Proposal: Performance Management of Large-Scale Microservices Applications

Thesis proposal

Gagan Somashekar

Department of Computer Science
Stony Brook University

This report is submitted for the thesis proposal requirement of
Doctor of Philosophy

June 2023

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this report are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This report is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text.

Gagan Somashekar
June 2023

Abstract

Microservices architecture has rapidly gained popularity for building large-scale latency-sensitive online applications. The architecture supports decomposing an application into a collection of fine-grained and loosely-coupled services called microservices. Each microservice typically implements a single business capability with inter-microservice communication enabled via Application Programming Interfaces (APIs). This modular architecture enables independent management of microservices for agility, scalability, and fault isolation. Given the benefits of this modular architecture, microservices architecture is widely replacing existing deployments implemented using monolithic or multi-tier architectures at Microsoft, Amazon, Netflix, Twitter, etc. However, the modular design of microservices architecture leads to a large graph of interacting microservices whose influence on each other is non-trivial. As a result, performance management and debugging of microservices architecture is a challenging problem. This thesis develops techniques built on optimization theory and machine learning to address performance management problems in microservices architecture. Specifically, this thesis focuses on solving two critical issues faced in microservices architecture: configuration tuning and bottleneck detection.

Application configuration tuning is essential to improve performance and utilization, but the microservices architecture leads to a very large configuration search space with interdependent parameters. We jointly optimize the parameters to deal with interdependence and develop practical dimensionality reduction strategies based on available system characteristics to reduce the size of the search space. Our pre-deployment (offline) evaluation of different optimization algorithms and dimensionality reduction techniques across three popular benchmark applications highlights the importance of configuration tuning in reducing tail latency (by as much as 46%). The right combination of optimization algorithms and dimensionality reduction techniques can provide substantial latency improvements by identifying the right subset of parameters to tune, reducing the search space by as much as 83%.

Post-deployment tuning of real-world applications requires dynamic reconfiguration as the workloads are complex and time-varying. Moreover, the tuning process must reduce application interruptions to maintain the quality of service and application uptime. We design

OPPerTune, a framework that uses various machine learning algorithms to handle the challenges involved with post-deployment tuning. We evaluate OPPerTune on a benchmarking application deployed on an enterprise cluster with synthetic and production traces to analyze its effectiveness in (a) determining which configurations to tune, and (b) automatically managing the scope at which to tune the configurations. Our experimental results show that OPPerTune reduces the end-to-end P95 latency of microservices applications by more than 50% over expert configuration choices made ahead of deployment.

Beyond configuration tuning, it is critical to detect and mitigate sources of performance degradation (bottlenecks) to avoid revenue loss. As part of the proposed work, we will investigate techniques to detect and mitigate bottlenecks in microservices applications. Specifically, we plan to use graph learning algorithms to exploit the inherent graph data in microservices application deployments to detect bottlenecks. We plan on exploring different mitigation strategies, including configuration tuning (e.g., autoscaling, application configuration tuning, etc.). Our preliminary results using graph neural networks show that we can improve bottleneck detection accuracy and precision by up to 15% and 14%, compared to the techniques used in existing work.

It is our thesis that optimization and machine learning algorithms coupled with system characteristics can effectively address the complexities of configuration tuning and bottleneck detection and mitigation in large-scale microservices applications.

Table of contents

Chapter 1

Introduction

Software architecture is the abstract structure of a software system that represents its design decisions and behaviors [75]. Among the various architectural styles, microservices architecture, an adaptation of service-oriented architecture [43], is gaining rapid industry acceptance in building large-scale latency-sensitive distributed applications. Performance and availability are paramount in such applications as they usually provide customer-facing services like video streaming, ride-hailing, and social media [61, 62, 73, 179, 112].

Microservices architecture arranges an application as a suite of loosely coupled, fine-grained services called *microservices* that communicate via well-defined APIs agnostic to the implementation. This design enables teams to manage and implement each microservice independently of others using the best software stack for the specific business functionality. Breaking down an application into small independent services results in many advantages, including scalability, fault isolation, data isolation, and ease of deployment and maintenance. Accordingly, distributed applications implemented using this architectural style are widely replacing those implemented using monolithic or multi-tier architecture at Amazon, Netflix, Uber, Twitter, etc. [73, 112]. Table 1.1 shows the difference between monolithic and microservices architecture.

With the microservices architecture, an online application can be implemented as a frontend (e.g., Nginx), database (e.g., MongoDB) and caching microservices (e.g., Redis) along with services that implement the business logic. The business logic is usually implemented using frameworks like Apache Thrift [1] and gRPC [8] that provide a number of developer-friendly features.

The modular architecture of microservices architecture offers numerous advantages. A few of the important advantages are:

- **Independent deployment:** Each microservice can be independently deployed.

Monolithic architecture	Microservices architecture
All the functionalities exist in a single code base	Each business functionality is implemented as a separate service
The whole application is redeployed for smaller changes	Modular architecture enables independent deployment
Scaling is expensive as the whole application has to be scaled	Only the microservices that are handling the large load can be scaled
Adapting to new technology is time, and cost-inefficient due to tight coupling between functionalities	Each microservice can be implemented using the technology suitable for the business logic it implements
Database is shared	Each microservice has its own database

Table 1.1 Differences between monolithic and microservices architecture.

- **Flexible scaling:** Each microservice can be independently scaled based on the load it handles.
- **Continuous development:** Updates are iterative through the redeployment of a small subset of microservices leading to constant improvements to the product.
- **Loosely coupled:** Each microservice can use technology suitable for the business logic it implements.
- **Improved fault tolerance:** Failure of a single microservice does not bring down the whole application.

Despite its many advantages, the microservices architecture has a few disadvantages too. Mainly, the highly complex and large graph of interacting microservices exacerbates some of the traditional software engineering problems:

- **Configuration tuning:** The large configuration state space and the interdependent parameters complicate configuration tuning [146, 147, 149, 97].
- **Operational complexity:** The distributed and independent design requires specialized teams (e.g., DevOps) and tooling to manage the application's life cycle [162, 84].
- **Resource management:** Fine-grained resource management approaches that are application-aware and latency-aware are essential to maintain the quality of service [73, 129, 74]
- **Bottleneck detection and mitigation:** The cascading performance degradation common to microservices aggravates bottleneck detection and mitigation [74, 129, 145].

- **Testing and monitoring:** As microservices are distributed, testing and monitoring them is a challenging task [167, 84].

1.1 Challenges

We will discuss the challenges of configuration tuning (both pre-deployment and post-deployment) and bottleneck detection and mitigation from the list of challenges presented in the previous section.

- **Pre-deployment configuration tuning**: The microservice architecture has numerous advantages, but one major challenge is adjusting the configuration settings of individual microservices before deployment. With microservices applications, configuration tuning is especially complicated owing to the large configuration search space, interdependent parameters, dependency between parameters of different microservices, interference among colocated microservices and non-linear relationship between microservices parameters and performance.

- **Post-deployment configuration tuning**: The configuration obtained via pre-deployment tuning may not work well in the long term. The parameters should be tuned continuously to optimize the performance and efficiency as software upgrades, dynamic workloads, and changes in the underlying hardware affect the application's behavior continuously. Recent efforts have proposed using machine-learning-based techniques to automate the process of configuration tuning, using online learning or reinforcement learning to set configurations, observe application state, and iteratively refine the configuration. However, this approach is not a complete solution as it only addresses one aspect of the end-to-end process of configuration tuning.

  The automated approach for configuration tuning should take into consideration several factors. First, it should determine which components or layers of the service to tune and which parameters to focus on. Second, it should consider the varying difficulty of tuning different types of parameters and use a small number of iterations to avoid disruptions. Third, it should determine the right context or scope for each tuning instance. Finally, the approach should work with numerical and categorical parameters and be efficient and quick to converge.

- **Bottleneck detection and mitigation**: Detecting and mitigating performance bottlenecks in online applications is crucial to provide a good customer experience [46, 62]. For example, experiments at Amazon showed that an additional 100ms of latency could reduce sales by 1%; similarly, experiments at Google showed that increasing the

time to display search results by 500ms can reduce the revenue by 20% [99]. Long tail latencies that significantly affect the revenues of online applications are often a result of *performance bottlenecks* that do not necessarily lead to errors or faults and instead arise due to resource saturation, resource contention, or microservices application misconfiguration [73, 74, 129, 146]. Regardless of the underlying cause of performance bottlenecks, it is essential to have a technique that quickly adapts to dynamic online workloads and accurately detects bottlenecks with high recall and precision. A low recall is especially problematic as it implies that performance issues go unaddressed.

Microservices architecture has unique characteristics compared to other architectural styles that complicate bottleneck detection. Mainly the back-pressure effects and cascading performance degradation due to the complex interaction between microservices, the scarcity of labeled production data for bottlenecked classes, time-varying interactions due to software updates, and components like caches, message queues, etc., exacerbate the problem. These complexities also necessitate mitigation approaches tailored to microservices [74].

1.2 Our Contributions

This section overviews our contributions in addressing the challenges elaborated in Section 1.1.

- **Pre-deployment configuration tuning**: We consider the challenges of pre-deployment configuration tuning and employ different techniques to tackle them. To address the problem of inter-dependent parameters, we consider *joint optimization* of the parameter space. We conduct an extensive experimental investigation of six black-box optimization algorithms with the goal of minimizing the tail latency (up to 46%) of benchmarking applications. To address the key challenge of a large configuration space when jointly tuning microservices applications, we investigate various *dimensionality reduction* approaches to identify a subset of microservices that are most likely to impact end-to-end application latency. We can significantly improve tail latency by employing dimensionality reduction while only having to tune about 18% of all microservices. In fact, within a given budget on the number of iterations of the optimization algorithm, optimizing with dimensionality reduction can further improve tail latency (by about 6.5%) compared to optimizing without any dimensionality reduction since the search space is reduced, thereby aiding the optimization.

- **Post-deployment configuration tuning**:

We use OPPerTune (**O**nline **P**ost-deployment **Per**formance **Tune**r), a configuration tuning service designed, developed and deployed with our collaborators to address the challenges of post-deployment configuration tuning. Application operators can use OPPerTune to create automatic tuning instances, specifying the configuration parameters they wish to tune. The OPPerTune service supports (a) various algorithms in its back-end that the tuning instances could use and (b) ways to automatically create, manage, and scope the instances needed to tune the performance of large-scale complex services. We evaluate the framework on a benchmarking application using synthetic and production traces showing tail latency improvements of up to 50%.

- **Bottleck detection and mitigation**: In the proposed work, we explore the use of *Graph Neural Networks* (GNNs) [59, 180] to detect bottlenecks in online microservices applications. GNNs are ideally suited for analyzing microservices applications, as they capture the back-pressure and cascading performance degradation along the call graphs [73, 126], learn the dependence of graphs via message passing between the nodes of graphs [180], generalize for the dynamic call graphs that are common in microservices [86, 83, 112, 129], and be regularized to ensure representation learning equilibrium across multiple classes thereby avoiding the multi-class imbalance problem seen in traditional ML algorithms [142]. In this preliminary work, we use open-source traces with single bottlenecks to evaluate the model on a simple set of traces. We propose to create a dataset of traces consisting of single and more realistic multiple bottlenecks from different sources (interference, misconfiguration, etc.). We will explore different detection and mitigation strategies for such multiple bottlenecks as part of the proposed work.

1.3 Chapter organization

The rest of this report is organized as follows. In Chapter 2 we provide background on microservices and important prior works that tackle the challenges of microservices architecture. Chapter 3 presents our approach to pre-deployment configuration tuning. Chapter 4 presents our work on using configuration tuning frameworks to tackle the challenges of post-deployment configuration tuning. In Chapter 5 we discuss the proposed work on bottleneck detection and mitigation. Finally, in Chapter 6, we discuss future work and conclude this report.

Chapter 2

Background

The adoption of microservices architecture in the industry is growing rapidly due to its numerous advantages. However, it also comes with certain disadvantages that can significantly impact performance management as the scale increases. To gain a deeper understanding of these implications and the current state of research, this section provides a brief overview of the history of microservices, its key features, benchmarking applications, and large-scale production applications, followed by a discussion of some of the related works.

2.1 Microservices Architecture

Software architecture presents the design and behavior of the system or the application. It also largely dictates how different aspects of the application's life cycle are managed. Hence, it is important to find the right architectural style keeping in mind the evolution of the application. Among the various architectural styles, Service Oriented Architecture (SOA), the precursor to microservices architecture, shares some characteristics (e.g., modularity) of microservices architecture. The main difference lies in the scope of the individual services [65, 23]. SOAs have an enterprise scope with a main focus on the reusability of the services, and microservices have an application scope focusing on decoupling the services.

The term *microservices* was first introduced in 2011 at a workshop on software architecture patterns [65]. However, similar architectural styles under different names (e.g., Fine-grained SOA [65]) existed before this. While there does not exist a strict definition of microservices architecture, certain characteristics wholly or partially make up microservices architecture. These characteristics include a) modular services, b) organized around business capabilities, c) decentralized governance, d) decentralized data management, e) infrastructure automation, f) fault tolerance, and g) evolutionary design [23, 65, 15].

2.1.1 Characteristics

In this section, we discuss a few important characteristics of microservices architecture in detail:

- **Modular and independent services:** With the microservices architecture, the goal is to design services (microservices) at the application level that can be independently deployed and upgraded. These microservices differ from other modular components, like libraries, because they are a single or group of processes communicating over a web service request or a Remote Procedure Call (RPC) with other microservices. Moreover, when modified, components like libraries would require redeploying the whole application, unlike microservices, where only the modified service is redeployed. The downside of this design is that remote calls are more expensive than in-process calls.

- **Organized around business capabilities:** Traditionally, the components of an application are decided based on the technology layer. For example, separate large teams would work on user interfaces, server-side logic, and databases across products in an enterprise. Due to tight coupling between such components, cross-team collaboration is necessary even for small changes. In the microservices architecture, a large and cross-functional team handles one particular service or application, i.e., based on business capability, with smaller groups owning individual services based on business functionality (microservices). This allows smaller teams to work together on an application, easing communication and collaboration efforts.

- **Decentralized data management:** In the microservices architecture, each microservice manages its own database. The database technology or the system used could be different across the microservices of the same application. This contrasts with the monolithic design, where a single logical database is shared across the application, sometimes across an enterprise's applications. The decentralized data management in the microservices architecture decouples the microservices, which is one of its main objectives. However, data might be duplicated across microservices, leading to data integrity and consistency problems [100].

- **Improved fault tolerance:** The architecture requires that the application be robust to individual microservice failures. Microservices are expected to gracefully handle the failure of a downstream microservice in their call path. This does add additional code complexity but makes the application resilient and robust. It is a common practice to test the application's ability to handle such failures through automated testing. For

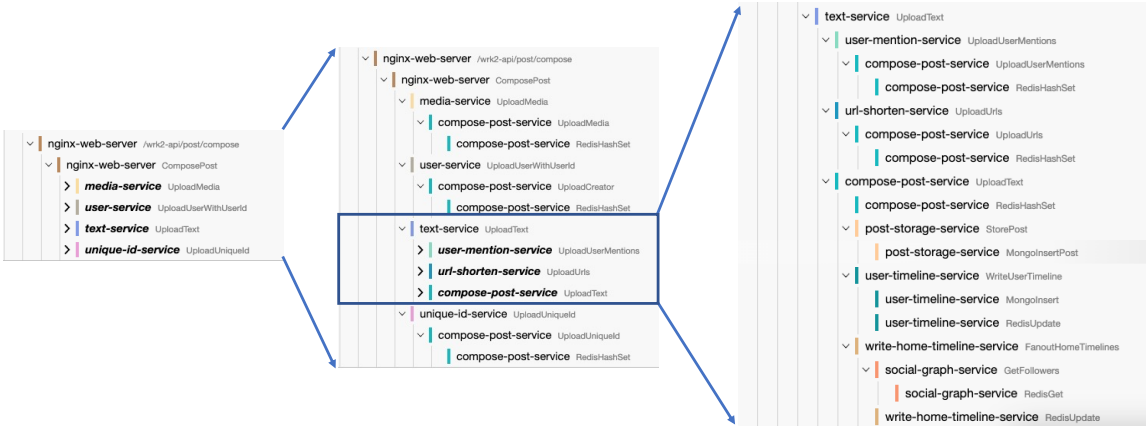


Fig. 2.1 The Jaeger trace of *compose-post* request of the social networking application. The leftmost rectangular block shows that the client's request arrives at the front end (*nginx-web-server*), which calls 4 other microservices (*media-service*, *user-service*, *text-service* and *unique-id-service*) in parallel. The middle and the right rectangular blocks show downstream calls at deeper levels in the call graph.

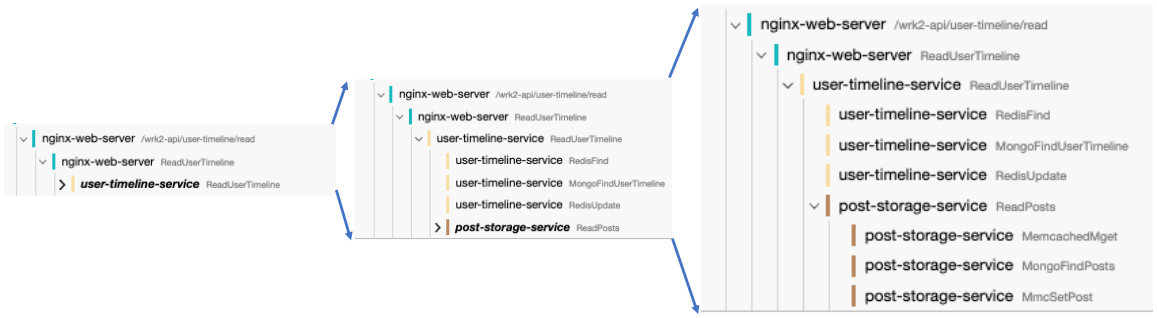


Fig. 2.2 The Jaeger trace of *read-usertimeline* request of the social networking application.

example, Netflix's chaos monkey [3] randomly terminates microservices in production to ensure that each microservice is implemented to handle such failures. For efficient debugging, distributed monitoring and logging are usually required and are part of most microservices applications [113].

2.1.2 Benchmarking Applications

Throughout our thesis, we use various open-sourced benchmarking applications to evaluate our hypothesis [73, 181]. We briefly discuss the architecture and functionalities of these applications here.

- **Social Network:** The social network application has 28 microservices that implement an end-to-end broadcast-style social networking application. The application consists of Nginx as a frontend, Memcached and Redis for caching, MongoDB for persistence storage, and Thrift-based microservices that implement the application's logic. Currently, it supports three request types: a) *compose-post*, where a user creates

Parameter	Microservice (type)	Description
worker_processes	frontend-nginx (Nginx)	The number of Nginx worker processes
maxmemory-policy	social-graph-redis (Redis)	The policy Redis uses to remove elements when memory usage reaches the limit.
io_threadpool_size	social-graph-service (Thrift)	The size of the IO thread pool in a Thrift server.
memory_limit	post-storage-memc (Memcached)	Memory limit for object storage.
serviceExecutor	url-shorten-mongodb (MongoDB)	Determines the threading and execution model.

Table 2.1 A subset of parameters of the social networking application.

and uploads a post embedded with text, media, links, and tags to other users b) *read-hometimeline*, where a user reads their own timeline, and c) *read-usertimeline*, where a user can read another user's timeline.

The call graphs of the *compose-post* and the *read-usertimeline* requests in the form of Jaeger traces are as given in Figure 2.1 and Figure 2.2, respectively. While we do not go into the details of the functionality of each microservice, the call graphs show the complex interaction between services, even for a benchmarking application.

Table 2.1 shows a subset of social networking application's parameters along with the specific microservice they belong to, the type of the microservices and the description of the parameter.

- **Media Microservices** The media microservices application consists of 31 microservices and implements a movie review system. The constituent microservices are similar to the ones in the social networking application. The application supports different request types, including a) *compose-review*, where a user reviews a particular movie, b) *read-plot*, where user reads the plot of the movie, c) *read-cast-info*, where the user reads the cast of the movie, and d) *read-review*, for reading an existing review.
- **Hotel Reservation** The hotel reservation application consists of 18 microservices that together implement an end-to-end hotel reservation system. The front end and the logic microservices are implemented in Go. It uses Memcached for caching and MongoDB for persistent storage. The application supports different request types a) *search-hotel*, to search a hotel in the user-specified date range and location, b) *recommend*, to recommend hotels based on distance, rate, or price and c) *reserve*, to reserve a hotel for the specified date range.

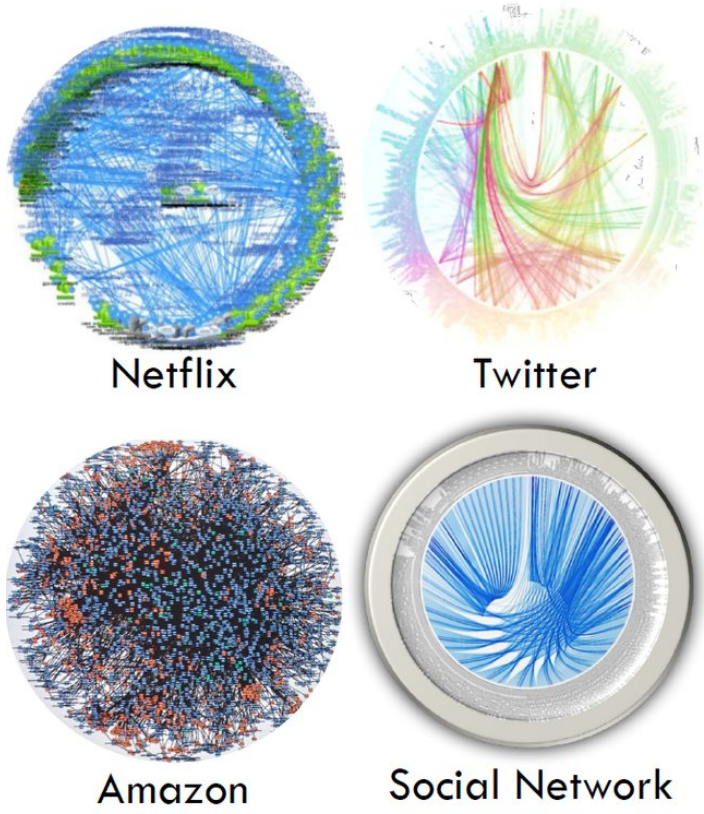


Fig. 2.3 The microservices graphs in three large cloud providers and the social networking application [73]. The points on the circumference correspond to microservices, and the lines between them represent interactions.

- **Train Ticket** The train ticket application is a train ticket booking system consisting of 41 microservices. In addition to the microservices that are part of the social networking application, this application includes a MySQL microservice. The application supports different request types but the ones we use in our experiments are a) *search*, to search all the trains between user-specified source and destination and b) *book*, to book a particular train taking into account user's preferences like food type, seat type, etc.

2.1.3 Large-Scale Applications

Although representative of real-world applications, the benchmarking applications discussed in the previous section fall short in scale. The real-world applications implemented using microservices architecture consist of 100s to 1000s of microservices [112, 73, 179] as shown in Figure 2.3. Uber's backend, for example, has around 4000 microservices interacting with

each other via RPCs [179]. With the increase in scale, the challenges of the microservices architecture are harder to solve manually or using traditional techniques, necessitating learning-based techniques [112]. Moreover, a large amount of data is collected through distributed tracing and logging, aiding data-driven approaches to solve the challenges of microservices architecture. However, the techniques used to solve the challenges must not affect the performance if they are on the critical path. Considering all these aspects, we explore various techniques to deal with the challenges of configuration tuning and bottleneck detection and mitigation of large-scale applications.

2.2 Related Work

In this section, we discuss some of the recent works related to microservices architecture and their challenges. First, we discuss works on benchmarking and real-world microservices applications, followed by configuration tuning, ending with bottleneck detection and mitigation. In each separate chapter, we delve into more specific related works.

2.2.1 Benchmarking and Real-World Applications

In this section, we discuss DeathStarBench [73], a benchmarking suite and Luo et al. [112], a work that characterizes microservices applications in Alibaba clusters.

Gan et al. [73] introduce the DeathStarBench benchmarking suite consisting of applications implemented using the microservices architecture. The applications are representative of large-scale end-to-end real-world applications and are built using popular open-sourced technologies. The authors use the suite to explore the implications of microservices applications across the system stack. Specifically, the authors study microservices' architectural characteristics, their implications on operating systems and networking, and their unique challenges in cluster management. In addition, they study the effect of scale by deploying one of the benchmarking applications in the real world and scaling it to hundreds of users. The study reveals various characteristics of microservices applications including a) the latency requirement for each microservice is stricter to provide QoS requiring predictable performance, b) the network cost is high due to RPCs, c) network acceleration improves latency drastically, d) back-pressure effects and cascading QoS violations make the performance unpredictable and e) performance management is harder at scale as minor mismanagement can degrade end-to-end latency significantly.

Luo et al. [112] characterize Alibaba's production services implemented using the microservices architecture. The traces used for analysis consists of data for around 20000 microservices over a 7-day period. The authors specifically characterize the microservice dependency and runtime performance. The critical observations from the authors are a)

microservice call graphs are significantly different from the traditional DAGs, b) there is a strong dependency between microservices which can be exploited to optimize the design, and c) stochastic models can simulate dynamic microservice call graphs. Other than these critical observations, some of the other important results of the analysis are a) the size of microservices call graphs are heavy-tailed, b) the number of microservices in a call graph follows a Burr distribution, c) for larger call graphs, about 50% of the microservices is Memcached, d) the largest call graph can consist of even hundreds to thousands of microservices, e) about 5% of the microservices are shared by 90% of the online services, f) microservices can form highly dynamic call dependencies in runtime, and g) the run time performance highly correlates with CPU utilization but not memory.

