/papers/v23/21-0888.html https://jmlr.org/papers/v23/21-0888.html.

[59] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, Orlando, FL, USA, 159–169. https://doi.org/10.1109/IC2E.2018.00039

[60] Tania Lorido-Botran, Sergio Huerta, Luis Tomás, Johan Tordsson, and Borja Sanz. 2017. An unsupervised approach to online noisy-neighbor detection in cloud data centers. *Expert Systems with Applications* 89 (2017), 188–204. https://doi.org/10.1016/j.eswa.2017.07.038

[61] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[62] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming Performance Variability. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 409–425. https://dl.acm.org/doi/10.5555/3291168.3291198

[63] Harry Markowitz. 1952. Portfolio Selection. *The Journal of Finance* 7, 1 (1952), 77–91. http://www.jstor.org/stable/2975974

[64] Oracle 2024. Compute Shapes. https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm#vm-standard

[65] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 10, 16 pages. https://doi.org/10.1145/3302424.3303963

[66] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71. http://doi.org/10.1145/369275.369291

[67] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, et al. 2018. Measuring the impact of spectre and meltdown. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–5.

[68] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. 2013. Who Is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds. *IEEE Transactions on Services Computing* 6, 3 (2013), 314–329. https://doi.org/10.1109/TSC.2012.2

[69] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[70] Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. 2003. Trust management for the semantic web. In *International semantic Web conference*. Springer, 351–368. https://doi.org/10.1007/978-3-540-39718-2_23.

[71] Giampaolo Rodola. 2024. psutil. https://github.com/giampaolo/psutil

[72] Roygara, Masha Thomas, Becca Goodman, Jeff Borsecnik, Henry Hagnäs, Raman Kumar, Sumanth Marigowda, Stephen Haas, Micah McKittrick, David Coulter, and et al. 2024. Azure managed disk types. https://learn.microsoft.com/en-us/azure/virtual-machines/disks-types#premium-ssd-v2

[73] Paul A. Samuelson. 1970. The Fundamental Approximation Theorem of Portfolio Analysis in terms of Means, Variances and Higher Moments1. *The Review of Economic Studies* 37, 4 (10 1970), 537–542. https://doi.org/10.2307/2296483 arXiv:https://academic.oup.com/restud/article-pdf/37/4/537/4353167/37-4-537.pdf

[74] Salvatore Sanfilippo. 2022. Redis Benchmark. https://redis.io/docs/management/optimization/benchmarks/

[75] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 460–471. https://doi.org/10.14778/1920841.1920902

[76] Joel Scheuner and Philipp Leitner. 2018. A Cloud Benchmark Suite Combining Micro and Applications Benchmarks. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 161–166. https://doi.org/10.1145/3185768.3186286

[77] K. Bernhard Schiefer and Gary Valentin. 1999. DB2 universal database performance tuning. *IEEE Data Eng. Bull.* 22, 2 (1999), 12–19.

[78] Mark R Segal. 2004. Machine learning benchmarks and random forest regression. *CSF: Center for Bioinformatics and Molecular Biostatistics* (2004).

[79] Prasoon Sinha, Akhil Guliani, Rutwik Jain, Brandon Tran, Matthew D. Sinclair, and Shivaram Venkataraman. 2022. Not All GPUs Are Created Equal: Characterizing Variability in Large-Scale, Accelerator-Rich Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Dallas, Texas, Article 65, 15 pages. https://dl.acm.org/doi/abs/10.5555/3571885.3571971

[80] D. Skinner and W. Kramer. 2005. Understanding the causes of performance variability in HPC workloads. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* IEEE Press, Austin, TX, 137–149. https://doi.org/10.1109/IISWC.2005.1526010

[81] Michael Stonebraker and Lawrence A. Rowe. 1986. The design of POSTGRES. *SIGMOD Rec.* 15, 2 (June 1986), 340–355. https://doi.org/10.1145/16856.16888

