#### Kuber: Cost-Efficient Microservice Deployment Planner

by

Harshavardhan Kadiyala

B.Tech, Manipal University, India, 2015

#### A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

#### MASTER OF APPLIED SCIENCE

in

#### THE FACULTY OF GRADUATE AND POSTDOCTORAL

#### STUDIES

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

December 2021

© Harshavardhan Kadiyala, 2021

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Kuber: Cost-Efficient Microservice Deployment Planner

submitted by Harshavardhan Kadiyala in partial fulfillment of the requirements for

the degree of MASTER OF APPLIED SCIENCE

in Electrical and Computer Engineering

#### **Examining Committee:**

Julia Rubin, Assistant Professor, Electrical & Computer Engineering, UBC Supervisor

Mohammad Shahrad, Assistant Professor, Electrical & Computer Engineering, UBC Supervisory Committee Member

Sathish Gopalakrishnan, Associate Professor, Electrical & Computer Engineering, UBC Additional Examiner

# Abstract

The microservice-based architecture – a SOA-inspired principle of dividing backend systems into independently deployed components that communicate with each other using language-agnostic APIs – has gained increased popularity in industry. Realistic microservice-based applications contain hundreds of services deployed on a cloud. As cloud providers typically offer a variety of virtual machine (VM) types, each with its own hardware specification and cost, picking a proper cloud configuration for deploying all microservices in a way that satisfies performance targets while minimizing the deployment costs becomes challenging.

Existing work focuses on identifying the best VM types for recurrent (mostly high-performance computing) jobs economically. Yet, identifying the best VM type for the myriad of all possible service combinations and further identifying the optimal subset of combinations that minimizes deployment cost is an intractable problem for applications with a large number of services. To address this problem, we propose an approach, called KUBER, which utilizes a set of strategies to efficiently sample the necessary subset of service combinations and VM types to explore. Comparing KUBER with baseline approaches shows that KUBER is able to find the best deployment with the lowest search cost.

# Lay Summary

Microservice-based architecture is a method for developing complex applications as a set of loosely coupled components that communicate with each other over lightweight interfaces. Microservice-based applications are becoming increasingly popular in industry due to their many advantages, including faster development cycles and the ability to scale independently. To take full advantage of these benefits, application developers use public cloud resources, such as virtual machines (VM), to run microservices. As public cloud providers offer a variety of VM types, each with its hardware specification and cost, application developers face the problem of picking VM types to run their services. Complicating the problem, multiple microservices can be co-located in the same VM to decrease costs, which raises the number of options to run a microservice-based application exponentially. Our work presents KUBER – a tool to help the application developer identify the right VM types and microservice co-locations that are performant and cost-effective.

## Preface

This thesis presents a deployment planner for microservices and its comparison with baseline approaches. The presented work was conducted by myself in collaboration with Alberto Misail, an undergraduate student, with the guidance and mentorship of my advisor, Prof. Julia Rubin. The work presented in this thesis was accepted as a full paper in SANER'22:

• Harshavardhan Kadiyala, Alberto Misail, Julia Rubin. Kuber: Cost-Efficient Microservice Deployment Planner, IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022.

I was responsible for the problem formulation, approach design and implementation, and the evaluation of KUBER. Details concerning the contribution of each collaborator to the manuscript are listed below:

- Alberto Misail: implementing and optimizing the approach, and manuscript write-up.
- Prof. Julia Rubin: concept formulation, manuscript write-up and revision.

# **Table of Contents**

| Ab  | strac            | tiii                                  |  |  |  |  |  |  |  |
|-----|------------------|---------------------------------------|--|--|--|--|--|--|--|
| La  | y Sun            | nmary iv                              |  |  |  |  |  |  |  |
| Pro | eface            | · · · · · · · · · · · · · · · · · · · |  |  |  |  |  |  |  |
| Tal | ble of           | Contents                              |  |  |  |  |  |  |  |
| Lis | st of T          | Tables                                |  |  |  |  |  |  |  |
| Lis | st of F          | Tigures                               |  |  |  |  |  |  |  |
| Lis | List of Acronyms |                                       |  |  |  |  |  |  |  |
| Ac  | Acknowledgments  |                                       |  |  |  |  |  |  |  |
| De  | dicati           | ion                                   |  |  |  |  |  |  |  |
| 1   | Intro            | oduction                              |  |  |  |  |  |  |  |
|     | 1.1              | Overview                              |  |  |  |  |  |  |  |
|     | 1.2              | Insights                              |  |  |  |  |  |  |  |
|     | 1.3              | Contributions                         |  |  |  |  |  |  |  |
|     | 1.4              | Structure                             |  |  |  |  |  |  |  |
| 2   | Back             | ground                                |  |  |  |  |  |  |  |
|     | 2.1              | Microservice-based Applications       |  |  |  |  |  |  |  |
|     | 2.2              | Cloud Infrastructure                  |  |  |  |  |  |  |  |

| 3 | Арр  | roach                                       |
|---|------|---|
|   | 3.1  | Problem Statement                           |
|   | 3.2  | A First-Approximation Solution              |
|   | 3.3  | Kuber Solution         12                   |
|   |      | 3.3.1 Combination Optimizer                 |
|   |      | 3.3.2 Deployment Planner                    |
|   |      | 3.3.3 Execution Engine                      |
|   | 3.4  | Implementation                              |
| 4 | Eval | luation Setup                               |
|   | 4.1  | Research Questions                          |
|   | 4.2  | VM Types                                    |
|   | 4.3  | Subject Applications                        |
|   | 4.4  | Performance Targets                         |
|   | 4.5  | Runtime Environment                         |
|   | 4.6  | Baseline Approaches                         |
|   | 4.7  | Measures and Metrics                        |
| 5 | Resi | ılts  |
|   | 5.1  | RQ1 (Configuration Selection Strategies)    |
|   | 5.2  | RQ2 (VM Selection Strategies)               |
|   | 5.3  | RQ3 (Sampling vs. Prediction)               |
|   | 5.4  | Threats to Validity                         |
|   |      | 5.4.1 External Validity                     |
|   |      | 5.4.2 Internal Validity 34                  |
| 6 | Rela | nted Work                                   |
|   | 6.1  | Identifying Cost-effective VM Types         |
|   |      | 6.1.1 Black-box Prediction-based Approaches |
|   |      | 6.1.2 White-box Prediction-based Approaches |
|   |      | 6.1.3 Black-box Sampling-based Approaches   |
|   |      | 6.1.4 White-box Sampling-based Approaches   |
|   |      |   |
|   | 6.2  | Identifying Interference between Workloads  |

| 7  | Sum    | mary and Conclusion         | 40 |
|----|--------|-----------------------------|----|
|    | 7.1    | Limitations and Future work | 40 |
|    | 7.2    | Conclusion                  | 41 |
| Bi | bliogr | aphy                        | 42 |
| A  | Oth    | er papers                   | 49 |

# **List of Tables**

| Table 4.1 | VM Types  | 20 |
|-----------|---|----|
| Table 4.2 | Subject Applications  | 21 |
| Table 4.3 | Average CPU and Memory Utilizations for a Single Service                |    |
|           | Running in Isolation  | 23 |
| Table 5.1 | Execution Time of SF, $SF_1$ , $SF_2$ , $SF_3$ , and $KUBER$ (in Hours) | 29 |

# **List of Figures**

| Figure 1.1 | A demo application: Sock Shop [11]   | 2  |
|------------|--------------------------------------|----|
| Figure 3.1 | Weighed Independent Domination (WID) | 10 |
| Figure 3.2 | KUBER overview                       | 12 |
| Figure 5.1 | Search cost (a-d) comparison.        | 27 |
| Figure 5.2 | Execution time (a-d) comparison.     | 28 |

# **List of Acronyms**

- API Application Programming Interface
- AWS Amazon Web Services
- **CPU** Central Processing Unit
- EC2 Elastic Compute Cloud
- GPU Graphical Processing Unit
- ML Machine Learning
- **OS** Operating System
- RAM Random Access Memory
- SLA Service-level Agreement
- SOA Service-oriented Architecture
- TMDB The Movie Database
- VM Virtual Machine
- WID Weighted Independent Domination

# Acknowledgments

I would first like to express my deep gratitude to my supervisor, Prof. Julia Rubin, for her ever-patient guidance, constructive advice, and persistent encouragement throughout my time working towards this Master's degree. I am also very grateful to my committee members Prof. Mohammad Shahrad and Prof. Sathish Gopalakrishnan for carefully reviewing this thesis and providing insightful feedback. Lastly, I am very fortunate to have friends and colleagues in the ReSeSS Research Lab and outside, who took part in my journey towards this Master's degree. Thank you for all the support over the past years.

# Dedication

To my parents Sankaranarayana Kadiyala and Lakshmi Devi

### **Chapter 1**

# Introduction

#### 1.1 Overview

Microservice-based architecture is a SOA-inspired principle of building complex backend systems as a composition of small, loosely coupled components that communicate with each other using language-agnostic APIs [15]. This architectural principle is now becoming increasingly popular in industry due to its advantages, such as greater software development agility, elasticity, and a pay-per-consumption deployment model. Realistic microservice-based applications contain tens or even hundreds of services deployed on private and public clouds [16, 17]. As cloud providers typically offer a variety of virtual machine (VM) types, each with its own hardware specification and cost, picking a proper cloud configuration for deploying all microservices, in a way that satisfies performance targets while minimizing the deployment costs, becomes challenging [18].

Existing work focused on identifying the cheapest yet performant VM types for recurrent (mostly high-performance computing) jobs in an economical manner. This is typically done either by *prediction-based* approaches that estimate the execution time of a job on each target VM type based on pre-existing data, e.g., [47, 60, 63], or by *sample-based* approaches which perform run-time sampling of job execution on a carefully selected subset of VMs and extrapolating this data on the remaining VMs, e.g., [18, 39–41]. However, microservice-based applications bring additional complexity: it is not practical to explore the myriad of all



Figure 1.1: A demo application: Sock Shop [11].

possible service combinations and, even if the performance of all service combinations on all possible VMs is known, finding the optimal subset of combinations that minimizes deployment cost is an NP-hard problem by itself, intractable for applications with a large number of services.

Consider, for example, the Sock Shop microservice-based demo application [11] in Figure 1.1, which contains seven services: *Front-end*, *Cart*, *Catalogue*, *Shipping*, *Order*, *Payment*, and *User*. The developers might decide to deploy the application on Amazon EC2 [4]: a cloud infrastructure offering more than 300 different VM types with a variety of CPU, GPU, memory, disk, and network options [19]. In the description of each VM type, Amazon provides recommendations for how this type of a machine could be used. With only around 50 VM types that are recommended for microservice-based applications, the space of all possible deployment configurations is still very large, e.g., the services *Carts*, *Catalogue*, *Shipping* could be combined and placed together on one VM, placed individually on three different types.

