SelfTune: Tuning Cluster Managers

Ajaykrishna Karthikeyan  
Nagarajan Natarajan  
Ranjita Bhagwan  
Rodrigo Fonseca  
Tatiana Racheva  
Lei Zhao  
Yogesh Bansal

1Microsoft Research India  
2Microsoft  
3Stony Brook University

Abstract

Large-scale cloud providers rely on cluster managers for container allocation and load balancing (e.g., Kubernetes), VM provisioning (e.g., Protean), and other management tasks. These cluster managers use algorithms or heuristics whose behavior depends upon multiple configuration parameters. Currently, operators manually set these parameters using a combination of domain knowledge and limited testing. In very large-scale and dynamic environments, these manually-set parameters may lead to sub-optimal cluster states, adversely affecting important metrics such as latency and throughput.

In this paper we describe SelfTune, a framework that automatically tunes such parameters in deployment. SelfTune piggybacks on the iterative nature of cluster managers which, through multiple iterations, drives a cluster to a desired state. Using a simple interface, developers integrate SelfTune into the cluster manager code, which then uses a principled reinforcement learning algorithm to tune important parameters over time. We have deployed SelfTune on tens of thousands of machines that run a large-scale background task scheduler at Microsoft. SelfTune has improved throughput by as much as 20% in this deployment by continuously tuning a key configuration parameter that determines the number of jobs concurrently accessing CPU and disk on every machine. We also evaluate SelfTune with two Azure FaaS workloads, the Kubernetes Vertical Pod Autoscaler, and the DeathStar microservice benchmark. In all cases, SelfTune significantly improves cluster performance.

1 Introduction

Large cloud services depend upon cluster managers such as Protean [37], Borg [65], Twine [61], and Kubernetes [6] for job scheduling [32, 33, 39, 47, 53], virtual machine pre-provisioning [43], and resource autoscaling [42, 50, 51]. Cluster managers employ algorithms or heuristics to improve metrics such as throughput, latency, and resource utilization. Often, these algorithms rely on multiple configuration parameters that critically influence their behavior, that we call cluster manager parameters. For instance, Kubernetes exposes parameters cpu-histogram-decay-half-life and recommender-interval to help the autoscaler [8] react promptly to changes in cluster utilization without reacting to extremely ephemeral changes in utilization.

Every cluster manager relies on developers to manually set these configuration management parameters to “suitable” values. Table 1 gives examples of such parameters (not exhaustive) for different cluster managers. Typically, developers set these values using a combination of domain-knowledge and a limited set of manually-run tests or canaries [38, 60, 62]. While using domain knowledge is a step in the right direction, limited testing has many disadvantages. First, the tests may not widely explore different values of these parameters in different environments. Second, the search space of feasible values explodes exponentially when multiple inter-dependent parameters can be tweaked simultaneously. Third, cluster usage can change with time, and the best parameter values would therefore change with time as well. Consequently, clusters with manually tuned parameter values may result in reduced throughput, high request latencies or low resource utilization.

For instance, we find that using the default values for cpu-histogram-decay-half-life and pod-recommendation-min-cpu parameters of the Kubernetes autoscaler drops the system throughput to nearly 50% when the workloads arrive in short, heavy bursts (Section 7).

To address this problem, we observe an interesting similarity between cluster manager algorithms and reinforcement learning (RL) algorithms. Cluster managers (examples in Table 1) often use state reconciliation: periodically, they observe the current state of a cluster in terms of health and utilization metrics, compare it to a desired state, and take action to move the observed state closer to the desired state [27]. For instance, the Kubernetes autoscaler [8] continuously determines how to update container sizes, by maintaining a histogram of recent resource utilization values. RL algorithms are also iterative...
Despite the simplicity of the Bluefin algorithm, SelfTune is successful and has low sample complexity (number of iterations to converge to the desired cluster state) across applications (Sections 5, 6, 7). This stems primarily from the fact that SelfTune does not learn a single complex model or “policy” for the various scenarios (e.g., high/medium/low workloads) and states (e.g., resource utilization levels, failures) of the deployment environment, unlike standard RL techniques previously used in systems [45], to tune parameters. Instead, SelfTune relies on natural, pre-determined “scoping” of scenarios to learn optimal parameters per scope (e.g., one model per machine in our WLM deployment). This scoping, along with light-weight parameter updates (Bluefin) within each scope, makes our solution sample efficient, requiring only about 20 iterations to converge in all the case-studies in this work.

This paper makes the following contributions.

(1) We present SelfTune, a framework that developers can use to automate parameter search for their cluster manager via a minimal interface (Section 3).

(2) We use a novel algorithm, Bluefin, based on rigorously-studied ideas in online learning [35, 52], which allows multiple parameters to be tuned quickly and simultaneously (Section 4). We show that, using this algorithm, SelfTune enables systems to converge to their objective, i.e., their most desired state faster than previous systems that use Bayesian Optimization [55] and state-of-the-art RL algorithms [26] (Sections 2.1, 7). We have open-sourced an implementation of SelfTune with the Client API and Bluefin [14].

(3) We describe our deployment of SelfTune on WLM and show results from multiple clusters where SelfTune achieves up to 15%–20% improvement in the throughput (Section 5).

(4) To the best of our knowledge, ours is the first work to describe a developer-centric framework that allows automated parameter tuning for online systems, not just cluster managers, with large-scale deployments. We show the generality of SelfTune in the contexts of (a) resource management for Azure FaaS with real production traces [54] (Section 6) yielding significant improvement in resource efficiency and (b) container rightsizing with Kubernetes and DeathStar benchmark [36] (Section 7), yielding significant improvements in tail latency as well as throughput.

### Table 1: Key numerical configuration parameters of popular cluster management frameworks.

<table>
<thead>
<tr>
<th>Cluster manager</th>
<th>Parameter Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kubernetes (Vertical Pod Autoscaler)</td>
<td>pod-recommendation-min-cpu</td>
<td>Minimum CPU recommendation for a pod</td>
</tr>
<tr>
<td></td>
<td>memory-aggregation-interval</td>
<td>Window length for memory utilization histogram</td>
</tr>
<tr>
<td></td>
<td>history-length</td>
<td>Time to wait before halving the weights of past memory measurements</td>
</tr>
<tr>
<td></td>
<td>recommendation-margin-fraction</td>
<td>Fraction of usage added as the safety margin to the recommended request</td>
</tr>
<tr>
<td></td>
<td>cpu-histogram-decay-half-life</td>
<td>How long to wait before halving the weights of past CPU measurements</td>
</tr>
<tr>
<td>Azure FaaS (App manager)</td>
<td>prewarm</td>
<td>Time to wait before pre-loading function code</td>
</tr>
<tr>
<td></td>
<td>keepalive</td>
<td>Time to wait before retiring the loaded VM</td>
</tr>
<tr>
<td>Azure Protean (VM allocator)</td>
<td>num-aa</td>
<td>Number of rule-based VM allocation agents</td>
</tr>
<tr>
<td></td>
<td>k</td>
<td>k-highest quality clusters for VM placement</td>
</tr>
</tbody>
</table>
2 Related Work

Optimally configuring systems is a long-studied research problem in both systems and machine learning [34, 40, 4, 63, 64, 69]. In this section, we describe how SelfTune improves upon previous work, in terms of both the core algorithm it uses, and the framework it provides to the developer.

2.1 Algorithm

Commonly-used techniques for tuning or learning system parameters are variants of Bayesian Optimization [25, 55], reinforcement learning [20, 68], and heuristic search [30].