2.2.2 Configuration Tuning

In this section, we discuss μTune [149], KEA [182], and Twine [156], which are frameworks for tuning applications.

Sriraman et al. [149] introduce μTune, which consists of a novel framework to abstract the threading model from the users and a load adaptation system that tunes the threading models depending on the load to reduce latency. The microservice applications considered are slightly less representative than the real-world applications as they only consist of a front end, a mid-tier, and a number of leaf nodes (workers). The authors provide a taxonomy for threading models on the dimensions of communication (synchronous vs. asynchronous), RPC processing (in-line vs. dispatch), and RPC reception (block vs. poll), leading to 8 different threading models. During the training phase, tail latency for different threading models and thread pool sizes are collected at various loads. Using this data, a piecewise linear model is constructed, which is used during runtime to switch the threading model to maximize performance.

KEA [182] is both a methodology and a data-driven system to tune cluster-wide configurations of a big-data infrastructure at Microsoft to reduce operational costs. The methodology involves three phases. The first phase involves a collaboration between the data scientists and the system specialists to define the problem, scale, objectives and constraints. The second phase involves developing machine learning models to capture the relationship between performance metrics and configurations and find the optimal configurations. The third phase involves testing the optimal configurations on a subset of the cluster before deploying it across the cluster. Additionally, the framework provides three different tuning approaches - a) Observational tuning, b) Hypothetical tuning and c) Experimental tuning, together which can cover a variety of scenarios. Observational tuning corresponds to configuration scenarios where the data collected from past cluster operations is sufficient to predict the performance

of new configurations. Hypothetical tuning also uses data collected from past cluster operations to predict configurations for future machines helping in the planning. Experimental tuning is used when additional experiments are needed to find the optimal configurations making it expensive compared to the other two. In summary, KEA uses domain knowledge and data science to continuously tune Microsoft's cluster configurations, saving tens of millions of dollars per year.

Twine [156] is Facebook's cluster management system that manages all the shared infrastructure at Facebook. Among various other features, Twine provides workload-specific customizations like tuning hardware and OS settings to improve performance and custom event-based handling of container life cycle management. The applications are mapped to hosts with specific settings that optimize their performance through an abstraction called *host profile* (17 in total). A host profile corresponds to specific values for kernel versions, sysctls (e.g., hugepages and kernel scheduler settings), cgroupv2 (e.g., CPU controller), storage (e.g., XFS or brtfs), NIC settings, CPU Turbo Boost, and hardware prefetch. The machines allocated to an application are dynamically reconfigured in accordance with the host profile before the workload is scheduled thus optimizing the performance.

2.2.3 Bottleneck Detection and Mitigation

FIRM [129] presents a fine-grained resource management framework for microservices applications. FIRM provides the following insights which shape the solutions a) the critical paths in the microservices applications are dynamic b) a microservice with high service time need not be the source of performance degradation and c) mitigation policies vary with user load and the resource in contention. Using these insights, FIRM first detects the bottlenecks to get a set of candidates for autoscaling. It performs critical path analysis to reduce the number of microservices considered candidates for bottlenecks, thereby reducing the number of parameters tuned. Following this, an SVM model classifies the microservices as potential bottlenecks and non-bottlenecks. For the potential bottlenecks, FIRM uses reinforcement learning to tune resource limits for CPU, memory, LLC, I/O and network (starting state corresponds to overprovisioned state). FIRM performs better than Kubernetes's autoscaling model and AIMD (rule-based auto-scaling technique). Transfer learning is shown to be effective. Thus, training time, which is a cost over black-box optimization techniques, is avoided.

Chapter 3

Pre-deployment Configuration Tuning

Microservice architecture is an architectural style for designing applications that supports a collection of fine-grained and loosely-coupled services, called microservices, enabling independent development and deployment. An undesirable complexity that results from this style is the large state space of possibly inter-dependent configuration parameters (of the constituent microservices) which have to be tuned to improve application performance.

This chapter investigates optimization algorithms to address the problem of configuration tuning of microservices applications. To address the critical issue of large state space, practical dimensionality reduction strategies are developed based on available system characteristics.

3.1 Introduction

The emerging microservice architecture allows applications to be decomposed into different, interacting modules, each of which can then be independently managed for agility, scalability, and fault isolation [73, 149]. Each module or microservice typically implements a single business capability with inter-microservice communication enabled via Application Programming Interfaces (APIs). Applications deployed using the microservice architecture thus enable flexible software development.

The microservice architecture is especially well suited for designing online, customer-facing applications where performance and availability are paramount [61, 62]. For example, an online application can be deployed as front-end microservices (e.g., Nginx), a set of microservices that implement the logic of the application each of which can have their own database (e.g., MongoDB) and caching (e.g., Memcached) microservices. Consequently, an application can have *numerous* microservices. Given the benefits of the modular architecture, microservices architecture is widely replacing existing deployments implemented using monolithic or multi-tier architectures at Amazon, Netflix, and Twitter [73].

Despite the benefits of the microservice architecture, a specific challenge that this distributed deployment poses is that of ***tuning the configuration parameters of the constituent microservices***. A change in configuration parameters can substantially impact application performance, motivating our investigation of configuration tuning. For example, sweeping over the valid range of values for the *worker_process* parameter of the nginx [2] microservice in the social networking application [73] (while keeping the rest of the parameters at default) can provide up to 13% improvement in latency over the default configuration. However, *joint optimization* of all the application parameters can provide 46% latency improvement over the default configuration (see Section 5.1.4). On the other hand, setting a sub-optimal (but still valid) value for the *worker_process* parameter of the nginx while setting the rest of the parameters to the optimal values can *deteriorate* the performance by up to 100× compared to the default configuration. Tuning the parameters of monolithic or N-tier applications for maximizing performance is already a difficult task [183, 166, 165, 159, 114, 149] (see Section 3.2). With microservice applications, configuration tuning is especially complicated owing to the following challenges:

- ***Very large configuration space.*** Microservices applications have hundreds to thousands of interacting microservices that each have several parameters that can be configured [112]. Frameworks that aid microservices development, such as Apache Thrift [33] and gRPC [90], introduce additional parameters that impact application performance. These parameters can take values that are discrete, continuous, or categorical, complicating attempts to optimize their values.

- ***Inter-dependent parameters.*** The parameter setting of a microservice can influence the optimal value of a different parameter of the *same* microservice. As a result, the numerous parameters of a given microservice cannot be independently optimized (see Section 5.1.4). For example, for MongoDB, a low value of the cache size parameter can amplify the number of concurrent read transactions, making it difficult to independently tune the latter parameter [17].

- ***Dependency between parameters of different microservices.*** The dependency between parameter values extends beyond a single microservice; parameters of upstream services are often dependent on the parameter settings of downstream services [165]. For example, the thread pool size of a microservice may dictate how many concurrent requests are sent to the downstream microservice.

- ***Interference among colocated microservices.*** Microservices, typically deployed as containers, can be colocated on the same physical host. Due to potential resource contention,

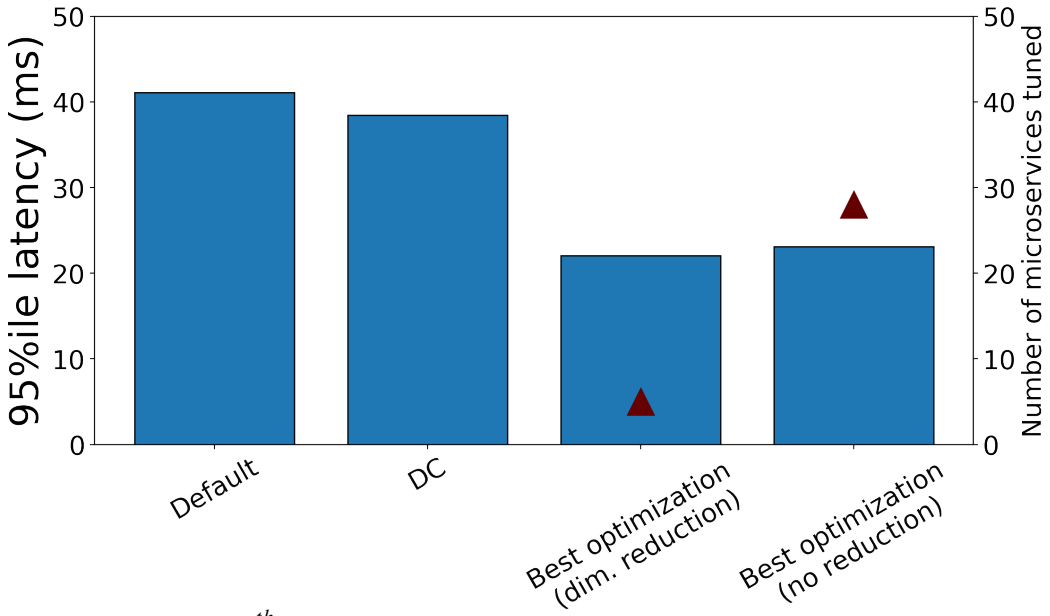


Fig. 3.1 Comparison of 95^{th} percentile of latency for the social networking application [73] under (i) default configuration values (Default), (ii) the configuration used by DeathStarBench benchmark developers [5] (DC), and the configuration found by the best optimization technique among those we explored (iii) with dimensionality reduction (considering only a subset of microservices for tuning) and (iv) without any reduction (tuning all microservices). The right y-axis shows the number of microservices tuned for (iii) and (iv).

the resource configuration of a microservice can impact the performance of other colocated microservices. For example, the cache size of two colocated caching microservices should not be set independently as they share the host's memory resources.

- ***Non-linear relationship between microservices parameters and performance.*** Application performance need not be monotonically or linearly dependent on parameter values, making it difficult to determine optimal configuration parameter settings. The thread pool size parameter is a classic example whereby a low value results in under-utilization of the CPU and a very high value results in contention for network sockets or CPU resources [149].

There is little prior work on the specific problem of configuration tuning of microservices, and that work relies on empirically exploring the configuration setting of only specific parameters of just stateless microservices [149]. There are, however, prior works that focus on optimizing the configuration of individual services [56, 159], but as explained above, the dependencies between the parameters of microservices makes it infeasible to optimize them in isolation.

This chapter explores the problem of configuration tuning of microservices applications. To address the problem of inter-dependent parameters, we consider *joint optimization* of the parameter space. We conduct an *extensive experimental investigation* of six black-box

optimization algorithms with the goal of minimizing the tail latency of a given microservice application deployment. As shown in Figure 3.1, the best optimization algorithm can significantly improve application tail latency (95^{th} percentile), by as much 46% and 43%, compared to the default configuration setting and the suggested configuration in prior work [73], respectively. We also find that combining different algorithms can result in efficient solutions that quickly (with few exploration points) explore the state space and provide significant latency improvements.

To address the key challenge of a large configuration space when jointly tuning microservices applications, we investigate various *dimensionality reduction* approaches to identify a subset of microservices that are most likely to impact end-to-end application latency. As illustrated by the two rightmost bars in Figure 3.1, by employing dimensionality reduction, we can achieve significant improvement in tail latency while only having to tune about 18% of all microservices. In fact, within a given budget on the number of iterations of the optimization algorithm, optimizing with dimensionality reduction can further improve tail latency (by about 6.5%) compared to when optimizing without any dimensionality reduction since the search space is reduced, thereby aiding the optimization.

Our investigation of different algorithms reveals that the optimal choice is application-dependent. While the hybrid algorithm we devise performs best for the social networking and the train ticket applications, Bayesian optimization performs best for media microservices application, in terms of tail latency reduction. In terms of the time taken to run the algorithm, dynamically dimensioned search (DDS) performs the best.

This work makes the following contributions:

- We formulate configuration tuning of microservices application as a *joint optimization problem*, making it amenable to optimization algorithms. Contrary to serial tuning, this provides an opportunity for the optimization algorithms to learn the dependencies among parameters of the same microservices and across microservices.

- We implement a *framework* [7] to experimentally explore and evaluate the configuration space of parameters for microservices. The framework is fully automated and can be integrated with any optimization technique.

- We implement six different representative optimization algorithms using open-sourced libraries and compare their efficacy in choosing the best configuration with respect to minimizing the application tail latency. To assess the optimization algorithms' applicability in practice, we also analyze their convergence and overhead.

- Based on our analysis of different algorithms, we design and evaluate an efficient hybrid algorithm that combines the strengths of different algorithms. In particular, the algorithm quickly explores the state space using a heuristic-based search and then uses the results of this search to initialize a model-based search algorithm.

- For scalability, we investigate different approaches, including *critical path* and *variability tracking*, to reduce the overhead of optimization by limiting the set of microservices whose parameters will be configured. We analyze the ability of these different techniques to capture the most important parameters that impact application tail latency.

- We use functional analysis of variance (fANOVA) [88] to find the most important parameters and analyze the values assigned to them by different optimization algorithms. We also examine the change in service time of individual microservices to assess the impact of optimization on different request types in the workload.

3.2 Background and Prior Work

Microservice architecture is a style of architecture where the application is implemented as a set of loosely coupled services, called microservices. This shift in the design of distributed applications requires revisiting some of the problems that have been addressed for monolithic and N-tier architectures. Resource management [129] and bottleneck mitigation [74] for microservices applications are some of the problems that have garnered significant attention from the research and development community. Prior work on performance improvement of microservices has primarily focused on resource allocation [74, 169, 129, 77]; We take a different approach to improving the performance of distributed applications implemented using a microservices architecture. In particular, we tackle the problem of tuning the parameters of microservices to improve the performance metric of interest (e.g., tail latency or throughput).

The general problem of tuning parameters of computer systems has gained significant attention [56, 159, 114]. However, these works do not focus on the specific problem of microservices configuration where several, inter-dependent parameter configurations have to be tuned. The one extensive prior work on configuration tuning of microservices that we are aware of is by Sriraman et al. [149]. In this work, the authors explore the tuning of a small subset of microservices parameters, limited to thread pool size and threading model. However, the state space of configuration parameters for microservices is very large, as discussed in Section 5.1.1, and hence a more comprehensive investigation of parameters is required for the performance optimization of microservices applications.

A naive approach to address the large state space of configurations for microservices applications is to tune one microservice at a time. While this approach significantly reduces the state space dimensionality for configuration tuning, it does proportionally increase the tuning effort. Further, this serial tuning approach cannot capture the complex relationship between different parameters and the cascading effects between different microservices [74, 50].

Based on the above discussion, we argue that there is a need to investigate *joint optimization* of the microservices' parameters. The joint optimization is needed in order to capture the impact of *multiple* parameters of one microservice on its performance as well as the impact of a microservice's parameters on the performance of other microservices. Further, mechanisms are needed to reduce the configuration state space, given the numerous parameters and microservices employed by modern applications.

We now briefly discuss prior works related to the general problem of configuration tuning in systems before we formalize our specific problem in Section 3.3.1.

Application configuration tuning. There has been considerable research in parameter tuning for individual applications, such as Apache web server [165], Memcached [161], database [159] and storage systems [56, 182], etc. OtterTune [159] is a tool that tunes parameters of the database management system (DBMS) using ML. OtterTune uses the learning experience of tuning other DBMS deployments to tune new ones, thereby reducing the time and resources required to tune the DBMS configuration of a new application. Cao et al. [56] use different black-box optimization algorithms to tune the parameters of storage systems and compare the performance of these algorithms. Wang et al. [165] study the performance implications of tuning thread pool and data connection pool size for an N-tier application and employ a queueing model to assign near-optimal values for the software configuration. The authors emphasize the dependency between parameters of different tiers, which is observed in microservices applications as well. While the above works can be used to tune individual microservices in isolation, the dependencies between microservices necessitates global optimization across microservices.

SmartConf [166] is a control-theoretic framework that automatically sets and dynamically adjusts parameters of software systems to optimize performance metrics while meeting the operating constraints set by the user. However, SmartConf is only applicable to parameters that have a linear relationship with performance; this is not necessarily the case for parameters of microservices [149]. BestConfig [183] uses sampling and search-based methods to tune the parameters of software systems. However, the sampling effort required increases exponentially with the number of parameters, suggesting that BestConfig is infeasible for microservices applications with a large configuration space. Fekry et al. [68] concentrate on

dynamically tuning configurations of data analytics frameworks for varying workloads and environments. While online tuning is an interesting research direction, it significantly limits the number of parameters available for (online) tuning. Alabed et al. [38] tune 10 parameters of RocksDB by optimizing multiple objectives using Bayesian optimization. The additional overhead of optimizing multiple objectives in place of one is addressed by reducing the dimensions of each optimization task. However, finding the low-level metrics and reducing the dimensionality of each optimization task requires expert knowledge of the system being tuned which is not feasible for microservices architecture due to the variety of microservices that are part of each application.

Resource allocation tuning. Bilal et al. [51] perform an exhaustive comparison of existing black-box techniques for the problem of finding the best cloud configuration that minimizes the execution time or cost. Vanir [50] optimizes the cloud configuration for analytics clusters using Mondrian forest-based performance model and transfer learning. OPTIMUSCLOUD [114] jointly optimizes VM configurations and database configurations for cloud-deployed database systems by training a performance prediction model. Kaminski et al. [94] employ black-box optimization algorithms to find cost-effective resource assignments while meeting performance targets for a multi-tenant, container-based cloud environment. CherryPick [39] uses Bayesian Optimization (BO) to build a performance model for Big Data systems, which is then used to find the best cloud configuration for these systems. Mostofi et al. [120] tune CPU allocations of a simple benchmarking application with three microservices. The configuration search space is too small compared to our work.

While some of the optimization algorithms explored in our evaluation are similar to the ones employed by the above works, we note that our focus is on tuning the *parameters of the numerous microservices that make up an application*, as opposed to only focusing on a handful of resource allocation parameters, such as number of CPUs, memory capacity, etc.

Performance management for microservices. There are several recent orthogonal works that aim to improve the performance of microservices applications via approaches other than configuration tuning. Sinan [178] is an ML-based CPU resource manager for microservices applications that takes into account short-term and long-term SLO conformance. Sage [71] is an unsupervised ML-based root cause analysis system that detects performance bottlenecks in microservices applications and then adjusts the allocation of the bottleneck resource at the identified microservice(s) to improve performance. FIRM [129] leverages tracing and telemetry data (from microservices) to find the critical path for an application and uses ML to detect and mitigate the bottleneck microservices. ATOM [77] leverages a layered queueing network model to assess the impact of horizontal (CPU) and vertical scaling on a given microservice; the scaling rules can then be determined for the application to optimize

the cost of deployment and application throughput. Similarly, HyScale [169] explores the combination of horizontal (CPU, memory, and network) and vertical scaling to improve the resource utilization of stateless microservices applications.

Reducing the configuration space. An orthogonal research direction is identifying configuration parameters that have the most effect on performance, thereby alleviating the configuration tuning effort by reducing the parameter search space.

Kanellis et al. [95] employ learning-based techniques to find the most important parameters of database systems that impact performance. Carver [55] employs Latin Hypercube Sampling to explore the effect of different parameters on storage system performance and uses the variance in performance caused by a parameter as an indicator of the parameter's importance. As discussed in Section 5.1.4, focusing on microservices on the critical path is a more effective approach. We reduce the dimensions by finding which of the microservices affect the performance the most and only tune such microservices as discussed in Section 5.1.4.

3.3 Problem Formulation and System Design

In this section, we formulate the microservices configuration setting problem as an optimization problem. We then describe our system design for the automated framework (which we have made publicly available [7]) that aids our experimental evaluation (presented in Section 5.1.4).

3.3.1 Microservices configuration setting problem

Let $f(c)$ denote the objective function (or performance metric) for the microservices application under the configuration c; here, c is the (potentially large) vector of parameter settings for all tunable parameters of all microservices. Note that a parameter refers to a configurable option and a configuration is a combination of parameter values. Let C denote the set of all configurations, i.e., all feasible values that vector c can take. Finally, let $c_{opt} \in C$ denote the configuration that minimizes the performance metric, $f()$. Thus, $c_{opt} = \text{argmin}_{c \in C} f(c)$. We could consider metrics that need to be maximized by minimizing the negative of the objective function. Our problem statement is *to find c_{opt} or a near-optimal configuration*. We focus on the realistic case where no assumptions can be made on the structure of $f()$ or on the availability of offline training data. We further assert, for practical purposes, that the (near-)optimal configuration should be determined in a reasonable amount of time.

While $f()$ can represent any metric of interest, including combinations of metrics, we consider the 95^{th} percentile of end-to-end application latency to be our metric, $f()$. We note

that customer-facing applications often employ such tail latency metrics to assess application performance [62, 61].

Given the dependencies between parameters and the possible non-linear relationship between performance and parameter values (as described in Section 5.1.1), it is unlikely that $f()$ can be determined or inferred accurately. Thus, classic convex optimization techniques cannot be readily applied to determine c_{opt}. However, for a given c, the value of $f(c)$ can be observed or measured by setting the parameter values in c for the microservices and running an experiment. This suggests that black-box optimization techniques, that iteratively observe the value of $f()$ at a given c and determine the next configuration value c' to explore, can be applied to find c_{opt} or near-optimal c values.

3.3.2 Automated framework to aid optimization

Unlike prior works [56, 51] that run optimization algorithms over readily available datasets, we evaluate the value of the objective function, $f()$, by running an experiment. To streamline the iterative exploration of configurations (for determining c_{opt}), we thus require a robust framework that can automatically: (i) configure the parameters of the microservices selected by the dimensionality reduction technique and run the application with these parameter settings, (ii) collect the required metrics, and (iii) run the optimization algorithm to obtain the next configuration to experiment with.

Figure 3.2 illustrates the design of our automated framework that we use to conduct our experiments. The *application deployment file* has the information necessary to create the docker-compose files, viz., the list of microservices, their images, the host details, environment variables, etc. The *parameters file* contains the list of parameters being tuned and their range. The size of this list depends on the *dimensionality reduction* method being employed. The *controller* passes the value of the measured objective function, $f(c(i))$, of the current iteration, i, and queries the *optimizer* for the next configuration setting, $c(i+1)$.

The *optimizer*, in its first iteration, queries the *dimensionality reduction module* to obtain a subset of the microservices parameters that will be subject to optimization. The *dimensionality reduction module* uses the *application deployment file*, *parameters file*, and the *request traces* to pass a *reduced list* of parameters to the optimizer. The dimensionality reduction techniques are discussed in Section 3.4.2. The *optimizer* then generates, via the optimization algorithm, the next configuration setting, $c(i+1)$, for the *reduced list* of parameters.

Using the details in the *application deployment file* and the $c(i+1)$ configuration passed by the *optimizer*, the *controller* generates *docker-compose files* on the fly with the necessary network settings and mounts. The application is then deployed on the servers using these

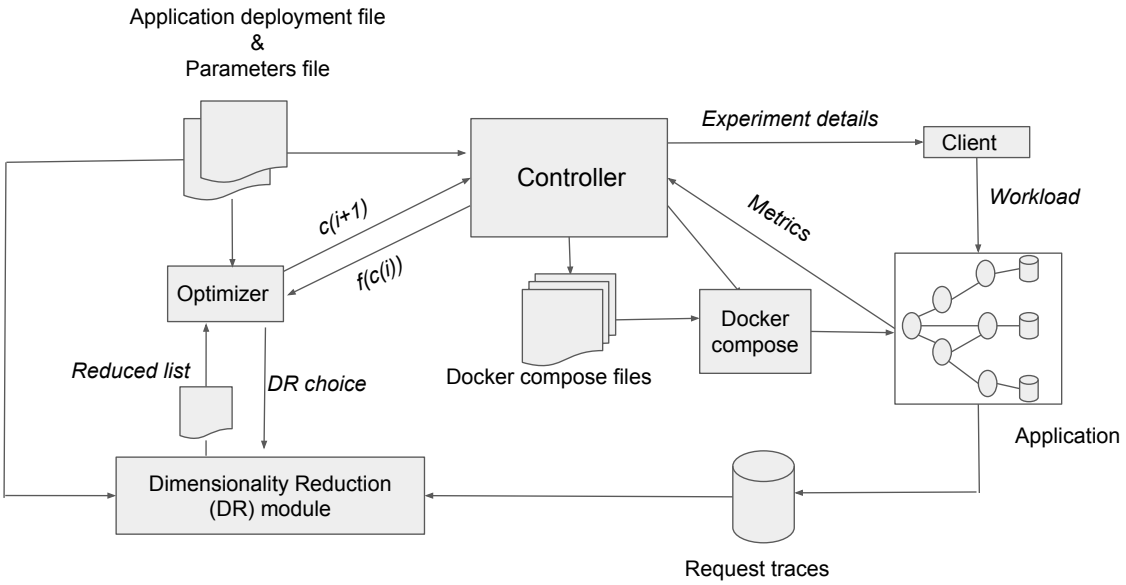


Fig. 3.2 Illustration of our solution framework. $f()$ is the objective function or performance metric of interest and $c(i)$ is the configuration setting for iteration i. The *optimizer* takes in the observed objective function value for a configuration, $f(c(i))$, and outputs the next configuration to employ, $c(i+1)$. The *dimensionality reduction module* trims the configuration vector size to speed up the optimization process. The *controller* interfaces with the application and invokes the execution based on the required configuration.

docker-compose files and the *client* sends the workload to the application. The *request traces* are collected by a tracing framework and the latency metrics are calculated by the *client*. These metrics are passed to the *controller* which then calculates the objective function, $f(c(i+1))$, and repeats the process iteratively until a good enough configuration is found or until an exploration time limit is reached. Our framework supports any combination of average, median, or tail latency for the objective function.