Finding the right VM type for each service combination depends on whether the services have competing CPU / memory / network requirements and on the capabilities of the particular VMs. A straightforward solution to this problem is to test all service combinations on all VM types. However, exploring the space of all combinations is exponential in the number of services: even for an application with seven services, there are  $2^7 - 1 = 127$  combinations, which quickly grows to millions for an application with only 20 microservices. Thus, testing all possible combinations of services on all VM types, to check which placement meets the performance target with the smallest cost, is infeasible. To further complicate matters, even if the right VM type for each combination is identified, selecting the optimal subset of combinations is a non-trivial task by itself. In our example, services *Carts*, *Catalogue*, *Shipping* can be combined and placed together on a certain VM or can be further combined with services *Orders* and *Payment* and placed on a more expensive VM. In fact, given the performance of all service combinations on all VM types, deciding on the cheapest deployment option translates to the *weighted independent domination* (Weighted Independent Domination) problem, which is known to be NP-hard [24].

In this thesis, we propose a sample-based approach for addressing this problem, called KUBER, which relies on (a) a set of strategies for carefully selecting service combinations and VM types to sample and (b) a deployment mechanism to efficiently test the performance of the chosen combinations on a VM. We choose to follow a sample- rather than a prediction-based approach as our and others experience, e.g., [18], shows that prediction-based approaches fail to accurately capture the correlation between workloads and VM capacity. We also confirm this claim in our evaluation.

#### **1.2 Insights**

KUBER performs an efficient search in the space of possible combinations and VM types by relying on three main insights:

- The partial ordering of service combinations allow KUBER to exclude a large number of VM types that will not meet the performance targets. For example, if a service combination {*Shipping*, *Orders*, *User*} does not meet the performance target on a certain VM type, any superset of this combination, e.g., {*Shipping*, *Orders*, *User*, *Carts*}, will not meet the target on that VM type either. KUBER thus implements logic for keeping and propagating prior execution results to future combinations.
- 2. Executing a service combination of a particular VM type is only worthwhile if the obtained solution has the potential to decrease the overall deployment cost. For example, it is not worthwhile to check whether the service combination {*Shipping*, *Orders*, *User*} meets a performance target on a VM that

costs \$4 if it is already established that each individual service can work on a VM costing \$1 each. KUBER thus employs several strategies to efficiently narrow down the space of considered configurations.

3. Given the cheapest VM type for each service combination, the problem of finding the cheapest deployment can be translated to a well-known NP-hard problem in graphs: Weighted Independent Domination (WID). The nature of the translation and the specific structure of the graph that we used for that enabled scaling existing heuristic solutions to the WID problem [26] to microservice-based applications of realistic size and complexity.

We evaluated KUBER on four open-source benchmark microservice-based applications, comparing it with a baseline sample-based approaches which do not use combination/VM selection strategies and a prediction-based approach built on top of existing work. Our evaluation shows that KUBER outperforms the baseline approaches, finding the best deployment configuration faster and with the lowest search cost. Moreover, the differences between the approaches become more pronounced as the size of the applications grow.

#### **1.3** Contributions

This thesis makes the following contributions:

- 1. It formulates the problem of picking a proper cloud configuration for deploying microservice-based applications.
- 2. It proposes a sample-based approach for addressing this problem, implemented in a tool called KUBER. KUBER consists of a set of strategies for minimizing the number of runtime experiments, an efficient solution for collecting performance data at runtime, and a problem-domain-inspired approach for improving the scalability of an existing heuristic WID solution, so it can be applied to applications of realistic size and complexity.
- 3. It evaluates the effectiveness of KUBER on four case-studies, comparing it with a number of baseline approaches.

4. It makes our implementation and evaluation setup publicly available to facilitate replication and further research [3].

#### 1.4 Structure

The remainder of this thesis is structured as follows: Chapter 2 introduces the necessary background on microservice-based applications and their deployment on the cloud. We present our approach in Chapter 3, Chapter 4 outlines our evaluation methodology. We discuss our results and threats to validity in Chapter 5. Chapter 6 discusses the related work. Lastly, Chapter 7 describes limitations of our approach, future research directions, and conclusions of the thesis.

### **Chapter 2**

## Background

In this section, we provide a short overview of microservice-based application development and deployment.

#### 2.1 Microservice-based Applications

Microservice-based architectures are closely related to service-oriented architectures (SOA), which is a style of software design where services represent application components that communicate over a network [52]. Microservices aim at shortening the development lifecycle while improving the quality, availability, and scalability of applications at runtime. Cutting one big application into small independent pieces reinforces the component abstraction and makes it easier for the system to maintain clear boundaries between components: APIs specified in the service contract are the only channel for accessing the service. Developers can focus on small parts of an application, without the need to reason about complex dependencies and large code bases. Microservice-based applications also promote autonomous teams working on services that are organized around business capabilities and assume end-to-end responsibility for these capabilities, from development to production. Another major advantage of microservice-based architectures is independent deployment, which reduces the coordination effort needed to align on common application delivery cycles and also leads to independent scaling at runtime [62].

A microservice-based development style is often used for in latency-critical applications, such as user-facing websites, where decreased performance leads to decreased user satisfaction and loss of business [13]. In such cases, it is common for developers to specify performance targets as part of their applications' service-level agreement (SLA) – a commitment between a service provider and a client. Along with application-level performance targets, service providers also include service-level performance targets [10], e.g., a response time of less than 100 milliseconds, for each API of the service. Service-level performance targets help service owners to monitor performance violations and ease troubleshooting of application. Performance targets are usually evaluated on a p-th percentile (e.g., the 99th or 95th percentile) of all requests the service receives [43].

#### 2.2 Cloud Infrastructure

Cloud providers, such as Amazon AWS [19] and Microsoft Azure [20], offer customers compute resources running on the providers' physical infrastructure. Specifically, they provide a wide range of Virtual Machine (VM) types, which differ in the processor architecture (e.g., Intel vs. AMD, CPU vs. GPU) and size (e.g., 2 vs. 16 CPU cores). VM types are grouped into families; VMs in a family typically have the same underlying architecture and differ by their size. At the time of writing, AWS provides more than 300 VM types grouped into more than 40 families [19]; Azure provides more than 400 VM types grouped into more than 50 families [51].

To increase utilization and achieve better cost-efficiency, multiple VMs are typically hosted on the same physical machine and thus share CPU, caches, memory, storage, and networking devices. While cloud providers guarantee certain resources, such as CPU, memory capacity, and storage, by dedicating them to a particular VM, other resources are shared by VMs running on the same physical machine [5]. Increased load on a physical machine might cause *VM interference*, which results in performance degradation for applications running on these VMs [45, 46]. Moreover, developers have the option to co-locate multiple workloads/microservices on the same VM. OS-Level virtualization solutions, such as Docker containers [30], help enable co-location of microservices by providing fault and dependency isolation, thereby preventing failures in one service from propagating to others. As containers do not guarantee performance isolation between workloads, when services running on the same VM rely on a particular shared resource, they face *service-level interference* [44].

Unlike the case of VM-level interference, no guarantees on the performance of interfering services are available and it is up to the development team to decide which co-locations of microservices are desirable given the performance targets. If a shared resource is heavily utilized by co-located services, none of the service gets all the resources they need to meet the required performance target. For example, if services A and B each satisfy their performance target on a VM with 2 CPU cores and 4 GB of RAM, they might miss their respective performance targets if co-located on a VM with twice the CPU and RAM capacity – 4 CPU cores and 8 GB of RAM. This is because the service could be, say, cache-intensive and face interference when sharing the cache, which will drive their performance down. Our work aims to address the challenge of arriving at the most cost-effective co-location of services that satisfies their performance targets.

### **Chapter 3**

# Approach

We now discuss our approach for finding a desired deployment configuration for a microservice-based application.

#### **3.1** Problem Statement

We assume as input an application S with *n* services  $S_1, ..., S_n$ , where each service  $S_i$  has  $i_j$  APIs, denoted by  $S_i:A_1, ..., S_i:A_{i_j}$ . We say that each API  $S_i:A_j$  has a performance target (e.g., measured in terms of time to process a request); we denote the performance target of  $S_i:A_j$  by  $S_i:A_j^t$ .

We also assume as input a compute cluster  $\mathbb{VM}$  with *m* VM types  $VM_1$ , ...,  $VM_m$ , where each VM type has its own hardware specification and cost; we denote the cost for  $VM_i$  by  $VM_i^c$ . We say that a service combination  $\pi$ , formed by co-locating a subset of  $\mathbb{S}$  on the same VM, satisfies the performance target if the performance targets of all APIs of all services in  $\pi$  are satisfied on that VM.

Our goal is to find a deployment configuration  $\Lambda$  for  $\mathbb{S}$ , which maps service combinations of  $\mathbb{S}$  to target VM types, such that: (a) every  $S_i \in \mathbb{S}$  is part of exactly one service combination  $\pi$  in  $\Lambda$ ; (b) every service combination  $\pi$  in  $\Lambda$  is mapped to a VM on which the performance target of  $\pi$  is satisfied; and (c) there is no other configuration  $\Lambda'$  such that the total cost of all VM types in  $\Lambda'$  is lower than in  $\Lambda$ .

That is, we aim at finding the cheapest deployment configuration that can colocate multiple services on the same VM and that satisfies the performance targets of all services in S.



Figure 3.1: Weighed Independent Domination (WID).

#### **3.2** A First-Approximation Solution

Prior work [63], as well as our experiments in Chapter 5, show that there is no direct correlation between the cost and the performance of a service on a VM. That is, using a costlier VM type may not necessarily improve the performance of a service. The most obvious solution to the problem of finding cheapest deployment configuration is thus to first order all VM types in  $\mathbb{VM}$  by their cost and then run each service combination in  $\mathbb{S}$  on each VM type one by one, until the cheapest VM type for each service combination is found. We refer to this solution as *Sort and Find* (SF).

Once the best VM type for each service combination is determined, we need to identify a subset of combinations that satisfy the conditions above. More formally, given a mapping of service combinations to the cheapest VM type for which each service combination satisfies its performance target, we need to find the subset of combinations that includes each service once and only once and has the lowest possible deployment cost. Consider, for example, an application with only three services,  $S_1$ ,  $S_2$ , and  $S_3$ , which is deployed on a cluster with three VM types,  $VM_1$ ,  $VM_2$ ,  $VM_3$ . Let us assume that the cost of these VM types are 2, 3, and 10, respectively. There are seven possible service combinations:  $\{S_1\}, \{S_2\}, \{S_3\}, \{S_1, S_2\}, \{S_1, S_3\}, \{S_2, S_3\}$ , and  $\{S_1, S_2, S_3\}$ . For illustration purposes, Figure 3.1a shows, for each combination, the cost of the cheapest VM type for which the performance

target is satisfied. Multiple deployment options are possible for this example: each of the services could be deployed individually on different instances of  $VM_1$ ; the overall cost of this solution would be 6. A cheaper deployment would be to deploy the combination  $\{S_1, S_2\}$  on  $VM_2$  and  $\{S_3\}$  on  $VM_1$ ; the cost of this solution would be 5. A deployment that contains service combinations  $\{S_1, S_2\}$  and  $\{S_1, S_3\}$  would be invalid as  $S_1$  would be deployed more than once. Similarly, a deployment that contains service combinations  $\{S_1, S_2\}$  only would be invalid as  $S_3$  would not be deployed.