**Bayesian Optimization (BO):** CherryPick [23] uses BO to pick the best cloud configuration for big data analytics while Metis [40] uses it to improve performance metrics like tail latency by tuning key system parameters. BO is meant for settings where one seeks global optima of a fixed reward function, and requires users to specify a model for the function. For example, CherryPick and Metis use Gaussian Processes to model the prior function. In contrast, Bluefin focuses on online settings where the reward function may change with time, and so does the optimal solution. Also, BO algorithms have high sample complexity, i.e., the number of parameter deployments needed to converge to an optimal solution is large, especially when the number of parameters is also large. Thus BO is not ideal for tuning parameters in deployment or online. Our evaluation in Section 7 confirms this.

**Reinforcement learning (RL):** RL solutions for tuning database systems [68] or learning scheduling algorithms [45] support continuous parameters but require the system to explicitly provide state information in addition to reward values. For instance, Decima [45] needs every node to specify state in terms of a feature vector consisting of average task duration, number of servers assigned to a node, etc. Defining and implementing states needs domain expertise and engineering effort which is hard to scale across diverse systems. In contrast, Bluefin works with just the reward values and does not need the system to explicitly define such state.

**Heuristic search:** Using branch-and-bound [30] based techniques for large combinational spaces, or domain-specific deductive search for high-dimensional spaces [69] are primarily meant for systems where the goal is to obtain the best configuration parameters for a fixed reward function, and a fixed set of workloads. Often, these techniques do not apply to online settings for the same reason as BO (discussed above); also, heuristic search space modeling lacks the generality of RL techniques like contextual bandits [20, 26] and Bluefin.

SelfTune’s Bluefin algorithm addresses the concerns in both BO and state-of-the-art RL techniques. It is a principled gradient-descent based algorithm which (a) needs no modeling, ML expertise, or non-trivial engineering effort, (b) works seamlessly with large real-valued and discrete parameter spaces, and (c) converges to local (or global) optima, with fewer samples than previous approaches.

2.2 Framework

MLOS [31] is a framework to automatically tune configuration parameters using BO; thus, its applicability is limited as discussed above. OpenTuner [24] provides a meta-framework using which domain-specific tuners can in turn be built. CG-PTuner [29] considers contextual data, e.g., workload information, for DBMS tasks and uses BO to guide tuning. Best-Config [24] finds good configuration settings using heuristic search and sampling techniques. As discussed in Section 2.1, these techniques do not generalize to dynamic environments unlike SelfTune, where the rewards observed change with time, and in turn the optimal configurations themselves.

OtterTune [64, 66] is a framework for tuning DBMS configuration parameters. Though it also uses a variant of BO for tuning, it incorporates a novel technique to mitigate the risk of using stale configurations for new workloads. It builds ML models for selecting an appropriate workload (from a workload repository) that best represents the current workload, and uses the selected workload to estimate the effect of parameters on the current workload. In contrast, our online setting is much more dynamic, where it is extremely challenging to characterize and maintain such repositories.

AutoPilot [51] reduces resource wastage for containerized workloads using ML techniques for setting job-specific resource limits based on resource utilization. SelfTune is orthogonal to such cluster management frameworks and solutions — in fact, we show how SelfTune helps tune the key parameters of the open-source version of AutoPilot, called Vertical Pod Autoscaler [8], that is part of Kubernetes, in Section 7.

State-of-the-art RL frameworks, e.g., Microsoft’s Decision Service [21] apply in settings when the parameter (“action”) space is discrete or categorical, as they rely on “multi-arm bandit” formulations [20]. Extending these techniques to multiple numerical parameters results in very large action spaces, and in turn, high sample complexity for learning optimal parameters (as we see in Section 7). RL frameworks like SmartChoices [28] naturally support numerical parameters, but rely on providing explicit reward separately for each parameter. We, on the other hand, do not require such disambiguation — our problem formulation, and Bluefin, work with a single reward value (i.e., the desired system state objective) for tuning several, possibly inter-dependent, parameters together. This makes our learning much more sample efficient (as highlighted in Section 4).

3 SelfTune Overview

We provide an overview of SelfTune and, using a simple example, explain how a developer uses it. Then we describe SelfTune’s main system components and their functions.
3.1 SelfTune Interface

To use SelfTune, a developer augments their cluster manager code in four ways. First, they specify the set of parameters SelfTune should tune. Second, they either initialize a fresh SelfTune instance or connect to an existing one. Third, they specify what point in the code and at what frequency SelfTune should update the values of these parameters. Finally, they use the current state of the cluster to determine a reward, which captures the difference between the desired state and the current state of the cluster. SelfTune’s algorithm (in our implementation this is Bluefin) uses this reward to set the parameter values in the next iteration. One of the main insights of this work is that the cluster manager already computes the current state of the cluster, and already has a notion of the desired state of the cluster. Hence SelfTune simply piggybacks on existing code to determine the reward, which is essential for any reinforcement learning platform.

The example in Figure 1 shows how a simple token-based job scheduler uses SelfTune to tune the frequency with which it makes scheduling decisions. Using this example, we now describe the SelfTune-specific additions to code in detail.

```java
1: public const double optLoad = 0.80;
2: // UpdateCycle = new TimeDelta("00:00:05");
3: Config UpdateCycle = new Config("UpdateCycle",
4: { Specification *TimeDelta*,
5: *00:00:01-00:00:30*,
6: *00:00:05*};
7: SelfTune st = new SelfTune.Create(UpdateCycle);
8: st.Connect();  Creation
9: // This is the scheduler loop
10: var currentLoad = 0.0;
11: while(1)
12: {
13: if (currentLoad < maxLoad)
14: {
15: int numTokens = GenerateTokens(currentLoad);
16: GrantTokensToJobs(numTokens);
17: }
18: Guid callId;  Prediction
19: UpdateCycle = st.Predict(callId, "UpdateCycle");
20: sleep(UpdateCycle);
21: currentLoad = CalculateLoad();  Feedback
22: st.SetReward(callId, currentLoad - optLoad);
23: }
```

Figure 1: Token-based scheduler augmented with SelfTune to tune the frequency with which its main algorithm runs — the highlighted lines show the three basic additions for SelfTune.

Specify Tunable Parameters: For each parameter, the developer specifies its data type, and optionally, initial value, a range of permissible values, and step-size (e.g., TimeDelta data type with values in multiples of 5 seconds). Line 3 in Figure 1 says that SelfTune should tune the UpdateCycle parameter, which determines the time between consecutive iterations of the main scheduler loop. Here, the developer has specified that this parameter can lie between 1 second and 30 seconds. They also specify 5 seconds as the initial value of UpdateCycle.

The developer has determined that UpdateCycle should be tuned because if the scheduler waits too long between iterations, it will not react fast enough to changes in cluster state, hence causing the cluster resources to be used sub-optimally. If, on the other hand, the scheduler iterations run very frequently, the scheduler may react prematurely to extremely transient changes to system state, thereby causing sub-optimal resource usage. Note that though this example shows SelfTune tuning only one parameter, one instance of SelfTune can tune any number of parameters simultaneously.

Initialize and Connect to SelfTune Instance: Line 7 in Figure 1 starts a new SelfTune instance. In a cluster-wide deployment, the developer decides how many instances of SelfTune to set up. In our WLM deployment, each machine initializes a separate SelfTune instance. However, if needed, cluster managers can reuse the same instance of SelfTune across various machines, simply by connecting to an existing SelfTune instance (Line 8).

