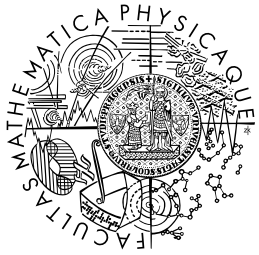


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Gábor Sándor

Performance assessment of cloud applications

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	prof. RNDr. Tomáš Bureš, PhD.
Study programme:	Software engineering
Study branch:	Software and Data Engineering

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my supervisor, prof. RNDr. Tomáš Bureš, PhD., for his help and valuable suggestions during the whole time. Furthermore, I would like to thank members of Department of Distributed and Dependable Systems for their advice and for providing the testing environment. Finally, I thank my family for their support.

Title: Performance assessment of cloud applications

Author: Bc. Gábor Sándor

Department: Department of Distributed and Dependable Systems

Supervisor: prof. RNDr. Tomáš Bureš, PhD., Department of Distributed and Dependable Systems

Abstract: Modern CPS and mobile applications like augmented reality or co-ordinated driving, etc. are envisioned to combine edge-cloud processing with real-time requirements. The real-time requirements however create a brand new challenge for cloud processing which has traditionally been best-effort. A key to guaranteeing real-time requirements is the understanding of how services sharing resources in the cloud interact on the performance level.

The objective of the thesis is to design a mechanism which helps to categorize cloud applications based on the type of their workload. This should result in specification of a model defining a set of applications which can be deployed on a single node, while guaranteeing a certain quality of the service. It should be also able to find the optimal node where the application could be deployed.

Keywords: edge-cloud; real-time requirements; resource sharing; performance optimization

Contents

1. Introduction

Over the past few years, the popularity of cloud-computing has massively increased. Hardware manufacturers have developed more advanced smartphones and embedded devices which introduce new possibilities for software developers. These include modern and complex Cyber-Physical Systems (CPS) such as augmented reality, autonomous driving, and smart-home, etc. where physical and software components are tightly intertwined. In such systems, the execution is not centralized i.e., does not happen on a single machine, however, it is distributed among multiple logical components, where each of them is executed on a particular device i.e., motion sensor, computation unit, actuator.

Most of the tasks in the domain of Internet of Things (IoT) are designed to be executed on resource-constrained devices. Nevertheless, there exist complex and resource-heavy tasks i.e., data processing and predictions, etc., that are more suitable for high-performance servers. This has led to change the software architecture, so that modern software systems are partitioned into two parts; server-side and client-side. The server-side application i.e., microservice run on the premises of a cloud-service provider acting as a back-end, while the client-side application represents a front-end which communicates with the microservice.

However, partially moving the computation into the cloud increases the global communication latency of the system. This contrasts with the concept of CPS, which imposes the need to operate and respond in real-time, requiring high data rates and ultra-low latency on the network side, and the guaranteed execution time on the microservice side. The former requirement can be fulfilled using a new concept, called *edge-cloud*. The idea is to move the cloud computing resources geographically closer to the end-users, eliminating not only the physical distance the data has to travel, but also the number of network devices e.g., switches and routers, etc. that increase the additional delay as well. This creates a layered architecture to balance the workload across all parts of the network beginning in large data centers and ending in edge data centers, close to the users, sometimes even at the Base Transceiver Station (BTS). Although, the concept of edge-cloud eliminates the network delay, a significant portion of response time is generated by the microservice. The current cloud technologies are optimized to easily autoscale and work on best-effort basis. Therefore, their primary aim is to provide optimal performance on average instead of guaranteeing worst-case execution time of individual requests. The problem of guaranteed execution time traditionally belongs to the domain of real-time programming, which relies on the usage of low-level programming languages and a limited set of libraries. However, this is in contrast with our goals, which target high-level programming languages e.g., Java, Scala, and Python, etc. and various workloads.

Thesis Objective

The current cloud technologies cannot ensure hard real-time guarantees, therefore, in this thesis, we target applications composed of multiple microservices that require soft real-time responses. In this frame, the general goal of thesis is to develop a new approach that provides soft real-time guarantees on the response time of microservices running in a container-based cloud e.g., Kubernetes, where

microservices are developed in high-level programming languages e.g., Java. In this context, statistical guarantee means that, for example, in 99% of cases, the response time will not be more than 40 ms and in 99.9% of cases, the request demand will be less than 70 ms.

Main Steps

The main concepts of cloud computing include resource sharing – one physical host machine provides its resources for multiple colocated applications. To reflect the changes in the service demand, the cloud orchestrator dynamically deploys new applications on the host machine or moves the deployed containers from one physical machine to another. The constantly changing set of colocated workloads may result in such an inappropriate combination of workloads, where the microservice response time cannot be guaranteed. In order to statistically guarantee the response time and to avoid the improper container placement, our approach combines the following main steps:

- **Microservice profiling:** To be able to guarantee the optimal behavior of the microservice in a shared production environment, it is required to assess the microservice in isolation and in predefined workload combinations. As the microservice response time is generally determined by the amount of hardware resources that can be dedicated for its execution, it is necessary to capture the relationships between the microservice resource usage and the response time. Therefore, we first analyze the attributes of the main system resources i.e., CPU, disk, and memory, in order to be able to characterize the microservice behavior by its resource usage pattern. Then, we design and implement a prototype of a deployment framework to automatically execute the microservice and to measure its resource usage both in isolation and in various workload combinations.

- **Predicting the response time:** Once the behavior of the microservice is known in predefined workload combinations, it can be used to predict its behavior in a shared production system. Therefore, we elaborate a method that builds a model from the measured resource usage pattern. Then, we use the model to predict the microservice response time, when it is colocated with other workloads in a single host machine. The result of the microservice characterization along with the predicted response time enables the cloud orchestrator e.g., Kubernetes to optimally schedule the container deployment to host machines. Furthermore, it prevents the overallocation of resources and enables to statistically guarantee the response time.

1.1 Structure of the Thesis

The second chapter focuses on the technical analysis of the problem. First, it introduces the problem using a real-life application. Then, it researches the available cloud technologies and analyzes if they are suitable to be used in the solution.

The third chapter is dedicated to the overview of the solution. It begins with the identification of requirements, then it introduces the concept of the proposed

deployment framework. Finally, the chapter states the detailed goals and outlines the basic structure of the solution.

The fourth chapter focuses on microservice resource usage assessment. First, it investigates the available system counters that enable to assess the utilization of various hardware resources i.e., CPU utilization, disk I/O rate, and memory usage. Then, it continues with the analysis of performance monitoring tools provided by the operating system and other third-party solutions.

The goal of the fifth chapter is to present the submission process executed by the deployment framework. The chapter divides the submission process into phases and describes each phase separately. First, it outlines the microservice submission phase and describes the requirements specification in more detail. Then, it focuses on the management of the microservice performance assessment phase that includes the distribution of the performance assessment tasks among the assessing agents and the management of these agents. Finally, the chapter presents the architecture of the performance assessment framework that automatically executes and assesses the submitted microservices.

The sixth chapter describes the implementation of artificial workloads for simulating real-life applications. It introduces and analyzes the existing solutions that can be transformed to artificial microservices and presents the custom microservices implemented by us. Furthermore, it presents the interface required for communication between the performance assessment framework and the microservice.

The seventh chapter introduces a prediction algorithm to predict the microservice response time in various workload combinations. In the first part, it describes how the data is collected and prepared for the prediction process. In the second part, it shows the steps of the prediction process that characterize the microservice, build a model using the collected data and predict the microservice response time. Finally, the chapter evaluates the prediction process.

The eight chapter focuses on the implementation details of the main components of the deployment framework. It serves as a programmer's documentation that includes UML class diagrams and the most important concepts for every component of the deployment framework.

The ninth chapter analyzes other works that are related to this problem and compares them with this thesis.

The tenth chapter concludes the thesis, and summarizes the achieved results.

2. Background

In this chapter, we provide a detailed technical background of our research. In the first section, we present our motivation using an example application. Then, we investigate the current cloud technologies, which could possibly be used in this research.

2.1 Running Example

To introduce how the final system will work, we use a simple augmented reality application. The application recognizes people in the captured video stream, draws a rectangle around the recognized faces, and displays the corresponding names.

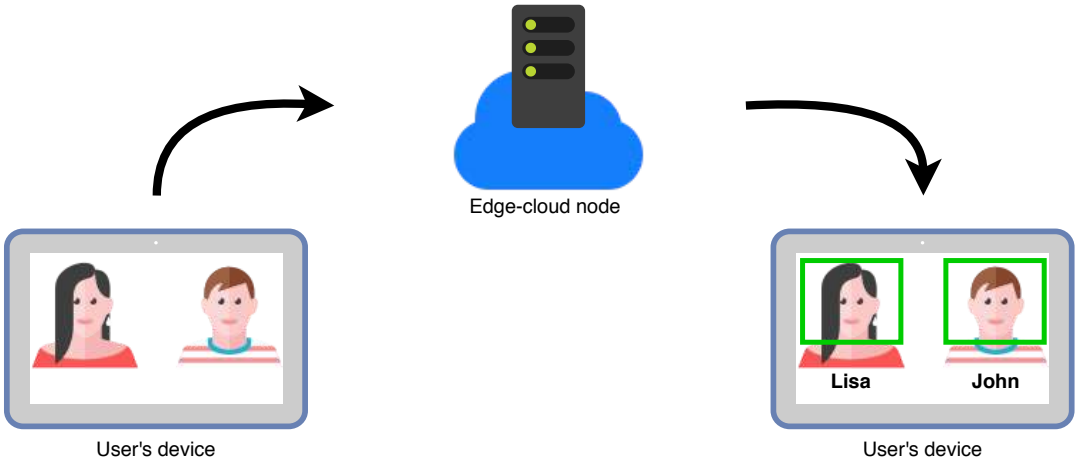


Figure 2.1: An augmented reality application example.

The example application is split into two parts: a client-side application running on the user's device i.e., smartphone or tablet, and a server-side application i.e., set of microservices deployed on a node in the edge-cloud. The client-side application captures video stream using the device's built-in camera and sends it to the server-side application. The server-side application processes the video stream and recognizes the faces using a previously prepared, trained database. The augmentation here is that the microservice returns the position and the size of the rectangles along with the names of the recognized people. To be able to run the distributed face recognition seamlessly, it is crucial to guarantee the communication latency between the client and the server. Ideally, the network should provide low latency and high throughput for high-quality video stream. Thus, the most suitable location of the microservice is an edge-cloud data center located close to the client. Furthermore, the deployment of microservice should be well-optimized in order to deliver the guaranteed response time even when colocated with a set of various microservices.

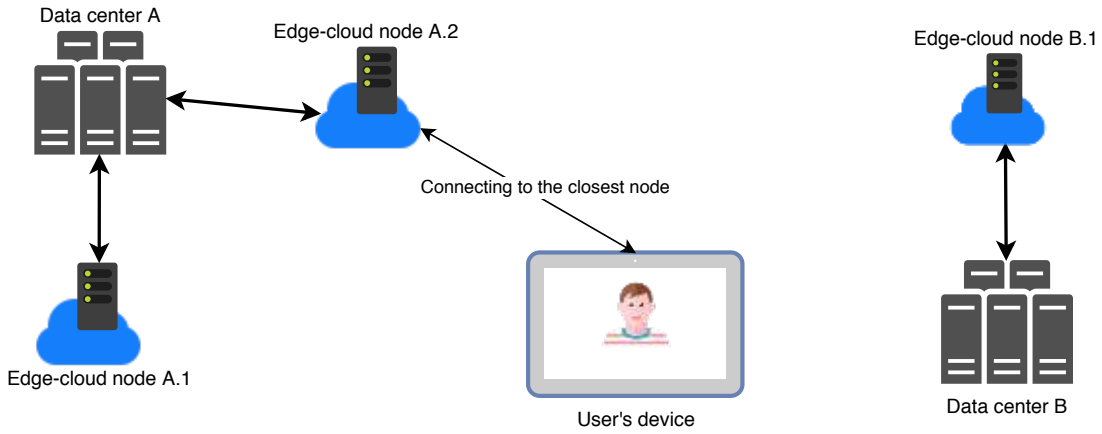


Figure 2.2: Example of selecting the closest edge-cloud node.

2.1.1 Application Model

To describe the application's response time requirements, we specify the application as a set of microservices. This design decision enables to assign a particular function to each microservice and thus it can be scaled independently from other parts of the application. We have designed a custom specification format, called *requirements specification*, which specifies the application and its microservices in detail (see Listing 2.1). The definitions of microservices include several test-points over which the timing-requirements are expressed. In addition, the test-points serve as special function allowing to assess the microservice's performance. Ideally, the test-points should be implemented in such a way, that they simulate the microservice's behavior. Thus, by assessing the behavior of a test-point, we can predict the microservice's behavior in real-life conditions. To do so, all test-points should be measured both in isolation and in combination with other workloads. The relationship between the test-points and the microservice ensures the software developer that the cloud guarantees the microservice's response time if the test-points' timing requirements are met. Thanks to the concept of test-points, our approach can automatically assess the microservices while still treating them as black-boxes.

2.1.2 Requirements Specification

Listing 2.1 shows an example requirements specification for the previously presented face recognition application. First, it defines the application's name and description. They are followed by the definition of microservices and their corresponding timing requirements. Each microservice has its own unique name and an entry-point. The entry-point is used by the assessing framework to run the microservice and assess its behavior. Next, the requirements are defined, which consist of definition of the required response time and its required probability (keys `time` and `probability`, respectively). The former defines the maximum response time in milliseconds, the latter defines the required probability in percentage. In our case, the `face_recognition` microservice requires maximum 50 ms response time in 99.9% of requests and maximum 30 ms response time in 95% of requests.

```yaml
application: face_recogn
description: Face recognition app
microservices:
  - name: video_processing
    entry_point: com.facerecognition.VideoProcessing
    requirements:
      - time: 100 # 100ms response time
        probability: 99 # 99% probability
      - time: 70 # 70ms response time
        probability: 95 # 95% probability
  - name: face_recognition
    entry_point: com.facerecognition.FaceRecognition
    requirements:
      - time: 50 # 50ms response time
        probability: 99.9 # 99.9% probability
      - time: 30 # 30ms response time
        probability: 95 # 95% probability
```

Listing 2.1: Example requirements specification.

2.2 Technical Background

In this section, we present and analyze cloud technologies that will be used throughout the thesis.

First, we describe the available cloud technologies that are designed for hosting various applications. Ideally, the selected cloud technologies should provide high-performance hardware infrastructure along with provisioning services for the deployed applications.

Next, we analyze the technologies that enable to encapsulate the applications. The advantages of encapsulation should result in application isolation, resource management, and metering.

Finally, we investigate the possibilities of orchestration. The desired orchestration technology should be able to manage the application deployment according to the specified resource and response time requirements.

2.2.1 Cloud Computing

Cloud computing is a model for hosting and delivering services over the Internet. In the recent years, it has become a popular choice among the business owners to deploy their services into the cloud. The main reasons of the constantly increasing popularity are the following: cloud computing eliminates the need of large upfront investments and provides adaptive provisioning regarding to the current demand. These two factors enable the business owners to start from small and scale the allocated resources accordingly to the ongoing service demand.

The architecture of cloud computing is divided into four layers: *hardware layer*, *infrastructure layer*, *platform layer* and *application layer* [1] (see Figure 2.3). The layers are loosely-coupled, which results in a modular architecture where the layers can be developed separately.

The *hardware layer* represents the data center's hardware infrastructure. Its main task is to ensure that a sufficient amount of physical resources i.e., computation power, system memory, disk storage, and network bandwidth are available

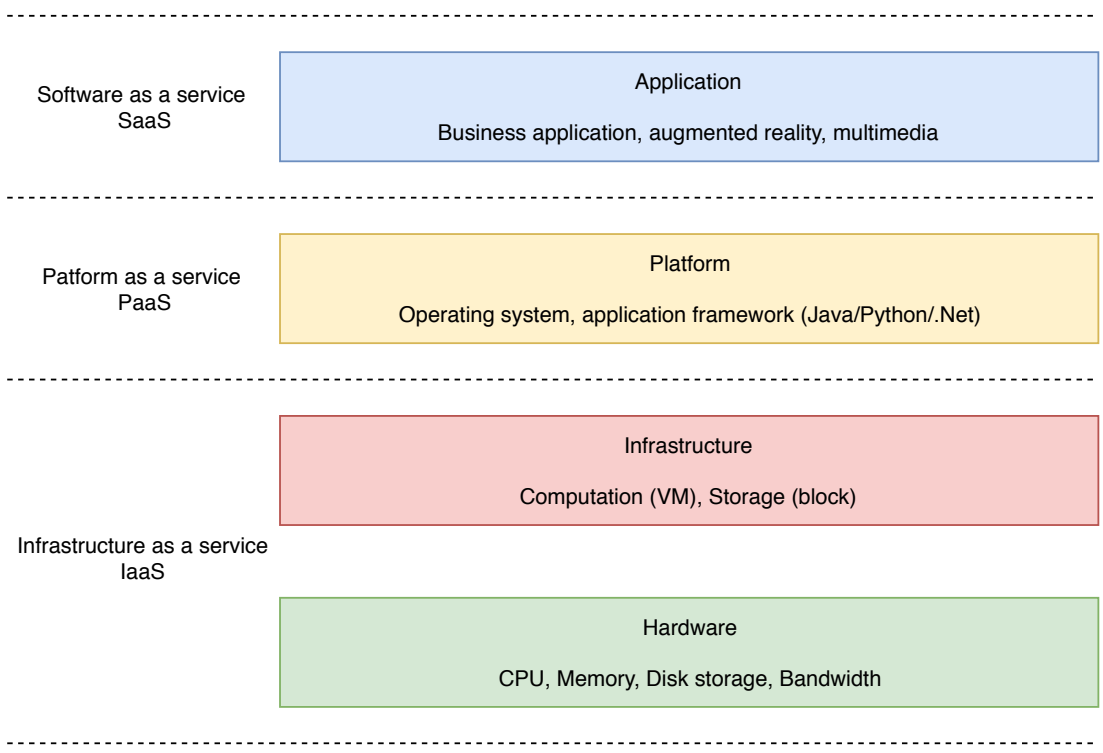


Figure 2.3: Layered architecture cloud computing [1].

for the upper layers. The large hardware infrastructure combined with the properties of the upper layers make the cloud computing suitable for effective batch processing e.g., big data, scientific computations. Moreover, it guarantees fast scalability, which ensures that the applications provide optimal performance even in peek hours. Lastly, the layer provides warranty for a certain level of availability (defined in SLA[1]) as the failed resources can be easily substituted.

The *infrastructure layer* is responsible for partitioning the physical hardware into virtualized pools of resources. Using the virtualization technologies (VMware [3] or Xen [4]), the layer enables to dynamically allocate and deallocate computing resources according to the current service demand.

The purpose of the *platform layer* is to create an environment where the applications can be deployed. This environment includes the operating system and the application framework, which minimize the application developer's effort to deploy the application. Furthermore, the layer enables to run multiple colocated applications on the same virtual node.

The *application layer* is the most visible layer of the cloud. It contains the cloud applications and exposes them to end-users by a well-defined interface e.g., web-page, REST API [5], etc. Furthermore, the layer enables the applications to exploit the cloud's auto-scale feature, thus the applications can provide the best performance in various load conditions.

The business model of cloud computing is built upon the previously described layers. The services can be grouped into the following three categories: *infrastructure as a service (IaaS)*, *platform as a service (PaaS)*, and *software as a service (SaaS)*. Figure 2.3 depicts the relation between the architectural layers and the provided services. Even though, the majority of current mobile applica-

[1]Service Level Agreement [2]

tions e.g., e-mail clients or social media apps already utilize the cloud as SaaS, they do not require guaranteed execution time. This is due to the fact that the applications are not part of the computation loop as they are used as a thin client, implementing only the presentation layer.

On the basis of the foregoing, cloud computing provides a solid environment where the applications can be deployed. The centralized, high-performance infrastructure alongside with the range of advanced services e.g., dynamic resource provisioning makes it capable to handle the most diverse and resource intensive tasks, which could not be executed on end-user devices.

However, there exist use-cases, where this concept is not suitable. For instance, in our case it is essential to guarantee the performance of individual requests. Since cloud-computing focuses on providing high throughput and best average performance, therefore it is unable to guarantee the microservice's execution time. Another important factor is the network latency. Based on the experiment of Ang Li et al [6], the average round trip time to the instances deployed on a cloud provider from 260 global vantage points is 74 ms. This response time does not include neither the microservice's execution time nor the latency between the end-user device and the wireless first hop.

2.2.2 Edge-cloud

As we described in the previous section, cloud-computing is built upon a centralized architecture, which concentrates on the computational capacity and data storage into data centers. However, there is an increasing trend of computation-intensive applications for mobile computing e.g., speech recognition, face recognition, and augmented reality, etc., and IoT technologies e.g., smart home, vehicle-to-vehicle communication, which require high data rates and ultra-low network latency. In order to satisfy the newly arisen requirements, the computation capabilities and the data storages must be placed closer to end-user devices.

The concept of edge-cloud is designed to address this problem in such a way, that is installs cloudlets [7] to the edge of the network i.e., WAN, MAN, etc. The cloudlets are micro data centers, which provide the same services i.e., resource pooling, scalability, multi-tenancy, and service provisioning, etc., as huge data centers do. Since, they are located close to the end-users (within one wireless hop), they ensure an ideal environment for hosting highly responsive cloud services.

In cloud computing, the end-user devices are directly connected to the data center, thus creating a two-layered architecture. However, the edge-cloud extends this architecture by inserting a new layer between the data center and the end users (see Figure 2.4). This layer is created by the edge-cloud nodes, and its purpose is to host applications, which require ultra-low network latency. Therefore, the mobile devices do not connect directly to the data centers, but to the edge-cloud nodes. In the new architecture, the role of the data center is to host non-time-critical applications and to provide supporting services e.g., data backup for edge-cloud nodes.

In cloud computing, the mobile application serves as a thin client, which only displays the output of the computation. However, in edge-cloud, the application is an essential part of the computation loop. First, it collects data about the environment using the end-user device's built-in sensors i.e., location sen-

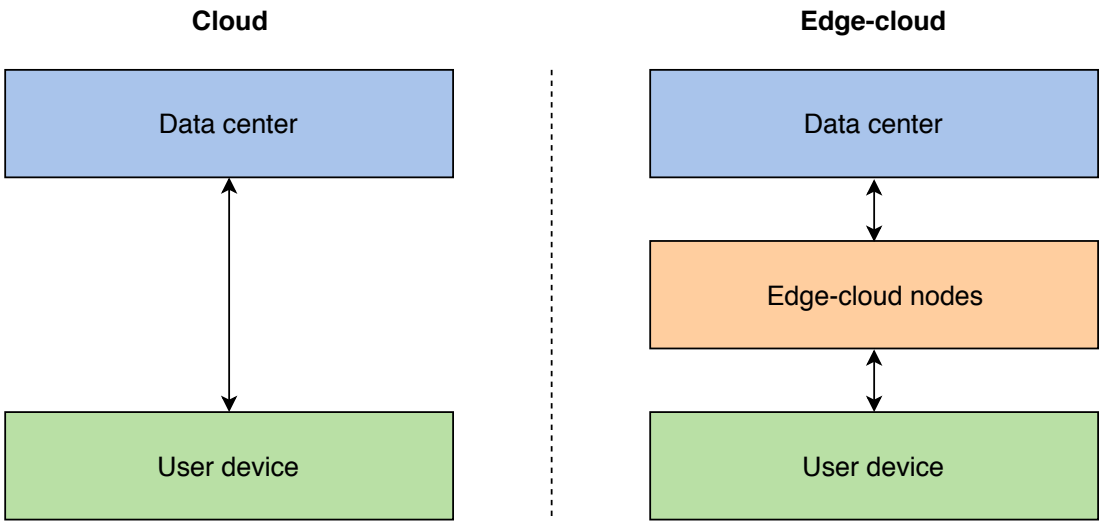


Figure 2.4: Comparison of architectural layers in cloud and edge-cloud.

sor, gyroscope, camera. Next, the application sends the collected data to the cloudlet for processing. Lastly, the cloudlet returns the computation result and the application displays it to the user. The ultra-low network latency provided by the edge-cloud technology makes it possible to implement a computation loop, where the resource-heavy sections are off-loaded into the cloud, and the result is displayed by the mobile application.

Ha et al. [8] carried out an experiment, which compared how the cloud data center's location affects the behavior of various computation intensive applications. The experiment included the following six locations, where the applications were executed:

- Amazon data centers using EC2 instances in Virginia, Oregon, Ireland and Singapore (labeled as *east*, *west*, *eu* and *asia*).
- Local data center using an obsoleted server machine (labeled as *1WiFi*).
- Local mobile device (labeled as *mobile*).

The tested applications include face recognition, speech recognition, object and pose identification, mobile augmented reality and physical simulation and rendering. Because the applications implement different algorithms, there exists cases e.g., speech recognition, which are more resource demanding than others. In order to compensate the lack of computation power on both local machines, it is reasonable to compare those applications, which are less resource demanding. Therefore, based on their description, we have selected face recognition and augmented reality applications. Figure 2.5 and Figure 2.6 show the cumulative distribution function (CDF) of response times in milliseconds. In case of face recognition application, we can see, that only the local server machine (1WiFi) provides fast enough response time (200 ms - 300 ms) in most of the cases. The augmented reality application behaves similarly. The local server machine performs the best, providing less then 100 ms response time in approximately 80% of cases. As the figures show, the increase of the response time corresponds to the distance between the end-user device and the data center.

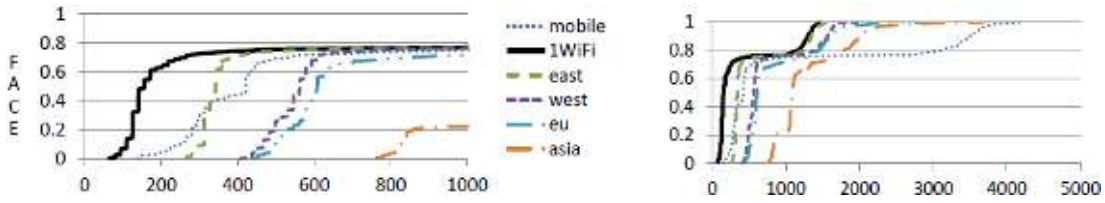


Figure 2.5: CDF of response times in milliseconds for a face recognition application processing 300 images. Source [8].

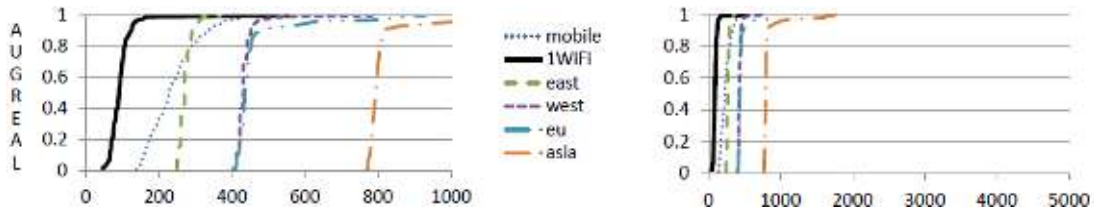


Figure 2.6: CDF of response times in milliseconds for an augmented reality application processing 100 images. Source [8].

Moving computation resources closer to the end-user devices results in decreased response time, latency and jitter. This improved network quality makes possible to implement mobile applications, which use external resources to augment the mobile devices' poor computational power. The concept of edge-cloud alongside the future 5G mobile network provide the best solution for ensuring the ultra-low first-hop latency. Furthermore, the inherent properties of cloud computing i.e., resource pooling, scalability, and multi-tenancy, enable to create an environment, which can guarantee the application's execution time.

2.2.3 Container-based Virtualization

Based on the previous description, we can state that the virtualization is a key concept of cloud computing. The goal of the virtualization is to provide abstraction of multiple virtual resources by adding an intermediate software layer on top of the host system. Such virtual resources are variously implemented virtual machines (VMs), which create isolated execution environments.

Nowadays, there are several virtualization technologies for cloud computing, but the most popular are the hypervisor-based and the container-based virtualization. The hypervisor-based virtualization provides full abstraction of VMs, thus they are completely separated from the host operating system. In contrast, container-based virtualization offers a more light-weight solution by leveraging the low-level mechanics of the host operating system. Using the operating-system-level virtualization, it provides abstraction directly to guest processes.

Figure 2.7 compares the differences between both virtualization technologies. As the figure shows, the hypervisor-based virtualization places a hypervisor layer (also called as Virtual Machine Monitor) upon the host operating system, which provides full abstraction of guest operating systems. This enables the VMs to have its own operating system, which can differ from the host operating system. The hypervisor completely isolates the VMs, preventing them to communicate as they are unaware of each other's existence. The advantages of full isolation results

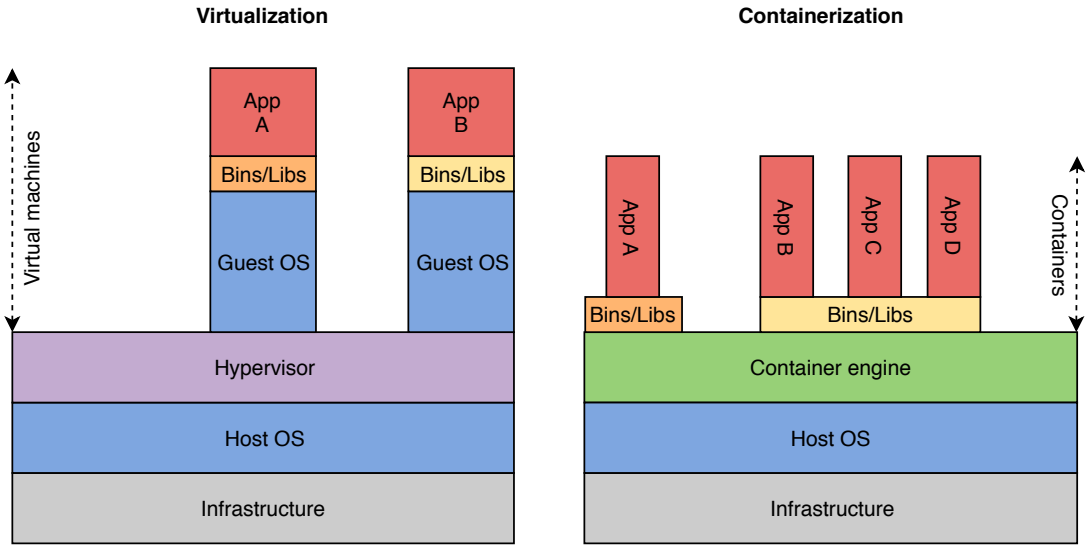


Figure 2.7: Comparison of virtual machines and containers.

in high-level of security, ensuring that problems in one of the guest operating systems do not affect the other virtual machines.

The container-based virtualization replaces the hypervisor layer with container engine layer, which ensures the operating-system-level virtualization. In contrast to completely isolated virtual machines, containers are not fully separated from the host. This allows the containers to omit the implementation of their own operating system, but share the host machine's operating system. Therefore, the container's operating system is given by the host machine, and it is not enabled to install a guest operating system. Furthermore, to avoid resource duplication, containers can share additional libraries with the host. The fact that containers share a single operating system, supposes a weaker isolation compared to the hypervisor-based virtualization. This enables the containers to communicate with each other via container engine. The presence of the host operating system makes the containers very light-weight, which results in fast boot-up and termination compared to hypervisor-based virtualization, since there is no need to create and destroy a custom VM.

Regarding to the performance, there is a noticeable difference between hypervisor- and container-based virtualization technologies. Hypervisor consumes about 10% to 15% of host machine resources, while containers consume a minimal amount [9]. Furthermore, based on the results of Felter et al. [10], the overall performance of container-based virtualization is similar to the performance of a non-virtualized system. As an advantage of the lower resource demand compared to VMs, it is possible to deploy up to 100 containers on one host machine.

