OPPerTune: Post-Deployment Configuration Tuning of Services Made Easy

Gagan Somashekar* 1  Karan Tandan* 2  Anush Kini2  Chieh-Chun Chang4  Petr Husak4
Ranjita Bhagwan+ 5  Mayukh Das3  Anshul Gandhi1  Nagarajan Natarajan2

1Stony Brook University  2Microsoft Research  3Microsoft365 Research  4Microsoft  5Google

Abstract

Real-world application deployments have hundreds of interdependent configuration parameters, many of which significantly influence performance and efficiency. With today’s complex and dynamic services, operators need to continuously monitor and set the right configuration values (configuration tuning) well after a service is widely deployed. This is challenging since experimenting with different configurations post-deployment may reduce application performance or cause disruptions. While state-of-the-art ML approaches do help to automate configuration tuning, they do not fully address the multiple challenges in end-to-end configuration tuning of deployed applications.

This paper presents OPPerTune, a service that enables configuration tuning of applications in deployment at Microsoft. OPPerTune reduces application interruptions while maximizing the performance of deployed applications as and when the workload or the underlying infrastructure changes. It automates three essential processes that facilitate post-deployment configuration tuning: (a) determining which configurations to tune, (b) automatically managing the scope at which to tune the configurations, and (c) using a novel reinforcement learning algorithm to simultaneously and quickly tune numerical and categorical configurations, thereby keeping the overhead of configuration tuning low. We deploy OPPerTune on two enterprise applications in Microsoft Azure’s clusters. Our experiments show that OPPerTune reduces the end-to-end P95 latency of microservice applications by more than 50% over expert configuration choices made ahead of deployment. The code and datasets used are made available at https://aka.ms/OPPerTune.

1 Introduction

The performance and efficiency of large services and application deployments depend heavily on how they are configured. Configurations can be system-level, such as the read_ahead_kb parameter, which decides how much extra data to read from disk during I/O in Linux, and resources.limits.cpu that limits the amount of CPU a Kubernetes container uses. They can also be application-level, such as maxmemory, the memory usage limit at which Redis starts evicting keys. Any large application deployment invariably includes hundreds, if not thousands, of such configuration parameters at multiple layers and components [25, 35, 48, 50, 66, 75].

Today, application operators determine the configuration values using domain-knowledge and canary testing on relatively small deployments before widely deploying the application. However, application behavior can change considerably with time and therefore the configuration values set before deployment may not work well in the longer term; e.g., developers continuously add features, the user population and their usage behavior varies [47], the underlying hardware hosting the applications also can also change [67, 75]. Consequently, to squeeze the most performance—say throughput or latency—or to make it run efficiently on as small a set of resources as possible without compromising performance, operators need to constantly monitor and modify these configuration parameters well after they have deployed the application.

Manually exploring and changing the configurations at regular time intervals can be tedious and risky, given that the number of parameters is large and, more often than not, the values of parameters can depend on each other and the deployment environment. Several recent efforts have proposed using machine learning (ML) based techniques [15, 25, 27, 40, 44, 46, 48, 61, 68, 69] to automate the process of configuration tuning. These efforts use online learning or reinforcement learning (RL) to set the configurations, observe the application state to determine how well it is doing, and then iteratively refine the configuration based on the observed states. This approach does reduce the burden on the operator, and yet, the problem is far from solved. The algorithm is only one necessary component of post-deployment configuration tuning. Through our experience deploying popular techniques (e.g., Bayesian Optimization (BO)) and state-of-the-art frameworks (e.g., SelfTune [40], Kubernetes Autoscaler [3]) for parameter tuning in application deployments at Microsoft, we have observed that there are significant gaps to be bridged in the end-to-end process of configuration tuning. Some of these gaps, discussed below, may have been overlooked by prior research, as they are more pronounced in actual production deployments.

*These authors contributed equally.
+Work done while employed at Microsoft Research.
First, since services can easily have hundreds of configuration parameters, it would be prohibitively expensive to automatically tune all of them simultaneously. Thus, an automated approach should determine which components or layers of the service to tune, and for each component or layer, which configuration parameters it should tune.

Second, when a service is running, all configuration parameters are not equally easy to tune. For instance, changing `worker_process` of Nginx (this sets the number of Nginx worker processes) requires only a service reload after changing a configuration file [2] with no downtime, but changing `wiredTigerCollectionBlockCompressor` in MongoDB requires a pod restart which leads to a downtime of close to 8 seconds when deployed with the recreate strategy on Kubernetes [6]. Changing `isolcups`, the parameter that isolates CPUs from the kernel scheduler in Linux, will require an entire system reboot [63]. Previously proposed algorithms do not consider this varying difficulty of tuning different types of parameters. Importantly, to reduce potential disruptions for deployed services, the tuning approach should use a very small number of iterations to converge on the right values.

Third, the tuning system needs to determine the right granularity for each tuning instance. It could tune a single set of configuration values for the entire service, or it could tune different values for each geography, or perhaps for every machine type. We refer to this as determining the right tuning scope. Currently, the operator has to scope the tuning instances manually irrespective of the tuning algorithm used.

Finally, standard algorithms for tuning work only on numerical [30, 40] or only on categorical [19, 20, 38, 65] parameters; real-world services will almost always have a combination of categorical and numerical parameters. For instance, Redis’s `maxmemory` and `maxmemory-policy` (sets the eviction policy) are related parameters that are of type numeric and categorical, respectively. The key difference between handling numerical and categorical parameters is the notion of continuity. Categorical parameters lack continuity so the performance can change drastically every time the value of the parameter is changed. There is no easy way to encode/decode categorical as numerals as the usual approaches like 1-hot encoding [34] would increase the search space substantially. Popular techniques like BO [16, 17] do handle such hybrid parameter spaces, but they are not meant for scenarios where the environment of the service changes continually. Deep neural models for configuration tuning [45, 57], on the other hand, are usually trained offline rather than in deployment.

We have designed, developed, and deployed OPPerTune (Online Post-deployment Performance Tuner), a configuration tuning service that addresses all the above challenges. Given a superset of configuration parameters\(^1\) that can be potentially tuned, OPPerTune can automatically create, manage, and scope tuning instances for application operators of large-scale services. The key contributions we make are:

1. OPPerTune introduces a novel tuning algorithm, as part of its backend, that can tune categorical and numerical parameters simultaneously within the same instance.
2. OPPerTune uses a novel decision-tree based algorithm to automatically determine the right scope of tuning instances, taking into account the varying application context in production environments, such as workload types and volumes.
3. We have built and deployed OPPerTune as a cloud service at Microsoft. First-party applications that are already in production can be on-boarded to our service with minimal engineering overhead.

We have evaluated OPPerTune on two applications: (i) the social networking application from the DeathStarBench benchmark suite driven by workload traces collected from a large-scale service that is part of Microsoft Teams application, and (ii) a large-scale ML experimentation pipeline that uses Azure ML for model development and Apache Spark for data processing. Our evaluation and comparison with closely related works [17, 40, 45, 58, 65] show that by using OPPerTune to tune configuration parameters: (i) the tail latency of the application consisting of tens of microservices reduces by more than 50% while tuning in deployment, even under significant workload variations, compared to carefully-chosen pre-deployment configurations; (ii) service disruptions that may occur due to configuration tuning are reduced by nearly 30%; and (iii) workload completion times drop by 10%–50% on two Azure ML clusters, over two weeks of deployment.