The framework currently supports automatic configuration management for the most widely used microservices [100]: Memcached, Redis, MongoDB, MySQL, Nginx, and microservices implemented using the thrift framework. The parameters of some of these microservices can be modified by creating a configuration file (e.g., for Nginx) whereas others expect them as command-line arguments with varying syntax. The user can be agnostic to these intricacies and treat all parameters similarly. The framework can be employed for any microservices application consisting of the supported microservices by including the *application deployment file* for that application. Optimization algorithms can be added by inheriting the *Optimizer* class and implementing its methods.

3.4 Evaluation

In this section, we first discuss our experimental setup and methodology and then present our experimental results. Our evaluation goal is to (i) investigate the efficacy of various optimization algorithms with respect to their running time and their ability to improve tail latency by configuration tuning; and (ii) investigate dimensionality reduction techniques that can speed up the optimization algorithms in practice.

3.4.1 Experimental setup

We use a cluster with four servers, each with 24 (hyper)cores, 40 GB of memory, and 250GB of disk space. We deploy the microservices of the application on these servers based on their functionality: one hosts back-end microservices as is the practice in industry [112], one server hosts front-end microservices, one hosts the microservices that implement the logic, and one server is dedicated for monitoring the microservices and the application performance. We restrict monitoring services, Jaeger [9] with Elasticsearch [6] back-end, to a different server to avoid interference with the application. *docker-compose* is used to deploy the application and *overlay* network connects the microservices across the servers.

Applications. We employ the *social networking* and *media microservices* applications from the DeathStarBench benchmark suite [73] and *train ticket* [181] application to evaluate the efficacy of different black-box optimization algorithms.

The social networking application has 28 microservices that together implement several features of real-world social networking applications. The constituent microservices are Nginx, Memcached, MongoDB, Redis, as well as microservices that implement the logic of the application. The application workload consists of 10% requests that create a post, 30% requests that read the timeline of other users, and 60% requests that read the user's own timeline.

The media microservices application implements a movie review system and consists of 31 microservices. The constituent microservices are similar to the ones in the social networking application. The workload consists of 25% requests that add a movie review, 70% requests that read a movie review, and 5% requests that read the plot of the movie.

The train ticket application is a train ticket booking system implemented using 41 microservices. In addition to the microservices that are part of the social networking application, this application also uses MySQL microservice. The application workload consists of 50% of requests that search for a train between two stations, and 50% of requests that reserve a train ticket.

We change the type of server for social networking and media microservices applications to *TNonblockingServer*. The Apache Thrift C++ *TNonblockingServer* provides better

performance and exposes numerous settings for the developer to customize the server [33]. We also make modifications to change the thread pool size dynamically based on the value suggested by the optimizer for each iteration.

3.4.2 Evaluation methodology

For evaluation, we consider the *95^{th} percentile of latency* as the performance metric; other latency metrics can be readily used as well. For each microservice, we select at most five parameters to tune; we refer to product documentation [2, 16, 22, 14, 33] to identify the performance-impacting parameters. Our framework supports parameters that can take continuous (e.g., factor parameter of memcached), discrete (e.g., number of processes in Nginx), or categorical values (e.g., maxmemory-policy in Redis). The range of allowed values for each parameter is decided based on product documentation (e.g., internal cache size of mongoDB) or the limits of the hardware (e.g., number of threads in memcached).

We report results averaged across multiple experimental runs and provide error bars where appropriate. Each run lasts for 20 minutes, with the first few minutes (5 minutes for social networking and media microservices and 10 minutes for train ticket) considered as warm up until the cache hit rate stabilizes. Performance metrics are collected after the warm up period.

Black-box optimization algorithms

We consider six existing representative optimization algorithms in our evaluation, and then propose a seventh hybrid algorithm based on our analysis of the existing six algorithms. The first 2 are representative of *heuristic-based probabilistic algorithms*, the next 2 are evolutionary algorithms inspired by population-based biological evolution, and the next 2 are *sequential model-based optimization algorithms* that approximate the objective function with a cheaper, surrogate function [47] to aid optimization. We use skopt [32], Hyperopt [48], and Nevergrad [131] libraries to implement the algorithms. We also compare the results of these algorithms with the best configuration obtained by performing a random search of the configuration space. Note that we also tried tuning one microservice at a time (as opposed to a joint tuning), but the results are inferior and are so omitted.

Simulated Annealing (SA) [110] exploits the neighbourhood points based on the value of the objective function at these points, with the degree of exploration determined by a time-varying parameter that decreases with each iteration (annealing). Since SA is known to be better at global optimization than the hill climbing algorithm [110], we do not evaluate the latter.

Dynamically Dimensioned Search (DDS) starts with an initial configuration and perturbs the values of the parameters of the configuration based on a perturbation factor [158]. With

each iteration, the probability of each parameter being included in the optimization reduces uniformly, thereby reducing the search space.

Particle Swarm Optimization (PSO) [110] works by moving a population (called swarm) of candidate solutions (called particles) around the search space depending on the particle's best-known position and the global best position.

Genetic Algorithms (GA) [110] mimic natural selection by first selecting a subset of candidate solutions based on the objective function value and then randomly changing the configurations of some parameters (mutation) and combining configurations of the candidates (crossover) to generate new candidates.

Bayesian Optimization (BO) starts with a prior distribution of the search space guided by the surrogate; we experiment with the popular Gaussian Process (GP) [47], Gradient Boosted Regression Trees (GBRT) [51], and Random Forests (RF) [51] surrogate models. The posterior distribution is updated at each step of exploration using Bayesian method.

Tree-structured Parzen Estimator (TPE) is similar to BO, but models the likelihood and prior instead of the posterior [47].

Hybrid algorithm is a new algorithm that we construct by combining the strengths of BO and DDS. BO models the relationship between performance and the parameters to efficiently search for the optimal configuration with a convergence rate that is dependent on the initial samples [38]. On the other hand, DDS is a computationally efficient heuristic-based search algorithm that performs well (See Section 5.1.4). Since DDS is not model-based, it makes no attempt to learn about the parameter space. With hybrid, we combine the *light-weight searching feature* of DDS with the *model-based searching feature* of BO. Specifically, the DDS algorithm is run for a fixed number of iterations and the resulting best configurations are used as initial samples for the Bayesian algorithm with the popular Gaussian Process as the surrogate model [39, 50]. By contrast, when not using hybrid, the initial samples for Bayesian are (by default) randomly generated.

Dimensionality reduction strategies

If an application has m microservices each with p_i parameters (for $i = 1, 2, \cdots, m$), then the number of dimensions in a configuration vector c is $n = \sum_{i=1}^{m} p_i$. For the purpose of illustration, if each parameter can take v different values, then the number of possible configurations is $|C| = v^n$. Clearly, the search space of configurations grows exponentially with the number of microservices. To reduce the search space, we thus consider strategies that allow us to focus our configuration tuning effort on only a subset of the microservices. Another advantage of dimensionality reduction is that several optimization algorithms, such as Bayesian Optimization (BO), do not work well in high dimensions (number of tunable parameters, in our case) [119]. We note that our dimensionality reduction strategies have a

different goal than those used in the machine learning community since our focus is on using system characteristics to reduce dimensions in a practical manner. For example, Principal Component Analysis (PCA) [118] can reduce the configuration space dimensions but would make it difficult to reconstruct the configuration value after optimization.

1. ***Critical path.*** In the call graph of a request, the critical path is the path formed by microservices that determine the latency of the request. Tuning the parameters of the microservices that fall on the critical path of a request is important as any performance improvements in these microservices will reduce the end-to-end latency of the request. Algorithm 1 provides an overview of our critical path determination algorithm. The algorithm takes the request traces as input T and outputs a list of microservices that form the critical path of each trace. In summary, the algorithm traverses the call graph of a request to find all the microservices on the critical path that have non-negligible latency (at least 1ms).

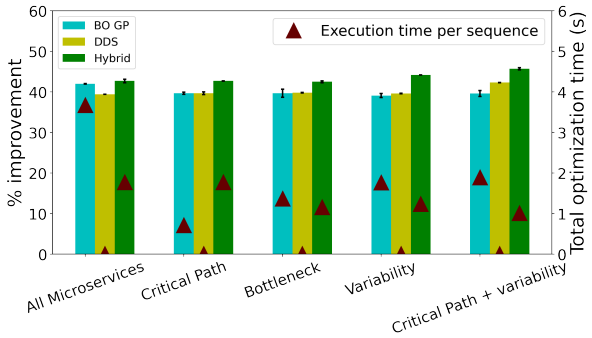
   We rely on the service time (or span) measurements provided by Jaeger for each microservice to determine the critical path. Using our algorithm, we identify microservices present on the critical path of most of the request types for all applications.

   Algorithm 1. *Find microservices along the Critical Path.*

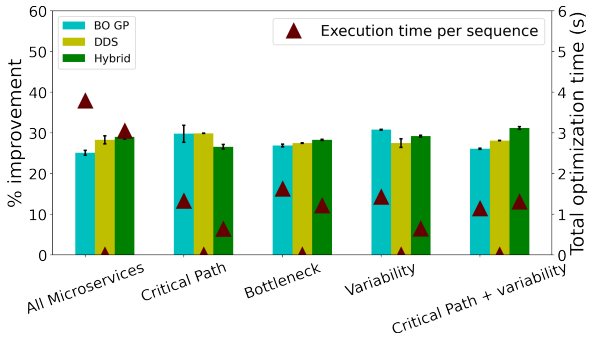
   ```
   Input: Request traces, T.
   Output: List of microservices along the Critical Path.
   criticalPathAll ← ∅
   for t ∈ T do
       currentCriticalPath ← getCriticalPath(t.root)
       append(criticalPathAll,currentCriticalPath)
   procedure GETCRITICALPATH(node)
       criticalPath ← ∅
       if node.children is NULL then
           lastChild = nextChild(node)
           getCriticalPath(lastChild)
           node.duration = updateDuration(node)
       if node.duration > 1ms then
           append(criticalPath, node)
   ```

   end

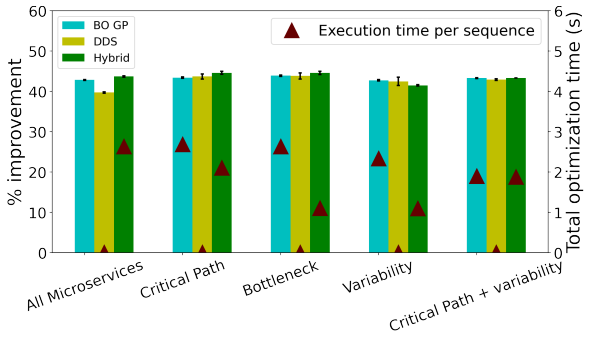
2. ***Bottlenecks.*** FIRM [129] uses a Support Vector Machine (SVM) to detect microservices that could be potential bottlenecks for an application.



(a) Social networking application.



(b) Media microservices application.



(c) Train ticket application.

Fig. 3.3 Evaluation of different dimensionality reduction techniques with respect to improvement in latency over the default configuration under different optimization algorithms. Error bars indicate the standard deviation around the reported mean over 3 runs. The total optimization time is the time taken by the algorithm across the 15 iterations (excluding the time to run the application with the required configurations).

We train an SVM model using the publicly available tracing data [128] for the social networking, media microservices, and the train ticket applications. We use this model to predict potential bottlenecks in all applications and tune only these.

3. ***Performance variance.*** Reducing the source of performance variance can improve the system performance [61, 151]. Accordingly, we consider configuration tuning only for

microservices that have a high service time coefficient of variation (above 0.5 in our experiments).

4. ***Performance variance along the critical path.*** To combine the strengths of different dimensionality reduction techniques, we consider the approach of first determining the critical path (via Algorithm 1) and then selecting the top five microservices on the critical path that have the highest variance in service time.

3.4.3 Experimental results

In practice, the optimization algorithms cannot be run indefinitely. Unless otherwise specified, we limit the number of configurations to be explored for each optimization algorithm to 15. We note that running each iteration of the algorithm involves bringing up the application, applying the configuration, and running the workload, which together takes about half an hour. By contrast, the time taken by an optimization algorithm to suggest a new configuration is typically in the order of seconds. Thus, a budget on the optimization time as a stopping criteria is not as practical as the number of iterations of the algorithm. For initialization, the optimization algorithms, except Hybrid, start with a random configuration. For the evaluation to be fair, we initialize all the algorithms with the same random samples. Note that (re)setting the configuration parameters between iterations does incur some overhead and may require restarting some microservices; during this time, the application may be momentarily offline. We acknowledge that this can be concerning for production deployments where application downtime is not tolerated. However, in a production deployment, the reconfiguration step can be carried out during planned maintenance or upgrade windows to avoid additional disruption to the application [20]. We defer online configuration tuning of microservices to future work.

subsubsectionEfficacy of dimensionality reduction strategies Figure 3.3 shows the percentage improvement in tail (95^{th} percentile) latency of all applications under different dimensionality reduction techniques, compared to the tail latency when using the default configuration for all parameters. For ease of illustration, we show results for three specific optimization algorithms. Note that comparison across optimization algorithms will be discussed in the next subsection and is not the focus here. Error bars in the figures indicate the standard deviation around the reported mean results.

In Figure 3.3a, we see that tuning all 28 microservices of the social networking application provides about 39–43% improvement in tail latency. Tuning all the microservices on the critical path provides similar improvements. However, tuning only the microservices on the critical path that show high variability (5 microservices) provides 40–46% improvement. Note that this improvement is greater than that obtained by tuning all 28 microservices. This is because dimensionality reduction reduces the configuration search space, enabling a

more efficient tuning within the budget of 15 configurations to explore. Tuning the known bottlenecks provides around 42% improvements, suggesting that the critical path approach correctly identifies the microservices that have the most impact. Finally, by focusing on the variability causing microservices, the latency improvement is about 39–44%.

In Figure 3.3b, we observe that tuning all the 31 microservices of the media microservices application produces about 25–29% improvements. We observe that tuning only the microservices on the critical path that show high variability (5 microservices) again provides superior performance with up to 31.2% improvement, highlighting the impact of dimensionality reduction. The performance improvements for the critical path, the bottleneck, and the variability techniques are 28–30%, 27–28%, and 28–30%, respectively

In Figure 3.3c, we see that tuning the 26 microservices of train ticket application results in 39–43% improvement. Tuning only the microservices on the critical path provides up to 46% improvement in tail latency. Since the train ticket application has the most parameters, the benefits of dimensionality reduction are more pronounced. The performance improvements are around 44%, 42%, and 43% for bottleneck, variability, and critical path+variability, respectively.

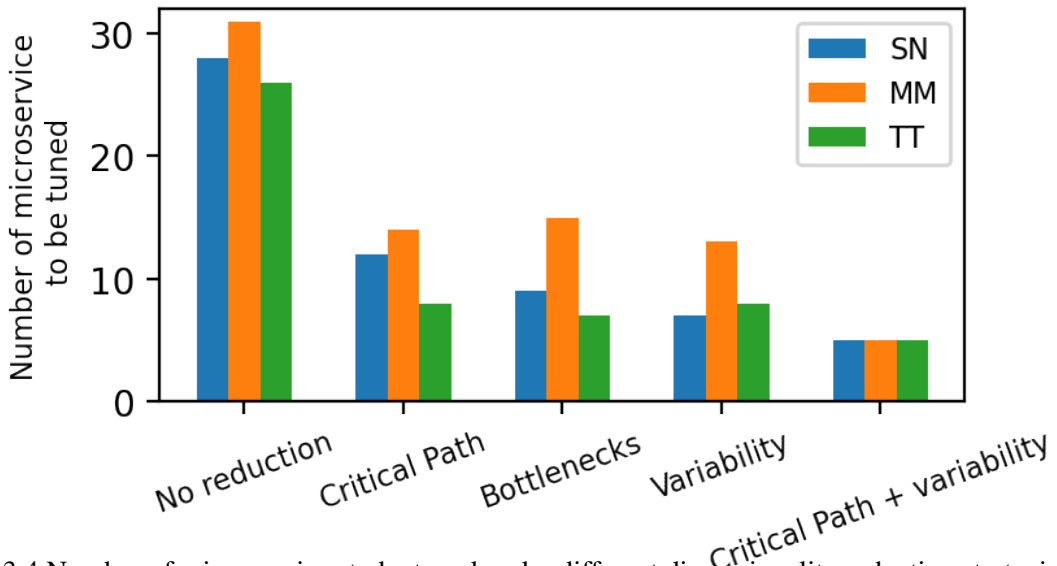


Fig. 3.4 Number of microservices to be tuned under different dimensionality reduction strategies for social networking (SN), media microservices (MM), and train ticket (TT).

Figure 3.4 shows the number of microservices tuned under different dimensionality reduction techniques, compared to no reduction, for all the three applications. While all techniques reduce the number of microservices to be tuned by at least 50%, the "Critical path + variability" approach (Performance variance along the critical path) allows us to customize and aggressively reduce the number to just 5. Despite this substantial reduction in the number

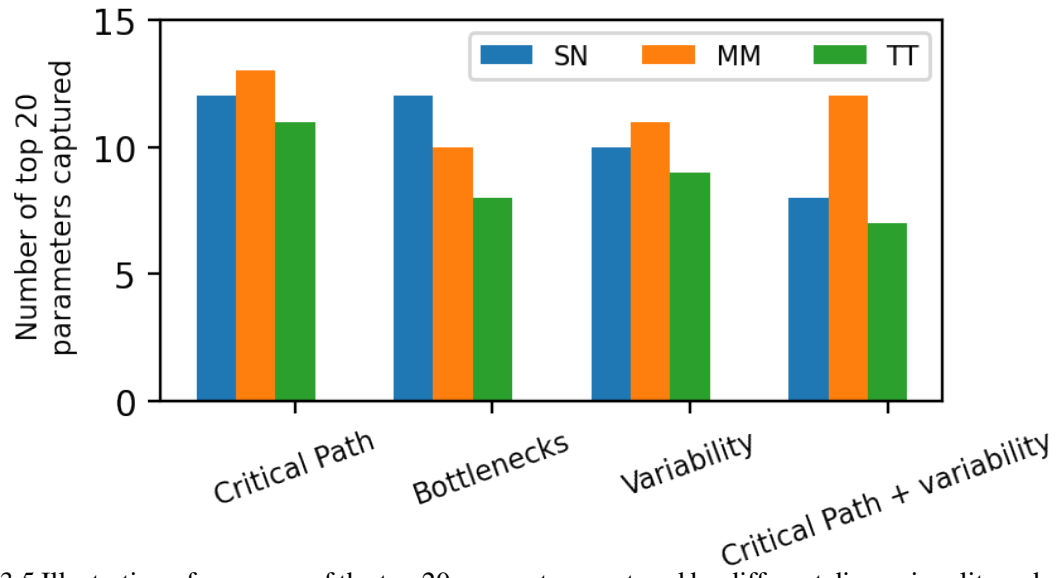
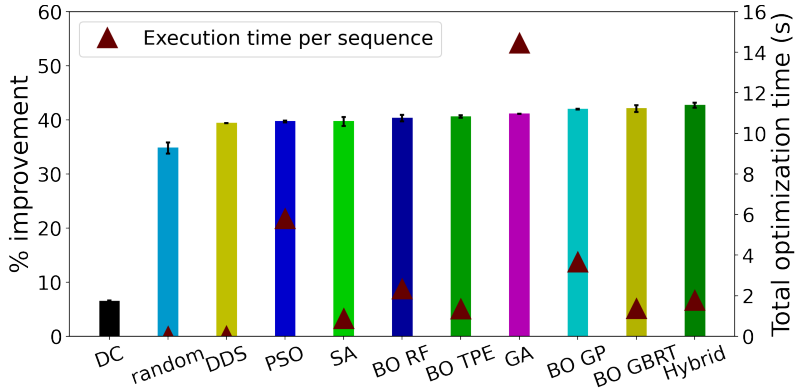


Fig. 3.5 Illustration of coverage of the top 20 parameters captured by different dimensionality reduction techniques.

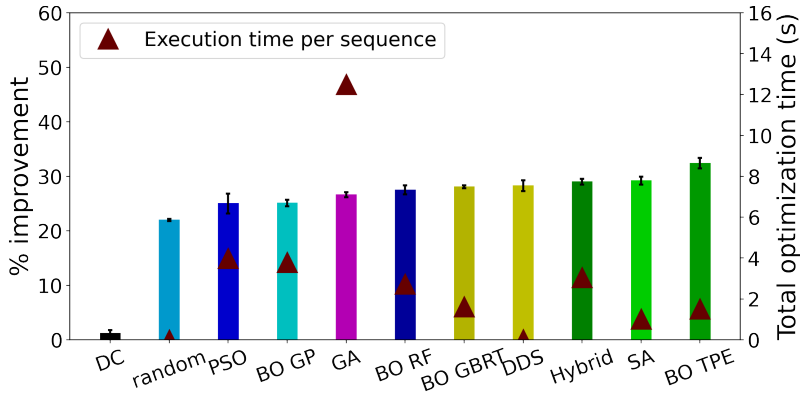
of microservices to be tuned, the "Critical path + variability" approach provides significant tail latency reduction for the applications we consider, as highlighted in Figure 3.3.

To further contrast the four different dimensionality reduction techniques, we consider the overlap in subsets of microservices chosen by the techniques. For the social networking application, we find that only two microservices are common among all the subsets: (i) *post-storage-memcached* is an important microservice as it caches posts that are read by requests that constitute 90% of the workload; and (ii) *compose-post-service* is critical in the call graph of the request that writes posts as it is called multiple times per request. This shows that, despite differences in the subsets, all techniques have the ability to identify some of the important, performance-impacting microservices.

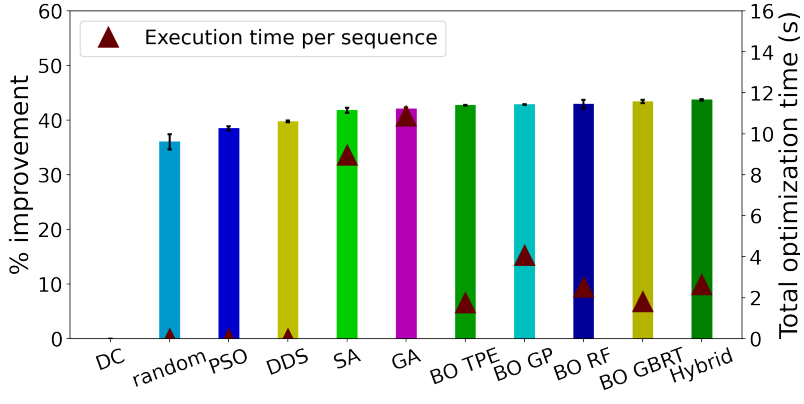
A potential drawback of reducing the dimensions by omitting microservices for optimization is that a dimensionality reduction technique could miss out on important parameters. To evaluate this hypothesis, we find the 20 most important parameters using the offline and expensive fANOVA [88] approach and determine how many of these 20 parameters are captured by different dimensionality reduction techniques in Figure 3.5. We find that while the different dimensionality reduction techniques do not capture all 20 important parameters, they do capture 3–4 parameters out of the top 5. We note that fANOVA parameter importance analysis can be used to reduce the number of dimensions, but the amount of training data and effort required makes this approach impractical.



(a) Social networking application.



(b) Media microservices application.



(c) Train ticket application.

Fig. 3.6 Improvement in latency compared to default configuration (left y-axis) and the time incurred by the optimization (right y-axis) for all algorithms when tuning the microservices of the applications with no dimensionality reduction.

Comparing different optimization algorithms

Figures 3.6a, 3.6b, and 3.6c show the (sorted) percentage improvement (on left y-axis) in tail latency over the default configuration afforded by different optimization algorithms with no dimensionality reduction for the social networking, media microservices, and the train ticket

applications, respectively. For comparison, we show (as DC) the improvement afforded by the configuration employed by the developers of the DeathStarBench [5] and the train ticket application [181].

For the social networking application in Figure 3.6a, we see that Hybrid algorithm provides the best improvement of around 43%, followed closely by BO GBRT (42%) and BO GP (41.8%). Using the configuration chosen by the developers provides a modest improvement of 6% over the default configuration. To evaluate the overhead of different optimization algorithms, we plot (as red triangles with right y-axis) the time taken by the optimization across all iterations in Figure 3.6. We find that DDS requires the least amount of time (10ms), followed by SA (0.8s) and BO TPE (1.4s). Hybrid is also relatively quick, requiring about 1.7s. GA and PSO incur a high overhead; this is expected as evolutionary algorithms are computationally intensive.

For the media microservices application, as seen in Figure 3.6b, the BO TPE algorithm provides the best configuration with an improvement of around 32%. DDS again takes the least amount of time, about 9ms. The Hybrid algorithm also performs well, with an improvement of around 29% and requiring about 3s of time. Using the configuration provided by the developers only provides a nominal 2% improvement over the default configuration.