The recent distributions of the Linux operating systems (as part of Linux container project (LXC) [11]) use *namespaces* and *cgroups* to isolate the processes and to share operating system and hardware resources. Namespaces are used to separate groups of processes, so they cannot be seen by resources in other groups. There are six different namespaces in the current kernel, which address specific parts of the system e.g., NET provides network resources, MNT provides file-system mount-points. The cgroups provide a mechanism to restrict the resource usage of a process or a group of processes. They can be used to specify

the resource limits for each container. For instance, limit the number of CPU-cores, disk input-output-rate or size of the available memory. This enables better isolation of containers in multi-tenant environments, and prevents the scenario where one container allocates all resources. In addition, cgroups are capable for metering resource usage. Special pseudo-files are available for each container describing the usage of a particular resource e.g., files `cpuacct.usage_percpu` and `blkio.io_service_bytes` contain CPU and disk usage, respectively .

Nowadays, Docker [12] is the most popular, open-source container-manager tool, which extends LXC with various APIs. Docker containers are created from Docker images. One can create an image using a special file, called *Dockerfile*, which contains operating system fundamentals and additional frameworks. The final Docker image consists of layers, where each command in the Dockerfile corresponds to one layer. Therefore, when re-building the image, the unmodified layers can be reused, thus the build-time can be decreased. The final images are stored in the Docker *registry*, which is either a local storage or a public online storage such as the Docker Hub[2]. Docker uses a client-server architecture. The client is responsible for providing user interface and passing commands to the server (called as *daemon*). The Docker daemon listens for requests and manages the containers accordingly.

The standardized environment provided by the container-based virtualization allows fast and consistent application delivery. The container images are created at build-time rather than deployment-time ensure a consistent environment, which can be carried from development to production. The light-weight container-based platform offers low overhead and high scalability, thus enables the applications to scale up or tear down accordingly to the demand. Furthermore, the ability to isolate applications along with fine-grained resource restriction make the platform ideal for high density environments, such as the edge-cloud hosts.

2.2.4 Orchestration

In small cloud environments, e.g., developing and testing on local machines, it is possible to manage containers manually. However, production environments nowadays host a vast amount of applications consisting of multiple services. These applications span over multiple virtual machines, creating colocated environments. The increasing number of containers impose growing demand on container management, which requires to automate certain processes in order to guarantee the quality of cloud services.

In recent years, Kubernetes [13] has become the most popular platform to automate Linux container operations. It is designed to eliminate manual processes required to manage orchestration and scaling. Kubernetes groups together containers that belong to a particular application into a logical unit (*cluster*). Therefore, it creates an abstraction layer, which enables the Kubernetes to manage and orchestrate the clusters efficiently. These capabilities provide support for deploying applications, which span over multiple containers, scale resources based on current demand or monitor the containers' health and replace the failed ones. Furthermore, Kubernetes eliminates the need to manually manage the infrastructure. It automatically sets up the network and assigns IP addresses to

[2] `https://hub.docker.com/`

the deployed containers. The platform mounts sufficient amount of storage supporting various network storage systems i.e., iSCSI, Gluster, and Ceph. Beyond providing telemetry, Kubernetes continuously monitors containers, and reschedules them to a different hosts if the current host dies or does not respond. In order to improve the cost-effectiveness and utilization of the cloud, Kubernetes automatically moves containers between hosts while respecting the applications' resource requirements.

In spite of all these advantages, we are not going to use any third-party container manager platform in this thesis, rather we are going to implement a custom one. Since, we are going to measure the containerized applications, we require overall control over the containers. Furthermore, the measurements will be executed on a small number of local machines (approx. 8 - 10 machines), which can be managed manually. However, excluding Kubernetes does not affect the results of this thesis. The proposed principle will be independent on the container management platform, therefore it will be applicable in any edge-cloud environment.

3. Solution Overview

In the Introduction chapter, we have already presented our goals, however, in this chapter we provide a more detailed description. First, we review the main idea of the problem. Then, we analyze the requirements that we should consider during the design process. Next, we introduce the concept of the proposed solution, where we present the main building blocks. Finally, we describe the structure of the solution including the main problems we are going to deal with.

3.1 The Big Picture

The goal of this thesis is to propose a solution that provides soft real-time guarantees on the microservice execution time. The targeted microservices are developed in high-level programming languages e.g., Java, Scala, and Python, etc., and are executed in a container-based cloud environment (Docker in our case), which is managed by a cluster orchestration system e.g., Kubernetes.

To statistically guarantee the upper bound of the response time, we design a deployment framework that exercises the submitted microservice in various workload combinations and analyzes the microservice's behavior. Then, the result of the analysis determines if the microservice response time requirements can be met. The deployment framework executes a sequence of predefined steps, referred as *submission process*, to determine whether the edge-cloud environment is capable of providing the statistical guarantees. This submission process adds an additional step to the typical approval process compared to other cloud submission processes. Since the submission process includes the analysis of the submitted microservice, we do not require any prior knowledge given by the developer. Therefore, we can treat the microservices as black-boxes and thus we can minimize the impact of the developer in the submission process.

The submission process includes two main parts: performance assessment and prediction of the execution time. Even though, this thesis will focus on these two areas, we are going to implement a prototype version of the whole deployment framework in order to be able to test and verify our proposed solution.

3.2 Identifying the Requirements

The goal of this section is to identify the requirements placed on the implementation of the deployment framework. The precise enumeration and analysis of the requirements are essential for the valid design and implementation of the proposed solution.

As we stated in previous sections, the main goal of this thesis is to target the performance assessment and the data evaluation along with the prediction. In this section we focus on the requirements analysis and design decisions of the performance assessment part. The data evaluation and prediction part is presented in more detail in Chapter 7.

Automation

One of the main requirements is the automation of the deployment framework. To enable the automation, the steps of the submission process should be predefined and their execution should be managed by a central component. The central component should be able to distribute the microservices over multiple agents to increase the effectiveness of the submission process.

Configurability

Even though the submission process is a sequence of predefined steps, each microservice may have its own parameters that are required for executing its test-points. Ideally, these parameters should be attached to the microservice in a configuration file. The concept of such a configuration file is already shown in Section 2.1.2 and is referred as *requirements specification*. Ideally, the final requirements specification file should be designed so that it can include all the desired parameters. In order to be able to easily manipulate and process the file, it should be written in a well-known data-serialization language e.g., JSON[1] or YAML[2].

Containerization

During the submission process, the microservices will mostly be executed with several number of colocated workloads. Thus, the efficient management of multiple workloads is an essential capability. Microservices require various artifacts for their execution e.g., source code, libraries, dependencies, runtime environment and settings. Therefore, the ideal solution would be packaging these requirements into one unit that can be executed separately. Furthermore, to comply with the requirements in the configuration file or to prevent the microservice from allocating large amount of resources, the chosen solution should support applying various resource usage limitations. These resource usage limitations include the number of CPU-cores, the amount of RAM or disk access speed. In order to meet the previously described requirements, the microservices should be executed in a containerized environment (like the popular Docker), which is a lightweight alternative of virtual machines.

Communication Interface

Even though our aim is to handle the microservice as black-boxes and to minimize the constraints against the developer, we require a certain level of communication between the microservice and the assessing environment. This communication should enable the performance assessment framework to load the microservice and exercise its test-points. In order to be able to communicate with the microservice, each microservice should implement a common interface specified by the performance assessment framework.

[1] `https://json.org`
[2] `https://yaml.org`

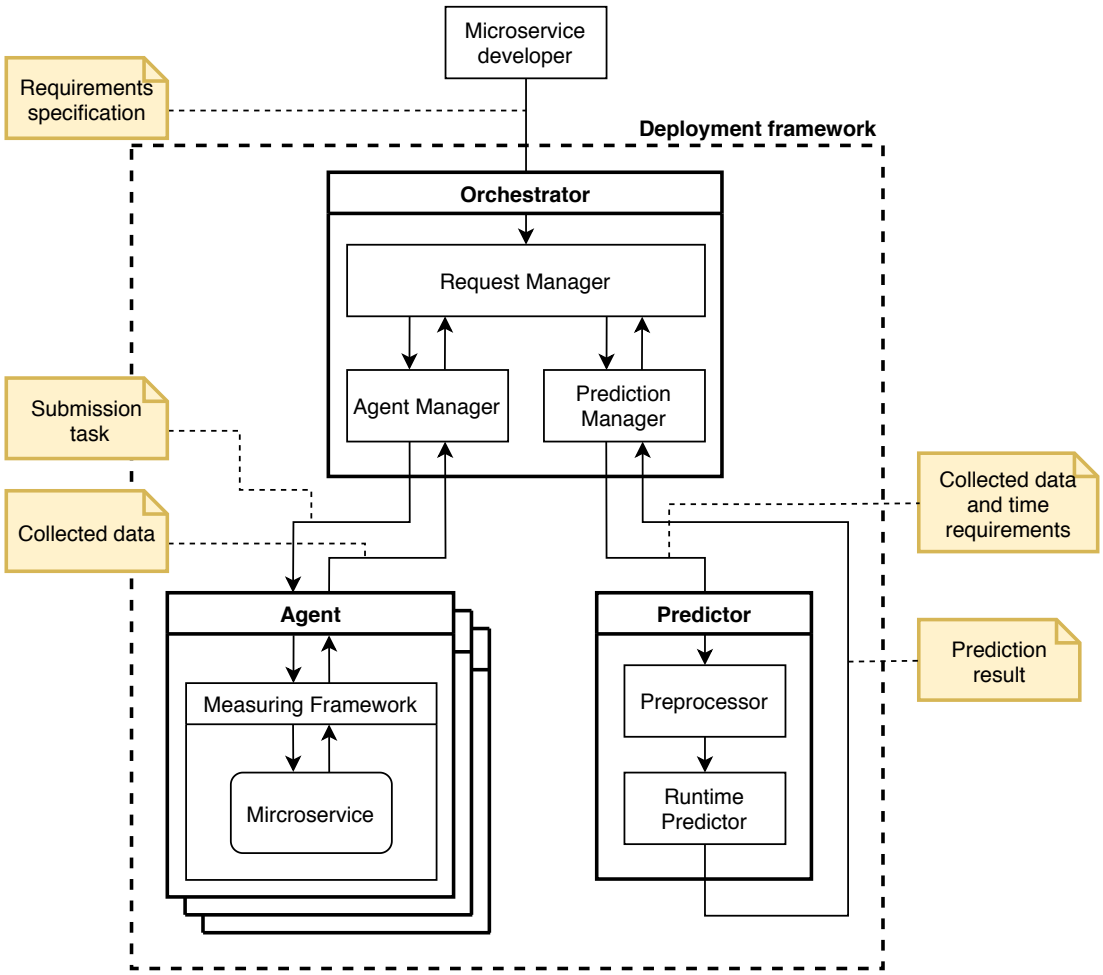


Figure 3.1: Concept of the solution.

3.3 Concept of the Solution

In the previous parts of the text we have provided only a basic overview of the problem. In this section, we introduce the problem and the proposed solution in a more detailed way. We divide the solution into distinct parts and analyze the possible problems that need to be solved.

Even though, we are focusing on the performance assessment of the microservices and the evaluation of the results, which includes prediction of the microservice's response time, our goal is to provide a fully working prototype of the entire deployment framework, including the definition of the input files. Figure 3.1 shows the concept of the main components, which are presented in more detail in the following sections.

Unified Requirements Specification

The microservice submission process starts by uploading its requirements specification to the deployment framework. The requirements specification is a configuration file created by the microservice developer. It should be designed in a way that it enables to precisely describe the microservice's test-points and their requirements e.g., response time and probability.

Ideally, a well-known data serialization language should be selected that makes it easy for the microservice developer to create and edit the file either manually or programmatically. Considering these requirements, we have decided to use the YAML data serialization language. The final form of the requirements specification will be shown in Chapter 5.

Orchestrator

There is a need for a component that enables the communication between the microservice developer and the instances of the deployment framework. In our proposed solution, this component will be referred as *orchestrator*.

The orchestrator will be responsible for handling the following tasks:

- Communicating with the microservice developer: The orchestrator enables the microservice developer to submit the requirements specification to the deployment framework, to monitor, and to display the current state of the submission process. At the end of the submission process, the orchestrator returns the prediction result to the microservice developer.
- Managing the submission tasks: The orchestrator processes the submitted requirements specifications and converts them into submission tasks for internal representation. This representation helps the orchestrator to easily manage i.e., add, update, and remove the tasks that are pending in the submission queue.
- Managing the agents: The orchestrator establishes the communication with the online agents and registers them into the group of active agents. Then the orchestrator distributes the tasks among the agents and monitors the state of the performance assessment process.
- Supervising the performance assessment: The orchestrator distributes the submission tasks among the active agents. It manages the execution of the performance assessment process according to the predefined steps of the submission process.
- Launching the predictor: The orchestrator starts the prediction process by passing the collected data and the submitted requirements specification to the predictor.

Measuring Agent

One of the most important parts of the submission process is the execution of the microservice's test-points. During the execution, the deployment framework determines the behavior and monitors the resource usage of the microservice. In the following part of the text, we will refer these two activities as *measurement*. In order to correctly execute the measurement, a precisely defined sequence of steps have to be performed before and after the measurement. The sequence of these steps together with the measurement compose the *measurement process*.

As a part of the submission process, the microservices undergo the measurement process, which results in a dataset representing the microservice's behavior and resource usage. To be able to perform the measurement process, a new

component must be added to the solution. This component will be referred as *measuring agent*, or shortly *agent*. The agent will be responsible for handling the following tasks:

- Communication and data transfer: The agent queries the orchestrator for new submission tasks and reports the current state of the measurement process. At the end of the measurement process, the agent sends the collected dataset to the orchestrator.
- Maintaining an isolated environment: The agent isolates the measurement by packaging the source artifacts into a container. Thus, the agent will be responsible for managing the container life-cycle e.g., deploying the container, executing the measurement process, and removing the container.
- Providing the measuring framework: The agent provides a framework to load the microservice and to exercise its test-points. Furthermore, the framework will be responsible for recording the microservice's behavior and its resource usage.

The agent is the only component that might have multiple instances running at the same time in the deployment framework. The instances communicate with the orchestrator and execute various submission tasks, each representing a particular workload combination of the submitted microservices.

Predictor

The final part of the submission process includes several subsequent tasks that create the *prediction process*, which is managed by the component called *predictor*. The goal of the prediction process is to predict the microservice execution time using the measured data. The prediction process includes the following tasks:

- Parsing requirements specification: The prediction process first parses the input requirements specifications and transforms them into an internal representation of requirements.
- Preprocessing the input data: The prediction process preprocesses the data collected by measuring agents and transforms them into a representation which is suitable for the prediction process.
- Prediction: Using the preprocessed input data the predictor predicts the execution time of the submitted microservice. The microservice execution time is predicted for various workload combinations that are predefined in the prediction process.
- Returning the results: The prediction process evaluates the prediction results against the parsed requirements specifications. Finally, the prediction process returns the prediction results along with the evaluation of the requirements specifications.

3.4 Structure of the Solution

The purpose of this section is to present the detailed goals of the thesis and the key parts of the solution that will be described in the subsequent chapters.

Measuring Resource Usage

Since each microservice is designed for different purpose, their resource usage requirements might be completely different as well. In addition, we treat the microservices as black-boxes, without prior knowledge of their properties. In order to predict the microservice's response time in a particular cloud environment, it is necessary to characterize the microservice's behavior. This can be done by executing the microservice's test-points and monitoring their resource usage.

Thus, in this part of the thesis we are going to research for resource usage indicators that can be used to infer the microservice's behavior. Then, we are going to investigate the features of the Linux platform, which enable us to measure the test-points' resource usage. Next, we will search for other third-party solutions to measure resource usage. Finally, we will evaluate the analyzed solutions and select those that meet our requirements.

Designing the Steps of the Submission Process

Creating a large and high-quality dataset requires a large amount of measurements. Furthermore, each measurement must be repeated several times to eliminate the outliers and thus to ensure data validity. It would be impossible to manually manage the measurements, therefore the submission process should be fully automated. Besides the measurement process, the submission process includes a sequence of additional steps that have to be carried out before the measurement can be started. Therefore, we will analyze the submission process and design the main steps, which cover the entire execution process, including the environment initialization, the measurement, and the final phase.

First, we focus on the environment initialization phase. Creating a unified virtual environment is crucial in order to make the collected data comparable, and to execute all the microservices under the same circumstances. To cover the measurement phase, we will design the steps of the measurement process one by one. Finally, we will compose the last steps that are responsible for finishing the measurement, cleaning up the working environment, and transferring the collected data to the orchestrator.

Generating Artificial Workload

In order to collect data, validate our assumptions, and evaluate the predictor, it is essential to own microservices that can be used throughout the development. Since the real-life microservices are not accessible at the time of writing, we will implement our own set of microservices that generate artificial workload and simulate the behavior of the real-life microservices. Each artificial microservice will behave differently as they simulate different resource usage e.g., CPU-intensive computation, processing large files, etc. The microservices will be executed both in isolation and along with other workloads. Therefore, the various workload

combinations (running 2-3-4 microservices in parallel) will be able to simulate various real-life scenarios. The majority of the artificial microservices will be implemented by us using open-source libraries. The remaining microservices will be integrated from third-party sources, like the Scala Benchmarking Project [14] or the stress-ng Linux tool [15].

Predicting the Execution Time

In this part of the thesis we will utilize the collected data. Our goal is to elaborate a prediction method, which predicts the microservice response time, when it is colocated with other workloads in a single machine. In order to do so, we will use the data generated by the artificial workloads and collected by the measuring agents. However, the collected data is in raw format that is not supported by the predictor. Therefore, in the first step we will preprocess the data into a suitable format for the predictor. Then, we will transform this data into a representation that can be used to characterize the microservice behavior. Next, we will build a model of the microservice, using the preprocessed data. Finally, we will use this model to predict the execution time of the microservice in various workload combinations.

Implementing a Prototype

To validate the designed architecture and to provide an example implementation, we are going to implement a prototype of the deployment framework. This will include a concrete representation of the requirements specification that enables us to specify both the properties and the requirements of the submitted microservice. Furthermore, we will implement the orchestrator to ensure the communication with both the microservice developer and the parts of the deployment framework. Next, we will implement the measuring agent that provides the framework for executing the submitted microservices and collecting data about their behavior and resource usage. Finally, we will implement a mechanism for processing the measured data and predicting the execution time. We will not only focus on the implementation of the separated modules but we will also pay an attention to the communication between these modules.

4. Measuring Resource Usage

In this chapter, we focus on measuring the resource usage indicators that can be used for characterizing the microservice's behavior. To be able to characterize the microservice's behavior and to compare the microservices during the prediction process, a common representation should be used for all types of microservices. Such a common representation can be defined by the resource usage pattern, which records various attributes of the utilized hardware resources e.g., CPU utilization, I/O (storage) bandwidth, and memory utilization throughout the microservice's execution. Recording the resource usage patterns during the execution of the microservice test-points, the system can create a model of the microservice. Combining this model with the model of any known microservice, the system will be able to predict the microservice's response time for a particular set of colocated workloads.

In the first part of the chapter, we analyze the main hardware resources, particularly the CPU, the disk drive, and the RAM, by presenting their operating principles and the concepts that affect the system's performance the most. In the second part of the chapter, we present and analyze various observability types and methodologies. Then, we describe a wide range of tools to monitor the system's resource usage throughout the microservice's execution.

The research will be executed on a 64-bit quad-core Intel system[1], where Turbo-Boost, Hyper-Threading and other performance-related features were disabled. Furthermore, the system is equipped with 32GB of physical memory and one dedicated 500GB HDD[2]. The installed operating system is Fedora Linux 28[3].

The structure of this chapter and the description of concepts are based on the systems performance book [16].

4.1 Concepts

4.1.1 CPU

In this section, we present the CPU concepts that are essential for measuring and analyzing the microservices' performance and resource usage pattern.

The CPU carries out the instructions of a computer program by performing various operations e.g., logical, arithmetical, and controlling. Since, the CPU is an essential component in a computer, it highly affects the microservice's performance along with its response time. Nowadays, host machines in data centers are equipped with multi-core processors, which enable the operating system a high level of parallelism. However, if the microservices' overall resource demand exceeds the CPU's capabilities, the tasks start to queue up. The growing waiting time lowers the microservices' performance and thus increases their response time.

[1] Intel Xeon E3-1230v6 @ 3.50GHz

[2] Seagate Constellation.2 ST9500620NS 500GB 7200 RPM 64MB Cache SATA 6.0Gb/s 2.5"

[3] Kernel version: 4.17.3-200.fc28.x86_64

Clock Rate

The clock rate refers to the frequency at which the CPU is running, therefore it is often referenced as CPU cycles. The clock rate is measured in Hertz (Hz) expressing that a 3 GHz CPU performs 3 billion clock cycles per second. Modern processors are able to dynamically change their clock rate. This enables to increase their performance when it is needed, or decrease the energy-consumption when the computer is idle. Although, the clock rate is an important metric when comparing various CPUs, there are situations where a faster processor cannot speed up the system. It often happens, when the CPU cycles are stalled, waiting for another resource e.g., disk access. In such cases, the higher clock rate would not increase the CPU's throughput.

Instruction Set

The instruction set is a list of commands that the CPU can execute. They include control flow, data, logic, and arithmetic instructions. Execution of various instructions requires one or more CPU cycles. For instance, memory I/O can consume tens of clock cycles, during which the instruction execution is suspended. These elapsed clock cycles are called *stall cycles*.

Cycles per Instruction

The Cycles per instruction (CPI) is a metric describing the nature of the CPU utilization. CPI helps to study how the CPU cycles are spent among the instructions. It shows how efficient the instruction processing is, but not the instructions themselves. Low CPI value indicates high instruction throughput. However, high CPI means that the processor is stalled in the majority of time. For instance, blocked by slow memory access.

Utilization

CPU utilization is measured in percentage and is defined as "the time a CPU instance is busy performing work during an interval" [16]. In other words, it is measured by the time, when the CPU executes a user-thread or a kernel-thread different from the idle-thread. Nowadays, the operating system kernels support thread priorities, preemption and time-sharing, which prevent the CPU from steep performance degradation even at high utilization.

User-time and Kernel-time

CPU utilization can be divided into *user-time* and *kernel-time*. Time spent on executing the user-level thread belongs to user-time, while time spent calling the kernel functions e.g., syscalls, interrupts belongs to kernel-time. The user-time/kernel-time ratio can be used to characterize the workload. The computation-intensive applications, which only utilize CPU without massive usage of other resources e.g., disk execute user-level code. In such cases, the user-time/kernel-time ratio can achieve even 99 to 1 [16]. In contrast, I/O-intensive applications perform large number of syscalls through the operating system's kernel. Therefore, their user-time/kernel-time ratio approximates 70/30 [16].

Saturation

The CPU becomes saturated when it is fully utilized (at 100%) and the execution of threads is limited as they wait to be scheduled. A saturated CPU results in a congested scheduler queue and in growing latency. The saturation can occur not only at 100% utilization, but when the resource exceeds a set threshold. For instance, the cloud environment could limit the container's CPU utilization at 25%, and thus ensures that the CPU can be equally shared among 4 containers.

4.1.2 Memory

In the following, we present the memory concepts that are essential for measuring and analyzing the microservices' performance and resource usage pattern.

The system main memory stores operating-system and user application code, their data, and file system caches. The main memory's size is relatively small (tens of gigabytes) compared to hard disk drives (up to 10 TB), however, access rate of main memory is higher by orders of magnitude. Once the main memory fills, the system is forced to store and retrieve the data form the slower disk device causing massive drop in the application's performance.

Apart from the disk storage, one should consider the CPU expenses e.g., allocating, freeing, copying memory, etc. when analyzing performance factors. In addition, performance can be affected by the memory architecture (uniform memory access vs. non-uniform memory access) as the locally attached memory ensures lower latency than the remote memory.

Virtual Memory

The virtual memory is a memory management technique which provides abstraction for the processes about the available memory storage. The usage of the virtual memory creates an illusion of a large, contiguous, and private address space for every process. This is beneficial for software developers, because the operating system handles virtual address spaces and physical memory placement. The operating system's memory management capabilities include extending the size of the virtual memory over the available physical memory. This mechanism is called *paging*.

Paging

It is a memory management scheme by which the operating system moves the memory pages between the main memory and the secondary storage device (swap device or swap disk). This mechanism enables the operating system to create an illusion of a large memory and therefore, execute applications which otherwise would not fit into the memory. In most of operating systems the page size is 4 KB, thus paging is a fine-grained approach to manage the main memory. However, paging can cause a significant performance bottleneck. When an application wants to access a page that has been moved to swap device, it will be blocked until the page is not read from the slow device.

Swapping

While paging moves only small parts of processes, swapping saves and retrieves complete processes from the swap device to the main memory. During swapping, all the private data of the process is saved to the swap disk, including the process heap and the structure of the threads. The operating system kernel, however, keeps track of the swapped-out processes, because their meta-data is still stored in the memory. Swapping-in a process is more I/O-intensive and takes more time than paging. To minimize the waiting time while a process is being swapped-in, the kernel prioritizes the smaller processes which have been waiting for long time on the swap device.

Nowadays, the term "swapping" is used in cases when the system moves not only complete processes to swap device but single pages as well. Solaris-based operating systems still support swapping processes when paging is not sufficient. On Linux, swapping refers to the mechanism of paging. Throughout this thesis, we will use the term swapping as a synonym of paging.

File System Cache Usage

As we described at the beginning of this section, memory devices (usually DRAM[4] modules) are orders of magnitude faster than storage devices (HDD[5] or SSD[6]). In order to improve file-access speed and application performance, the operating system can utilize the unused part of the main memory as file system cache. Although the cached files can allocate a significant part of the memory, it will not cause performance costs, because the kernel can quickly free the memory when applications need it.

Utilization and Saturation

The memory utilization is the ratio of used memory and total memory. The memory allocated by cached files does not count into the used memory as it can be quickly released. If the applications' overall memory demand exceeds the total available memory, the system becomes *saturated*. In this case the operating system tries to free memory using the mechanism of paging, swapping or even killing the most memory-intensive application. The excessive usage any of these mechanisms indicate a highly saturated system.

4.1.3 Disk

In the following, we present the disk concepts that are essential for measuring and analyzing the microservices' performance and resource usage pattern.

The disk storage refers to the primary storage devices for the system, which enables to permanently store and retrieve large amount of data. Compared to the main memory and the processor cache, disk drives are relatively slow, and therefore they can strongly influence the system's performance. Under high load,

[4]Dynamic Random Access Memory
[5]Hard disk drive
[6]Solid-state drive

the disk storage becomes quickly the performance bottleneck which forces the CPU to wait for the I/O completions.

Nowadays, the most popular storage drives are hard disk drives and solid-state drives. The former is popular for its high capacity/price ratio, while the latter is preferred for its high I/O performance. Regardless of the underlying technology, both drives can be the source of performance issues.

Response Time

The response time expresses the time spent between the initialization and completion of an I/O request. It is the summary of the *service time* and the *wait time*. The service time represents the time that an I/O request requires to be actively processed. This time period excludes the time spent in the queue, waiting to be executed, which is exactly represented by the wait time. Despite of the exact definitions, the way the times are measured depends on the particular context. From the point of the OS, the service time is the time from the request initialization until the completion interrupt is signalized. From the point of the disk, the service time is the time that is spent while the disk was actively executing the request, excluding the time spent in the disk queue.

Caching and Asynchronous I/O

In order to compensate the high I/O latency, file systems support the mechanism of *caching*. This mechanism is invisible for applications, which are not aware of whether the data is stored in the main memory or on the disk drive. Caching utilizes the unused part of the main memory, however, it can be quickly released when an application requires more memory. The mechanism of caching is actively used in both read and write operations.

The applications often work with large files that could not be fit into the memory. Therefore, they read and process the large file in smaller chunks. The file systems are able to detect that an application performs sequential reads on a large file. To improve the application's read performance, the file system reads the file asynchronously and caches the data. Storing the data in the cache improves the application's performance, as the read operations result in cache hit.

To improve the write performance, the file systems commonly use *write-back* caching. This mechanism stores the data into the fast memory instead of the slow disks and thus improves the application's latency. Later, a process called *flushing* goes through the memory asynchronously and writes the cached data to disk.

Random vs. Sequential I/O

A series of I/O operations can be described as *random* or *sequential* based on the relative offset of the I/O operations. In case of the random I/O workload, there is no apparent relationship between each I/O operations, while in case of the sequential workload, the next I/O begins immediately after the end of the previous operation. As it implies from the architecture of HDDs, the access patterns have various impacts on their performance. The random I/O operations increase the latency as the disk platter rotates and the disk heads seek to a specific

position. The sequential I/O operations enable the file system to read ahead or prefetch the data and thus improve the performance.

Read-Write Ratio

Another possibility to characterize the workload is the ratio of read and write operations. Being familiar with the read-write ratio helps to predict the resource requirements. Applications with high percentage of read operations would benefit from larger system cache as it would eliminate the unnecessary disk I/O. However, the application with high write rate would take advantage of more disk devices, which increase the throughput.

Utilization and Saturation

The disk utilization is defined as "the time disk was busy actively performing work during an interval" [16]. Due to the fact, that the disk I/O is a typically slow activity, any level of disk utilization can lead to decreased performance. To make sure, that the performance degradation is caused by the I/O utilization, it is recommended to monitor the disk response time and whether the application is blocked on this I/O. If the amount of requests exceed the performance that the disk can deliver, the requests queue up and the disk becomes saturated.

In cloud computing, the disk resources are often virtualized, and therefore it is not straightforward to measure the disk utilization. For instance, 100% utilization not necessarily indicates that all the disk drives are fully loaded, but there can be idle disk drives. Another typical case is, when the virtual disks are equipped with the write-back cache. The disk controller might report I/O completion immediately as the I/O operations are handled by the fast cache, but in the meanwhile, disks are busy until they store the cached data.

4.2 Monitoring Utilities

The monitoring tools have been continuously developed alongside the operating system. Modern operating systems include a wide variety of built-in monitoring tools and provide support for installing external ones. There are two common perspectives that can be used to divide the resource monitoring tools into groups. The first perspective focuses on the observed target. In this case, we can distinguish two types of resource monitoring tools: *system-wide* and *per-process*. The second perspective focuses on the monitoring method the resource monitoring tools are based on. This includes the following three groups: *counters*, *profiling*, and *tracing*.

In this section, we present the most popular monitoring tools divided into categories by their monitoring method. Then, we compare the presented tools and analyze how effectively they could be used for monitoring the containerized microservices' resource usage patterns along three dimensions: CPU utilization, I/O (storage) utilization, and memory utilization. We prefer solutions that can be tightly connected to the measured microservice and provide detailed information about the microservice's resource usage.

4.2.1 Sample Application

The following sections cover a wide range of monitoring tools that differ in many ways e.g., focus on the observed target, monitoring method, etc. Despite of the differences, our aim is to objectively evaluate all the monitoring tools. For this purpose, we have designed a sample use-case scenario that generates the same load in every execution.

The sample use-case scenario includes 3 microservices, which are executed in separate Docker containers. Furthermore, the microservices are divided into one main and two background workloads. The main workload is represented as a disk- and CPU-intensive application that extracts a 188 MB zip archive containing 4369 files and 184 directories. The operation produces a total of 1.08 GB of uncompressed data written disk. One of the two background workloads is the exact copy of the main workload, therefore it extracts a copy of the same zip archive. The second background workload sequentially inserts and deletes 5 000 000 nodes into an AVL tree. For more details about the microservices please see Chapter 6.

4.2.2 Counters

The operating system kernel provides statistics for the system events in a form of counters. Traditionally, counters are represented by unsigned integers, which are increased every time the particular event occurs. Counters are implemented by the kernel, and their continuous maintenance imposes a negligible overhead on the system. The current values can be easily accessed by reading the appropriate pseudo-files. These properties make the counters widely popular among various system-wide and per-process monitoring tools.