The problem of finding the cheapest valid deployment given the mapping from a service combination to its cheapest working VM type (like in Figure 3.1a) can be translated into the Weighed Independent Domination (WID) problem [24]. The input to WID is a weighted undirected graph G = (V, E), where nodes  $v \in V$  and edges  $e = (v, u) \in E$  have non-negative weights  $w(v) \ge 0$  and  $w(v, u) \ge 0$ , respectively. WID then finds a subset of nodes  $D \subseteq V$  which satisfy the following criteria: (1) *Independent*: no two nodes in *D* are adjacent. (2) *Dominant*: any node in *V* is either in *D* or adjacent to a node in *D*. (3) *Least Weight*: *D* minimizes the following cost function:  $f(D) = \sum_{u \in D} w(u) + \sum_{v \in V \setminus D} min\{w(v, u), \text{ for } u \in D \text{ and } (v, u) \in E\}$ , which is the sum of the weights of the nodes in *D* plus the sum of the weights of the minimum-weight edges connecting nodes in  $V \setminus D$  to nodes in *D*.

To rephrase the deployment detection problem as WID, we define V to be the set of all possible service combinations. We place an edge e between a pair of nodes in V iff their corresponding combinations have at least one service in common. We set the node weights to be the cost of the cheapest VM type on which the combinations meet their performance target. We do not use edge weights and thus set them all to 0. Figure 3.1b shows such graph for the mapping in Figure 3.1a.

A solution *D* produced by the WID algorithm (highlighted in grey in the example in Figure 3.1a) results in a cheapest valid deployment  $\Lambda$ : (1) By the *Independent* property, every  $S_i \in \mathbb{S}$  is part of at most one service combination in  $\Lambda$  because if a service is part of a combination that was chosen in *D*, no other service combination that contains the service is in *D*. (2) By the *Dominant* property, every  $S_i \in \mathbb{S}$  is part of at least one service combination in  $\Lambda$  because the service combination  $\{S_i\}$  is either in *D* or one of its adjacent nodes (that also contain  $S_i$ ) is in *D*. (3) By the *Least Weight* property, there is no other  $\Lambda'$  such that the total cost of all VM



Figure 3.2: KUBER overview.

types in  $\Lambda'$  is lower than in  $\Lambda$  because WID's cost function translates to the sum of the weights of all nodes in *D*, i.e., their deployment cost. Thus, minimizing this function means that no other valid deployment with lower costs exists.

As WID is an NP-hard problem [24], we rely on an iterative greedy algorithm by Davidson et al., which approximates the result and was shown to outperform existing work [26]. In a nutshell, the algorithm starts from greedily selecting an initial set of nodes in D and then iteratively improves the initial result by taking a random subset of nodes out of D (partial destruction phase) and greedily completing it to form a valid solution again (reconstruction phase).

Davidson et al. evaluated their approach on randomly generated graphs of varying sizes (between 100 and 1000 nodes). Yet, our graphs tend to be substantially larger (more than a million service combinations for an app with 20 services) and much more dense (as many nodes share common services); we thus modify and adapt this approach to our scenario, as discussed in Section 3.3.

#### **3.3 Kuber Solution**

An overview of KUBER, which further extends the approach outlined in the previous section, is shown in Figure 3.2. KUBER consists of two main parts. The first part, *Combination Optimizer*, improves the SF solution with a more efficient selection of service combinations to explore. The second part, *Execution Engine* takes care of service deployment and runtime data collection. We now discuss these two components in detail.

#### **3.3.1** Combination Optimizer

The SF solution performs a runtime experiment for every non-empty subset of S, i.e.,  $\mathbb{P}(S)$ -1 times. Such runtime experiments are costly, with respect to both time and budget. The reduce this cost, Combination Optimizer relies on a number of strategies, summarized in Algorithm 1.

It first initializes a set of variables: the map M, which keeps, for each service combination, the cheapest VM type where the performance target of the combination is satisfied (line 3);  $\Lambda$ , which keeps the best deployment configuration identified so far (line 4); and  $\Lambda^c$ , which keeps the cost of that deployment (line 5). The algorithm then computes the set of all non-empty service combinations of the input application  $\mathbb{S}$  and sorts them by the number of services in a combination, i.e., first the combinations with one service, then combinations with two services, etc. (line 6). It iterates over all combinations in order and, for each combination, explores all VM types in order (lines 7-30). Before collecting performance data for each combination  $\pi$  on a VM v, it checks that the following conditions hold:

Condition 1 (lines 10-12): If the cheapest working VM type for at least one subset of services  $\bar{\pi} \subset \pi$  is more expensive than v, it implies that the performance target of  $\bar{\pi}$  was not met on v. As adding more services to  $\bar{\pi}$  cannot improve the performance of services that are already in that set,  $\pi$  cannot meet its performance target on that VM type either, and this runtime experiment can be skipped altogether. For example, the algorithm will skip executing the service combination  $\{S_1, S_2, S_3\}$  on  $VM_i$  if a subset of services, say  $\{S_1, S_2\}$ , does not meet the performance target on that VM type.

Condition 2 (lines 13-17): If executing  $\pi$  cannot lead to a deployment that is cheaper than the current solution, executing the experiment is unnecessary and can be skipped as well. To estimate whether the experiment has a chance to improve the cost of the current solution, we conservatively assume that still unexplored combinations have a chance to meet their performance target on certain VM types. More specifically, for each still unexplored combination, we utilize our knowledge about best VM types selected for its subset combinations (if any) and optimistically assume that the target combination will work on the most expensive of those VM types (lines 33-40).

Like in the previous case, we leverage the idea to order all explored combinations by size, making sure smaller combinations are executed earlier and their performance data can be propagated to larger combinations. Moreover, we conservatively pick the cheapest possible VM type (or  $VM_1$  for combinations of one service) to ensure we do not skip any experiments that have a chance to lead to a better deployment placement in the future. For example, if  $S_1$  and  $S_2$  meet their performance targets on  $VM_2$  and  $VM_4$ , respectively, we optimistically assume that a still unexplored combination { $S_1$ ,  $S_2$ } will meet its performance target on  $VM_4$ .

We rely on the *Deployment Planner* component (lines 41-52) to decide whether an experiment is worthwhile to execute. It accepts as input a map M (from a combination to its best VM type) and an experiment of interest m; it calculates the deployment solution using our extended version of the WID algorithm (described below) or returns  $\emptyset$  if at least one of the services does not have any VM type mapped to it yet (lines 43-45). When m is given, the method ensures m is part of the produced solution (lines 46-48). Otherwise, it returns any solution for the given map of combinations (lines 49-51).

To decide whether to execute an experiment  $(\pi, v)$ , we pass to the *Deployment Planner* a map containing all previously explored and optimistically projected service combinations, as well as the experiment of interest (line 14). We only proceed to actually executing the experiment if placing  $\pi$  on v could indeed lead to a cheaper deployment that includes this placement. We continue to the next combination otherwise, as placing  $\pi$  on even a more expensive VM type cannot further improve the cost (lines 15-17).

If placing  $\pi$  on v has the potential to lead to a better solution, we proceed to executing the experiment and collecting real performance data (line 18). For combinations that satisfy the performance target on the given VM type, we update the combination to best VM type map (line 20) and then rely on the *Deployment Planner* again to calculate the best current solution and its cost (lines 21-22).

This time, we only pass *M* as the parameter as we are interested in the best possible realistic solution rather than a solution that contains  $(\pi, v)$  or that relies on predicted data.

```
1 Input: Application \mathbb{S} = \{S_1, ..., S_n\},\
    Cluster \mathbb{VM} = \{VM_1, ..., VM_m\} (ascending order by VM cost)
    Output: Deployment \Lambda
 2 begin
        M \leftarrow \emptyset
                                                                                                   ▷ A map of combination ~→ best VM type
 3
        \Lambda \leftarrow \emptyset
                                                                                                                                ▷ No solution yet
 4
 5
        \Lambda^{c} \leftarrow \infty
                                                                                                     > Upper bound for current solution cost
        \Pi \leftarrow \mathbb{P}(\mathbb{S}) \setminus \emptyset  > All non-empty combinations of services in \mathbb{S}, arranged by the number of services in a combination
  6
        while \pi \in \Pi do
 7
             \pi = popFirst(\Pi)
                                                                                             \triangleright Fetch and remove the first combination in \Pi
  8
            foreach v \in \mathbb{VM} do
  9
                 if \exists \bar{\pi} \subset \pi such that M(\bar{\pi}) = \bar{v} \wedge \bar{v}^c > v^c then
 10
                     \triangleright <u>Condition 1</u>: One of the subsets of \pi did not meet the performance target on \nu, hence \pi cannot meet the
                       performance target on v \Rightarrow proceed to the next VM type
 11
                     continue
                 end
 12
                 M' = OptimisticGuess(\Pi, M)  \triangleright Optimistically find the best possible VM type for unexplored combinations
 13
                 > Deployment under this assumption
                 \Lambda' \leftarrow DeploymentPlanner(M \cup M', (\pi, v))
 14
                 if cost(\Lambda') \ge \Lambda^c then
 15
                  break
                                       \triangleright <u>Condition 2</u>: Solution does not lead to a better deployment \Rightarrow explore next combination.
 16
 17
                 end
                                                                                                          > Collect runtime performance data
 18
                 execute(\pi, v)
                 if performance targets of \pi is satisfied on v then
 19
                     M[\pi] \leftarrow v
                                                                                                         \triangleright This is the cheapest VM type for \pi
 20
                     \Lambda \leftarrow DeploymentPlanner(M, \emptyset)
 21
                                                                                                                                     ▷ current best
                     \Lambda^c \leftarrow cost(\Lambda)
                                                                                                                                ⊳ current best cost
 22
                     \Lambda' \leftarrow DeploymentPlanner(M \cup M', \emptyset)
 23
                     if cost(\Lambda') \not\leq \Lambda^c then
 24
 25
                      return A
                                                                                               ▷ Condition 3: No better solution is possible
 26
                     end
                     \triangleright The cheapest VM type for \pi is found \Rightarrow explore next combination
                     break
 27
 28
                 end
            end
29
        end
30
31
       return \Lambda
32 end
33 Procedure OptimisticGuess (Π, M)
        begin
34
            M' \leftarrow \emptyset
 35
 36
            foreach \pi \in \Pi do
 37
                M'[\pi] \leftarrow the most expensive VM type of all subsets of \pi in M or VM<sub>1</sub> if non of the subsets is in M
            end
 38
            return M
 39
 40
        end
41 Procedure DeploymentPlanner (M, m = (\pi, v))
42
        begin
 43
             if \exists s \in \mathbb{S} such that M[\{s\}] = \emptyset then
                 \triangleright Some individual services were not explored yet \Rightarrow no solution
                return Ø
 44
 45
             end
            if m \neq \emptyset then
 46
             return WID solution for M \cup m which includes m
 47
 48
            end
 49
            else
             return WID solution for M
 50
51
            end
 52
        end
```

Algorithm 1: Combination Optimizer.