Get Parameter Values: Lines 11 to 23 show the main scheduler loop. Lines 13 to 17 capture the main algorithm of the scheduler. The developer measures the current cluster state as the currentLoad (set to 0 in Line 10 and updated by the function CalculateLoad() in Line 21). The developer states the desired cluster state, i.e. optLoad, in Line 1. If the current load of the system currentLoad is less than a specified optimal load optLoad, it generates a number of tokens proportional to the difference between the maximum permissible load and the current load.

After this, in Line 19, the scheduler invokes SelfTune’s Predict function to determine the value of UpdateCycle, and sleeps for UpdateCycle seconds. Without SelfTune, the scheduler loop would have slept for a fixed value of 5 seconds, as the commented Line 2 shows.

Set Reward Function: SelfTune needs the developer to specify a domain-specific function to determine the outcome of tuning the specified parameters. Note that the developer’s code already defined both optLoad and currentLoad since the core scheduling algorithm uses them both. The developer reuses this pre-existing code: in Line 22, the developer inputs the difference between currentLoad and optLoad to SelfTune’s SetReward function as the reward value.

Every reward is a result of a particular set of parameter values. Hence, the code associates the calls to Predict and SetReward by using the same callId. The Data Collector stores this information for later use, as explained in Section 3.2.
3.2 SelfTune Components

In this section, we describe the different components SelfTune needs to support the functions in Section 3.1. Figure 2 depicts the four main components: the Client API (which supports the \texttt{Predict} and \texttt{SetReward} functions), the Learning Engine, the Data Collector, and the Reward Tracker. Appendix A discusses the specifics of the client API. We describe the rest of the components here.

Learning Engine: The learning engine implements the necessary optimization algorithms such as Bluefin. While SelfTune primarily uses Bluefin, the framework itself is generic and can therefore include other algorithms, e.g., Azure Decision Service’s Contextual Bandits.

Section 4 describes Bluefin algorithm that uses a variant of gradient descent. It determines the next value of the parameter based on how the cluster reacted to past parameter choices. For instance, in the example explained in Figure 1, Bluefin observes past values of \texttt{UpdateCycle} and the corresponding reward, and then determines the next value with the objective of getting the load as close as possible to \texttt{optLoad}.

Data Collector: The data collector is a background service that maintains the history of all parameter values and the corresponding reward for each SelfTune instance. In the example of Figure 1, whenever the client code makes a call to \texttt{Predict} and a subsequent call to \texttt{SetReward}, the data collector associates the parameter values and reward using the \texttt{callId} and stores this as the tuple \((\text{callId}, a, r)\) where \(a\) is the value for \texttt{UpdateCycle} that the client code obtained by calling \texttt{Predict}() and \(r\) is the resulting reward. The learning engine uses this data to set future parameter values as described in Section 4. The data collector also stores references to each SelfTune instance so that cluster managers can lookup existing SelfTune instances and connect to them.

Characterestic 1: Bluefin uses One-point Feedback. Cluster state is the result of a complex combination of parameter values and external factors such as sudden bursts in workloads and time-of-day effects. Therefore, the reward, which is a function of the cluster state, also changes with time. Modeling this behavior using a fixed function is extremely difficult, if not impossible.

Bluefin (like any other RL approach) uses rewards only at the parameter values that the cluster manager obtains by calling \texttt{Predict}. It does not assume any other information about the inherent, unknown function that determines the reward. In other words, following standard practice in RL literature [21,56], Bluefin assumes only bandit-feedback or zeroth-order access to the reward function. This constraint is referred to as “one-point feedback” [35], as against multi-point feedback [22] in learning theory. Techniques such as Bayesian

![Figure 2: SelfTune architecture. The cluster manager interacts with the SelfTune server via client API. The learning happens on the server side, and it is transparent to the cluster manager.](image-url)
Optimization, branch-and-bound heuristics [30], and genetic algorithms [46, Chapter 1.6] need to compute the reward for multiple parameter values that may not have been deployed in the system. Hence they need a model to represent the potentially complex and unknown reward function. Thus, these techniques are much more suited to offline tuning than to our setting of tuning in deployment to optimize cumulative time-varying rewards.

**Characteristic 2: Bluefin has Low Sample Complexity and Low Engineering Overhead.** Our goal is to reach the optimal parameters that maximize the cumulative reward over time. The metric of efficiency is sample complexity, i.e., the number of rounds it takes to converge to the optimal values. Each round can be very resource-intensive (as we discovered in SelfTune’s deployment on WLM, explained in Section 5), so the fewer the number of rounds the less the overhead of the parameter tuning framework itself. SelfTune reduces the engineering overhead and makes tuning highly sample efficient by letting developers statically identify suitable “scopes” for tuning. That is, rather than learning a single global model to account for all the complex behaviors of the underlying system being tuned, it allows developers to instantiate a SelfTune instance per scope (e.g., WLM uses machine as the scope, in Section 5). Each instance executes Bluefin to learn optimal parameters within its scope, thereby solving a relatively easier problem. Second, Bluefin algorithm (in each SelfTune instance) can be thought of as learning a model of size equal to the number of parameters tuned, unlike standard RL techniques that use sophisticated models with orders of magnitude more parameters to capture system states and behaviors. Thus, both the sample and the engineering complexities of Bluefin is much lower than the standard RL approaches.

**Algorithm:** We first define a few terms used to explain the algorithm. Say the developer wants to tune $m$ parameters. In each round, the cluster manager receives an $m$-dimensional vector $a(t)$ when it calls Predict, and as a result of setting these values, it measures cluster behaviors and reports back a reward value $r_i : a(t) \rightarrow \mathbb{R}$. SelfTune then uses this reward to update the parameter values.

Algorithm 1 presents the core function of Bluefin, which leverages ideas from the rigorously-studied derivative-free online optimization [35, 52] in the machine learning theory community. There are two key challenges in our tuning setting. First, if we know the exact reward function, $r_i$, then we can apply the standard online gradient descent techniques [70]. However, in a deployed cluster, we do not have any information on $r_i$ other than the one-point black-box access to it. Second, standard gradient-descent style updates are derived for real-valued parameters. However, cluster manager parameters can be discrete as well as real-valued.

(1) To tackle the two challenges, we leverage the derivative-free optimization ideas studied in learning theory [35, 52]. They showed that we can reliably estimate the gradient of the black-box reward function by randomly perturbing the parameters once, albeit under some assumptions on the function. In particular, the theory requires that the problem be continuous, i.e., parameters are all real-valued. In practice, we often have to tune discrete-valued parameters. To this end, Bluefin introduces a function $g$, which it appropriately defines during the Create call, to map the real-valued parameters and the generic data-types that can be deployed in the system. In other words, Bluefin executes the well-studied online gradient descent updates in a suitably transformed parameter space. We discuss the details next.

**Algorithm 1 Online tuning of parameters in SelfTune**

1: procedure Bluefin (radius $\delta > 0$, learning rate $\eta > 0$)
2: Initialize the parameters $w(0) \in \mathbb{R}^m$ // Create
3: Initialize $g(\cdot)$ // Create
4: for $t = 0, 1, 2, \ldots$ do
5: Uniformly sample $u \in \mathbb{R}^m$ from $\{u : \|u\|_2 = 1\}$. // Predict calls
6: Compute the perturbed parameters $\hat{w} := w + \delta u$
7: Client receives perturbed decisions $a(t) := g(\hat{w}(t))$
8: // SetReward
9: Do “one-point” gradient-ascent update (to maximize the reward): $w(t+1) \leftarrow w(t) + \frac{1}{\delta} \cdot r(t) \cdot u$

**Initialization (Line 2).** The algorithm works with a real-valued parameter vector $w \in \mathbb{R}^m$, where $m$ is the total number of parameters to tune. If the developer does not give initial value for parameter $i$, the algorithm samples $w_i$ uniformly at random from the specified range (suitably scaled, see below). If the developer has not provided a range, it initializes $w_i$ to 0.