Tool	Description
`vmstat`	Reports information about CPU and disk activity, memory, running processes, etc.
`mpstat`	Reports processor-related statistics.
`iostat`	Reports CPU and I/O statistics for disk drives and partitions.
`netstat`	Reports information about the network subsystem.

Table 4.1: System-wide monitoring tools.

Tool	Description
`ps`	Reports a snapshot of the current processes, including the process ID, the executable name, etc.
`top`	Provides a dynamic real-time view of the system summary and the list of processes or threads currently running.
`pmap`	Reports a memory map of a particular process or processes.

Table 4.2: Per-process monitoring tools.

In this section, we present monitoring tools that use the built-in system counters. We show how they work and analyze how they could be used for monitoring the containerized applications' resource usage.

Table 4.1 includes a selection of system-wide tools that monitor the system-activity and the hardware resources by reading system-wide counters. Table 4.2 includes a selection of per-process tools that display the values of process-related counters.

`vmstat` Monitoring Tool

It is a light-weight and low-overhead monitoring tool reporting information about processes, memory, paging, block device input-output, and CPU activity. By default, `vmstat` prints the output to the console. The first line of the output contains the system averages since the last boot, then the following lines report information about the sampling period specified as a command-line argument. Beyond displaying CPU and virtual memory statistics, the tool can also display disk statistics if the `-d` option is present.

```
1 $ vmstat -Sm 1 5
2 r  b   swpd   free  buff  cache  si  so   bi     bo   in    cs us sy id wa st
3 7  0      0  30816     2   1221   0   0  457   8468 1099  2926 54 10 32  5  0
4 6  0      0  30553     2   1485   0   0   16  19428 4444  3741 60 17 19  5  0
5 2  1      0  30339     2   1679   0   0    0  48804 6720 20314 52 20 26  3  0
6 3  1      0  30202     2   1784   0   0    0  45508 4291 12755 53 12 27  8  0
7 3  0      0  29965     2   2069   0   0    0  13592 4763  5987 68  7 24  0  0
```

Listing 4.1: Sample output of `vmstat` monitoring tool.

Listing 4.1 displays the default statistics for 5 seconds with 1 second iteration-step. To make the output more compact, the output unit is set to megabytes. The attributes of `vmstat` shown in Listing 4.1 are described in Table 4.3.

Interpreting the Output: Using the CPU-related attributes, the sample output can be analyzed as follows. The number of running or waiting processes is between 2 and 7 and there is 1 process that is in uninterruptible state for a short time. The CPU is not fully utilized, as the average CPU utilization is equal to 69% (`us` + `sy` + `wa`). The high value of interrupts and context-switches indicate that the computation uses other resources than the CPU e.g., disk or memory which blocks the CPU. This assumption is supported by the relatively high amount of time spent in the kernel code (`sy`), and the time the CPU was waiting for I/O (`wa`).

The decreasing amount of free memory (`free`) indicates that the application allocates memory continuously. Moreover, we can see that the memory is allocated for cache (`cache`), where usually the I/O data is stored. There is no need for swapping (`swpd`), as the amount of free memory is sufficiently high.

Analyzing the disk I/O-related attributes of the output, we can see from the high amount of sent blocks (`bo`) that the currently running microservice utilizes the disk device. More specifically, the disk utilization is mainly generated by write operation, because the amount of read blocks (`bi`) is 0 in most of the time.

Based on the analysis of the CPU, the memory, and the disk attributes, the executed microservice could be characterized as a moderately CPU- and memory-intensive application that generates high-amount of data and stores it on the disk.

Despite the fact that the `vmstat` is also able to monitor the disk usage, in the following section we present the `iostat` tool, because it provides a more detailed view on the disk usage.

Attribute	Description
r	The number of processes running or waiting for run time.
b	The number of processes in uninterruptible sleep.
swpd	The amount of virtual memory used.
free	The amount of idle memory.
buff	The amount of memory used as buffers.
cache	The amount of memory used as cache.
si	The amount of memory swapped in from disk (/s).
so	The amount of memory swapped to disk (/s).
bi	The amount of blocks received from a block device (blocks/s).
bo	The amount of blocks sent to a block device (blocks/s).
in	The number of interrupts per second, including the clock.
cs	The number of context switches per second.
us	Time spent running non-kernel code (user time, including nice time).
sy	Time spent running kernel code (system time).
id	Time spent idle. Prior to Linux 2.5.41, this includes I/O-wait time.
wa	Time spent waiting for I/O. Prior to Linux 2.5.41, included in idle.
st	Time stolen from a virtual machine. Prior to Linux 2.6.11, unknown.

Table 4.3: Attributes of the `vmstat` monitoring tool. Source [17].

`iostat` Monitoring Tool

The `iostat` is a low-overhead monitoring tool designed for monitoring CPU statistics and I/O statistics for devices, partitions and network file-systems. By default, `iostat` prints its output to the console. The first line of its output contains the system averages since the last boot, then the following lines report information about the sampling period specified as a command-line argument. The `iostat` tool is able to generate three types of reports: the CPU utilization report, the device utilization report, and the network file-system report. From our perspective, the device utilization report is the most valuable, therefore in the following we analyze the `iostat` monitoring tool from that perspective.

```
1  $ iostat -d sda 1
2  Linux 4.17.3-200.fc28.x86_64 (cirrus-1.edge.d3s.hide.ms.mff.cuni.cz)
       06/25/19  _x86_64_ (4 CPU)
3
4  Device             tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
5  sda              29.58         2.93       315.40   84702348 9126822328
6  sda               0.00         0.00         0.00          0          0
7  sda              37.00         0.00     21504.00          0      21504
8  sda             155.00         0.00    100640.00          0     100640
9  sda              96.00         0.00     87016.00          0      87016
10 sda             107.00         0.00    107776.00          0     107776
11 sda             110.00         0.00    110592.00          0     110592
12 sda             107.92         0.00    110003.96          0     111104
13 sda             105.00         0.00    105728.00          0     105728
14 sda             107.00         0.00    109056.00          0     109056
```

Listing 4.2: Sample of basic output of `iostat` monitoring tool.

The device utilization report can be activated by the `-d` option. The output prints a brief summary of the system statistics including the kernel version, the host name, the current date, the system architecture, and the number of CPUs. This is followed by the disk statistics, where each disk is shown in a separated row (see Listing 4.2).

```
1  $ iostat -xd sda 1
2  Linux 4.17.3-200.fc28.x86_64 (cirrus-1.edge.d3s.hide.ms.mff.cuni.cz)
       06/25/19  _x86_64_ (4 CPU)
3
4  Device   w/s      wkB/s wrqm/s  %wrqm  w_await  aqu-sz  wareq-sz  svctm  %util
5  sda    29.56     314.96   0.39   1.30     1.87    0.06     10.66   0.71   2.10
6  sda   162.00   78280.00  56.00  25.69   756.64  143.05    483.21   6.17 100.00
7  sda    84.00   86016.00 109.00  56.48  1555.26   92.94   1024.00  11.90 100.00
8  sda   129.00  114944.00   0.00   0.00   781.43  230.15    891.04   7.75 100.00
9  sda   101.00  100100.00  30.00  22.90  1477.98  165.66    991.09   9.90 100.00
10 sda   107.00  106240.00  31.00  22.46  1660.72  153.23    992.90   9.36 100.10
11 sda   109.00  109912.00  31.00  22.14  1728.89  155.35   1008.37   9.17 100.00
12 sda   109.00  110336.00  32.00  22.70  1357.32  150.82   1012.26   9.17 100.00
13 sda   107.00  107776.00  31.00  22.46  1508.79  150.64   1007.25   9.35 100.00
14 sda   107.00  109056.00  32.00  23.02  1459.26  149.79   1019.21   9.35 100.00
```

Listing 4.3: Sample of extended of output of `iostat` monitoring tool.

The `iostat` also implements an extended mode, which can be enabled by the `-x` option. Enabling this mode, the monitoring tool provides more detailed disk statistics (see Listing 4.3).

Attribute	Description
tps	The number of transfers per second that were issued to the device.
kB_read/s (kB_wrtn/s)	The amount of data read (written) from the device expressed in kilobytes, per second.
kB_read (kB_wrtn)	The total number of kilobytes read (written).
r/s (w/s)	The number (after merges) of read (write) requests completed per second for the device.
rkB/s (wkB/s)	The number of kilobytes read (written) from the device per second.
rrqm/s (wrqm/s)	The number of read (write) requests merged per second that were queued to the device.
%rrqm (%wrqm)	The percentage of read (write) requests merged together before being sent to the device.
r_await (w_await)	The average time (in milliseconds) for read (write) requests issued to the device to be served
rareq-sz (wareq-sz)	The average size (in kilobytes) of the read (write) requests that were issued to the device.
aqu-sz	The average queue length of the requests that were issued to the device.
svtcm	The average service time (in milliseconds) for I/O requests that were issued to the device.
%util	Percentage of elapsed time during which I/O requests were issued to the device (bandwidth utilization for the device).

Table 4.4: Attributes of the iostat monitoring tool. Source [18].

Table 4.4 describes the displayed columns both in default and extended mode. In the rest of this section, we focus on the extended output as it provides more details about the system.

Interpreting the Output: To present how the output of the iostat can be interpreted, we have executed a disk intensive application, that writes data to the disk. Listing 4.3 shows the extended output that was collected during the execution of the application. We have omitted the disk reading-related columns, as their values were constantly zero.

The sample output shows that the disk utilization (%util) is at 100%. This is also observable by the write operations throughput (wkB/s) that are close to the maximum bandwidth of that particular HDD drive [19]. Even though the disk is fully utilized, it cannot process the requests fast enough, which is observable from the high average time for write request (w_await) and the average queue length (aqu-sz). Beyond the utilization, the I/O-pattern can be also concluded. The high average size of the write requests (wareq-sz) indicates an efficient execution of the I/O operations, which is typical for the sequential I/O.

Based on the analysis of the extended output of the iostat monitoring tool, we can see that the disk demanded with a large sequential load. Although the disk is fully utilized, the huge load forces the requests to wait in the disk scheduler

queue, which leads to increased application latency.

Summary of `vmstat` and `iostat`

Comparing the presented monitoring tools, we can conclude that they both share the same positive properties. The tools are easy to use and provide detailed information about the system's resources. In addition, both `vmstat` and `iostat` create a negligible overhead to the system as they use the built-in counters maintained by the operating system kernel.

However, as the goal of the chapter states, our aim is to find such monitoring tools that can be used for monitoring the Docker containers' resource usage. Even though the presented tools provide a detailed view of the system, they are unable to monitor a particular container or a set of containers. In addition, the tools are implemented as independent applications, which should be executed in parallel with the microservice in order to collect the system data. This fact makes their management more complicated if the desired goal is to tightly interconnect the monitoring tools with microservice execution. For our purposes, the ideal solution would be directly accessing the system counters from the measuring framework. This direct access would enable us to select the relevant system attributes and transform them into a representation that the most suits our requirements.

Per-process Monitoring Tools

The counter-based per-process monitoring tools use the same principle as the previously presented system-wide tools. The majority of per-process tools use the `/proc` interface to retrieve information about the available processes.

```
1  top - 18:52:54 up 17 days, 8:04, 1 user, load average: 3.90, 1.07, 0.37
2  Tasks: 159 total, 3 running, 71 sleeping, 0 stopped, 0 zombie
3  %Cpu(s): 42.8 us, 24.0 sy, 0.0 ni, 17.9 id, 15.0 wa, 0.2 hi, 0.2 si, 0.0 st
4  KiB Mem : 32759864 total, 24444276 free, 2021236 used, 6294352 buff/cache
5  KiB Swap: 0 total, 0 free, 0 used. 30322004 avail Mem
6
7   PID USER PR NI    VIRT    RES   SHR S %CPU %MEM   TIME+ COMMAND
8  7215 root 20  0 10.463g 447440 19956 S 99.7  1.4 0:34.53    java
9  7070 root 20  0 10.527g 317864 19808 S 56.8  1.0 0:12.57    java
10 7167 root 20  0 10.463g 358288 19892 S 44.9  1.1 0:14.54    java
11 6962 root 20  0       0      0     0 R 12.0  0.0 0:01.62    kworker/u8:4
12 6872 root 20  0       0      0     0 I 11.3  0.0 0:01.38    kworker/u8:0
13 7336 root 20  0       0      0     0 I 11.3  0.0 0:00.96    kworker/u8:12
14 6959 root 20  0       0      0     0 I  9.0  0.0 0:01.68    kworker/u8:1
15 6963 root 20  0       0      0     0 R  8.0  0.0 0:01.98    kworker/u8:6
16 6980 root 20  0       0      0     0 I  7.3  0.0 0:00.82    kworker/u8:8
17 6961 root 20  0       0      0     0 I  4.7  0.0 0:01.31    kworker/u8:3
```

Listing 4.4: Sample of the `top` monitoring tool.

Similarly to `vmstat`, `iostat`, and other third-party tools, `ps` and `top` print their output to the console by default. Listing 4.4 shows an example of the `top` monitoring tool. The listing starts with the system-wide CPU load averages, summary of tasks and total CPU load followed by the total memory usage. Then, the output lists the processes, which are sorted by the top CPU intensity by default.

Comparing the output of `top` with the output of `vmstat`, it is observable that the displayed attributes are very similar in both monitoring tools. However, the `top` lacks several important attributes. For instance, there is no information available about the disk usage of the particular process.

Furthermore, the presented per-process monitoring tools are third party solutions as well as the system-wide tools. This implies, that their output can be configured only in a limited manner, and the tools cannot be tightly connected with the measuring framework. With regard to these properties, our overall conclusion is not to use the third-party per-process monitoring tools.

Sources of Resource Counters

The previously presented monitoring tools use a central data source that is maintained by the operating system kernel. The most commonly used interface for accessing system counters are hierarchically ordered virtual files that represent particular system resources. The virtual files provide both system-wide and per-process resource statistics. Because the resource statistics are easily accessible form any user-land application, they can be directly accessed from the measuring framework as well. Although, the kernel provides a huge amount of interfaces for monitoring resource usage statistics, we focus on the `/proc` and `/sys` interfaces as they include the most relevant CPU, disk, and memory statistics. The complete description of the system counters is available in the official Linux Documentation [20].

The **`proc`** file-system is a process information interface for accessing the runtime system information. It is commonly mounted at `/proc`. The interface has a tree-like structure that contains directories and files. However, the files at `/proc` are not stored physically on any disk device, but they are in-memory files acting as pointer to where the data is actually stored in the kernel. It is important to note that the presented files may be missing from the reader's system as their presence depends on the kernel configuration and the loaded modules.

The **`proc`** file-system contains a separate directory for each running process, which is named after the process ID. The details of a particular process can be examined by reading the files in the corresponding process directory. A selection of the process specific entries is listed in Table 4.5.

Beyond process-related directories, the **`proc`** file-system contains kernel specific files that represent the kernel data and the system-wide information. A selection of the available files is listed in Table 4.6.

File	Content
`cmdline`	Command line arguments of the process.
`cpu`	Current and last CPU in which it was executed.
`cwd`	Link to the current working directory of the process.
`mem`	Memory held by the process.
`root`	Link to the root directory of the process.
`stat`	Status information about the process.
`pagemap`	Page table of the process.

Table 4.5: Selection of process-specific entries in the `/proc`. Source [21].

Attribute	Content
`cpuinfo`	Collection of CPU and system architecture information.
`diskstats`	Disk I/O statistics for each disk device.
`loadavg`	Load average of last 1, 5 and 15 minutes.
`meminfo`	System's memory usage statistics.
`net`	Information about the networking layer.
`stat`	Overall system statistics.
`uptime`	The uptime of the system and the amount of time spent in idle.

Table 4.6: Selection of system-specific entries in the `/proc`. Source [21].

Table 4.5 and Table 4.6 show that both per-process and system-wide entries provide overall statistics and additional information about numerous system resources. However, in the following we only focus on CPU, disk, and memory usage statistics. More precisely, we focus on the system-wide statistics of the `/proc` interface because the per-process statistics provide limited amount of resource usage information.

The `/proc/stat` Statistics File

The `/proc/stat` virtual file reports kernel activity along with CPU-related statistics. The displayed values are aggregates since the system last booted. Listing 4.5 shows an example of the `/proc/stat` on our test system.

The first line displays the summed CPU statistics, followed by the statistics of the individual CPUs of the system. The values express the amount of time (in hundreds of a second, USER_HZ) the CPU has spent executing various tasks.

```
$ cat /proc/stat
cpu 50294 4515 65077 196476043 26122 16410 11855 0 0 0
cpu0 11851 1243 19318 49123098 7164 1934 1516 0 0 0
cpu1 14730 1238 6550 49132363 3863 6374 1235 0 0 0
cpu2 10935 971 31903 49099522 3559 2641 2998 0 0 0
cpu3 12776 1062 7304 49121059 11534 5459 6104 0 0 0
intr 43588975 16 0 0 0 0 0 0 0 1 983515 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    43749 58 1 253554 253516 253846 250418 250306 254261 246755 249331 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0
ctxt 82552988
btime 1562777237
processes 4934
procs_running 2
procs_blocked 1
softirq 53259869 1 28994326 89150 1276791 44105 0 254 17472695 0 5382547
```

Listing 4.5: Sample of `/proc/stat` during the execution of the sample applcation.

Attribute	**Description**
`user`	Normal processes executing in user mode.
`nice`	Niced processes executing in user mode.
`system`	Processes executing in kernel mode.
`idle`	Twiddling thumbs.
`ioswait`	Waiting for I/O to complete.
`irq`	Servicing interrupts.
`softirq`	Servicing softirqs.
`steal`	Involuntary wait.
`guest`	Running a normal guest.
`guest_nice`	Running a niced guest.

Table 4.7: CPU attributes of the `/proc/stat` file. Source [22].

Table 4.7 describes the CPU attributes that are the most common in all Linux versions.

The second part of the `/proc/stat` file consists of kernel activity statistics. These include attributes such as the total number of interrupts and the context switches. Furthermore, the statistics report the amount of threads running or are ready to run along with the amount of currently blocked processes. For more details, please read the Documentation [22].

Overall, the `/proc/stat` virtual file provides detailed CPU and kernel activity statistics. The CPU statistics are displayed separately for all CPUs, therefore it

helps to determine the number of threads the microservices use. Furthermore, the kernel statistics include valuable attributes that help to characterize the behavior of the microservice.

The `/proc/diskstats` Statistics File

The `/proc/diskstats` provides system-wide I/O statistics of the block devices. Traditionally, the report includes multiple lines, where each line corresponds to one block device. The advantage of displaying statistics of multiple block devices in a single file is to eliminate the number of open/close operation if one monitors the activity of a large number of block devices. Listing 4.6 shows an example of the `/proc/diskstats` on our test system.

```
1 $ cat /proc/diskstats
2    8 0 sda 8371 1424 1981280 398569 62378 96670 12199288 12163164 138 263140
       12580468
3    8 1 sda1 59 0 4280 2858 0 0 0 0 0 2648 2858
4    8 2 sda2 8283 1424 1974880 395412 60618 96670 12199288 12104218 138 235358
       12518359
5  253 0 dm-0 9220 0 1968112 455049 158333 0 12211304 81562704 188 262971
      82043537
6  253 1 dm-1 92 0 4192 2929 0 0 0 0 0 1678 2929
```

Listing 4.6: Sample of `/proc/diskstats` during the execution of the sample applcation.

Field	Description
1	Device major number.
2	Device minor number.
3	Device name.
4	Total amount of reads completed.
5	Total amount of reads merged.
6	Total amount of sectors read.
7	Total amount of milliseconds spent reading.
8	Total amount of writes completed.
9	Total amount of writes merged.
10	Total amount of sectors written.
11	Total amount of milliseconds spent writing.
12	Amount of I/Os currently in progress.
13	Total amount of milliseconds spent doing I/Os.
14	Total amount of weighted time spent doing I/Os (ms).
15	Total amount of discards completed.
16	Total amount of discards merged.
17	Total amount of sectors discarded.
18	Total amount of milliseconds spent discarding.

Table 4.8: Attributes of the `/proc/diskstats` file. Source [23].

Generally, the `/proc/diskstats` file contains 14 attributes. However, since the kernel version 4.18, each line has been extended with 4 additional attributes. Table 4.8 shows the available attributes, including the extended ones.

All attributes between 4-18, except attribute 12, represent a cumulative value since the system last booted. Although, all the attributes are unsigned integers, they are not the same size. Most of the attributes are 64-bit integers, but there are 32-bit integers as well. The applications that read the file should be prepared for this inconsistency, as the 32-bit values can easily overflow on long-lived or disk-intensive systems. Please refer to the manual [23] for identifying the 32-bit and 64-bit attributes.

Overall, the `/proc/diskstats` file is easily accessible and provides detailed information about block devices. Although, the attributes display raw values, they can be simply converted into a particular format required by the statistical processing tool, or into a human-readable format, and therefore simulate the output of well-known disk monitoring tools e.g., `iostat`. Thus, the reported values can be used for characterizing the microservices I/O behavior e.g., sequential vs. random I/O, read-write ratio, etc.

The `/proc/meminfo` Statistics File

The `/proc/meminfo` provides information about system memory utilization and distribution. The report traditionally includes the amount of free and used system memory. In addition, it includes the buffers and the shared memory used by the operating system kernel. Listing 4.7 shows the content of the `/proc/meminfo` on our test system. Similarly to other system-related files, the presented attributes can be slightly different on other systems.

The contents of the `/proc/meminfo` can be divided into high-level and more detailed statistics. The high-level statistics capture the summary of the most common values that provide the user a brief picture about the system's memory utilization. These attributes are presented in Table 4.9. The detailed statistics present the memory distribution among various parts of the operating system. The most often used attributes are described in Table 4.10.

Overall, the `/proc/meminfo` provides a large amount of valuable information about the memory usage of the system. For instance, combined with the disk statistics, it can be determined whether the application reads the data directly from the disk or retrieves them from the memory cache.

The `sysfs` file-system is an interface that provides directory-based structure for kernel statistics. It is commonly mounted at `/sys`. Similarly to the `proc` file-system, the `sysfs` file-system also has a tree-like structure of directories and files that are acting as a pointer to where the data is actually stored in the kernel. The `proc` file-system was originally not intended to provide the system-wide statistics, but the top-level statistics had been added over time. Although, the `sysfs` file-system was originally meant to provide device driver statistics, it has been extended with additional types of statistics.

```
$ cat /proc/meminfo
MemTotal: 32759864 kB
MemFree: 26753136 kB
MemAvailable: 30483984 kB
Buffers: 2116 kB
Cached: 4066516 kB
SwapCached: 0 kB
Active: 3730768 kB
Inactive: 1882620 kB
Active(anon): 1545720 kB
Inactive(anon): 456 kB
Active(file): 2185048 kB
Inactive(file): 1882164 kB
Unevictable: 0 kB
Mlocked: 0 kB
SwapTotal: 0 kB
SwapFree: 0 kB
Dirty: 1506060 kB
Writeback: 255956 kB
AnonPages: 1545156 kB
Mapped: 191272 kB
Shmem: 1020 kB
Slab: 176156 kB
SReclaimable: 65192 kB
SUnreclaim: 110964 kB
KernelStack: 5264 kB
PageTables: 9892 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
WritebackTmp: 0 kB
CommitLimit: 16379932 kB
Committed_AS: 2824060 kB
VmallocTotal: 34359738367 kB
VmallocUsed: 0 kB
VmallocChunk: 0 kB
HardwareCorrupted: 0 kB
AnonHugePages: 0 kB
ShmemHugePages: 0 kB
ShmemPmdMapped: 0 kB
CmaTotal: 0 kB
CmaFree: 0 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
Hugetlb: 0 kB
DirectMap4k: 260512 kB
DirectMap2M: 4800512 kB
DirectMap1G: 28311552 kB
```

Listing 4.7: Sample of `/proc/meminfo` during the execution of the sample applcation.

Attribute	Description
`MemTotal`	Total amount of usable physical memory.
`MemFree`	The amount of physical memory, left unused by the system.
`Buffers`	Temporary storage for raw disk blocks.
`Cached`	In-memory cache for files read from the disk.
`SwapCached`	Memory that is present both in the main memory and in the swap file.

Table 4.9: High-level memory attributes of `/proc/meminfo`. Source [22].

Attribute	Description
`Active`	Memory that has been used more recently and usually not reclaimed unless absolutely necessary.
`Inactive`	Memory which has been less recently used. It is more eligible to be reclaimed for other purposes.
`Dirty`	Memory which is waiting to get written back to the disk.
`Writeback`	Memory which is actively being written back to the disk.
`Slab`	In-kernel data structures cache.
`SwapTotal`	Total amount of swap space available.
`SwapFree`	Memory which has been evicted from RAM, and is temporarily on the disk.
`Mapped`	Files which have been mmaped, such as libraries.
`Shmem`	Total memory used by shared memory (`shmem`) and `tmpfs`.

Table 4.10: Detailed memory attributes of `/proc/meminfo`. Source [22].

File	Content
`block`	Contains one symbolic link for each block device that has been discovered on the system.
`bus`	Contains one subdirectory for each of the bus types in the kernel.
class	Contains a single layer of further subdirectories for each of the device classes that have been registered on the system.
`dev`	Contains two subdirectories `block/` and `char/`, corresponding, respectively, to the block and character devices on the system.
`devices`	Contains a file-systems representation of the kernel device tree.
`firmware`	Contains interfaces for viewing and manipulating firmware-specific objects and attributes.
`fs`	Contains subdirectories for file-systems.

Table 4.11: Selection of entries in `/sys`. Source [24].

Table 4.11 shows a selection of the directories under the `sysfs` file-system on our test system. Each directory contains several files and sub-directories that provide information about a specific area of the system. For instance, `/sys/devices/system/cpu` lists the available CPUs and includes CPU-related information, `/sys/devices/system/memory` contains memory information of the running processes and `/sys/block` lists the block devices.

As we can see, the `/sys` contains numerous files and directories for reporting information and statistics about various resources. However, in the following we focus on the `/sys/block` and `/sys/fs/cgroup` folder as they include statistics that are valuable for us.

The `/sys/block` Directory

The `/sys/block` directory contains a symbolic link for each block device that has been discovered on the system. Traditionally, the local system is installed at block device named as `sda`, therefore `/sys/block/sda` exists and its content is shown in Listing 4.8. Similar to other directories in the `/sys`, the `/sys/block/sda` contains additional files and sub-directories. These entries report disk-statistics (`stat`), list partitions (`sda1` and `sda2`) or enable to read and modify the settings of the block device (`size`, `events_poll_msecs`). As our goal is to monitor the block device's resource usage, we focus on the `stat` file that reports the disk usage statistics.

```
1 alignment_offset  discard_alignment  hidden     queue      sda2  slaves
2 bdi               events             holders    range      sda3  stat
3 capability        events_async       inflight   removable  sda4  subsystem
4 dev               events_poll_msecs  integrity  ro         sda5  trace
5 device            ext_range          power      sda1       size  uevent
```

Listing 4.8: Entries in `/sys/block/sda`.

The `/sys/block/sda/stat` [25] file contains a single line of text consisting of 11 or 15 integer values (see Listing 4.9), depending on the kernel version (15 integers values in kernel version 4.18 and above). Both `/sys/block/sda/stat` and `/proc/diskstats` use the same information source, therefore the displayed

values should not differ. More precisely, the displayed attributes and their order in `/sys/block/sda/stat` are equivalent to the fields 4-14 (or 4-18) in the `/proc/diskstats` (see Table 4.8).

```
1 55569 1892 3589002 240894 17896816 3481525 241751664 2244776984 0 34983505
    2245012978
```

Listing 4.9: Content of `/sys/block/sda/stat`.

Even though, both files report the same statistics, the usage of the `stat` can be advantageous over the `/proc/diskstats` if the goal is to monitor a particular block device. While the `/proc/diskstats` reports statistic of all block devices, the `stat` file is connected to only one device. Therefore, the `stat` file is smaller in size, and it can be read and parsed faster.

Control Groups

Control groups (cgroups) are kernel feature that enable to hierarchically group processes in order to either apply resource usage limitations or monitor their resource usage. The cgroups interface is provided by the kernel and is accessible via the virtual file-system called *cgroupfs*, which is usually mounted at `/sys/fs/cgroup`.

The cgroups feature can be split into two parts. The *cgroup* kernel code handles the grouping of processes and applies the group-level limits and settings to each member of the group.

The second most important kernel component is the *subsystem*, often referred as *resource controllers* or *controllers*. The goal of the subsystem is to modify the processes' behavior in the cgroup, apply resource limitations to them or monitor the cgroup's resource usage. Since the cgroups were introduced, various subsystem have been implemented. For instance, subsystem for limiting the process' CPU time or the total amount of memory the processes in the particular cgroup can allocate.

There can be specified numerous cgroups for one resource controller. The cgroups can be added hierarchically by adding and removing subdirectories in the cgroup file-system. The limits and modifications set on any level of hierarchy are valid for all subdirectories and therefore, for all lower levels in the hierarchy. Based on this property, limits set at higher levels cannot be exceeded on any descendant levels of the hierarchy.

Even though, the concept of cgroups provides a wide range of tools to modify the group's settings and to limit its resource usage, in the following we focus on the property that enables to monitor the resource usage of particular cgroups. Because each Docker-container is represented as a separate cgroup with a unique ID, the container's resource usage can be easily monitored by monitoring the cgroup that matches the container's ID.

Listing 4.10 shows the available resource controllers in our test system. To emphasize that the particular cgroup represents a Docker-container, each of the subsystems contains a directory named `docker`, where each subdirectory represents a running Docker-container. For instance, to access the total CPU usage of a container, whose ID is `d95a6cef3e44`, one should read the `cpuacct.usage` file at `/sys/fs/cgroup/cpu,cpuacct/docker/d95a6cef3e44`. As the Listing 4.10 shows, there are numerous subsystems for all type of resources. However, in

the following we focus only on subsystems related to CPU, disk and memory resources.

```
1  dr-xr-xr-x. 5 root root 0  Jul 10 18:47 blkio
2  lrwxrwxrwx. 1 root root 11 Jul 10 18:47 cpu -> cpu,cpuacct
3  dr-xr-xr-x. 5 root root 0  Jul 10 18:47 cpu,cpuacct
4  lrwxrwxrwx. 1 root root 11 Jul 10 18:47 cpuacct -> cpu,cpuacct
5  dr-xr-xr-x. 3 root root 0  Jul 10 18:47 cpuset
6  dr-xr-xr-x. 5 root root 0  Jul 10 18:47 devices
7  dr-xr-xr-x. 3 root root 0  Jul 10 18:47 freezer
8  dr-xr-xr-x. 3 root root 0  Jul 10 18:47 hugetlb
9  dr-xr-xr-x. 5 root root 0  Jul 10 18:47 memory
10 lrwxrwxrwx. 1 root root 16 Jul 10 18:47 net_cls -> net_cls,net_prio
11 dr-xr-xr-x. 3 root root 0  Jul 10 18:47 net_cls,net_prio
12 lrwxrwxrwx. 1 root root 16 Jul 10 18:47 net_prio -> net_cls,net_prio
13 dr-xr-xr-x. 3 root root 0  Jul 10 18:47 perf_event
14 dr-xr-xr-x. 5 root root 0  Jul 10 18:47 pids
15 dr-xr-xr-x. 6 root root 0  Jul 10 18:47 systemd
16 dr-xr-xr-x. 5 root root 0  Jul 10 18:47 unified
```

Listing 4.10: List of the available subsystems.

CPU-related Subsystems: In our test system, the following subsystems are available for limiting and monitoring the CPU resource usage: `cpu`, `cpu,cpuacct`, `cpuacct` and `cpuset`. However, `cpu` and `cpuacct` are only symbolic links to `cpu,cpuacct`.