[82] William R. Thompson. 1933. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika* 25, 3/4 (1933), 285–294. http://www.jstor.org/stable/2332286

[83] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. 2009. A scalable auto-tuning framework for compiler optimization. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2009.5161054

[84] Inside Track. 2020. https://www.microsoft.com/insidetrack/blog/microsoft-reinvents-sales-processing-and-financial-reporting-with-azure/

[85] Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool that "Reads the Manual". In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 190–203. https://doi.org/10.1145/3514221.3517843

[86] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56. https://doi.org/10.1109/MC.2005.163

[87] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. 2020. Is Big Data Performance Reproducible in Modern Cloud Networks?. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 513–527. https://www.usenix.org/conference/nsdi20/presentation/uta

[88] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-Based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (apr 2021), 1241–1253. https://doi.org/10.14778/3450980.3450992

[89] Paul Veitch, Edel Curley, and Tomasz Kantecki. 2017. Performance evaluation of cache allocation technology for NFV noisy neighbor mitigation. In *2017 IEEE Conference on Network Softwarization (NetSoft)*. 1–5. https://doi.org/10.1109/NETSOFT.2017.8004214

[90] Rishab Verma, Jeff Borsecnik, Willie Williams, Aarthi Vijayaraghavan, Micah Mckittrick, Styli Tsinaroglou, Justin Chizer, Stephen Haas, Alec, Zach Olinske, and et al. 2024. B-series burstable virtual machine sizes. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable

[91] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, and Sri Sakthivelu. 2021. Intel Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[92] Chengwei Wang, Vanish Talwar, Karsten Schwan, and Parthasarathy Ranganathan. 2010. Online detection of utility cloud anomalies using metric distributions. In *2010 IEEE Network Operations and Management Symposium - NOMS 2010*. 96–103. https://doi.org/10.1109/NOMS.2010.5488443

[93] Chengwei Wang, Vanish Talwar, Karsten Schwan, and Parthasarathy Ranganathan. 2010. Online detection of utility cloud anomalies using metric distributions. In *2010 IEEE Network Operations and Management Symposium - NOMS 2010*. 96–103. https://doi.org/10.1109/NOMS.2010.5488443

[94] Guolu Wang, Jungang Xu, and Ben He. 2016. A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 586–593. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0088 http://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0088.

[95] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. 2017. Selecting the best VM across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 452–465. https://doi.org/10.1145/3127479.3131614

[96] Andrew Yoo, Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. 2021. Fail-Slow Fault Tolerance Needs Programming Support. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 228–235. https://doi.org/10.1145/3458336.3465299

[97] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. 2021. Don't Forget the I/O When Allocating Your LLC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 112–125. https://doi.org/10.1109/ISCA52012.2021.00018

[98] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. 2018. A Demonstration of the Ottertune Automatic Database Management System Tuning Service. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1910–1913. https://doi.org/10.14778/3229863.3236222

[99] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 415–432. https://doi.org/10.1145/3299869.3300085

[100] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating Database Tuning with Hyper-Parameter Optimization: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 15, 9 (jul 2022), 1808–1821. https://doi.org/10.14778/3538598.3538604

[101] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 379–391. https://doi.org/10.1145/2465351.2465388

[102] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. 2022. Towards Dynamic and Safe Configuration Tuning for Cloud Databases. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 631–645. https://doi.org/10.1145/3514221.3526176

[103] Zongpu Zhang, Jiangtao Chen, Banghao Ying, Yahui Cao, Lingyu Liu, Jian Li, Xin Zeng, Junyuan Wang, Weigang Li, and Haibing Guan. 2024. HD-IOV: SW-HW Co-designed I/O Virtualization with Scalability and Flexibility for Hyper-Density Cloud. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 834–850. https://doi.org/10.1145/3627703.3629557

[104] Aviad Zuck, Philipp Gühring, Tao Zhang, Donald E. Porter, and Dan Tsafrir. 2019. Why and How to Increase SSD Performance Transparency. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 192–200. https://doi.org/10.1145/3317550.3321430

Appendices

A Artifact Appendix

A.1 Abstract

In our artifact, we release the source code used for "TUNA: Tuning Unstable and Noisy Cloud Applications", the data from the tuning runs, and the longitudinal data used in the motivation. The code can be used to run the main experimental results from this paper. Users interested in using these techniques for their own works can make small modifications to the provided scripts to integrate their systems.