Condition 3 (lines 23-26): Finally, when a combination  $\pi$  can successfully run on a VM type v, the algorithm checks whether any further improvements are still possible. To this end, it uses the *Deployment Planner* again, this time passing it the map containing both executed and predicted combinations (line 23). If no solution that can improve the cost of current deployment (with or without the executed combination  $\pi$ ) is possible, the algorithm terminates and returns the current result (lines 24-26). Otherwise, it proceeds to exploring the next combination in order (line 27), as the cheapest VM type for this combination is already identified.

#### 3.3.2 Deployment Planner

As discussed in Section 3.2, we build up on the algorithm by Davidson et al. [26] for heuristically solving the WID problem. When computing a solution (in both initial and reconstruction phases), this algorithm iteratively and greedily chooses the next node to be one that has the highest ratio between the number of edges to remaining candidate nodes and the weight of the node. The rationale for this decision is to choose a dominant node (one that has a large number of edges) with a low weight. For the example in Figure 3.1b, the first node picked would be  $\{S_1, S_2\}$  as it has five edges to the neighbor nodes and the weight of 2, giving a ratio of 2.5 – larger than that of any other node. Then, the selected node and all its neighbors are removed from the set of possible candidates, to satisfy the *Independent* property. For the example in Figure 3.1b, that would remove all but the node  $\{S_3\}$ ; that node is selected next to complete the solution.

This algorithm does not scale well for inputs of our size. E.g., for apps with 20 services, the number of nodes would be more than a million and it will contain more than half a trillion edges; storing this information explicitly is not possible at this scale. Our main observation is that our graph has a very particular structure – its nodes are the service combinations and edges represent a partial order over the set of combinations. To choose the next node in every iteration, we mainly need to know the number of other candidate nodes a node is connected to. Moreover, when a particular node is selected, we only need to compute and remove from the set of future candidates all other nodes it is connected to. Using our knowledge about the graph structure, we adapt the algorithm by Davidson et al. [26] to compute this information on-demand, without explicitly storing the underlying graph, thus

improving the algorithm's scalability.

Assuming that a certain number of nodes has already been selected to be part of the solution, let *R* be a subset of services that have not yet been included in any of these nodes. Let *r* be the number of these services, i.e., r = |R|. The number of nodes remaining for selection is then  $2^r - 1$ . Let *v* be a candidate node; it can have edges to at most all still unselected nodes composed from services in *R* but itself:  $2^r - 1 - 1$ .

To calculate the exact number of neighbors of v, we consider the services it contains. Assuming there are r' such services, there are r - r' services in R that are not part of v and there are  $2^{r-r'} - 1$  nodes composed from these services. As two nodes have an edge only if they share at least one service, v has no edges connecting it to any of these nodes. As such, v has  $(2^r - 1 - 1) - (2^{r-r'} - 1) = 2^r - 2^{r-r'} - 1$  edges.

We use this formula to calculate the number of edges for each remaining candidate node and pick the one with the maximal ratio. For the example in Figure 3.1b, when the algorithm starts,  $R = \{S_1, S_2, S_3\}$  and r = 3. For the node  $\{S_1, S_2\}$ , r' = 2, thus, the number of neighbors is  $2^3 - 2^{3-2} - 1 = 5$ .

After a node is selected, we compute the remaining candidates by leveraging the fact that nodes are adjacent only if they share at least one service. Thus, the remaining candidates are nodes that do not share any service with the selected node v, i.e., the power set of all services in R minus the services in v. In our example, when  $\{S_1, S_2\}$  is selected, the remaining nodes are formed by all the combination of  $S_3$ , which is the combination  $\{S_3\}$  itself.

#### **3.3.3** Execution Engine

This component is responsible for performing the runtime experiments and collecting the performance data for each service combination  $\pi$  on a VM type v (line 18 in Algorithm 1). To accurately collect such data, we must deploy  $\pi$  on v in isolation. Yet, services in  $\pi$  interact with the rest of the system, i.e.,  $\mathbb{S} \setminus \pi$ . To ensure the performance of the services in  $\pi$  is not negatively affected by "lagging" services of the rest of the system, we deploy each remaining service in  $\mathbb{S} \setminus \pi$  in isolation, on a separate instance of the least expensive VM type ( $VM_1$ ). The *Controller* component in Figure 3.2 takes care of such deployment. We assume as input a set of tests that exercise the input application. To collect response times of APIs of the services in  $\pi$ , *Controller* executes these tests and, for each API of a service in  $\pi$ , captures the incoming request and the response times. Since the response time of an API depends not only on its own execution time but also on the response times of the outbound service it triggers, we measure and subtract the response times of such calls, as was also done in earlier work [35]. For the example in Figure 1.1, when measuring the response time of API of the Order service, we subtract from its execution time the response times of the outbound calls to the Payment and Shipping services.

The obtained execution time of each API in  $\pi$  is recorded in a centralized database (*Log* in Figure 3.2). *Deployment Planner* then reads this data to determine whether  $\pi$  meets the performance target for all APIs on *v*.

#### **3.4 Implementation**

To avoid VM interference, we use a private cluster with three physical machines. Two of the machines have an Intel Xeon E5-2640 v4 @ 2.40GHz processor with 40 cores, 128 GB of RAM, 25 MB cache, and 63 GB/s Memory bandwidth. The third machine has an Intel Xeon E5-2680 v4 @ 2.40GHz processor with 56 cores, 256 GB of RAM, 35 MB Cache, 76 GB/s Memory bandwidth.

We create and manage VMs using the OpenNebula cloud computing platform [9] deployed on a separate machine. We use Kubernetes cluster manager [8] and Istio monitoring system [6] to deploy and monitor microservices. We use Istio's logging functionality to store the execution time of each API in a time series database.

Finally, our implementation of the *Combination Optimizer* and *Execution Engine* components is written in Python and takes around 5000 lines of code. Our system implementation is publicly available to facilitate further research in this area [3].

### **Chapter 4**

# **Evaluation Setup**

#### 4.1 **Research Questions**

The goal of our evaluation is to answer the following research questions:

**RQ1 (Configuration Selection Strategies)**: How effective are configuration selection strategies applied by KUBER?

**RQ2 (VM Selection Strategies)**: How effective is the VM selection strategy applied by KUBER?

**RQ3 (Sampling vs. Prediction**): How effective is KUBER when compared with a baseline prediction-based approach?

We now discuss our experimental setup, including our selection of VM types, subject applications, and baseline approaches for comparison. To facilitate reproducibility, our experimental package is available online [3].

#### 4.2 VM Types

We used the three physical machines in our private cluster to simulate a number of VM types from Amazon EC2. Specifically, we choose three different families of VMs suggested for microservice-based applications: the basic A1 family, which provide cost savings for CPU-intensive workloads; the more expensive T3 family, which provides burstable general-purpose instances, thus increasing the price of

| VM Type                 | AWS VM      | CPU   | RAM  | US\$/Hour |
|-------------------------|-------------|-------|------|-----------|
|                         | Туре        | Cores | (GB) |           |
| $VM_1$                  | A1.medium   | 1     | 2    | 0.0255    |
| $VM_2$                  | M6g.medium  | 1     | 4    | 0.0385    |
| $VM_3$                  | A1.large    | 2     | 4    | 0.051     |
| $VM_4$                  | M6g.large   | 2     | 8    | 0.077     |
| VM <sub>5</sub>         | A1.xlarge   | 4     | 8    | 0.102     |
| $VM_6$                  | T3.micro    | 2     | 1    | 0.1104    |
| $VM_7$                  | T3.small    | 2     | 2    | 0.1208    |
| $VM_8$                  | M6g.xlarge  | 4     | 16   | 0.154     |
| $VM_9$                  | T3.large    | 2     | 8    | 0.1832    |
| <i>VM</i> <sub>10</sub> | A1.2xlarge  | 8     | 16   | 0.204     |
| <i>VM</i> <sub>11</sub> | M6g.2xlarge | 8     | 32   | 0.308     |

Table 4.1: VM Types

each VM type; and the higher-performance M6g family.

We picked four VM types from each family (12 VMs in total), starting from a VM type on which all services of our subject applications can boot and run individual. That excluded the smallest VM type from the T3 family: T3.nano with only 0.5 GB of RAM. We could not simultaneously simulate two of the selected VM types in our cluster: t3.medium and A1.large, because they both have 2 CPU cores and 4 GB of RAM. We thus excluded t3.medium from our analysis. The resulting 11 VM types, together with their mapping to the corresponding Amazon EC2 instance, the number of CPU cores, RAM size, and the cost per hour (as of January 2020) are given in Table 4.1. We deployed all the VMs corresponding to the same VM type family onto the same physical machine, allocating our largest physical machine (56 cores, 256 GB of RAM) to the M6g family and the remaining two machines (40 cores, 128 GB of RAM) to the T3 and A1 families. The obtained size and the capacity of our simulated cluster is similar to prior experiments of the same type [18].

#### 4.3 Subject Applications

We used a recent benchmark of microservice-based applications, called DeathStar-Bench [35]. It consists of three applications: Hotel Reservation, Media Service,

Table 4.2: Subject Applications

| Benchmark         | #Services | #APIs | Avg. #APIs/Service |
|-------------------|-----------|-------|--------------------|
| Hotel Reservation | 8         | 14    | 2                  |
| Media Service     | 11        | 29    | 3                  |
| Social Network    | 12        | 27    | 2                  |
| Sock Shop         | 7         | 42    | 6                  |

and Social Network. In addition, we used a popular open-source microservices demo application called Sock Shop [11]. We selected these applications because they are explicitly designed to represent real-world systems, are deployable onto a Kubernetes cluster, and include test suites allowing us to effectively trigger services/APIs.

