Auto-tuning elastic applications in production

Adalberto R. Sampaio Jr.[*] — Huawei Canada
Ivan Beschastnikh — University of British Columbia
Daryl Maier, Don Bourne, Vijay Sundaresen — IBM Canada

***Abstract*—Modern cloud applications must be tuned for high performance. Yet, a single static configuration is insufficient since a cloud application must deal with changes in workload, varying numbers of replicas due to auto-scaling, and upgrades to the environment and the application code itself. These dynamics can only be observed altogether during the application execution and affects different layers of the application stack. In this paper, we describe SmartTuning, a technique and tool to auto-tune cloud applications on the fly, improving resource utilization and performance under dynamic workloads.**

SmartTuning reacts to different workloads over time and automatically explores and adapts the application's configuration through Bayesian Optimization. SmartTuning searches for configurations that better use resources when the application is subject to auto-scaling and dynamic workloads. It minimizes the need for the operations team to instrument code or manually try out configurations in testing environments. Our evaluation of three industrial applications indicates that SmartTuning can, on average, improve application efficiency by 58% and reduce cost by 27%.

***Index Terms*—Auto-tuning, Elastic cloud environments, Microservices, Kubernetes**

I. Introduction

Cloud-based applications must use their resources efficiently since each minute of execution has a cost. Yet, these applications are deployed in complex stacks that may include containers, language runtimes, and numerous dependencies. Achieving high performance with minimal use of resources requires tuning across the stack, a task that is typically done by an operations team. Success in this context is usually a configuration that provides high performance most of the time.

Unfortunately, workloads change over time and applications are continuously re-deployed [1], [2]. This requires re-tuning. One strategy to overcome these issues is to *over-provision* the application and use an *auto-scaler* that starts new application replicas as the individual replica resources are exhausted and shuts down replicas when they are not needed. Since cloud pricing is proportional to resource usage, this strategy has two drawbacks. First, over-provisioning creates a coarse scaling granularity, so the application cannot efficiently handle low-volume workloads. Second, not all resources need to scale linearly along with the application and indiscriminately scaling all resources harms efficiency. Ideally, the application should scale both vertically and horizontally to address these issues. However, today's auto-scaling tools do not recommend the use of both scaling strategies simultaneously [3].

[*]Work done during Adalberto's postdoc at the University of British Columbia and IBM Canada during 2019–2021.

In summary, today's elastic cloud applications lack a solution to adjust their many parameters. For example, a Java application using the Open Liberty [4] application server running on the OpenJ9 JVM [5] has dozens of distinct parameters that can be tuned, each with many valid options to try and several workloads to address. These quickly explode to thousands or even millions of possible configurations, without even considering multiple replicas.

We introduce SmartTuning, which we designed to deal with a specific aspect of the auto-tuning problem: *tuning multiple production replicas of an application under varying workloads.* Conceptually, SmartTuning can tune any type of application stack that exposes configuration parameters and metrics. However, in this work, we limit our scope and only deal with web applications that can tolerate replicas being added and removed frequently. By contrast, data analytics applications (e.g., Hadoop and Spark), databases, and messaging middleware cannot easily accommodate frequent scaling. Short-lived services (e.g., Lambda functions) are also out of scope of SmartTuning, since they do not run long enough for a proper validation of the tuned configuration.

SmartTuning treats services as black-box functions that cannot be analytically modelled. We borrow the concept of automatic hyper-parameter tuning for parameter optimization [6] to tune general-purpose services. We use Bayesian Optimization (BO) [7] to find the values for tunable settings that optimize application efficiency modelled as an objective function. For example, BO finds values for connection timeout settings, garbage collection settings, and resource limits that increase application throughput while reducing cost.

SmartTuning samples configurations based on runtime monitoring and current workload. For each workload, SmartTuning explores configurations looking for the most suitable one for the current workload type, avoiding wasted resources due to the auto-scaling of over-provisioned replicas. Finally, SmartTuning automatically applies its best known configuration for each observed workload as the service's workload changes over time.

Workload classification is a broad topic [8] and we built SmartTuning with the assumption that such a classifier already exists. SmartTuning auto-tunes services to satisfy multi-objective requirements based on their workloads. It uses the workload classifier to learn about variations in the workload. Based on workload change notifications, SmartTuning uses statistical analysis to sample configurations that may improve the performance for each workload type while coping with auto-scaling that is controlled by an independent service, such as Kubernetes. SmartTuning performs all these processes *online* with no need for service downtime.

In summary we make the following contributions:

⋆ To the best of our knowledge, we are the first to propose and implement an automatic configuration tuning system for *live deployments in production.* Our system performs on-the-fly tuning and automatically re-deploys applications with better suited configurations as the workload changes.

⋆ We propose a design for tuning *elastic* applications with multiple replicas with the objective of minimizing resource contention and resource waste.

II. Related work

Tuning services is a well known problem that has been tackled by many groups [9], [10]. Most of these initiatives tune monolithic applications and single-layer services, such as databases [11]–[13], big data processors [14]–[16], message middleware [17], and application servers [18], to improve, for example, their performance [19]–[21], reduce energy consumption [22]–[24], or improve the safety of the tuned configurations [25]. There are fewer initiatives, such as BestConfig [26], ClassyTuner [27], and Auto-Tune [28] proposing a general tuner for different kinds of services as we do in this paper. But, none of them tune multiple layers of the application stack at runtime under a dynamic load and subject to a horizontal auto-scaler.

In previous work, tuning is done mostly off-line and assumes the same workloads [29]–[31]. Furthermore, tuning for cloud applications usually targets a single aspect, such as tuning the thread model [32], host nodes [33], the services placement [34], or a single service layer such as the JVM [35] and may only provide advice [36] rather than actively update the application. Zhang *et al.* [37] suggest the use of proactive techniques to improve application parameters; a strategy that we adopt in SmartTuning. Finally, [38] is a complementary approach to SmartTuning that seeks the best initial configuration for the application before deployment in production.

The fundamental difference between SmartTuning and these approaches is SmartTuning's focus on tuning *elastic* applications that use multiple replicas to dynamically auto-scale. Conversely, Autopilot [39] actively scales out vertically and horizontally to improve efficiency, although it does not act on an application's parameters like SmartTuning.

Fig. 1: Daytrader application stack.

SmartTuning also customizes the tuning search to the workload. Other approaches, like Fekry *et al.*'s [15] do not use workload to guide the tuning. Instead, they identify common workloads and create tuning policies for those workloads to avoid computing new policies at runtime.

III. Tuning challenges

We overview the tuning challenges by using a benchmark cloud application called Daytrader [40]. Daytrader (Figure 1) is a Java service for stock trading. Daytrader must quickly serve requests and deal with dynamic workloads. We use *dynamic workload* to refer to a set of concurrent clients that send a mix of requests to the application at a high rate, while the number of clients varies periodically.

Several runtime layers make up Daytrader: (1) Open Liberty application server handles connections to the database, (2) the JVM abstracts code execution and resource management, (3) a Docker container abstracts the file system and allocation of resources, and (4) a Kubernetes Pod provides auto-scaling, deployment mechanisms, and resource allocation.

Tuning multiple layers simultaneously. A cloud application layer exposes tunable settings, and a combination of settings across these layers controls application behaviour. Some settings have dependencies and cannot be tuned independently. For example, the number of open HTTP connections in Open Liberty is affected by Open Liberty's thread and connection pools, which rely on the heap size of the JVM. Therefore, the number of tunable setting combinations quickly becomes impractical for DevOps to explore.