**Defining g (Line 3).** If the developer specified a step-size, $g$ appropriately scales the corresponding components of $w$. For instance, if the developer specifies that the $i$th parameter is an integer that needs to have a step-size of 5, then $g(w_i) = 5 \cdot \text{round}(w_i)$, where $w_i$ is the real value that the algorithm manipulates, and round is the round-to-the-nearest-integer function. Similarly, if the developer specified range constraints on the parameter, then $g$ appropriately projects $w_i$ to lie within the specified bounds. Predict applies the $g$ function before returning the parameter values, as in Line 7 of the Algorithm.

**Update parameters (Lines 5, 6, 9).** To update $w$, we use the technique of [35], where we estimate the gradient of $r_i$ with respect to $w(t)$ by a random perturbation of $w(t)$. Line 6 effectively samples a vector $\hat{w}$ from the hyper-sphere centered at $w$ with a radius $\delta$ (input to the algorithm, appropriately set
as discussed below). Line 9 computes a gradient-ascent style update (to maximize the cumulative reward) in the direction of the random vector \( u \) scaled appropriately by the learning rate \( \eta \), and the observed reward value \( r_t \) at the perturbed vector. In some cases, such as in simulation settings, one may be able to perturb the vector more than once and make reward measurements. It turns out that with just two-point feedback, we can get a very accurate estimate of the gradient (in lieu of Section 6) where it is possible to obtain reward \( r_t(\cdot) \) at two different \( a \) values. In that case, the gradient estimator in Line 9 of Algorithm 1 can be replaced with:

\[
 w^{(t+1)} \leftarrow w^{(t)} + \frac{1}{\eta} \left( r(g(w^{(t)} + \delta u)) - r(g(w^{(t)} - \delta u)) \right) u.
\]

Setting radius \( \delta \), learning rate \( \eta \). In general, choosing a single real number \( \delta \) can be tricky especially when the parameters have different scales. But, decoupling the deployed parameters \( a \) (that may have very different scales) from the weights \( w \) (that are in a normalized scale) via \( g(\cdot) \) mitigates this issue in practice. Given \( \|w^{(t)}\|_2 = O(1) \) and \( \|u\|_2 = 1 \), we set \( \delta = O(1) \) and \( \eta = O(\delta^2) \), so that \( w^{(t+1)} \) retains the scale after the update in Line 9 of Algorithm 1.

5 Large-scale Workload Scheduling System

In this section, we describe our experiences deploying SelfTune with WLM (short for “workload manager”), the background task scheduler for Substrate, a large data management engine used by many of Microsoft’s services. We first describe Substrate and WLM, and then the deployment of SelfTune with WLM, and finally present evaluation.

5.1 Substrate

Substrate is a large-scale data management engine at Microsoft which hosts data for several of Microsoft’s enterprise services such as Exchange Online, an enterprise email service, and SharePoint, an online collaboration platform. Substrate stores data in a local database on each machine. Substrate runs upon hundreds of thousands of machines worldwide and hosts billions of data items.

In Substrate, compute and storage are tightly coupled. Each machine runs many user-facing tasks, such as reading emails, writing documents, and searching through data. These tasks are latency-sensitive and need to complete within a few milliseconds. Simultaneously, Substrate runs a vast range of background tasks on the same machines such as data indexing, data analytics, machine learning, and data defragmentation. More than 70% of all tasks that run on Substrate are background tasks. An example background task analyses a customer’s mailbox to provide daily to-do lists [10]. Most tasks are defined to finish very quickly (e.g., process one mailbox and return), in the order of a few seconds.

5.2 WLM

To ensure that background tasks do not interfere with user-facing tasks, Substrate uses a background task scheduler called WLM which regulates these tasks’ access to disk,\(^2\) CPU, memory, and network on that machine. WLM continuously polls the background task queues, granting the tasks access to resources when permissible (to ensure high throughput), while trying to keep resource utilization on the machine within a specified range (to ensure room for user-facing tasks).

Figure 3 depicts WLM’s scheduling algorithm. WLM’s resource monitor continuously tracks CPU, disk, network, and memory usage (IO latency for disk, utilization % for CPU and memory, and a function of bandwidth utilization and ping losses for network). For each resource, developers specify a lower and a higher usage thresholds. If the resource’s utilization is under the lower threshold, the resource is said to be under-utilized. Similarly, if the resource’s utilization is over the higher threshold, WLM considers it to be over-utilized.

Configuring WLM: Every few seconds, determined by a configuration parameter called RefreshCycle, WLM updates a state variable called MaxConcurrency. MaxConcurrency determines the maximum number of background tasks that can run on a machine simultaneously. WLM operates an Additive Increase, Multiplicative Decrease (AIMD) algorithm to determine MaxConcurrency: every RefreshCycle seconds, it determines resource usage. If all the four resources are under-utilized, WLM increments MaxConcurrency by 1. Even if even one of the resources is over-utilized, WLM cuts MaxConcurrency to half its current value. For instance, the developer may set the higher threshold for CPU usage to 60%, the idea being to reserve 40% for the more important user-facing tasks. If WLM observes that background tasks are

\(^2\)majority of Substrate data is hosted on cost-effective HDD media.
using more than 60% CPU, it decreases MaxConcurrency to half the current value, thereby going into a mode of rejecting tasks until the usage comes sufficiently down. WLM thus gradually schedules more tasks and increases resource utilization, while also checking that no resource is over-utilized.

The ideal value of RefreshCycle depends on machine type and workload characteristics. A less powerful machine might benefit from a larger RefreshCycle. A smaller value of RefreshCycle may help machines with variable workloads. In the absence of an automated tuning framework, WLM’s developers have set up different versions of this parameter such as CPU-RefreshCycle, machine-type-A-RefreshCycle, etc. to control it in different contexts. This approach increases the number of configuration parameters, hence management overhead, as well as the developer burden to continuously check cluster state in these various contexts and manually tweak parameters.

Our deployment of SelfTune automatically and continuously tunes only one configuration parameter – RefreshCycle – for every machine independently, which is the scope identified by the domain experts. Developers can now stop using the context-specific RefreshCycle parameters, and also stop the continuous manual monitoring of the parameter value and its effect on cluster state.

Performance metrics: WLM measures its performance using two metrics, and hence SelfTune uses either one of these as its reward metric. The first, called a resource’s Healthy Utilization Percent (HUP), measures the fraction of time the resource is neither over-utilized nor under-utilized. The ideal value of HUP is 1. WLM usually calculates HUP for every hour and for every resource.

The second metric, grant ratio (GR), measures the ratio of the total number of tasks that WLM runs in a given time-period to the total number of tasks that were submitted to it in the same time-period. A grant-ratio of 1 implies that WLM did not reject any task. Thus ideally, WLM needs to drive the cluster to have HUP=1 and GR=1. We use the same metrics, aggregated over a day, as the reward function for SelfTune in our deployments.