\(^1\)This work focuses on application- and infrastructure-layer parameters.
OPPerTune architecture: Figure 1 presents the high-level architecture of the OPPPerTune service. The basic unit of the service is a tuning instance that consists of (a) configuration parameters, their data types, enumerations of possible values/ranges for categorical/numerical configurations; and (b) a tuning algorithm for updating the instance. Applications can create one or more tuning instances to tune configuration parameters across various layers of the application stack, based on its requirements (as shown for ‘App 2’ in the figure). Alternatively, OPPPerTune provides an automatic scoping component (autoscooper) to help applications create, manage, and scope the tuning instances in deployment based on dynamic context information they provide (as shown for ‘App 1’). OPPPerTune can also aid applications to pre-determine (using an offline step) which configuration parameters to tune in deployment via the selector module.

Creating, fetching, and updating tuning instances: An application intending to use OPPPerTune makes an API call to create a tuning instance along with a list of parameters and their meta-data (e.g., range of parameters). This step can be automated, as in prior works [62, 72], or can involve the developer to convey a super set of parameters along with their meta-data. The operators can use their domain-knowledge to select a subset of parameters to pass to OPPPerTune, but this is optional. The parameter list can be arbitrarily large, and OPPPerTune’s selector module will pick the performance-critical parameters as discussed later in this section. OPPPerTune persists tuning instances on a database for the application to access at any point in time, possibly from multiple machines. For each tunable configuration parameter, the operator can optionally supply the cost (e.g., is a system restart required) associated with changing it. When such cost information is available, OPPPerTune uses it to decide how often to tune the configuration parameters. OPPPerTune implements the standard fetch and update client-API paradigm of existing work [14, 40] for online tuning. The application invokes (a) Predict to fetch the recommended configuration values from a tuning instance, and (b) SetReward, at some point in time after (one or more) Predict calls, with a reward value. The SetReward call updates the associated tuning instance, as per the tuning algorithm it uses. Any delay in sending back the rewards will only delay the model update/convergence but not affect the application performance.

OPPerTune supports parallel exploration when enough servers/resources are available to deploy multiple instances of the same application. If the servers in the cluster are identical, a single OPPPerTune instance can be employed to which all servers issue Predict and SetReward calls. OPPPerTune would use all deployments to explore and converge towards a single optimal configuration for the application. For heterogeneous servers, AutoScope (Section 4) can enlist contextual attributes (e.g., region, hardware type) to cluster the servers, and tune parameters for different deployments.

AutoScope: Applications may have different performance characteristics on machines with different CPUs or memory sizes, and hence may consider using a different configuration tuning instance for each machine type. Similarly, applications could behave differently for light versus heavy workloads, and for different API call types. For instance, if the application runs independently on the cluster machines with varying hardware and workloads, then it could create one tuning instance per machine (as is done in [40] for tuning configuration parameters of a workload scheduler). Thus, tuning instances for the application could be “scoped” along (at least) three dimensions: infrastructure (e.g., machine type), functionality (e.g., API call), and workload (e.g., requests per second). Currently, determining the “right” scope for tuning instances is usually done, if at all, by domain experts [40], and periodically revisited. As an alternative, in Section 4, we present AutoScope, an automatic, efficient, and interpretable way of scoping tuning instances. A Predict call to an AutoScope instance returns configuration values corresponding to the context presented.

Configuration selector: For applications that have several hundreds or thousands of configuration parameters to tune across various layers of the application stack, OPPPerTune employs a selector module to pick the most promising configuration parameters to tune. The selector module uses a simple and effective microbenchmarking strategy to identify such configurations, as discussed in Section 5. The module provides a list of parameters, ordered in decreasing order of their importance, along with a score that quantifies the importance.
The operator can then choose the top-\(n\) parameters from this list, based on their importance score.

**Rounds and Sample complexity:** Online tuning algorithms, implemented in OPPerTune’s back-end, iteratively tune the configuration parameters. Each iteration is called a *round*. At each round, the tuning algorithm (i) determines the next set of parameter values for the application, (ii) observes a reward computed by the application over a predetermined period (1 hour, 24 hours, etc.), and (iii) updates its “policy” (which prescribes how to choose parameters) based on the reward. Changes made by the algorithm to configuration values can cause disruptions, e.g., may necessitate application restarts or even cause downtime. Thus, a desired property of a tuning service is that it require only a few rounds to learn suitable configuration values. This quantity, proportional to the number of rewards measured, is called *sample complexity*. OPPerTune achieves low sample complexity for tuning in real deployments, as we demonstrate in Section 7, using multiple techniques including a novel tuning algorithm (Section 3), automatic scoping, and microbenchmarking.

### 3 Configuration Tuning in Hybrid Spaces

In this section, we present a novel algorithm for the post-deployment configuration tuning problem (Section 2) on a hybrid space wherein the configuration parameters are a mix of numerical and categorical ones. We consider the basic setting when OPPerTune has no additional knowledge (“context”) about the system being tuned; the setting where some context may be available is considered in the next section.

#### 3.1 Problem Definition and Terminology

We pose the problem of configuration tuning for an application post-deployment as that of online learning with bandit feedback. That is, we want to tune iteratively, only interfacing with the application for setting new parameter values (e.g., # CPU cores or memory size for containers), and for obtaining feedback in response to the set parameter values, as an observed *reward* value that is to be optimized (e.g., latency or throughput of the application).

A key aspect of bandit formulation is the *explore-exploit tradeoff*. We want to exploit the “best” parameters as per policy learned so far to ensure that the application is functioning well without disruption; at the same time, we want to explore potential parameter choices that might yield better rewards. This tradeoff is especially important in practical scenarios where the reward function itself changes with time—the same parameter choices could have different effects on the service at different time points. For instance, diurnal workload fluctuations can induce very different reward values for the same setting of memory requirements for a container, depending on how and when the reward is computed, e.g., hourly P95 latency can vary significantly between peak and off-peak hours.

Bandit learning techniques that can handle time-varying rewards, therefore, are more appropriate to our problem than popular alternatives such Bayesian Optimization (BO) [16, 17], heuristic search and global optimization [52], and genetic algorithms [32]. For instance, BO needs to evaluate the *same* reward function at multiple parameter values by design. This is infeasible for post-deployment tuning because we cannot evaluate a deployed application multiple times. Most systems research that leverage BO typically use it in offline scenarios (i.e., pre-deployment tuning in controlled settings) [17, 22], in contrast to our post-deployment tuning scenario.

#### 3.2 Hybrid Configuration Space

In practical scenarios, the space of configuration parameters can be complex: (i) it can be very large; if there are \(n\) parameters to tune, with, say, \(s\) possible values each, we have \(s^n\) choices, and (ii) they may be discrete (e.g., number of CPU cores), real-valued (e.g., CPU utilization threshold), or categorical (e.g., cache eviction policy). Some state-of-the-art techniques for bandit learning/RL work for categorical spaces [19, 20, 38, 65] or numerical spaces [30, 40], but not both. Others have high sample complexity for tuning in deployment [42, 57]. To address this gap, we design a novel learning algorithm to handle hybrid configurations efficiently.

Formally, the “hybrid configuration space” comprises: (a) categorical space \(C\) over \(k\) categorical parameters, and (b) numerical space \(\mathbb{V}\) over \(m\) numerical parameters which is a bounded subspace of \(\mathbb{R}^m\). In our formulation, we treat discrete parameters as numerical rather than categorical to exploit the fact that they are ordered spaces.

#### 3.3 Proposed Algorithm: HybridBandits

Our configuration tuning HybridBandits algorithm is presented in Algorithm 1. It leverages two simple but key ideas. At each round, (1) it maintains different types of policies for sampling categorical and numerical actions; in particular: (i) \(\varepsilon\)-greedy policy for the categorical configuration space, standard in multi-arm bandit algorithms, where with probability \(\varepsilon\) a random arm is explored, and with probability \(1 - \varepsilon\), high-reward arms are exploited; and (ii) a “perturbation” policy for numerical configurations, where the algorithm samples numerical configurations from an \(\varepsilon\)-radius ball centered around the “current best” configuration vector, and (2) it uses a single reward that the system provides as feedback to update both the policies simultaneously. In particular, it applies sample-efficient gradient-descent update [30, 40] for the numerical parameters, and the exponential weights update [41, Chap. 11] for the categorical parameters.