Tuning during resource contention. An application with a fixed configuration and workload may perform inconsistently due to other applications running on the same cluster. For example, Daytrader might be co-located with other services. While the configuration and workload remain constant for Daytrader, it may suffer from resource contention.

Adjusting tuning to workload. Cloud applications must deal with workloads that change over time [8]. CI/CD will also induce changes in the application. A DevOps team has little time to master the most relevant tunable settings for a given workload and application version.

Tuning the application while scaling. Cloud applications add/remove replicas as the load increases/decreases. As the number of replicas changes, tunable settings must be adjusted. For instance, Daytrader may be scaled out to address a CPU-bound workload. The new replicas would increase the available memory, but also the number of database connections. These extra connections may surpass database capacity.

Tuning to satisfy multiple objectives. Cloud providers charge for consumed resources. Therefore, when a developer tunes an application, they usually want to improve its efficiency — *increasing performance while reducing cost.* Tuning becomes even more complex when considering cost.

IV. SmartTuning assumptions

Cloud services model. We assume a black-box application whose behaviour is observable through metrics that change as workload varies. We cannot know *a priori* the frequency and period of the application's workload patterns, nor the rules that govern the application's performance and resource usage.

Tuning duration. It takes some time for an application to reach steady-state after its settings change. We assume that the application we tune has a well-known stabilization interval, i.e., how long the application takes to reach steady performance from start-up. Usually, this information comes from performance tests done prior to a production deployment.

Horizontal autoscaling. We assume that the number of replicas may be controlled by a Horizontal Autoscaler (HA). If HA is used, we assume that SmartTuning is aware of the resource the HA is triggering on and we design SmartTuning to avoid tuning this resource.

Load balancing. SmartTuning tunes applications with multiple replicas. To ensure an unbiased analysis, SmartTuning assumes that each replica receives an identical mix and equal proportion of request traffic to the application.

Specifically, we assume that the tuned application sits behind a fair load balancer that guarantees that $1/(n+1)$ of traffic is sent to each of the $n+1$ replicas: one training replica for experimenting with configurations, and n production replicas using a reliable configuration. This allows us to replicate the production traffic volume so that it is observed by the training replica.

Replica representativeness. We assume that any one replica is representative of the other replicas given identical volume of requests. This allows us to assume that a good configuration found for our training replica can be promoted to production and attain comparable performance.

Ignoring connected microservices. SmartTuning tunes microservices without regard for connected microservices or applications which can shift the performance bottleneck [41]. For example, tuning one service may harm the efficiency of other, chained, services. We plan to deal with bottleneck shifts in our future work.

V. SmartTuning background

SmartTuning's goal is to optimize cloud services based on their workloads to satisfy an objective set by the DevOps team. SmartTuning observes the application[1] and its varying workloads (via the workload classifier), searches among configurations, and determines how well each new configuration satisfies the target objective.

A. Application modeling

The cloud application is the function we are aiming to tune. We consider the application as a black-box function $f : C, W \rightarrow \mathbb{R}$ that maps a configuration $c \in C$ and a workload $w \in W$ to a score value. The DevOps team encodes their tuning goal into an *objective function* $\hat{f}$, which is expressed using only the runtime-monitored observables. SmartTuning uses a numeric approach to search for values that make $\hat{f}$ yield high scores.

An *observable* is any information exposed by the application or environment that can be monitored. SmartTuning allows the developer to use arithmetic operators to freely combine observables to describe $\hat{f}$. Classical observables are CPU/memory consumption, availability, throughput, and response time. SmartTuning also interprets tunable setting values as observables. For example, if the DevOps team wants to minimize their cost to run the application in a public cloud, and the provider charges \$$k$ per CPU hour and \$$m$ per gigabyte hour, then they can use the objective function in Equation 1:

$$\hat{f} = \frac{throughput}{k * CPU + m * memory} \tag{1}$$

By convention, SmartTuning minimizes $\hat{f}$. To maximize $\hat{f}$, we minimize $-\hat{f}$. Ultimately, SmartTuning aims to optimize an application by searching for a configuration c^* that minimizes $\hat{f}$ for a given workload w: $c^* = \arg\min_{c \in C} \hat{f}(c, w)$

SmartTuning allows composition of objective functions using arithmetic and max and min functions to combine fractions p/q, where each fraction expresses one aspect of the overall objective. The parameters to maximize go into p, and those to minimize go into q. Independent parameters are sums of fractions, while dependent parameters are modelled as a product. Care should be taken to normalize and weigh each factor according to the desired goal. Exponents and logarithms can also be used. For example, we can model the SLO max throughput with memory consumption $< 80\%$, as in Equation 2:

$$\hat{f} = \frac{-throughput}{1 + \max(0, memory\ utilization - 0.8)} \tag{2}$$

[1]We use service and application interchangeably, although the term application primarily refers to the collection of replicas.

B. Bayesian Optimization

Every tunable setting is defined as a range of values that bounds valid regions. We call the list of all tunable settings a *configuration*, and the list of tunable setting ranges a *search space*. Bayesian optimization [42] is an enhancement of random search [43], [44] that uses Bayes' theorem to sample the next value based on previous observations and to speed up the convergence to a global optimal. In general, Bayesian optimization converges faster than Random search [45].

Bayesian optimization samples several configurations from the search space and checks which one has the greatest chance to yield a higher objective function score. We choose the Tree Parzen Estimator (TPE) [45], [46], a specialization of Bayesian optimization, due to its ability to handle conditional search spaces [44]. Conditional search spaces bring more flexibility to describe complex relationships between tunable settings in cloud applications. This approach deals with categorical values using one-hot encoding, although this is not the most efficient approach [47].

C. Workload classification

In this paper our aim is *not* to contribute a new workload classifier. We assume that one exists, based, for example, on previous work [8], [48]. SmartTuning requires a classifier $h : \mathbb{W} \rightarrow \mathbb{K}$ that maps a workload $w_{a,b} \in W$ observed in a time window (a, b) to a class $k \in \mathbb{K}$. The classifier observes and labels the application's workload and signals if the current workload has changed. In our evaluation we use two simple workload classifiers. Our classifiers identify workloads based on the number of requests arriving to the application and the mix of request types [49].

VI. SmartTuning design

The architecture of SmartTuning follows the MAPE-K framework [50], which defines four phases in a feedback loop: monitoring, analysis, planning, and execution. These phases share a knowledge-base that is built up during the tuning process. In SmartTuning's design, we merged the analysis and planning into a phase we call *tuning analysis*. To simplify management and to smoothly integrate with modern deployment environments, we designed and implemented SmartTuning as a custom Kubernetes controller.

Figure 2 overviews our design. At a high-level, SmartTuning tunes an application by repeating three steps. First it iterates by sampling an experimental configuration, applying it on a newly deployed training replica, and running the *training replica* along with the *application in production* (Figure 2). After t time units, SmartTuning scores the training and production configurations using the objective function f. Second, after k iterations, SmartTuning checks if the best configuration so far maintains a good score over a longer time period. Third, and finally, SmartTuning promotes the best configuration found so far from training into production, but only if the score of this configuration is higher than the current configuration in production. The tuning ends when SmartTuning reaches N iterations. Table I lists the set of parameters that a DevOps engineer provides to SmartTuning. The table also lists default parameter values that we have found to work well in practice.