For the train ticket application, the Hybrid algorithm again performs the best with 43% improvement over the default configuration, closely followed by BO GBRT (42.84%) and BO RF (42.72%). The developer's configuration performs worse than the default configuration because of which it is excluded from Figure 3.6c. It is interesting to note the impressive performance of random search (36% improvement) considering the negligible run time (~ 1ms). The existence of multiple optimal regions, as discussed in Section 5.1.4, is one likely reason for its good performance. Further, randomized configuration settings have been shown to perform well when tuning databases [56, 38].

Based on the above results, we conclude that, for our evaluation, Hybrid is the best performing algorithm for the social networking and train ticket applications whereas Tree-structured Parzen Estimator (TPE) provides a good tradeoff between latency improvement and optimization runtime for media microservices.

Convergence analysis of algorithms

The results shown thus far are based on the best configuration picked by the algorithms from among 15 iterations. To analyze the significance of number of iterations and variance across different sequences (runs), we plot the best improvement afforded until different iterations for BO GP and Hybrid, across 3 different sequences of these algorithms, in Figure 3.7 for the social networking application. Although the different sequences vary during the

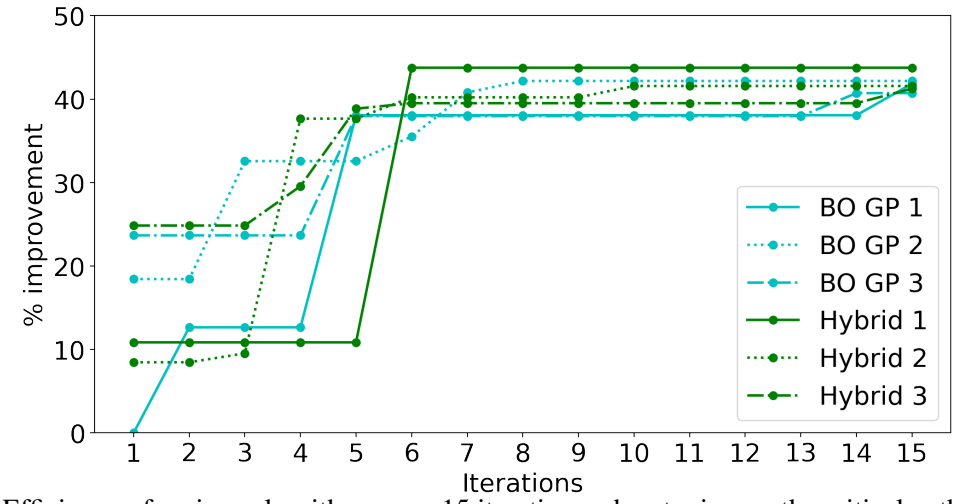


Fig. 3.7 Efficiency of various algorithms over 15 iterations when tuning on the critical path of social networking application.

initial iterations, they eventually converge well within 15 iterations. This suggests that the variability *between* runs is low, explaining the narrow error bars in our results.

We also analyzed the results for 100 iterations and found that the additional performance benefit afforded over 15 iterations is only about 1–2% compared to the best solution in Figure 3.6, suggesting that the optimization algorithms converge quickly. This is useful in practice given that each additional iteration imposes certain overhead and application downtime.

Significance of initial configuration

The optimization algorithms typically start with a randomly sampled configuration. To assess the significance of this initial configuration on performance improvement and convergence, we specifically set the initial configuration of the social networking application to one that we know performs poorly to check how the optimization recovers; we use BO GP for this evaluation. For example, we limit the number of processes for the Nginx microservice to 1, set the Memcached cache size to 16MB, etc. We find that, despite the poor initial configuration, the algorithm does provide significant improvement over the default configuration, with only a 3.4% relative drop in performance compared to the randomly chosen initial configuration case.

Analysis of configurations set by algorithms

To better understand the optimal configurations, we now analyze the specific parameter configuration values determined by different algorithms. Without loss of generality, we consider the social networking application and analyze the values selected by each algorithm for the top 5 important parameters. To identify the important parameters, we employ

Parameter (associated microservice)	Range	Default	Optimization algorithms								
			BO GP	BO GBRT	TPE	DDS	PSO	SA	hybrid	GA	BO RF
worker_processes (frontend-nginx)	1-48	24	24	21	23	19	20	26	23	19	20
zset-max-ziplist-entries (social-graph-redis)	64-512	128	64	108	77	68	92	238	109	120	89
io_threadpool_size (social-graph-service)	1-48	13	33	36	43	34	33	46	29	33	30
memory_limit (MB) (post-storage-memc)	32-20k	64	4k	6.8k	5.8k	7k	8k	9.1k	8.8k	13k	6.4k
hz (user-timeline-redis)	1-100	10	43	64	60	57	41	61	52	71	39

Table 3.1 Top-5 important parameters (as identified by fANOVA analysis) for the social networking application.

fANOVA [88], which uses an empirical performance model based on random forests to analyze how much of the observed performance variation in the configuration space is explained by a single parameter or combinations of few parameters. To obtain the data for fANOVA, we sample the configuration space by running up to 1000 experiments for various configurations and collecting the corresponding 95^{th} percentile latency.

For the social networking application, the top 5 parameters (along with the associated microservice), in the order of importance, and the values assigned to them by each algorithm, are given in Table 3.1. The top parameter is the *worker_processes* parameter of the *frontend* microservice (NGINX). While the default value of this parameter is 1, for a fair comparison, we override the default value to the number of cores in the server (24) as suggested in product documentation [2]. We set the allowable range for this parameter to be 1–48. As seen in Table 3.1, the values set by different algorithms are close to 24. Since the worker processes for the social networking application do not perform any I/O, a high value for *worker_process* would lead to contention and a low value would lead to decreased processor utilization. This shows that all algorithms judiciously choose the *worker_process* value.

The second most important parameter is the *zset-max-ziplist-entries* of the social-graph-redis microservice. This parameter sets a limit on the number of entries allowed in a ZSET (sorted sets) for it to be encoded as a ziplist. The memory savings due to ziplist come at the cost of CPU usage—CPU cycles are spent on decoding every read, partially re-encoding every write, and may require moving data in memory. As seen from the table, the value of this parameter is always set below the default, except for the one generated by SA, signifying the benefits of prioritizing CPU over memory savings.

The next important parameter is the *io_threadpool_size* of the *social-graph-service*, which dictates the size of the I/O thread pool for TNonblockingServer [33]. We see that the *io_threadpool_size* value selected by different algorithms is consistently higher than the default value (13), suggesting that the default configuration was under-utilizing the resources.

The next important parameter is the *memory_limit* value for *post-storage-memcached* micorservice, which is set at 64MB by default. Table 3.1 shows that the various optimization algorithms have a *memory_limit* value of at least 4GB for this microservice. This is a critical

microservice that is along the critical path of 90% of the requests, substantiating the extra memory allocation to improve performance.

Finally, the *hz* parameter sets the frequency of invocations of background tasks to remove expired keys in Redis [22]. For the *user-timeline-redis* microservice, the value selected for *hz* is much higher than the default of 10, indicating that the additional use of CPU by this microservice (at the expense of other microservices) is worth the improvement in performance.

Table 3.1 highlights the similarities (e.g., for *worker_processes*) and differences (e.g., for *zset-max-ziplist-entries*) in the parameter values chosen by the optimization algorithms. The differences suggest that the algorithms do converge to different locally optimal configurations (as opposed to a single globally optimal one); despite the different configurations, the resulting latency benefits are comparable (as seen in Figures 3.6a, 3.6b, and 3.6c). The similarities suggest that minor differences in values (within a range) of some parameters may not significantly impact performance; a valuable future direction is to discretize some of the parameter ranges to reduce the configuration space.

Microservice-level analysis of latency reduction afforded by the best configuration

The workloads used in our experiments consist of a mixture of different request types (see Section 3.4.1). The 95^{th} percentile latency depends heavily on the request type that takes the longest time. To analyze the ability of optimizations in prioritizing microservices that serve the long-tailed request types, we compare the service time (95^{th} percentile) of all microservices along the call graph of different request types for the best configuration across all experiments (i.e., across all algorithms and all dimensionality reduction strategies) with the default configuration.

For the social networking workload, based on the experiment logs, we find that read-user-timeline requests influence the 95^{th} percentile of the workload latency the most, followed by read-home-timeline. *post-storage-memcached* and *post-storage-mongo* microservices, which are along the critical path of both these request types, can thus have a significant impact on workload latency. In case of the read-user-timeline request type, the best configuration results in 65% and 28% reduction in service time of *post-storage-memcached* and *post-storage-mongo* microservices, respectively. The *user-timeline-redis*, which is on the critical path of read-user-timeline, sees a 56% reduction in its service time. On the other hand, the microservices along the call graph of light-tailed compose-post request type experience a nominal *increase* in service time, notably *user-timeline-mongo* (7% increase), where the user's post IDs are written as part of the compose-post request type. For the *user-timeline-mongo* microservice, the best configuration across all experiments chooses *zlib* as the compression algorithm which uses more CPU than the default (*snappy*) [16]. Likewise,

the best configuration sets *wiredTigerConcurrentWriteTransactions* to 74 (lower than the default of 128), limiting the maximum concurrent writes, and increasing the service time of *user-timeline-mongo*.

We find similar patterns (of prioritizing parameters of microservices that serve heavy-tailed request types) for other applications as well. For example, for media microservices, compose-movie-review request type influences the workload's 95^{th} percentile latency the most. *compose-review-service* microservice, which is along the critical path of the compose-movie-review requests, sees a 35% reduction in 95^{th} percentile of the service time when the best configuration across all experiments is applied. The key takeaway here is that despite the optimization algorithms being oblivious to the workload mix, they sample the search space well enough to find configurations that are near-optimal for the workload.

3.5 Conclusion

Despite the recent shift in application design to microservices architecture, the fundamental problem of setting the configuration of individual microservices to improve performance has received very little attention, with practitioners instead settling for sub-optimal performance via default or ad-hoc configuration settings. This work *makes the case for configuration tuning of microservices*.

Our investigation of different joint optimization techniques shows that significant improvements in tail latency, up to 46%, can be realized via configuration tuning. While most algorithms perform well, the optimal algorithm is application-dependent; further, combining different algorithms can provide superior performance for some applications. Our analysis reveals that the optimal configuration of a microservice (e.g., MongoDB) need not be the same across applications or even across instances within the same application.

We also investigate techniques to reduce the tuning effort across algorithms. We consider different approaches to dimensionality reduction and find that focusing on tuning the microservices on the critical path that have the highest service time variability is an effective dimensionality reduction technique.

We conclude that dimensionality reduction based on system characteristics is an effective approach to the otherwise intractable problem of optimizing a large state space.

Chapter 4

Post-deployment Configuration Tuning

Real-world application/service deployments have hundreds to thousands of inter-dependent configuration parameters, many of which significantly influence performance and efficiency. With today's complex and dynamic services, operators need to continuously monitor and set the right configuration values (*configuration tuning*) well after a service is widely deployed. This is challenging since experimenting with different configurations post-deployment may reduce application performance or even disrupt its functions. Existing approaches to this problem use ML algorithms to automatically change configuration values, but they do not address several other important steps needed to effectively tune configuration in deployed applications.

This chapter presents SelfTune and OPPerTune, frameworks that enable configuration tuning of applications in deployment. SelfTune uses domain expertise to tune cluster managers and complex distributed applications. OPPerTune reduces application interruptions while maximizing the performance of deployed applications as and when the workload or the underlying infrastructure changes. It automates three essential processes that facilitate post-deployment configuration tuning: a) it determines which configurations to tune, b) it automatically manages the scope at which to tune the configurations, and c) it uses a novel reinforcement learning algorithm to simultaneously tune numerical and categorical configurations fast, thereby keeping the overhead of configuration tuning low.

We first briefly discuss SelfTune and its application to tuning Kubernetes' Vertical Pod Autoscaler and DeathStarBench benchmarking application. Following this, we discuss OPPerTune, its design and evaluation in detail.

4.1 Introduction: SelfTune

Large cloud services depend upon cluster managers such as Protean [82], Borg [160], Twine [156], and Kubernetes [10] for job scheduling [137, 63, 125, 64, 92], virtual machine

pre-provisioning [111], and resource autoscaling [109, 132, 135]. 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 [12] react promptly to changes in cluster utilization without reacting to extremely ephemeral changes in utilization.

Cluster manager	Parameter	Description	Default
Kubernetes (Vertical Pod Autoscaler)	`cpu-histogram-decay-half-life`	How long to wait before halving the weights of past CPU measurements	24 hours
	`recommendation-margin-fraction`	Fraction of usage added as the safety margin to the recommended request	0.15
	`pod-recommendation-min-cpu`	Minimum CPU recommendation for a pod	25 millicores
	`history-length`	Window length for CPU utilization histogram	24 hours
	`pod-recommendation-min-memory`	Minimum memory recommendation for a pod	250 MB
	`memory-histogram-decay-half-life`	How long to wait before halving the weights of past memory measurements	24 hours
	`memory-aggregation-interval`	Window length for memory utilization histogram	24 hours
	`recommender-interval`	How often the resource utilization metrics should be fetched	1 minute
Azure FaaS (App manager)	`prewarm`	Time to wait before pre-loading function code	5
	`keepalive`	Time to wait before retiring the loaded VM	99
Azure Protean (VM allocator)	`num-aa`	Number of rule-based VM allocation agents	
	`k`	*k*-highest quality clusters for VM placement	[8,16]

Table 4.1 Key numerical configuration parameters of popular cluster management frameworks.

Every cluster manager relies on developers[1] to manually set these configuration management parameters to "suitable" values. Table 4.1 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 [87, 157, 155]. 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.

To address this problem, we observe an interesting similarity between cluster manager algorithms and reinforcement learning (RL) algorithms. Cluster managers (examples in Table 4.1) often use *state reconciliation*: periodically, they observe the current state of a

[1]For brevity, we refer to anyone developing, deploying or monitoring cluster managers – developers, operators, service engineers – as "developers".

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 [54]. For instance, the Kubernetes autoscaler [12] continuously determines how to update container sizes, by maintaining a histogram of recent resource utilization values. RL algorithms are also iterative in nature, and use "rewards" to periodically improve and converge a system to an optimal state. Hence we observe that cluster managers are naturally amenable to RL techniques for tuning configuration parameters.

In this chapter, we propose SelfTune, a framework that automatically tunes such configuration parameters *in deployment*, rather than through testing. Three key aspects of our framework are: (i) SelfTune piggybacks *solely* on cluster manager's periodic metric measurements, to help tune the cluster manager parameters, so that both tuning and the cluster state reconciliation can occur simultaneously with the same goal of moving the cluster *continuously* towards optimal state; (ii) SelfTune provides a light-weight API for the developers to augment the cluster manager code specifying which parameters to tune, and an objective, e.g., average CPU utilization should be at least 60% but less than 90%; and (iii) SelfTune uses a principled RL algorithm called Bluefin, based on theoretically-founded ideas for time-varying rewards [70, 136], to optimize the developer-specified objective; it gradually explores "perturbed" choices for the cluster manager parameters, observes the cluster state, and iteratively tunes the parameters to achieve the objective.

We have deployed SelfTune on WLM, a scheduler which manages background job scheduling for many Microsoft M365 services including Exchange Online. WLM runs on hundreds of thousands of machines, of which about a third currently uses SelfTune's parameter tuning. Our deployment has been running for the last six months. We find that SelfTune has improved cluster throughput by 15%–20% in multiple clusters, while simultaneously improving the resource health in some cases. Based on this, operators are in the process of rolling out SelfTune on the entire fleet of machines.

This work 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 4.4).
(2) We use a novel algorithm, Bluefin, based on rigorously-studied ideas in online learning [70, 136], which allows multiple parameters to be tuned quickly and simultaneously. 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 [140] and state-of-the-art RL algorithms [49] (Sections 4.2). We have integrated and open-sourced Bluefin with a popular machine-learning toolkit [27].
(4) To the best of our knowledge, ours is the first work to describe a developer-centric frame-

work 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 container rightsizing with Kubernetes and DeathStar benchmark [73] (Section 4.2), yielding significant improvements in tail latency as well as throughput.

4.2 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 DeathStarBench microservices benchmark [73]. 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 [146], the workload generator [28] 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 4.1 for the Kubernetes VPA (Vertical Pod Autoscaler) [12], 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 DeathStarBench (as identified in [146]), which impact the application latency.

Compared methods: We compare SelfTune's Bluefin with two standard techniques: (i) Bayesian Optimization — the Gaussian Process (GP) method [47], implemented in [24], and used in [39, 175, 146], (ii) Contextual Bandits [49] RL technique — the ε-greedy algorithm implemented in [27], and used in [35, 34]. For all the experiments, we initialize SelfTune and BO (GP) with the default parameter values as well as random values, and report the best results for each method. 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 algorithm to explore for the first 25 rounds and then exploit for 25 rounds.

4.2.1 Results

Optimizing throughput: We now demonstrate the significance of tuning Kubernetes VPA parameters. We set up a barebones version of DeathStarBench 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 [11]. As the

Metric	Bluefin	BO (GP)	ε-greedy	Default
Throughput %	**86.1 ± 2.2**	83.9 ± 3.1	71.2 ± 4.3	49.8 ± 1.8
# Samples	**12**	14	13	-

Table 4.2 Tuning key parameters of Kubernetes VPA.

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 [25]. 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 might help quickly catch up with the heavy workload, but will lead to severe resource wastage once the peak dies.

A natural question is if we can tune these VPA parameters 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 4.2 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, 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 4.1). All the 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.

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 [19, 26, 18, 22, 14] that influence their performance. For instance, the number of threads of performance-critical microservices (e.g., *compose-post-service* in DeathStarBench) is known to significantly improve latency [149, 146]. We tuned 85 key numerical parameters of the microservices in DeathStarBench with P95 latency as the reward for all the methods.

Metric	Bluefin	BO (GP)	ε-greedy	Default
P95 latency (ms)	**19.5**	19.9	20.0	31.1
# Samples	**8**	41	30	-
P50 iter. cost (ms)	**20.5**	23.3	29.2	-
P75 iter. cost (ms)	**21.1**	33.0	33.2	-
P95 iter. cost (ms)	**28.3**	76541.9	67640.3	-

Table 4.3 Tuning parameters of microservices in DeathStarBench: 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.

Table 4.3 shows the best tail (P95) latency attained by each of the methods and the number of samples they took to achieve the same. 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. 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. In contrast, the other two methods have much worse convergence behavior, as witnessed by P50 iteration cost being far from the best values attained by these methods. We deployed each parameter setting three times, and report the median value.

4.3 Introduction: OPPerTune

The performance and efficiency of large services and applications [1] depend heavily upon how they are configured. Configurations can be *system-level*, such as read_ahead_kb which decides how much extra data to read from disk during I/O in Linux, and resources.limits.cpu that limits the amount of CPU a Kubernetes container uses. They can also be *application-level*, such as maxmemory, the memory usage limit at which Redis starts evicting keys. Any large application invariably includes hundreds, if not thousands, of such configuration parameters at multiple layers and components [117, 155, 85, 114, 182, 58] .

Today, application operators determine the configuration values using domain-knowledge and canary testing on relatively small deployments before widely deploying the application. But operators can hardly be expected to have perfect domain-knowledge and canary testing can only emulate a limited environment. Moreover, application behavior can change considerably with time and therefore the configuration values set before deployment may not work well in the longer term. Developers continuously add features and optimize code, the user population increases or drops, and usage behavior varies [112]. Further, machines

[1] Henceforth, we use the word "application" to mean any cloud-deployed service or application.

that host the application are continuously retired and newer machines with very different profiles are introduced [156, 182]. Consequently, to squeeze the most performance—say throughput or latency—or to make it run efficiently on as small a set of resources as possible without compromising performance, operators need to constantly monitor and modify these configuration parameters well after they have deployed the application.

Manually exploring and changing the configurations at regular time intervals can be extremely tedious and risky too, given that the number of parameters is large and, more often than not, the values of parameters can depend on each other and the deployment environment. Several recent efforts have proposed the use of machine-learning based techniques [66, 183, 159, 104, 114, 147, 58, 37, 97, 106] to automate the process of configuration tuning. These efforts use online learning or reinforcement learning to set the configurations, observe application state to determine how well it is doing, and then iteratively refine the configuration based on the observed states. This approach does reduce the burden on the operator and yet, the problem is far from solved. The algorithm is only one necessary component of post-deployment configuration tuning: to solve the problem, one has to consider the *end-to-end process of configuration tuning* which consists of various other components.

First, since services can easily have thousands of configuration parameters, it would be prohibitively expensive to automatically tune all of them simultaneously. Thus, an automated approach should determine which components or layers of the service to tune, and for each component or layer, which configuration parameters it should tune.

Second, when a service is running, all configuration parameters are not equally easy to tune. For instance, changing worker_process of Nginx (this sets the number of Nginx worker processes) requires only a service reload after changing a configuration file [4] with no downtime, but changing wiredTigerCollectionBlockCompressor in MongoDB requires a pod restart which leads to a downtime of close to 8 seconds when deployed with the recreate strategy on Kubernetes [21]. Changing isolcpus, the parameter that isolates CPUs from the kernel scheduler in Linux, will require an entire system reboot [148]. Previously proposed algorithms do not consider this varying difficulty of tuning different types of parameters. Moreover, to reduce potential disruptions for deployed services, the tuning approach should use a very small number of iterations to converge on the right values.

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

Finally, standard algorithms for tuning may be inapplicable or inefficient. They work only on numerical [70, 97] or only on categorical [44, 93, 153, 49] parameters; real-world services will almost always have a combination of categorical and numerical parameters. For instance, Redis's maxmemory and maxmemory-policy (sets the eviction policy) are related parameters that are of type numeric and categorical, respectively. Moreover, previously proposed techniques like Bayesian Optimization [40, 38] are not meant for reinforcement learning scenarios when the environment of the service can continually change. Deep neural models for configuration tuning [105, 134] can take a very long time to train which is infeasible in deployment. A practical, deployable approach should generalize to all types of configurations, *and* should converge quickly.

We have designed, developed, and deployed OPPerTune (**O**nline **P**ost-deployment **Per**formance **Tune**r), a configuration tuning service that addresses all the above challenges. Application operators can use OPPerTune to create automatic tuning instances, specifying the configuration parameters they wish to tune. The OPPerTune service supports (a) various algorithms in its back-end that the tuning instances could use, and (b) ways to automatically create, manage, and scope the instances needed to tune the performance of large-scale complex services. The contributions we make are:
1. OPPerTune introduces a novel tuning algorithm which can tune categorical and numerical parameters simultaneously, within the same instance.
2. OPPerTune uses a novel decision-tree based algorithm to automatically determine the right scope of tuning instances.
3. We have built OPPerTune as a cloud service which application developers can invoke for tuning enterprise-scale applications with large number of categorical and numerical parameters.

We have evaluated OPPerTune using two applications: the social networking application from the DeathStarBench benchmark suite and a popular data processing platform used within an enterprise for ML model prototyping and development. Our results show that by using OPPerTune to tune configuration parameters, (i) the tail latency of the application consisting of tens of microservices reduces by more than 50% while tuning in deployment, compared to carefully-chosen pre-deployment configurations; (ii) the workload completion times drop by 10%-50% on two enterprise-scale data processing production clusters. We also show that OPPerTune is effective in reducing service disruptions that may occur due to configuration tuning by nearly 30%.

4.4 OPPerTune Overview

In this section, we give a brief overview of OPPerTune. We will use the term 'application' to refer to the system/service/application being tuned to avoid confusion with the OPPerTune service itself.

Configuration tuning problem: The goal of OPPerTune is to continuously tune the specified configuration parameters of an application such that, over time, a given *reward* metric (e.g., daily P95 latency, or how far off the hourly resource utilization is from the desired bounds) is maximized, and the application sustains good performance through long-term and short-term hardware changes and workload fluctuations. OPPerTune works under the least knowledge of the application being tuned, i.e., black-box access. In particular, it does not have access to the code-base of the application or any knowledge of how its performance metrics are computed. OPPerTune relies only on the reward as feedback from the application (after a certain amount of time) for a set of configuration values it sets for the application. It uses this feedback to tune the configurations iteratively.

Consider the following example. A web application uses two containers on a single machine: one to host a front-end webserver, and the other to run a back-end database. While serving user requests, the application can enlist OPPerTune to learn how to distribute the machine's memory and compute between the webserver and the database so as to minimize P95 request latency. OPPerTune consumes feedback (or reward) from the application in the form of observed hourly P95 latency, and uses multiple hours' feedback to converge on the right memory and compute distribution between the two containers. Request characteristics can vary with time; thus, OPPerTune may need to change the distribution of memory and compute frequently, *and* converge quickly to stable configuration values while continuing to minimize P95 latency.

OPPerTune **architecture:** Figure 4.1 presents the high-level architecture of the OPPerTune service. The basic unit of the service is a *tuning instance* that consists of (a) configuration parameters, their data types, enumerations of possible values for categorical configurations or ranges for numerical configurations; and (b) a tuning algorithm for updating the instance. Applications can create one or more tuning instances to tune configuration parameters across various layers of the application stack, based on its requirements (as shown for 'App 2' in the figure). Alternatively, OPPerTune provides an automatic scoping component (*autoscoper*) to help applications create, manage, and scope the tuning instances in deployment based on dynamic context information they provide (as shown for 'App 1'). OPPerTune also aids applications pre-determine (using an offline step) which configuration parameters to tune in deployment via the *selector* module.