**Algorithm description:** Algorithm 1 maintains a multinomial distribution \(p(t)\) over categorical actions \(C\), i.e., there is a probability associated with each possible \(k\)-tuple of categorical parameter choices at every round \(t\). For the numerical actions, it maintains a vector \(\mathbf{w}(t) \in \mathbb{R}^m\).

**Initialization:** The weights for the numerical parameters \(w\) are initialized to default choices that the application provides. The multinomial \(p\) is initialized to the uniform distribution.
Algorithm 1 HybridBandits: Post-Deployment Configuration Tuning for Hybrid Spaces

1: **Input:** exploration parameter \( \varepsilon \in (0, 1) \), learning rate \( \eta > 0 \), space \( \mathcal{C} \) of \( k \) categorical parameters, space \( \mathcal{W} \) of \( m \) numerical parameters.
2: **Initialize:** categorical space weights \( p_i(0) = 1/|\mathcal{C}| \), for \( 1 \leq i \leq |\mathcal{C}| \) \( \text{// uniform distribution} \), and numerical parameters \( w(0) \in \mathcal{W} \) \( \text{// default} \)
3: for \( t = 0, 1, 2, \ldots \) do
4:   Let \( \tilde{p}_i := (1-\varepsilon)p_i^{(t)} + \varepsilon \frac{1}{\mathcal{C}} \) // Define explore-exploit multinomial distribution over the categorical space
5:   Sample categorical and numerical actions to deploy
6:   Sample \( c \sim \tilde{p} \) from the multinomial and let \( \alpha_c^{(t)} \) be the corresponding \( k \)-tuple of categorical parameters
7:   Sample numerical parameters from a ball centered at \( \tilde{w}(t) \), radius \( \varepsilon \); i.e., \( \tilde{w}(t) := w(t) + \varepsilon u \), where \( u \in \mathbb{R}^m \) is sampled from \( \{ u : \| u \|_2 = 1 \} \) // Identical to Bluefin [40]
8:   Deploy actions and receive reward
9:   Update numerical parameters center: \( w^{(t+1)} = w(t) + \frac{1}{\tilde{p}} \cdot \eta \cdot r(t) \cdot u \), where \( u \) is the sample obtained in Step 6.
10: Define scaled reward: \( \tilde{r}(t) := r(t)/\tilde{p} \), where \( c \) is the sample obtained in Step 5
11: Update categorical distribution: \( p_c^{(t+1)} = p_c^{(t)} e^{\eta \cdot \tilde{r}(t)} \), and for \( i \neq c \), \( p_i^{(t+1)} = p_i^{(t)} \), Renormalize \( p^{(t+1)} \) to sum to 1

### i.e., \( p_i = \frac{1}{|\mathcal{C}|} \) for \( i \in \mathcal{C} \). At each round, the algorithm performs:

**Sampling actions (Steps 5-6):** For the categorical actions, following the standard exponential weights algorithm (EXP3, [41, Chap. 11]), it samples a \( k \)-tuple from the distribution \( \mathbf{p} \) (exploit) with probability \( 1-\varepsilon \), and from the uniform distribution (explore) with probability \( \varepsilon \). For the numerical actions, it samples a \( m \)-dimensional vector from a ball centered at the current \( \mathbf{w} \), with radius \( \varepsilon \).

**Deploy actions and receive reward (Steps 7-8):** The sampled numerical (scaled appropriately) and categorical configurations are then deployed in the application, and (after a certain amount of time) the algorithm receives a reward value from the application (implementation details in Section 7).

**Update policies (Steps 9-11):** For the numerical parameter weights, the algorithm follows the gradient estimation scheme studied in the optimization literature [30], as well as applied in the context of online system parameter tuning [40]. For the categorical parameters, it: (a) computes an unbiased estimate of the reward for the sampled choices, and (b) scales the probability of the sampled choices using a factor that is exponential in the reward estimate.

The algorithm has hyperparameters \( \varepsilon \) and \( \eta \), and we set these to default values in all of our experiments (following SelfTune [40]). In practice, \( \mathcal{C} \) can be very large: the microbenchmarking strategy (Section 5) can be used to restrict \( \mathcal{C} \) to the most impactful categorical parameters, and to ensure that the algorithm has low sample complexity. We conjecture that Algorithm HybridBandits has convergence guarantees for certain classes of reward functions (for instance, if the reward functions \( r_i \) are convex, for any fixed combination of the categorical parameters in \( \mathcal{C} \)). Empirically (in Section 7, and in our synthetic problem setup in Appendix B), we find that the algorithm convergences well in practice; obtaining a formal proof of convergence is an exciting open problem. We also note that when the configuration space contains only numerical parameters, HybridBandits is the same as SelfTune’s Bluefin algorithm.

## 4 Automatic Scoping of Tuning Instances

Consider an operator who wants to tune the parameters of a distributed application that is I/O-bound. There are two extreme options available to the operator in terms of how they can set up tuning instances on the OPPerTune server (Figure 1): (1) set up one “global” instance to tune all the application parameters across all machines/workloads, that, say, uses HybridBandits presented in Section 3 for tuning; or (2) set up multiple “local” instances based on the domain expertise that the workloads are I/O-bound; e.g., one instance per disk type or one instance per spindle speed, where each instance independently tunes parameters using HybridBandits.

The latter option is more appealing as the application performance, and therefore the optimal parameter choices will likely vary with the disk type the workloads are accessing.

In this section, we consider the setting when OPPerTune is provided some context of the application (i.e., disk type and spindle speed in the above example) being tuned at every round. OPPerTune can exploit the observed context to simultaneously do scoping and configuration tuning.

### 4.1 Joint Scoping and Configuration Tuning

To perform joint scoping and configuration tuning, at each round, along with the reward, the application must provide additional context information such as machine type, disk type, spindle speed, workload volume, etc. Using this additional context, OPPerTune determines a lightweight and interpretable scoping policy that the operators can understand.

For instance, given job type `jobtype` and requests per second (rps) as context, and `numcores` and `mem` as the configuration
values to tune, it learns rules of the form \(\text{if } (\text{jobtype} = \text{"cpu_bound"}) \text{ and } (\text{rps} > 1000) \text{ then } \text{numcores}=16, \text{mem}=2G \text{ else } \text{numcores}=4, \text{mem}=2G.\)

These kinds of scoping rules can be captured by decision tree models illustrated in Figure 2(d). Each root-to-leaf path in the tree constitutes a scope, and each leaf maintains a tuning instance for the scope. In the Figure, ‘Scope 7’ is interpreted as application running in Cluster A, when its workload volume \((\text{RPS}) > 1000, \text{involving jobs of type 1 or 2.}\) Its leaf node maintains a tuning instance, i.e., values for the two parameters numcores and mem. These values will be returned for Predict requests satisfying this scope, and will be updated when a reward arrives for these requests from the application.

4.2 Proposed Algorithm: AutoScope

Learning decision trees in the bandit setting is a challenging problem, and popular tree learning algorithms do not apply (see Section 8). We extend a state-of-the-art technique proposed in [39] (for trees with only one parameter in leaf nodes) to our general setting where each leaf node is a tuning instance with several (hybrid) parameters.

We start by giving the key intuitions, before giving a detailed technical description of the tree learning algorithm.