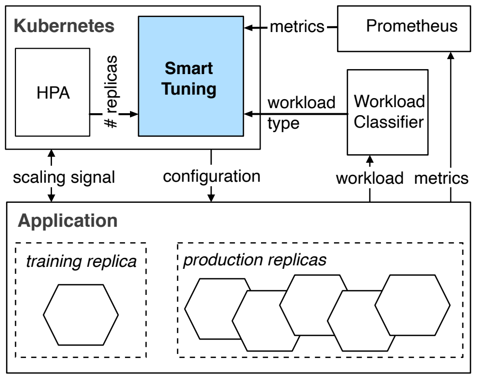


Fig. 2: SmartTuning architecture.

Parameter	Default	Description
N	50	Stopping criteria
k	10	Number of search iterations
t	App dependent	Iteration length
$\hat{f}$	Goal dependent	Objective function
Monitoring queries	App dependent	Cfg. Tunables

TABLE I: Parameters for configuring SmartTuning.

A. Knowledge base

The knowledge base stores the *search space*, *workload types*, and *tuning trials.*

Search space. The search space is a multi-dimensional structure that maps an application's parameters (tunable settings) to a range of values. During tuning, SmartTuning samples a single value for each tunable.

The search space captures dependencies between tunables. SmartTuning relies on the operation team to define these dependencies. A tunable can depend on another tunable in any of the application's layers. We use these dependencies to provide higher configuration reliability. For example, the JVM's heap size should not exceed the memory available to its container. SmartTuning filters out tunable setting combinations that do not satisfy dependencies, reducing the set of configurations to try. In our future work we plan to detect dependencies between tunables automatically [51]–[53].

Workload types. The knowledge base also stores workload types. These are generated by the workload classifier and are associated with each configuration experiment.

Trials. The knowledge base tracks all attempted configurations. A trial is a triplet that links the list of tunables in the configuration, the score calculated by the objective function, and the workload observed during a tuning iteration.

We assume that workload types eventually repeat, and to properly track a workload's behaviour a trial holds statistics of the relevant metrics, such as mean, median, and standard deviation. These metrics are updated any time the same tuple of workload and configuration repeats.

During tuning, each trialed configuration belongs to either the *nursery* or is considered *tenured*. The nursery holds recent trials until SmartTuning progresses to select a better candidate to be promoted into production. These promoted trials are then stored as tenured. After tuning ends, SmartTuning uses only tenured trials to keep the application tuned as the workload changes.

B. Monitoring metrics and workloads

At runtime, SmartTuning observes metrics exposed by the application to evaluate the objective function $\hat{f}$.

The workload classifier runs along with SmartTuning and identifies the inbound workload on the fly. SmartTuning maintains the state of tuning for several workload classes, relying on the classifier to detect and report the observed workload over time. SmartTuning creates one tuning context per workload class, and tracks all configuration and application metrics related to the workload in this context. The state management of a tuning context is delayed until the end of the current iteration it is processing. At the end of an iteration, SmartTuning checks with the workload classifier to see if the current workload has changed. If so, SmartTuning either creates a new context for a first time identified workload, or adds the results to the context corresponding to the already observed workload.

If there are changes to the workload type in the middle of the iteration, SmartTuning only checks at the end of the iteration which workload has lasted the longest during the iteration. If the longest workload does not match the current context, SmartTuning switches to the appropriate context. Otherwise, SmartTuning continues the tuning without switching tuning contexts.

C. Tuning analysis

Algorithm 1 details how SmartTuning tunes an application in three steps: *search*, *reinforcement*, and *probation*.

Searching. At every searching iteration, SmartTuning samples a configuration, applies it to the training replica at the end of the iteration, and saves it into the nursery. SmartTuning iterates at least k times to search for a configuration that achieves a score higher than the configuration currently used by the production replicas. Then, SmartTuning ranks all the trials in the nursery by their score and workload and selects the trial with the highest score that matches the current workload observed in the application (Algorithm 1 line 16).

During the first k iterations, the *exploration phase*, SmartTuning samples the configurations at random to

Algorithm 1 Tuning algorithm

```
 1: global production, training ← {workload, score, cfg}
 2: procedure TUNING($N, k, t$, workload)
 3:     nursery, tenured ← ∅
 4:     while $N \geq 0$ do
 5:         bestCfg ← searching($k, t$, workload)
 6:         progress ← reinforcement($k, t$, bestCfg, workload)
 7:         if progress then
 8:             promoted ← probation($k, t$, bestCfg, workload)
 9:             if promoted then
10:                 nursery ← ∅
11:             end if
12:         end if
13:     end while
14: end procedure
15:
16: procedure SEARCHING($k$,t, workload)
17:     bestCfg ← nil
18:     for $k$ iterations or bestCfg = nil do
19:         training ← sampleCfg()   ▷ updates training replica
20:         trial ← monitoring($t$)
21:         nursery ← nursery ∪ trial
22:         bestCfg ← lookupCfg(nursery, production, workload)
23:         $N \leftarrow N - 1$
24:         scheduleIfWorkloadHasChanged(workload)
25:     end for
26:     return bestCfg
27: end procedure
28:
29: procedure REINFORCEMENT($k, t$, bestCfg, workload)
30:     training ← bestCfg              ▷ updates training replica
31:     for $k/3$ iterations do
32:         trial ← monitoring($t$)
33:         updateScore(trial, nursery, tenured)
34:         $N \leftarrow N - 1$
35:         scheduleIfWorkloadHasChanged(workload)
36:     end for
37:     return training.score > production.score
38: end procedure
39:
40: procedure PROBATION($k, t$, bestCfg, workload)
41:     lastCfg, lastScore ← production       ▷ save previous cfg
42:     production ← bestCfg    ▷ updates production replicas
43:     for $k/3$ iterations do
44:         trial ← monitoring($t$)
45:         updateScore(trial, nursery, tenured)
46:         $N \leftarrow N - 1$
47:         scheduleIfWorkloadHasChanged(workload)
48:     end for
49:     if production.score < lastScore then
50:         production ← lastCfg     ▷ reverts cfg in production
51:         return FALSE
52:     end if
53:     tenured ← tenured ∪ trial
54:     return TRUE
55: end procedure
```

cover more of the search space. After these k iterations, SmartTuning changes to the Bayesian Optimization (BO) approach, or the *exploitation phase*. During exploitation, BO considers the search space region that the exploration phase has yielded to find the best configuration.

A larger k value promotes *search space exploration*, while a smaller k value promotes *exploitation of a bounded region.* In our experiments, we found $k = 10$ to be a reasonable value.

While SmartTuning is searching, two or more consecutive configurations with similar tunable settings and scores might be found. SmartTuning handles this by only selecting the configuration with the highest score, and deleting the remaining ones in the nursery after each promotion.

At the end of this step, SmartTuning pauses this loop and moves to *reinforcement.*

Reinforcement. SmartTuning uses this loop to ensure that the selected configuration maintains its high score when run for a longer period. For the subsequent $k/3$ (Algorithm 1 line 31) iterations SmartTuning monitors the score of the configurations at both the production and the training replicas. If at the end of this loop, the median of the score of the training replica is greater than the production replica, SmartTuning moves the configuration in training to the *probation step* for a final quality check. Otherwise, this configuration is discarded and SmartTuning returns to the *searching step* for k more iterations.

Probation. During this loop, SmartTuning configures both the training and production replicas to use the same configuration from *reinforcement* (Algorithm 1 line 40). This lasts for $k/3$ iterations. Probation is used by SmartTuning to ensure that the candidate configuration works as expected when scaled out to all production replicas. This helps detect problems that show up at scale, e.g., contention on shared backend resources like a database.

If after the $k/3$ iterations SmartTuning observes that the production service does not achieve better performance, it reverts the configuration. Alternatively, SmartTuning promotes the configuration as tenured, keeps it in production, and erases the nursery region.