Table 4.2 shows, for each application, the number of services it contains, the total number of APIs, and the average number of APIs per service. Overall, our applications contain between 7 and 12 services, with 14 to 42 APIs in total, and 2 to 6 APIs per service, on average. As the performance of an application varies based on the number of requests it receives (the API load provided by the test) and the volume of data stored in its associated database(s), we applied the following strategy to populate applications with realistic data.

For Hotel Reservation, which allows the users to obtain information and rates of nearby hotels, check hotels' availability during a given time period, make reservations, and also obtaining recommendations for hotels matching their selection criteria, we populate the hotel information database with real-world data from Yelp's Hotels Dataframe [14]. It contains 438 hotels and 172,159 hotel reviews. Similarly, for Media Service, which allows users to browse movie information, and then rent, stream, review, and rate movies, we use data from a real movies database, TMDB [12], which contains information about 5,000 movies and 5,000 casts.

Similar to Twitter, in the Social Network application, users can create posts embedded with text, media, and links, can tag users, and broadcast posts to their followers. The application uses three separate databases for persisting user profiles, posts, and media. We load the profiles database from existing social network data [56] with 962 users and 18,800 relations (representing followers). The volume of posts and media databases does not affect the performance of the application and we thus only use them for data generated at runtime.

Finally, for Sock Shop, an e-commerce application allowing users to browse and buy socks, we first searched for all socks sold by Amazon [2]. We learned that Amazon sells around 40,000 types of socks at the time of writing; we thus loaded the database with the same number of items.

Each of our subject applications contains a test suite provided by the developers, which simulates its typical usage scenario. For example, the test suite of Hotel Reservation simulates the scenario where the user logs in into the application, searches for a hotel, gets hotel recommendations, and reserves a hotel. We set the number of concurrent users served by each application to 165, as specified by DeathStarBench. We define a workload for an application as a set of API calls made by concurrent users under the test.

#### 4.4 Performance Targets

We use API execution time to represent API performance, with high performance translating to low execution time. To set the performance target for an API, we follow existing work that typically selects targets within a certain percentage of the best possible performance [28]. We thus assumed that the largest VM type (in all dimensions) has the best performance [67] and, without loss of generality, set the targets to be 50% of that performance. That is, we set the individual API performance targets to be twice their execution time on  $VM_{11}$ . Such selection ensures that performance target can be reached on some but not all VM types.

#### 4.5 **Runtime Environment**

Given 165 concurrent users and our database load, all user requests terminate within two minutes of execution on any VM type. We thus picked two minutes execution time for each test. Deploying and booting services on the right configuration of VMs takes another five minutes. We reset the VMs and repeat each experiment three times, to avoid performance variability due to underlying infrastructure. Thus, the total execution time of each experiment is 21 minutes. To avoid any performance bottlenecks, we make sure to deploy the test scripts and all external dependencies of each microservice, including databases.

| Benchmark         | CPU U  | tilization (%)          | Memory Utilization (%) |                         |  |  |  |
|-------------------|--------|-------------------------|------------------------|-------------------------|--|--|--|
| Deneminark        | $VM_1$ | <i>VM</i> <sub>11</sub> | $VM_1$                 | <i>VM</i> <sub>11</sub> |  |  |  |
| Social Network    | 37.7   | 3.1                     | 30                     | 2                       |  |  |  |
| Media Service     | 26.27  | 2.91                    | 29.73                  | 1.89                    |  |  |  |
| Hotel Reservation | 27.13  | 3.13                    | 28.5                   | 1.8                     |  |  |  |
| Sock Shop         | 33.86  | 4.57                    | 30.71                  | 1.93                    |  |  |  |

**Table 4.3:** Average CPU and Memory Utilizations for a Single Service Running in Isolation

As the WID algorithm is evolutionary, it requires a time limit to stop performing iterations. We experimented with the algorithm, running it on the largest set of combinations in our subjects set for 30 minutes. Our experiments showed that the best solution is achieved within one minute and minimal to no improvements are achieved afterwards. We thus set one minute as a time limit for the algorithm.

To make sure our VM selection and configuration are appropriate for the selected subject applications and their workload, we run each service on each VM type in isolation. Table 4.3 shows the average CPU and memory utilization on the smallest and largest VM types,  $VM_1$  and  $VM_{11}$ , respectively. As none of our VMs are overloaded by a single service, we believe our experiments are wellconstructed.

#### 4.6 **Baseline Approaches**

To answer **RQ1**, we implemented the basic SF approach described in Section 3.2. We then augmented it with each of the three conditions described in Section 3.3 one-by-one, producing three different implementations, which we refer to as  $SF_1$ ,  $SF_2$ , and  $SF_3$ . We compared these approaches with KUBER, which uses a combination of all three conditions simultaneously.

To answer **RQ2**, instead of ordering VM types by their cost, as SF does, we used two different strategies for selecting the next VM to explore. The first strategy selects the next VM at random. The second strategy performs a binary search [48] over the set of VM types ordered by cost (as our goal is to find the cheapest VM type that satisfies the performance target). When performing binary search, algorithm first selects the VM type that is in the middle of the list. If the service

combination of interest meets its performance target on this VM type, the search continues recursively to the first half of the list. Otherwise, it proceeds to the second half. For fair comparison with KUBER, we applied Conditions 1-3 for both strategies. We refer to the obtained solutions as RND<sub>123</sub> and BS<sub>123</sub>, respectively.

To answer **RQ3**, we implemented an approach that borrows and adapts ideas from a prominent prediction-based approach, PARIS [63], making it work in the microservices context. The goal of PARIS is to predict the performance of a service on a VM type. It does so by profiling a set of benchmarks that are assumed to be similar to the real applications of interest. For each benchmark, PARIS collects resource utilization (e.g., CPU usage) and performance information on all VM types, scaling it relatively to a few reference VM types (typically two). Then, to predict the performance of a service, PARIS collects features of the service by running it on the reference VM types and uses an ML-based model to predict performance on the remaining VM types based on the service similarity with the benchmarks.

To directly apply PARIS for predicting the best VM type for a service combination, we would need profiling information from all various combinations of benchmark services, which is untenable in our setting. We thus use individual benchmarks to predict the performance of combinations. To fairly evaluate the prediction properties of the approach, without relying on our ability to chose benchmarks similar to services in our dataset, we opted to use individual services themselves as benchmarks to train the model. That is, we run single services in isolation on each VM type and collect resource utilization and performance data. We use this data to train an ML-based model similar to the one used by PARIS. Then, to predict performance of a combination of services  $\pi$ , we execute  $\pi$  on only two VM types and use the model to predict the performance on the remaining VM types. To validate the prediction, we execute a runtime experiment on the cheapest VM where  $\pi$  is predicted to work and continue to the next predicted VM type, if the performance target is not met.

To further make sure we do not disadvantage this approach when compared with KUBER, we apply Conditions 1-3 to this approach as well. That is, we execute  $\pi$  only on VMs where all subset combinations were shown to work successfully, we only execute combinations that are expected to improve the deployment cost,

and stop the search when no further improvements are possible. We refer to the obtain approach as P (for prediction).

#### 4.7 Measures and Metrics

For each of the compared approaches, we calculate the cost of the deployment configuration  $\Lambda$  it finds. While AWS prices VMs per hour, microservice-based applications run for several days, months, or even years. Thus, without loss of generality, we calculate the cost of deployment per month. That is, when comparing the deployment cost found by each of the approaches, we multiply the hourly cost of each VM type in  $\Lambda$  by 24 hours and 30 days.

As the quality of the solution identified by each of the approaches improves as a function of the number of experiments it performs, we calculate the deployment cost identified by each approach as functions of: (1) the *search cost*, which represents the amount of money (in US dollars) spent in finding a solution and includes the cost of VMs used during the experiments, and (2) the *total execution time*, which represents the time (in hours) taken by an approach and includes the time of runtime experiments and WID execution.

### Chapter 5

# Results

Figures 5.1a-5.1d show, for each subject, the deployment cost achieved by each of the evaluated approaches as a function of the invested search cost. The baseline for the graphs, i.e., point x=0, is a deployment that places each individual service on the most expensive VM type. We do not depict this solution in the figure to avoid clutter, starting from the point where each approach found the cheapest working VM for each individual service. For example, for the Sock Shop application in Figure 5.1d, the cost of such deployment is \$312 and it takes \$2 to find this solution.

We mark with a cross the point on each graph where the corresponding approach terminates and we list the (search cost, deployment cost) values at this point, for clarity. E.g., for the Sock Shop application, KUBER terminated after spending \$6, identifying a deployment that costs \$238. SF<sub>1</sub> spends \$13 to find the same deployment, and SF<sub>3</sub> spends \$27. While SF found the same deployment after spending \$27, this approach continues to run and explore additional combinations. We stopped approaches which take substantially longer to terminate than others and do not show their termination points in the figure.

Figures 5.2a-5.2d show similar information: the deployment cost achieved by each approach as a function of its execution time. For example, Figure 5.2d shows that it took KUBER 26 hours to terminate with the \$238 solution while  $SF_1$  terminated after 38 hours.



Figure 5.1: Search cost (a-d) comparison.

#### 5.1 RQ1 (Configuration Selection Strategies)

All sort-and-find approaches evaluated in this research question perform exhaustive search over the space of combinations. Thus, given enough time and budget, they all arrive at the optimal solution. Yet, comparing KUBER with SF<sub>1</sub>, SF<sub>2</sub>, SF<sub>3</sub>, and SF shows that the combination of all conditions that KUBER applies is the most beneficial for finding lowest-cost deployment at minimal search cost and execution time: KUBER spends \$12 on average (min: 4, max: 24) and runs for 54 hours on average (min: 26, max: 103). In comparison, SF<sub>1</sub> spends \$64 on an average (min: 13, max: 140) and runs for 174 hours on average (min: 38, max: 357); SF<sub>2</sub> spends



Figure 5.2: Execution time (a-d) comparison.

more than \$94 on an average (min: 25, max: >150);  $SF_3$  spends more than \$57 on an average (min: 15, max >150); and SF spends more than \$144 on an average (min: 126 for the Sock Shop app, not shown in the figure to avoid clutter, max >150).