The subsystem `cpu,cpuacct` provides accounted CPU usage for the grouped processes. The accounted CPU usage is stored in multiple files, each representing a different perspective. The `cpuacct.stat` file contains statistics of CPU time (in USER_HZ) spent by user-level and system-level tasks. The remaining files prefixed with `cpuacct.usage_` report the CPU time (in nanoseconds) spent by tasks in this cgroup and the descendant groups. The difference between multiple `cpuacct.usage_` files is in the combination of perspectives they display. For instance, the `cpuacct.usage_percpu_user` reports for each CPU the time spent on user tasks, while the `cpuacct.usage_percpu` reports for each CPU the time spent on both user and system tasks.

A more detailed description of the CPU-related files is available in the Documentation [26] and in the Linux manual page [27].

The `cpuset` subsystem [28] assigns a set of CPUs and memory nodes to set of tasks. As it implies from the description, the subsystem does not report any resource usage statistics that could be useful for our measuring framework. Therefore, we skip the description of this subsystem.

Disk-related Subsystems: The `blkio` subsystem implements the resource controller for block devices. It enables to apply various I/O limitations and to monitor the block device's resource usage by writing and reading the appropriate files in the `blkio` directory.

File `blkio.io_service_bytes` enables to monitor the number of bytes transferred from/to the block device by the processes in the particular cgroup (see Listing 4.11). In addition, the transferred data is counted separately for each type of operation (read, write, synchronous and asynchronous). Each line in the

file consists of four columns. The first two columns represent the device's major and minor number. They are followed by the name of the operation and the accounted number of bytes.

```
1 8:0 Read 4763648
2 8:0 Write 524288000
3 8:0 Sync 529051648
4 8:0 Async 0
5 8:0 Total 529051648
6 Total 529051648
```

Listing 4.11: Sample of the `blkio.io_service_bytes` file of the sample Docker-container.

The `blkio.sectors` file is similar to the `blkio.io_service_bytes`, however, it is less detailed. The `blkio.sectors` file identifies each block device with their major and minor number and includes a counter, that is increased every time a sector is transferred from/to the disk device. The amount of transferred data can be calculated by multiplying the number of transferred sectors with the sector size (traditionally 512 B).

Beyond the amount of transferred data, the subsystem provides reports about size of the I/O queue and the time spent by requests waiting in this queue. These statistics are available in `blkio.io_queued` and `blkio.io_wait_time` files.

The previously described files report resource usage statistics of the local cgroup. In order to monitor the descendant cgroups' resource usage statistics, the `blkio` subsystem contains files postfixed with the string `_recursive`. These files show the same information as the non-recursive ones, but the statistics include recursive resource usage of descendant cgroups.

Memory-related Subsystems: The `memory` subsystem is responsible for limiting, modifying, and monitoring the processes' memory, kernel memory, and swap memory. Compared to the CPU-related subsystems, the `memory` subsystem contains a relatively high amount of writable files that enable to apply multiple kind of limitations. However, we focus on the files that report memory usage statistics. One of these files is the `memory.stat`, which is shown in Listing 4.12. When we compare its output with the previously described `/proc/meminfo` file, we can see that they include multiple matching attributes. However, it also includes additional attributes, that are described in detail in the Documentation [29]. Based on its content, the `memory.stat` file can be divided into two parts. The first part (lines 1-17) reports values related to the processes in the local cgroup. In the second part, values prefixed with `total_` (lines 20-36), include the sum of all descendant cgroups' values.

Besides the `memory.stat`, the `memory` subsystem includes additional files reporting memory-related statistics. For instance, the `memory.usage_in_bytes` and the `memory.max_usage_in_bytes` that expresses the processes' current and the maximum memory usage, respectively, during the lifespan of the container.

In our test system, we have maximized the memory usage at 500 MB of the sample Docker-container. The memory limit can be verified inside form the cgroup, by reading the `memory.limit_in_bytes` file. Every time the container tries to exceed this memory limit, a counter is increased in the `memory.failcnt` file and the memory will be reclaimed.

```
cache 7225344
rss 475136
rss_huge 0
shmem 0
mapped_file 2838528
dirty 270336
writeback 135168
swap 0
pgpgin 55407
pgpgout 53543
pgfault 3993
pgmajfault 0
inactive_anon 0
active_anon 548864
inactive_file 2437120
active_file 4767744
unevictable 0
hierarchical_memory_limit 524288000
hierarchical_memsw_limit 1048576000
total_cache 7225344
total_rss 475136
total_rss_huge 0
total_shmem 0
total_mapped_file 2838528
total_dirty 270336
total_writeback 135168
total_swap 0
total_pgpgin 55407
total_pgpgout 53543
total_pgfault 3993
total_pgmajfault 0
total_inactive_anon 0
total_active_anon 548864
total_inactive_file 2437120
total_active_file 4767744
total_unevictable 0
recent_rotated_anon 2285
recent_rotated_file 1356
recent_scanned_anon 2285
recent_scanned_file 54479
```

Listing 4.12: Sample of the `memory.stat` file of the sample Docker-container.

The `memory` subsystem includes additional files prefixed with `memory.kmem` and `memory.memsw` that report the cgroup's memory statistics from the perspective of the kernel memory, the sum of the process memory, and the swap memory, respectively.

Summary of cgroups: The mechanism of control groups enables to limit and monitor the resource usage of a particular Docker-container. Since, each container is represented with a separate cgroup, the monitored resource usage statistics are precise and do not include the resource usage of other containers running in parallel.

The CPU-related cgroup (`cpu,cpuacct`) provides statistics about CPU time spent on the user and the system processes. The statistics are not only displayed as a total of all CPUs, but are further divided for each CPU. However, the CPU time is represented in many ways, this is the only attribute reported by the subsystem. We are missing additional attributes introduced in the Section 4.1.1 e.g., executed instructions, number of interrupts, and context-switches, that would significantly improve the workload characterization.

In contrast with the CPU-related subsystems, the statistics provided by the `blkio` subsystem include a wide range of attributes that cover the most important disk concepts. In addition, the attributes are further divided by the type of the operation (read, write, synchronous, and asynchronous). However, at the time when we were analyzing the `blkio` subsystem, the Docker containerization tool included a bug [30]. This bug prevented the `blkio` subsystem to accurately measure the data transferred from/to the Docker-container, which resulted in invalid statistics. Even though, the bug was fixed in the later version of the Docker, we have had already implemented an alternative solution and collected a significant amount of data that would be incompatible after the fix was applied.

The `memory` subsystem reports detailed statistics about the container's memory usage. The statistics are similarly formatted as in the `/proc/meminfo`, thus it is easy to process the collected data. Out of the presented subsystems, the `memory` subsystem would be an ideal candidate as a data source in the measuring framework.

Summary of `/proc` and `/sys` Interfaces

The presented files in `/proc` and `/sys` interface include all the necessary information to extract the resource usage attributes displayed in various monitoring tools. Therefore, monitoring the counter-based performance counters is fully available for us, and the monitoring tools can be completely replaced with the presented files. However, the available data include only the system-wide resource usage, they can be still useful for the workload characterization and prediction of the execution time. Although, the control cgroups provide more precise, container-focused data that would be ideal for us, their usage must be discarded due to the lack of CPU attributes and the implementation bug in the disk statistics.

Overall, the introduced resource usage monitoring files are ideal alternative of the third-party monitoring tools. The proposed measuring framework can be used to extract the resource usage information directly from the presented pseudo-files. This custom implementation enables us to be independent from the restrictions

of monitoring tools, and to fully utilize the potential of the available resource usage information.

4.2.3 Profiling

Profiling is a principle that analyzes the target by gathering program events. A wide variety of techniques - including hardware interrupts, code instrumentation, operating system hook, and performance counters - can be used for data collection. From our perspective, the performance counters are the most significant source of data, because they provide information about the resource usage pattern. Modern CPUs are equipped with hardware performance counters that use hardware logic to monitor events without the need of any active code. These hardware performance counters are the set of special-purpose registers that can be programmed to count particular hardware evens e.g., amount of CPU-cycles, instructions, interrupts, etc. As the counters are physically located in the CPU, their amount is highly restricted. For instance, our system supports 11 hardware counters.

Depending on the implementation how program events are collected, profilers can be divided into two groups: *statistical* and *event-based*.

Statistical profiling collects information by periodically sampling the target. Sampling is usually executed at fixed rate, e.g., 100 Hz or 1000 Hz, however, it is recommended to avoid sampling in the lockstep as it could lead to over- or under-counting.

Event-based sampling uses non-timed hardware events e.g., cache-misses, branch-instructions to profile programs. Several programming languages and frameworks, for instance Java and .NET, include event-based profilers that provide callbacks to profilers for trapping various events e.g., class-load, class-unload, object creation.

Statistical and Event-based Profilers

In this section we present both statistical and event-based profilers. We show how they work and analyze, and how they could be used for profiling the containerized applications' resource usage e.g., CPU, disk and memory utilization. Table 4.12 includes a selection of statistical and event-based profilers.

Tool	Description
`OProfile`	Open-source statistical profiler and event counting tool for Linux systems.
`perf`	Performance analysis tool and profiler for Linux.
`Intel VTune Amplifier XE`	Commercial application for software performance analysis both for Microsoft Windows and Linux operating systems.
`Microbenchmarking Agent for Java`	Multiplatform Java agent for high-precision microbenchmarking.

Table 4.12: Statistical and event-based profilers.

The `perf` Tool: `perf` is a performance analyzer and profiler tool for Linux that is capable for static and dynamic tracing using tracepoints, kprobes, and uprobes [31]. Furthermore, the tool supports hardware performance counters, tracepoints, and software performance counters. Despite of the wide range of provided possibilities, `perf` is a lightweight tool that imposes low performance overhead to the target.

The `perf` tool is included in the main Linux kernel and can be easily executed by entering the `perf` command in the terminal. The particular action that the tool should execute can be specified as a unique sub-command. Table 4.13 shows a selection of the `perf` subcommands.

Subcommand	Description
`stat`	Run a command and gather performance counter statistics.
`record`	Run a command and record its profile into perf.data.
`report`	Read perf.data (created by perf record) and display the profile.
`annotate`	Read perf.data (created by perf record) and display annotated code.
`sched`	Tool to trace/measure scheduler properties (latencies).
`top`	System profiling tool.

Table 4.13: Selection of `perf` subcommands.

Since, our goal is to collect the microservices' resource usage, which is expressed by the performance counters. In the following, we focus on the `stat` sub-command. Running the `perf` tool using the `stat` sub-command enables statistical profiling of the system, including user- and kernel-space. In addition, it provides per-task, per-CPU, and also per-workload counters.

The basic usage of the `perf stat` is to run a command or attach to an already running process and gather this particular process's resource usage statistics. Using the `-d` option, the output is extended by additional performance counters.

```
1  [root@cirrus-1 ~]# perf stat -dp 23210 sleep 10s
2
3   Performance counter stats for process id '23210':
4
5  | 9742.783285   task-clock (msec)      # 0.974 CPUs utilized
6  |        2119   context-switches       # 0.217 K/sec
7  |          43   cpu-migrations         # 0.004 K/sec
8  |          28   page-faults            # 0.003 K/sec
9  | 34082431474   cycles                 # 3.498 GHz
10 | 45163531937   instructions           # 1.33 insn per cycle
11 |  7751895914   branches               # 795.655 M/sec
12 |   546218815   branch-misses          # 7.05% of all branches
13 | 10830972076   L1-dcache-loads        # 1111.692 M/sec
14 |   659391188   L1-dcache-load-misses  # 6.09% of all L1-dcache hits
15 |    52488316   LLC-loads              # 5.387 M/sec
16 |     6175391   LLC-load-misses        # 11.77% of all LL-cache hits
17
18  10.001135353 seconds time elapsed
```

Listing 4.13: Extended per-process resource usage statistics of the main workload.

Listing 4.13 shows the collected CPU statistics of the main workload from the sample application. The collected data represents a 10-second-long interval. The output includes important metrics, such as the amount of context-switches, CPU-cycles, or processed instructions. Furthermore, the statistics not only display the counted values, but also place them into context. For instance, the statistics include the percentage of branch-misses or the IPC (instructions per cycles), which is the inverse of the CPI (cycles per instructions) introduced in Section 4.1.1.

Another type of usage of the `perf stat` is to collect the system-wide resource usage. This type of collection does not focuses only on a particular process but also measures the CPU usage throughout the system. The system-wide resource usage statistics can be enabled by the `-a` option.

```
[root@cirrus-1 ~]# perf stat -ad sleep 10s

 Performance counter stats for 'system wide':

| 40005.005389   cpu-clock (msec)       # 4.000 CPUs utilized
|        44000   context-switches       # 0.001 M/sec
|          672   cpu-migrations         # 0.017 K/sec
|         8120   page-faults            # 0.203 K/sec
| 105325825796   cycles                 # 2.633 GHz (88.34%)
| 102657350959   instructions           # 0.97 insn per cycle (88.34%)
|  18467676168   branches               # 461.634 M/sec (88.34%)
|   1011724832   branch-misses         # 5.48% of all branches (88.35%)
|  25060521512   L1-dcache-loads       # 626.435 M/sec (85.43%)
|   1880647385   L1-dcache-load-misses  # 7.50% of all L1-dcache hits (85.43%)
|    477752989   LLC-loads             # 11.942 M/sec (85.43%)
|    171217650   LLC-load-misses       # 35.84% of all LL-cache hits (85.42%)


 10.001338462 seconds time elapsed
```

Listing 4.14: Extended system-wide resource usage statistics of the sample applicaion.

Listing 4.14 shows the system-wide CPU statistics, including both main and background workloads of the sample application.

Comparing the per-process and the system-wide results, we can see that they measure the same performance counters, up to one exception. The per-process statistics measure the `task-clock`, which refers to the amount of time that the CPU spent executing the process. The system-wide statistics, however, measure the `cpu-clock` that expresses the total CPU time throughout the available CPUs.

As we can see, the difference between the values of per-process and system-wide counters is significant. While the per-process resource usage statistics present only the resource usage of the main workload, the system-wide resource usage statistics include the resource usage of all three workloads. Therefore, the same peaks and changes in the system-wide scale are less noticeable than in the per-process scale. As a result, the per-process statistics are more suitable for the data analysis and prediction due to the more detailed and precise output.

Besides the default counters, the `perf stat` enables the user to specify the counters to collect. The available counters can be listed by the `perf list` command (see Listing 4.15). It is important to note that the list of counters varies

for each system, as their availability depends on many factors e.g., CPU-type, OS kernel, installed libraries, etc. The particular events can be specified using the `-e` option. Listing 4.16 shows the output of the specified hardware counters.

Moreover, `perf stat` provides support for specifying the counters by their hexadecimal values if their descriptive names do not exist. However, if one would like to examine these less-known counters, the hexadecimal counters can be found in the CPU's manual.

```
$ perf list

   branch-instructions OR branches                    [Hardware event]
   branch-misses                                      [Hardware event]
   bus-cycles                                         [Hardware event]
   cache-misses                                       [Hardware event]
   cache-references                                   [Hardware event]
   cpu-cycles OR cycles                               [Hardware event]
   instructions                                       [Hardware event]
   ref-cycles                                         [Hardware event]
   ...
   L1-dcache-load-misses                              [Hardware cache event]
   L1-dcache-loads                                    [Hardware cache event]
   L1-dcache-stores                                   [Hardware cache event]
   L1-icache-load-misses                              [Hardware cache event]
   LLC-load-misses                                    [Hardware cache event]
   LLC-loads                                          [Hardware cache event]
   LLC-store-misses                                   [Hardware cache event]
   LLC-stores                                         [Hardware cache event]
   branch-load-misses                                 [Hardware cache event]
   branch-loads                                       [Hardware cache event]
   ...
```

Listing 4.15: Selection of the available performance counters.

```
$ perf stat -e cycles,instructions,cache-references,cache-misses,bus-cycles
    tar -czf archive.tar.gz ./file_to_tar

 Performance counter stats for 'tar -czf archive.tar.gz ./file_to_tar':

| 19882069212   cycles
| 48039909027   instructions      # 2.42 insn per cycle
|   289922744   cache-references
|    18889870   cache-misses      # 6.515 % of all cache refs
|   136334345   bus-cycles

 5.380559886 seconds time elapsed

 5.227331000 seconds user
 0.625385000 seconds sys
```

Listing 4.16: Resource usage statistics of the particularly chosen events.

Analysis of the `perf` Tool: Overall, `perf` is a highly accurate low-overhead tool that is ideal for monitoring a wide range of performance counters that can be specified by the user. Beyond providing the system-level statistics, it is capable to monitor a particular process. Therefore, it is ideal for precisely measuring the process's resource usage.

Similar to the `vmstat` and `iostat` tools, `perf` is also an independent application that is executed in parallel with the microservice in order to collect its resource usage. Unfortunately, this independence prevents the tight interconnection between the microservice and the monitoring tool. Furthermore, the `perf` tool's statistics are limited by the CPU's hardware counters. If the user exceeds the number of available counters, the measured values are going to be multiplexed, which leads to incorrect result.

Microbenchmarking Agent for Java

Microbenchmarking Agent for Java (MAFJ) [32] is a high-precision multiplatform library for monitoring per-process resource usage. It uses the Java Virtual Machine Tool Interface (JVMTI) to create software agents enabling to monitor Java applications. The agent's basic features include reporting major JVM[7] events such as GC[8] and JIT[9] runs, collecting performance counters and accurate time through JNI[10]. Furthermore, the agent can collect any event supported by Performance Application Programming Interface (PAPI) [33] if the agent is built on a Linux machine with available `libpapi` package [34]. MAFJ provides a generic interface, which allows the software developer to bind the measurement to a specific thread and its inherited threads. Therefore, the agent can be used to collect detailed information about a particular microservice.

The library is available for download from the project's GitHub page [32]. To be able to use the agent, one will need to compile the source code accordingly to the manual.

```
1  list-events:
2       [java] JVM:compilations
3       [java] PAPI:PAPI_L1_DCM
4       [java] PAPI:PAPI_L1_ICM
5     ...
6       [java] PAPI:perf::INSTRUCTIONS
7       [java] PAPI:perf::PERF_COUNT_HW_CACHE_REFERENCES
8       [java] PAPI:perf::CACHE-REFERENCES
9       [java] PAPI:perf::CACHE-MISSES
10      [java] PAPI:perf::PERF_COUNT_HW_BRANCH_INSTRUCTIONS
11      [java] PAPI:perf::BRANCH-INSTRUCTIONS
12      [java] PAPI:perf::BRANCHES
13      [java] PAPI:perf::BRANCH-MISSES
14      [java] PAPI:perf::BUS-CYCLES
15      [java] PAPI:perf::REF-CYCLES
16    ...
17      [java] SYS:forced-context-switches
18      [java] SYS:thread-time
19      [java] SYS:thread-time-rusage
20      [java] SYS:wallclock-time
21      [java] Found 492 events.
```

Listing 4.17: Selection of supported events.

[7] Java Virtual Machine
[8] Garbage collection
[9] Just-In-Time
[10] Java Native Interface

Similar to the `perf` tool, the MAFJ supports to print the list of events that can be monitored. This can be done either by running the `ListEvents` demo program or executing the `ant list-events` command. Listing 4.17 shows a selection of the available events.

```
private final int LOOPS = 1;
/* Defining the set of events. */
private final String[] EVENT_SET = {
                "PAPI:perf::REF-CYCLES",
                "PAPI:perf::INSTRUCTIONS",
                "PAPI:perf::CACHE-REFERENCES",
                "PAPI:perf::CACHE-MISSES",
                "PAPI:perf::BRANCH-INSTRUCTIONS",
                "PAPI:perf::BRANCH-MISSES",
                "PAPI:PAPI_L1_DCM"};

public void processBenchmark() {
  /* We should have LOOPS measurements and we want to record these EVENTS for
      this thread including the inherited threads.*/
  int id = Measurement.createEventSet(LOOPS, EVENT_SET,
      Measurement.THREAD_INHERIT);

  for (int i = 0; i < LOOPS; i++) {
    /* Start the measurement. */
    Measurement.start(id);

    /* The code to be measured. */

    /* Stop the measurement. */
    Measurement.stop(id);
  }

  /* Get the results (available as Iterable<long[]>). */
  BenchmarkResults results = Measurement.getResults(id);
  /* Either print them in CSV (to be later processed)... */
  BenchmarkResultsPrinter.toCsv(results, System.out);

  /* Freeing the allocated memory. */
  Measurement.destroyEventSet(handle);
}
```

Listing 4.18: Measuring per-process resource usage. Based on [32].

The basic usage of the agent is shown in the Listing 4.18. Being a library, its usage is different from the tools presented so far. In this case, the benchmarking agent is directly referenced from the source code, and functions as a wrapper that encapsulates the measured process. More precisely, to measure the main workload's resource usage from the sample application, the microbenchmarking agent is used for wrapping around the main workload's source code. Because the implementation details are out of the scope of this chapter (see Chapter 8 for more details), we only describe the main principle of the process of measurement that consists of the following steps:

1. Initializing the number of loops to specify the number of repetitions the measurement will be executed.

2. Defining the set of events to be measured.

3. Binding the benchmarking agent to the current thread including its inherited threads, setting the amount of loops and the set of events to measure.

4. Creating the measurement loop, where the target code is enclosed within functions that start and stop the measurement.

5. Collecting the results and printing them to the output.

6. Unbinding the microbenchmarking agent from the measured thread.

Although, we omit the implementation details, it is important to highlight the `Measurement.THREAD_INHERIT` flag. The presence of this flag enables to collect the inherited threads' resource usage, which results in a more detailed and precise measurement.

Analysis of MAFJ: The result of the measurement is shown in Listing 4.19. Each column represents an event from the defined event set (the order is the same as in Listing 4.18), while the rows correspond to iterations. Because the measurement focuses on a particular process, the values present the CPU usage of only the main workload of the sample application. Consequently, the produced data is less noisy, captures more precisely the target's behavior, and therefore it helps to create a more accurate prediction.

```
1  9874508520; 13847251842; 108785100; 4843105; 2215065165; 214881834; 162961634;
2  9899974716; 13847310184; 111069038; 5851337; 2215087977; 215272607; 162301031;
3  9866418222; 13847277284; 106544823; 4588890; 2215073673; 214933800; 162517581;
4  9877898640; 13847282407; 106758957; 4988304; 2215077432; 214950611; 162021795;
5  9866506990; 13847298412; 106347696; 4629453; 2215080971; 214840031; 162690719;
6  9896665626; 13847226738; 108782774; 5402074; 2215061370; 215438720; 162790995;
7  9891971726; 13847252658; 109820639; 5223037; 2215065575; 215514084; 162644033;
8  9870721280; 13847342027; 106387796; 4544514; 2215097397; 215238170; 162579799;
9  9876824080; 13847346109; 105357926; 5011882; 2215096735; 215229326; 162825262;
10 9873285332; 13847259996; 105954467; 4614835; 2215070412; 215328820; 162242379;
11 9905868006; 13847229445; 110599974; 5682945; 2215059715; 215587201; 163204394;
12 9881382492; 13847326454; 109841290; 5000903; 2215092192; 214910547; 162607744;
13 9860877668; 13847348001; 103653493; 4500625; 2215097094; 214896044; 162360512;
14 9876247088; 13847260915; 106374337; 4895446; 2215070364; 214960674; 162141149;
15 9883302538; 13847297436; 105827740; 5015335; 2215081354; 215255363; 162763371;
16 9869380562; 13847291112; 104405368; 4751866; 2215080247; 214952893; 161866045;
17 9868724584; 13847271794; 104940728; 4939874; 2215071785; 214891550; 162218421;
18 9852935560; 13847311267; 101869403; 4450543; 2215087045; 214758622; 162185580;
19 9867942024; 13847249721; 104443911; 4850647; 2215064394; 214945984; 162419363;
20 9873521122; 13847351100; 108002658; 4875771; 2215099943; 214998276; 162808112;
21 9854830786; 13847343723; 102226972; 4587510; 2215095986; 214746630; 163405992;
```

Listing 4.19: Maasuring per-process resource usage. Based on [32].

As we can conclude from the description, the MAFJ is a low-overhead agent that supports our main requirements. The library provides an easy-to-use measuring interface, which can be directly bound to the target process, and therefore enables to accurately measure a wide-range of supported events.

Although it is an effective tool, it faces with similar problem as the previously presented `perf` tool: the amount of hardware counters is physically limited by the CPU. This forces the user to **wisely** select the appropriate resource counters to measure.

4.2.4 Tracing

Performance analysis focuses on the summary of discreet events that operate the system. However, the important details are often hidden in individual events, therefore, it is important to be able to inspect each event individually. Often, performance issues can be solved by analyzing the attributes of the inspected events, e.g., time-stamp to find out where the high latency comes from.

Tracing is a principle of the data collection that collects per-event data for analysis. In contrast with counters, tracing is not enabled by default, because it imposes a relatively high performance overhead, which could affect the behavior of the measured microservice. In addition, frequent events generate increasing amount of data that requires large storage.

A well-known implementation of tracing is the system logging that is traditionally enabled by default on all operating systems. Logging is a low-overhead type of tracing that focuses on less frequent events, like warnings and errors.

Tracing can be divided into two subtypes: *static* and *dynamic* tracing. Static tracing traces a small set of instrumentation points located in the OS kernel and other software. However, it provides only limited visibility of particular events. Dynamic tracing enables to place instrumentation point anywhere in the system, which produces a more detailed view of the events. Therefore, it provides data for in-depth analysis that helps to better understand the system behavior.

Even-though, tracing is ideal for both system-wide and per-process analysis, the underlying principle and the provided dataset is not suitable for our case. Our goal is to find a monitoring mechanism producing such an output that can be used to characterize the overall behavior of the submitted microservice. Because tracing focuses on particular events inside the given software, the type of the produced data is not sufficient to characterize the overall behavior of the microservice. Furthermore, instrumenting and analyzing the microservices in detail is contrary to the concept of handling microservices as black-boxes.

4.3 Summary

From the description of various monitoring tools, it is apparent that each tool is designed for a particular area: in some cases the system-wide monitoring is preferred over the per-process, or counters are preferred over profiling. Thus, there is no universal solution that would be suitable for every situation. In order to be able to monitor the attributes of the system resources, we decided to combine the observability types and the methodologies.

Within the counters methodology, we analyzed the presented third-party monitoring tools and the pseudo-files provided by the `/proc` and the `/sys` interface. As a result of the analysis, we decided to select the pseudo-files. Their advantage resides in the ability that enables any user-land application to read the content of the files. This simplifies their integration into the measuring framework. Furthermore, the provided data can be easily processed and transformed into any kind of representation that will be further required by the prediction process. The disadvantage of the presented statistics files is in reporting system-wide resource usage, therefore the collected data will not be precise and focused on a particular process.

Next, we analyzed the event-based profiling. After comparing and analyzing the `perf` profiling tool and the MAFJ microbenchmarking agent, we decided to select the latter one. Due to its architecture, the agent can be easily integrated into the measuring framework. Moreover, the agent supports monitoring various events of the selected thread including its inherited threads. This ensures a precise measurement that focuses only on the selected microservice. The weakness of the microbenchmarking agent resides in the CPU hardware limitations that restrict the number of hardware counters that can be used to collect the hardware events.

In conclusion, the selected tools include both system-wide and per-process observability, and both counters and event-based profiling methodology. This combination of various observability types and methodologies results in a high-quality dataset that is required for characterizing the microservices.

5. Deployment Framework

In this chapter, we focus on the description of deployment framework that is responsible for executing the submission process in order to assess the microservice resource usage and to determine if it can be admitted to the cloud. The submission process is a sequence of predefined, but configurable steps that support the following concepts:

- **Automation:** Starts assessing the microservices at the time of submission. The result is a faster submission process, compared to manual execution. The automation is possible thanks to the predefined steps, which are configurable by their input parameters.
- **Repeatability:** Due to the fact that the submission process consists of fixed steps, and is executed on the same hardware, the results are comparable to each other and are reproducible at any time.
- **Distributed measurement:** Since the steps are fixed and the hardware is the same for all agents, the measurement of various workloads can be easily distributed among the measuring agents.
- **Support for black-boxiness:** The predefined steps enable to design a common microservice interface. This interface is then used by a performance assessment framework to execute and measure the microservices without needing to know concrete implementation of the microservice.

The concept of solution is already shown in Section 3.3. However, in this chapter, we provide a more detailed view of the objects and main steps in the submission process. As shown in Figure 5.1, the submission process includes one actor, three different object types, and three main steps. The main steps are grouped into reference fragments to make the sequence diagram more compact, and to give the reader a better understanding about the main phases of the submission process. In the following sections, we show a more detailed sequence diagram of each phase and explain the main steps.

5.1 Task Generation

This section is split into multiple parts, as shown in Figure 5.1. The first part presents the detailed steps of the *Task generation* phase. The second part demonstrates samples of the requirements specification and the batch configuration files, and explains their structure. The third part of this section presents how the requirements specifications are submitted into the system. The next part presents how the agents are managed, how the submission tasks are distributed among them, and how the measurement process can be monitored.

5.1.1 Main Steps

The detailed steps of the *Task Generation* phase are shown in Figure 5.2. First, the microservice developer connects to the orchestrator and submits the require-

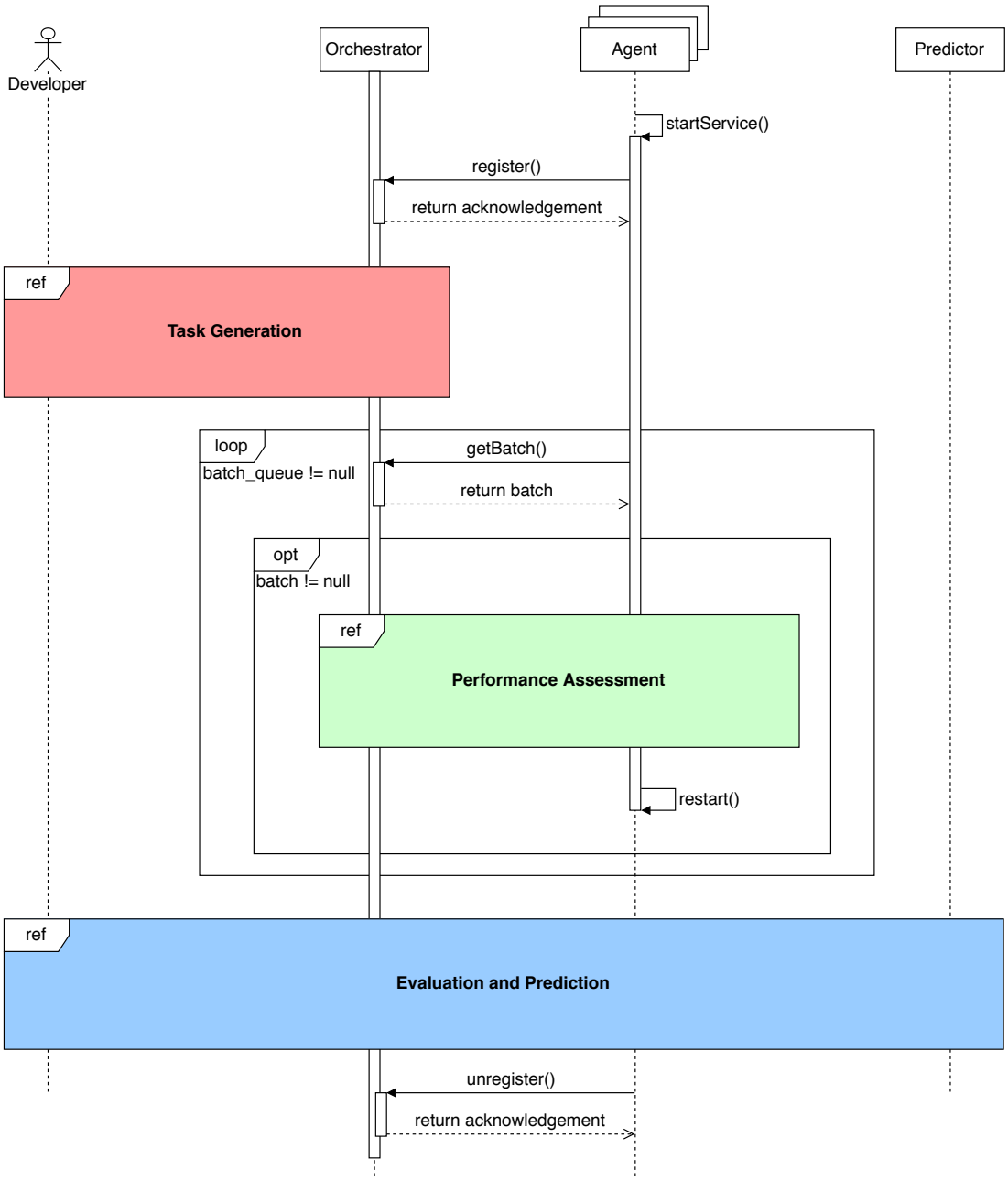


Figure 5.1: High-level view of the submission process.