After the probation loop SmartTuning transitions back to the *searching step* where it will again search for better configurations.

D. Execution

When SmartTuning decides to update a service in training or production, it serializes the new configuration and updates the Kubernetes configuration files that hold the environment variables for the replicas. These changes will trigger Kubernetes to restart all pods with the new values.

If a new workload type appears, SmartTuning starts a new tuning session for it, using the latest configuration as the initial configuration for the session. If a new application version is deployed, SmartTuning resets the tuning, using the latest tenured configurations applied to the application as its starting point.

E. Implementation

We built SmartTuning as a Kubernetes controller in Python. It uses a black-box workload classifier and Prometheus to observe the application and its metrics.

Knowledge base. SmartTuning defines a search space that lists several tunables. A tunable is either numeric continuous, numeric discrete, or categorical, e.g., a list of strings. We express dependency between tunables using a direct acyclic graph (DAG). We implemented the search space as a Kubernetes Custom Resource Definition (CRD) [54]. All tunables are exposed through environment variables in Kubernetes' Config Maps or Resources Limits from a Pod Specification.

Monitoring. At runtime, SmartTuning can observe any metric exposed by the application through Prometheus [55] to evaluate the objective function. SmartTuning considers the first third of the iteration length to be a warm-up interval.

Execution. The update process uses Kubernetes probes [56] and a custom rolling update strategy [57] to restart the application replicas without downtime. We customized this strategy to eliminate unavailable pods and to cope with workload surges. We briefly over-provision the replica set during an update to ensure that fresh replicas will not crash due to a sudden burst of requests while it is warming up. Finally, we install custom PreStop hooks in Kubernetes to enforce a delay during pod exclusion.

VII. Evaluation

Our evaluation targets three research questions:

1) RQ1: How efficient are the configurations that SmartTuning finds as the application is scaled up and down and the application workload varies?
2) RQ2: What is the cost of tuning? And, how long does it take before a configuration found with SmartTuning starts to pay off?
3) RQ3: How much does an application's performance degrade while SmartTuning is doing the tuning?

In our experiments we use Equation 3 as the objective function. We measure memory in GBs and CPU in vCores. This equation aims to *maximize the application efficiency by increasing its performance while reducing the processing time, and resource utilization, which reduces the cost of deployment.* SmartTuning calculates the total throughput and resource utilization of each training and production replica-set, and the average of processing time of all requests per replica.

We have experimented with alternative objective functions, including those that minimize cost and maximize client-observed performance (see Section VII-F). Throughout, we have found that SmartTuning's search process is objective-agnostic. Ultimately, choosing the right objective function is a matter of requirements. By collaborating with our clients we found that Equation 3 strikes a good balance between the cost of cloud resources and application performance.

$$\hat{f} = \frac{1}{1 + response\ time} \times \frac{throughput}{CPU + memory} \quad (3)$$

We run all experiments in a Kubernetes cluster on Azure with four nodes, each configured with 16 vCpus and 56GB of RAM. The experiments run with the Horizontal Pod Autoscaler [58] (HPA) enabled to scale the application whenever the CPU consumption of the application is above 50%.

A. Evaluated applications

We used three open source cloud applications: Quarkus-HTTP-Demo (QHD) [59], AcmeAir [60], and Daytrader [40]. RedHat uses QHD to evaluate its Quarkus framework, and IBM uses AcmeAir and Daytrader internally to evaluate their products. Daytrader is the most complex of the three.

The three applications implement the traditional model-view-controller architecture in Java. AcmeAir and Daytrader use the Open Liberty application server [61] while QHD uses Quarkus [62]. Each application has one attached database: AcmeAir uses MongoDB [63], Daytrader uses DB2 [64], and QHD uses PostgreSQL [65]. QHD is a generic REST service to expose database CRUD (create, read, update, delete) operations. AcmeAir is an implementation of an airline website. It allows users to login, update profiles, and manage flight bookings. Finally, Daytrader is an application built to simulate an online stock trading system. It allows users to login, view their portfolio, lookup stock quotes, and buy or sell stock shares. Daytrader is our most complex application: it has an asynchronous messaging communication mechanism and provides two different endpoints implemented in distinct frameworks. All applications use the OpenJ9 [5] JVM and run inside Docker [66] deployed in a Kubernetes cluster.

All applications are deployed with default configuration from their repositories. We use these defaults as baselines in our evaluation to compute the efficiency of the configuration that SmartTuning finds after 50 iterations[2] for each observed workload. We also use these to evaluate the pay-off in tuning each application.

We tune a different mix of parameters for different layers of each application. Common to all of these are container memory limits, JVM heap size, and the size of thread and connection pools. Table II summarizes our SmartTuning experimental settings. We selected the parameters to tune based on the recommendation of IBM engineers who know these applications well.

B. Auto-scaling and workload classifier setup

For all applications we allocate one vCPU per pod. We use HPA to handle the CPU needs of the application, scaling it up/down. We follow HPA documentation in avoiding vertical scaling based on the HPA metrics. As a result, SmartTuning does not tune the CPU allocation in each Pod.

	QHD	**AcmeAir**	**Daytrader**
# parameters	7	12	22
iteration length	5min	10min	20min
warm up period	100s	200s	400s
total iterations per workload	50	50	50
number workloads	3	3	5
HPA max replicas	8	8	16

TABLE II: SmartTuning evaluation details.

We run experiments with two workload classifiers.

Workload classes based on volume. The default classifier identifies workloads based on the number of simultaneous clients (sessions) accessing the application. For each application we used three workloads to exercise CPU needs, so that HPA creates/deletes replicas over time. We use Jmeter [67] to periodically vary the number of virtual clients sending requests to the applications (every two hours for Daytrader and every 30 minutes for the other applications). For AcmeAir and QHD, we set the workloads to be 50, 100, and 200 simultaneous clients, for Daytrader the workloads set is 5, 10, and 50 clients[3]. We use `app_workload` to denote an app running with a specific workload. Across all settings, each client used a *thinking time*, or delay, that was sampled for each request from a Poisson distribution with $\lambda = 150ms$.

Workload classes based on request endpoints. For Daytrader we use a second classifier that identifies the workload based on the set of URLs the clients request. Daytrader has two URL sets that route requests to two Java presentation frameworks: Jakarta Server Pages (JSP) [68], and Jakarta Server Faces (JSF) [69]. Our second classifier determines if the current flow of requests targets JSP or JSF endpoints.

C. Tuning results (RQ1)

How efficient are the configurations that SmartTuning finds as the application is scaled up and down and workload varies?

SmartTuning *improves application efficiency by 59% on average. It can find optimal configurations even under dynamically changing workload and scaling conditions.*

Figure 3 shows the variation in the number of replicas throughout the tuning (time runs left to right). We can see that for all applications, HPA scales the number of replicas up and down repeatedly, especially during workload transitions.

Figure 4 shows the improvement achieved by a configuration found with SmartTuning for all app_workload

[2]We found that using 50 as the total iterations per workload balances the number of configurations SmartTuning tries and the overall duration of our experiments. In practice, this stop criteria can be set to reflect the needs of the operations team.

[3]These values are typically used to benchmark these applications by performance engineers at IBM.