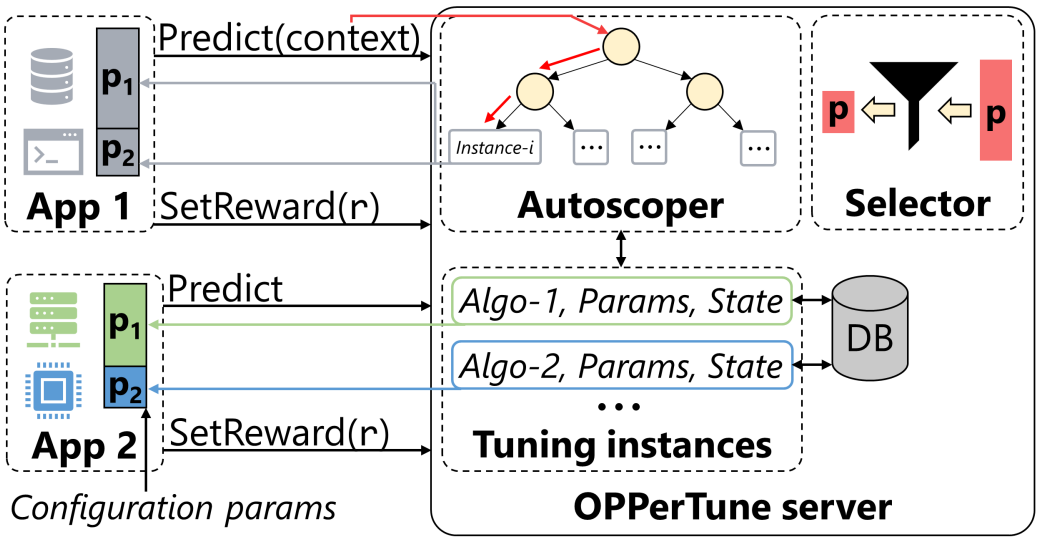


Fig. 4.1 OPPerTune service architecture: Applications create *tuning instances* on the server to tune various configuration parameters. *Autoscoper* helps automatically create, manage, and scope tuning instances based on the application's dynamic context. *Selector* helps pick the most promising configuration parameters to tune.

Creating, fetching, and updating tuning instances: An application wishing to use OPPerTune makes an API call to create a *tuning instance*. OPPerTune persists tuning instances on a database for the application to access at any point in time, possibly from multiple machines. For each configuration parameter it wants to tune, the application can optionally supply the cost associated with changing it. Of particular interest in our work is whether, to update a parameter value, a system (container or VM) needs to restart. Such parameters should have a high cost associated with them. When the application provides such costs for some parameters, OPPerTune uses them to decide how often to tune the configuration parameters. OPPerTune implements the standard fetch and update client-API paradigm of existing work [35, 97] for online tuning. The application invokes (a) Predict to fetch the recommended configuration values from a tuning instance, and (b) SetReward, at some point in time after (one or more) Predict calls, with a reward value. The SetReward call updates the associated tuning instance, as per the tuning algorithm it uses.

Autoscoper: Applications may have different performance characteristics on machines with different CPUs or memory sizes, and hence may consider using a different configuration tuning instance for each machine type. Similarly, applications could behave differently for light versus heavy workloads, and for different API call types. For instance, if the application runs independently on the cluster machines with varying hardware and workloads, then it could create one tuning instance per machine (as is done in [97] for tuning configuration parameters of a workload scheduler). Thus, tuning instances for the application could be

"scoped" along (at least) three dimensions: *infrastructure* (e.g., machine type), *functionality* (e.g., API call), and *workload* (e.g., requests per second). Currently, determining the "right" scope for tuning instances is usually done, if at all, by domain experts [97], and periodically revisited. As an alternative, in Section 4.6, we introduce an automatic, efficient, and interpretable way of scoping tuning instances, that we call AutoScope.

Configuration selector: For applications that have several hundreds or thousands of configuration parameters to tune across various layers of the application stack, OPPerTune employs a selector module to pick the most *promising* configuration parameters to tune. The selector module uses a simple and effective *microbenchmarking* strategy to identify such configurations, as discussed in Section 4.7.

Rounds and Sample complexity: Online tuning algorithms, implemented in OPPerTune's back-end, iteratively tune the configuration parameters. Each iteration is called a *round*. At each round, the tuning algorithm (i) determines the next set of parameter values for the application, (ii) observes a reward computed by the application over a predetermined period (1 hour, 24 hours, etc.), and (iii) updates its "policy" (which prescribes how to choose parameters) based on the reward. Changes made by the algorithm to configuration values can cause disruptions, e.g., may necessitate application restarts or even cause downtime. Thus, a desired property of a tuning service is that it requires only a few rounds to learn suitable configuration values. This quantity, proportional to the number of rewards measured, is called *sample complexity*. OPPerTune achieves low sample complexity for tuning in real deployments, as we demonstrate in Section 4.9, using multiple techniques including a novel tuning algorithm (Section 4.5), automatic scoping, and microbenchmarking.

4.5 Configuration Tuning in Hybrid Spaces

In this section, we give a novel algorithm for the post-deployment configuration tuning problem stated in Section 4.4 in the basic setting when OPPerTune has no additional knowledge ("context") about the system being tuned. We begin by setting up some notation and terminology.

4.5.1 Problem Definition and Terminology

We pose the problem of tuning configuration parameters of an application after it is deployed as that of online learning with bandit feedback. That is, we want to tune iteratively, and the only interface we have with the application is setting new parameter values (e.g., number of CPU cores and memory size for the container), and the only feedback we get from the application, in response to the set parameter values, is an observed *reward* value that is to be maximized (e.g., latency or throughput of the application).

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

Bandit learning techniques that can handle time-varying rewards, therefore, are more appropriate to our post-deployment parameter tuning problem than popular alternatives such Bayesian Optimization (BO) [40, 38], heuristic search and global optimization [123], and genetic algorithms [76]. For instance, BO needs to evaluate the *same* reward function at multiple parameter values by design. This is infeasible for post-deployment tuning because we cannot evaluate the application multiple times when it has exactly the same workload. A lot of systems research that leverages BO typically uses it in offline scenarios (i.e., pre-deployment tuning in controlled settings), in contrast to our post-deployment tuning scenario.

4.5.2 Hybrid Configuration Space

In practical scenarios, the space of configuration parameters can be complex: (i) it can be very large; if there are m parameters to tune, and if each parameter has, say, s possible values, then the total number of choices is s^m, and (ii) the parameters to tune may be discrete (e.g., number of CPU cores), real-valued (e.g., CPU utilization threshold), or categorical (e.g., cache eviction policy). Some state-of-the-art techniques for bandit learning/RL work for categorical spaces [44, 93, 153, 49] or numerical spaces [70, 97], but not both. Others have prohibitive sample complexity to be useful for tuning in deployment [105, 134]. To address this gap, we design a novel learning algorithm that can handle hybrid configurations efficiently.

Formally, in the "hybrid configuration space" setting, we are given (a) categorical space $\mathcal{C} := \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_k$ over k categorical parameters, where each $\mathcal{C}_i$ denotes the possible choices for the categorical parameter i, and (b) numerical space $\mathcal{W} = \mathcal{W}_1 \times \mathcal{W}_2 \times \cdots \times \mathcal{W}_m$ over m numerical parameters, where each $\mathcal{W}_i$ indicates a subset of the real line $\mathbb{R}$, e.g., specified by lower and upper bounds for the i^{th} parameter. Note that this characterization also allows discrete parameters p such as number of CPU cores ranging from 2 through 16, in steps of

2, by letting $\mathcal{W}_p = \{2,4,\ldots,16\}$. In our formulation, we treat such discrete parameters as numerical rather than categorical to exploit the fact that they are ordered spaces

4.5.3 Proposed Algorithm: HybridBandits

Our configuration tuning HybridBandits algorithm is presented in Algorithm 2. It leverages two simple but key ideas. At each round,
(1) it maintains different *types* of policies for sampling categorical and numerical actions; in particular: (1) ε-greedy policy for the categorical configuration space, standard in multi-arm bandit algorithms, where with probability ε a random arm is explored, and with probability $1-\varepsilon$, high-reward arms are exploited; and (2) a "perturbation" policy for numerical configurations, where the algorithm samples numerical configurations from an ε-radius ball centered around the "current best" configuration vector, and
(2) it uses a single reward that the system provides as feedback to update both the policies simultaneously. In particular, it applies sample-efficient gradient-descent update [70, 97] for the numerical parameters, and the exponential weights update [101, Chap. 11] for the categorical parameters.

Algorithm 2. HybridBandits*: Post-Deployment Configuration Tuning for Hybrid Spaces*

1: ***Input:*** *exploration parameter* $\varepsilon \in (0,1)$*, learning rate* $\eta > 0$*, categorical parameter space* $\mathcal{C} := \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_k$*, numerical parameter space* $\mathcal{W} = \mathcal{W}_1 \times \mathcal{W}_2 \times \cdots \times \mathcal{W}_m$
2: ***Initialize****: categorical space weights* $p_i^{(0)} = 1/|\mathcal{C}|$*, for* $1 \le i \le |\mathcal{C}|$ *// uniform distribution, and numerical parameters* $w_i^{(0)} \in \mathcal{W}_i$*, for* $1 \le i \le m$ *// default choices*
3: ***for*** $t = 0,1,2,\ldots$ ***do***
4: *Let* $\widetilde{p}_i := (1-\varepsilon)p_i^{(t)} + \varepsilon\frac{1}{|\mathcal{C}|}$ *// Define explore-exploit multinomial distribution over the categorical space*
(1) *Sample categorical and numerical actions to deploy*
5: *Sample* $c \sim \widetilde{\mathbf{p}}$ *from the multinomial and let* $\mathbf{a}_c^{(t)}$ *be the corresponding k-tuple of categorical parameters*
6: *Sample numerical parameters from a ball centered at* $\mathbf{w}^{(t)}$*, radius* ε*; i.e.,* $\widetilde{\mathbf{w}}^{(t)} := \mathbf{w}^{(t)} + \varepsilon\mathbf{u}$*, where* $\mathbf{u} \in \mathbb{R}^m$ *is sampled from* $\{\mathbf{u} : \|\mathbf{u}\|_2 = 1\}$ *// Identical to* Bluefin *[97]*
(2) *Deploy the actions and measure reward*
7: *Deploy numerical* $\mathbf{a}_r^{(t)} := \Pi_{\mathcal{W}}(\widetilde{\mathbf{w}}^{(t)})$ *// appropriately scaled and categorical actions* $\mathbf{a}_c^{(t)}$ *in the application*
8: *Receive reward* $r^{(t)} := r_t\left(\mathbf{a}_c^{(t)}, \mathbf{a}_r^{(t)}\right) \in \mathbb{R}$ *// black-box access to a metric, e.g., hourly P95 latency, computed by the application*
(3) *Perform updates based on the reward received*
9: *Update numerical parameters center:* $\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} + \frac{1}{\varepsilon} \cdot \eta \cdot r^{(t)} \cdot \mathbf{u}$*, where* $\mathbf{u}$ *is the sample obtained in Step 6.*

10: Define scaled reward: $\widetilde{r}^{(t)} = r^{(t)}/\widetilde{p}_c$*, where c is the sample obtained in Step 5*

11: Update categorical distribution: $p_c^{(t+1)} \leftarrow p_c^{(t)} e^{\eta \widetilde{r}^{(t)}}$*, and for* $i \neq c$, $p_i^{(t+1)} \leftarrow p_i^{(t)}$*; Renormalize* $\mathbf{p}^{(t+1)}$ *to sum to 1*

end

Algorithm description: Algorithm 2 maintains a multinomial distribution $\mathbf{p}^{(t)}$ over categorical actions $\mathcal{C}$, i.e., there is a probability associated with each possible k-tuple of categorical parameter choices at every round t. For the numerical actions, it maintains a vector $\mathbf{w}^{(t)} \in \mathbb{R}^m$.

Initialization: The weights for the numerical parameters $\mathbf{w}$ are initialized to default choices that the application provides. The multinomial $\mathbf{p}$ is initialized to the uniform distribution, i.e., $p_i = \frac{1}{|\mathcal{C}|}$ for $i \in \mathcal{C}$. At each round, the algorithm performs:
(1) Sampling actions (Steps 5-6): For the categorical actions, following the standard exponential weights algorithm (EXP3, [101, Chap. 11]), it samples a k-tuple from the distribution $\mathbf{p}$ (*exploit*) with probability $1 - \varepsilon$, and from the uniform distribution (*explore*) with probability ε. For the numerical actions, it samples a m-dimensional vector from a ball centered at the current $\mathbf{w}$, with radius ε.
(2) Deploy actions and receive reward (Steps 7-8): The sampled numerical (scaled appropriately) and categorical configurations are then deployed in the application, and (after a certain amount of time) the algorithm receives a reward value from the application (implementation details in Section 4.9).
(3) Update policies (Steps 9-11): For the numerical parameter weights, the algorithm follows the gradient estimation scheme studied in the optimization literature [70], as well as applied in the context of online system parameter tuning [97]. For the categorical parameters, it: (a) computes an unbiased estimate of the reward for the sampled choices, and (b) scales the probability of the sampled choices using a factor that is exponential in the reward estimate.

In practice, $\mathcal{C}$ can be very large; the microbenchmarking strategy discussed in Section 4.7 can be used to restrict $\mathcal{C}$ to the most impactful categorical parameters, and to ensure that the algorithm has low sample complexity. We conjecture that Algorithm HybridBandits has convergence guarantees for certain classes of reward functions (for instance, if the reward functions r_t are convex, for any fixed combination of the categorical parameters in $\mathcal{C}$). Empirically (in Section 4.9), we find that the algorithm convergences well in practice; obtaining a formal proof of convergence is an exciting open problem.

4.6 Automatic Scoping of Tuning Instances

Consider an operator who wants to tune parameters of a distributed application that is I/O-bound. There are two extreme options available to the operator in terms of how they

can set up tuning instances on the OPPerTune server (Figure 4.1): (1) set up one "global" instance to tune all the application parameters across all machines/workloads, that, say, uses HybridBandits presented in Section 4.5 for tuning; or (2) set up multiple "local" instances based on the domain expertise that the workloads are I/O-bound; e.g., one instance per disk type or one instance per spindle speed, where each instance independently tunes parameters using HybridBandits. The latter option is more appealing as the application performance, and therefore the optimal parameter choices, are likely to vary with the disk type that the workloads are accessing.

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

4.6.1 Joint Scoping and Configuration Tuning

To perform joint scoping and configuration tuning, at each round, along with the reward, the application must provide additional context information such as machine type, disk type, spindle speed, workload volume, etc. Using this additional context, OPPerTune determines a light-weight and interpretable scoping policy that the operators can understand. For instance, given job type `jobtype` and requests per second (`rps`) as context, and `numcores` and `mem` as the configuration values to tune, it learns rules of the form `if (jobtype == 'cpu_bound') and (rps > 1000) then numcores=16, mem=2G else numcores=4, mem=2G`. These kinds of scoping rules can be captured by decision tree models. Each root-to-leaf path in the tree constitutes a scope, and each leaf maintains a tuning instance for the scope.

4.6.2 Proposed Algorithm: AutoScope

Learning decision trees in the bandit setting is a challenging problem, and popular tree learning algorithms do not apply (See Section 4.10). We extend a state-of-the-art technique proposed in [96] (for trees with only one parameter in leaf nodes) to our general setting where each leaf node is a tuning instance with several (hybrid) parameters. We give the key intuitions below.

AutoScope maintains a binary decision tree f_T of max specified height h ($h = 3$ suffices for scenarios evaluated in Section 4.9). At first, the tree $f_T^{(0)}$ effectively behaves like a single tuning instance, initialized identical to Algorithm 2. At round t, the algorithm observes a context vector, denoted as $\mathbf{c}_t$. When the current tree model $f_T^{(t)}(\mathbf{c}_t)$ is applied to $\mathbf{c}_t$, it will land in a unique leaf node containing a tuning instance. The root-to-leaf path $\mathbf{c}_t$ traverses is its "current scope", and AutoScope will invoke the leaf's tuning instance. This amounts to doing one round of Algorithm 2 on the instance, thereupon updating the instance. Now the

technical challenge is updating the tree model $f_T^{(t)}$ parameters, i.e., the weights in the internal nodes of the tree which make the branching (scoping) decisions. For this, we leverage the trick from [96], that allows us to update all the internal nodes along the path traversed by $\mathbf{c}_t$ *based on the same reward that was used to update the leaf instance*. At the end of round t, all the nodes in the scope of $\mathbf{c}_t$ will be updated using a *single* reward value.

4.7 Configuration Selection

For applications that have several hundreds or thousands of configuration parameters to tune across various layers of the application stack, OPPerTune employs a configuration selector module (Figure 4.1) to first pick the most *promising* configuration parameters. This module uses a simple and effective *microbenchmarking* strategy to identify such configurations; while the techniques we outline here are heuristic, they are inspired by optimization theory [122, 124].

The role of the selector module is two-fold: (a) it prunes the size of the configuration space for the tuning algorithm, which in turn helps reduce the algorithm's sample complexity; and (b) it helps minimize the number of disruptions (e.g., container restarts) in the application while tuning. If (b) is the only goal, then an obvious strategy is to select only those configuration parameters that do *not* require restarts. However, such a strategy might compromise on application performance by ignoring configurations that could significantly impact the performance. In Section 4.9.3, we show this is indeed the case.

OPPerTune uses a *microbenchmarking* strategy to assess the effect of changing *each* configuration parameter on the application's performance (i.e., the reward value), while keeping the others fixed. Let us consider numerical configurations for the moment. The strategy is inspired by how co-ordinate descent algorithms [122], that are rigorously-studied in the optimization community, work. These algorithms pick one coordinate (i.e., configuration parameter) at a time and compute the gradient of the reward function with respect to only that parameter. They iteratively pick coordinates (cyclically or randomly) to optimize the reward function.

We do not know of any variants of these algorithms that provably work in our online bandit setting. But, we find that the basic idea is empirically effective for the goal of selecting candidate parameters to tune. We use the same gradient estimation technique employed in Step 9 of Algorithm 2 for each configuration parameter, while holding all other parameters fixed (to the default choices, for example). OPPerTune accomplishes this by simply creating microbenchmarking instances, each with just one configuration parameter, and performing one round of the HybridBandits algorithm. The *magnitude* of the (scalar) gradients computed at the instances tells us the impact of each configuration parameter. In practice, this idea

can also be extended to categorical spaces—perform one round of the algorithm on each categorical parameter, and compute the magnitude of change in reward for a randomly chosen value vs. the default value for the parameter.

The next question is how to use these "gradients" to select the most promising parameters. The configuration selector module picks top-n parameters, sorted by decreasing magnitudes of gradients where n is customizable by the application. This greedy selection strategy, i.e., picking the coordinate (or parameter) yielding maximum absolute gradient, has been shown to be provably better than other heuristics for selecting coordinates, for some classes of reward functions [124].

Microbenchmarking can be done in canary/test rings of the application. The tuning instances for the application can work with the selected configuration parameters in deployment.

4.8 OPPerTune Implementation

OPPerTune's implementation has three major components: the server, client, and the algorithm backend. We have implemented the server in Go using Fiber [30], and the client in Python (for ease of integration with applications which are often written in Python). We have implemented our proposed algorithms in Python, and have integrated the server with existing Python implementations of other algorithms. We now describe each component in some detail.

1. OPPerTune **Server:** The server implements three key interfaces for the applications (clients) submitting requests via REST API calls: a) creation of tuning instances, b) fetching the values from the instances, and c) updating the instances using the the reward values sent back to the server. The server persists configuration tuning instances (consisting of the list of parameters to tune and their constraints, and the model for tuning) in a database. Persisting instances enables resuming from the saved model state at a later point of time, and freeing the memory taken up by instances that are not needed. For each fetch call from the client, the server responds with the configuration values along with a `requestId`. The client is expected to pass the reward value along with the associated `requestId`, for the server to be able to correctly issue an update to the corresponding tuning instance. We host the server on a large enterprise-scale cloud platform that provides persistent storage, high availability, and wide accessibility.

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

error-checking. We provide an installable package of OPPerTune client for applications to use.

3. Service Backend/Tuning Algorithms: The backend consists of implementations of various tuning algorithms, and *autoscoper* and *configuration selector* components. Any tuning algorithm is expected to implement Predict and SetReward interfaces. We have implemented (i) the proposed algorithms HybridBandits and AutoScope; (ii) state-of-the-art online parameter tuning algorithm Bluefin [70, 97] and deep reinforcement learning (RL) algorithm DDPG [105, 129, 134]; and (iii) with minimal effort, we have integrated the contextual bandits-based algorithm Slates [153], and BayesianOptimization from the popular Python libraries scikit-optimize [32] and VowpalWabbit [27] respectively. The challenge in using deep RL techniques such as DDPG in deployment, typically, is their prohibitive model sizes and sample complexity. They use context differently than AutoScope to learn policies that require large complex models (the notion of scoping is implicit in DDPG). Hence, we have implemented a custom version of DDPG with light-weight models, similar to AutoScope, to be of use in post-deployment tuning.

4.9 Evaluation

To evaluate OPPerTune, we use a combination of microbenchmarking and real deployment of a benchmarking application. Our evaluation focuses on the following aspects: 1) *How does application performance improve using* OPPerTune?, 2) *How does* OPPerTune *reduce the cost of tuning (e.g., system restarts)*? and 3) *How effectively does automatic scoping accelerate the tuning process in real deployments by reducing sample complexity*?

4.9.1 Evaluated Application

To evaluate OPPerTune, we use the social networking application from DeathStarBench [73] with synthetic and production traces.

Social Networking Application

We use the SocialNetwork application from the DeathStarBench [73] benchmarking suite which mimics a stack consisting of a gateway server (Nginx), database engine (MongoDB), caches (Redis), and application logic. The application creates a network of users, and supports API calls to create and read messages from the users' home pages. We use wrk2 [28] to emulate two workloads: (a) **constRPS**: a mix of 90% GET (read timeline) and 10% POST (create posts) requests (this mix has been used in previous work [97]), with the requests generated at a constant rate (requests-per-second), and (b) **AMStraces**: real access traces collected from AMS, a large-scale enterprise image-sharing service (real name withheld). We

Layer	Microservice type	Number of parameters		
		Categorical	Continuous	Discrete
Microservices	memcached	0	4	16
	MongoDB	12	6	12
	Nginx	0	0	8
	Redis	4	0	16
	App logic	0	0	23
Right-sizing (Kubernetes)	memcached	0	8	8
	MongoDB	0	12	12
	Nginx	0	4	4
	RabbitMQ	0	2	2
	Redis	0	8	8
	App logic	0	24	24

Table 4.4 SocialNetwork application configuration parameters (217 in total) used for Figures 4.2, 4.3, and Tables 4.5, 4.6.

use traces collected from 3 production clusters over 4 weeks from Sep 14, 2022 to Oct 10, 2022.

For this application, the performance metric of interest is **P95 latency of requests** submitted. This metric is often critical to consumer-facing services [73, 61]. For the constRPS workload, we measure the P95 latency for each 10-minute period, and for AMStraces, we measure it for each hour. This is fed as the reward value to the OPPerTune service.

Table 4.4 outlines the list of microservices that SocialNetwork uses. Here, "right-sizing" layer refers to configuration parameters in Kubernetes that are used to determine the compute and memory limits for containers running the microservices. For each of the microservices, we tune *a mix of real-valued, discrete, and categorical parameters* that prior work has identified as important to the performance of that microservice [97, 147, 2, 16, 14, 22, 26, 33]. The optimal values of these parameters depend on the workload characteristics; e.g., to support the same P95 latency, the MongoDB microservice will require higher resource limits for a higher request volume, and a larger number of clients would require higher concurrency setting for Nginx, etc.

We deploy the SocialNetwork application on a cluster with 7 virtual machines (VMs) provisioned on a large public cloud provider. Each VM hosts a copy of the application stack, with the component microservices running on individual containers on the same VM, managed by Kubernetes. Thus, by tuning the parameters in Table 4.4 appropriately, OPPerTune allocates each VM's resources in the right proportions to the various microservices so as to minimize P95 latency.

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

4.9.2 Improving Application Performance

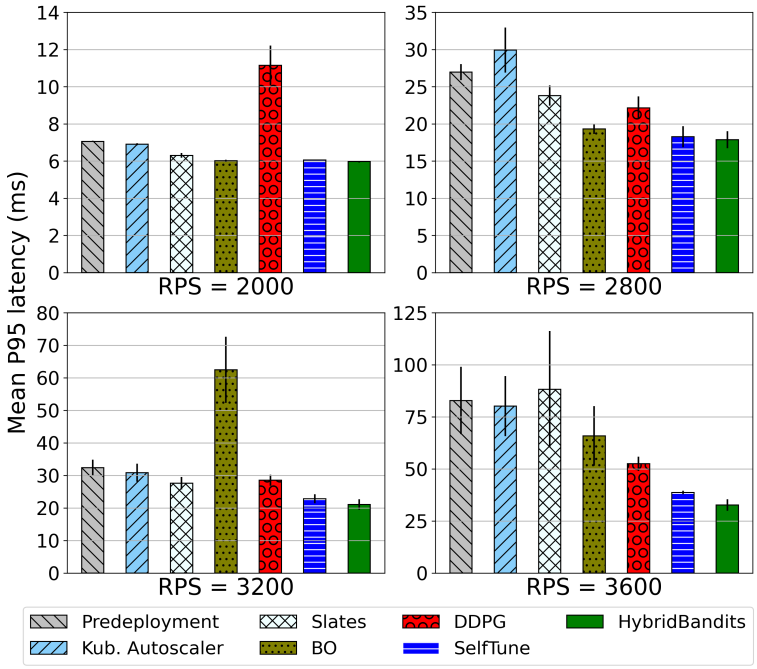


Fig. 4.2 Comparison of various techniques for post-deployment configuration tuning of the SocialNetwork application using constRPS workloads. 'Predeployment' is the baseline performance achieved with configuration choices we manually chose based on 3500-rps workload.