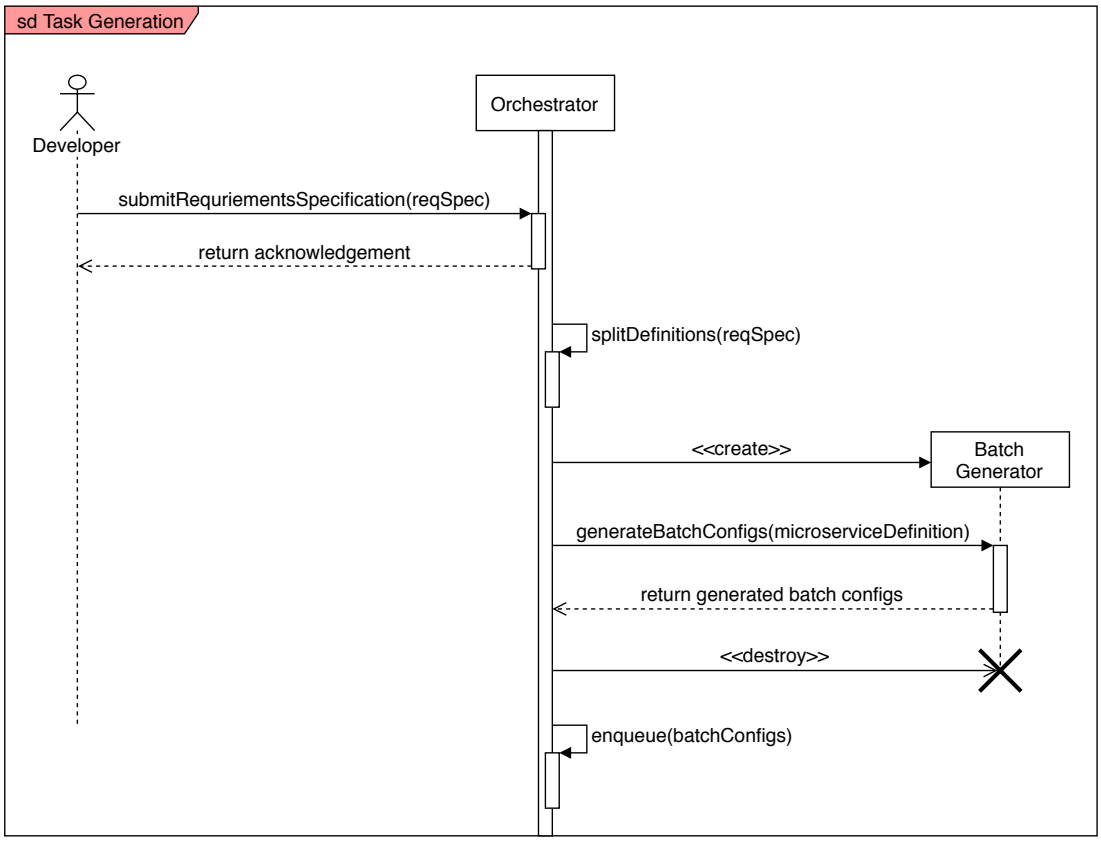


Figure 5.2: Detailed view of Task Generation phase.

ments specification. Then the orchestrator parses the requirements specification using the following steps:

- The orchestrator splits the requirements specification into microservice definition and execution time requirements.
- The orchestrator generates *batch configurations* from the microservice definition. The batch configurations are generated by a specific tool, the *Batch Generator*. The Batch Generator accepts the microservice definition and uses it to generate batch configurations. Batch configurations include multiple *batches* that represent a particular workload combination.
- The orchestrator parses the batch configurations into distinct batches and loads them into the execution queue. Afterwards, the batches are waiting there until they are not assigned to an agent asking for a new batch.

5.1.2 Requirements Specification and Batch Configuration

The requirements specification is an input file submitted by the microservice developer. It contains the definition of the microservice and the requirements of their execution time. However, the microservice definition specifies the attributes of the microservice, it misses other attributes required by the measuring agents. Therefore, the microservice definition is further processed by the Batch

```yaml
requirements_specification:
  - specification:
      microservice:
        microservice_id: AVL
        microservice_name: avl_tree
        microservice_class_name: com.microservices.trees.Microservice
        microservice_class_path: ./microservices/microservice_trees.jar
        microservice_params: avl
      requirements:
        - time: 10000
          probability: 100
        - time: 9500
          probability: 70
  - specification:
      microservice:
        microservice_id: A
        microservice_name: sunflow
        microservice_class_name: com.microservices.scalabench.Microservice
        microservice_class_path: ./microservices/microservice_scalabench.jar
        microservice_params: sunflow --no-validation
      requirements:
        - time: 8000
          probability: 90
        - time: 6500
          probability: 75
```

Listing 5.1: Example of requirements specification.

Generator, which uses it to generate batch configuration. The batch configuration precisely defines all attributes of the measurement process, and therefore it can serve as an input for the measuring agents.

The requirements specification and batch configuration files are written in YAML data-serialization language, because it is easy to read and write for any microservice developer. Furthermore, thanks to its popularity, there exists a significant amount of libraries for well-known programming languages that provide a simple way for parsing and generating YAML files.

Requirements Specification

Listing 5.1 shows an example of the requirements specification file. The requirements specification file includes one or more `specification` nodes. Each node represents a combination of a microservice definition and the corresponding execution time requirements.

Definition of microservice. Contains attributes essential for executing the microservices. These include the ID and the name of the microservice that help to identify the microservice. Furthermore, the definition includes the name of the test-point, path to the executable file and the command-line arguments.

Definition of requirements. Contains attributes that define the requirements over the microservice execution time. There can be many requirements assigned to one microservice. Each requirement includes `time` and `probability` attributes. The `time` attribute defines the maximum length of the microservice execution time in milliseconds. The `probability` attribute defines the percent-

age of the cases where the execution time will be less than or equal to the time specified in the `time` attribute.

Batch Configuration

Listing 5.2 shows an example of the batch configuration file. In this section we explain the idea of each segment. For detailed description of each attribute please see Appendix A.1.

Definition of batch: The batch configuration file is composed of `batch` nodes. Each batch node represents a set of workloads that will be assigned to one agent. For each batch node, it is mandatory to include at least one workload– the main workload. The main workload is the microservice itself, that is intended to be measured. For more information about the workloads, please see Section 5.2.

Definition of tasks: Each batch node includes one or more task definitions, which are essential for assessing the microservice's behavior. We distinguish three types of tasks: *initial*, *microservice* and *final* task (denoted as `init_task`, `microservice_tasks` and `final_task`, respectively). The presence of initial and final tasks are optional, however, the microservice task must be included in each batch node.

The concept of initial task is to prepare the environment where the microservice will be executed. In this particular case, the initial task is responsible for building the Docker image, including the assets e.g., libraries, runtime environment, input resources, etc. required to execute and measure the microservice.

The initial task definition is followed by the definition of microservices. As the name implies, it defines the attributes of the microservices that will be executed and measured. The attributes are gathered into multiple logical groups according to their domain.

The first group contains attributes essential for executing the microservices (lines 12-14). These attributes are already covered in the description of requirements specification. The next group of attributes (lines 15-17) focus on the settings of the measuring framework. The attributes enable or disable the data collection, set whether the current workload is the main one and determine the duration of the measurement. The final group comprises the environment-related attributes (lines 18-28). The attributes listed here determine both the container-related settings e.g., name of the container or the container image tag, and the required constraints e.g., container's hardware limitations, JVM's software limitations.

The task definitions end with the definition of the final task. In our case, the final task ensures that the collected data is compressed into a zip archive and is sent to the orchestrator.

Although, the example shows a specific realization of the initial and the final tasks, the user is free to change their content. The tasks are allowed to include any kind of other attributes, however, the `id` and the `def_type` attributes are mandatory. The value of the `def_type` attribute defines the task definition type, which determines the type of the parser required by the agent to parse the definition. Therefore, it is allowed to change the content of the initial or the final task definition, however, it requires to implement a new parser that is able to parse the modified definition.

```yaml
configs:
- batch: batch--A-F_gc_no_heap_cpu1
  init_task:
    id: BUILD
    def_type: build
    dockerfile_path: /image_definitions/
    image_tag: microservice_image
  microservice_tasks:
  - id: MS0
    def_type: microservice
    microservice_name: sunflow
    microservice_class_name: com.microservices.scalabench.Microservice
    microservice_class_path: ./microservices/microservice_scalabench.jar
    microservice_params: sunflow --no-validation
    collect_data: true
    main: true
    run_duration: 12m
    container: maincont
    image: microservice_image
    docker:
      options:
      - option: --cpus=1
    java:
      id: gc_no_heap
      options:
      - option: -XX:+PrintGCTimeStamps
      - option: -Xloggc:/tmp/gc.log
      - option: -XX:+PrintGCDetails
  - id: MS1
    def_type: microservice
    microservice_name: h2
    microservice_class_name: com.microservices.scalabench.Microservice
    microservice_class_path: ./microservices/microservice_scalabench.jar
    microservice_params: h2 --size large --no-validation
    collect_data: false
    main: false
    run_duration: 100m
    container: h2cont1
    image: microservice_image
    docker:
      options:
      - option: --cpus=1
    java:
      id: gc_no_heap
      options:
      - option: -XX:+PrintGCTimeStamps
      - option: -Xloggc:/tmp/gc.log
      - option: -XX:+PrintGCDetails
  final_task:
    id: SEND
    def_type: send
  commands:
    init_command: BUILD
    microservice_commands: MS0 MS1
    final_command: SEND
  runs: 5
```

Listing 5.2: Example of batch configuration.

Definition of commands: Task definitions are followed by the definitions of commands. The command definition is an assignment, that assigns the task definition to the corresponding command. Similar to task definitions, there are three types of command definitions: `init_command`, `microservice_commands` and `final_command`. For the `init_command` and the `final_command`, the user can assign one task definition at most. If the user does not assign any initial or final task, the `null` definition should be assigned to the corresponding command definition. For the `microservice_commands`, it is mandatory to assign at least one microservice task definition. Furthermore, it is allowed to assign multiple microservice definitions, since the microservices are executed in parallel within the `microservice_commands`.

The commands are executed sequentially, in a predefined order. Initially, the initial task is executed. If the task finishes successfully, it is followed by the microservice tasks. At the end, the final task is executed. Whenever a task fails, the submission process terminates immediately and the remaining tasks will not be executed.

Definition of the number of runs: The batch configuration is terminated with the `run` key. The key defines the amount of iterations the batch and the defined tasks will be repeatedly executed. For more details about repeated execution, please see Section 5.2.

5.1.3 Orchestrator

The orchestrator ensures the communication among the microservice developer, the measuring agents, and the predictor.

In order to be able to exchange up-to-date information during the communication, the orchestrator has to supervise the state of every batch and the connected agents within the deployment framework. Therefore, the orchestrator is responsible for managing the queue of available batches and the measuring agents. Furthermore, the orchestrator controls the distribution of the batches and their assignment to measuring agents.

To prevent collisions during batch distribution among the agents, the orchestrator applies the following rules when assigning batches to agents. Each batch can be assigned to one agent at once. After a batch is assigned to an agent, all the subsequent iterations of that batch will be executed on the same agent. If any iteration of the measurement fails, the batch is marked as failed and will not be assigned to any agent anymore. Besides assigning the batches to agents, the orchestrator also provides the ability to stop the execution of a running batch forcefully. If the batch is stopped forcefully by the orchestrator, it is marked as waiting and is reset to the initial state. Later, the batch can be assigned to any agent that asks for a new batch to execute.

By default, the orchestrator communicates with the developer and the measuring agents via REST API. Besides that, our implementation provides an optional command-line user interface for the developer to interact with the orchestrator. Listing 5.3 and Listing 5.4 show sample usages of the command-line interface, while the REST API is described in more detail in Chapter 8.

The command-line-like user interface makes it simple to monitor the current state of the batches and the measuring agents. Furthermore, the user inter-

face provides a simplified management tool that enables the user to manage the batches and the agents. For instance, the management tool supports adding and removing batches, or adding batches into groups and assigning these groups to a particular set of agents that execute batches only from that group. Regarding the management of the agents, the batch execution process can be suspended and later resumed or stopped forcefully. If the batch execution process is stopped forcefully, the agent does not ask for more batches.

Besides monitoring and managing the batch execution, the orchestrator also communicates with the predictor and manages the evaluation of the collected data. After the execution of the submitted batches finishes, the orchestrator is responsible for passing the collected dataset along with the submitted time requirements to the predictor and propagating the result of the prediction to the microservice developer.

```
1  orchestrator >lsb
2  singles
3  ------------------------------------------------------------
4  batch--CYPHERD_gc_no_heap_cpu1   BatchState.RUNNING  2 of 5
5  batch--AVL_gc_no_heap_cpu1       BatchState.FAILED   0 of 5
6  batch--RB_gc_no_heap_cpu1        BatchState.RUNNING  3 of 5
7  batch--FLOYD_gc_no_heap_cpu1     BatchState.WAITING  0 of 5
8  batch--ROD_gc_no_heap_cpu1       BatchState.WAITING  0 of 5
9  batch--EGG_gc_no_heap_cpu1       BatchState.WAITING  0 of 5
10 batch--FACE_gc_no_heap_cpu1      BatchState.WAITING  0 of 5
11
12 doubles
13 ------------------------------------------------------------
14 batch--FLOYD-AVL_gc_no_heap_cpu1     BatchState.RUNNING  1 of 5
15 batch--ROD-SORTD_gc_no_heap_cpu1     BatchState.RUNNING  4 of 5
16 batch--EGG-CYPHERD_gc_no_heap_cpu1   BatchState.WAITING  0 of 5
17 batch--FACE-PDFD_gc_no_heap_cpu1     BatchState.WAITING  0 of 5
18 batch--FACE-RB_gc_no_heap_cpu1       BatchState.WAITING  0 of 5
19
20 triples
21 ------------------------------------------------------------
22 The batch queue is empty!
```

Listing 5.3: Grouped batches.

```
1 orchestrator >lsa
2 cirrus-2  singles  10.10.54.3:50000  batch--CYPHERD_gc_no_heap_cpu1    running
3 cirrus-3  singles  10.10.54.4:50000  batch--RB_gc_no_heap_cpu1      restarting
4 cirrus-4  doubles  10.10.54.5:50000  batch--FLOYD-AVL_gc_no_heap_cpu1  running
5 cirrus-5  doubles  10.10.54.6:50000  batch--ROD-SORTD_gc_no_heap_cpu1  running
6 cirrus-6  triples  10.10.54.7:50000  No batch                          running
7 cirrus-7  triples  10.10.54.8:50000  No batch                         stand_by
8 cirrus-8  triples  10.10.54.9:50000  No batch                         stand_by
```

Listing 5.4: List of available agents and the assigned batches.

5.2 Measurement Process

5.2.1 Prerequisites

Before we present the measurement process in more detail, we first describe the most important properties and design decisions that we have made.

Workload Combinations

To make our measurement and prediction that reflect the real-life conditions precisely, we are going to execute the submitted microservice in a single machine with various colocated workloads. Therefore, we design our test scenarios in such a way, that they measure the microservice not only as a single instance running on a particular server, but also with a combination of multiple microservices. The selection process of the combined microservices is covered in Chapter 7.

For easier orientation, we divide the microservices into two groups during the measurement process. The first group includes the submitted microservice only. We are going to refer this microservice as the *main* microservice. The second group includes the other microservices running on the same machine. We are going to refer this group of microservices as *background* microservices. The naming also expresses the importance of the microservices. Even though, both the main and the background microservices will be executed in parallel, the data collection will only be focused on the main workload.

Repetitive Measurement

One of the most straightforward way to improve the predictor's accuracy is to provide a large dataset that covers a wide application domain. This large dataset can be built from repeated execution of the submitted microservice that is exercised both alone and with various combinations of colocated workloads. The repetition of execution enables to compensate the short execution time of the microservices (up to 12 seconds in our case), therefore it helps to eliminate the effect of one-off events i.e., running service in the background, network event, etc. and runtime environment peculiarities occurring during the measurement process. The more we understand the runtime environment's behavior, the more we can fine-tune the data collection process, which results in a more consistent dataset. In the following sections, we present the issues occurred during the tuning process and the solutions we provided for them.

Warm-up: The first issue associated with the runtime environment is related to the Java application achieving its steady state. The application performance is often influenced at the startup by various one-off events that slow down the execution and prevent the application from providing sustainable performance. In order to accomplish the steady state, these one-offs need to be eliminated during the so-called *warm-up* phase. In Java-based performance experiments, the warm-up phase includes two major aspects: class loading and just-in-time compilation [35]. The warm-up process can be enhanced by various configurations that enable better optimization. However, a common criterion for a warmed-up application is a sufficiently high number of iterations the particular performance-aware method was invoked.

Figure 5.3: Iteration-restricted measurement process.

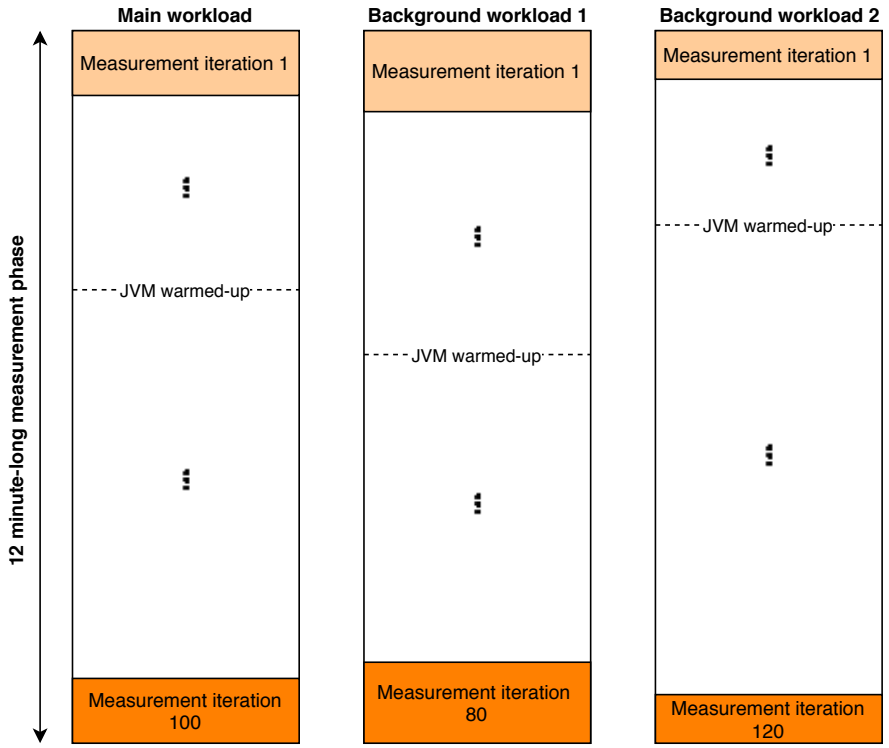


Figure 5.4: Time-restricted measurement process.

In this thesis, we refer to the steady application state as the application is *warmed-up*, and we refer to the state where the application does not provide sustainable performance as the application is *cold-out*.

During the design and development of the thesis, we have experimented with various implementations for keeping the application warmed-up.

In our first attempt, we separated the microservice measurement process into two phases (*warm-up phase* and *measurement phase*) and executed a predefined amount of iterations in both phases (see Figure 5.3). The original idea was to execute the phases in parallel in case of colocated workloads - do not start the measurement phase until the warm-up phase is not finished on all workloads. The only difference between the two phases is the enabled data collection in the measurement phase for the main workload. Based on the results of our analysis, we have decided to execute 30 iterations in warm-up phase and 50 iterations in measurement phase for both the main workload and the background workloads. This concept worked well for workloads with matching execution time. However, when the execution time of workloads did not match, the faster microservices (shorter execution time) were suspended after their warm-up phase and were kept waiting for the slower microservices (longer execution time) to finish. This caused the faster microservices to cold out and therefore, the first few iterations of the measurement phase provided unstable results.

Due to these issues, we have decided to focus on the total length of the execution instead of the prescribed amount of iterations (see Figure 5.4). This implementation does not distinguish the warm-up and the measurement phases of the microservices, but considers the entire execution as a single measurement phase, which is only limited by the execution time. Therefore, the data collection of the main microservice is active since the beginning of the execution. Based on the experiments that we have carried out with various lengths of the execution time, we have decided to set the execution length to 12 minutes. This time interval is long enough to collect multiple iterations of long-running microservices. For instance, it gives 60 iterations of a 12 second-long microservice.

The latter solution (time-restricted) offers several advantages over the former one (iteration-restricted). Thanks to the continuous data collection started at the beginning of the execution, we can use the collected data to determine when the application is already warmed-up. This allows us to dynamically select the amount of iterations that are used for setting up the model in the prediction process. The dynamic data selection is a big benefit compared to the iteration-restricted solution, were the amount of collected data was previously fixed.

Restarting the Measuring Agent: Modern computers consists of highly complex hardware and software components. Execution of a non-trivial operation can be affected by various factors. We distinguish frequently occurring, but less significant factors caused by the non-determinism in the execution, and rarely occurring but more significant factors caused by external events. There are factors e.g., internal memory cache that can be changed by the operation execution, referred as *mutating* sections of the state. Other factors, such as the CPU clock speed, are unchangeable by the operation execution and are referred as *initial* sections of the state.

In order to make the measurement reproducible, both the microservice and the measuring agent should be set to a well-defined state. Achieving the same state of

```
...;write_time_in_ms;...;weighted_io_time;...
...
...;4269679575;.........;4294451153;...
...;4269819695;.........;4294595941;...
...;4269972419;.........;4294739535;...
...;4270119238;.........;4294884904;...
...;4270270110;.........;60492;...
...;4270409142;.........;204908;...
...;4270555015;.........;348823;...
...;4270717003;.........;490014;...
...;4270835850;.........;633594;...
...;4270971714;.........;774175;...
...
...;4294491533;.........;24284485;...
...;4294632314;.........;24427116;...
...;4294762449;.........;24571132;...
...;4294921046;.........;24715064;...
...;89337;..............;24857697;...
...;240941;.............;25001225;...
...;371935;.............;25144577;...
...;526735;.............;25287425;...
...;670760;.............;25431815;...
...;811740;.............;25574457;...
...
```

Listing 5.5: Overlow of counters in `/proc/diskstats`.

the mutating parts can be done by executing warm-up steps of the microservice. Furthermore, setting the initial parts of the state can be precisely defined by a particular system configuration. However, as it is shown in the experiment using the Fast Fourier Transform in [36], there is no guarantee for setting the same initial state of the microservice for each repetition of the measurement. Therefore, we have to consider the slightly random initial state for each measurement.

In order to analyze how the random initial state affects the measured data, we have designed and conducted the following experiment. To carry out the experiment, we have measured all the microservices that have been selected in Chapter 6. Changes of the initial state were enforced by restarting the measuring agents. During the experiment, each measurement was 5-times subsequently repeated on the same machine of the available agents, in two different sessions.

In the first session, agents were not restarted between the 5 subsequent measurements. However, in the second session the agents were restarted before the first measurement and after all subsequent measurements. Then, we compared the length of the execution times of the microservices from both sessions. The results are shown in Figure 5.5. The x-axis shows the names of the microservices while the y-axis shows the execution time in milliseconds. As shown in the figure, there is no significant difference between the execution times, however, the measurements separated by restarts performed slightly better, providing less noisy results. Therefore, we can conclude that it is recommended to restart the agents between the subsequent measurements.

Besides improving the measurement quality, restarting the agent might help to prevent some other issues as well. While we have been continuously evaluating the collected data, we have noticed a sudden significant drop in the

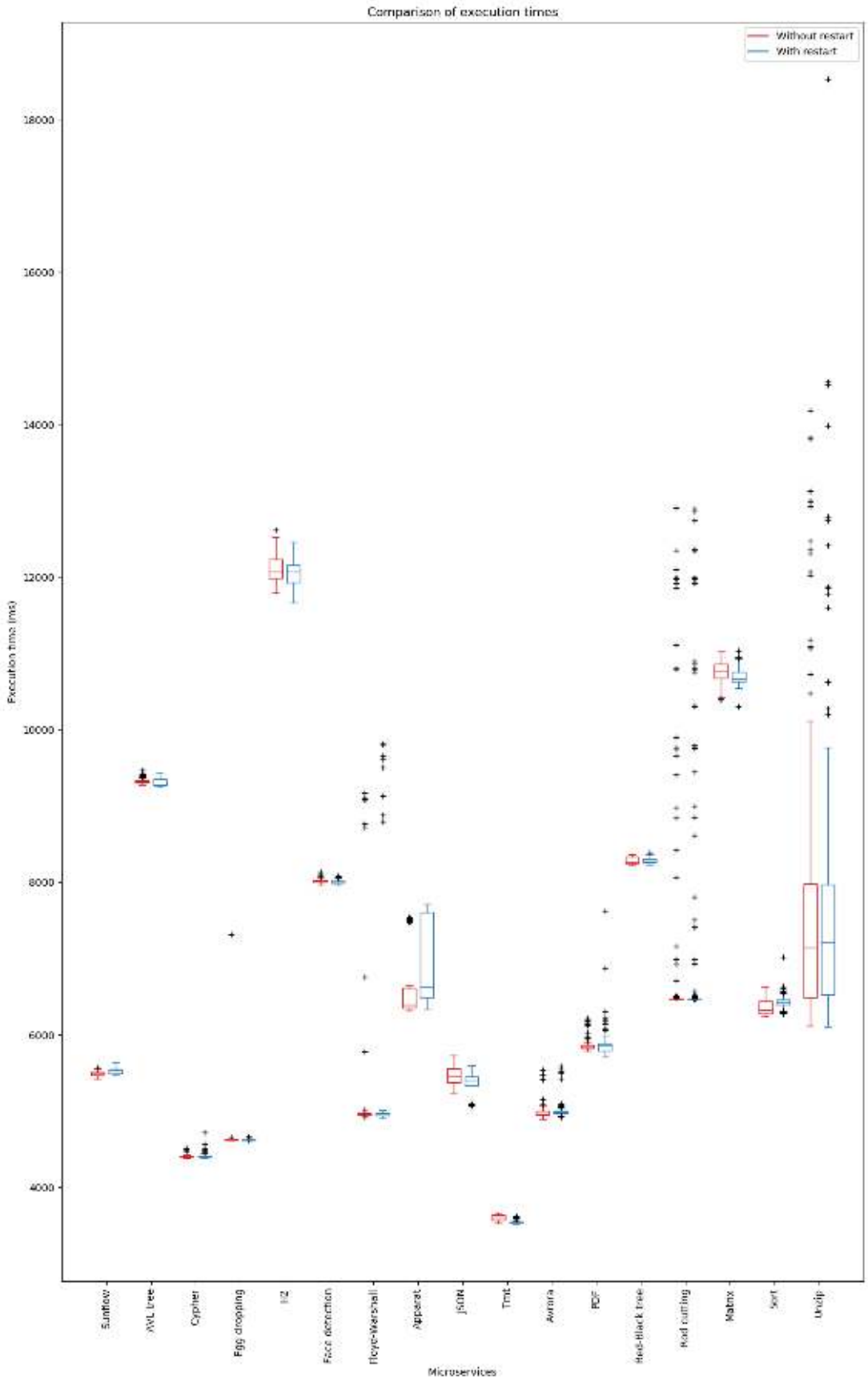


Figure 5.5: Comparison of execution times in two different measurement scenarios.

measured values. During the execution of highly disk-intensive microservices some of the counters (`Field 8 -- number of milliseconds spent writing` and `Field 10 -- weighted amount of milliseconds spent doing I/Os`) in the `/proc/diskstats` file exceeded their limit ($2^{32} - 1$) and overflew, which reset the counter to 0. A snippet of the affected dataset is shown in Listing 5.5.

To prevent the counter from overflow and to avoid the inconsistency in the dataset, the total amount of transferred data should be kept below a certain limit. In our case, the overflow happened after approx. 17.84 TB data has been written to the disk, which is a result of uptime of several months. Resetting the counters in the `/proc/diskstats` file can only be done by restarting the measuring agent. If the restarts are executed after every measurement i.e., in every 12 minutes, it is guaranteed that the total amount of transferred data at maximum transfer speed (approx. 72 GB at 100 MB/s continuous read/write speed) will not exceed the limit and therefore the counters will not overflow.

5.2.2 Performance Assessment

In this section, we focus on the second phase of the submission process, the *Performance Assessment* phase that is depicted in more detail in Figure 5.6. This detailed sequence diagram shows the sequence of steps among the objects that take part in the measurement process and are responsible for repetitive execution and data collection of the microservice.

The measurement process consists of sequence of predefined but configurable steps. The predefined steps ensure the same execution of the measurement tasks, regardless of the current instance of the agent they are running on. Furthermore, the unified measurement process guarantees the comparability of the measured data irrespective of the measuring agent that collected the data. Configurability of the predefined steps helps to parametrize the submitted microservices and enables to apply various hardware and software constraints accordingly to the batch configuration.

From our perspective, the most important steps in the measurement process are the ones that execute and measure the submitted microservice. Besides the measurement steps, there are multiple essential steps as well, which are responsible for setting up the measuring environment and managing the measurement process.

From Figure 5.6, it is clear that the measurement process can be divided into three parts, as there are three objects that take part in the measurement process. Therefore, we first describe what each object is responsible for, and afterwards we present the steps of the measurement process. The objects of the Performance Assessment phase are the following:

- **Agent:** The agent manages the execution of the command definitions from the received batch configuration file. Furthermore, it monitors the state of the execution process and reports the execution result to the orchestrator.
- **Framework:** The measuring framework is responsible for loading the microservice and running data collection during its execution. The framework runs in a virtual environment that is provided by a Docker container.