Algorithm outline: AutoScope maintains a binary decision tree \(f_T\) of max specified height \(h\) (\(h = 3\) suffices for scenarios evaluated in Section 7), as illustrated in Figure 2. At first, the tree \(f_T^{(0)}\) effectively behaves like a single tuning instance, initialized identical to Algorithm 1, i.e., all the leaf instances \(I_0^\ell\) in Figure 2(a), for \(0 \leq k \leq 7\), are initialized identically. At round \(t\), the algorithm observes a context vector, denoted as \(c_t\). When the current tree model \(f_T^{(t)}(c_t)\) is applied to \(c_t\), it will land in a unique leaf node containing a tuning instance. That is, the context vector is first applied to a linear model in the root node (whose weights are initialized to default value in the beginning). Depending on the sign of the resulting value, \(c_t\) traverses left or right, and continues in this fashion making branching decisions at every intermediate node until it reaches a leaf. The root-to-leaf path \(c_t\) traverses is its “current scope”, and AutoScope will invoke the leaf’s tuning instance. Figure 2(b) shows the attributes \((\text{RPS} > 500, \text{cluster} = A, \text{Job} = 3)\) getting resolved to the scope \(6 (l_6^\ell)\) in the first round.

This amounts to doing one round of Algorithm 1 on the leaf’s tuning instance, thereupon updating it. Now the technical challenge is updating the tree model \(f_T^{(t)}\) parameters (i.e., the internal node weights for making branching or scoping decisions), besides the configuration parameter values in each of the leaf nodes. We describe the algorithm in detail next.

Algorithm description: We present the procedure for jointly learning the scoping and the tuning instances, i.e., parameters corresponding to each scope, formally in Algorithm 2 in Appendix A. The algorithm maintains a decision tree model denoted \(f_T\); each internal node \(j\) in the tree has a linear model \(z_j \in \mathbb{R}^d\) (where \(d\) is the number of context variables) that makes routing decisions of the form \((c^{(t)}, z_j) > 0\).

Each root-to-leaf path in the tree corresponds to a scope. Each scope ends at a leaf node \(\ell\) which holds a tuning instance, i.e., \(p_\ell \in [0, 1]^{[C]}\) (for categorical actions) and \(w_\ell \in \mathbb{R}^m\) (for numerical actions). We work with binary trees of height \(h\) with \(2^h\) scopes (and tuning instances).

Learning a tree policy entails learning the internal node parameters \(z_j\), for \(0 < j < 2^h - 2\) and the leaf instances \(p_\ell, w_\ell\), for \(0 \leq \ell \leq 2^h - 1\). Widely-used tree learning algorithms like CART [23], C4.5 [56], and their variants do not apply to the bandit feedback setting because they need access to labeled training data (which in our scenario means optimal parameters for different context vectors, which is difficult to obtain in practice). There has been recent work on learning trees using bandit feedback [28, 29], but they work only for categorical spaces. The closest to our setting is the technique proposed in [39], but it handles only a single parameter (either numerical or categorical) in the leaf nodes. We extend their technique to handle multi-dimensional, hybrid tuning instances in the leaf nodes.

Algorithm 2 follows the structure of Algorithm 1 closely. Each leaf node is initialized (Step 2) identical to Algorithm 1, Step 2. The first key difference is that, at each round \(t\), the observed context \(c_t\) determines which tuning instance, i.e., leaf node is selected (Steps 4–5). Once the leaf node is selected, in Step 6, we use the corresponding tuning instance to obtain the configuration and deploy the system with this configuration (Step 7) to observe the reward just as in Algorithm 1. The second key difference is how the tree is updated (Steps 8–9), as discussed below.

The main challenge in learning decision trees in general, not just in the bandit setting, is that the tree function \(f_T : c_t \mapsto \ell\) is highly discontinuous and non-differentiable. If
we can approximate $f_T$ with a differentiable function, then
we will be able to jointly learn $f_T$ model as well as the leaf
node parameters using online gradient descent techniques in
a sample-efficient manner. We leverage the relaxation in [39]
that, in particular, replaces discontinuous branching decisions
(“go left or right”) at the internal nodes with differentiable
sigmoid functions (“go left with probability 0.9 and right with
probability 0.1”).

Using chain rule of differentiation, we can show that the
gradient of the reward function $r_i$ with respect to the tree
model parameters (both internal node and leaf) can be written
as a tensor product of the gradient of $r_T$ with respect to the
leaf node parameters and the gradient of $f_T$ with respect to
the tree model parameters.

**Scalability:** AutoScope scales well with arbitrary state
spaces. Its model size scales linearly in the size of context
vector $c$ and the tree’s height is typically very small. Though
there are far too many ways of jointly slicing the context
dimensions, AutoScope tries to automatically find scopes that
are most beneficial in improving the reward metric without
other hints.

5 **Configuration Selection**

For applications that have several hundreds or thousands of
configuration parameters to tune across various layers of the
application stack, OPPerTune employs a configuration selec-
tor module (Figure 1) that leverages a simple and effective
microbenchmarking strategy to identify the most promising
configuration parameters. While the techniques outlined here
are heuristic, they are inspired by optimization theory [51,53].

The role of the selector module is two-fold: (a) it prunes
the size of the configuration space, which in turn helps reduce
the algorithm’s sample complexity; and (b) it helps minimize
the number of disruptions (e.g., container restarts) in the
application while tuning. If (b) is the only goal, selecting just
the configuration parameters that do not require restarts, may
suffice. However, that might compromise on application per-
formance by ignoring configurations that could significantly
impact the performance (as illustrated in Section 7.3).

OPPerTune uses a microbenchmarking strategy to assess
the effect of changing each configuration parameter on the
application’s performance (i.e., the reward value), while keeping
the others fixed. Let us consider numerical configurations
for the moment. The strategy is inspired by how coordinate
descent algorithms [51], which are rigorously studied in the
optimization community, work. These algorithms pick one
coordinate (i.e., configuration parameter) at a time and com-
pute the gradient of the reward function with respect to only
that parameter. They iteratively pick coordinates (cyclically
or randomly) to optimize the reward function.

We do not know of any variants of these algorithms that
provably work in our online bandit setting. But, we find that
the basic idea is empirically effective for the goal of selecting
candidate parameters to tune. We use the same gradient esti-
mation technique employed in Step 9 of Algorithm 1 for each
configuration parameter while holding all other parameters
fixed (to the default choices, for example). OPPerTune ac-
complishes this by simply creating microbenchmarks in-
stances, each with just one configuration parameter, and per-
forming one round of the HybridBandits algorithm. The mag-
nitude of the (scalar) gradients computed at the instances tells
us the impact of each configuration parameter. In practice,
this idea can also be extended to categorical spaces—perform
one round of the algorithm on each categorical parameter, and
compute the magnitude of change in reward for a randomly
chosen value vs. the default value for the parameter.

The configuration selector module then picks top-\(n\) param-
eters, sorted by decreasing magnitudes of these “gradients”
where \(n\) is customizable by the application. This greedy se-
lection strategy, i.e., picking the coordinate (or parameter)
yielding maximum absolute gradient, has been shown to be
provably better than other heuristics for selecting coordinates,
for some classes of reward functions [53].

Microbenchmarking can be done in canary/test deploy-
ments of the application. The tuning instances for the applica-
tion can work with the selected configuration parameters in
deployment. OPPerTune provides a flag to periodically
revisit microbenchmarking and re-assess the top-\(n\) parameter
selections. Operators can turn this flag on when there are
long-term changes (e.g., hardware, new workloads).

6 **OPPerTune Implementation**

OPPerTune’s implementation has three major components:
the server, client, and the algorithm backend. We have imple-
mented the server in Go using Fiber [10], and the client in
Python (for ease of integration with applications which are
often written in Python). We have implemented our proposed
algorithms in Python, and have integrated the server with ex-
isting Python implementations of other algorithms. We now
describe each component in some detail.