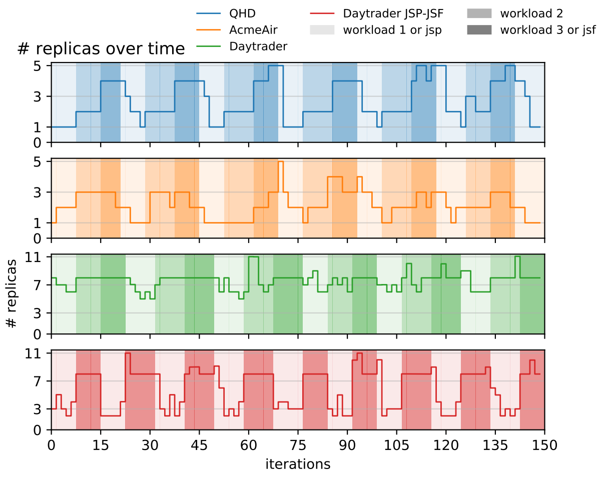


Fig. 3: Number of replicas over time per application. The background shading identifies each workload.

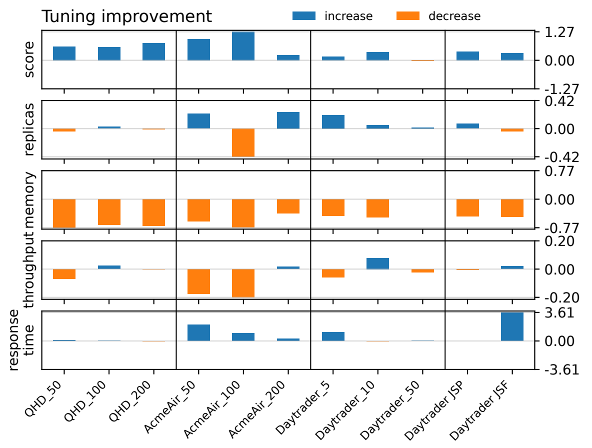


Fig. 4: Comparison of initial and tuned configuration for each application and workload. Top row is the objective function score. The other rows list specific resource types. Each bar represents the average over 10 iterations. A value of 0.0 is normalized to the initial configuration. Blue bars are above 0, while orange bars are below 0.

scenarios. For each condition SmartTuning found at least one better configuration. In all cases, the amount of memory allocated to the application was the parameter with the largest impact on the tuning score. SmartTuning reduced at least 50% of total memory allocation for all applications. For example, for QHD SmartTuning reduced memory by 75% from the initial configuration. In all cases, all applications started with 1GiB per replica, and the memory reduction happened even when the new configuration required the horizontal auto-scaler to create more replicas than the initial configuration.

In general, extra replicas were a consequence of vertical scaling that SmartTuning promotes for some parameters. Hence, the horizontal auto-scaling creates new replicas to avoid harming application performance. In these cases, efficiency is not an issue due to the smaller memory footprint.

In some cases the tuned configuration caused the application to have a higher response time. Despite the high relative value for this metric, the absolute values are low. For instance, Daytrader JSF has a response time of 3ms with the initial configuration and 17ms with the tuned configuration. This outcome is *expected*, since we did not set any weight in our objective function to prioritize response time over other metrics, such as resources consumption.

QHD. SmartTuning found configurations that improved QHD by 66% on average for all workloads with the minimum improvement for workload 2 (59%) and the maximum for workload 3 (78%). The improvement in throughput was $< 2\%$ and the number of replicas did not change, while the memory allocation was cut by 75% and the response time increased by 3% on average.

AcmeAir. SmartTuning found configurations that improved AcmeAir by 81% on average for all workloads with the minimum improvement for workload 3 (24%) and maximum for workload 1 (127%). On average AcmeAir had its throughput reduced by 12%, and its response time increase by 110%. Although the large memory reduction of 58% compensated for the performance reduction, specially for workload 2, which had a boost in its score due to a reduction in memory (77%) and replicas (-3 units), which helps to explain the high score.

Daytrader. SmartTuning found configurations that improved Daytrader by 43% on average for workloads 1, 2, and 3; and, by 35% on average for workloads JSP and JSF. In both workload types the main improvement was the memory (50% across all workloads). Workloads 3 and JSF also had a boost due to a throughput improvement of 8% and 2%, respectively.

For Daytrader, JSF improvements in replicas, memory, and throughput, compensated for the 4x response time degradation. JSF has a larger memory footprint than JSP, and the reduction in memory for this workload frequently triggered the JVM gc, which impacted response time. As SmartTuning performs multi-objective tuning, we can expect trade-offs: improvements in some objectives and degradation in others. If the application requires strict optimization, the objective function must capture this, see Section V-A.

Overall, these experiments show that SmartTuning can find better configurations for all applications under most workloads, even while HPA actively varies the number of replicas during tuning. This answers RQ1.

D. *Cost of tuning (RQ2)*

What is the cost of tuning? And, how long does it take before a configuration found with SmartTuning starts to pay off?

SmartTuning *reduced application cost by 27% on average as compared to the initial configuration. The cost break-even point was reached after at least 2.08 hours and at most 34 hours.*

To answer RQ2 we calculated the break-even time with Equation 4. This equation calculates the time before there is a tuning pay off, where *cost(x)* is the cumulative cost of running a set of replicas x over time. *cost(prod)* calculates the cost of running the production replica while it is adjusted with better configurations found during tuning. *cost(train)* is the cost of the training replica that SmartTuning updates at every iteration. *cost(untuned)* is a virtual replica set that emulates the application if it remains with its initial configuration during its entire execution. *Untuned* replica set is an extrapolation from the initial 10 iterations of the *production* replicas, during these iterations it is guaranteed that the application will retain the same initial configuration.

$$\int_0^{T_{\text{payoff}}} [cost(untuned) - cost(prod) - cost(train)]\ dt = 0 \tag{4}$$

Figure 5 depicts the break-even time for all applications and workloads. The no tuning line (green) is the cumulative *cost(untuned)* replicas set. We use least squares fitting [70] to extrapolate the initial 10 iterations of the replicas in production to model an entire execution. The production line (blue) is the actual cumulative *cost(prod)* of replicas running and being tuned in the system. Training line (orange) is the cumulative *cost(train)* of the single training replica. Finally, the payoff line (red) is the tuning payoff.

We calculate the cost of each replica according to the Azure VM cost. In our experiments we used instances of DS5_V2 with 16 vCpus and 56GB ram at the cost of \$1.12/hour [71]. Thus, the hourly cost for CPU and memory follows: $16 \times \text{cpu} + 56 \times \text{memory} = 1.12$. At every iteration we calculated the cost of all vCPU and memory allocated to all pods in the different contexts – untuned, production, and training.

In Figure 5 the dashed portion of all the lines are extrapolations to show when the payoff is likely to be positive for those cases where it does not turn positive within the 50 tuning iterations. For the training line, after the tuning ends, there is no more training replica running so their cost remains constant from that point.

The break-even point happens before the end of 50 iterations in five out of 11 workloads. The break-even point of Figure 5d workload 100 happens after +6% of the total tuning duration. In the other five experiments, the break-even point happens in between +20% and +88% of the tuning duration, with Figure 5c workload JSP having the longest break-even point.

On average, after SmartTuning finished tuning, it has saved 27% from the cost of running the application as compared with the application running with its initial configuration. Daytrader JSP had the smallest cost improvement of 10% (Daytrader JSP), while AcmeAir_100 had the maximum reduction of 49%. The applications with the largest cost saving for all workloads was QHD (41%), followed by AcmeAir (29%), Daytrader JSP-JSF (21%), and Daytrader (16%).