A.2 Description & Requirements

A.2.1 How to access. We have released the code for running TUNA on GitHub [3]. The data from the tuning runs will also be available here. Detailed instructions on how to run the experiment can be found on GitHub. We have also released the longitudinal study datasets at [4]. This includes all of the data required to make the figures for motivation. These repositories can simply be copied from the respective

[3] https://aka.ms/mlos/tuna-eurosys-artifacts
[4] https://aka.ms/mlos/tuna-eurosys-dataset

code bases. Note, that we use submodules, so we recommend pulling using the instructions listed in the repository.

A.2.2 Hardware dependencies. The experiments in this paper were run on Azure and CloudLab. We specifically ran D8s_v5 VMs on Azure, and c220g5 nodes on CloudLab.

A.2.3 Software dependencies. We have a relatively long list of dependencies that are described in the GitHub README file. There are scripts that will automatically install them into a virtual environment on all nodes in the cluster.

A.2.4 Benchmarks. We use a series of benchmarks from Benchbase [21] and YCSB [17]. These are managed by `nautilus`, and are automatically installed through the `docker` containers.

A.3 Setup

The environment for TUNA requires two different types of nodes: an Orchestrator and a set of Workers. These two different types of nodes have slightly different setup instructions, found below. All scripts are found in `src/processing`. One script that may be useful is

`add_hosts.sh <hosts> <port>`

as a way to add all of the hosts listed in the specified host file to the trusted hosts list. This is required for `pssh` which we use to deploy our scripts. One point of note is that you cannot have your username in the host file when using this script. These scripts should work on any platform, however, we have only tested this on Azure, and CloudLab. We provide experiment files for c220g5 nodes on CloudLab, and D8s_v5 nodes in Azure. These instance types will become important later, however users can provide their own files in `/src/spaces`. If you are trying to replicate the work found in our paper, we recommend using 10 worker nodes and 1 orchestrator node, where the 10 worker nodes are the first 10 nodes that were created. This will allow you to use our provided hosts files (`hosts.azure` or `hosts.cloudlab`). Alternatively, a custom host file can be used.

A.3.1 Workers. To install and copy our files, there are two commands we will need to run.

`./worker_setup_remote.sh <hosts>`

The first command will install all of the dependencies, as well as set up the environment.

`./worker_deployment.sh <hosts> <node_type>`

The second command will start all of the required processes. Note that the second command will say some of the commands fail. This is expected, as they simply ensure that any previous instances of stopped and deleted before beginning the initialization process.

At some point during running this command, there will be a required interaction to specify that the docker image that is building in the background has been completed. There are two options here. First, you can connect to one of the workers and run `sudo tmux a -t install`, and sure that the pane has completed all of its commands. Alternatively, you can wait around 20 minutes, and this will most likely be long enough for the image to complete building.

A.3.2 Orchestrator. Building the orchestrator requires slightly more interaction from the user.

`bash orchestrator_deploy.sh <orchestrator_host> 22`

First, like before we will set up the environment using a deployment script. This will, again, automate the file transfer and environment setup.