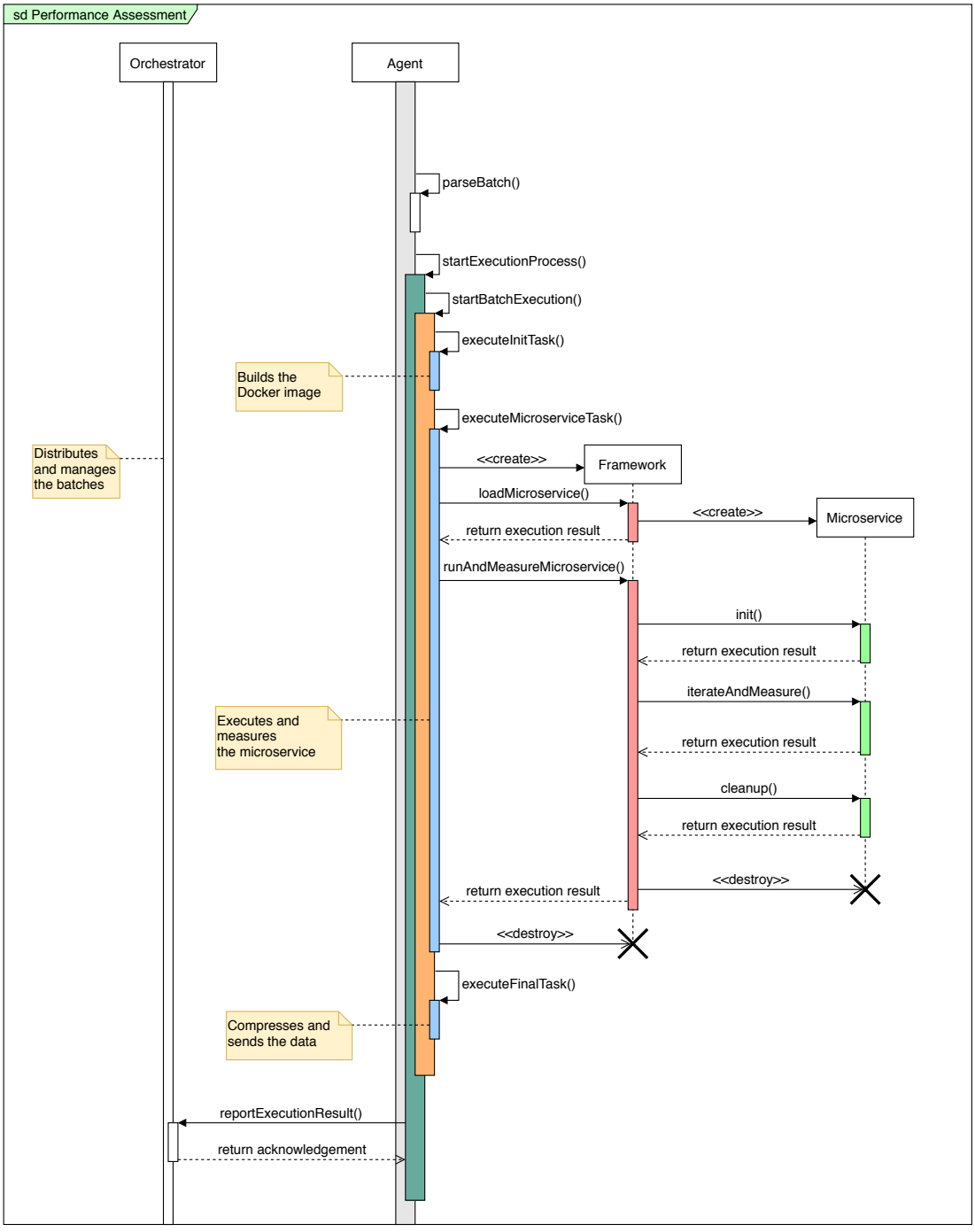


Figure 5.6: Detailed view of Performance Assessment phase.

- **Microservice:** The microservice itself that is submitted for measurement. The microservice implements the required interface (see Section 6.2) that enables the framework to load the microservice and communicate with it.

Before the Performance Assessment phase begins, the orchestrator and the agent are required to execute the following sequence of steps in order to achieve a particular state. Afterwards, the Performance Assessment phase can be started.

Both the orchestrator and the agent applications are running on the appropriate server machines. The agent connects to the orchestrator and registers itself as an active agent. Afterwards, the agent starts an infinity loop, in which the agent starts querying the orchestrator for new batches. Meanwhile, the microservice developer submits the requirements specification to the orchestrator, which parses the input into separate batches. If the returned batch is `null`, then the agent sleeps the loop for a particular amount of time (10 minutes in our case). If the returned batch is not `null`, then the phase starts as it is shown in the Performance Assessment fragment.

Steps of the Agent:

- Parse batch: The agent parses the batch and converts the batch tasks into an internal representation of tasks that are going to be executed by the next steps.
- Start execution process: The agent starts the execution process. From this point, the agent will not query any batch until the execution process is not finished.
- Start batch execution: The agent loads the parsed batch tasks, and executes them subsequently in the following order: initial task, microservice tasks, and final task. If there is more than one microservice task, they are executed in a parallel way.
- Execute initial task: The agent builds a container image and assigns it the given image tag. The `Dockerfile`'s[1] path and the assigned image tag are properties of the initial task definition. If the Docker image with the same image tag already exists, the image is reused and no new image going to be built.
- Execute microservice tasks: The agent loads the microservice tasks and starts to execute them in parallel.
- Start the framework: The agent deploys a Docker container and creates an instance of the measuring framework for each microservice task.

Steps of the Framework:

- Load the microservice: The framework loads the microservice using the information in the microservice task definition (location of the `.jar` file, class path, and the command-line parameters).

[1] A text document containing sequence of commands to assemble a Docker image.

- Execute and measure the microservice: The framework starts the microservice execution that includes the following three phases: initialization, iteration and measurement, and cleanup.

Steps of the Microservice:

- Initialize: The microservice runs its initialization procedure accordingly to its command-line parameters. The procedure includes the tasks that should be executed only once e.g., setting up the working environment, propagating the database, etc., before the iteration and data collection starts.
- Iteratively execute and measure: The microservice is repeatedly executed for 12 minutes. In parallel with the microservice execution, the data collection collects the microservice's resource usage and behavior.
- Cleanup: The microservice deletes the working environment, removes the temporary files or the database created during the initialization phase.

Steps of the Framework:

- Finish the microservice: The framework terminates the microservice execution.
- Report the execution result: The framework reports to the agent whether the execution succeeded, failed or it was force stopped by the orchestrator.

Steps of the Agent:

- Stop the Framework: The agent terminates the measuring framework and destroys the Docker container. Since the measured data are stored on a volume mounted to the server machine, the data remains available even after the container is destroyed.
- Execute final task: The agent compresses the collected data into a zip archive and sends it to the orchestrator.
- Report the execution result: The agent reports the entire batch execution result to the orchestrator. According to the received result the orchestrator marks the batch as `DONE`, `WAITING` or `FAILED`.

After the Performance Assessment phase is finished, the agent restarts the server machine it is running on. Then the loop starts from the beginning until the orchestrator's batch queue is not empty.

5.3 Evaluation and Prediction

The Performance Assessment phase is followed by the last phase of the submission process, the *Evaluation and Prediction* phase. This phase focuses on the data analysis, prediction of the microservice's future behavior in various conditions, and evaluation of the requirements submitted via the requirements specification.

The Evaluation and Prediction phase is depicted in more detail in Figure 5.7. As the figure shows, the phase includes three main steps:

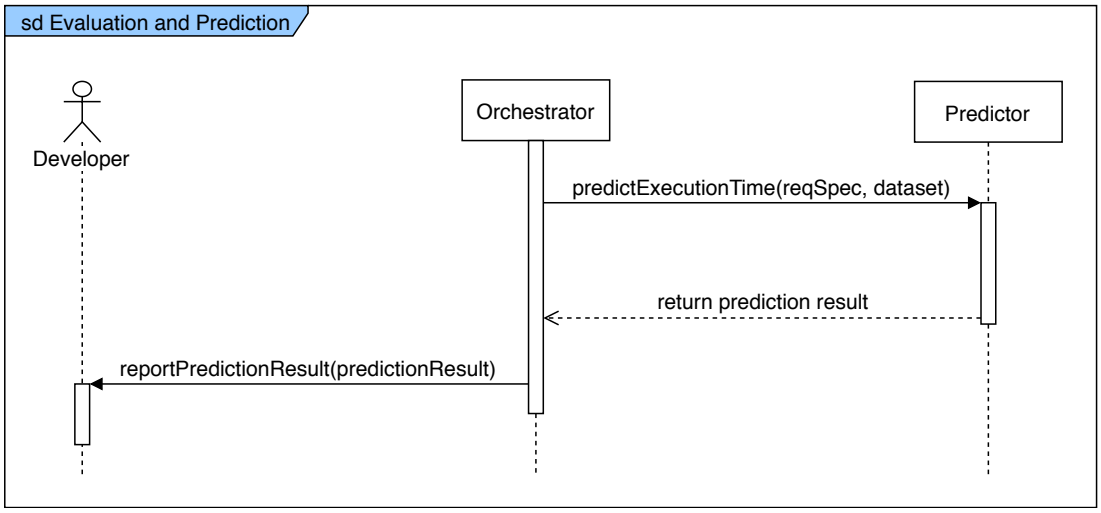


Figure 5.7: Detailed view of Evaluation and Prediction phase.

- The orchestrator sends the requirements specification and the collected data to the predictor.
- The predictor processes the data, runs the prediction, and returns the result to the orchestrator.
- The orchestrator forwards the prediction result to the microservice developer, who originally submitted the requirements specification.

Due to the complexity of the prediction step, it will be covered separately in Chapter 7.

6. Workload Generation

Traditional prediction algorithms analyze historical and current facts and use them to predict the future trends. For the majority of algorithms it holds that a more detailed and high quality source dataset can significantly improve the predictor's accuracy. Therefore, before a prediction algorithm is designed, it is crucial to record a sample dataset that covers the behavior of a wide range of microservices. The collection of a high quality dataset requires a vast amount of measurements of different microservices in various workload combinations.

Ideally, the measurements should be accomplished with a set of microservices that are targeted to be deployed in such a cloud environment. However, due to the fact that this is a prototype project, the microservices are not available at the time of implementation. Therefore, it is our responsibility to design and implement artificial microservices that simulate the behavior of future microservices in real-life conditions.

To cover the widest possible range of real-life applications, and therefore to improve the predictor's accuracy, there should be distinct microservices, each representing a particular example of the most common cloud applications. The variability should not be expressed only by the implemented algorithm, but also by the generated resource usage pattern. Ideally, the microservices should be implemented in such a way that they impose various resource demands in the measured dimensions i.e., CPU, I/O (storage) and memory utilization.

It would be extremely time-consuming to manually set up the measuring environment and collect the data for each scenario. To address this problem, we have decided to develop a measuring framework that automates the measurement of microservices in various workload conditions. The automation of measuring, however, requires a certain level of communication e.g., starting or stopping the measurement, and reporting failure, etc. between the measuring framework and the measured microservice. This can be solved by a common interface implemented by both parties, which enables the communication and therefore the measurement automation.

In this chapter, we first present the implemented microservices and then we design the interface to enable the communication between the measuring framework and the microservice.

6.1 Implementing the Microservices

In this section, we focus on the implementation of various microservices that cover a wide application domain. The implementation of a huge amount of microservices is time consuming, therefore, in the first part we conduct a research for existing solutions. These solutions include various benchmark suites and libraries that might serve as a base of the microservices. In the second part, we analyze the selected benchmarks and the microservices designed upon the libraries from resource usage perspective. Then, we select the designed microservices that will be implemented and used throughout this thesis.

6.1.1 Sources of Microservices

The microservices suitable for our needs can be implemented in two different ways. The first way is to research for existing solutions that are designed for similar problems and require only a slight modification in order to make them usable. The second option is to design and implement the microservices from scratch or using third-party libraries.

However, before we start the research, we list the requirements that the microservices should satisfy:

- Relevance: The microservice should cover a particular application domain.
- Repeatability: The microservice should execute the same code so that the results can be verified.
- Representativeness: The microservice's resource usage should be eligible for performance characterization.

In the following sections we analyze the existing solutions and compare them to custom implementations built upon third-party libraries.

Existing Solutions

As the computer architecture has evolved and become more advanced, it is a difficult task to compare their performance. To address this problem, so called *benchmarks* were developed. The benchmarks simulate various types of workloads and therefore allow to compare the performance of different architectures. Traditionally, benchmarks are divided into two categories: *synthetic* and *application* benchmarks. Synthetic benchmarks focus on a particular hardware component, while application benchmarks mimic real-life applications.

Generally, benchmarks are collected into a benchmark suite. The benchmark suites usually include applications from the same domain i.e., all benchmarks are written in the same programming language or are targeted to a particular hardware resource. Following are the most known benchmark suites:

- SuperPi[1] and Linpack[2] for CPU benchmarking.
- Bonnie++[3] and Iometer[4] for file-system and disk benchmarking.
- NBench[5] for memory benchmarking.
- DaCapo[6] Benchmark Suite and Scala Benchmark Suite[7] for benchmarking JVM-based applications.

[1] `http://www.superpi.net`
[2] `http://www.netlib.org/benchmark/hpl`
[3] `https://doc.coker.com.au/projects/bonnie`
[4] `http://www.iometer.org`
[5] `https://www.math.utah.edu/~mayer/linux/bmark.html`
[6] `https://github.com/dacapobench/dacapobench`
[7] `http://www.benchmarks.scalabench.org/modules/scala-benchmark-suite`

As a secondary option, test programs that allow to compare computer performance are transformed into *stress tests* (also called *stressors*). Stressors are designed to exercise both hardware and software components of a computer. This excessive testing helps to reveal possible hardware or software failures. The most known stressors are the following: Memtest86+[8], Aida[9], stress-ng[10]. While Memtest86+ focuses on exercising memory, the other stressors are designed to exercise physical subsystems of a computer.

Bearing in mind the requirements, we have chosen the Scala Benchmark Suite and stress-ng, as they cover a wide application domain and enable to generate a wide range resource load. Both the benchmark suite and the stressor include CPU, disk and memory-intensive applications.

Scala Benchmark Suite: The Scala Benchmark Suite (scalabench) is built upon the well-known DaCapo Benchmark Suite (dacapobench), by extending the existing Java benchmarks with Scala benchmarks. Since both languages are compiled into Java bytecode, they can be executed by the Java Virtual Machine (JVM) and thus the results can be compared.

The scalabench contains 26 benchmarks in total. It includes 12 new benchmarks written in Scala, and 14 Java benchmarks taken from dacapobench. Moreover, the input size of each benchmark can be specified between two and four input sizes[11], thus it multiplies the total number of benchmarks. Table 6.1 summarizes the currently available benchmarks.

Since scalabench is designed as a benchmark suite, it fulfills all our requirements. The implemented benchmarks cover a wide application domain. Most of the benchmarks are designed for performance evaluation, therefore they are designed for repetitive execution. Furthermore, the execution can be split into *warmup* and *measurement* phases that are described in more detail in Chapter 5. Regarding the applied load, the benchmarks create various resource demands. For instance, the benchmark suite contains both single- and multi-threaded benchmarks of multiple CPU utilization levels and I/O-intensive benchmarks that process large amount of files and also the memory-intensive benchmarks that exercise the memory.

stress-ng: stress-ng is a set of computer stressors that aim to stress various physical subsystems and kernel interfaces. Most of the implemented stressors generate excessive load in order to reveal the possible hardware and software issues of the system by pushing it to the limit. The stress-ng is able to measure the output rates that can be helpful for comparing the performance of multiple systems. However, the measured values should be handled with care, since the stress-ng was not designed as a benchmark suite, and therefore the measured values might not be precise.

The stress-ng features over 220 stress tests. These tests can be run either by individually selecting the particular tests or selecting a predefined test group. Selecting tests manually enables to stress the system by a particular combination

[8] `http://www.memtest.org`
[9] `https://www.aida64.com`
[10] `https://kernel.ubuntu.com/~cking/stress-ng`
[11] Does not hold for all benchmarks in dacapobench.

Benchmark	Description
avrora	Simulates a number of programs run on a grid of AVR microcontrollers.
batik	Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.
eclipse	Executes some of the (non-gui) jdt performance tests for the Eclipse IDE.
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file.
h2	Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb benchmark.
jython	Inteprets the pybench Python benchmark.
luindex	Uses lucene to index a set of documents; the works of Shakespeare and the King James Bible.
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.
pmd	Analyzes a set of Java classes for a range of source code problems.
sunflow	Renders a set of images using ray tracing.
tomcat	Runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages.
tradebeans	Runs the daytrader benchmark via a Jave Beans to a GERONIMO backend with an in memory h2 as the underlying database.
tradesoap	Runs the daytrader benchmark via a SOAP to a GERONIMO backend with in memory h2 as the underlying database.
xalan	Transforms XML documents into HTML.
actors	Trading sample with Scala and Akka actors.
apparat	Framework to optimize ABC, SWC, and SWF files.
factorie	Toolkit for deployable probabilistic modeling.
kiama	Library for language processing.
scalac	Compiler for the Scala 2 language.
scaladoc	Scala documentation tool.
scalap	Scala classfile decoder.
scalariform	Code formatter for Scala.
scalatest	Testing toolkit for Scala and Java programmers.
scalaxb	XML data-binding tool.
specs	Behaviour-driven design framework.
tmt	Stanford Topic Modeling Toolbox.

Table 6.1: Available benchmarks in dacapobench and scalabench. Source [14], [37].

```
$ stress-ng --class cpu 3
```

Listing 6.1: Running 3 instances of various CPU-specific stressors.

of stressors that simulate a specific workload combination. The already prepared test groups focus on a particular area of the system e.g., `cpu`, `cpu-cache`, `io`, `filesystem`, `memory`, and `scheduler`. In both cases i.e., individual and grouped tests, one can specify the number of workers to invoke. Listing 6.1 shows how to run three instances of various CPU-specific stressors.

Analysis of Existing Solutions: The use of external, existing benchmarks, and stressors has some merits and demerits as well. On one hand, the tools are specifically designed to generate various resource loads. They either target a specific resource or a combination of resources. These solutions often implement well-known benchmarking and stressor algorithms. Furthermore, the developers can maintain, fine-tune, and test the existing algorithms based on the feedback of a large user-base. On the other hand, external solutions place restrictions on the design process. Therefore, the final product will not be fully designed to meet our requirements.

Custom Implementation

Although, the existing implementations cover a wide application domain and various resource usages, there are examples of missing microservices. In order to complete the set of microservices, we have decided to implement the missing microservices either from scratch or by using existing libraries.

To make our microservices comparable with the existing benchmarks, we have set the following two requirements. The first is to implement a well-known algorithms that would possibly be used in microservices implemented in the future. The second requirement is to generate various types and levels of resource usage. By satisfying these requirements, the custom microservices will be suitable for simulating real-life microservices.

Table 6.2 presents the custom-implemented microservices. We have implemented 17 new microservices in total[12]. From the description of microservices, is evident that they meet both requirements: the microservices cover various sorting algorithms, process well-known file formats (JSON), utilize disk by reading and writing files (compress files into a zip archive, generate PDFs), exercise memory by setting up and destroying tree-like data structures (AVL-tree, Red-Black-tree), and generate CPU usage by solving problems using dynamic programming (rod cutting, egg dropping) or processing images (face detection).

Analysis of Custom Implementation: The advantages of custom implementation include the fact that each microservice is designed for a particular problem. Furthermore, they are fully compatible with the measuring framework. Since, we are the authors of the microservices, we own full control over the final product

[12]Four microservices were implemented in a cooperation with another student, whose thesis targeted the same domain. These microservices are marked with *.

Microservice	Description
JSON*	Generates and writes JSON data to disk.
PDF*	Generates images and writes them as PDF files to disk.
Sort*	Generates, sorts and writes random numbers to disk.
Cypher*	Generates random string, cyphers it and writes to disk.
AVL tree	Sets up an AVL tree by inserting 1 000 000 random items, then destroys the three by removing the nodes.
Red-Black tree	Sets up a Red-Black tree by inserting 1 000 000 random items, then destroys the three by removing the nodes.
Floyd-Warshall	Uses the Floyd-Warshall algorithm to find the lengths of shortest paths between all pairs of 2 200 vertices.
Prim	Uses the Prim's algorithm to find the minimum spanning tree.
Rod cutting	Solves Rod cutting problem using dynamic programming.
Egg dropping	Solves Egg dropping problem using dynamic programming.
Edit distance	Solves Edit distance problem using dynamic programming.
Longest common subsequence	Solves Longest common subsequence problem using dynamic programming.
Face detection	Detects human faces in the images located in the source directory.
Unzip	Extracts the given zip archive into the target directory.
Zip	Archives the target directory into a zip archive.
Compare-zip	Compares two zip archives. The archives are equal if they contain the same files.
Check-zip	Checks whether the zip archive contains the specified file.

Table 6.2: List of custom microservices.

and we are not dependent on external factors. Therefore, the microservices can be changed in a short time and they can be combined in multiple ways to generate various levels of resource demand.

However, the custom microservices have their disadvantages too. For instance, the implementation and maintenance is time-consuming. Moreover, the final solution is not as complete as the existing solutions are, because we are lacking the feedback from a larger user-base.

6.1.2 Selecting the Most Suitable Microservices

Based on the details mentioned above, the choice between the sources of the artificial microservices is not straightforward. Both source groups have their advantages and disadvantages, therefore, we have decided to select the most suitable microservices from both groups.

In order to choose the most suitable test microservices, we conducted an iterative, multi-stage selection process, which consists of two main parts. First, we

executed the microservices and measured their runtime properties. In the second part, we analyzed the collected data and selected the most suitable microservices that met our criteria.

Before starting the selection process, we set up the test environment. The goal of the test environment was to simulate the production environment along with its hardware and software constraints. In this case, we required the same hardware components and the same version of the operating system, including the installed applications. Furthermore, we executed each microservice in a Docker container where the CPU usage was restricted to 25% of the total available CPU resources.

Data collection was performed by various built-in and third-party monitoring tools. At the time of execution, no other microservices or resource intensive applications were enabled on the host machine in order to make the data collection more precise and to minimize the effect of external factors.

After analyzing and evaluating the measured data of current iteration we have reduced the number of microservices based on the following criteria:

- **Resource usage:** The selected microservices should produce different resource usage patterns along with the following three dimensions: CPU, disk I/O, and memory usage. More precisely, we focus on the attributes presented in Section 4.1. Therefore, we select benchmarks that meet at least one of the following attributes: single-or multi-threaded, high user- or kernel-time, random or sequential I/O, various read/write ratio, and various levels of memory usage.

- **Execution time:** The too much lengthy execution time unnecessarily prolongs the measurement process and thus reduces the number of microservices that can be measured during a specific time period. However, the shorter execution time prevents the measuring framework to collect sufficient amount of data, which makes the microservice's analysis less precise. Furthermore, in most of the cases the execution time increases when the microservice is colocated on a single host machine with other resource intensive microservices. Based on these attributes and our analysis, we have defined the ideal length of the execution as a time interval between 4 and 12 seconds.

- **Stability and Repeatability:** During the measurement process each microservice is executed up to one-hundred-times. This requires a microservice that is stable during its execution, does not crash and, does not produce errors. Furthermore, the repeatable execution flow and the consistent resource usage pattern is also required. Otherwise, the irregular behavior could distort the result of the analysis.

- **Uniqueness:** All the microservices should be different, each focusing on a particular area. This helps to cover a wide application domain even with a low number of microservices.

Table 6.3 summarizes the microservices that satisfy these criteria. Columns *CPU* and *Disk* present the utilization of the CPU and the hard disk drive, respectively. Column *RAM* expresses the amount of memory allocated by the microservice.

Here we describe the notation of signs: For the *CPU* column, each + represents an additional 25% of total CPU usage. For the *Disk* column, * represents negligible disk usage i.e., 0% - 5%, while each + represents steps of 25% additional disk usage. The first + sign in disk usage represents 5% - 25% disk utilization. For the *RAM* column, each + represents an additional 1000 MB of allocated memory.

Microservice	**Source group**	**Resource Demand**			**Exec. time**
		CPU	**Disk**	**RAM**	
sunflow	scalabench	++++	*	++	5519 ms
h2	scalabench	++	*	++++	9330 ms
apparat	scalabench	++	*	++	4390 ms
tmt	scalabench	+++	*	++	4621 ms
avrora	scalabench	++	*	++	12041 ms
matrix	stress-ng	+	*	++	7998 ms
JSON	custom	++	+	+++	5073 ms
PDF	custom	+	+++	++++	7138 ms
Sort	custom	+	++	++	5423 ms
Cypher	custom	+	++	++	3554 ms
AVL tree	custom	+	*	++	4983 ms
Red-Black tree	custom	+	*	++	5836 ms
Floyd-Warshall	custom	+	*	+	8272 ms
Rod cutting	custom	++	*	+	6878 ms
Egg dropping	custom	+	*	+	10640 ms
Face detection	custom	+	+	++	6277 ms
Unzip	custom	+	++++	++++	6354 ms

Table 6.3: List of selected microservices.

6.2 Interface Design

Even though, we have selected the most suitable microservices in the previous step, they are not ready to be executed and measured yet. As is it described in Section 5.2.2, the microservice's measurement is part of a complex execution process. This execution process is designed to be able to execute and measure any kind of microservice, including our artificial microservices and the future microservices submitted by the microservice developers.

The Performance Assessment phase in Section5.2.2 describes, that the microservices are executed by the measuring framework. The measuring framework loads the particular microservice that is specified in the batch configuration file and manages the microservice's measurement process.

Making the measuring framework to be able to handle various kind of microservices submitted by multiple developers requires a communication interface implemented by both parties. This interface enables the microservice to be attached to the measurement process, and enables the measuring framework to measure the particular microservice's resource usage.

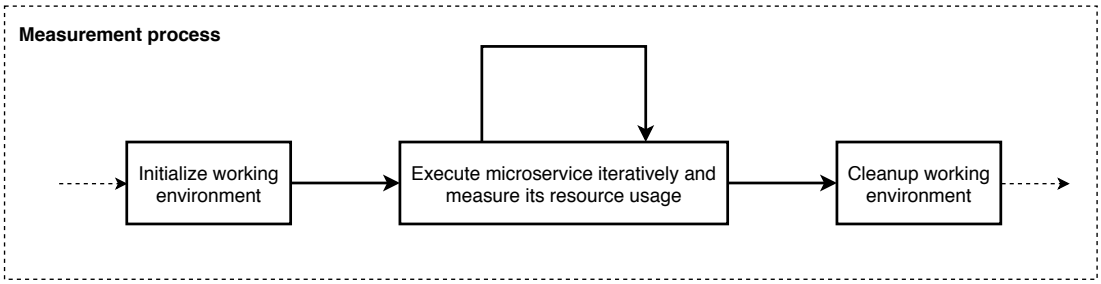


Figure 6.1: Main steps of the microservice within the measurement process.

The goal of this section is to design the communication interface. In the first part, we analyze the communication between the measuring framework and the microservice. Then, in the second part we use the result of this analysis to design the interface.

6.2.1 Designing the Interface

Analysis

Since the steps of the measurement process are already described in Section 5.2.2, now we focus on the steps related to the microservice execution. The microservice execution consists of the following three steps:

- Initializing the microservice and its working environment.
- Executing the microservice iteratively and measuring its resource usage.
- Finishing the measurement and cleaning up the working environment.

The measurement process starts with the initialization phase, where the microservices can prepare their working environment e.g., creating the working directory, setting up the database, etc. This step is executed only once, after the measuring framework starts and loads the microservice.

The initial step is then followed by the execution and measurement step. The microservice is iteratively executed for 12 minutes. During the microservice execution the data collector records the microservice resource usage.

In the final step, the run is finished by cleaning up the environment created in the initial step. This step ensures that the current run is terminated properly and thus the following run can be started.

At the end of each step, the result of the execution is returned to the measuring framework. In case of success, the execution is continued with the next step. However, in case of failure, the execution is terminated and the failure is handled.

Although, the measurement process is more complicated and the presented steps may contain multiple sub-steps, we do not continue the analysis in a more detailed level of abstraction. Otherwise, we would set up tight restrictions against the submitted microservices, which would make difficult to handle them as black-boxes.

```
1
2 public interface MeasurableMicroservice {
3
4     MicroserviceResult init(String[] args);
5
6     MicroserviceResult iterateAndMeasure();
7
8     MicroserviceResult cleanup();
9 }
```

Listing 6.2: Interface functions representing the steps of the microserivce execution.

Implementing the Interface

As a first step, we assign a particular function for each of the steps presented in the Analysis section (see Listing 6.2). By doing so, we enable the measuring framework to start any step according to the measurement flow.

Most of the functions are parameter-less, however, the `init(String[] args)` function accepts an array of `String`s as its parameter. The passed parameters can be used for setting up the microservice, based on the requirements in the batch configuration file.

As it is described in the previous section, the result of the execution is returned at the end of each step. For this, the interface defines a custom `enum`, the `MicroserviceResult`, that is set as the return type of all three functions executing a step. Listing 6.3 shows the possible results of the execution.

The functions shown in Listing 6.2 enable the measuring framework to execute any of the microservice's steps. However, they do not enable the microservice to report its state changes to the measuring framework. For instance, the microservice should notify the measuring framework before the current iteration starts and finishes, so the measuring framework can start and stop the iterative data collector at the right moment. Therefore, each microservice should have a reference that can be used for notifying the measuring framework. In our case, the reference will be set by the `setMicroserviceRunFlowController( IMicroserviceRunFlowController controller)` function. The function is designed to be called from the measuring framework, which should pass a reference to the run flow controller that is responsible for managing the flow of the current run.

The final interface is presented in Listing 6.4. Implementing this interface by the microservice enables the communication with the measuring framework. Therefore, the microservice can be executed and measured by the measuring framework.

```java
public enum MicroserviceResult {
    /**
    * The microservice's initialization was successfully finished.
    */
    INIT_FINISHED,
    /**
    * Failed to initialize the microservice.
    */
    INIT_FAILED,
    /**
    * The microservice's execution was force stopped.
    */
    FORCE_STOPPED,
    /**
    * The microservice's execution successfully finished.
    */
    RUN_FINISHED,
    /**
    * The microservice's execution failed.
    */
    RUN_FAILED,
    /**
    * The working environment was cleaned up.
    */
    CLEAN_UP_FINISHED,
    /**
    * Failed to clean up the working environment.
    */
    CLEAN_UP_FAILED
}
```

Listing 6.3: Possible results of the microservice measurement process.

```java

public interface MeasurableMicroservice {

    MicroserviceResult init(String[] args);

    MicroserviceResult iterateAndMeasure();

    MicroserviceResult cleanup();

    void setMicroserviceRunFlowController(IMicroserviceRunFlowController
        controller);
}
```

Listing 6.4: Interface for measuing the microservices.

7. Prediction Process

This chapter focuses on developing a prediction process that will be able to predict the microservice execution time. In the first step, we perform the data collection process. Then, we preprocess the collected data to prepare them for the prediction process. Next, we use the processed data to characterize the measured microservice. Finally, we describe the predictor and evaluate its accuracy.

7.1 Data Collection

The goal of this section is to describe how the data is collected during the microservice execution.

7.1.1 Data Collectors

During the analysis of different data sources in Chapter 4, we realized that different data sources require different data collection approaches to collect them efficiently. Based on this observation, we have decided to divide the data collection approaches into the following categories.

- Sampling at a particular frequency.
- Repeatedly measuring along with every microservice iteration.
- Combination of both.

As we see, the listed approaches are significantly different. Consequently, it would be unnecessarily complicated to design a single data collector that maximizes the quality of the collected data, while minimizes the data collection overhead. Therefore, we have designed two types of data collectors: *sampling data collector* and *iterative data collector*. Each data collector is targeted to a specific collecting approach.

The sampling data collector is scheduled at a predefined frequency e.g., every 2 seconds and collects the data accordingly to its implementation. It is independent from the current state of the measurement process i.e., whether the currently executed microservice is at the beginning or at the end of its iteration. The sampling data collector starts at the beginning of the 12 minute long execution period and runs until the period elapses.

The iterative data collector is more tightly connected to the microservice execution than the sampling data collector. The iterative data collector collects data during each iteration of the microservice. The data collection starts when a microservice iteration begins, and stops when the microservice iteration finishes. Then the collection is repeated again along with the next iteration of the microservice. Therefore, the number of data entries collected by iterative data collector equals to the number of repetitions of the microservice during the 12 minute long measuring interval.

Combination of sampling and iterative data collectors is not implemented as a third data collector, but as a data processing step. The data collection step is

followed by the data processing step (see Section 7.2), where the measured data can be processed and transformed to any representation. As a result, the measuring framework uses one less data collector, which minimizes the data collection overhead.