Table 5.1 shows time spent by each of the approaches, separately in each of the phases (setting up VMs for the experiments, executing the experiments, and running the WIP algorithm) and in total. While KUBER take 53 hours on average (the last column), the other three approaches execute for hundreds hours on an average. In fact, the total execution time of all the experiments is more than four

|    | Арр               | SF    |        | SF <sub>1</sub> |       |       | SF <sub>2</sub> |     |       | SF <sub>3</sub> |        |     |       | KUBER |        |     |       |       |        |     |       |
|----|-------------------|-------|--------|-----------------|-------|-------|-----------------|-----|-------|-----------------|--------|-----|-------|-------|--------|-----|-------|-------|--------|-----|-------|
|    |                   | Setup | Exper. | WID             | Total | Setup | Exper.          | WID | Total | Setup           | Exper. | WID | Total | Setup | Exper. | WID | Total | Setup | Exper. | WID | Total |
| 29 | Hotel Reservation | >395  | >158   | 0               | >553  | 55    | 22              | 0   | 77    | 150             | 60     | 16  | 226   | 55    | 22     | 2   | 79    | 15    | 6      | 7   | 28    |
|    | Media Service     | >372  | >149   | 0               | >521  | 150   | 60              | 0   | 210   | 412             | 165    | 36  | 613   | 90    | 36     | 2   | 128   | 30    | 12     | 15  | 57    |
|    | Social Network    | >362  | >145   | 0               | >507  | 245   | 98              | 0   | 343   | 322             | 129    | 26  | 477   | 357   | 143    | 5   | 505   | 58    | 23     | 22  | 103   |
|    | Sock Shop         | >265  | >106   | 0               | >371  | 20    | 8               | 0   | 28    | 130             | 52     | 7   | 189   | 60    | 24     | 1   | 85    | 10    | 4      | 12  | 26    |
|    | Average           | >348  | >139   | 0               | >487  | 117   | 47              | 0   | 164   | 253             | 101    | 21  | 375   | 140   | 56     | 2.5 | 198   | 28    | 11     | 14  | 53    |

Table 5.1: Execution Time of SF,  $SF_1$ ,  $SF_2$ ,  $SF_3$ , and KUBER (in Hours)

months.

The differences between the approaches are more pronounced as the size of the applications grows. For example, for Sock Shop, which is the smallest subject application with only seven services, the search cost of KUBER is 53% lower than that of its closest competitor,  $SF_1$ ; for Social Network, the largest application with 12 services, the difference is 82%. Similarly, the execution time of KUBER is lower than that of  $SF_1$  by 32% for Sock Shop and by 71% for Social Network.

Our experiments show that without any termination condition, SF continues executing experiments that do not improve the overall deployment cost, even if it arrives at the optimal solution, like in the case of Hotel Reservation, Media Service, and Sock Shop applications. SF<sub>3</sub> mitigates this issue by inducing a stopping condition (Condition 3 in Section 3.3) when no better solution is possible. In fact, for the Social Network application, the largest in our dataset, SF does not reach the desired solution within the allocated budget/time. Even though this application is larger than Media Service by only one service, it has double the number of combinations (4096 vs. 2048 combinations for Social Network and Media Service, respectively). Executing these extra combinations increases search cost and execution time.

 $SF_1$  is the only sort-and-find approach besides KUBER that reaches the optimal solution for Social Network, demonstrating that the pruning technique preventing  $SF_1$  from running combinations that are expected to fail (because their subset already failed on the same VM type: Condition 1 in Section 3.3) is the most effective strategy to reduce the number of unnecessary experiments.

While SF<sub>2</sub> (Condition 2 in Section 3.3) by itself performs worse than SF<sub>1</sub>, our experiments show that the combination of all conditions applied by KUBER help it converge on the desired solutions fastest and with the lowest search cost. That is because while Condition 1 eliminates a large number of lower-cost non-working VM types via propagation of negative results, Condition 2 eliminates a small number of experiments on very costly VM types. For example, the cheapest VM type for which the *Order* service in Sock Shop can meet its performance target is  $VM_{10}$ ; the compilation of *Payment* and *User* services can work on  $VM_1$ ; and the combination of all three services together does not meet the performance target on any of the given VM types. For that combination, SF<sub>1</sub> will not perform runtime

experiments for  $VM_1$ - $VM_9$  and will check  $VM_{10}$  and  $VM_{11}$ . While SF<sub>2</sub> will execute experiments on  $VM_1$ - $VM_{10}$ , it will determine that placing the combination of these three services on  $VM_{11}$  (which costs \$0.308 per hour) is more expensive than placing them on  $VM_1$  and  $VM_{10}$  separately (\$0.0255 + \$0.204 = \$0.2295) and will eliminate this experiment. The combination of these conditions is the most beneficial to improve the efficiency of the search. This observation also explains why SF<sub>1</sub> by itself typically terminates before SF<sub>2</sub> and with a lower search cost.

Interestingly, the search cost for  $SF_1$  is lower than that of  $SF_3$  for the Social Network and Sock Shop applications but is higher than SF<sub>3</sub> for Hotel Reservation and Media Service. That is because in Social Network and Sock Shop, there are a few highly interfering services. Placing them on the same VM would require an expensive VM type to ensure they meet their performance target. In fact, the optimal solution for both Social Network and Sock Shop involves a combination of three services placed on  $VM_{10}$ . SF<sub>1</sub> thus has an advantage due to its ability to skip executing many combinations of size three or more that contain pairs of these services, as such pairs are already known to interfere on VM types cheaper than  $VM_{10}$ . As a result, SF<sub>1</sub> reaches the optimal solution faster than SF<sub>3</sub>, which continues exploring such non-working combinations. On the contrary, in Hotel Reservation and Media Service, many service pairs works well on cheaper VM types but larger combinations require costlier VM types. For example, the optimal deployment for Hotel Reservation includes four pairs of services placed on  $VM_1$ and three instances of  $VM_3$ . Thus, SF<sub>3</sub> can quickly determine that additional experiments increase the cost of deployment and stop the execution while SF<sub>1</sub> will keep running these experiment.

Answer to RQ1: Conditions employed by KUBER allow it to arrive at optimal deployments with the minimal search cost and execution time for each of the subject applications. This is because its Condition 1 (SF<sub>1</sub>) helps eliminate many relatively cheap experiments, Condition 2 (SF<sub>2</sub>) helps eliminate a few relatively expensive experiments, and Condition 3 (SF 3) provides a global stopping condition. The savings achieved by KUBER increase as the number of services grows.

#### 5.2 RQ2 (VM Selection Strategies)

Comparing the performance of KUBER to that of  $RND_{123}$  and  $BS_{123}$  shows that KUBER always reaches the optimal solution with lower search cost and execution time. While  $RND_{123}$  is able to reach the optimal deployment for Hotel Reservation and Sock Shop, it executes more experiments than KUBER (thus a longer search time) and also executes more expensive experiments (thus a higher search cost). That is because it randomly attempts more expensive VM types that do not work, e.g., for a combination that does not meet its performance target on  $VM_6$ , which has 2 CPU cores and 1 GB of RAM, but does meet the target on  $VM_3$ , with 2 cores and 4 GB of RAM.

In case of Sock Shop, the technique also fails to terminate soon after fining the optimal solution. That is because the technique is "lucky" enough to find a solution while skipping lower-cost experiments for pairs of services, which leads to it trying a large number of combinations that include these pairs on all the skipped VM types. For both Media Service and Social Network, RND<sub>123</sub> misses the optimal deployment as it stops after finding working but sub-optimal VM type for many of the combinations (26 and 50, respectively).

 $BS_{123}$  finds more expensive deployment compared with KUBER for all four applications (31% increase, on average). This is because  $BS_{123}$  assumes performance is correlated with costs, i.e., that a more expensive VM type yields better performance. Like existing literature [18], we observe that not to be the case in practice. For example, several combinations that do not meet their performance target on  $VM_6$ , which has 2 CPU cores and 1 GB of RAM will could meet the target on  $VM_3$ , with 2 cores and 4 GB of RAM. Yet,  $BS_{123}$  will not try this configuration, proceeding to  $VM_9$  instead, which leads to a more expensive solution overall.

Answer to RQ2: The exhaustive search strategy applied by KUBER allows it to reach the optimal deployment for all subject applications.  $RND_{123}$  VM selection strategy does not explore the space of VM types systematically and thus could miss optimal choices.  $BS_{123}$  misses optimal choices as it relies on assumptions that do not hold in practice.

#### 5.3 RQ3 (Sampling vs. Prediction)

Comparing the performance of KUBER to that of P shows that P found a costlier solution for two out of four subjects: Hotel Reservation and Media Service. This is due to inaccuracies in predicting optimal VM type for combinations. For Media Service, P provided incorrect predictions for 20% of combinations it tried, resulting in (a) unnecessary executions and (b) missing some VM types that could have worked in practice. In fact, for combinations that were not predicted correctly, P made two wrong predictions on average, with only a third being a successful one. It also missed 14 correct placements and, as a result, missed the optimal deployment cost by around 20% (\$309 vs. \$257). For Hotel Reservation, the obtained solution was 11% more expensive than the optimal one found by KUBER (\$92 vs. \$83). In this case, P provided incorrect predictions for 7% of combinations it tried and made one wrong predictions on average.

In all four case studies, including Sock Shop and Social Network applications where P was able to identify the optimal solution, the search induced higher cost and longer execution time: P spent \$66 vs. \$12 for KUBER, on average (450% increase) and executed for 194 vs. 53 hours for KUBER, on average (266% increase). The increase in search cost is more substantial than in execution time because incorrect predictions lead the approach to skip less expensive and execute costlier experiments. For example, in Sock Shop, P incorrectly predicts the combination of the *Order* and *User* services not to meet its performance target on the cheaper VMs,  $VM_2$  and  $VM_3$ , and executes on a costlier  $VM_{10}$ .

Answer to RQ3: Prediction errors cause P to both execute unnecessary experiment and miss experiments that can lead to optimal deployments. As a result, KUBER is able to find a substantially less costly deployment for one of the subject applications. KUBER converges on a solution with lower execution time and search cost for all subject applications.

#### 5.4 Threats to Validity

We now discuss threats to validity of our approach:

#### 5.4.1 External Validity

Our results may be affected by the selection of applications that we used and may not generalize beyond our subjects. We attempted to mitigate this threat by using a set of benchmark applications provided by a highly cited related work on microservices. As we used applications of reasonable size and complexity, we believe our results are reliable. Moreover, our selection of performance targets could influence the selection of VM types on which subject applications can work successfully. We mitigated this threat by using the same criteria to calculate targets for all subject applications and all compared approaches. We also make our implementation and evaluation setup publicly available [3] to encourage validation and replication of our results.

#### 5.4.2 Internal Validity

Our implementation of KUBER, the WID algorithm, and our re-implementation of PARIS as part of building the P solution could have deficiencies. We controlled for the threat by having collaborators of this thesis reviewing KUBER code. Two collaborators of the thesis also manually and independently analyzed the obtained results, discussing their findings and any possible inconsistencies.

### **Chapter 6**

## **Related Work**

Existing work on decreasing cloud deployment costs can be divided into two main categories: identifying cost-effective VM types and identifying interference between workloads. We discuss these in Sections 6.1 and 6.2. Complementary to our approach are works that perform dynamic workload adaptations and dynamic pricing. We briefly outline them in Section 6.3.

#### 6.1 Identifying Cost-effective VM Types

The approaches in this category have the same goal as KUBER: to find the most cost-effective VM type where an application (task/job/service) satisfies its performance target.