Next, connect to your orchestrator node and run the following commands. Note, that the first command is `tmux`. We recommend using this as tuning runs are long-running. Without `tmux`, disconnects over `ssh` are possible, however this is not technically required.

```
tmux
make -C src/MLOS conda-env
conda activate mlos
```

A.4 Evaluation workflow

A.4.1 Major Claims. *We list the major claims and corresponding experiments below:*

- *(C1)*: sampling variability significantly decreases the average rate of convergence. This is proven by the experiment (E1) described in 3.1.
- *(C2)*: TUNA can generalize across various workloads and achieve better performance, lower variability, or both compared to traditional sampling and the default configuration. This is proven by the experiment (E2) shown in Figure 11.
- *(C3)*: TUNA can generalize across various sets of execution environments and achieve better performance, lower variability, or both compared to traditional sampling and default configuration. This is proven by the experiment (E3) shown in Figure 13 and Figure 12.
- *(C4)*: TUNA can generalize across various systems and achieve better performance, lower variability, or both compared to traditional sampling and default configuration. This is proven by the experiment (E4) shown in Figure 14 and Figure 15.
- *(C5)*: TUNA can generalize across different optimizers and achieve better performance, lower variability, or both compared to traditional sampling and default configuration. This is proven by the experiment (E5) shown in Figure 18.

A.4.2 Experiments.

Experiment (E1): [5 human-minutes + 800 compute hours]: This experiment compares the rate of convergence for tuning epinions with various levels of synthetic sampling noise.
[Preparation] Set up the environment as described above.
[Execution] Run the `parallel_prior.py` *command described in the* `README` *for three levels of noise:* 0%, 5% *and* 10%.

[Results] The results should show that as the noise levels increase, the rate of convergence decreases.

Experiment (E2): [20 human-minutes + 4000 compute hours]: This experiment compares the average performance and average standard deviation of various configurations found for various workloads on PostgreSQL .
[Preparation] Set up the environment as described above.
[Execution] Run the `parallel.py` *and* `python3 TUNA.py` *command as described in the* `README` . This will collect the results into data frames. These results can then be aggregated, and deployment can be run on, ideally, a new cluster using `mass_reruns_v2.py`
[Results] The results should show TUNA improves the target performance metrics, decreases variability, or both.

Experiment (E3): [20 human-minutes + 2000 compute hours]: This experiment compares the average performance and average standard deviation of various configurations found for TPC-C on PostgreSQL running on CloudLab Nodes and in a new Azure region.
[Preparation] Set up the environment as per the above instructions in both a new Azure region and CloudLab.
[Execution] Run the `parallel.py` *and* `python3 TUNA.py` *command as described in the* `README` in both environments. This will collect the results into data frames. These results can then be aggregated, and deployment can be run on, ideally, a new cluster using `mass_reruns_v2.py`
[Results] The results should show TUNA improves the target performance metrics and decreases variability on CloudLab Nodes and in a new Azure region.

Experiment (E4): [20 human-minutes + 2000 compute hours]: This experiment compares the average performance and average standard deviation of various selected configurations found Redis and NGINX .
[Preparation] Set up the environment as described above.
[Execution] Run the `parallel.py` *and* `python3 TUNA.py` *command as described in the* `README` for both systems. This will collect the results into data frames. These results can then be aggregated, and deployment can be run on, ideally, a new cluster using `mass_reruns_v2.py`
[Results] The results should show TUNA improves the target performance metrics and decreases variability on CloudLab Nodes and in a new Azure region.

Experiment (E5): [20 human-minutes + 1000 compute hours]: This experiment compares the average performance and average standard deviation of various configurations found for various workloads on PostgreSQL using a Gaussian process optimizer.
[Preparation] Set up the environment as described above.
[Execution] Run the `paralle_gp.py` *and* `python3 TUNA_gp.py` *command as described in the* `README` . This will collect the results into data frames. These results can then be aggregated, and deployment can be run on, ideally, a new cluster using `mass_reruns_v2.py`
[Results] The results should show TUNA improves the target performance metrics, decreases variability, or both using a Gaussian Process optimizer.

A.5 Notes on Reusability

If you want to further extend these to new systems, they simply need to be added to the version of Nautilus included in the artifact. A good place to start is to compare the Redis implementation with the PostgreSQL implementation to understand how different systems interact with Nautilus, and how the new target system could be added. In the future we intend to integrate TUNA to MLOS [5] as well.

[5] https://github.com/microsoft/MLOS/issues/926