Even though, we have implemented concrete data collectors in our prototype, they can be easily replaced, because the architecture is prepared for adding new data collectors. We have provided the detailed implementation in Chapter 8.

7.1.2 Environmental Settings

In Chapter 5, we described that the microservices are executed and measured in Docker containers. There are multiple advantages of isolating the microservices in containers from the hosting environment e.g., security, unified environment for all microservices, etc.

Just like Docker, the JVM also serves as a virtual environment, that isolates the Java application from the operating system. Even though, Docker container and the JVM are used for different reasons, they both support changing the default settings of the virtual environment or applying various restrictions on it.

In order to be able to apply changes on the virtual environments, we have added support for command-line options to the batch configuration file. As it is presented in Section 5.1.2, the batch configuration file supports `docker` and `java` attributes, where the command-line options of the particular virtual environment can be specified.

During the measurements, we used the `docker` attribute to apply hardware-limitations on the containers. More specifically, we used the `--cpus=1` Docker option to limit the container CPU-usage to 1/4 of the total CPU capacity [38]. The idea behind limiting the CPU-usage is to equally share the available hardware capacity among the containers and to prevent resource-intensive microservices from allocating high portion of resources and thus slowing down the colocated microservices. Beyond the CPU-limitations there were no restrictions applied on the available RAM memory or disk usage.

In the batch configuration file, we used the `java` attribute as well. Although, in our case we did not apply any limitation e.g., limiting the JVM heap size, we used the command-line options to enable the logging for Garbage collector and to specify the output file location.

Listing 5.2 shows, that the attributes support adding multiple command-line options at once. Moreover, they accept any option[1] supported by the Docker or the JVM.

7.2 Data Processing

The sampling and the iterative data collectors capture the data from different perspectives, thus the produced outputs are different as well. However, building the model for prediction requires a unified input format. Therefore, there is a need for data preprocessing that converts the collected data into the required

[1]The validity of the option should be checked by the developer who adds the new option.

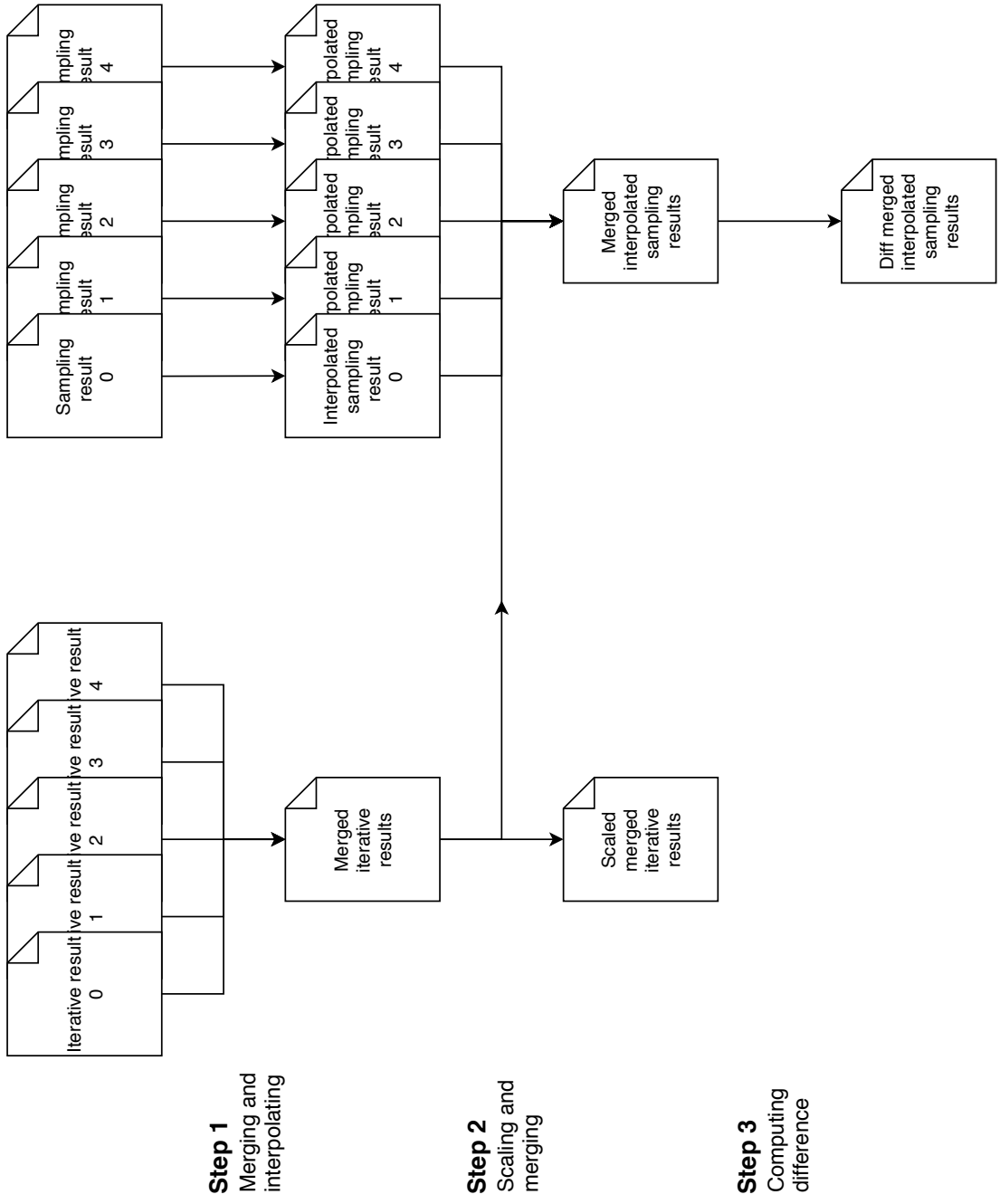


Figure 7.1: Data processing steps.

representation. Moreover, the execution of preprocessor includes combining the output of the two data collectors to generate the combined dataset.

The data preprocessing is the first phase of the prediction process. The steps of data processing are shown in Figure 7.1. The input is represented by two types of files: data collected by the iterative data collector and the sampling data collector. Since each measurement is repeated five-times, there are five output files generated by each data collector.

In the first step, the two types of input files are processed in a different manner. The result of the iterative data collector (named as *Iterative result 0*, .., *Iterative result 4*) is merged into *Merged iterative results* file. The merging process takes the last 50 lines (which equals to the last 50 iterations of the microservice) of each file indexed from 0 to 4 and subsequently copies them into the Merged iterative results file. Because the data are the result of the time-restricted measurement process (described in Section 5.2.1), merging the last 50 measurement results in a stable, consistent dataset.

As a part of the first data processing step, the result of the sampling data collector (named as *Sampling result 0*, .., *Sampling result 4*) is processed as well. The sampling results represent the system's state in every 2 seconds, because the sampling data collector is executed in every 2 seconds. However, the input required by the subsequent steps of the prediction process should represent the system's state in every second. Therefore, in this data processing step the sampling results are interpolated to 1 second interval.

The second step of the data processing is divided into two parts: *scaling* and *merging*. In the scaling part, the Merged iterative results file is processed. The process scales the measured resource usage attributes with the respect to the execution length. Values of the measured attributes are divided with the elapsed time (execution length) and the result is saved into the *Scaled merged iterative results* output file.

In the merging step, sections of the Interpolated sampling results are merged into an output file (see Figure 7.2). Each line in the Merged iterative results represents the microservice's resource usage during its execution. Since the usual execution time of the selected microservices ranges from 3 to 12 seconds, for each line in the Merged iterative results, there are multiple corresponding lines in one of the Interpolated sampling result files. During the merging process, for each line in the Merged iterative results the corresponding lines of the Interpolated sampling results are found and added to the *Merged interpolated sampling result* output file. As a result, the content of the output file covers the same time interval as the Merged iterative results, but uses the interpolated one-second sampling results. This representation will be useful for the Data Transformer described in the next section.

In the third step of data processing, the previously created Merged interpolated sampling results file is processed. During the processing, the values of subsequent rows are subtracted. Thus, the *Diff merged interpolated results* output file contains the differences of the measured values between each elapsed second.

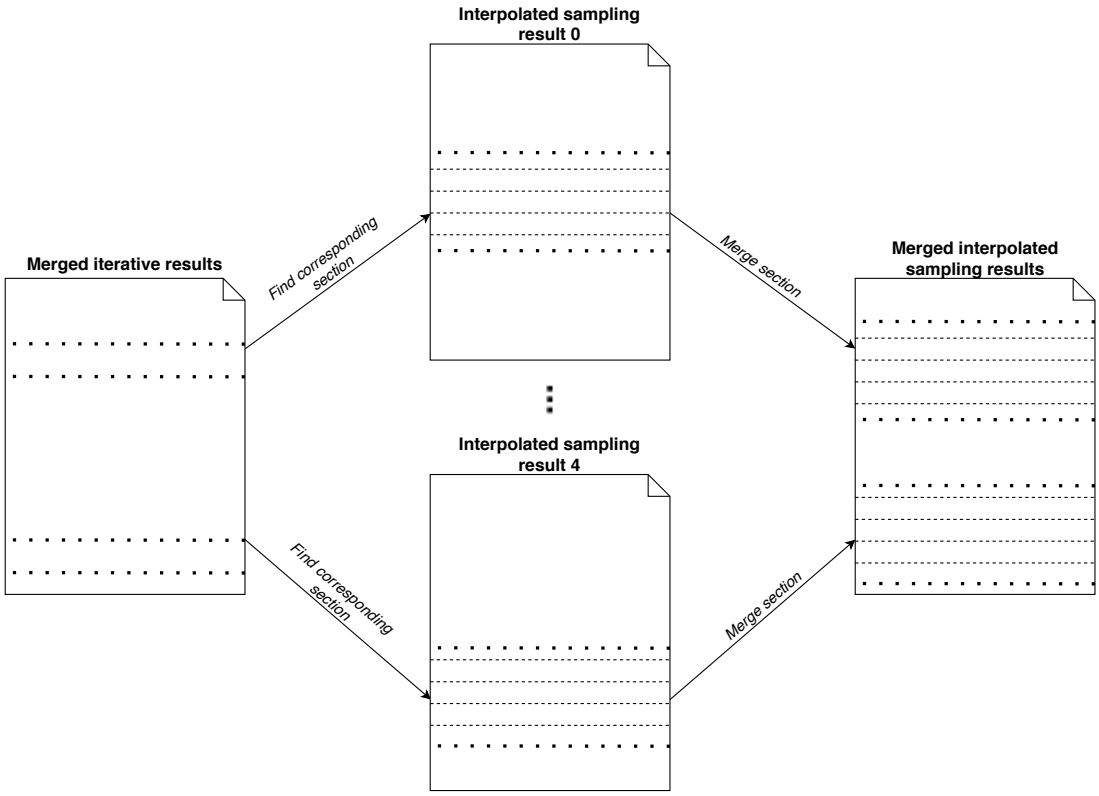


Figure 7.2: Merging Interpolated sampling results.

7.3 Characterization

The purpose of this section is to characterize the microservice behavior based on its resource usage. The overall characteristics of the microservice enable a better understanding of its behavior for the cluster orchestration system. This can be later used as an additional property when planning the deployment of colocated workloads.

The goal of data processing is to prepare data for further use in the prediction process. Although, the preprocessed files return the measured resource usage in different representations, they are not suitable for characterizing the microservice behavior. Therefore, we designed additional transformation steps that return a representation similar to the `vmstat` and `iostat` monitoring tools, presented in Chapter 4. The final attributes should outline the most important resource usage indicators throughout all three dimensions i.e., CPU, disk, and memory usage. Using the attributes and applying the concepts from Chapter 4 the microservice can be easily characterized.

7.3.1 Data Transformation

The data transformation step is depicted in Figure 7.3. The input files entering the data transformer are the *Merged interpolated sampling results* and the *Diff merged interpolated sampling results*. Both files are product of the data processing steps described earlier in this chapter. The data transformation step produces two output files: *Detailed characterization result* and *Short characterization result*. The output files differ both in their format and content.

The content of the Detailed characterization file focuses on the characterization of the microservice by transforming the input data into a format similar to the output of `vmstat` and `iostat` monitoring tools. The Detailed characterization file consists of 26 columns, separated by semicolons. Each column stands for a particular resource usage attribute. The attributes are stored in a special header file located next to the Detailed characterization file. An example of the Detailed characterization file along with the corresponding header file is located in the attached medium in the `/prediction_resources` directory.

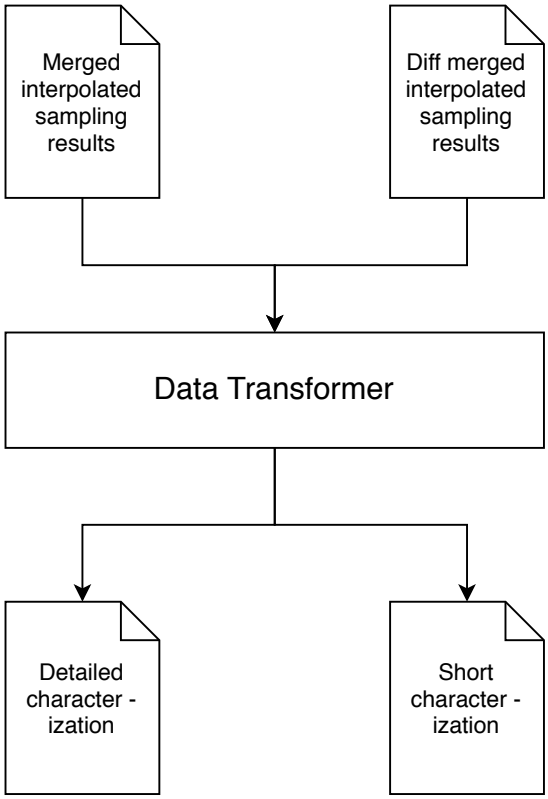


Figure 7.3: Data transformation process.

The Short characterization result is formatted as a YAML file, that presents the result of the characterization as a structure of nodes and attributes. Listing 7.1 shows an example of the characterization file. The programmatically generated YAML file contains exactly one `CharacterizationResult` node, which is further divided into `cpu_result`, `disk_result`, and `mem_result` nodes. Each of the sub-nodes include several attributes expressing the usage of the particular hardware resource. The values are calculated as the average for every attribute of the detailed characterization file.

The `cpu_result` node (lines 2-6) is a type of `CpuResult` and contains the following attributes. The `total_time` attribute shows the total available CPU time of the installed CPUs. The `system_time` shows the time spent by executing system-level code e.g., syscalls, interrupts, while the `user_time` shows the time spent by user-level processes e.g., the microservice itself, including the overhead of the measuring framework. The time-related attributes show the total time, summarized over the installed CPUs. The `utilization` attribute expresses the overall CPU utilization in percentage.

```
1  !yamlable/data_transformation.characterization_result.CharacterizationResult
2  cpu_result: !yamlable/data_transformation.characterization_result.CpuResult
3    total_time: 399.4059281437127
4    system_time: 58.855808383233516
5    user_time: 42.76676646706589
6    utilization: 25.736074172490213
7  disk_result: !yamlable/data_transformation.characterization_result.DiskResult
8    r_s: 1.3532112527147375
9    w_s: 190.81813482505507
10   rkB_s: 65.7192063075226
11   wkB_s: 73495.57132322335
12   avgqusz: 139.7915910988561
13   avgrqsz: 833.4796535740735
14   utilization: 99.73854215713382
15 mem_result: !yamlable/data_transformation.characterization_result.MemoryResult
16   used: 3320506.178443112
17   cached: 2092573.3067664658
18   dirty: 6962.30263473053
19   writeback: 726378.0895808374
```

Listing 7.1: Example of short characterization result file.

The `disk_result` node (lines 7-14) is type of `DiskResult` and contains the following attributes. Attributes `r_s` and `w_s` show the the amount of read and write requests per second. Similarly, the attributes `rkB_s` and `wkB_s` show the amount of data read and written per second, in kilobytes. The attributes `avgqusz` and `avgrqsz` show the average size of requests and the average queue length of request sent to the disk device. The `utilization` attribute expresses the overall disk utilization in percentage.

The `mem_result` node (lines 15-19) is type of `MemResult` and contains the following attributes. The attribute `used` shows the total amount of physical memory used by the system (including buffers, cache, etc.). The attribute `cache` shows the amount of physical memory used for files read from the disk. The attribute `dirty` shows the amount of physical memory waiting to be written back to disk. The attribute `writeback` shows the amount of physical memory actively being written back to disk. Values shown by each attributes are expressed in kilobytes.

7.4 Prediction

The goal of this section is to design a prototype of a prediction method that can be used for predicting the microservice execution time when it is colocated with other workloads. First, we describe the collected data and separate it into training and testing datasets. Then we design the prediction process. Finally, we evaluate the prediction by predicting and verifying the execution time of the testing microservices.

7.4.1 Workload Combination

Traditionally, the prediction methods rely on historical datasets when predicting future values. Therefore, there is a need for a large dataset, that covers a wide range of microservice. To build this dataset, we use the artificial microservices introduced in Chapter 6.

The microservices are measured in multiple combinations. The goal of measuring various workload combinations is to simulate real-life scenario, where multiple colocated workloads are deployed on a single server machine. Using the available microservices, we executed and measured the following workload combinations:

- Single: 17 microservices.
- Double measurements: 289 workload combinations.
- Triple measurements: 2601 workload combinations.
- Quadruple measurements: 3000 workload combinations.
- Quintuple measurements: 3000 workload combinations.

Even though, we experimented with tuples of higher arity e.g., sextuples, septuples, octuples, etc., we rejected this approach. While executing more than five colocated microservices, the server machine became overloaded, which resulted in extremely high microservice execution time.

Furthermore, we measured only a fraction of all possible quadruple and quintuple combinations (6.78% and 0.39%, respectively). Because each measurement lasts for approx. 75 minutes, it would not be possible to measure 44217 quadruple and 751689 quintuple combinations in a reasonable time. Therefore, we selected the measured $3000 - 3000$ combinations randomly.

As it is described in Chapter 5, each workload combination consists of one main microservice and zero or more background microservices. From the perspective of the collected data, the main microservice is the only relevant workload. Moreover, regardless of their definition order, each background workload is handled with the same priority. Therefore, we consider two workload combinations equal if their main microservice is equal and their background workloads are equal, irrespective of their order. For instance, workload combinations `A-AVL-RB` and `A-RB-AVL` are equal. In order to maximize the number of various workloads measured during a limited time period, we do not measure the duplicates of a workload combination.

The selected combinations are stored in the attached medium,
in the `/prediction_resources/workload_combinations/` directory. The combinations are structured in `YAML` language and serve as input for the Batch Generator.

7.4.2 Dataset Separation

In order to be able to design the prediction process, evaluate, and test the results, there is a need for separating the dataset into training and testing datasets. Using the training dataset, we are going to build the prediction model, while the testing dataset would be used for evaluating its correctness.

Since, the amount of available microservices is limited at the time of writing, we do not add new microservices that could represent the testing dataset. We split the available microservices into two groups. From the microservices that create the testing dataset, we expect to simulate the microservices that will be submitted by the microservice developer in the future. Furthermore, we require from the set of testing microservices to cover the widest possible application domain and to utilize a specific resource only (CPU, disk I/O or RAM) or the combination of these resources. As a result, the selected microservices that represent the testing dataset are the following: *Cypher, Egg dropping, Face detection, Red-black tree, Unzip*. In the following part of this chapter, we will refer to microservices using their IDs: *CYPHERD, EGG, FACE, RB, ZB*. For more information about the microservices, please see Chapter 6. The training dataset is represented by the remaining 12 artificial microservices.

To provide the most detailed dataset, the microservices in training dataset are measured in all combinations for double and triple workloads. For quadruple workloads, a randomly selected set of combinations are measured (see Section 7.1). Furthermore, microservices from both datasets are measured in workload combinations with themselves - every microservice A is measured as single, double, triple, and quadruple combinations (noted as A, A-A, A-A-A and A-A-A-A).

7.4.3 Designing the Prediction Process

The prediction process consists of multiple subsequent steps that the submitted microservice goes through in order to predict its execution time in various workload combinations. In this section, we focus on the description of these steps. First, we present the steps in general. Afterwards, we present details of each step.

Main steps

1. Transforming the microservice into a common representation that enables to compare the submitted microservice with the existing microservices.

2. Finding a microservice from the training dataset that is most similar to the submitted microservice.

3. Computing the execution time ratio of the submitted microservice and its most similar microservice.

4. Computing the execution time of the submitted microservice along with various workload combinations using the execution time ratio and the exciton time of its most similar microservice.

It is noticeable from the above mentioned steps, that the Step 2 is dependent on the Step 1 as well as the Step 4 is dependent on the Step 3. Therefore, we group the dependent steps into phases and describe the design of prediction process as a sequence of subsequent phases.

Phase 1: Step 1 and Step 2

The first phase of the prediction process focuses on finding a microservice from the training dataset that is most similar to the submitted microservice. In order to do so, the phase includes two steps.

First, the microservices should be presented in a specific format that enables to compare them with each other. This requires a step that transforms the microservices into that particular representation. Second, there is a need for a mechanism that is able to compare the microservices using their specific representations.

To compare the microservices, we use the Scaled merged iterative results file created during the Data processing steps. This file consists of 250 lines, describing the microservice's resource usage during each iteration by measuring 19 different attributes. Although, the high number of lines along with the attributes help to characterize the microservice behavior during each iteration, they reduce the correctness of the microservice comparison. In order to increase the comparison success rate, we decided to represent each microservice with a single line, which is computed from the content of the Scaled merged iterative results file.

In the next step, this specific representation is used for comparing the microservices. To compare any two selected microservices, we use a distance function (metric). Using a metric, we can calculate the distance between two microservices, which in our case represents the similarity of the microservices. The lower the distance between two microservices is, the more similar they are.

During our work, we considered multiple combinations of representations and metrics. Following are the analyzed representations:

- Calculate the average value for every attribute.
- Calculate the median value for every attribute.
- Calculate the geometric mean value for every attribute.

The analyzed metrics are the following:

- Euclidean metric.
- Manhattan metric.
- Chebyshev metric.

The 3-3 representations and metrics result in 9 different combinations. However, at the time of designing Step 1 and Step 2, we did not have enough data to choose the best combination. Therefore, we selected a random combination and finished the design of Step 3 and Step 4. Afterwards, when each step of the prediction process was implemented, we returned to Step 1 and Step 2 to test the combinations. In our test case, we used 289 measured double workloads to find the most similar microservice, predict the execution time of the workload combination, and to verify against the measured data. Comparing the predicted and the real execution time we calculated the error rate of each combination and summarized the results in Table 7.1. As the results show, the combination of *average representation* and the *Manhattan metric* matches the microservices with the lowest error rate. Therefore, this combination is selected for the prediction process.

Combination		Error rate				
Representation	**Metric**	**Average**	**Median**	**Minimum**	**Maximum**	**Sum**
Average	Euclidean	0.14151	0.07083	0.00001	1.17654	40.89641
Average	Manhattan	0.13500	0.05661	0.00001	1.17654	39.01622
Average	Chebyshev	0.15508	0.06955	0.00002	0.88302	44.81821
Median	Euclidean	0.15360	0.06255	0.00001	1.17654	44.38921
Median	Manhattan	0.15031	0.05973	0.00001	0.88302	43.43866
Median	Chebyshev	0.14477	0.07450	0.00016	0.98717	41.83751
Geometric mean	Euclidean	0.16832	0.07083	0.00001	1.17654	48.64592
Geometric mean	Manhattan	0.14372	0.05907	0.00001	0.88302	41.53378
Geometric mean	Chebyshev	0.16265	0.07690	0.00001	0.98717	47.00549

Table 7.1: Error rates of every combination of representation and metric.

Phase 2: Step 3 and Step 4

The second phase of the prediction process focuses on computing the execution time of the submitted microservice colocated with various workloads. To achieve this goal, this phase is split into two steps. The goal of the first step is to compute the execution time ratio of the submitted and its most similar microservice. Then, in the second step, this ratio is used to predict the average execution time of the particular workload combination.

Using the appropriate representation of the Scaled merged iterative results file and applying the appropriate metric from Step 2, we can determine the microservice whose behavior is most similar to the submitted microservice within a single unit of time (1 second in our case). However, the similar behavior of the microservices does not imply that their execution time will also be close to each other. For instance, the submitted microservice processes a larger task than its most similar microservice, therefore the execution requires more time. The difference of execution time has to be considered, when predicting the execution time of any workload combination using the execution time of the most similar microservice. To be able to apply this time difference in the predicted execution time, we decided to compute the execution time ratio of the submitted and its most similar microservice. Let's denote the newly submitted microservice with N, its most similar microservice with S, and the average execution time as $et(N)$ and $et(S)$. Then, the execution time ratio is calculated as follows:

$$execime\_ratio = \frac{\frac{et(N)}{et(S)} + \frac{et(NN)}{et(SS)} + \frac{et(NNN)}{et(SSS)} + \frac{et(NNNN)}{et(SSSS)}}{4}$$

The formula shows that the execution time ratio is calculated as the average execution time ratio of single, double, triple, and quadruple combinations of the submitted and the most similar microservice.

In the second step, we use this formula to predict the average execution time of the submitted microservice colocated with any workload combinations. Let's denote the particular workload combination with X. Then, the predicted average execution time of the submitted microservice colocated with workload combination X is calculated as follows:

$$predicted\_avg\_exectime\_N\_X = avg\_exectime\_S\_X * execime\_ratio$$

As the formula describes, we use the average execution time of the most similar microservice with colocated workload X from the corresponding Merged iterative results file, and we multiply it by the execution time ratio. The result is then the predicted average execution time of the combination N-X.

7.4.4 Evaluation

In this section we evaluate the prediction process. The goal of the evaluation is to validate the correctness of the presented steps and the overall prediction process.

Before we start the evaluation, we divide the entire available dataset into training and testing dataset accordingly to the testing microservices selected in Section 7.4.2. The testing microservices are selected in a manner, that there should exist at least one similar microservice in the training dataset. Therefore, the prediction process should be able to select the microservice that is most similar to the one from the testing microservices. Beyond that, to show that how the prediction process behaves in a situation, where the training dataset does not include any similar microservice, we decided to add the *Unzip* (denoted as *ZB*) microservice to the testing dataset.

During the evaluation process, we focus on the already measured double, triple, and quadruple workload combinations of the testing microservices. The execution time of single workloads will not be predicted because it is determined during the microservice measurement.

We evaluate the correctness of the prediction process from the aspect of error rate of the predicted and the real execution time of particular workload combinations. Since the workload combinations we are going to evaluate are extracted from our original measured dataset, the real execution time of each combination is known. Thus, it enables us to evaluate the correctness of the prediction process by verifying the predicted execution time against the real measured execution time.

In the following part of this section, we are going to use figures to evaluate the prediction process in case of each test microservice. The figures combine the real and the predicted execution time in milliseconds for every measured workload of double, triple, and quadruple combinations. Furthermore, we are going to refer to Table 7.2, that summarizes the error rate over the testing microservices and their predicted double, triple, and quadruple combinations.

Microservice	**Error rate**								
	Double			**Triple**			**Quadruple**		
	Min	**Max**	**Average**	**Min**	**Max**	**Average**	**Min**	**Max**	**Average**
CYPHERD	0.069103	0.219291	0.185672	0.036139	0.288659	0.195137	0.007625	0.359415	0.197647
EGG	0.036147	0.074182	0.057391	0.000921	0.15564	0.041813	0.000237	0.276021	0.046179
FACE	0.026932	0.117689	0.075302	0.001154	1.401065	0.076166	0.001862	1.278915	0.103014
RB	0.00684	0.035495	0.019303	0.004248	0.055577	0.021804	0.003426	0.031801	0.019649
ZB	0.564034	0.88302	0.738443	0.210829	0.832983	0.535543	0.084218	0.627021	0.426497

Table 7.2: Comparison of error rates for double, triple and quadruple workload combinations.

Microservice CYPHERD

Figure 7.4 shows the real and the predicted execution time of the CYPHERD microservice in double, triple, and quadruple combinations. As a result of the prediction process, microservice PDFD was selected as the most similar microservice to CYPHERD.

Both microservices belong to the group of disk-intensive microservices. Furthermore, the behavior of the microservices is very similar, since they both generate and write relatively large files to disk. Analyzing the characteristics of the microservices, the disk-intensive behavior is also supported by the high value of the average request size, the high transfer speed, and the high amount of I/O requests. Both microservices generate medium disk utilization: 43% in case of CYPHERD and 63% in case of PDFD microservice. Based on the characterization result, it is apparent that the prediction process successfully selected a similar microservice from the set of available microservices.

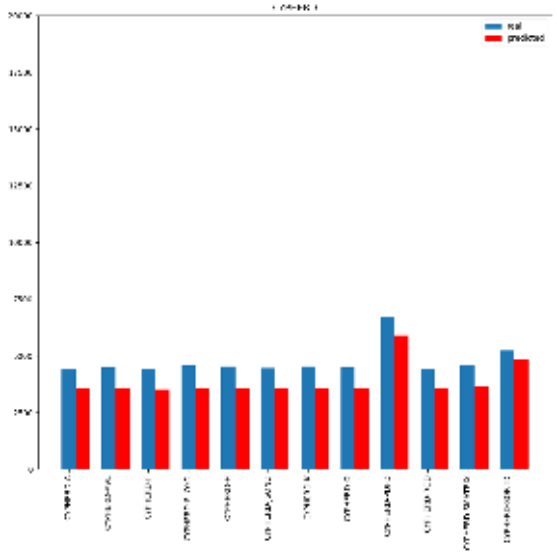
The figure shows that the predicted execution time follows the shape of the real execution time, but the predicted values are below the real execution time in every workload combination. The difference is 18.6%, 19.5%, and 19.8% in average, for double, triple, and quadruple combinations, respectively.

The high error rate is caused by the behavior of the PDFD microservice. Due to its higher disk utilization in general, the execution time of the PDFD microservice increases faster than the execution time of the CYPHERD microservice in double, triple, and quadruple combinations with itself i.e, PDFD-PDFD, PDFD-PDFD-PDFD and PDFD-PDFD-PDFD-PDFD; similarly to CYPHERD. The high values in the denominator decreases the final value of the execution time ratio, and therefore causes the prediction process to predict the execution time below the real values.

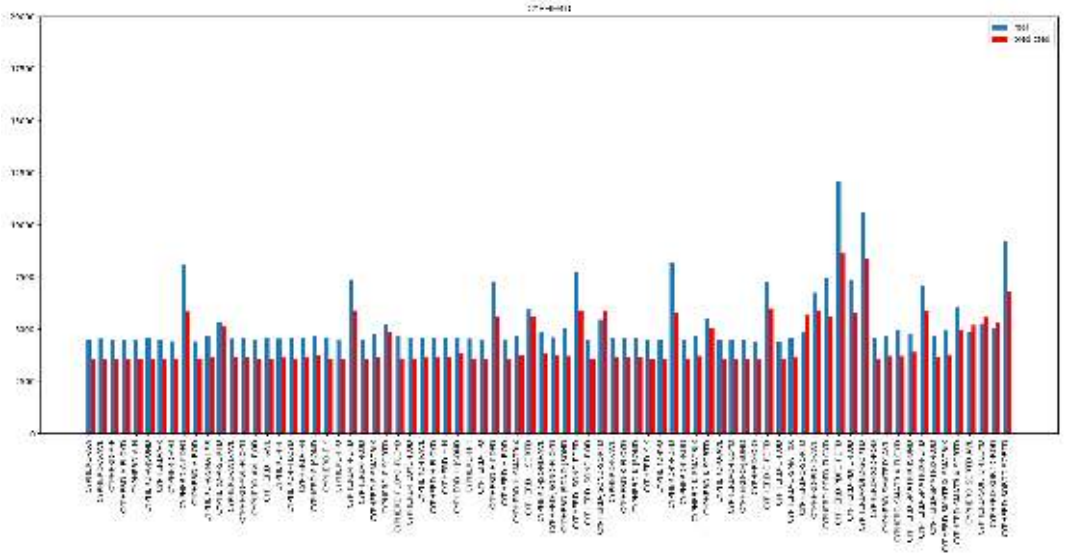
This phenomenon is particularly observable in workload combinations, where at least one of the background workloads is a disk-intensive microservice as well e.g., CYPHERD-A-PDFD, CYPHERD-PDFD-K-SORTD. In such cases, the error rate can exceed even 35%.