A) Effectiveness of HybridBandits**:** We first look at how effective the proposed algorithm HybridBandits is for performance tuning of applications in deployment, and how it fares relative to various state-of-the-art tuning techniques that we implement as part of OPPerTune backend (as discussed in Section 4.8). For this evaluation, we use the SocialNetwork application, as it has a mix of real-valued, discrete, and categorical parameters (Table 4.4). For each rps (constRPS workload), we run each tuning algorithm for a maximum of 50 rounds. The results are presented in Figure 4.2. We report the mean and standard deviation of P95 latencies, for the converged configuration values, over 5 trials (10-minute windows each).

"Predeployment" in Figure 4.2 refers to the baseline performance achieved with configurations we manually chose that optimized the performance for a 3500 rps workload (which is close to the peak capacity supported by our cluster), and keeping them fixed for the rest of the rps. "Kubernetes Autoscaler" refers to the Vertical Pod Autoscaler (VPA) [13] for determining the rightsizing parameters. VPA performs poorly in general, as it makes rightsizing

decisions solely based on container utilization, and not on P95 latency of the application. Our experiments also compared several existing approaches, i.e. Bayesian Optimization (BO), state of the art RL techniques Slates and DDPG , and Bluefin with HybridBandits.

First, the benefit of using OPPerTune service post-deployment is clear: every algorithm, almost for every rps, finds better configuration choices to adapt to changing workload volumes. Second, almost for every rps, HybridBandits significantly outperforms BO, Slates, and DDPG, and achieves the best P95 latency. For instance, at the peak workload of 3600 rps, HybridBandits achieves nearly 2x reduction in P95 latency compared to the best configuration predicted by BO that has a large variance. Third, the utility of tuning categorical parameters together with the numerical parameters, using HybridBandits, is clear at high workloads, compared to Bluefin algorithm in the SelfTune framework[97] that supports only numerical parameters. In particular, HybridBandits (32.6ms) achieves about 15% reduction in P95 latency relative to Bluefin (38.6ms) for the 3600-rps workload. DDPG performs reasonably well in high workloads, despite the absence of informative context.

Method	(P50, P95, max) of P95 latency of each hour over the 4th week (ms)									Sample Complexity (#rewards for tuning)
	Cluster 1			Cluster 2			Cluster 3			
	P50	P95	max	P50	P95	max	P50	P95	max	
Pre-deployment choices	12.7	19.4	23.8	12.4	18.2	1959.1	12.3	44.1	4018.4	-
$\text{HybridBandits}_{\text{cluster, hour}}$	10.6	17.3	19.3	9.5	19.6	21.0	10.9	17.4	36.7	756
AutoScope	10.1	15.8	23.7	8.5	13.5	17.1	7.9	15.7	33.4	252
DDPG	10.4	17.0	23.2	8.3	14.2	18.4	9.2	18.1	32.6	756

Table 4.5 Comparison of various techniques for post-deployment configuration tuning of the SocialNetwork application using real workloads (AMStraces). Algorithms in the last three rows, implemented in OPPerTune, use the first 3 week-traces for tuning. The first row is the baseline performance achieved with manually-chosen configuration choices.

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

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

1. SocialNetwork + **AMStraces:** We compare AutoScope with a domain-expertise based scoping strategy, informed by the diurnal patterns of workloads in AMStraces. We use one tuning instance for every 2 consecutive hours in a cluster-day. Each tuning instance runs HybridBandits independently to learn suitable configuration parameters for its 2-hour scope. We refer to this as $\text{HybridBandits}_{\text{cluster,hour}}$. For AutoScope, we use average rps over every 2 consecutive hours in a day as context. We build a simple estimator for rps values using the

first week traces, and use them for all the weeks. This is because we can not know the true rps values (in the future) at the time of Predict calls.

We let all the methods use the first 3 weeks' traces to tune the configuration parameters for SocialNetwork in deployment. We then evaluate the converged parameters on the last week's trace. In Table 4.5, we present, for each technique and for each cluster, (a) P50, P95, and maximum value of hourly P95 latency, computed over the last week, i.e., over 168 hours, and (b) sample complexity of the technique (i.e., # rewards used while tuning). We compare AutoScope with (i) the baseline of using "Pre-deployment choices" of configuration parameters, (ii) the domain-expertise based scoping $\text{HybridBandits}_{\text{cluster,hour}}$, and (iii) the deep-RL based DDPG that uses rps, and CPU and memory utilization of microservices and VMs as features ("states") for implicit scoping. We see that the max P95 latencies for Clusters 2 and 3 are in the order of seconds with the Pre-deployment choices. On the other hand, OPPerTune, using each of the three algorithms, significantly reduces the worst-case P95 latencies. Importantly, AutoScope achieves significantly better performance in general compared to $\text{HybridBandits}_{\text{cluster,hour}}$, and DDPG in Clusters 2 and 3 especially.

What is particularly noteworthy about AutoScope is that it achieves this performance using one-third of samples (i.e., # rewards) compared to other techniques. All the 3 clusters have very similar diurnal patterns; Clusters 2 and 3 have similar workload volumes. AutoScope is able to exploit these using the context provided, using as few as 8 tuning instances (a height-3 tree), compared to manual scoping that uses 36 (=3 clusters $\times$ 12 time-windows in a day) tuning instances.

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

4.9.3 Mitigating the cost of tuning in deployment

So far, we have focused on the impact of tuning on the performance of the application. We now turn to the cost of tuning in deployment—every change to configuration parameters in production introduces potential risk. In particular, as we discussed in Section 4.4, tuning certain configuration parameters necessitate microservice/pod restarts, causing downtimes. Improving latency of the application at the expense of throughput, or service reliability, may not be acceptable.

We evaluate OPPerTune, in terms of how it trades off improving performance and mitigating restarts while tuning, on the SocialNetwork application and constRPS workloads. The results are summarized in Table 4.6 and in Figure 4.3. We consider various strategies for picking configuration parameters, followed by tuning the selected parameters with HybridBandits, to mitigate the number of pod restarts. The first row of the table ("Microservices $\cup$ Rightsiz-

Parameters/Layers Tuned via OPPerTune-HybridBandits	P95 Latency (ms)			
	RPS = 2000	RPS = 2800	RPS = 3200	RPS = 3600
Microservices ∪ Rightsizing	5.973 ± 0.046	17.864 ± 1.150	21.068 ± 1.641	32.656 ± 2.798
NoRestarts	6.916 ± 0.076	24.545 ± 1.184	31.281 ± 1.958	70.542 ± 22.485
Microservices	6.405 ± 0.039	24.209 ± 1.594	26.764 ± 2.184	38.853 ± 2.500
Rightsizing	6.828 ± 0.022	25.373 ± 0.607	26.774 ± 2.168	70.175 ± 8.267
Microbenchmark-Top 25	6.820 ± 0.060	23.697 ± 1.802	27.022 ± 1.278	34.503 ± 2.642
Microbenchmark-Top 50	6.094 ± 0.075	19.248 ± 1.362	21.888 ± 0.853	37.821 ± 1.489

Table 4.6 Comparison of various ways of selecting parameters to tune in the SocialNetwork application stack using constRPS workloads. Microbenchmarking strategy (last row) yields performance nearly as good as tuning all the parameters (first row).

ing") corresponds to tuning all the parameters listed in Table 4.4. The second row of the table ("NoRestarts") corresponds to tuning only the parameters that do *not* require any restarts. As expected, they achieve the best and the worst P95 latency values, respectively.

In Section 4.7, we introduced the microbenchmarking strategy in OPPerTune for picking the most promising configurations ahead of tuning in deployment. The last row of Table 4.6 shows the performance achieved using HybridBandits on the top-50 parameters: (i) in 3 out of 4 workload rates, we see that the strategy achieves statistically similar performance as the best ("Microservices ∪ Rightsizing"); (ii) with the reduced configuration space, HybridBandits converges within 30 rounds, compared to the 50 rounds needed by the best method (not indicated in the table); and (iii) HybridBandits is superior to tuning only the microservices layer parameters (third row) or rightsizing layer parameters (fourth row). We have also included the performance using top-25 in the fifth row for comparison.

Figure 4.3 shows the number of pod restarts per round and the corresponding average P95 latency for each of the five strategies shown in Table 4.6. We see that our microbenchmarking strategy achieves a good trade-off between cost and performance, using 2800-rps workload

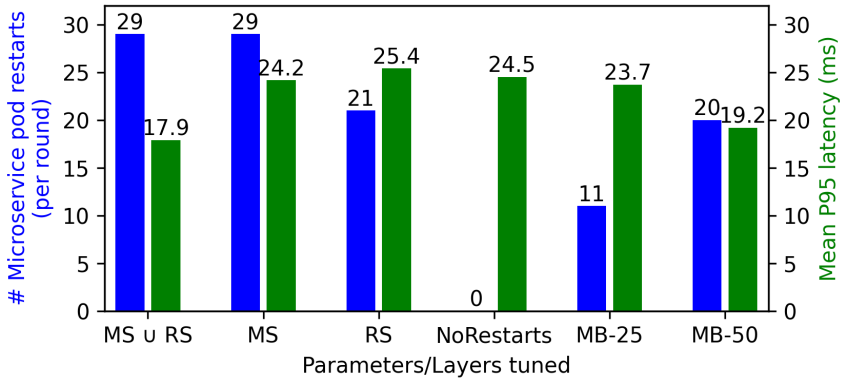


Fig. 4.3 Number of microservice pod restarts (per round) and mean P95 latency (RPS=2800) while tuning different parameters/layers of SocialNetwork app using HybridBandits.

as example (though the findings are consistent across all rps). The best average P95 latency (17.9ms) of "Microservices ∪ Rightsizing" or "MS ∪ RS" comes at the expense of 29 pod restarts *per round* as seen from Figure 4.3. The microbenchmarking strategy ("MB-50" in 4.3) nearly matches the best method's P95 latency (19.2ms), with nearly 30% fewer pod restarts *per round* (20 restarts).

Takeaway 3. HybridBandits + *microbenchmarking strategy of* OPPerTune *reduces the cost of tuning in terms of service disruptions in deployment significantly, while achieving competitive performance.*

4.10 Related Work

Performance optimization of systems through configuration tuning is a long-studied problem [91, 141, 138] that continues to garner interest from the systems research community. Here, we discuss closely related work.

4.10.1 Configuration tuning

Prior works on configuration tuning mainly focus on parameters of specific subsystems of applications [55, 38, 161, 40, 159, 106, 172, 127] such as db and storage or of the infrastructure on which the application is hosted [129, 135, 50, 51, 148]. In such works, the configuration search space is relatively small and the advantages of jointly tuning parameters across the software stack are not considered. Moreover, the approaches are tailored to the specific subsystem being tuned, sometimes requiring domain expertise [55, 38]. Recently, tuning parameters across the software stack [114, 58] and jointly tuning parameters of multiple components of an application [149, 147] are gaining attention. Such works either ignore the cost of reconfiguration [147] or require an expensive offline mode training to perform post-deployment tuning[58, 149, 114].

4.10.2 RL/Bandit algorithms

While there are several RL-based configuration tuning approaches [134, 129, 176], they are either limited in the type of parameters being tuned or are inefficient for online post-deployment tuning scenarios. Some approaches do handle hybrid parameter spaces [102, 115] via cascaded optimization which are quite effective if trained offline. Further, RL algorithms require accurate state information (e.g, utilization metrics), which is additional engineering overhead in enterprise scenarios. Our proposed HybridBandits, on the other hand, can tune all types of parameters in deployment without the need for state information.

Recent work on learning trees using bandit feedback [69, 67] are meant for categorical spaces. Popular tree learning algorithms like CART [53] and C4.5 [130] do not apply to

the bandit feedback setting because they need access to labeled training data (which in our scenario means optimal parameters for different context vectors, which we do *not* have).

4.10.3 Tuning frameworks

Recent works have addressed the need for a generic configuration tuning framework for production systems [182, 156, 97, 135, 78]. KEA [182] is Microsoft's internal tuning framework for cluster-wide configurations. It uses historical data to make decisions on parameters in the pre-deployment stages, unlike our work that focuses on post-deployment tuning. Google's Vizier [78] is their internal service for hyper-parameter tuning of ML workloads, in the offline setting. SelfTune [97] is a recent framework from Microsoft for tuning cluster managers, with support for post-deployment tuning. However, it lacks support for tuning categorical parameters and requires domain expertise for setting up tuning instances for complex, distributed applications. Twine [156] is Meta's cluster management system that provides workload-specific customizations such as tuning of hardware and OS settings. OpenTuner [42] provides a framework to build domain-specific tuners.

4.11 Conclusions

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

Chapter 5

Proposed Work: Bottleneck Detection and Mitigation

This chapter focuses on our proposed work. We will first discuss some preliminary results, followed by the proposed work that we plan to explore.

The microservices architecture enables independent development and maintenance of application components through its fine-grained and modular design. This has enabled rapid adoption of microservices architecture to build latency-sensitive online applications. In such online applications, detecting and mitigating sources of performance degradation (bottlenecks) are critical. However, the modular design of microservices architecture leads to a large graph of interacting microservices whose influence on each other is non-trivial, complicating bottleneck detection and mitigation. This chapter begins with a discussion of our preliminary work, in which we investigate the efficiency of Graph Neural Network models in identifying bottlenecks. This is followed by a review of additional challenges that require resolution and our proposed solution to overcome them.

5.1 Preliminary Work

5.1.1 Introduction

The microservices architecture is an architectural style that allows applications to be decomposed into fine-grained, modular, and interacting services, called *microservices*. Under this architecture, each microservice can be independently designed, thereby enabling independent development, maintenance, scaling, and fault isolation (at the level of microservices) [73]. These benefits make the microservices architecture well suited for designing online, customer-facing applications where performance and availability are critical [61, 62]. Accordingly,

microservices applications are widely replacing monolithic or multi-tier applications at Amazon, Netflix, Alibaba, Twitter, Apple, and Ebay [73, 112].

Detecting and mitigating performance bottlenecks in online applications is crucial to provide a good customer experience [46, 62]. For example, experiments at Amazon showed that an additional 100ms of latency can reduce the amount of sales by 1%; similarly, experiments at Google showed that increasing the time to display search results by 500ms can reduce the revenue by 20% [99]. Long tail latencies that significantly affect the revenues of online applications are often a result of *performance bottlenecks* that do not necessarily lead to errors or faults and instead arise due to resource saturation, resource contention, or microservices application misconfiguration [73, 74, 129, 146]. Regardless of the underlying cause of performance bottlenecks, it is essential to have a technique that quickly adapts to dynamic online workloads and accurately detects bottlenecks with high recall and precision. A low recall is especially problematic as it implies that performance issues go unaddressed.

Microservices architecture has unique characteristics compared to other architectural styles that complicates bottleneck detection:

- While the modular architecture allows isolating performance issues at the level of individual microservices, the complex interaction *between* microservices leads to back-pressure effects and cascading performance degradation, making it difficult to precisely pinpoint the performance bottleneck(s) [126].
- Employing data-driven approaches that can learn such complex interactions is difficult due to scarcity of labeled data for bottlenecked class in production systems [74].
- Frequent software updates, and components like caches, message queues, etc., which are inherent to microservices architecture, lead to time-varying interactions between microservices [129, 112] necessitating a technique that can generalize to such dynamicity.

For applications implemented using monolithic or multi-tier architecture, the problem of bottleneck detection has been studied extensively [133, 45, 80, 36, 57, 139, 154, 163, 41, 164]; these studies continue to influence bottleneck detection research for microservices. For the microservices architecture, a popular approach to detect bottlenecks is to employ end-to-end distributed tracing systems like Jaeger [9], that are commonly employed by distributed systems deployed in the industry today [113]. However, such systems cannot capture the complex relationships between different microservices [129]; further, such systems still require manual effort and insight to actually detect performance bottlenecks. In general, the problem of detecting bottlenecks has garnered wide attention from the academic community as well [77, 81, 52, 171, 79, 107, 173, 162, 103, 170]. Recognizing the complex relationship

between different microservices and the conditions that lead to anomalous behaviour, which are not always known in advance, is difficult for simpler algorithmic approaches to detect [74]. Recently, the availability of vast amount of tracing data has motivated data-driven approaches for performance management of microservices architecture [74, 129, 112, 72]. However, prior works that incorporate data-driven approaches either fail to fully use the structural information of the application deployment [129, 74], or use multiple complex models, thereby complicating the solution [72].

This preliminary work explores the use of *Graph Neural Networks* (GNNs) [59, 180] to detect bottlenecks in online microservices applications. GNNs are ideally suited for analyzing microservices applications:

- GNNs and their variants have produced ground-breaking performance on graph data [180] making them a natural choice for analyzing microservices call graphs [126, 112, 116].
- Models like GNNs are ideally suited to capture back-pressure and cascading performance degradation [73, 126] along the call graphs as they learn the dependence of graphs via message passing between the nodes of graphs [180].
- GNNs can generalize to dynamic graphs through transfer learning [86, 83] making them an ideal choice for microservices architecture where the call graphs are dynamic in nature [112, 129], saving retraining costs.
- GNN architectures can be regularized to ensure representation learning equilibrium across multiple classes thereby avoiding the multi-class imbalance problem seen in traditional ML algorithms [142]. The difficulty in collecting traces with bottlenecks in production systems makes GNN an ideal choice as it does not overfit on the majority (non-bottlenecked) class [74].

Motivated by the above observations, this preliminary work explores the use of GNNs for detecting performance bottlenecks in microservices applications by designing B-MEG (Bottlenecked-Microservices Extraction using GNNs), a framework with two stages of GNN models. Preliminary results on a public dataset [128] are encouraging and show that B-MEG performs better than existing work that we compared against [129] for benchmark applications with a large number of microservices and complex call graphs (even when the training dataset is highly imbalanced). Compared to the Support Vector Machine (SVM) model used in existing work, B-MEG provides up to 15% and 14% improvements in accuracy and precision, respectively, and close to 10× improvement in recall of the bottlenecked classes. A detailed empirical comparison of B-MEG against other models and tools, such as those discussed in Section 5.1.2, is left for future work.

5.1.2 Background and Related Work

Performance Anomalies and Bottlenecks

In systems performance, an anomaly is a deviation from the expected performance and performance bottlenecks are the root causes of such performance anomalies. The bottlenecks can be broadly categorized into the resource saturation bottlenecks and the resource contention bottlenecks [89]. These performance bottlenecks can manifest as single, multiple, and shifting or cascading bottlenecks [89]. The single bottlenecks are usually the result of resource saturation, whereas the multiple bottlenecks result from interdependency between different components of the system. The cascading bottlenecks, where bottleneck shifts from component to component, is a common behaviour seen in microservices architecture that complicates performance management [89, 73, 126].

Call Graphs and Traces

The series of Remote Procedure Calls (RPC) between microservices that service a user request is called a *call graph* [112]. The nodes of the call graph are RPCs of microservices and the edges correspond to an invocation of RPC from an upstream microservice to a downstream microservice. An analysis of microservices deployment in Alibaba clusters showed that at least 10% of the call graphs contain more than 40 microservices, and some call graphs can have thousands of microservices [112].

A single request type can have different call graphs due to different user parameters, components like caches and message queues, and asynchronous executions [112]. Further, agility in microservices architecture can lead to updates in microservices that can change the dependencies between them, thereby changing the call graphs.

Call graphs can be obtained using end-to-end tracing systems like Jaeger [9]. A *trace* is a data/execution path through the system, and can be thought of as a directed acyclic graph of spans, where a span is a logical unit of work. A distributed application can be instrumented at the RPC-level to get call graphs of each request.

Luo et al. [112] analyze large-scale deployments of microservices at Alibaba clusters. The authors note that at least 10% of the call graphs contain more than 40 microservices, and some call graphs can have hundreds and thousands of microservices. The authors also talk about the highly dynamic nature of call graphs during runtime noting that a single online service can have more than nine classes of topologically different call graphs.

Graph Neural Networks (GNN)

GNNs are neural network models that are designed to learn representations on graph-structured data via feature propagation and aggregation. The input to a GNN is the graph

representation of the problem being solved, where the graph could be explicit like in the case of call graphs, or implicit where an effort is involved to build the graph [180]. GNN outputs a representation for the input graph, called the embedding, using the features of the initial graph representation and the structure of the graph. These learnt representations are used to perform downstream tasks like graph classification, graph clustering, node classification, etc. The key advantage of GNN compared to standard ML frameworks is that GNNs can provide hierarchical convolutions in non-euclidean spaces. This is accomplished by a message passing process aggregating the embeddings of the neighbors of individual nodes, which in turn contain information about their neighbors. This way, the influence of neighboring microservices in a call graph can be learnt and the patterns that lead to propagation of bottlenecks to neighbors can be detected.

Related Work

Bottleneck detection in multi-tier distributed systems: Wang et al. [164] empirically show that system services like garbage collectors and practices such as VM colocation can lead to fluctuating bottlenecks which are difficult to detect as no single physical or virtual resource is completely saturated. In a follow-up work [163], the authors define bottlenecks that are caused by transient events like garbage collection and bursty workloads as "transient bottlenecks" and detect them using correlation analysis. The above works do not consider the effects of the connected components of a distributed application, which is necessary when analyzing microservices applications as bottlenecks propagate to neighbors.

Bottleneck detection in microservices applications: There is a large body of literature related to the general problem of bottleneck detection; we refer interested readers to a recent survey [144]. We now discuss more closely related prior works to put our work in context. FIRM [129] uses a Support Vector Machine (SVM) model to detect bottlenecks on the critical path of the call graph. The SVM model is trained using hand-crafted features that capture the per-critical-path and per-microservice performance variability. However, FIRM does not capture structural effects of call graphs as it treats each microservice independently for bottleneck detection.

Seer [74] is an online cloud performance debugging system that leverages deep learning to detect and prevent QoS violations. Seer uses a hybrid neural network consisting of CNN and LSTM networks to learn spatial and temporal patterns that lead to QoS violations. However, analysis of Alibaba's production systems suggests that CNN-based approaches fail to characterize complex graph dynamics and are not applicable to real-world applications; instead, the authors suggest the use of GNNs [112], motivating our work.

Sage [72] uses Causal Bayesian Network (CBN) to capture the dependencies between microservices. However, the assumption in Sage that the latency of non-leaf nodes in the call

graph is determined by the wait time of its child nodes might not always hold. When a non-leaf child node is a message queue [112], the parent communication could be synchronous or asynchronous, depending on whether the queue is full or not [112]. Recent works [174, 168] have shown the ability of GNNs to capture such causal relations, making additional models to capture causality redundant.

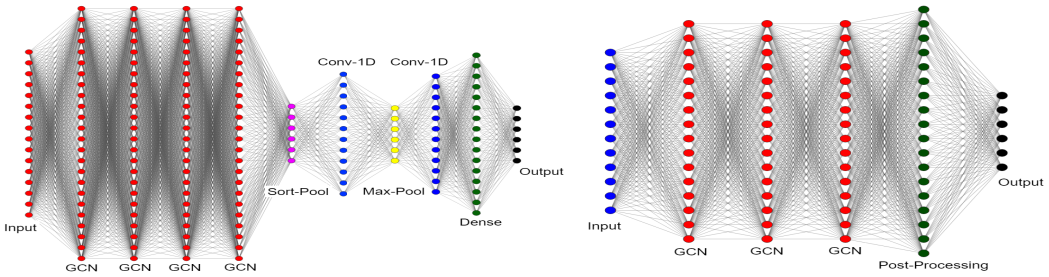
SuanMing [79] presents a framework for predicting future root causes to prevent the consequent performance loss. The framework consists of a *Load Forecaster* which predicts the number of user requests, which is an input to the next stage, *Predictor* which predicts the performance of each microservice. The framework compares the predictions with user-specified goals to detect anticipated performance degradation. However, the assumption in SuanMing that the performance of the application is only dependent on type and amount of requests arriving at each service instance need not hold for data stores of the application which affect performance significantly [73, 74]. Even for stateless microservices, performance can depend on the payload size.

T-Rank [173], using latency as a bottleneck metric, detects bottlenecks based on Spectrum Based Fault Localization (SBFL). However, SBFL cannot capture the complex nature of microservices and incorrectly categorizes hot-spots, microservices that are shared across a significant number of call graphs [112], as bottlenecks.

Brandón et al. [52] present a graph-based framework that employs expert knowledge to detect bottlenecks. Through this framework, the authors also demonstrate the advantages of using graph techniques over ML techniques that do not exploit graph data. Our framework combines these two strategies by using a graph ML technique and alleviates the need of expert knowledge.

Application Performance Monitoring (APM) tools: APM tools provide the ability to observe and manage the performance of an online application, including bottleneck detection. AppDynamics [29] leverages specialized ML models and various metrics collected across the application to detect bottleneck microservices. Dynatrace [31] uses context (topology, traces, and code-level) information to build and analyse a fault-tree to pinpoint bottlenecks.