Experiment	**Cfgs found**	**Improvement**		**Tuning**	
		Efficiency	*Cost*	*Duration*	*Pay off*
QHD_50	1	60.28%	39.44%	5.25h	6.33h
QHD_100	4	58.77%	43.17%	4.83h	5.16h
QHD_200	3	77.55%	41.83%	4.41h	2.08h
AcmeAir_50	3	93.40%	13.83%	5.25h	9.58h
AcmeAir_100	3	126.56%	48.93%	4.75h	2.75h
AcmeAir_200	4	24.27%	23.22%	4.42h	6.33h
Daytrader_5	2	62.11%	14.34%	14.33h	18h
Daytrader_10	2	24.69%	16.36%	14.66 h	14h
Daytrader_50	3	40.82%	18.32%	14.6 h	13h
Daytrader_JSP	3	38.37%	10.15%	18h	34h
Daytrader_JSF	1	31.33%	32.05%	18.33h	6.33h

TABLE III: Number of configurations promoted to production, efficiency (objective fn.) and cost improvements, tuning overhead, and payoff

Table III summarizes the tuning time, the improvements achieved, and also how long it would take for tuning to pay off for each application_workload.

E. Performance degradation (RQ3)

How much does an application's performance degrade while SmartTuning is running?

On average, the tuning process resulted in a 3.5% degradation in throughput for some workloads and a 7% increase in response time for others. Over-provisioning during configuration handover and the reinforcement and probation steps mitigated performance degradation and avoided application downtime.

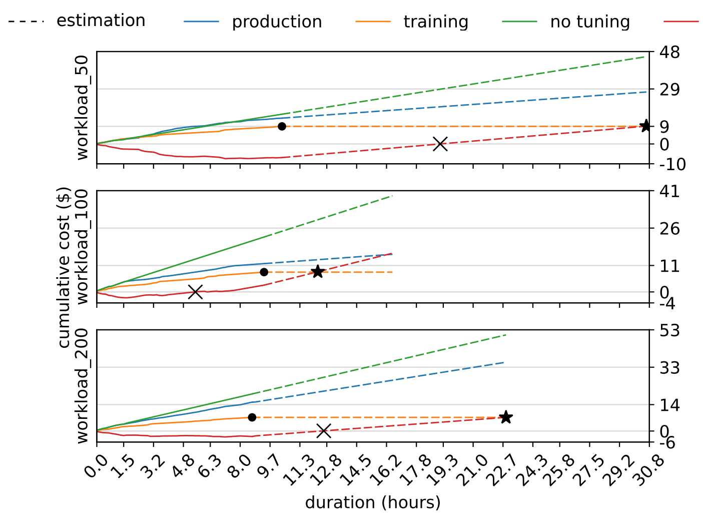
In all experiments we carefully monitored the training and production replicas. We did not observe any downtime. In general, we observed negative variation of 3.5% on average for throughput, with QHD_100, AcmeAir_200, Daytrader_50, and Daytrader_JSF with positive variation. Similarly, we observed a higher response time median (7%) for most applications, with only QHD_200 and Daytrader_50 having lower response times.

We believe that our strategy of over-provisioning and actively monitoring probation configurations in production allows for a robust and safe tuning process that avoids downtime and performance degradation.

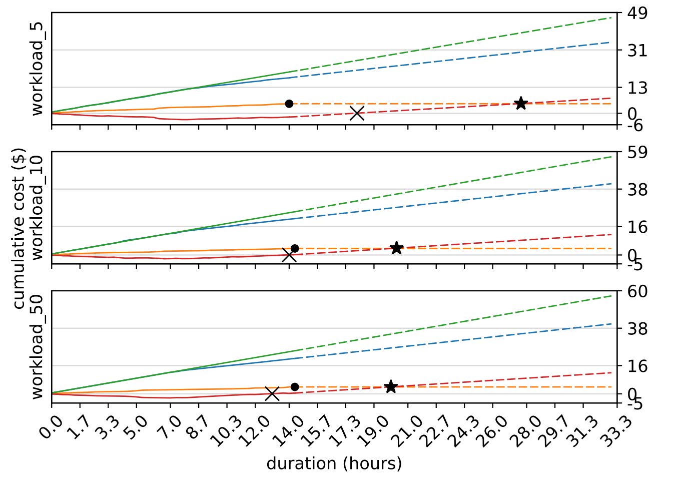
F. Tuning with other objective functions

We informally validated that SmartTuning generalizes to other objective functions. We experimented with using the functions and applications listed in Table IV. For these experiments we manually varied the workload request volumes and used tunables that directly impacted just the features in each objective function. Below we briefly report on our experience with these alternative objectives.

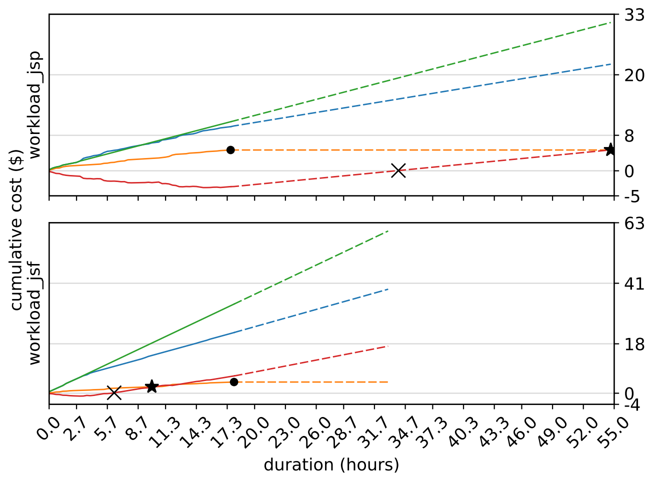
We observed that tuning an application with a single target performance metric, such as throughput in FN 1, only briefly improved application performance. In the long



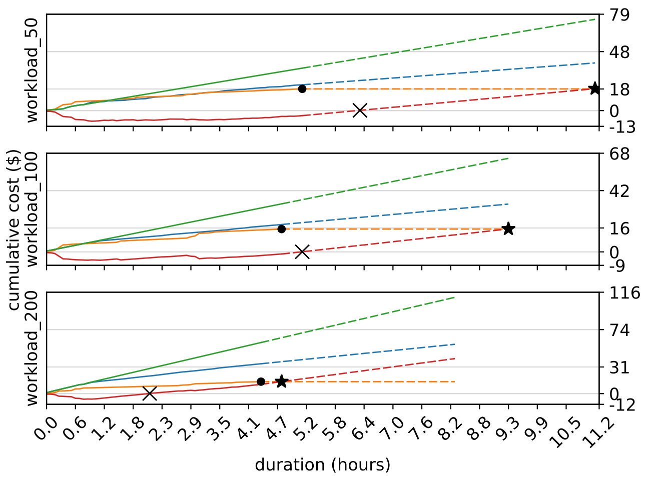
(a) AcmeAir



(b) Daytrader



(c) Daytrader JSP-JSF



(d) QHD

Fig. 5: Break-even time. The dashed lines are the estimated cost of the application running after the experiment.

run, some tunables were under- or over-adjusted and this harmed other aspects of the application, such as memory usage.

For simpler applications like QHD, function FN 2 is a reasonable choice, since it covers both memory allocation and performance, two key features that DevOps engineers monitor and tune. But, for more complex applications, like AcmeAir and Daytrader, FN 2 is limited and SMARTTUNING hardly finds better configurations with this objective.

We found that for complex applications like Daytrader, the more tunables that are available to SMARTTUNING the better. Due to the internal async communication in Daytrader, we experimented with FN 3 to increase its volume of requests processed without inflating response time. We did not reach our aim with this function because the tuned memory allocation would cause the application to saturate, with long pauses due to garbage collection. To handle this, we used FN 4, which considers memory saturation. In this case, SMARTTUNING could find configurations that improved application performance without harming memory allocation.