1. **OPPerTune Server:** The server implements three key in-

faces for the applications (clients) submitting requests via
REST API calls: a) creation of tuning instances, b) fetching
the values from the instances, and c) updating the instances
using the the reward values sent back to the server. The server
persists configuration tuning instances (consisting of the list
of parameters to tune and their constraints, and the model
for tuning) in a database. Persisting instances enables re-
suming from the saved model state at a later point of time,
and freeing the memory taken up by instances that are not
needed. For each fetch call from the client, the server responds
with the configuration values along with a requestId. The
client is expected to pass the reward value along with the
associated requestId, for the server to be able to correctly
issue an update to the corresponding tuning instance. System
changes that are bursty and short-lived can potentially impact
the observed reward. OPPerTune mitigates this by eliciting
an optional reward measurement period from developers that
sustains such spikes (e.g., aggregation of the metric over a time period). Furthermore, our bandit algorithms (Section 3 and 4) accommodate such adversarial changes. We host the server on Microsoft Azure that provides persistent storage, high availability, and wide accessibility.

2. OPPeRTune Client: The client is a library which implements easy-to-use REST API calls; these calls provide abstraction over raw HTTP requests, and applications use them to create, fetch (i.e., Predict), or update (i.e., SetReward) instances at the OPPeRTune server. The library also manages mapping client requests to API endpoints, payload preparation, and error-checking. We provide an installable package of OPPeRTune client for applications to use.

3. Service Backend/Tuning Algorithms: The backend consists of implementations of various tuning algorithms, and autoscooper and configuration selector components. Any tuning algorithm is expected to implement Predict and SetReward interfaces. We have implemented (i) the proposed algorithms HybridBandits and AutoScope; (ii) state-of-the-art online parameter tuning algorithm Bluefin [30, 40] and deep reinforcement learning (RL) algorithm DDPG [45, 55, 57]; and (iii) with minimal effort, we have integrated the contextual bandits-based algorithm Slates [65], and BayesianOptimization from the popular Python libraries VowpalWabbit [12] and scikit-optimize [11], respectively. The challenge in using deep RL techniques such as DDPG in deployment, typically, is their prohibitive model sizes and sample complexity. They use context differently than AutoScope to learn policies that require large complex models (the notion of scoring is implicit in DDPG). Hence, we have implemented a custom version of DDPG with light-weight models, similar to AutoScope, to be of use in post-deployment tuning.

7 Evaluation
To evaluate OPPeRTune, we use a combination of microbenchmarking and real deployments of two applications. Our evaluation focuses on the following aspects: 1) How does application performance improve using OPPeRTune?, 2) How does OPPeRTune reduce the cost of tuning (e.g., system restarts)? 3) How effectively does automatic scoping accelerate the tuning process in real deployments by reducing sample complexity?, and 4) How scalable is our service implementation (OPPerTune server and backend algorithms)?

7.1 Evaluated Applications
To evaluate OPPeRTune, we use two applications, each serving certain classes of workloads.

7.1.1 Social Networking Application
We use the SocialNetworking application from the DeathStarBench [31] benchmarking suite which mimics a stack consisting of a gateway server (Nginx), database engine (MongoDB), caches (Redis), and application logic. The application creates a network of users, and supports API calls to create and read messages from the users’ home pages. We use wrk2 [9] to emulate two workloads: (a) constRPS: a mix of 90% GET (read timeline) and 10% POST (create posts) requests (this mix has been used in previous work [40]), with the requests generated at a constant rate (requests-per-second), and (b) AMStraces: real access traces collected from AMS, a large-scale asynchronous media-sharing service that is part of the Microsoft Teams application, on 3 production clusters over 4 weeks from Sep 14, 2022 to Oct 10, 2022 (see Appendix C.1 for details on the traces and experiments).

For this application, the performance metric of interest is P95 latency of requests submitted. This metric is critical to consumer-facing services [26, 31]. For the constRPS workload, we measure the P95 latency for each 10-minute period, and for AMStraces, we measure it for each hour. This is fed as the reward value to the OPPeRTune service.

Table 1 outlines the list of microservices that SocialNetworking uses. Here, “rightsizing” layer refers to configuration parameters in Kubernetes that are used to determine the compute and memory limits for containers running the microservices. For each of the microservices, we tune a mix of real-valued, discrete, and categorical parameters picked from prior works and product documents [1, 4, 5, 7, 8, 13, 40, 61]. We note that operators can use such prior knowledge to reduce the number of parameters. However, the design of OPPeRTune does not enforce this approach and the operators can pass arbitrarily large number of parameters. OPPeRTune can automatically select a subset of performance-critical parameters if needed, as discussed in Section 7.3. The optimal values of these parameters depend on the workload characteristics; e.g., to support the same P95 latency, the MongoDB microservice will require higher resource limits for a higher request volume, and a larger number of clients would require higher concurrency setting for Nginx, etc. Additionally, the cost of tuning these parameters is passed to OPPeRTune so it can decide how often to tune (as noted in Section 2). For example, some of these parameters require a container restart. OPPeRTune piggybacks on scheduled maintenance (once daily for AMS) to tune such parameters. We defer the evaluation of other types of tuning costs (e.g., performance degradation without impacting availability and revenue cost associated with changing resource parameters) to future

Table 1: SocialNetworking application configuration parameters (217 in total) used for Figures 4, 5, and Tables 3, 6.
research. We use the default values of all the parameters as the starting configuration for all the algorithms.

We deploy the SocialNetwork application on a cluster with 7 virtual machines (VMs) provisioned on Azure. Each VM hosts a copy of the application stack, with the component microservices running on individual containers on the same VM, managed by Kubernetes. Thus, by tuning the parameters in Table 1 appropriately, OPPerTune allocates each VM’s resources in the right proportions to the various microservices so as to minimize P95 latency.

We use separate, dedicated VMs (7 more) in the same cluster as Kubernetes master nodes, to generate the workloads. Each VM has Intel Xeon Platinum 8272CL processor (32 vCPUs), 64 GiB RAM and 250 GiB storage (large enough to support the entire application stack).

### 7.1.2 ML-Experimentation Pipeline

We deploy OPPerTune in one of Microsoft’s ML experimentation pipelines called MLExp (real name withheld) that uses Apache Spark to prepare data before training various ML models. This is an actual production deployment consisting of workloads and jobs. Each workload is an experiment, consisting of a collection of jobs arranged in a directed acyclic graph (DAG). A job loads, selects, filters, or processes data in different ways. MLExp supports 11 job types.

Table 2 shows the configuration parameters that MLExp developers manually set for each job before submitting the workload to the cluster. We use OPPerTune to tune the same set of parameters for each job, initialized to modest initialization values elicited from the developers for our and the baseline algorithms.

Developers wish to minimize average job completion time over all jobs that comprise a workload. Lower job completion times imply lower workload completion time, which further implies faster iterations of building ML models. When a workload completes, the platform provides the individual job completion times, which OPPerTune uses as the reward for updating tuning instances.

One could argue that setting the parameters in Table 2 to maximum values (such as setting spark.driver.memory to 25GB, spark.driver.cores to 4, etc.) would minimize average job completion time. However this does not work in practice because MLExp’s scheduler will enqueue this workload until all the specified resources are available, thereby increasing the completion time. Hence, OPPerTune attempts to find the right balance between increasing these values and decreasing the workload’s wait time in the MLExp scheduler’s queue.

We have integrated OPPerTune in two production compute clusters: Cluster1, with 120 nodes, 1800 cores, and 7.03 TB of memory, and Cluster2 with 100 nodes, 1500 cores and 5.86 TB of memory. Each of these clusters serves 24 workloads on average every day, submitted by the ML pipeline developers. Figure 3 shows the spread of completion times for the 11 job types in the two clusters over a period of 1 week before integrating with OPPerTune.