Use of GNN for microservices application: GNNs have been used to analyze microservices applications but not for the specific problem of bottleneck detection. For instance, GRAF [126] is a resource allocation framework that utilizes gradient descent, using a trained GNN model, to detect if a given CPU configuration would violate the SLO. Luo et al. [112] perform a comprehensive analysis of the large-scale deployment of microservices in Alibaba's production clusters and observe that the same service (application) can have multiple call graphs, complicating performance analysis. Hence, they use InfoGraph [150], a GNN based learning scheme, to cluster call graphs that are similar in terms of topology and composition



(a) Graph classifier (stage 1) (b) Node classifier (stage 2)

Fig. 5.1 The two stages in the B-MEG framework.

to simplify performance analysis. Mathai et al. [116] use GNNs to analyze a monolithic application by translating it into microservices and employing a novel heterogeneous GNN to analyze and cluster the monolithic application's programs, tables, and files (as nodes) and relationships between them (as edges).

5.1.3 Objective and System Design

We divide the problem of detecting bottlenecks into two sub-problems, the detection of potential anomalous traces (i.e., traces affected by bottlenecks), followed by detection of potential bottlenecks in such anomalous traces. We translate the sub-problems of detecting anomalous traces and potential bottlenecks into graph classification and node classification tasks, respectively. This division of problem is motivated by the benefits of hierarchical classifiers [143].

In a flat classification, where a single classifier classifies all the examples, the number of classes for an application with n microservices would be $n+1$, one for each microservice and one additional class that corresponds to no bottlenecks. Based on the intuition that traces with bottlenecks would be similar to each other irrespective of the specific bottlenecks [121], we categorize them into one meta class—anomalous traces. This allows the use of a binary classifier as the first stage that classifies a trace as anomalous or regular. The traces classified as anomalous are provided as input to the second stage that detects potential bottlenecks in them. The main disadvantage of this design is the error propagation from first stage which can be controlled by varying the classification threshold of the first stage. We empirically compared the performance of a flat classification model versus the hierarchical model (B-MEG) and found that the hierarchical model leads to two simpler models with better performance which further motivated this design.

The B-MEG framework, as shown in Figure 5.1, consists of 2 stages with the first stage responsible for classifying potential anomalous traces and the second stage responsible for classifying potential bottlenecks. The first stage uses a Deep Graph Convolutional Neural Network (DGCNN) [60] for classifying if a trace is anomalous, and the second stage uses

an inductive graph convolution training regime for pinpointing the microservices that are responsible for causing the anomaly. The choice of DGCNN for graph classification is due to its superior performance on inductive learning of graph representations without feature engineering. The node classifier is a vanilla Graph Convolution Network (GCN) architecture where the number of convolution layers were decided based on experiments.

The architecture of the DGCNN model, shown in Figure 5.1a, consists of four sequential stages: (i) four GCN layers to hierarchically extract the local substructure features of a node and define a node ordering [98]; (ii) one Sort Pooling layer for sorting the ordering under a pre-defined ordering and unifying the input sizes [177]; (iii) a sequence of traditional Convolution 1D layer, a max-pooling layer, and another Convolution 1D layer to read the sorted graph representations; and (iv) one post-processing dense layer followed by a softmax layer to make predictions.

For the node-classification task we use a semi-supervised graph convolution framework with three GCN layers, followed by a post-processing feed-forward and a softmax layer for predictions. The GCN layers hierarchically extract node features and pass it on to post-processing layer for classification.

5.1.4 Evaluation

In this section, we first describe the publicly available dataset [128] that is used to evaluate the efficacy of B-MEG in detecting potential bottlenecks in microservices applications. Following this, we describe the methodology and compare the preliminary results against SVM, the model used to detect bottlenecks in FIRM [129].

Dataset

The dataset [128] released as part of the FIRM project [129] contains traces of social networking, media microservices, and hotel reservation applications from the DeathStarBench [73] suite and TrainTicket benchmark [181] application. Most traces consist of a single bottleneck, the cause of which is an artificially induced resource interference, while the remaining traces have no bottlenecks. We refer the readers to Gan et al. [73] and Zhou at al. [181] to learn more about the functionalities of these applications.

Methodology

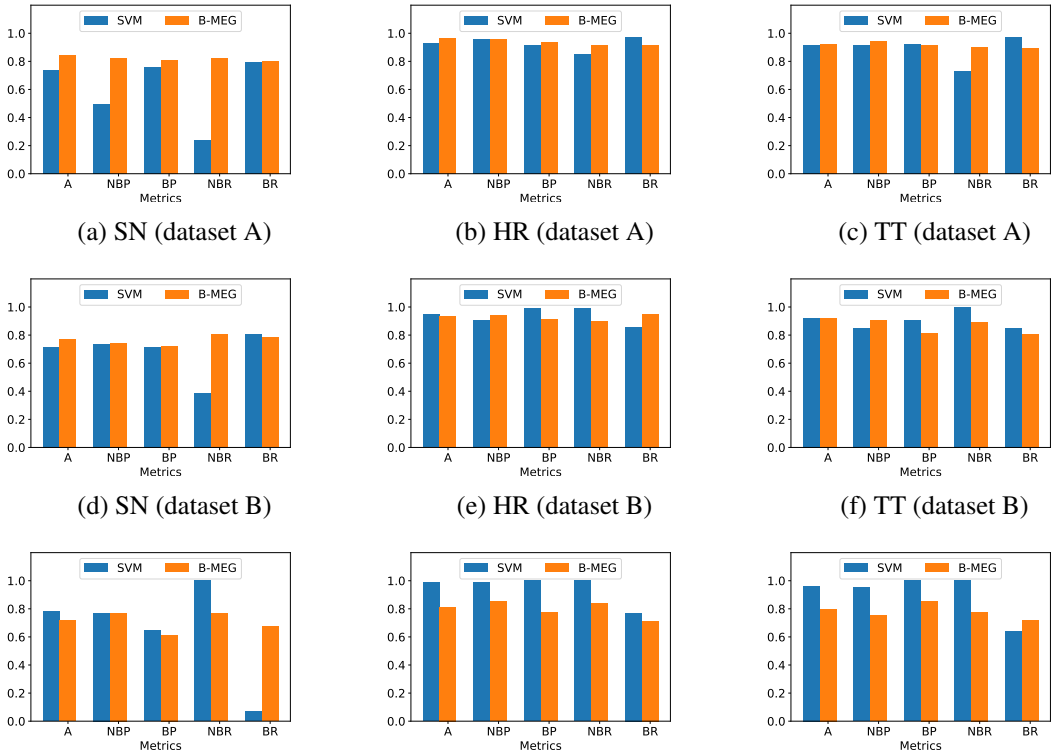
In this preliminary work, we focus our methodology on studying the effectiveness of GNN models on *imbalanced* datasets, which are the norm given the scarcity of production systems traces with bottlenecks [74]. To evaluate B-MEG's ability to handle the multi-class imbalance problem, we create three datasets each consisting of 790,000 traces—A, B, C—with the ratio of number of traces in the dataset with a microservice as the bottleneck to the number

of traces without bottlenecks being 0.3, 0.1, and 0.01, respectively. The choice of 0.3 is to evaluate the performance of B-MEG for a fairly balanced dataset. The choice of 0.1 and 0.01 is motivated by similar ratios reported in production systems [103]. The datasets are created by random sampling to avoid any unexpected bias in them. We empirically evaluated how the performance of B-MEG varies with the total size of the dataset and chose the size at which the performance plateaued. The training time for the applications varies from 2–3 hours.

We use the bottleneck detection technique from FIRM [129] as the baseline to evaluate B-MEG's performance. FIRM [129] derives two features, the relative importance and congestion intensity, from service time of microservices to train an SVM model to detect bottlenecks. Similar to FIRM [129], we train both the models using service time of microservices as feature as it correlates well with bottleneck occurrence, but without any feature engineering. Using 80% of the traces from each class as the training data, both the models are trained separately and inductively where each trace is treated as a stand-alone instance; the remaining 20% dataset forms the test data. Unlike prior works [74, 129] that focus only on accuracy, we use other metrics like recall (measure of how many of the true bottlenecks get detected) and precision (measure of how many of the classified bottlenecks are truly bottlenecks) which, as discussed in Section 5.1.1, are important when the dataset is imbalanced.

Results

Evaluation on Imbalanced datasets Figure 5.2 shows the results for datasets A, B, and C (with different degree of class imbalance) and different benchmarking applications for SVM and B-MEG. For the social networking (SN) application, as seen in Figure 5.2a, B-MEG outperforms SVM with respect to all the metrics for dataset A. This suggests B-MEG's ability to effectively learn patterns that cause bottlenecks with a fairly imbalanced dataset without any feature engineering. For dataset B, B-MEG does better than SVM for all the metrics except for recall of bottlenecked classes, with SVM's value being 0.81 and B-MEG's 0.78. However, this advantage of SVM comes with a very small recall (0.39) for the non-bottlenecked class, an undesirable trade-off. Moreover, B-MEG is capable of maintaining a good trade-off between overall precision (0.74) and recall (0.8) among all the classes, providing a high recall (0.81) for the non-bottlenecked class even when there is significant class imbalance. For dataset C, where the class imbalance is extreme, SVM has higher accuracy (0.78) than B-MEG (0.71), but suffers from a poor recall for bottlenecked classes (0.07). B-MEG on the other hand, provides a reasonable recall for bottlenecked classes (0.67), proving its ability to balance precision and recall even when the class imbalance is extreme. We see similar trends as dataset C when we further increase the class imbalance ratio to 0.001 (note that dataset C has a ratio of 0.01). We note that the call graph of social networking application in the FIRM dataset [128] has 31 microservices and 18 different paths



(a) SN (dataset A) (b) HR (dataset A) (c) TT (dataset A)

(d) SN (dataset B) (e) HR (dataset B) (f) TT (dataset B)

(g) SN (dataset C) (h) HR (dataset C) (i) TT (dataset C)

Fig. 5.2 Performance comparison of SVM and B-MEG on the traces of social networking (SN) [73], hotel reservation (HR) [73], and train ticket (TT) [181] applications. Metrics employed are accuracy (A), precision for non-bottlenecked (NBP) and bottlenecked classes (BP), recall for non-bottlenecked (NBR) and bottlenecked classes (BR). For all metrics, higher values are better.

from the root of the call graph to the leaf nodes, advocating the effectiveness of B-MEG in learning patterns in complex call graphs to detect bottlenecks.

Figures 5.2b, 5.2e, and 5.2h show that SVM either outperforms or performs similarly to B-MEG across all the datasets. Figures 5.2c, 5.2f, and 5.2i show similar trends for the train ticket application. Considering that the call graphs of hotel reservation and train ticket applications consist of 5 microservices with 3 different paths, and 11 microservices with 7 different paths, respectively, the results are not surprising. SVM's inability to exploit the structural information does not penalize its performance for these applications with a simple call graph. On the contrary, the simple graphs aid SVM in learning thresholds that signal bottlenecks. However, we note that B-MEG still does maintain a good balance between precision and recall for these two applications.

The above evaluation results show that even when the class imbalance is extreme, B-MEG effectively detect bottlenecks for microservices applications with large and complex call graphs. Given that such imbalance is the norm in production system traces [112, 74], we are

encouraged by B-MEG's ability to maintain a good trade-off between precision and recall in such cases.

5.2 Proposed Work

The preliminary work makes a case for employing GNNs to detect bottlenecks in applications designed using the microservices architecture. In our preliminary experiments, B-MEG performs better at detecting bottlenecks on imbalanced datasets for large and complex call graphs than SVM. However, we only considered traces with single bottlenecks due to the unavailability of multiple and cascading ones. In real-world applications, multi and cascading bottlenecks are more common and harder to detect [74]. Our preliminary work doesn't account for the dynamicity of call graphs, which is inherent to microservices applications [112, 129]. Moreover, real-world applications comprise a wide range of telemetry (e.g., utilization) data that are not exploited in our preliminary and other related works. To have a more representative solution that is applicable to real-world applications, we propose to work on the following broad tasks for the completion of this thesis:

- **Dataset:** We plan to create and open source a dataset with bottlenecks to further research on bottleneck detection. The lack of open source datasets, a major concern in the general area of performance bottleneck detection in systems [89], also applies to the microservices architecture. We plan to work with our industry collaborators to create a dataset that is representative of the real-world scenarios.

- **Improve the bottleneck detection methodology:** In our preliminary work, we make several assumptions to simplify the problem. We don't consider multi and cascading bottlenecks, which are common and harder to detect in real-world applications. We don't exploit temporal features that are useful in detecting bottlenecks [74]. Dynamic call graphs, inherent to microservices applications, can be tackled using transfer learning, which requires further investigation. The superior performance of simpler models like SVM on simple call graphs motivates exploring a multi-model inference where the model used for inference would depend on the complexity of the call graph. Architectural changes to the model, aiming to obtain better performance and reduce training and inference time, require further attention.

- **Explore different bottleneck mitigation strategies:** The source of bottlenecks in real-world applications can be resource saturation, resource contention, application misconfiguration, cluster misconfiguration, scheduling policies, etc., each of which can be mitigated in various ways [74, 129, 109, 152, 108]. It is important to effectively detect the source of the bottleneck and take the best action to mitigate it.

5.2.1 Challenges

The various challenges related to our proposed work are:

- **Dataset:** The publicly available traces [128] only consist of single bottlenecks caused via a single source (resource interference). In production systems, the bottlenecks could also be of other types, multiple and cascading, due to different sources, including resource saturation, misconfiguration, application bug, etc. Moreover, the publicly available traces only contain information on the latency of each microservice. Other metrics, like resource utilization by the microservices, can aid bottleneck detection. However, simulating other types of bottlenecks, especially the cascading ones, is difficult as it results from a complex interaction among the microservices [74]. Collecting traces and telemetry has a compute cost and, occasionally, performance cost. Depending on the telemetry implementation, it could be part of the critical path, making the granularity of telemetry collection an interesting problem [129].

- **Bottleneck detection:** The bottleneck detection technique should ideally detect all the types of bottlenecks (single, multiple, cascading) irrespective of the source of bottlenecks. The cascading bottlenecks, common in microservices, are generally harder to detect [129, 74]. It should also be capable of learning patterns without needing a lot of data, as there is a scarcity of labeled data for bottlenecked classes in production systems. Additionally, the technique should generalize to time-varying interactions between microservices.

- **Bottleneck mitigation:** Assuming a list of bottlenecks and their characteristics (features) are passed to the mitigation technique, it should first detect the source of the bottleneck. Once the source is detected, the steps taken by the technique to alleviate the bottleneck should take action quickly as microservices tend to have long recovery periods [74]. Since the mitigation steps would be applied on an online application, the steps themselves must not cause performance degradation.

5.2.2 Proposed Solution

We plan to explore the following tasks to tackle the challenges of our proposed work:

- **Dataset:** Firstly, we will build an automated framework to create the dataset. The framework, through simple configuration files that provide the requirements for the traces, should generate traces that can contain different bottlenecks via different sources. We will create resource interference using anomaly injectors that consume resources like CPU, memory, and network on the same node where the target microservices

are running. To simulate resource saturation, we will vary the container's resource limits. For application misconfiguration, we will manually set the parameters of the target microservice to values that lead to poor performance. We will carry out these experiments on popular benchmarking applications [73, 181].

- **Bottleneck detection:** We will explore a) learning temporal patterns by using Recurrent Neural Networks, b) transfer learning to make B-MEG generalize for time-varying interactions between microservices, c) Multi-model (GNN and SVM) inference where SVM is chosen for simpler call graphs and GNN is chosen for complex call graphs to reduce inference latency, and d) detailed analysis of dataset size's impact on performance and training effort.

- **Bottleneck mitigation:** We will use data-driven approaches to detect the source of bottlenecks. Once the source is detection, we will employ different techniques appropriate to the source to mitigate the bottleneck. For example, if resource saturation is the source of the bottleneck, we will vertically or horizontally scale the microservice.

5.2.3 Conclusion

Our initial work supports the use of GNNs for detecting bottlenecks in applications designed using microservices architecture. In our preliminary experiments, the B-MEG model outperformed the SVM model in identifying bottlenecks on imbalanced datasets for large and complex call graphs. We plan to extend the preliminary work by creating a representative dataset of traces for bottleneck detection, improving the bottleneck detection methodology and exploring different mitigation strategies.

Chapter 6

Conclusion

Microservice Architecture has gained widespread use for building large-scale online applications. It involves breaking down an application into small, independent services known as microservices that each handle a specific business function and communicate through APIs. This approach offers greater agility, scalability, and fault tolerance, leading to its adoption by companies such as Microsoft, Amazon, Netflix, and Twitter over traditional monolithic or multi-tier architectures. Despite its benefits, managing the complex interactions between microservices presents challenges in performance management. In this thesis, we specifically target the problems of configuration tuning and bottleneck detection and mitigation.

We examine the effectiveness of various optimization and dimensionality reduction techniques for the pre-deployment tuning of microservices applications. Despite the popularity of microservices architecture, there is a lack of focus on setting configurations of individual microservices for improved performance, often resulting in sub-par performance due to default settings. Our work highlights the importance of configuration tuning for microservices. We examine the problem, identify key challenges, and evaluate techniques to address them. Our research on different optimization techniques shows significant improvement in tail latency, up to 46%, through configuration tuning. Although most algorithms perform well, the optimal one depends on the application and combining algorithms may result in even better performance for some applications. Our analysis also reveals that the optimal configuration of a microservice, such as MongoDB, varies across applications and even instances within the same application. We also investigate ways to reduce the tuning effort across algorithms. Our study of dimensionality reduction techniques reveals that focusing on tuning the critical path microservices with the highest service time variability is an effective way to reduce tuning effort, typically by 2-6 times, while ensuring good coverage of important parameters (confirmed by fANOVA analysis). Our conclusion is that dimensionality reduction based on

system characteristics is a viable solution to the complex problem of optimizing a large state space.

Recognizing the difficulties in the post-deployment tuning of real-world applications, we introduce SelfTune and OPPerTune, frameworks designed to address these challenges. SelfTune, an RL-based framework, allows operators to tune large-scale microservices applications and significantly improves system performance through experiments on Kubernetes's Vertical Pod Autoscaler and the DeathStarBench microservice benchmark. We have created, developed, and widely deployed the OPPerTuneconfiguration tuning service. OPPerTunedistinguishes itself from other systems work on performance optimization and configuration tuning in multiple ways: 1) addressing post-deployment tuning challenges, 2) prioritizing sample complexity and tuning cost, unlike systems relying on offline training or controlled settings, 3) providing a complete solution for configuration tuning that is widely applicable to various systems optimization scenarios. Through extensive experiments on a benchmarking application using synthetic and production traces, we demonstrate that our techniques achieve state-of-the-art performance, are sample-efficient, and reduce tuning costs.

We present our preliminary work on bottleneck detection and mitigation. Our approach uses graph learning algorithms to identify bottlenecks in an open-sourced dataset of DeathStarBench's distributed traces. Our model provides better results than prior works on complex call graphs. Furthermore, we delve into our proposed work, examine its challenges, and briefly discuss the potential solutions we intend to probe.

References

[1] Apache thrift. https://thrift.apache.org/.

[2] Beginner's guide. nginx.org/en/docs/beginners_guide.html.

[3] Chaos monkey. https://netflix.github.io/chaosmonkey/.

[4] Controlling nginx. http://nginx.org/en/docs/control.html.

[5] Deathstarbench. github.com/delimitrou/DeathStarBench.

[6] Elastic search. https://www.elastic.co/elasticsearch/.

[7] Framework for tuning microservices applications. https://github.com/PACELab/microservices-tuning.

[8] grpc. https://grpc.io/.

[9] Jaeger. https://www.jaegertracing.io/.

[10] Kubernetes. https://kubernetes.io/.

[11] Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/.

[12] Kubernetes Vertical Pod Autoscaler. https://github.com/kubernetes/autoscaler/blob/master/vertical-pod-autoscaler/pkg/recommender/main.go.

[13] Kubernetes Vertical Pod Autoscaler. https://github.com/kubernetes/autoscaler/blob/master/vertical-pod-autoscaler/.

[14] memcached(1). https://linux.die.net/man/1/memcached.

[15] Microservices. https://martinfowler.com/articles/microservices.html.

[16] MongoDB. https://docs.mongodb.com/manual/ref\protect\discretionary{\char\hyphenchar\font}{}{}erence/parameters/.

[17] Mongodb jira. jira.mongodb.org/browse/SERVER-19911.

[18] MongoDB Server Parameters. https://docs.mongodb.com/manual/ref\protect\discretionary{\char\hyphenchar\font}{}{}erence/parameters/.

[19] Nginx core functionality. https://nginx.org/en/docs/ngx_core_module.html.

[20] Planned maintenance window in aks. https://azure.microsoft.com/en-in/updates/public-preview-planned-maintenance-windows-in-aks/.

[21] Recreate Deployment. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#strategy.

[22] Redis configuration. https://redis.io/topics/config.

[23] Soa vs microservices. https://www.ibm.com/cloud/blog/soa-vs-microservices.

[24] The `scikit-optimize` library: Bayesian Optimization using Gaussian Process. https://scikit-optimize.github.io/stable/modules/generated/skopt.gp_minimize.html.

[25] Tuning CPU half-life decay parameter. https://github.com/kubernetes/autoscaler/issues/3684.

[26] Tuning nginx for performance. https://www.nginx.com/blog/tuning-nginx/.

[27] The Vowpal Wabbit library. https://github.com/VowpalWabbit/vowpal_wabbit/wiki/Contextual-Bandit-algorithms.

[28] wrk2 HTTP Workload Generator. https://github.com/giltene/wrk2.

[29] Anomaly detection. https://docs.appdynamics.com/4.5.x/en/appdynamics-essentials/alert-and-respond/anomaly-detection, 2022.

[30] Fiber. https://github.com/gofiber/fiber, 2022.

[31] Root cause analysis. https://www.dynatrace.com/support/help/how-to-use-dynatrace/problem-detection-and-analysis/problem-analysis/root-cause-analysis, 2022.

[32] Scikit-optimize. https://github.com/scikit-optimize/scikit-optimize, 2022.

[33] ABERNETHY, R. *The Programmer's Guide to Apache Thrift*. 2018.

[34] AGARWAL, A., BIRD, S., COZOWICZ, M., HOANG, L., LANGFORD, J., LEE, S., LI, J., MELAMED, D., OSHRI, G., RIBAS, O., ET AL. Making contextual decisions with low technical debt. *arXiv preprint arXiv:1606.03966* (2016).

[35] AGARWAL, A., BIRD, S., COZOWICZ, M., HOANG, L., LANGFORD, J., LEE, S., LI, J., MELAMED, D., OSHRI, G., RIBAS, O., ET AL. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966 7* (2016).

[36] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), SOSP '03.

[37] AKGUN, I. U., AYDIN, A. S., BURFORD, A., MCNEILL, M., ARKHANGELSKIY, M., AND ZADOK, E. Improving storage systems using machine learning. *ACM Trans. Storage* (nov 2022).

[38] ALABED, S., AND YONEKI, E. High-dimensional bayesian optimization with multi-task learning for rocksdb. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (New York, NY, USA, 2021), EuroMLSys '21, Association for Computing Machinery, p. 111–119.

[39] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 469–482.

[40] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (2017), NSDI'17.

[41] ALSAYASNEH, M. *On the identification of performance bottlenecks in multi-tier distributed systems*. PhD thesis, Université Grenoble Alpes [2020-....], 2020.

[42] ANSEL, J., KAMIL, S., VEERAMACHANENI, K., RAGAN-KELLEY, J., BOSBOOM, J., O'REILLY, U.-M., AND AMARASINGHE, S. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation* (2014), pp. 303–316.

[43] ARSANJANI, A. Service-oriented modeling and architecture. *IBM developer works* (01 2004).

[44] AUER, P., CESA-BIANCHI, N., FREUND, Y., AND SCHAPIRE, R. E. The non-stochastic multiarmed bandit problem. *SIAM journal on computing 32*, 1 (2002), 48–77.

[45] BALBO, G., AND SERAZZI, G. Asymptotic analysis of multiclass closed queueing networks: Multiple bottlenecks. *Performance Evaluation* (1997).

[46] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.

[47] BERGSTRA, J., BARDENET, R., BENGIO, Y., AND KÉGL, B. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems* (2011).

[48] BERGSTRA, J., YAMINS, D., AND COX, D. D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning* (2013), ICML'13.

[49] BIETTI, A., AGARWAL, A., AND LANGFORD, J. A contextual bandit bake-off. *Journal of Machine Learning Research 22*, 133 (2021), 1–49.

[50] BILAL, M., CANINI, M., AND RODRIGUES, R. Finding the right cloud configuration for analytics clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (2020), SoCC '20.

[51] BILAL, M., SERAFINI, M., CANINI, M., AND RODRIGUES, R. Do the best cloud configurations grow on trees? an experimental evaluation of black box algorithms for optimizing cloud workloads. *Proc. VLDB Endow.* (2020).

[52] BRANDON, A., SOLÉ-SIMÓ, M., HUÉLAMO, A., SOLANS, D., PÉREZ, M., AND MUNTÉS-MULERO, V. Graph-based root cause analysis for service-oriented and microservice architectures. *Journal of Systems and Software* (2020).

[53] BREIMAN, L., FRIEDMAN, J., OLSHEN, R., AND STONE, C. Classification and regression trees.

[54] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Borg, omega, and kubernetes. *Communications of the ACM 59*, 5 (2016), 50–57.

[55] CAO, Z., KUENNING, G., AND ZADOK, E. Carver: Finding important parameters for storage system tuning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (2020).