#### 6.1.1 Black-box Prediction-based Approaches

Black-box prediction-based approaches [25, 47, 49, 63, 66], aim to infer performance of an application by assessing its similarity with previously profiled benchmarks. For example, AROMA [47] extracts resource consumption patterns (execution time and CPU, memory, network, and disk utilization) from a set of MapReduce benchmark jobs provided by Hadoop, runs them in a staging cluster of lowcapacity VMs using a reduced workload, and further clusters the jobs by the extracted patterns. To obtain a resource utilization signature of a new job, it runs a new job in the staging cluster, using only a fraction of input data. It uses the obtained signature to determine the similarity of the job with a particular cluster, and applies the cluster's trained Machine Learning model to predict performance of that job. PARIS [63], which was extensively discussed in Section 4, uses a prediction-based approach but, instead of running a job on a staging cluster with a smaller workload, runs it on a subset of VM types and infers its performance on other VM types.

#### 6.1.2 White-box Prediction-based Approaches

White-box prediction-based approaches assume certain properties for the applications to build an analytical model. Specifically, OptEx [58], relies on benchmarks containing domain-specific libraries and aims to infer the performance of a target job based on the similarity of the libraries used in that job and in the benchmarks.

All prediction-based approaches heavily rely on similarity of the profiled workloads with each other. As our evaluation shows, assuming such similarity can lead to erroneous predictions. Moreover, in the context of microservice-based applications, profiling various possible combinations of services becomes a challenging and expensive task by itself. Sampling-based approaches described below aim to address this problem.

#### 6.1.3 Black-box Sampling-based Approaches

Black-box sampling-based approaches [18, 23, 39] do not assume any application properties or similarity to existing benchmarks. Instead, they typically rely on VM similarity metrics (e.g., CPU cores, frequency, memory specifications, etc.) to predict the performance and execution time of a workload on a new VM type using data collected from already executed VMs. These approaches iteratively update their prediction models by selecting the next best VM type to sample. For example, CherryPick [18] uses an ML-based optimization technique to predict the VM type where the cost of running the job (price of the VM for the duration of the job execution) is minimized. It then samples this VM type, collects runtime data, and updates the ML model. Arrow [39] improves CherryPick by augmenting VM similarity metrics with additional information, such as I/O wait. Instead of finding the best VM type for a job, Micky [40] finds one VM type that may work

nearly best for multiple jobs, allowing the users to reduce the search cost at the expense of inevitably finding sub-optimal solutions. It uses a reinforcement learning technique to predict optimal VM type by balancing exploration (running dissimilar VM types) and exploitation (running similar VM types). Vanir [21] combines the benefits of both prediction and sampling by using a similarity threshold to decide between the approaches: it profiles a new job and scores it based on the similarity with previously executed jobs. If sufficiently similar, the job is passed to a prediction-based approach; otherwise, it is passed to a sampling-based approach. Once the new job is executed, Vanir updates its performance model for future optimization. Similarly, Scout [41] combines prediction- and sampling-based techniques. It uses performance predictions similar to that of PARIS [63] to improve CherryPick [18] by avoiding running experiments on VM types that are predicted not to work.

Our work largely falls into this category. Yet, while most black-box samplingbased approaches focus on predicting execution time given the specification of a job and a VM, we assess the performance through direct sampling, focusing on addressing an orthogonal scalability problem induced by a large number of service combinations.

#### 6.1.4 White-box Sampling-based Approaches

White-box sampling-based approaches [32, 55, 60, 61] assume certain application properties, e.g., that computation scales linearly with data. They mainly work by building an analytical performance model specific to an application, executing workloads in a carefully selected subset of VMs and estimating performance of the same workload for different configurations (e.g., on other VMs or for larger input data). For example, Ernest [60] builds a mathematical performance model of a job based on the behavior of a job on small samples of data and then predicts its performance on larger data and cluster size. Such approaches are not easily extendable beyond applications with fixed internal structure, e.g., Spark jobs, and thus have limited applicability for general-purpose applications, like microservices.

#### 6.2 Identifying Interference between Workloads

Interference occurs when multiple applications share common resources, such as CPU, cache, memory, storage, and network, which affects the performance of these applications. A few approaches, such as Bubble-Up [50] and Bubble-Flux [64], aim at identifying interference between jobs by creating a micro-benchmark (called a *bubble*) that can incrementally increase pressure on a particular resource. To measure maximum interference tolerated by an application, they co-locate the bubble with the application and increase pressure on the resource until the application faces performance issues. To further improve scalability and avoid exploring each resource individually, Paragon [28] and Quasar [29] first collect information about interference for a large set of benchmarks and then use the similarity between benchmarks and new jobs to make predictions. While such approaches can be further integrated with our work to improve performance, they rely on the assumption that a complete physical machine is available for testing and only the bubble or application can occupy the resource. Such assumption is not always practical for organizations that do not have access to a private cloud and have to perform the analysis on VMs of a cloud provider, which are co-located with other VMs.

#### 6.3 Other Approaches

An number of *autoscaling approaches* have been proposed in industry and academia. These approaches aim at provisioning resources to applications at runtime based on the dynamically changing workload. That is, given that the number of user requests fluctuates over time, these approaches increase or decrease the amount of resources allocated to the application, making sure it satisfies its performance target. In the simplest form, rule-based approaches employed in industry [1, 7] require the user to define a set of scaling-in and scaling-out conditions. The autoscaler then makes scaling decisions only when the conditions are met. For example, Amazon autoscaling service can add a number of VM instance, as specified by the user in the autoscaling policy, if CPU utilization reaches 70% or remove a number of instances if CPU utilization decreases below 40%. Such solutions induce extra burden on the user and also result in inefficient resource utilization due to coarse-grained utilization thresholds.

To address these issues, several approaches aim at automatically predicting the anticipated application workload [22, 31, 34, 37, 38, 42, 54, 65, 68]. Others focus on find the maximal load supported by each VM type, in order to assign the right VM types to the application [27, 33, 36, 53, 57, 59] and use this information to intelligently pick a VM type at runtime. The goal of KUBER is different. It does not focus on predicting workloads. Also, instead of fining the maximal load for *each* VM type, it terminates the search when the cheapest VM type on which a combination of services satisfy its performace target is identified. Yet, KUBER could be extended by ideas proposed in these approaches as part of future work, to find optimal deployment for dynamically changing workloads.

### **Chapter 7**

# **Summary and Conclusion**

#### 7.1 Limitations and Future work

We identify a number of limitations of our approach and directions for possible future work:

**1. Approaches for adapting to changing workload:** KUBER currently accepts as input an application test that can produce a fixed workload. In reality, the workload can fluctuate over time and optimal deployments produced under test scenarios may miss the performance target for a different workload. Autoscaling approaches, discussed in Chapter 6, address the workload fluctuation problem by dynamically provisioning resources to applications at runtime. Yet, re-running the KUBER from scratch for each new workload is impractical. Future work could look at approaches to identify the optimal deployment incrementally, during autoscaling, without performing the complete search for each new setup.

**2. Approaches to account for services shared between applications:** KUBER assumes that all services belong to one application and produces co-locations with services of that application. In reality, services can be shared between microservice applications; placing shared services as part of the optimal deployment of one application can negatively affect the performance of other applications. Future work adapting KUBER to such scenarios is needed.

**3.** Approaches for adapting to changing applications: As developers change their applications, e.g., to fix bugs and introduce new features, its resource consumption might change. Future work could look at approaches for assessing the

impact of changes in the application and incrementally adapting solutions produced by KUBER to work in new versions.

**4. Approaches to deal with services that are not part of combination under test:** When evaluating the performance of a particular service combination, our approach requires deploying all other services that are not part of the combination on separate machines. As part of future work, investigating other approaches for accurately evaluating the performance of a combination, including approaches that rely on mocking of other services, is needed.

**5.** Approaches for adapting current interference analysis tools to work on shared infrastructure: As discussed in Chapter 6, current interference analysis approaches [28, 29, 50, 64] rely on the assumption that a complete physical machine is available for testing and can be occupied by the application under test. Finding interfering pairs of services accurately on the cloud VMs could be another productive direction for possible future work. If successful, KUBER could be extended to use such techniques to further prune interfering combinations without running them.

**6.** Approaches to account for co-located VM interference in a public cloud: If KUBER is executed on a public cloud, performance fluctuations due to interference with co-located VMs could affect the accuracy of the obtained results. Future work is needed that can produce optimal deployments taking fluctuations in public clouds into account.

#### 7.2 Conclusion

As cloud providers typically offer a variety of virtual machine (VM) types, each with its own hardware specification and cost, and microservice-based applications contain multiple services that can be co-located on different VM types, selecting the cheapest VM types for deploying a microservice-based application becomes a time-consuming and costly task. This thesis formally defined the problem of identifying an optimal deployment for a microservice-based application and proposed a scalable solution that addresses this problem, implemented in a tool named KU-BER. We empirically evaluated KUBER on four open-source microservice-based applications and showed that it can identify the desired deployment faster and with lower search cost than existing alternatives.

# **Bibliography**

- [1] Amazon Auto-Scaling Service. https://aws.amazon.com/autoscaling/, .  $\rightarrow$  page 38
- [2] Amazon Socks. https://www.amazon.com/s?k=socks, .  $\rightarrow$  page 22
- [3] Supplementary Materials. https://resess.github.io/artifacts/Kuber.  $\rightarrow$  pages 5, 18, 19, 34
- [4] Amazon Elastic Compute Cloud. https://aws.amazon.com/ec2.  $\rightarrow$  page 2
- [5] Instance Performance Variability. https://forums.aws.amazon.com/thread.jspa?threadID=22830. → page 7
- [6] Istio Distributed Tracing. https://istio.io/latest/docs/tasks/observability/distributed-tracing/. → page 18
- [7] Kubernetes Horizontal Pod Autoscale. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, .  $\rightarrow$  page 38
- [8] Kubernetes. https://kubernetes.io/, .  $\rightarrow$  page 18
- [9] OpenNebula. https://opennebula.io/.  $\rightarrow$  page 18
- [10] Monitoring Microservices: Divide and Conquer. https://engineering.salesforce.com/ monitoring-microservices-divide-and-conquer-acca62b209cc. → page 7
- [11] A Microservices Demo Application: Sock Shop. https://microservices-demo.github.io/.  $\rightarrow$  pages x, 2, 21
- [12] TMDB 5000 Movie Dataset. https://www.kaggle.com/tmdb/tmdb-movie-metadata.  $\rightarrow$  page 21