---

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>spark.driver.memory</td>
<td>4GB - 25GB</td>
</tr>
<tr>
<td>spark.driver.cores</td>
<td>1 - 4</td>
</tr>
<tr>
<td>spark.executor.memory</td>
<td>4GB - 24GB</td>
</tr>
<tr>
<td>spark.executor.cores</td>
<td>1 - 4</td>
</tr>
<tr>
<td>spark.executor.instances</td>
<td>24 - 384</td>
</tr>
<tr>
<td>spark.default.parallelism</td>
<td>100 - 10000</td>
</tr>
<tr>
<td>spark.sql.shuffle_partitions</td>
<td>100 - 10000</td>
</tr>
</tbody>
</table>

Table 2: Job configuration parameters used for Tables 4, 5.

---

![Figure 3: Apache Spark job completion times sorted by the P95 percentile in the third week of November 2022 in two production clusters of MLExp application.](image)

### 7.2 Improving Application Performance

**A) Effectiveness of HybridBandits:** We first look at how effective the proposed algorithm HybridBandits is for performance tuning of applications in deployment, and how it fares relative to various state-of-the-art tuning techniques that we implement as part of OPPerTune backend (as discussed in Section 6). For this evaluation, we use the SocialNetwork application, as it has a mix of real-valued, discrete, and categorical parameters (Table 1). For each rps (constRPS workload), we run each tuning algorithm for a maximum of 50 rounds. The results are presented in Figure 4. We report the mean and standard deviation of P95 latencies, for the converged configuration values, over 5 trials (10-minute windows each).

“Predeployment” in Figure 4 refers to the baseline performance against manually chosen configurations that optimized the performance for a 3500 rps workload (which is close to the peak capacity supported by our cluster), and keeping them fixed for the rest of the rps. “Kubernetes Autoscaler” refers to the Vertical Pod Autoscaler (VPA) [3] for determining the rightsizing parameters. VPA performs poorly in general, as rightsizing decisions are solely based on container utilization, and not P95 latency of the application. We also baseline against several existing approaches, i.e. Bayesian Optimization (BO)\(^2\), state-of-the-art RL techniques Slates and DDPG (we set episode length to 1 for the constRPS experiments, so DDPG is effectively standard contextual bandits [74]), and SelfTune’s Bluefin with HybridBandits.

---

\(^2\)Our implementation of BO differs from CherryPick [17] in the choice of acquisition function. We use GP-UCB instead of EL, motivated by the superior performance of GP-UCB as demonstrated by Hoffman et al. [36].
The benefits of using OPPerTune service post-deployment are clear: every algorithm, almost for every rps, finds better configuration choices to adapt to changing workload volumes. Second, almost for every rps, HybridBandits significantly outperforms BO, Slates, and DDPG, and achieves the highest P95 latency. For instance, at the peak workload of 3600 rps, HybridBandits achieves nearly a 2x reduction in P95 latency compared to the best configuration predicted by BO that has a large variance. Third, the utility of tuning categorical parameters together with the numerical parameters, using HybridBandits, is clear at high workloads, compared to Bluefin algorithm in the SelfTune framework [40] that supports only numerical parameters. In particular, HybridBandits (32.6ms) achieves about 15% reduction in P95 latency relative to Bluefin (38.6ms) for the 3600-rps workload. DDPG performs reasonably well in high workloads, despite the absence of informative context. We see similar results when the initial configuration to our algorithm is bad (i.e., yields very high latency), indicating HybridBandits’s ability to converge to near-optimal configurations quickly even if the configurations are poor in the initial few rounds.

**Takeaway 1.** OPPerTune with HybridBandits achieves the best performance, especially at peak workloads, among the state-of-the-art ML techniques used in systems performance optimization.

**B) Effectiveness of AutoScope:** We now evaluate the benefits of automatically scoping tuning instances using AutoScope, in terms of the application performance as well as sample complexity. We consider SocialNetwork with AMStraces and MLExp for this evaluation.

1. **SocialNetwork + AMStraces:** We compare AutoScope with a domain-expertise based scoping strategy, informed by the diurnal patterns of workloads in AMStraces. We use one tuning instance for every 2 consecutive hours in a cluster-day. Each tuning instance runs HybridBandits independently to learn suitable configuration parameters for its 2-hour scope. We refer to this as HybridBandits_{cluster, hour}. For AutoScope, we use average rps over every 2 consecutive hours in a day as context. We build a simple estimator for rps values using the first week traces, and use them for all the weeks. This is because we can not know the true rps values (in the future) at the time of Predict calls.

   We let all the methods use the first 3 weeks’ traces to tune the configuration parameters for SocialNetwork in deployment. We then evaluate the converged parameters on the last week’s trace. In Table 3, we present, for each technique and for each cluster, (a) P50, P95, and maximum value of hourly P95 latency, computed over the last week, i.e., over 168 hours, and (b) sample complexity of the technique (i.e., # rewards used while tuning). We compare AutoScope with (i) the baseline of using “Pre-deployment choices” of configuration parameters, (ii) domain-expertise based scoping HybridBandits_{cluster, hour}, (iii) deep-RL based DDPG that uses rps, and CPU and memory utilization of microservices and VMs as features (“states”) for implicit scoping, and (iv) AutoScope_{cluster} where a separate AutoScope instance is created for each cluster with rps as the scoping attribute. We see that max P95 latencies for Clusters 2 and 3 are in the order of seconds with the Pre-deployment choices, OPPerTune, using each of the three algorithms, significantly reduces the worst-case P95 latencies. Importantly, AutoScope achieves significantly better performance in general compared to HybridBandits_{cluster, hour}, and DDPG in Clusters 2 and 3 especially.

   Remarkably, AutoScope achieves this performance using one-third of samples (i.e., # rewards) as that of other techniques. AutoScope exploits the overlap of diurnal patterns and workload volumes (Appendix C.1) across clusters to improve overall performance, using as few as 8 tuning instances (a height-3 tree), compared to manual scoping using 36 (3 clusters × 12 time-windows in a day) tuning instances. Moreover, AutoScope_{cluster} (one AutoScope instance per cluster) performs similarly (except for max latencies) to AutoScope; a single instance of AutoScope can adapt to application dynamics across deployments when the presented context (here, workload volumes) adequately captures the dynamics.

   **Takeaway 2.** OPPerTune with AutoScope is able to significantly improve the application performance, using 3x fewer samples needed by manual scoping strategies.

2. **MLExp:** For this application, we have 3 types of context information available at the job submission time, namely, job type (11 possible values), dataset size (‘large’ or ‘small’), and cluster used (1 or 2). So, as a baseline, we use the following domain-expertise based scoping strategy. We create one tuning instance per combination, yielding $11 \times 2 \times 2 = 44$
instances. Each instance runs Bluefin independently to tune Apache Spark configuration parameters (Table 2) for the jobs in the scope. We refer to this as Bluefin\(_{\text{cluster, type, size}}\). AutoScope uses the 3 context values for scoring via height-3 trees, i.e., at most 8 tuning instances (in the leaves). We initialized all the instances using the default choices for job parameters that the developers provided.

We use workloads submitted to the 2 clusters in the Nov 20-Dec 03, 2022 period to tune job parameters using Bluefin\(_{\text{cluster, type, size}}\) (suffix indicates the manual scoping strategy) and AutoScope. We ensured that each of the 44 instances using Bluefin\(_{\text{cluster, type, size}}\) as well as the 8 instances of AutoScope get at least 5 reward values during the period (to make meaningful updates). We then evaluate the converged instances on the workloads submitted to the 2 clusters between Dec 04, 2022 and Dec 10, 2022. Whenever a developer submits an ML workload, a Predict call is made which decides the scope. Rewards (completion times) are sent back when workloads are complete.