While these are the two primary metrics that WLM directly exerts influence over, there are other workload-specific metrics, that are outside the scope of WLM, instrumented by the teams who rely on the scheduler. For instance, background task developers use a higher-level metric, i.e., background task throughput, to determine how promptly WLM schedules their tasks. This is measured as the total number of background tasks successfully completed within one day. While SelfTune does not use this as a reward metric, we use this metric to determine if SelfTune does indeed help improve the efficiency of the system (Section 5.3).

Integrating SelfTune: We integrate SelfTune with WLM to tune RefreshCycle separately for every machine. While the WLM code-base consists of tens of thousands of lines of code, we required less than 50 lines of code to integrate SelfTune, most of which is replacing parameter usage with Predict, and setting up the Reward Tracker service (to invoke SetReward asynchronously, as discussed in Section 3.2) with the appropriate reward function.

We look at aggregate metrics over a subset of machines for a month to set the scale of δ (which helps exploration) and η appropriately. We find that a single, fixed choice of δ and η works across clusters; we do not shrink these parameters to 0 with increasing iterations, which is needed in theory. This helps prevent stagnation when tuning in deployment.

Minimal overhead of running SelfTune: Each Substrate machine runs its own local SelfTune (i.e., its component services) instance; so Predict calls (executing Steps 5 and 6 of Bluefin) are just like any other local function calls in the WLM code-base. Parameter updates (Step 9 of Bluefin) are extremely light (at most 5 FLOPS) and are made once a day when the reward arrives. To enable debugging, the Data Collector (introduced in Section 3) persists a history of (callId, a, r) tuples from previous 30 days; this takes at most a few hundred KBs space per instance in production. Overall, there is minimal overhead to operationalizing SelfTune in production, in terms of both compute and space.

We enable parameter tuning with SelfTune on individual production servers via flights, a mechanism used for gradually deploying any code change in production. Deployment starts with a few hundreds of servers, and then slowly expands to more servers. This helps us perform controlled experiments.

5.3 Evaluation

In the section, we first describe our evaluation methodology. Then, we describe our experiments and results.

Evaluation Methodology: A significant challenge we faced while evaluating SelfTune is that resource HUP varies widely week over week in Substrate. Figure 4 shows the disk HUP over six weeks in Aug-Sep 2021 for two randomly chosen machine sets in a representative cluster in South America consisting of 450 servers. The sets contained 225 machines each, and were completely disjoint. The figure shows that, for the same machine set, utilization changes significantly from one week to the next. Hence we cannot evaluate the efficacy of SelfTune simply by observing HUP on the machine set before deploying SelfTune, and comparing it to HUP after deploying SelfTune. However, we also observe that the distributions of disk HUP computed on the two disjoint machine sets are very similar (e.g., relative difference between HUP P50 percentiles of the two sets was ≤ 0.5% for all weeks). Therefore, to evaluate SelfTune, we deploy it on one machine set, called the Treatment Group, and compare this machine
Similarly, we evaluate grant ratios across the self-tune significantly. Table 2 describes the duration of the metrics. We find that all and utilization improves each cluster. Table 2 gives the P25, P50 and We see drastic improvements in the GR SelfTune for Even though the im-

3.1 With predominantly CPU-intensive workloads, had low CPU
3.2 With predominantly disk-intensive workloads, faced heavy
disk throttling and consequently had poor GR. (3) Cluster 3, with predominantly CPU-intensive workloads, had low CPU HUP (CPUs were mostly in the over-utilized state), and consequently, low GR (recall that the CPU MaxConcurrency will quickly drop to 1, when it is in a over-utilized state for a short amount of time).

Our objective was to see if SelfTune, by tuning RefreshCycle on each machine in the cluster, could improve GR for Cluster 1 and Cluster 2, and CPU HUP for Cluster 3. Thus, in Cluster 1 and Cluster 2, we set up SelfTune with the Grant Ratio (GR measured over a period of one day) as the reward metric. For Cluster 3, we set up SelfTune with CPU HUP as the reward. In Cluster 1, we initialized RefreshCycle to 20 minutes since it was the default value used for the cluster. For Cluster 2 and Cluster 3, we initialized RefreshCycle to the default value of 6 seconds that was already in use.

(1) SelfTune improves utilization metrics in all three clusters significantly, Table 2 describes the duration of the experiments, sizes of the control group and treatment group, and the impact on the performance metrics using SelfTune. In particular, for each cluster, it shows the improvements in the resource HUP and the GR metrics. Given confidentiality requirements, we are unable to present absolute numbers, but present the percentage improvements. Since SelfTune separately tunes RefreshCycle on every machine, we present improvement in utilization in terms of 25th %-ile (P25), 50%-ile and 75%-ile of metric values across all machines in the treatment group relative to the corresponding percentile values in the control group (during the deployment period). For all the results, we ensure statistical significance using the standard t-test, at a p-value of 0.05.

From Table 2, we observe significant improvements in GR, between 18% and 178% improvement in the median, across all three clusters. We see drastic improvements in the GR metric in Cluster 1, chiefly due to the sub-optimal and obsolete choice of RefreshCycle value used in this cluster (reflected in the Control Group). In Cluster 3, where SelfTune employed CPU HUP as the reward, the improvement in the median CPU HUP was around 2% (also see Figure 5 that shows relative values for confidentiality reasons).

Even though the improvement in HUP is small (2%–3%), it is statistically significant; importantly, even a 2% improvement in the median HUP implies several minutes to an hour of better resource utilization per machine per day for at least 50% of the machine-days in the cluster. The actual impact is magnified manyfolds by the number of machines in the cluster over weeks and months. Furthermore, the small improvement in HUP led to significant improvements (18%–20%) in the GR metric.

(2) SelfTune has to tune RefreshCycle separately and continuously for each cluster. Table 2 gives the P25, P50 and P75 values of RefreshCycle that SelfTune converged to in each cluster. We find that Cluster 1’s RefreshCycle values converged to a median value of about 6 minutes, Cluster 2’s median value was about 10 minutes, whereas Cluster 3’s median value was much lower, i.e., 2.6 seconds. Additionally, in some cases, there is a significant spread of converged values within a cluster, as the P25 and P75 values show. Such differences in the ideal values of RefreshCycle are due to

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Control Size</th>
<th>Treatment Size</th>
<th>Experiment Duration</th>
<th>Reward</th>
<th>Metric Improvement</th>
<th>RefreshCycle Value (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P25</td>
<td>P50</td>
<td>P75</td>
<td>P25</td>
<td>P50</td>
<td>P75</td>
</tr>
<tr>
<td>1</td>
<td>144</td>
<td>144</td>
<td>July 1–July 30</td>
<td>GR</td>
<td>SI</td>
<td>SI</td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
<td>1000</td>
<td>Aug 25–Oct 12</td>
<td>GR</td>
<td>SI</td>
<td>SI</td>
</tr>
<tr>
<td>3</td>
<td>1950</td>
<td>1950</td>
<td>Oct 11–Nov 17</td>
<td>(CPU) HUP</td>
<td>2%</td>
<td>1%</td>
</tr>
</tbody>
</table>

Table 2: SelfTune experiment details, resulting performance improvement (SI=Statistically Insignificant) & RefreshCycle values.

Figure 4: Disk HUP for a cluster in South America (of 450 servers) during Aug-Sep 2021: The (normalized) percentiles drift across weeks (1% – 32%) significantly; but they vary much less (1% – 2%) across the two disjoint server sets.