- [13] The Psychology of Web Performance. https://blog.uptrends.com/ web-performance/the-psychology-of-web-performance/.  $\rightarrow$  page 7
- [14] The Yelp Dataset. https://hanlululu.github.io/SocialGraphYelp.io/Page1\_Dataset.html.  $\rightarrow$  page 21
- [15] Microservices: a Definition of This New Architectural Term. https://martinfowler.com/articles/microservices.html, 2014. → page 1
- [16] How Uber Monitors 4,000 Microservices. https://www.cncf.io/case-studies/uber/, 2019.  $\rightarrow$  page 1
- [17] Netflix Architecture: How Much Does Netflix's AWS Cost? https://www.cloudzero.com/blog/netflix-aws, 2021. → page 1
- [18] O. Alipourfard, H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In Proc. of the Symposium on Networked Systems Design and Implementation (NSDI), pages 469–482, 2017. → pages 1, 3, 20, 32, 36, 37
- [19] Amazon. Amazon EC2 Instance Types Amazon Web Services. https://aws.amazon.com/ec2/instance-types/, . → pages 2, 7
- [20] Amazon. Azure Virtual Machine Series. https://azure.microsoft.com/en-au/pricing/details/virtual-machines/series/, .  $\rightarrow$  page 7
- [21] M. Bilal, M. Canini, and R. Rodrigues. Finding the Right Cloud Configuration for Analytics Clusters. In *Proc. of the Symposium on Cloud Computing (SoCC)*, pages 208–222, 2020. → page 37
- [22] X. Bu, J. Rao, and C.-Z. Xu. Coordinated Self-configuration of Virtual Machines and Appliances Using a Model-free Learning Approach. *IEEE* transactions on parallel and distributed systems, 24(4):681–690, 2012. → page 39
- [23] M. Casimiro, D. Didona, P. Romano, L. Rodrigues, W. Zwaenepoel, and D. Garlan. Lynceus: Cost-efficient Tuning and Provisioning of Data Analytic Jobs. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS)*, pages 56–66, 2020. → page 36

- [24] S.-C. Chang, J.-J. Liu, and Y.-L. Wang. The Weighted Independent Domination Problem in Series-parallel Graphs. *Intelligent Systems and Applications*, 274:77–84, 2015. → pages 3, 11, 12
- [25] A. Chung, J. Park, and G. Ganger. Stratus: Cost-aware Container Scheduling in the Public Cloud. In *Proc. of the ACM Symposium on Cloud Computing (SoCC)*, pages 121–134, 2018. → page 35
- [26] P. P. Davidson, C. Blum, and J. A. Lozano. The weighted independent domination problem: Ilp model and algorithmic approaches. In *Proc. of the European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP)*, pages 201–214, 2017. → pages 4, 12, 16
- [27] J. Dejun, G. Pierre, and C.-H. Chi. Resource provisioning of web applications in heterogeneous clouds. In *Proc. of the conference on Web application development (WEBAPPS)*, pages 5–5, 2011. → page 39
- [28] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 77–88, 2013. → pages 22, 38, 41
- [29] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), page 127–144, 2014. → pages 38, 41
- [30] Docker. Docker. https://www.docker.com/.  $\rightarrow$  page 7
- [31] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck. From Data Center Resource Allocation to Control Theory and Back. In *Proc. of the International Conference on Cloud Computing (CLOUD)*, pages 410–417, 2010. → page 39
- [32] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proc. of the european conference on Computer Systems (EuroSys)*, pages 99–112, 2012. → page 37
- [33] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling web applications in heterogeneous cloud infrastructures. In *Proc. of the International Conference on Cloud Engineering (IC2E)*, pages 195–204, 2014. → page 39

- [34] A. Gambi, G. Toffetti, C. Pautasso, and M. Pezze. Kriging controllers for cloud applications. *IEEE Internet Computing*, 17(4):40–47, 2012. → page 39
- [35] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An Open-source Benchmark Suite for Microservices and their Hardware-software Implications for Cloud & Edge Systems. In Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 3–18, 2019. → pages 18, 20
- [36] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, Robust Capacity Management for Multi-tier Data Centers. ACM Transactions on Computer Systems (TOCS), 30(4):1–26, 2012. → page 39
- [37] M. Ghobaei-Arani, S. Jabbehdari, and M. A. Pourmina. An Autonomic Resource Provisioning Approach for Service-based Cloud Applications: A Hybrid Approach. *Future Generation Computer Systems*, 78:191–210, 2018. → page 39
- [38] D. Grimaldi, V. Persico, A. Pescapé, A. Salvi, and S. Santini. A Feedback-control Approach for Resource Management in Public Clouds. In *Proc. of the Global Communications Conference (GLOBECOM)*, pages 1–7, 2015. → page 39
- [39] C.-J. Hsu, V. Nair, V. W. Freeh, and T. Menzies. Arrow: Low-Level Augmented Bayesian Optimization for Finding the Best Cloud VM. In Proc. of the International Conference on Distributed Computing Systems (ICDCS), pages 660–670, 2018. → pages 1, 36
- [40] C.-J. Hsu, V. Nair, T. Menzies, and V. Freeh. Micky: A Cheaper Alternative for Selecting Cloud Instances. In *Proc. of International Conference on Cloud Computing (CLOUD)*, pages 409–416, 2018. → page 36
- [41] C.-J. Hsu, V. Nair, T. Menzies, and V. W. Freeh. Scout: An Experienced Guide to Find the Best Cloud Configuration. Technical report, 2018.  $\rightarrow$  pages 1, 37
- [42] W. Iqbal, M. N. Dailey, and D. Carrera. Unsupervised Learning of Dynamic Resource Provisioning Policies for Cloud-hosted Multitier Web Applications. *IEEE Systems Journal*, 10(4):1435–1446, 2015. → page 39

- [43] H. Jayathilaka, C. Krintz, and R. Wolski. Service-level Agreement Durability for Web Service Response Time. In Proc. of the International Conference on Cloud Computing Technology and Science (CloudCom), pages 331–338, 2015. → page 7
- [44] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan. A Holistic Evaluation of Docker Containers for Interfering Microservices. In *Proc. of the International Conference on Services Computing (SCC)*, pages 33–40, 2018. → page 8
- [45] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring Interference between Live Datacenter Applications. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2012. → page 7
- [46] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An Analysis of Performance Interference Effects in Virtual Environments. In Proc. of the International Symposium on Performance Analysis of Systems & Software (ISPASS), pages 200–209, 2007. → page 7
- [47] P. Lama and X. Zhou. Aroma: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud. In *Proc. of the International Conference on Autonomic Computing (ICAC)*, pages 63–72, 2012. → pages 1, 35
- [48] D. H. Lehmer. Teaching combinatorial tricks to a computer. In Proc. of Symposia in Applied Mathematics, pages 179–193, 1960. → page 23
- [49] X. Li, M. A. Salehi, M. Bayoumi, and R. Buyya. CVSS: A Cost-efficient and QoS-aware Video Streaming Using Cloud Services. In Proc. of the International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pages 106–115, 2016. → page 35
- [50] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proc. of the International Symposium on Microarchitecture* (*MICRO*), pages 248–259, 2011. → pages 38, 41
- [51] Microsoft. Azure Linux Virtual Machines Pricing. https://azure.microsoft.com/en-ca/pricing/details/virtual-machines/linux/. → page 7

- [52] Microsoft. Chapter 1: Service Oriented Architecture (SOA). https://web.archive.org/web/20160206132542/https: //msdn.microsoft.com/en-us/library/bb833022.aspx, 2016. → page 6
- [53] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proc. of the International Conference on Autonomic Computing (ICAC)*, pages 69–82, 2013. → page 39
- [54] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated Control of Multiple Virtualized Resources. In *Proc. of the European conference on Computer systems (EuroSys)*, pages 13–26, 2009. → page 39
- [55] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan. Perforator: Eloquent Performance Models for Resource Optimization. In *Proc. of the Symposium* on Cloud Computing (SoCC), pages 415–427, 2016. → page 37
- [56] R. A. Rossi and N. K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*, 2015. → page 21
- [57] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A Cost-aware Elasticity Provisioning System for the Cloud. In *Proc. of the International Conference* on *Distributed Computing Systems (ICDCS)*, pages 559–570, 2011. → page 39
- [58] S. Sidhanta, W. Golab, and S. Mukhopadhyay. Optex: A Deadline-aware Cost Optimization Model for Spark. In *Proc. of the International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 193–202, 2016. → page 36
- [59] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini. Dejavu: Accelerating Resource Allocation in Virtualized Environments. In Proc. of the International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 423–436, 2012. → page 39
- [60] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics. In Proc. of the Symposium on Networked Systems Design and Implementation (NSDI), pages 363–378, 2016. → pages 1, 37

- [61] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: Automatic Resource Inference and Allocation for Mapreduce Environments. In *Proc. of the International Conference on Autonomic Computing (ICAC)*, pages 235–244, 2011. → page 37
- [62] Y. Wang, H. Kadiyala, and J. Rubin. Promises and Challenges of Microservices: An Exploratory Study. *Empirical Software Engineering*, 26 (63), 2021. → page 6
- [63] N. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. Katz. Selecting the Best VM Across Multiple Public Clouds: A Data-driven Performance Modeling Approach. In *Proc. of the Symposium on Cloud Computing (SoCC)*, page 452–465, 2017. → pages 1, 10, 24, 35, 36, 37
- [64] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. ACM SIGARCH Computer Architecture News, 41(3):607–618, 2013. → pages 38, 41
- [65] L. Yazdanov and C. Fetzer. Vscaler: Autonomic Virtual Machine Scaling. In Proc. of the International Conference on Cloud Computing (CLOUD), pages 212–219, 2013. → page 39
- [66] N. Zacheilas and V. Kalogeraki. ChEsS: Cost-Effective Scheduling Across Multiple Heterogeneous Mapreduce Clusters. In Proc. of International Conference on Autonomic Computing (ICAC), pages 65–74, 2016. → page 35
- [67] L. Zhao, Y. Yang, K. Zhang, X. Zhou, T. Qiu, K. Li, and Y. Bao. Rhythm: Component-distinguishable Workload Deployment in Datacenters. In *Proc.* of the European Conference on Computer Systems (EuroSys), pages 1–17, 2020. → page 22
- [68] Q. Zhu and G. Agrawal. Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments. *IEEE Transactions on Services Computing*, 5(4):497–511, 2012. → page 39

### Appendix A

# **Other papers**

During my studies, I have also contributed to two other publications that are not included in this thesis:

- Yingying Wang, **Harshavardhan Kadiyala**, Julia Rubin. Promises and Challenges of Microservices: An Exploratory Study. Journal of Empirical Software Engineering, Volume 26, Issue 4, 2021.
- Adalberto R Sampaio, **Harshavardhan Kadiyala**, Bo Hu, John Steinbacher, Tony Erwin, Nelson Rosa, Ivan Beschastnikh, Julia Rubin. Supporting microservice evolution (short paper), IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017.