The mean and standard deviation of the workload (i.e., experiment) completion times over one week for different techniques are presented in Table 3. We compare AutoScope with (i) Pre-deployment choices, (ii) Expert choices, which are job-specific configuration choices we elicited from MLExp developers; they have implemented hand-crafted heuristics (refined over several months) to improve the job completion times on the clusters, based on individual job characteristics such as the number of data records processed in the pipeline, type of the job, repartitioning costs, etc., and (iii) Bluefin\(_{\text{cluster, type, size}}\).

We see that Bluefin\(_{\text{cluster, type, size}}\) and AutoScope have reduced the mean workload completion times by more than 50% that of the Pre-deployment choices in Cluster 1; and by about 10% in Cluster 2. Also, they perform as well as the Expert choices in Cluster 1, and better (significance determined using standard \(t\)-test at \(p\) value of 0.05) in Cluster 2. For 8 out of 11 job types, AutoScope achieves up to 2× smaller P95 completion times than expert choices (see Appendix C.2).

A highlight of this deployment study is that AutoScope, using only 90 samples (rewards) for 8 instances, is able to achieve performance competitive to Bluefin\(_{\text{cluster, type, size}}\) that uses 407 samples for 44 instances. The manual clustering of Bluefin\(_{\text{cluster, type, size}}\) indeed is marginally better than

### Table 3: Comparison of various techniques for post-deployment configuration tuning of the SocialNetwork application using real workloads (AMStraces). Algorithms in the last three rows, implemented in OPPerTune, use the first 3 week-frames for tuning. The first row is the baseline performance achieved with manually-chosen configuration choices.

<table>
<thead>
<tr>
<th>Method</th>
<th>(P50, P95, max) of P95 latency of each hour over the 4th week (ms)</th>
<th>Sample Complexity (#rewards for tuning)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cluster 1</td>
<td>Cluster 2</td>
</tr>
<tr>
<td>Pre-deployment choices</td>
<td>12.7 19.4</td>
<td>23.8</td>
</tr>
<tr>
<td>HybridBandits(_{\text{cluster, hour}})</td>
<td>10.6 17.3</td>
<td>19.3</td>
</tr>
<tr>
<td>DDPG</td>
<td>10.4 17.0</td>
<td>23.2</td>
</tr>
<tr>
<td>AutoScope(_{\text{cluster}})</td>
<td>10.6 15.8</td>
<td>17.0</td>
</tr>
<tr>
<td>AutoScope</td>
<td>10.1 15.8</td>
<td>23.7</td>
</tr>
</tbody>
</table>

### Table 4: Comparison of techniques for post-deployment configuration tuning of Apache Spark parameters in MLExp application, on 2 production clusters, over 1 week of evaluation. AutoScope is competitive to domain-expertise based scoping strategy (Bluefin\(_{\text{cluster, type, size}}\)) using far fewer samples.

<table>
<thead>
<tr>
<th>Method</th>
<th>Experiment Completion Time (in minutes)</th>
<th>Sample Complexity (#rewards)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cluster 1</td>
<td>Cluster 2</td>
</tr>
<tr>
<td>Pre-deployment choices</td>
<td>105.85 ± 16.75</td>
<td>36.66 ± 1.60</td>
</tr>
<tr>
<td>Expert choices</td>
<td>42.41 ± 5.28</td>
<td>34.46 ± 4.72</td>
</tr>
<tr>
<td>Bluefin(_{\text{cluster, type, size}})</td>
<td>38.56 ± 6.55</td>
<td>30.79 ± 0.52 94</td>
</tr>
<tr>
<td>AutoScope</td>
<td>38.98 ± 5.90</td>
<td>32.71 ± 0.26 29</td>
</tr>
</tbody>
</table>

### 7.3 Mitigating Cost Of Tuning In-Deployment

So far, we have focused on the impact of tuning on the application performance. We now turn to the cost of tuning in deployment—every change to configuration parameters in production introduces potential risk. This section also demonstrates how operators can use the selector (Section 5) module of OPPerTune to select a subset of parameters that improve another metric of interest along with performance. In this case, we use availability as the metric of interest. As we discussed in Section 2, tuning certain configuration parameters necessitates microservice/pod restarts, causing downtimes. Improving latency of the application at the expense of throughput, or service reliability, may not be acceptable.

We evaluate OPPerTune, in terms of how it trades off improving performance and mitigating restarts while tuning, on the SocialNetwork application and constRPS workloads. The results are summarized in Table 6 and in Figure 5. We consider various strategies for picking configuration parameters, followed by tuning the selected parameters with HybridBandits, to mitigate the number of pod restarts. The first row of the
### 7.4 Scalability

We evaluate the scalability of OPPerTune service with respect to various tuning algorithms that we implemented (or integrated) as part of the backend (as discussed in Section 6). We perform this study on the same VM type as in Section 7.1.1.

We focus on the throughput of OPPerTune service, i.e., number of Predict and SetReward requests served, with various back-end algorithms; so, we use a simple synthetic application to stress-test the service and the algorithms.

Using wrk2, we simulate a scenario where there is one client that (a) creates a new instance, specifying 30 configuration parameters (25 numerical and 5 categorical for algorithms that allow hybrid configurations) and the algorithm to use, and (b) then repeatedly sends Predict and SetReward requests to the created instance on the OPPerTune server, for 120 seconds. For algorithms which need context (AutoScope, DDPG), we use random context vectors of size $d = 8$. For rewards, we use a random value between 0 and 1 (actual reward value does not matter for this study). We maintain the instance state in memory for all the tuning algorithms for this study.

In this setting, we first verified that our Go-based server can handle 4096 rps without dropping any request, bypassing the tuning algorithm. Table 7 shows the OPPerTune service throughput for various backend algorithms. First, we find that the ordering of Bluefin, HybridBandits, AutoScope, and DDPG (our implementations) is as expected. The latency of the requests is proportional to the size of the model—HybridBandits achieves lower throughput than Bluefin, because it needs to sample from a probability distribution over all possible categorical choices ($|C| = 720$ in this study).

#### Figure 5

Figure 5 shows the relationship between average P95 latency (measured in milliseconds) and the number of pod restarts per round for each of the five strategies shown in Table 6. We see that our microbenchmarking strategy achieves a good trade-off between cost and performance, using 2800-rps workload as example (though the findings are consistent across all rps). The best average P95 latency (17.9ms) of “Microservices ∪ Rightsizing” or “MS ∪ RS” comes at the expense of 29 pod restarts per round as seen from Figure 5. The microbenchmarking strategy (“MB-50” in 5) nearly matches the best method’s P95 latency (19.2ms), with nearly 30% fewer pod restarts per round (20 restarts).

#### Takeaway 3

HybridBandits + microbenchmarking strategy of OPPerTune reduces the cost of tuning in terms of service disruptions in deployment significantly, while achieving competitive performance.
### Related Work

**Configuration Tuning:** Performance optimization of systems through configuration tuning is a long-studied problem [37, 59, 60] that continues to garner interest from the systems research community. Prior works on configuration tuning mainly focus on parameters of specific subsystems of applications [16, 17, 24, 46, 54, 69, 60] such as database and storage or of the hosting infrastructure [21, 22, 55, 58, 63]. In such works, the configuration search space is relatively small, and the advantages of jointly tuning parameters across the software stack are not considered. Moreover, the approaches are tailored to the specific subsystem being tuned, sometimes requiring domain expertise [16, 24]. Recently, jointly tuning parameters across the software stack [25, 48] and across multiple components of an application [61] is gaining attention. Such works either ignore the cost of reconfiguration [61] or require an expensive offline training [25, 48, 64].