Our objective was to see if SelfTune, by tuning RefreshCycle on each machine in the cluster, could improve GR for Cluster 1 and Cluster 2, and CPU HUP for Cluster 3. Thus, in Cluster 1 and Cluster 2, we set up SelfTune with the Grant Ratio (GR measured over a period of one day) as the reward metric. For Cluster 3, we set up SelfTune with CPU HUP as the reward. In Cluster 1, we initialized RefreshCycle to 20 minutes since it was the default value used for the cluster.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Control</th>
<th>Treatment</th>
<th>Experiment</th>
<th>Reward</th>
<th>Metric Improvement</th>
<th>RefreshCycle Value (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P25</td>
<td>P50</td>
<td>P75</td>
<td>P25</td>
<td>P50</td>
<td>P75</td>
</tr>
<tr>
<td>1</td>
<td>144</td>
<td>144</td>
<td>July 1–July 30</td>
<td>GR</td>
<td>SI</td>
<td>SI</td>
</tr>
<tr>
<td>2</td>
<td>1000</td>
<td>1000</td>
<td>Aug 25–Oct 12</td>
<td>GR</td>
<td>SI</td>
<td>SI</td>
</tr>
<tr>
<td>3</td>
<td>1950</td>
<td>1950</td>
<td>Oct 11–Nov 17</td>
<td>(CPU) HUP</td>
<td>2%</td>
<td>1%</td>
</tr>
</tbody>
</table>
various reasons, such as varying workload characteristics and provisioned hardware even within the same cluster. Moreover, these workload and hardware characteristics also change with time, which means SelfTune should continuously run on every cluster for WLM to be able to react appropriately and quickly to such changes. Figures 11, 12, and 13 in Appendix B show how RefreshCycle converges differently for the three clusters over the course of deployment duration.

Figure 5: Both HUP (CPU) and GR (weekly P50 values) are significantly better after SelfTune was enabled in Cluster 3, with a 1% to 3% relative improvement over the control set in utilization and a 12% to 34% improvement in GR.

Figure 6: Background task throughput (normalized w.r.t. Jan 21st) clearly improved when SelfTune was enabled.

(3) SelfTune significantly improves background task throughput. SelfTune uses either resource HUP or GR as reward metrics since WLM already calculates these metrics. Ultimately, however, background task developers want a high background task throughput. We therefore evaluate how SelfTune improves this metric. Figure 6 shows the improvement in the task throughput when SelfTune was enabled in the first week of Jan 2022, in a random half of a 750-machine cluster in the Asia-Pacific region. Before we enabled SelfTune, the throughput of both control and treatment groups were similar. Once enabled, SelfTune improves the background task throughput by as much as 17%. We disabled SelfTune on Jan 21, and the treatment group’s throughput went back to being the same as that of the control group. This shows that by improving resource HUP and/or GR, SelfTune significantly improves the background task throughput as well.

Since SelfTune has shown significant improvements in multiple metrics in our experiments, starting January 2022, we have enabled SelfTune in all Substrate clusters in North America, consisting of tens of thousands of machines.

6 Serverless Scheduling in the Cloud

Customers are increasingly using serverless computing, or “Functions as a Service” (FaaS), for deploying applications on the cloud [1, 3–5]. Previous work has proposed informed resource management strategies to use cluster resources efficiently for FaaS applications [54]. We evaluated SelfTune with this work and observed significantly improved resource usage with minimal to no performance loss. This section describes the problem, experiments, and results.

6.1 FaaS Resource Usage

Cloud providers charge FaaS-based applications for the number of functions executed, and not for the resources that the applications use. Hence, to maximize their benefit, providers seek to offer good function performance to customers with the least resources assigned to run the customers’ functions.

To achieve good function performance, the provider should load the customer’s application into memory before the customer invokes the function (warm start), as opposed to loading it from persistent storage only after the customer invokes the function (cold start). However, keeping all applications in memory at all times is prohibitively expensive. Ideally, the provider should pre-load the customer code just before the function is invoked. This approach will minimize the resources that the provider assigns this application and yet provide good performance.

To achieve this, Shahrad et al. [54] have proposed a policy that predicts two key parameters for a FaaS platform: 1) prewarm: The time the policy waits, since the last execution, before it loads the application image expecting the next function invocation. A large value of prewarm reduces resource usage but may cause cold starts. 2) keepalive: The time for which an application is kept in memory after it has been loaded in memory. A larger keepalive can reduce cold starts but will also waste resources. Therefore, the challenge is to predict suitable values of prewarm and keepalive that will provide good function performance and, at the same time, reduce resources used.

To determine these parameters, Shahrad et al. maintain a histogram of time between function invocations for each application, called the Idle Time (IT) histogram. Based on an empirical study, they suggest using keepalive = 99th percentile3 and prewarm = 5th percentile of the IT values in the histogram for all applications.

3henceforth, we write keepalive = 99, dropping the percentile
6.2 Evaluation Setup and Goals

We hypothesize that it may be sub-optimal to set `prewarm` and `keepalive` to the same value for all applications. Moreover, the IT histogram can change with time, and therefore these parameters should be set not once, but periodically. In this section, we seek answers to the following two questions:

1. Per-application Tuning: Can `SelfTune` set `prewarm` and `keepalive` for each application (i.e., use application as the scope for `SelfTune` instance) based on its invocation patterns, to achieve a better performance trade-off for the cloud provider? (Section 6.3)

2. Time-varying Tuning: Can `SelfTune` periodically tune these parameters to improve the trade-off, as the invocation patterns could change over time? (Section 6.4)

Simulation setup: We use the Python-based simulator used in [54] which replays real function invocation traces (obtained from Azure as described in Sections 6.3 and 6.4), and infers if each invocation creates a warm or cold start. The simulator also keeps track of when each application image is loaded in order to aggregate the wasted memory time for the application (i.e., the time the image is kept in memory without executing any functions). Following [54], we simulate (a) function execution times equal to 0 to quantify the worst-case wasted resource time, and (b) all applications use the same amount of memory (as memory data is only partially available).

Performance metrics: We focus on two metrics, following the analysis presented in [54]: (i) distribution (in particular, P75) of percentage of cold start invocations per application (i.e., what fraction of invocations of the app during the time period were cold starts), and (ii) wasted memory time (as defined above). We normalize (ii) w.r.t. a baseline policy of using no `prewarm` and a fixed 10-minute `keepalive` (absolute value, unlike the percentile values used throughout this section). We use the same metrics as reward for `SelfTune`.

6.3 Per-application Tuning

To answer the first question, we use the AzureFaaS dataset [2] consisting of 14 days of function invocation traces for about 22,000 applications running on Azure Functions.

Optimal global parameters: We first obtain the pareto-optimal trade-off frontier for the two parameters when they are fixed to the same value for all applications. To obtain this frontier, we did a simple grid-search with 7 `keepalive` values (100, 99, 97, 95, 90, 85, 80) and 5 `prewarm` values (1, 5, 10, 15, 20), i.e., we ran 35 simulations which took under three hours on a standard 64-core machine for this dataset (obviating the need for clever optimization/search algorithms). Figure 7 (left) plots normalized wasted memory time vs P75 app cold start percent. We see that one metric improves at the expense of the other metric, for various choices of `prewarm` and `keepalive` parameters. Our findings here align with [54], and

Optimal application-specific parameters: Doing a grid-search to determine application-specific parameters is very expensive since there are tens of thousands of applications. So we leverage `SelfTune` to determine per-application values of `keepalive` and `prewarm`. On one week of data, every time a function is invoked in the trace, we call `Predict` to determine the values of `keepalive` and `prewarm` for the application. The reward metric used is either wasted memory time or number of cold starts. We then evaluate the converged per-application parameter values on the second week of data.