#	**Objective function $\hat{f}$**	**App**
FN 1	throughput	All
FN 2	$\frac{\text{throughput}}{\text{mem}}$	All
FN 3	$\frac{\text{throughput}}{(1+\text{resp. time})}$	D.trader
FN 4	$\frac{1}{1+\max(0,\text{mem util}-0.5)} \times \frac{1}{1+\text{resp. time}} \times \frac{\text{throughput}}{\text{mem + cpu}}$	D.trader

TABLE IV: Alternative objective functions we evaluated.

G. *A closer look at tuning Daytrader*

We looked more closely at Daytrader under workloads based on request endpoints (JSP and JSF) to better understand the tuning process. We consider experiment scenarios in Table V.

Scenario #	Description
0-[JSP|JSF]	baseline as described in Section VII-C
1-[JSP|JSF]	in a shared environment with contention
2-[JSP|JSF]	random sampling
3-[JSP|JSF]	no dependencies among tunables
4-[JSP|JSF]	no memory vertical scaling
5-[JSP|JSF]	tuning no parameter related to memory
6-[JSP|JSF]	no thread pool tuning
7-[JSP|JSF]	combining scenario 5 and 6
8-[JSP|JSF]	add weight=10 to response time in Eq. 3
9-[JSP|JSF]	add weight=1000 to response time in Eq. 3

TABLE V: In each scenario the application was under two workloads JSP and JSF that hits different APIs in the applications. We use *"#-[JSP|JSF]"* for the scenario number and the type of workload, e.g., "0-JSP" is the application in scenario 0 with the JSP workload.

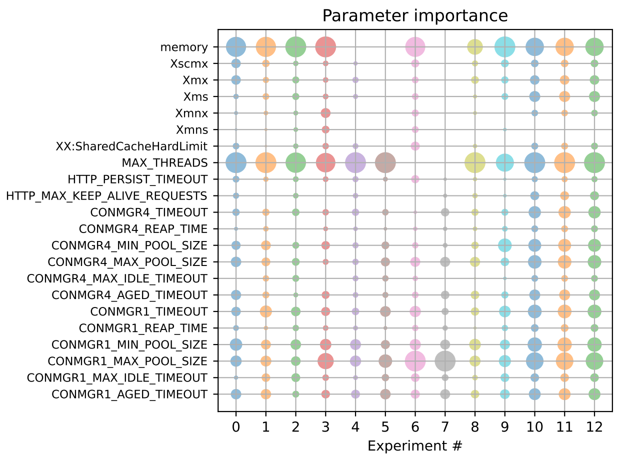


Fig. 6: Parameter importance. The experiments 0 to 9 are summarized in Table V. The experiments 10, 11, and 12 are virtual, and combine the values achieved from other experiments showing the overall importance of the parameters when workloads are not specified (#10), and their importance for all JSP (#11) and JSF (#12) workloads.

Figure 6 plots the importance of each tuned parameter for each scenario (calculated with XGBoost[4].) Overall, the container memory and the max thread pool size (`MAX_THREADS`) had the highest impact on the tuning. When omitting these parameters during tuning, the connection pool to the application database (`CONMGR1_[MAX|MIN]_POOL_SIZE`) becomes the most important parameter. The importance of the other parameters are more or less uniformly distributed.

No experiment could optimize all metrics of interest (memory, response time, and throughput) modeled in Equation 3, as response time is negatively correlated to other metrics (Table VI). Thus, any non-trivial objective function requires tuning to make trade offs.

[4]We used the default configuration for XGBoost [72]

	Score	Memory	Throughput	Resp. Time
Score	1.00	0.20	-0.72	0.26
Memory	0.20	1.00	-0.02	-0.20
Throughput	-0.72	-0.02	1.00	-0.41
Resp. Time	0.26	-0.20	-0.41	1.00

TABLE VI: Pearson correlation of the variables in Equation 3, data from all experiments in Table V.

VIII. Limitations and Future work

Our current implementation assumes that services in a multi-service application can be tuned individually. This is not true in general as there may be dependencies among services that would require simultaneous tuning. Our results do not generalize to this broader class of applications.

We used artificial workloads in our experiments to demonstrate that SmartTuning can tune to specific workloads. However, real workloads will have other features that we cannot claim to reproduce or match.

Finally, the performance of some services may be dominated by external stateful services, such as a database. As the database fills up over time, the application's performance will change. We explicitly avoid this issue in our evaluation by minimizing the database size during experiments. A general approach would require extending the objective function with database metrics.

Our next steps include enhancing SmartTuning to handle multi-service applications and stateful systems. We aim to explore new techniques for identifying interdependencies among tunables and services and incorporating the dynamics of stateful services in the objective function to account for changes in the database during the tuning process. With these enhancements, we expect to optimize all services' combined performance while avoiding bottleneck shifting during the tuning process.

IX. Conclusion

We presented an approach called SmartTuning to tune blackbox elastic cloud applications at runtime. We evaluated SmartTuning across 11 experimental conditions (3 industry applications and multiple workloads). We found that SmartTuning finds configurations that improve application efficiency by 58% on average, and decrease memory usage by up to 77% and the application cost by 27% on average. We also show that SmartTuning finds configurations in coordination with active auto-scaling while the application is in production and without inducing any downtime and minimal performance degradation.

X. Acknowledgements

This research was supported by grants from IBM CAS and Azure Education. We thank Huawei Canada for supporting our ongoing work on this topic. We also thank the anonymous reviewers for their valuable suggestions.

References

[1] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, 2013.

[2] L. Chen, "Microservices: architecting for continuous delivery and devops," in *International conference on software architecture (ICSA)*, 2018.

[3] "Kubernetes autoscaling proposal," accessed: 2023-01. [Online]. Available: https://github.com/kubernetes/design-proposals-archive/blob/main/autoscaling/vertical-pod-autoscaler.md#combining-vertical-and-horizontal-scaling

[4] "Open Liberty reference," accessed: 2023-01. [Online]. Available: https://openliberty.io/docs/21.0.0.6/reference/config/server-configuration-overview.html

[5] "OpenJ9," accessed: 2023-01. [Online]. Available: https://www.eclipse.org/openj9/

[6] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast bayesian optimization of machine learning hyperparameters on large datasets," in *Artificial Intelligence and Statistics*, 2017.

[7] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems (NeurIPS)*, 2012.

[8] M. Masdari and A. Khoshnevis, "A survey and classification of the workload forecasting methods in cloud computing," *Cluster Computing*, 2020.

[9] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using bad learners to find good configurations," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.

[10] S. Chaudhuri and V. Narasayya, "Self-tuning database systems: A decade of progress," in *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, 2007.

[11] D. Shasha, "Tuning databases for high performance," *ACM Computing Surveys (CSUR)*, 1996.

[12] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2017.

[13] C. Zheng, Z. Ding, and J. Hu, "Self-tuning performance of database systems with neural network," in *International Conference on Intelligent Computing (ICIC)*, 2014.

[14] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[15] A. Fekry, L. Carata, T. Pasquier, A. Rice, and A. Hopper, "To tune or not to tune? in search of optimal configurations for data analytics," in *Proceedings of the 26th ACM International Conference on Knowledge Discovery & Data Mining (SIGKDD)*, 2020.

[16] H. Herodotou, Y. Chen, and J. Lu, "A survey on automatic parameter tuning for big data processing systems," *ACM Computing Surveys (CSUR)*, 2020.