[56] CAO, Z., TARASOV, V., TIWARI, S., AND ZADOK, E. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[57] CASALE, G., AND SERAZZI, G. Bottlenecks identification in multiclass queueing networks using convex polytopes. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.* (2004).

[58] CEREDA, S., VALLADARES, S., CREMONESI, P., AND DONI, S. Cgptuner: A contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions. *Proc. VLDB Endow. 14*, 8 (oct 2021), 1401–1413.

[59] CHAMI, I., ABU-EL-HAIJA, S., PEROZZI, B., RÉ, C., AND MURPHY, K. Machine learning on graphs: A model and comprehensive taxonomy, 2021.

[60] CHEN, M., WEI, Z., HUANG, Z., DING, B., AND LI, Y. Simple and deep graph convolutional networks. In *ICML* (2020).

[61] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Communications of ACM 56*, 2 (2013), 74–80.

[62] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP '07.

[63] DELIMITROU, C., AND KOZYRAKIS, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices 48*, 4 (2013), 77–88.

[64] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not. 49*, 4 (Feb. 2014), 127–144.

[65] DRAGONI, N., GIALLORENZO, S., LAFUENTE, A. L., MAZZARA, M., MONTESI, F., MUSTAFIN, R., AND SAFINA, L. Microservices: yesterday, today, and tomorrow, 2016.

[66] DUAN, S., THUMMALA, V., AND BABU, S. Tuning database configuration parameters with ituned. *Proc. VLDB Endow. 2*, 1 (aug 2009), 1246–1257.

[67] ELMACHTOUB, A. N., MCNELLIS, R., OH, S., AND PETRIK, M. A practical method for solving contextual bandit problems using decision trees. *Uncertainty in Artificial Intelligence* (2017).

[68] FEKRY, A., CARATA, L., PASQUIER, T., RICE, A., AND HOPPER, A. To tune or not to tune? in search of optimal configurations for data analytics. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2020).

[69] FÉRAUD, R., ALLESIARDO, R., URVOY, T., AND CLÉROT, F. Random forest for the contextual bandit problem. In *Artificial intelligence and statistics* (2016), PMLR, pp. 93–101.

[70] FLAXMAN, A. D., KALAI, A. T., AND MCMAHAN, H. B. Online convex optimization in the bandit setting: gradient descent without a gradient. In *Proceedings of the sixteenth annual ACM-SIAM Symposium on Discrete Algorithms* (2005), pp. 385–394.

[71] GAN, Y., DEV, S., LO, D., AND DELIMITROU, C. Sage: Leveraging ML To Diagnose Unpredictable Performance in Cloud Microservices. In *Workshop on ML for Computer Architecture and Systems (MLArchSys)* (June 2020).

[72] GAN, Y., LIANG, M., DEV, S., LO, D., AND DELIMITROU, C. Sage: Practical and scalable ml-driven performance debugging in microservices. ASPLOS 2021.

[73] GAN, Y., ZHANG, Y., CHENG, D., SHETTY, A., RATHI, P., KATARKI, N., BRUNO, A., HU, J., RITCHKEN, B., JACKSON, B., HU, K., PANCHOLI, M., CLANCY, B., COLEN, C., WEN, F., LEUNG, C., WANG, S., ZARUVINSKY, L., ESPINOSA, M., HE, Y., AND DELIMITROU, C. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2019).

[74] GAN, Y., ZHANG, Y., HU, K., HE, Y., PANCHOLI, M., CHENG, D., AND DELIMITROU, C. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2019).

[75] GARLAN, D., BACHMANN, F., IVERS, J., STAFFORD, J., BASS, L., CLEMENTS, P., AND MERSON, P. *Documenting Software Architectures: Views and Beyond*, 2nd ed. Addison-Wesley Professional, 2010.

[76] GEN, M., AND CHENG, R. *Genetic algorithms and engineering optimization*, vol. 7. John Wiley & Sons, 1999.

[77] GIAS, A. U., CASALE, G., AND WOODSIDE, M. ATOM: Model-Driven Autoscaling for Microservices. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems (ICDCS)* (2019), pp. 1994–2004.

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

[79] GROHMANN, J., STRAESSER, M., CHALBANI, A., EISMANN, S., ARIAN, Y., HERBST, N., PERETZ, N., AND KOUNEV, S. Suanming: Explainable prediction of performance degradations in microservice applications. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2021), ICPE '21, Association for Computing Machinery, p. 165–176.

[80] GUNTER, D., TIERNEY, B., CROWLEY, B., HOLDING, M., AND LEE, J. Netlogger: a toolkit for distributed system performance analysis. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2000).

[81] GUO, X., PENG, X., WANG, H., LI, W., JIANG, H., DING, D., XIE, T., AND SU, L. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2020), ESEC/FSE 2020, Association for Computing Machinery, p. 1387–1397.

[82] HADARY, O., MARSHALL, L., MENACHE, I., PAN, A., GREEFF, E. E., DION, D., DORMINEY, S., JOSHI, S., CHEN, Y., RUSSINOVICH, M., ET AL. Protean:{VM} allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 845–861.

[83] HAN, X., HUANG, Z., AN, B., AND BAI, J. Adaptive transfer learning on graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery amp; Data Mining* (New York, NY, USA, 2021), KDD '21, Association for Computing Machinery, p. 565–574.

[84] HEINRICH, R., VAN HOORN, A., KNOCHE, H., LI, F., LWAKATARE, L. E., PAHL, C., SCHULTE, S., AND WETTINGER, J. Performance engineering for microservices: Research challenges and directions. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion* (New York, NY, USA, 2017), ICPE '17 Companion, Association for Computing Machinery, p. 223–226.

[85] HERODOTOU, H., CHEN, Y., AND LU, J. A survey on automatic parameter tuning for big data processing systems. *ACM Comput. Surv. 53*, 2 (apr 2020).

[86] HU*, W., LIU*, B., GOMES, J., ZITNIK, M., LIANG, P., PANDE, V., AND LESKOVEC, J. Strategies for pre-training graph neural networks. In *International Conference on Learning Representations* (2020).

[87] HUMBLE, J., AND FARLEY, D. Continuous delivery: Reliable software releases through build. *Test, and deployment automation. Pearson Education 1* (2010).

[88] HUTTER, F., HOOS, H., AND LEYTON-BROWN, K. An efficient approach for assessing hyperparameter importance. In *Proceedings of International Conference on Machine Learning 2014 (ICML 2014)* (2014).

[89] IBIDUNMOYE, O., HERNÁNDEZ-RODRIGUEZ, F., AND ELMROTH, E. Performance anomaly detection and bottleneck identification. *ACM Comput. Surv.* (2015).

[90] INDRASIRI, K., AND KURUPPU, D. *gRPC: Up and Running*. 2020.

[91] ISHIKAWA, C., SAKAMURA, K., AND MAEKAWA, M. Dynamic tuning of operating systems. In *Operating Systems Engineering* (Berlin, Heidelberg, 1982), M. Maekawa and L. A. Belady, Eds., Springer Berlin Heidelberg, pp. 119–142.

[92] JIN, T., CAI, Z., LI, B., ZHENG, C., JIANG, G., AND CHENG, J. Improving resource utilization by timely fine-grained scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–16.

[93] KALE, S., REYZIN, L., AND SCHAPIRE, R. E. Non-stochastic bandit slate problems. *Advances in Neural Information Processing Systems 23* (2010).

[94] KAMINSKI, M., TRUYEN, E., BENI, E. H., LAGAISSE, B., AND JOOSEN, W. A framework for black-box slo tuning of multi-tenant applications in kubernetes. In *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds* (New York, NY, USA, 2019), WOC '19, Association for Computing Machinery, p. 7–12.

[95] KANELLIS, K., ALAGAPPAN, R., AND VENKATARAMAN, S. Too many knobs to tune? towards faster database tuning by pre-selecting important knobs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)* (2020).

[96] KARTHIKEYAN, A., JAIN, N., NATARAJAN, N., AND JAIN, P. Learning accurate decision trees with bandit feedback via quantized gradient descent. *Transactions on Machine Learning Research* (Sep 2022).

[97] KARTHIKEYAN, A., NATARAJAN, N., SOMASHEKHAR, G., ZHAO, L., BHAGWAN, R., FONSECA, R., RACHEVA, T., AND BANSAL, Y. SelfTune: Tuning cluster managers. In *To Appear in NSDI* (2023). https://www.microsoft.com/en-us/research/uploads/prod/2022/10/SelfTune_NSDI2023_Cameraready.pdf.

[98] KIPF, T. N., AND WELLING, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[99] KOHAVI, R., HENNE, R. M., AND SOMMERFIELD, D. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2007).

[100] LAIGNER, R., ZHOU, Y., SALLES, M. A. V., LIU, Y., AND KALINOWSKI, M. Data management in microservices: State of the practice, challenges, and research directions. *Proc. VLDB Endow. 14*, 13 (oct 2021), 3348–3361.

[101] LATTIMORE, T., AND SZEPESVÁRI, C. *Bandit algorithms*. Cambridge University Press, 2020.

[102] LI, B., TANG, H., ZHENG, Y., JIANYE, H., LI, P., WANG, Z., MENG, Z., AND WANG, L. Hyar: Addressing discrete-continuous action reinforcement learning via hybrid action representation. In *International Conference on Learning Representations* (2021).

[103] LI, Z., CHEN, J., JIAO, R., ZHAO, N., WANG, Z., ZHANG, S., WU, Y., JIANG, L., YAN, L., WANG, Z., ET AL. Practical root cause localization for microservice systems via trace analysis. In *IEEE/ACM International Symposium on Quality of Service (IWQoS) 2021* (2021).

[104] LI, Z. L., LIANG, C.-J. M., HE, W., ZHU, L., DAI, W., JIANG, J., AND SUN, G. Metis: Robustly tuning tail latencies of cloud systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 981–992.

[105] LILLICRAP, T., HUNT, J., PRITZEL, A., HEESS, N., EREZ, T., TASSA, Y., SILVER, D., AND WIERSTRA, D. Continuous control with deep reinforcement learning. *CoRR* (09 2015).

[106] LIN, C., ZHUANG, J., FENG, J., LI, H., ZHOU, X., AND LI, G. Adaptive code learning for spark configuration tuning. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)* (2022), pp. 1995–2007.

[107] LIU, D., HE, C., PENG, X., LIN, F., ZHANG, C., GONG, S., LI, Z., OU, J., AND WU, Z. Microhecl: High-efficient root cause localization in large-scale microservice systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice* (2021).

[108] LORIDO-BOTRAN, T., AND BHATTI, M. K. Adaptive container scheduling in cloud data centers: A deep reinforcement learning approach. In *Advanced Information Networking and Applications* (Cham, 2021), L. Barolli, I. Woungang, and T. Enokido, Eds., Springer International Publishing, pp. 572–581.

[109] LORIDO-BOTRAN, T., MIGUEL-ALONSO, J., AND LOZANO, J. A. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing 12*, 4 (2014), 559–592.

[110] LUKE, S. *Essentials of Metaheuristics*. Lulu, 2013.

[111] LUO, C., QIAO, B., CHEN, X., ZHAO, P., YAO, R., ZHANG, H., WU, W., ZHOU, A., AND LIN, Q. Intelligent virtual machine provisioning in cloud computing. In *IJCAI* (2020), pp. 1495–1502.

[112] LUO, S., XU, H., LU, C., YE, K., XU, G., ZHANG, L., DING, Y., HE, J., AND XU, C. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2021), SoCC '21, Association for Computing Machinery, p. 412–426.

[113] MACE, J. End-to-End Tracing: Adoption and Use Cases. Survey, Brown University, 2017.

[114] MAHGOUB, A., MEDOFF, A. M., KUMAR, R., MITRA, S., KLIMOVIC, A., CHATERJI, S., AND BAGCHI, S. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020).

[115] MASSON, W., RANCHOD, P., AND KONIDARIS, G. Reinforcement learning with parameterized actions. In *AAAI* (2016).

[116] MATHAI, A., BANDYOPADHYAY, S., DESAI, U., AND TAMILSELVAM, S. Monolith to microservices: Representing application software through heterogeneous gnn, 2021.

[117] MEHTA, S., BHAGWAN, R., KUMAR, R., BANSAL, C., MADDILA, C., ASHOK, B., ASTHANA, S., BIRD, C., AND KUMAR, A. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 435–448.

[118] MILLSAP, R. E. *Journal of Educational and Behavioral Statistics* (1995).

[119] MORICONI, R., DEISENROTH, M., AND SESH KUMAR, K. High-dimensional bayesian optimization using low-dimensional feature spaces. *Mach Learn 109, 1925–1943* (2020).

[120] MOSTOFI, V. M., KRISHNAMURTHY, D., AND ARLITT, M. Fast and efficient performance tuning of microservices. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)* (2021), pp. 515–520.

[121] NEDELKOSKI, S., CARDOSO, J., AND KAO, O. Anomaly detection from system tracing data using multimodal deep learning. In *2019 IEEE 12th International Conference on Cloud Computing* (2019).

[122] NESTEROV, Y. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization 22*, 2 (2012), 341–362.

[123] NORKIN, V. I., PFLUG, G. C., AND RUSZCZYŃSKI, A. A branch and bound method for stochastic global optimization. *Mathematical programming 83*, 1 (1998), 425–450.

[124] NUTINI, J., SCHMIDT, M., LARADJI, I., FRIEDLANDER, M., AND KOEPKE, H. Coordinate descent converges faster with the gauss-southwell rule than random selection. In *International Conference on Machine Learning* (2015), PMLR, pp. 1632–1641.

[125] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), pp. 69–84.

[126] PARK, J., CHOI, B., LEE, C., AND HAN, D. Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies* (2021).

[127] PRATS, D. B., PORTELLA, F. A., COSTA, C. H. A., AND BERRAL, J. L. You only run once: Spark auto-tuning from a single run. *IEEE Transactions on Network and Service Management 17*, 4 (2020), 2039–2051.

[128] QIU, H., BANERJEE, S. S., JHA, S., KALBARCZYK, Z. T., AND IYER, R. Preprocessed tracing data for popular microservice benchmarks, 2020.

[129] QIU, H., BANERJEE, S. S., JHA, S., KALBARCZYK, Z. T., AND IYER, R. K. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020).

[130] QUINLAN, J. R. Program for machine learning. *C4. 5* (1993).

[131] RAPIN, J., AND TEYTAUD, O. Nevergrad - A gradient-free optimization platform. https://GitHub.com/FacebookResearch/Nevergrad, 2018.

[132] RATTIHALLI, G., GOVINDARAJU, M., LU, H., AND TIWARI, D. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (2019), IEEE, pp. 33–40.

[133] ROLIA, J., AND SEVCIK, K. The method of layers. *IEEE Transactions on Software Engineering* (1995).

[134] ROSSI, F., CARDELLINI, V., LO PRESTI, F., AND NARDELLI, M. Dynamic multi-metric thresholds for scaling applications using reinforcement learning. *IEEE Transactions on Cloud Computing* (2022), 1–1.

[135] RZADCA, K., FINDEISEN, P., SWIDERSKI, J., ZYCH, P., BRONIEK, P., KUSMIEREK, J., NOWAK, P., STRACK, B., WITUSOWSKI, P., HAND, S., AND WILKES, J. Autopilot: Workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[136] SAHA, A., NATARAJAN, N., NETRAPALLI, P., AND JAIN, P. Optimal regret algorithm for pseudo-1d bandit convex optimization. In *International Conference on Machine Learning* (2021), PMLR, pp. 9255–9264.

[137] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), pp. 351–364.

[138] SEMKE, J., MAHDAVI, J., AND MATHIS, M. Automatic tcp buffer tuning. *SIGCOMM Comput. Commun. Rev. 28*, 4 (oct 1998), 315–323.

[139] SERAZZRI, G., CASALE, G., AND BERTOLI, M. Java modelling tools: an open source suite for queueing network modelling andworkload analysis. In *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)* (2006).

[140] SHAHRIARI, B., SWERSKY, K., WANG, Z., ADAMS, R. P., AND DE FREITAS, N. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE 104*, 1 (2015), 148–175.

[141] SHASHA, D. Lessons from wall street: Case studies in configuration, tuning, and distribution. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1997), SIGMOD '97, Association for Computing Machinery, p. 498–501.

[142] SHI, M., TANG, Y., ZHU, X., WILSON, D., AND LIU, J. Multi-class imbalanced graph convolutional network learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence* (2020).

[143] SILLA, C., AND FREITAS, A. A survey of hierarchical classification across different application domains. *Data Mining and Knowledge Discovery* (2011).

[144] SOLDANI, J., AND BROGI, A. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Comput. Surv.* (2022).

[145] SOMASHEKAR, G., DUTT, A., VADDAVALLI, R., VARANASI, S. B., AND GANDHI, A. B-meg: Bottlenecked-microservices extraction using graph neural networks. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2022), ICPE '22, Association for Computing Machinery, p. 7–11.

[146] SOMASHEKAR, G., AND GANDHI, A. *Towards Optimal Configuration of Microservices*. Association for Computing Machinery, New York, NY, USA, 2021, p. 7–14.

[147] SOMASHEKAR, G., SURESH, A., TYAGI, S., DHYANI, V., DONKADA, K., PRADHAN, A., AND GANDHI, A. Reducing the tail latency of microservices applications via optimal configuration tuning. In *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)* (Los Alamitos, CA, USA, sep 2022), IEEE Computer Society, pp. 111–120.

[148] SRIRAMAN, A., DHANOTIA, A., AND WENISCH, T. F. Softsku: Optimizing server architectures for microservice diversity @scale. In *Proceedings of the 46th International Symposium on Computer Architecture* (New York, NY, USA, 2019), ISCA '19, Association for Computing Machinery, p. 513–526.

[149] SRIRAMAN, A., AND WENISCH, T. F. Mtune: AutoTuned Threading for OLDI microservices. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (USA, 2018), OSDI 2018, USENIX Association, p. 177–194.

[150] SUN, F., HOFFMAN, J., VERMA, V., AND TANG, J. Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization. In *International Conference on Learning Representations* (2020).

[151] SURESH, A., AND GANDHI, A. Using variability as a guiding principle to reduce latency in web applications via os profiling. In *The World Wide Web Conference* (2019), WWW '19.

[152] SURESH, A., SOMASHEKAR, G., VARADARAJAN, A., KAKARLA, V. R., UPADHYAY, H., AND GANDHI, A. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)* (2020), pp. 1–10.

[153] SWAMINATHAN, A., KRISHNAMURTHY, A., AGARWAL, A., DUDIK, M., LANGFORD, J., JOSE, D., AND ZITOUNI, I. Off-policy evaluation for slate recommendation. *Advances in Neural Information Processing Systems 30* (2017).

[154] TAN, Y., NGUYEN, H., SHEN, Z., GU, X., VENKATRAMANI, C., AND RAJAN, D. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems* (2012).

[155] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic configuration management at facebook. In *Proceedings of the 25th symposium on operating systems principles* (2015), pp. 328–343.

[156] TANG, C., YU, K., VEERARAGHAVAN, K., KALDOR, J., MICHELSON, S., KOOBURAT, T., ANBUDURAI, A., CLARK, M., GOGIA, K., CHENG, L., CHRISTENSEN, B., GARTRELL, A., KHUTORNENKO, M., KULKARNI, S., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., VENKATESAN, V., AND ZHANG, P. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 787–803.

[157] TARVO, A., SWEENEY, P. F., MITCHELL, N., RAJAN, V., ARNOLD, M., AND BALDINI, I. Canaryadvisor: a statistical-based tool for canary testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015), pp. 418–422.

[158] TOLSON, B. A., AND SHOEMAKER, C. A. Dynamically dimensioned search algorithm for computationally efficient watershed model calibration. *Water Resources Research* (2007).

[159] VAN AKEN, D., PAVLO, A., GORDON, G. J., AND ZHANG, B. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, Association for Computing Machinery, p. 1009–1024.

[160] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), pp. 1–17.

[161] WAJAHAT, M., MASOOD, S., SAU, A., AND GANDHI, A. Lessons Learnt from Software Tuning of a Memcached-Backed, Multi-Tier, Web Cloud Application. In *Proceedings of the 8th International Green and Sustainable Computing Conference* (2017), IGSC '17.

[162] WANG, H., WU, Z., JIANG, H., HUANG, Y., WANG, J., KOPRU, S., AND XIE, T. Groot: An event-graph-based approach for root cause analysis in industrial settings. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (2022), ASE '21, IEEE Press, p. 419–429.

[163] WANG, Q., KANEMASA, Y., LI, J., JAYASINGHE, D., SHIMIZU, T., MATSUBARA, M., KAWABA, M., AND PU, C. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *2013 IEEE 33rd International Conference on Distributed Computing Systems* (2013).

[164] WANG, Q., KANEMASA, Y., LI, J., JAYASINGHE, D., SHIMIZU, T., MATSUBARA, M., KAWABA, M., AND PU, C. An experimental study of rapidly alternating bottlenecks in n-tier applications. In *2013 IEEE Sixth International Conference on Cloud Computing* (2013).

[165] WANG, Q., ZHANG, S., KANEMASA, Y., PU, C., PALANISAMY, B., HARADA, L., AND KAWABA, M. Optimizing n-tier application scalability in the cloud: A study of soft resource allocation. *ACM Trans. Model. Perform. Eval. Comput. Syst.* (2019).

[166] WANG, S., LI, C., HOFFMANN, H., LU, S., SENTOSA, W., AND KISTIJANTORO, A. I. Understanding and auto-adjusting performance-sensitive configurations. *SIGPLAN Not.* (2018).

[167] WASEEM, M., LIANG, P., SHAHIN, M., DI SALLE, A., AND MÃ¡RQUEZ, G. Design, monitoring, and testing of microservices systems: The practitionersâ perspective. *Journal of Systems and Software 182* (2021), 111061.

[168] WEIN, S., MALLONI, W., TOMÉ, A. M., FRANK, S. M., HENZE, G., WÜST, S., GREENLEE, M. W., AND LANG, E. W. A graph neural network framework for causal inference in brain networks, 2020.

[169] WONG, J., KWAN, A., JACOBSEN, H., AND MUTHUSAMY, V. HyScale: Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems (ICDCS)* (2019), pp. 80–90.

[170] WU, L., TORDSSON, J., BOGATINOVSKI, J., ELMROTH, E., AND KAO, O. Microdiag: Fine-grained performance diagnosis for microservice systems. In *2021 IEEE/ACM International Workshop on Cloud Intelligence* (2021).

[171] WU, L., TORDSSON, J., ELMROTH, E., AND KAO, O. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium* (2020).

[172] XIN, J., HWANG, K., AND YU, Z. Locat: Low-overhead online configuration autotuning of spark sql applications. In *Proceedings of the 2022 International Conference on Management of Data* (New York, NY, USA, 2022), SIGMOD '22, Association for Computing Machinery, p. 674–684.

[173] YE, Z., CHEN, P., AND YU, G. T-rank:a lightweight spectrum based fault localization approach for microservice systems. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)* (2021).

[174] ZEČEVIĆ, M., DHAMI, D. S., VELIČKOVIĆ, P., AND KERSTING, K. Relating graph neural networks to structural causal models, 2021.

[175] ZHANG, B., VAN AKEN, D., WANG, J., DAI, T., JIANG, S., LAO, J., SHENG, S., PAVLO, A., AND GORDON, G. J. A demonstration of the OtterTune automatic database management system tuning service. *Proceedings of the VLDB Endowment 11*, 12 (2018), 1910–1913.

[176] ZHANG, J., LIU, Y., ZHOU, K., LI, G., XIAO, Z., CHENG, B., XING, J., WANG, Y., CHENG, T., LIU, L., RAN, M., AND LI, Z. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 415–432.

[177] ZHANG, M., CUI, Z., NEUMANN, M., AND CHEN, Y. An end-to-end deep learning architecture for graph classification. *Proceedings of the AAAI Conference on Artificial Intelligence* (2018).

[178] ZHANG, Y., HUA, W., ZHOU, Z., SUH, E., AND DELIMITROU, C. Sinan: Data-Driven Resource Management for Interactive Microservices. In *Workshop on ML for Computer Architecture and Systems (MLArchSys)* (June 2020).

[179] ZHANG, Z., RAMANATHAN, M. K., RAJ, P., PARWAL, A., SHERWOOD, T., AND CHABBI, M. CRISP: Critical path analysis of Large-Scale microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 655–672.

[180] ZHOU, J., CUI, G., HU, S., ZHANG, Z., YANG, C., LIU, Z., WANG, L., LI, C., AND SUN, M. Graph neural networks: A review of methods and applications, 2021.

[181] ZHOU, X., PENG, X., XIE, T., SUN, J., XU, C., JI, C., AND ZHAO, W. Poster: Benchmarking microservice systems for software engineering research. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)* (2018).

[182] ZHU, Y., KRISHNAN, S., KARANASOS, K., TARTE, I., POWER, C., MODI, A., KUMAR, M., ZHANG, D., MUTHYALA, K., JURGENS, N., SAKALANAGA, S., DARBHA, S., IYER, M., AGARWAL, A., AND CURINO, C. Kea: Tuning an exabyte-scale data infrastructure. In *Proceedings of the 2021 International Conference on Management of Data* (2021), SIGMOD/PODS '21.

[183] ZHU, Y., LIU, J., GUO, M., BAO, Y., MA, W., LIU, Z., SONG, K., AND YANG, Y. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), pp. 338–350.