Figure 7 (right) plots wasted memory time vs cold start percent when using `SelfTune`. We first observe that `SelfTune`, with memory wastage as the reward, reduces memory wastage by nearly 10% relative to the fixed optimal global choices (indicated in circled dots on both the right and left plots) without worsening cold starts. Second, application-specific tuning yields strictly better choices than the global frontier — the crosses (corresponding to the converged parameters) lie below the dots (initial values). We made similar observations when we used number of cold starts as the reward.

Figure 8 shows that the overall cost reduction with `SelfTune` can be attributed to less than 20% of the apps. `SelfTune` is able to exploit the behavior of a fraction of apps to find better choices of parameter values, while for the other apps, the default global policy parameters already work quite well.
Figure 8: CDF of app-wise reduction in the cost metrics relative to the best global policy (circled in Figure 7) achieved via SelfTune on AzureFaaS. All improvements come from $\lesssim 20\%$ of the apps (axis curtailed for clarity).

### 6.4 Time-varying Tuning

To answer the second question, i.e., whether SelfTune’s periodic parameter tuning helps reduce resource usage over time, we collected a much larger set of traces from the Azure FaaS infrastructure between July 15 and Oct 31, 2019 and used them to drive the simulator. As in the previous section, in this large-scale study, we divide the traces into pairs of consecutive weeks, use SelfTune to tune parameters per-application on the first week, and evaluate the impact on the second week. Since we use 14 weeks of data, we have 7 such pairs.

Figure 9: Performance of VM management policy [54] with app-specific tuning of parameters via SelfTune on the large Azure dataset: SelfTune is consistently superior or competitive w.r.t. the baselines along both the metrics, across weeks.

Figure 10: Distribution of differences in converged values, for the two parameters, over 3 months. SelfTune picked significantly different values in October vs. August, for over 25% apps, reflecting the temporal shift in the invocation patterns.

Figure 10 shows how SelfTune changes parameter values (for a random subset of apps) in October compared to August. SelfTune picks significantly different values, up to 300% relative change, for over 25% of the applications. This underscores the importance of continuously tuning the parameters.

### 7 Container Rightsizing

In this section, we show how SelfTune can be integrated with microservices architecture and Kubernetes to improve (a) cluster resource utilization, and (b) tail latencies of microservices-based cloud applications. We also present comparisons with BO and RL techniques.

**Simulation setup:** We use the social networking application in the DeathStar microservices benchmark [36]. We set up a cluster with 4 servers, each with 24 cores, 40GB of memory and 250GB of disk space. We restrict monitoring services to one server to avoid interference and deploy the microservices on the other three servers based on the functionality (e.g., all backend microservices are on one server). We simulate a diurnal workload, with short traffic bursts. Following [58], the workload generator [16] issues GET (read timeline), POST (create new post) requests continuously for 15 minutes at 500 requests per second, in the ratio 9 (GET):1 (POST).

**Configuration parameters:** We tune two types of parameters: (i) the first 4 CPU-related parameters listed in Table 1 for the Kubernetes VPA (Vertical Pod Autoscaler) [8], which impact the efficiency of autoscaling and throughput, and (ii) about 85 key numerical configuration parameters (2–5 parameters per microservice) for the 28 microservices in DeathStar (as identified in [58]), which impact the application latency.

**Compared methods:** We compare SelfTune’s Bluefin with two standard techniques: (i) Bayesian Optimization — the Gaussian Process (GP) method [25], implemented in [15], and used in [23, 58, 66], (ii) Contextual Bandits [26] RL technique — the $\epsilon$-greedy algorithm implemented in [19], and used in [20, 21], and (iii) Deep Deterministic Policy Gradient (DDPG) [41], a popular deep RL technique for continuous action spaces used by prior works to tune system parameters [49, 67].

For all the experiments, we initialize...
Bluefin and BO (GP) with the default parameter values as well as random values, and report the best results for each method. We note that, in this scenario, the initialization does not have significant effect on the algorithms’ convergence. For both the algorithms, the difference in performances yielded by the best configurations obtained with either initializations is around 2%–4%. Each 15-minute peak workload constitutes a sample (a round). We fix a budget of 50 samples for all the methods for fair comparison. We configure the ε-greedy and DDPG algorithms to explore for the first 25 rounds and then exploit for 25 rounds.

### 7.1 Results

#### Optimizing throughput:

We now demonstrate the significance of tuning Kubernetes VPA parameters. We set up a barebones version of DeathStar application, where Nginx microservice with two replicas serves static content for the GET requests. We use one of the servers in the cluster as controller node and another as the worker node [7]. As the requests are light-weight, we ramp the workload up to 10000 rps, and see how quickly Kubernetes autoscales to catch up with the workload. In general, it has been found that default configuration for the Kubernetes VPA can hurt system performance [17].

For instance, with the default value of recommendation-margin-fraction = 0.15, Kubernetes will add a margin of 0.15 * computed CPU recommendation to allow the container to adapt to sudden changes in the workload. This ramp up can be quite slow at such high workloads. On the other hand, setting the parameter to a very large value that is close. The multi-arm bandits approach (ε-greedy) treats the parameter values as categorical choices and does not exploit continuity of the problem or correlations across the parameters. On the other hand, the deep RL method, DDPG, does exploit, but it has a much higher sample complexity.

A natural question is if we can tune the VPA parameters (the CPU parameters from Table 1) to help improve resource utilization. We use the throughput attained (over the 15-minute peak workload), with a penalty on the cpu-histogram-decay-half-life value as the reward function, to minimize wastage during off-peak hours.

Table 3 shows the best throughput achieved (mean and std. dev. over 5 deployments of the best parameters) and the number of samples needed by each of the methods to attain the best value. We find both BO and Bluefin converge, fairly quickly, yielding over 75% better throughput relative to the default configuration; Bluefin achieves the best throughput overall (statistically significant), an absolute improvement of 2.2% compared to BO. At convergence, Bluefin sets recommendation-margin-fraction to 1.5, and pod-recommendation-min-cpu to 850 millicores (see Table 1). This helps Kubernetes auto-scale the containers sufficiently quickly (compared to the default values of 0.15 and 250 millicores respectively) and serve the peak workload of 10000 rps. At the same time, Bluefin (and the other methods) converged to a small value (about 45 seconds) of cpu-histogram-decay-half-life, which is ideal for short bursts of workloads: Kubernetes evicts the worker containers right after the peak, thereby freeing up resources.

In what follows, we show how we can also tune the configuration parameters of microservices (running in containers) themselves, in order to improve application latency based on the workload.

#### Optimizing tail latency:

Microservices that are deployed in containers have multiple configuration parameters [9, 11–13, 18] that influence their performance. For instance, the number of threads of performance-critical microservices (e.g., compose-post-service in DeathStar) is known to significantly improve latency [58, 59]. We tuned 85 key numerical parameters of the microservices in DeathStar with P95 latency as the reward for all the methods.

Table 4 shows the best tail (P95) latency attained by each of the methods and the number of samples they took to achieve the same. This high-dimensional tuning setting clearly brings out the superiority of Bluefin over the popular techniques in terms of sample complexity. Even though there are 85 parameters, there are only a few parameters that critically influence the reward value. Indeed, Bluefin quickly converges to 19.5ms P95 latency (starting from 31.1ms, corresponding to the default values), with just 8 samples; in contrast, BO and ε-greedy algorithms take 3-5 times as many samples to attain a latency value that is close. The multi-arm bandits approach (ε-greedy) treats the parameter values as categorical choices and does not exploit continuity of the problem or correlations across the parameters. On the other hand, the deep RL method, DDPG, does exploit, but it has a much higher sample complexity.