**RL/Bandit Algorithms:** While there are several RL-based configuration tuning approaches [55, 57, 73], they are either limited in the type of parameters being tuned or are inefficient for online post-deployment tuning scenarios. Some approaches do handle hybrid parameter spaces [43, 49] via cascaded optimization which are effective only when trained offline. Deploying parameters, observing a reward and making updates in real systems fits the bandits paradigm (sample; reward; update) better than long-horizon RL paradigm (sample; reward; sample; reward; . . .; update). Long-horizon/episodic RL has higher sample complexity and needs hand-crafted state information (e.g., utilization metrics) to learn effective policies. While this allows robustness in dynamic environments, it poses additional engineering overhead, and hand-designing state information is challenging in enterprise scenarios. In our experience working with developers, determining a useful reward cycle (time horizon) is fairly easy with some domain expertise making our bandits approach effective compared to RL and our proposed HybridBandits can tune all types of parameters in deployment without such overhead.

### 9 Conclusion

We have designed, built, and deployed the OPPerTune configuration tuning service at Microsoft. Our work differs from related work on configuration tuning in many ways: 1) we tackle challenges arising in post-deployment tuning, 2) we focus on sample complexity of algorithms as well as the cost of tuning, unlike systems tuning efforts that rely on offline training or controlled settings, 3) we give an end-to-end solution for configuration tuning that is fairly general and readily applicable. We demonstrate through two real-world deployments that our techniques yield state-of-the-art performance, are sample-efficient, and reduce the tuning cost.

This work addresses many challenges in post-deployment configuration tuning through OPPerTune. However, future work can address the following challenges and also improve the framework for easier adoption. Firstly, OPPerTune’s scalability can be evaluated by using it to tune applications with a very large (say, thousands) number of parameters. Secondly, a fine-grained analysis of different costs (e.g., performance and revenue) associated with tuning parameters can be conducted. Thirdly, tuning parameters from different layers of the software stack (e.g., OS and hardware), along with an analysis of their interrelationships, would be an exciting direction. Lastly, OPPerTune can be implemented as a Kubernetes operator for seamless integration with the application ecosystem.

### 10 Acknowledgment

We are grateful to Sayak Ray Chowdhury for discussions and feedback on the tuning algorithms. We would like to thank our shepherd Arpit Gupta and the anonymous reviewers for their valuable feedback. This work was supported by NSF grants CNS 2324859, 2214980, 2106434, 1909356, and 1750109.
References


[66] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind


A AutoScope Algorithm

Algorithm 2 AutoScope: Automatic Scoping of Configuration Tuning via Decision Trees

1: Input: tree height \( h \), learning rate \( \eta > 0 \), categorical parameter space \( C := C_1 \times C_2 \times \cdots \times C_k \), numerical parameter space \( \mathcal{W} = \mathcal{W}_1 \times \mathcal{W}_2 \times \cdots \times \mathcal{W}_m \)
2: Initialize: Tree node weights \( z_j^{(0)} \in \mathbb{R}^d \), for each internal node \( 0 \leq j \leq 2^h - 2 \), \( \ell \)th leaf node categorical space weights \( p_{i,\ell}^{(0)} = 1/|C| \), for \( 1 \leq i \leq |C| \) // uniform distribution, and \( \ell \)th leaf node numerical parameters \( w_{i,\ell}^{(0)} \in \mathcal{W}_i \), for \( 1 \leq i \leq m \), \( 0 \leq \ell \leq 2^h - 2 \) // default choices

\( \blacklozenge \) Each leaf node := a “tuning instance”, and is initialized identical to Algorithm HybridBandits Step 2.
3: for \( t = 0, 1, 2, \ldots \) do
   1: Determine the scope using the tree guided by the observed context, and use the appropriate tuning instance
   2: Observe context \( c^{(t)} \in \mathbb{R}^d \)
   3: Get the instance at the leaf node \( \ell \) by navigating the tree \( f_f(c^{(t)}; \{z_j^{(t)}\}) \)
   4: Sample categorical and numerical actions following Steps 4–6 of Algorithm HybridBandits on the \( \ell \)th instance
   5: Deploy the actions and measure reward
   6: Follow Steps 7–8 of Algorithm HybridBandits and observe the reward \( r^{(t)} = r(\cdot) \)
   7: Perform updates on the tree
   8: Update the \( \ell \)th instance using Steps 9–11 of Algorithm HybridBandits // Other tuning instances remain the same
   9: Update tree node weights, for \( 0 \leq j \leq 2^h - 2 \): \( z_j^{(t+1)} \leftarrow z_j^{(t)} + \eta \nabla \hat{r}(\cdot) \) // \( \nabla \) is the estimated gradient as in [38]

B HybridBandits Convergence

We design a simple synthetic function characterized by five continuous parameters and one categorical parameter. The categorical parameter offers a choice between two options \( \{f1, f2\} \) while all continuous parameters are bounded within the range \([0, 1]\). The blackbox function, \( f: [C, \mathcal{W}] \rightarrow \mathbb{R} \), calculates the root mean squared error of the continuous parameters. Additionally, it incorporates a bias term that depends on the categorical parameter’s choice (bias=1 if \( f2 \) else bias=0). We compare the convergence of our proposed HybridBandits algorithm, with Hyperopt’s Tree of Parzen Estimators (TPE) and the random search baseline in Figure 6. The metric plotted here is the mean cumulative “regret” (i.e., the difference between the estimates of the parameters and their optimal values, accumulated over rounds) with shaded 95% Confidence Interval (CI). We can see that random search does not converge as expected, while HybridBandits demonstrates significantly faster convergence compared to TPE.

C Experiment Details

In this section, we provide additional experimental details.

C.1 SocialNetwork + AMStraces

The AMStraces consist of traces collected from 3 different data center regions across two continents. The traces consist of the number of GET and PUT requests that arrive each minute over four weeks along with additional details like payload size, latency, etc. The peak requests per minute (RPM) across the 3 clusters are around 30000, 40000, and 140000. As seen in Figure 7, the traces show a diurnal pattern with a reduced load over the weekend. We sample from these traces such that a day’s original trace can be replayed in 1.2 hours. We started with a higher sampling rate and arrived at this as it succinctly captures the patterns in the original traces yet makes the experiment iteration feasible.

The PUT and GET requests in the AMStraces are mapped to the PUT (compose-post) and GET (read-user-timeline and read-home-timeline) requests of the social networking application. The payload size of the PUT requests is also driven by the payload sizes from the traces. The call graphs of the GET and PUT requests in the social networking application capture the complexities of the GET and PUT requests in the AMS service.

C.2 MLExp

In Table 8, we provide 95\(^{th}\) percentile of the job completion time in minutes. For 8 out of 11 job types, AutoScope achieves up to smaller P95 completion times than expert choices.

D Microbenchmarking: Top-50 Parameters

Figure 8 gives the top-50 parameters used in Table 6 and Figure 5. The figure also indicates the importance of tuning the application (30 out of top-50) and Kubernetes parameters.
Figure 7: Requests Per Minute (RPM) across the three clusters over a week. The figure shows a diurnal pattern with a significant reduction in traffic over the weekends.

Table 8: P95 Job completion times for various job types submitted to 2 MLExp clusters over 1 week of evaluation.

<table>
<thead>
<tr>
<th>Method</th>
<th>P95 Job Completion Time in minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Select</td>
</tr>
<tr>
<td>Default choices</td>
<td>58.39</td>
</tr>
<tr>
<td>Expert choices</td>
<td>17.36</td>
</tr>
<tr>
<td>Bluefin_{cluster, type, size}</td>
<td>13.47</td>
</tr>
<tr>
<td>AutoScope</td>
<td>11.45</td>
</tr>
</tbody>
</table>

(20 out of top-50) jointly as they are both critical to the application’s performance. We can also see a mixture of numerical and categorical parameters among the top 50 parameters.
Figure 8: Top 50 parameters selected by the micro-benchmarking strategy in Section 6, and the maximum absolute relative difference in P95 latencies observed by perturbing each of the 50 parameters one at a time w.r.t. a fixed baseline.