[17] L. Bao, X. Liu, Z. Xu, and B. Fang, "Autoconfig: Automatic configuration tuning for distributed message systems," in *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[18] M. Lešnik, B. Bošković, and J. Brest, "Performance tuning of java ee application servers with multi-objective differential evolution," in *IEEE Symposium on Differential Evolution (SDE)*, 2013.

[19] J. Lu, Y. Chen, H. Herodotou, and S. Babu, "Speedup your analytics: Automatic parameter tuning for databases and big data systems," *Proc. VLDB Endow.*, 2019.

[20] S. Rathnayake, L. Ramapantulu, and Y. M. Teo, "Characterizing the cost-accuracy performance of cloud applications," in *49th International Conference on Parallel Processing (ICPP)*, 2020.

[21] B. Raza, A. Sher, S. Afzal, A. K. Malik, A. Anjum, Y. J. Kumar, and M. Faheem, "Autonomic workload performance tuning in large-scale data repositories," *Knowledge and Information Systems*, 2019.

[22] Y. Ngoko and C. Cérin, "Reducing the number of comatose servers: automatic tuning as an opportunistic cloud-service," in *IEEE International Conference on Services Computing (SCC)*, 2017.

[23] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," *ACM SIGARCH computer architecture news*, 2011.

[24] M. Poess and R. O. Nambiar, "Tuning servers, storage and database for energy efficient data warehouses," in *IEEE 26th International Conference on Data Engineering (ICDE)*, 2010.

[25] S. Ma, F. Zhou, M. D. Bond, and Y. Wang, "Finding heterogeneous-unsafe configuration parameters in cloud systems," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021.

[26] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of Symposium on Cloud Computing (SoCC)*, 2017.

[27] Y. Zhu and J. Liu, "Classytune: A performance auto-tuner for systems in the cloud," *IEEE Transactions on Cloud Computing*, 2022.

[28] M. A. Chang, A. Panda, H. Wang, Y. Tsai, R. Balakrishnan, and S. Shenker, "Autotune: Improving end-to-end performance and resource efficiency for microservice applications," 2021. [Online]. Available: https://arxiv.org/abs/2106.10334

[29] R. Han, Z. Zong, L. Y. Chen, S. Wang, and J. Zhan, "Adaptiveconfig: Run-time configuration of cluster schedulers for cloud short-running jobs," in *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[30] R. Han, C. H. Liu, Z. Zong, L. Y. Chen, W. Liu, S. Wang, and J. Zhan, "Workload-adaptive configuration tuning for hierarchical cloud schedulers," *Transactions on Parallel and Distributed Systems*, 2019.

[31] M. C. Calzarossa, L. Massari, and D. Tessera, "Workload characterization: A survey revisited," *ACM Computing Surveys (CSUR)*, 2016.

[32] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for OLDI microservices," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[33] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "Softsku: Optimizing server architectures for microservice diversity @scale," in *ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.

[34] A. R. Sampaio, J. Rubin, I. Beschastnikh, and N. S. Rosa, "Improving microservice-based applications with runtime placement adaptation," *Journal of Internet Services and Applications*, 2019.

[35] C. Thalinger, "Performance tuning twitter services with graal and machine learning," 2019 (Accessed: 2023-01)", Devoxx Poland. [Online]. Available: http://cfp.2019.devoxx.pl/talk/HYA-1885/Performance_tuning_Twitter_services_with_Graal_and_Machine_Learning.html

[36] T. Chiba, R. Nakazawa, H. Horii, S. Suneja, and S. Seelam, "Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2019.

[37] Y. Zhang, H. He, O. Legunsen, S. Li, W. Dong, and T. Xu, "An evolutionary study of configuration design and implementation in cloud systems," in *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.

[38] V. M. Mostofi, D. Krishnamurthy, and M. Arlitt, "Fast and efficient performance tuning of microservices," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

[39] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: workload autoscaling at google," in *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020.

[40] "Daytrader," accessed: 2023-01. [Online]. Available: https://github.com/OpenLiberty/sample.daytrader8

[41] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *IEEE Computer Architecture Letters*, 2018.

[42] P. I. Frazier, "A tutorial on bayesian optimization," 2018. [Online]. Available: https://arxiv.org/abs/1807.02811

[43] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research*, 2012.

[44] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, 2015.

[45] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *25th annual conference on neural information processing systems (NeurIPS)*, 2011.

[46] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proceedings of the 30th International Conference on Machine Learning (PMLR)*, 2013.

[47] E. C. Garrido-Merchán and D. Hernández-Lobato, "Dealing with categorical and integer-valued variables in bayesian optimization with gaussian processes," *Neurocomputing*, 2020.

[48] S. Shishira, A. Kandasamy, and K. Chandrasekaran, "Workload characterization: Survey of current approaches and research challenges," in *Proceedings of the 7th International Conference on Computer and Communication Technology (ICCCT)*, 2017.

[49] M. C. Calzarossa, L. Massari, and D. Tessera, "Evaluation of cloud autoscaling strategies under different incoming workload patterns," *Concurrency and Computation: Practice and Experience*, 2020.

[50] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, 2003.

[51] Q. Chen, T. Wang, O. Legunsen, S. Li, and T. Xu, "Understanding and discovering software configuration dependencies in cloud and datacenter systems," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[52] S. Solorio-Fernández, J. A. Carrasco-Ochoa, and J. F. Martínez-Trinidad, "A review of unsupervised feature selection methods," *Artificial Intelligence Review*, 2020.

[53] K. Yu, X. Guo, L. Liu, J. Li, H. Wang, Z. Ling, and X. Wu, "Causality-based feature selection: Methods and evaluations," *ACM Computing Surveys*, 2020.

[54] "Kubernetes custom resources," accessed: 2023-01. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[55] "Prometheus," accessed: 2023-01. [Online]. Available: https://prometheus.io/

[56] "Kubernetes container probes," accessed: 2023-01. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes

[57] "Kubernetes rolling update deployment," accessed: 2023-01. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-update-deployment

[58] "kubernetes hpa."

[59] "Quarkus benchmark," accessed: 2023-01. [Online]. Available: https://quarkus.io/blog/runtime-performance/

[60] "AcmeAir," accessed: 2023-01. [Online]. Available: https://github.com/blueperf/acmeair-monolithic-java

[61] "Open Liberty framework," accessed: 2023-01. [Online]. Available: https://openliberty.io/

[62] "Quarkus framework," accessed: 2023-01. [Online]. Available: https://quarkus.io/

[63] "Mongo database," accessed: 2023-01. [Online]. Available: https://www.mongodb.com/

[64] "BD2," accessed: 2023-01. [Online]. Available: https://www.ibm.com/analytics/db2

[65] "PostgreSQL," accessed: 2023-01. [Online]. Available: https://www.postgresql.org/

[66] "Docker," accessed: 2023-01. [Online]. Available: https://www.docker.com/

[67] "Jmeter," accessed: 2023-01. [Online]. Available: https://jmeter.apache.org/

[68] "Jakarta Server Pages," accessed: 2023-01. [Online]. Available: https://jakarta.ee/specifications/pages/

[69] "Jakarta Server Faces," accessed: 2023-01. [Online]. Available: https://jakarta.ee/specifications/faces/

[70] Y. Dodge, *The concise encyclopedia of statistics*. Springer Science & Business Media, 2008.

[71] "Azure VMs," accessed: 2023-01. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/

[72] "Kubernetes autoscaling proposal," accessed: 2023-01. [Online]. Available: https://xgboost.readthedocs.io/en/stable/python/python_api.html#xgboost.XGBRegressor