We also show the iteration cost, i.e., the latency incurred through each round of tuning (which matters in deployments). The spread of the iteration costs for SelfTune indicates convergence close to 20ms. We deployed each parameter setting three times, and report the median value. Even though all the compared algorithms eventually converge to statistically similar latency values, they incur several orders of magnitude worse P95 iteration costs than Bluefin. This is a strong evidence of the effectiveness of Bluefin for tuning in live deployments, where the reward function can be highly ill-conditioned and can vary wildly in some regions of the explored parameter space.
<table>
<thead>
<tr>
<th>Metric</th>
<th>Bluefin</th>
<th>BO (GP)</th>
<th>ε-greedy</th>
<th>DDPG</th>
</tr>
</thead>
<tbody>
<tr>
<td>P95 latency (ms)</td>
<td>19.5</td>
<td>19.9</td>
<td>20.0</td>
<td>20.2</td>
</tr>
<tr>
<td># Samples</td>
<td>8</td>
<td>41</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>P50 iter. cost (ms)</td>
<td>20.5</td>
<td>23.3</td>
<td>29.2</td>
<td>20.6</td>
</tr>
<tr>
<td>P75 iter. cost (ms)</td>
<td>21.1</td>
<td>33.0</td>
<td>33.2</td>
<td>22.1</td>
</tr>
<tr>
<td>P95 iter. cost (ms)</td>
<td>28.3</td>
<td>76541.9</td>
<td>67640.3</td>
<td>148543.1</td>
</tr>
</tbody>
</table>

Table 4: Tuning parameters of microservices in DeathStar:
The second row indicates the number of samples (i.e., rounds) it took for each method to attain the best P95 latency reported in the first row. The last three rows show the spread of the latencies while tuning over 50 rounds.

8 Conclusion

This paper presents SelfTune, an RL-based framework using which cluster managers can tune parameters to improve cluster performance. We have deployed SelfTune with a large-scale task scheduler at Microsoft and show how it has improved overall system throughput. We show that SelfTune significantly improves system performance with experiments on Azure FaaS workloads, Kubernetes’s Vertical Pod Autoscaler, and the DeathStarBench microservice benchmark.

References

discretionary\char\hyphenchar\font{}){|}herence/parameters/.
[23] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In 14th USENIX Symposium on Networked Systems Design...


[61] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia,


A SelfTune’s Client API Implementation

We now formally present the syntax and the semantics of SelfTune’s client API (introduced informally in Section 3, and in Figure 1).

Creation. The Create API creates an instance of the parameter learning problem for SelfTune. This API allows optional arguments that encode domain knowledge for tuning the parameters:
(a) names for the parameters to learn,
(b) (optional) initial values for the parameters,
(c) (optional) constraints on the parameters to be tuned; the API supports range constraints (min and max), type constraints (e.g., isInt = TRUE if a parameter takes only integral values),
(d) (optional) for user-defined types, one could specify step-size (e.g., memory sizes in multiples of 64MB), or scale (e.g., logarithmic or linear).

```
string Create(string[] params,
            double initValue = 0,
            Dictionary<string, Constraints> constraints,
            Dictionary<string, Type> type)
```

Connection. The Create API sets up a data store instance in the back-end for tuning the specified parameters, initializes the necessary background services to maintain/update this store. A unique identifier to this store instance is returned by the call to Create. The Connect API connects a parameter learning instance to a SelfTune object.

```
void Connect (int problemId)
```

Note that if a store already exists (for the parameter(s) of interest), then the client can directly connect to the instance by referencing the unique identifier to the instance, as the store instances are persistent. This also enables multiple clients (distributed spatially and/or temporally) to query the latest decisions for as well as give feedback to the same learning problem.

Prediction. With the Predict interface, the developer can query the current values for the parameters. These values are decided by the learning algorithm (presented subsequently).

```
(int, double[]) Predict (string[] params)
```

Note that Predict returns a pair of values – a unique identifier which identifies the particular invocation of Predict, and the predicted value.
Feedback. As shown in Figure 1, the SetReward interface allows the client to specify a reward value. More generally, it allows the client to associate the value with a particular invocation of Predict:

```java
void SetReward(int invocationId, double reward)
```

The invocation id helps associate the reward to the parameters (and their values) returned by previous Predict calls — in particular, the reward value applies to all the parameters that were part of all the Predict calls since the last SetReward call, as noted in the following remark.

B Parameter Convergence

In this appendix, we provide graphs to give the reader an idea of how long it takes for RefreshCycle to converge in our experiments with WLM.

Figure 11: Convergence of RefreshCycle with SelfTune in the experiment using Cluster 1.

Figure 12: Convergence of RefreshCycle with SelfTune in the experiment using Cluster 2.

C Baselines

In this section, we discuss the implementation details of different baselines used in Section 7.

For Bayesian Optimization (BO), we used the skopt library [15] with gp_hedge as the acquisition function. The algorithm was initialized with the default configuration or with 3 random configurations (uniform sampling), and we reported the best results in Tables 3 and 4.

For Contextual Bandits (CB), we used the popular Vowpal Wabbit library [19]. Since the configuration space is too huge for the bandits formulation to handle, we restrict tuning to the four important parameters (memory limit parameter of the post-storage-memcached microservice, worker_processes and worker_connections parameters of the frontend microservice, memory limit of the post-storage-mongodb microservice) based on empirical observations and recommendations from prior work [58]. Since the algorithm expects discrete actions spaces, we suitably quantize the configuration space of each parameter. We use a `step_factor` for each parameter which yields `(upper_limit − lower_limit)/step_factor` number of quantized values per parameter. The value of `step_factor` is chosen such that the resulting (discrete) action space is not too large. After discretizing the four parameters in this fashion, we arrived at 24960 actions for the CB algorithm. We used the “explore first” strategy in the ε-greedy algorithm via the command `-cb_explore num_actions -first num_random`, which implies that the algorithm will (only) explore the action space with uniform probability for the first `num_random` iterations.

We implemented Deep Deterministic Policy Gradient (DDPG) [41] using PyTorch [48]. DDPG is a popular policy-based Reinforcement Learning algorithm used by prior works to tune system parameters [49,67]. We use the CPU and memory utilization of microservices on the nodes where microservices are running, workload volume (requests per second), number of clients, and request composition as state features. Both the actor and the critic networks consist of 1 hidden layer. The number of inputs to the actor layer is equal to the number of state features and the output is equal to the number of actions (i.e., parameters tuned). The input and the hidden layer use ReLU as the activation function while the output layer uses Tanh. For the critic network, the number of inputs is equal to the number of state features + the number of actions while the output is just 1-dimensional.

We use 1 step for each episode (to match how the iterations of the baselines and Bluefin proceed) and run the algorithm for 50 episodes. We let the algorithm explore random points for the first 25 episodes followed by 25 episodes where the explored configurations are chosen by the algorithm. To im-
prove the algorithm’s ability to explore, we add a Gaussian noise to the action chosen which is controlled by a parameter $\gamma$ ($\gamma = 0.1$ in our experiments). We update the model after every 5 steps. Once the 50 episodes are complete, we query the model to provide the best configuration for the initial state. We average the rewards over 5 such queries.