It is shown in Table 7.2, that the error rate varies between 0.8% and 35.9%, while the average error rate remains below 20%. The results show that there are substantial differences between the real and the predicted execution time.

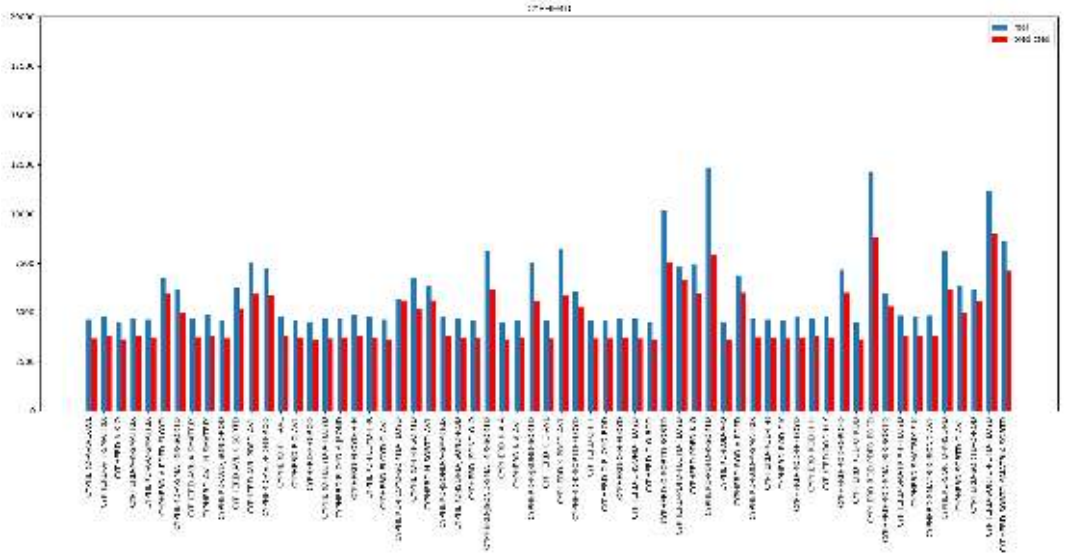
Even though, the microservices share similar characteristics, their different behavior in more resource intensive load conditions leads to a distorted prediction result. In order to improve the error rate of the prediction, a new microservice should be added to the training dataset, whose behavior is closer to the the behavior of CYPHERD microservice in more resource intensive workload combinations.



(a) Double workload combinations.



(b) Triple workload combinations.



(c) Quadruple workload combinations.

Figure 7.4: Comparison of real and predicted execution time for CYPHERD microservice.

Microservice EGG

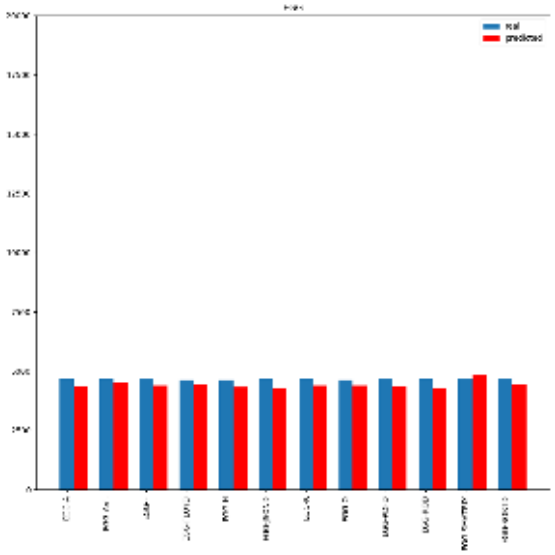
Figure 7.5 shows the predicted execution time of the `EGG` microservice in double, triple, and quadruple combinations. As a result of the prediction process, microservice `FLOYD` was selected as the most similar microservice to `EGG`.

Microservices `EGG` and `FLOYD` belong to the group of microservices that are not pushing the available resources to their extremes. However, both microservices intensively execute operations over arrays and matrices. Therefore, they generate a constant, but solid load on the CPU and on the memory bus. Given the nature of the tasks the microservices are designed for, the disk usage of the microservices during their execution is 0%. Based on the characterization result, it is apparent that the prediction process successfully selected a similar microservice from the set of available microservices.

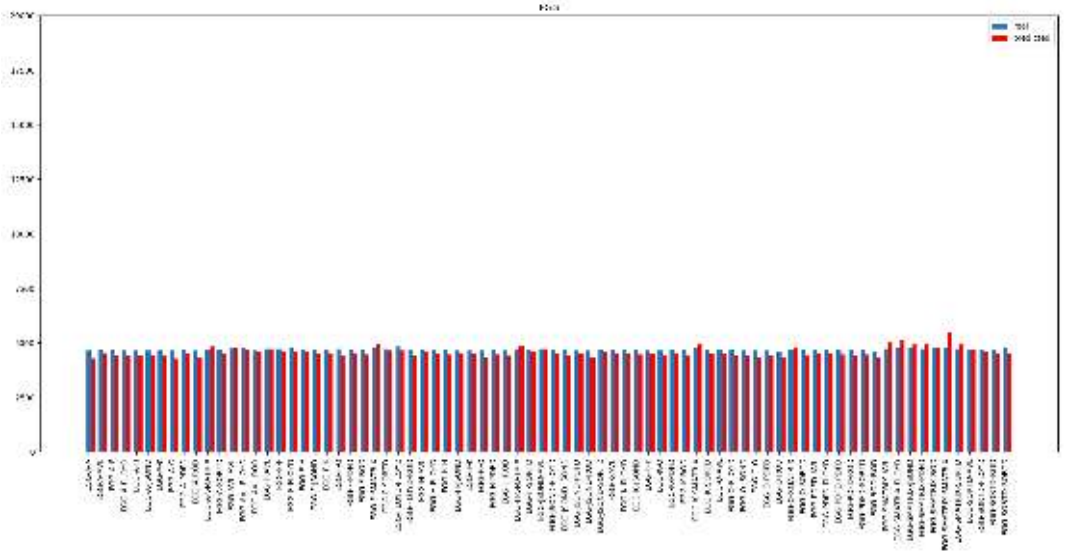
The figure shows that the predicted and the real execution time are close to each other, with a few exceptions. In quantitative terms, the differences between the predicted and the real execution time are 5.7%, 4.2%, and 4.6% in average, in double, triple, and quadruple combinations, respectively. The highest differences occur in quadruple combinations and reach up to 27.6%.

In most of the combinations, the predicted execution time stays below the real execution time, however, in some cases the predicted execution time is higher. This behavior is mainly visible among the most memory-intensive combinations of triple and quadruple workloads e.g., `EGG-SMATRIX-AVL` or `EGG-SMATRIX-F-K`. Such a memory-intensive microservice is the `SMATRIX` that repeatedly transposes the generated matrix. After analyzing the measured workload combination, the data show, that adding the `SMATRIX` as a background workload into a combination where `FLOYD` is the main workload, increases the execution time of the `FLOYD` microservice. Since, the predictor uses the combinations of the `FLOYD` microservice to predict the execution time of the combinations of the `EGG` microservice, the increased execution time of `FLOYD` in particular combinations affects the predicted execution time of the `EGG`.

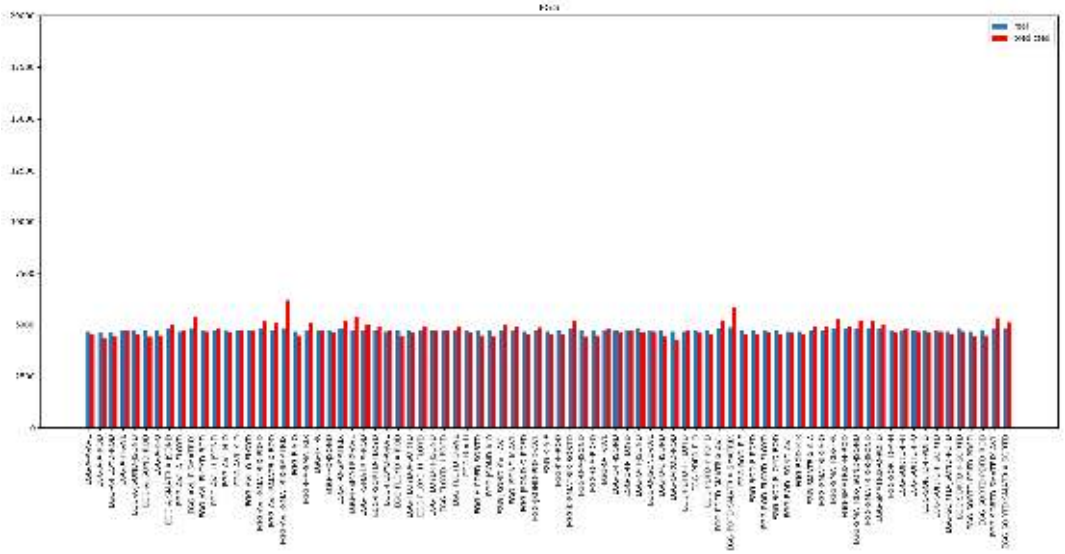
It is shown in Table 7.2, that the error rate varies between 0.02% and 27.6%, however, the average error rate does not exceed 6%. These results support our statement based on Figure 7.5 that microservice `FLOYD` is an appropriate choice as the most similar microservice to `EGG`, and enables to predict the execution time with a small error rate in average.



(a) Double workload combinations.



(b) Triple workload combinations.



(c) Quadruple workload combinations.

Figure 7.5: Comparison of real and predicted execution time for EGG microservice.

Microservice FACE

Figure 7.6 shows the predicted execution time of the `FACE` microservice in double, triple, and quadruple combinations. As a result of the prediction process, microservice `H` was selected as the most similar microservice to `FACE`.

The `FACE` microservice is designed to accept input images, detect human faces in them, and generate output images that serve as an overlay to identify the location of the faces in the submitted input images. Its most similar microservice is `H`, which represents the `apparat` benchmark from the Scala benchmark suite. The benchmark optimizes a selection of SWC files by using four tools of the Apparat framework [39]. Regarding the resource usage, the microservices generate similar resource load. Both microservices intensively exercise the CPU and the memory. Furthermore, they work with a limited amount of small files. Thus, their disk usage is minimal, however, it is noticeable.

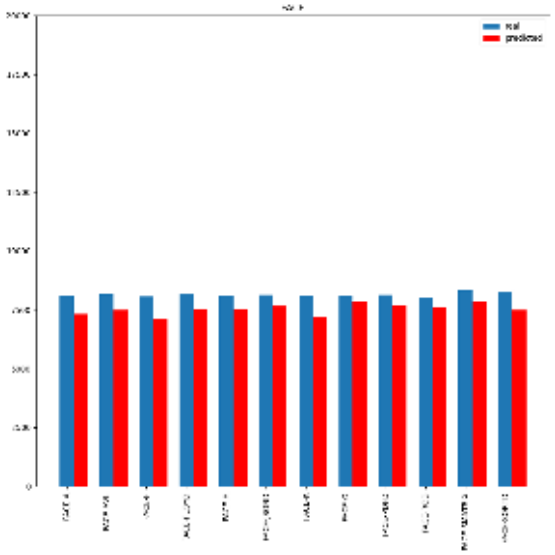
The figure shows that the predicted execution time is close to the real execution time in double, triple, and quadruple combinations. In all double and in the most of the triple combinations, the predicted execution time is below the real execution time. Contrary to this, in most of the quadruple combinations, the predicted execution time is above the real execution time.

Event though the two microservices share a very similar behavior in different workload combinations, the execution time of the `H` microservice increases in a slightly faster rate than the execution time of the `FACE` microservice. Analyzing the intermediate results of the prediction process, the average execution time ratio of single, double, triple, and quadruple combination of the `FACE` and `H` microservices is higher than the execution time ratio of the quadruple combinations of `FACE` and `H`. This causes that the predicted execution time is above the real execution time in the quadruple combinations.

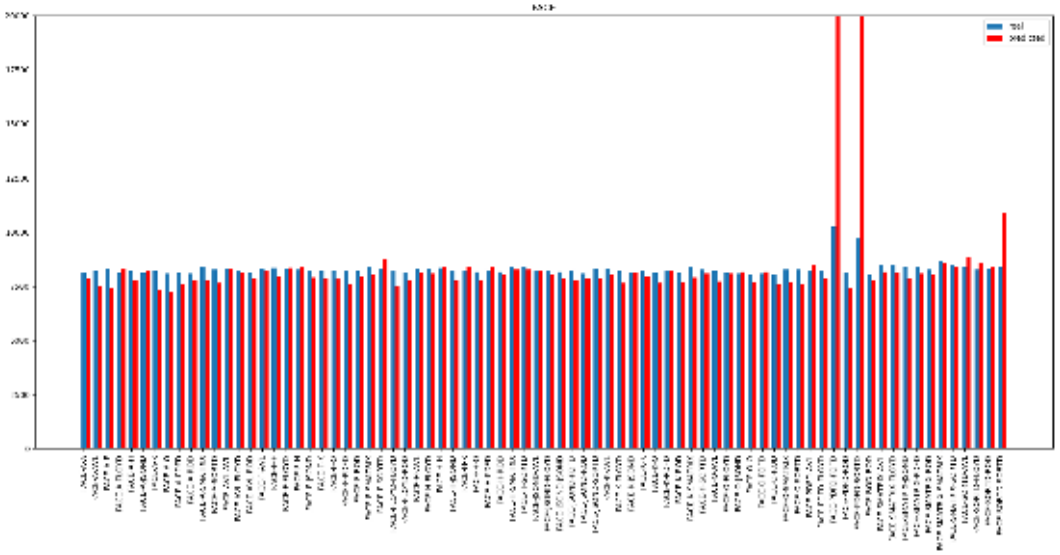
It is observable from the triple, and quadruple combinations, that there are six cases altogether, where the predicted execution time extremely differs from the real execution time. The common property of these combinations is that they include either one of the `PDFD` or the `SORTD` microservice, or both of them. These microservices represent the most resource intensive microservices that generate a solid load on the CPU, disk, and the memory at the same time. The high resource demand of the colocated workloads combined with the fact that the execution time of microservice `H` increases faster in more resource intensive combinations leads to those cases where the execution time of `H` exceed 17 seconds. Because the predicted execution time is calculated on the basis of the real execution time of the most similar microservice, it will be extremely distorted as well.

It is shown in the figure, that the difference between the predicted and the real execution time is small in the majority of combinations. The differences are 7.5%, 7.6%, and 10.3% in average, in double, triple, and quadruple combinations, respectively. There are altogether six cases in triple and quadruple combinations where the extreme inaccuracies occur.

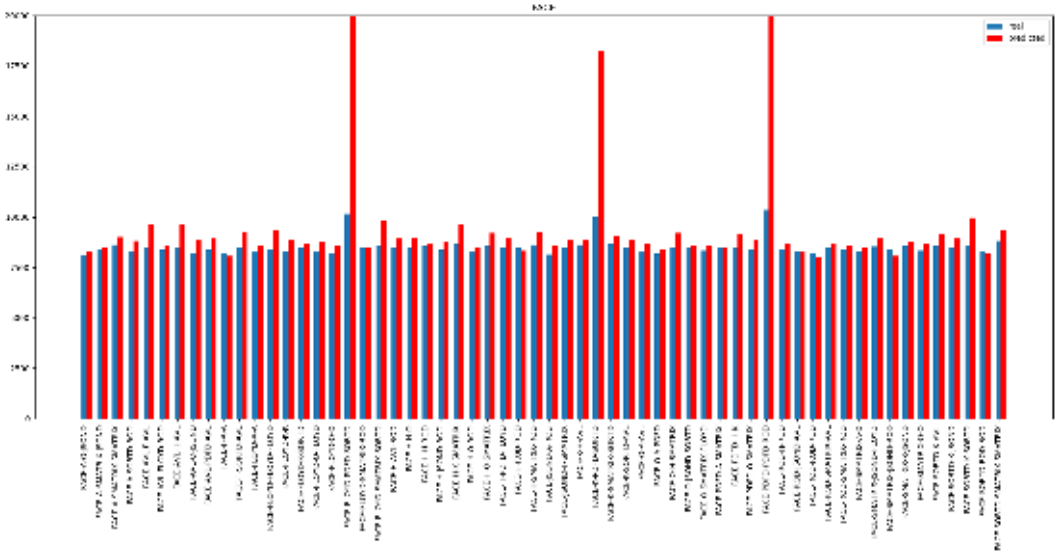
It is shown in Table 7.2, that the error rate varies between 0.1% and 140.1%, however, the average error rate does not exceed 11%. This means that even though, there are cases where the prediction is extremely invalid, in the majority of cases the predicted execution time is close to the real execution time. Therefore this confirms that microservice `F` was correctly selected as the most similar microservice to `FACE`.



(a) Double workload combinations.



(b) Triple workload combinations.



(c) Quadruple workload combinations.

Figure 7.6: Comparison of real and predicted execution time for FACE microservice.

Microservice RB

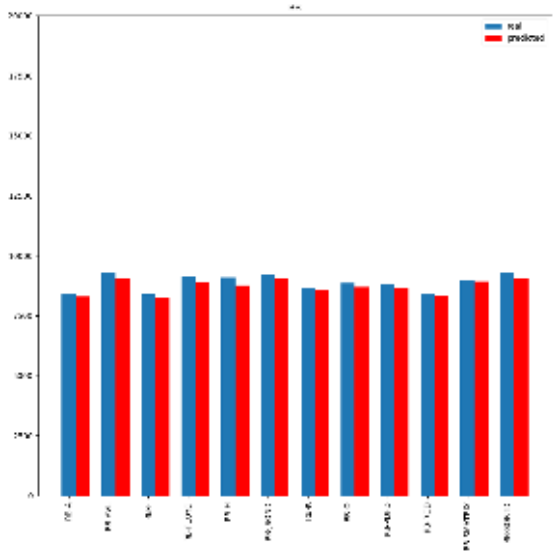
Figure 7.7 shows the predicted execution time of the RB microservice in double, triple, and quadruple combinations. As a result of the prediction process, microservice AVL was selected as the most similar microservice to RB.

Algorithms of microservices RB and AVL share the same base steps. First, both set up a tree (Red-Black and AVL tree, respectively) by inserting nodes with randomly generated numbers. In the second step, both microservices remove the nodes from the existing tree one-by-one. What they differ in, is the type of the tree they exercise on. The repeated execution of the presented algorithm generates solid CPU and memory load. Because the algorithms do not execute any file I/O operation and the disk utilization is 0% for both microservices.

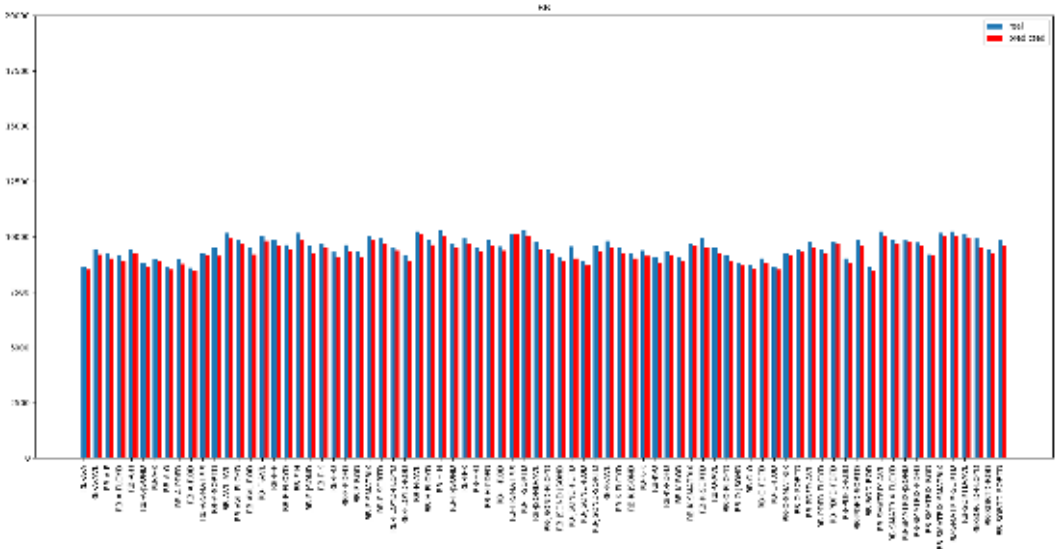
Based on this characterization, the combination of RB and AVL resembles the combination of EGG and FLOYD microservice. However, from the figures of double, triple, and quadruple combination is obvious, that the execution time of the RB microservice changes significantly in various workload combinations, while there are only negligible changes in execution time of the EGG microservice. The higher execution time is particular in combinations, where one of the background workloads include the CPU and memory intensive SMATRIX microservice. Furthermore, the characteristics of the microservices show that RB uses 30% more memory than EGG, and AVL uses 100% more memory then FLOYD. This behavior strengthens the assumption that the RB and AVL are more resource-intensive microservices than the EGG and FLOYD.

The difference between the predicted and the real execution time is very small in general, even though, the real execution time changes significantly in almost every workload combination. However, the predicted execution time always stays below the real execution time, the differences are 1.9%, 2.2%, and 2.0% in average, in double, triple, and quadruple combinations, respectively. The highest differences are present in triple combinations, but they do not exceed 6%.

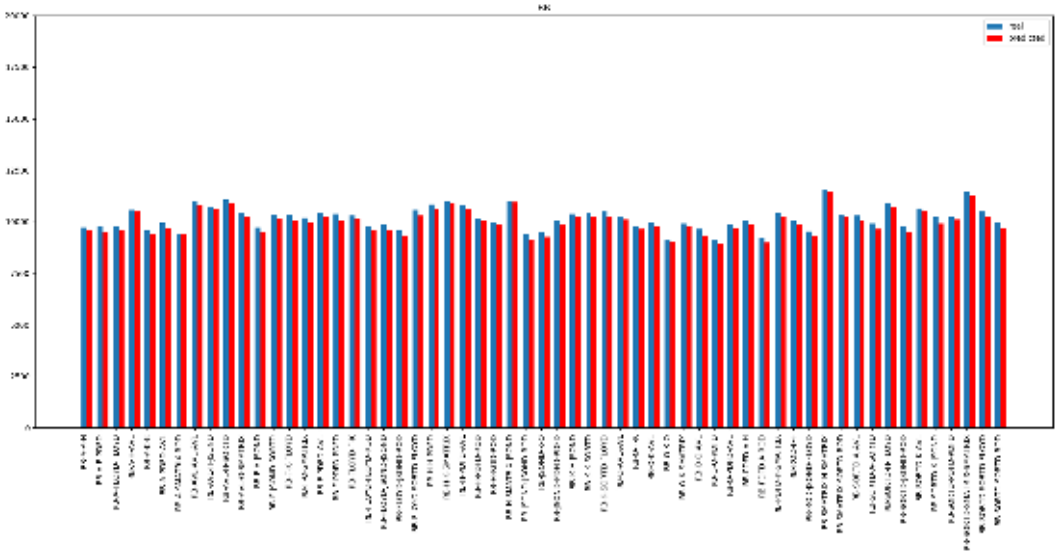
It is shown in Table 7.2, that the error rate varies between 0.3% and 5.6%, however, the average error rate does not exceed 2.2%. The small difference between the minimum, maximum, and the average error rate implies a precise prediction in every scenario, and confirms that microservice AVL is an appropriate choice as the most similar microservice to RB.



(a) Double workload combinations.



(b) Triple workload combinations.



(c) Quadruple workload combinations.

Figure 7.7: Comparison of real and predicted execution time for RB microservice.

Microservice ZB

Figure 7.8 shows the predicted execution time of the ZB microservice in double, triple, and quadruple combinations. As a result of the prediction process, microservice PDFD was selected as the most similar microservice to ZB.

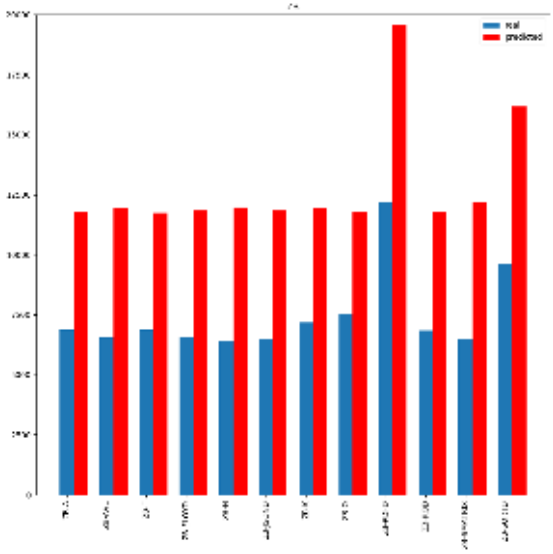
The figure shows that the differences between the predicted and the real execution time are the highest among the presented testing microservices. This unusual behavior corresponds to the fact, that microservice ZB was added into the testing dataset intentionally. Analyzing the characteristics of the ZB microservice, it is evident that this is the most resource-intensive microservice of the 17 available microservices. Because ZB extracts a 188MB zip archive, which results in 4396 files, each size of several hundreds of kilobytes, the generated disk load exceeds the disk limits even if the microservice is executed without any colocated workloads.

Based on the available characteristics, we can see that the microservice is resource-intensive enough to utilize the disk device at 100% even running as a single instance. Therefore, the various double, triple, and quadruple combinations overload the disk, and cause a massive bottleneck, which results in highly increasing execution time. For instance, the average execution time of the quadruple combination ZB-ZB-ZB-ZB is 7.37-times slower than the execution time of the single run of ZB, without background workloads. However, in case of PDFD microservice, the average execution time of the quadruple combination PDFD-PDFD-PDFD-PDFD is only 3.36-times slower than the execution time of the single run.

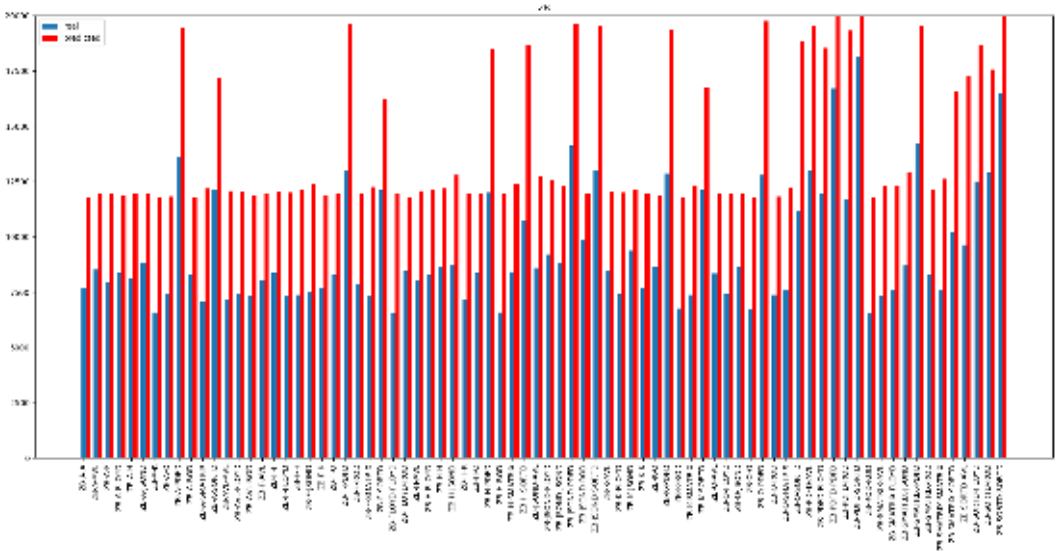
This serious difference of the scalability of the compared microservices affects the prediction process, and results in a distorted execution time ratio. This is the cause, why all double, triple, and quadruple combinations are significantly predicted above the real execution time.

It is shown in Table 7.2, that the error rate varies between 8.4% and 88.3%. The average error rate is 73.8%, 53.6%, and 42.6% for double, triple, and quadruple combinations, which makes ZB the worst predicted microservice among all test microservices.

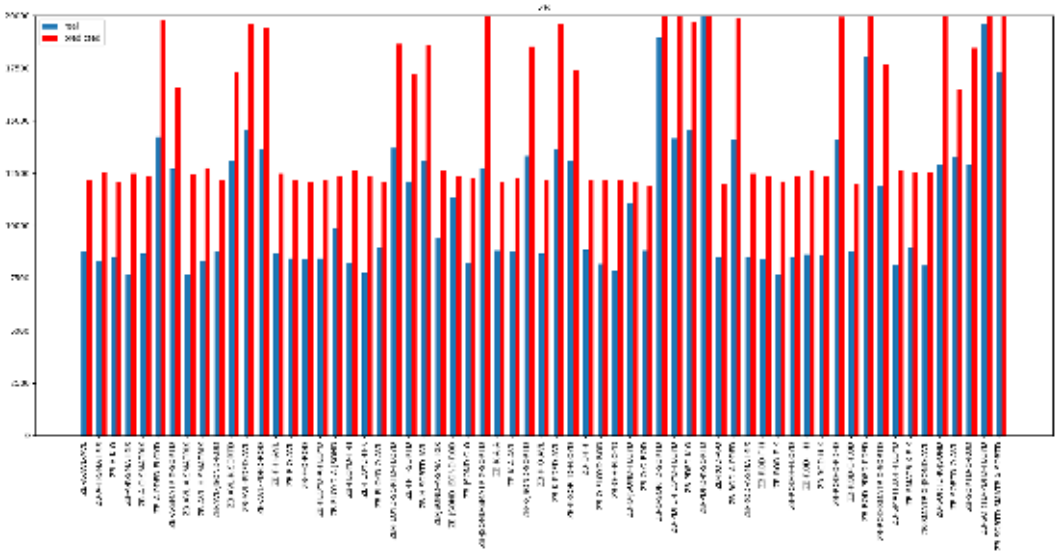
The presented test-case is a perfect example to show the weakness of our prediction process. The correctness of the prediction process highly depends on the availability of a microservice from the training dataset that is similar to the submitted microservice. If the training dataset does not include such a microservice, the prediction process will not be able to provide a reliable prediction.



(a) Double workload combinations.



(b) Triple workload combinations.



(c) Quadruple workload combinations.

Figure 7.8: Comparison of real and predicted execution time for ZB microservice.

Prediction Input & Output

In Section 5.1.2 we presented the structure of the requirements specification file and described how it is used to define the time and probability requirements of the microservice. This file serves as one of the input files of the prediction process and determines the cases the predictor should predict.

As we described in the previous part, the predictor predicts the execution time of the submitted microservice colocated with the available double, triple, and quadruple combinations of its most similar microservice. To decide whether the specified time and probability requirements can be met, the predictor evaluates the predicted execution time with respect to the submitted requirements. If the submitted requirements specification lists more than one requirement, the predictor subsequently evaluates every requirement. At the end of the prediction process, the predictor summarizes the prediction results into a programmatically generated YAML output file which is available for the microservice developer in a predefined location. Listing 7.2 shows a sample of the generated output file.

The prediction result contains exactly one `MultiRequirementResult` node, which includes the results of prediction for each case defined in the submitted requirements specification file.

Input requirements: In the first part (lines 2-10), the prediction result file contains the input requirements. They help the microservice developer to identify the microservice and the requirements that this prediction result refers to.

Output requirements: The most important content of the prediction result is presented within the `_output_requirement_results` node (lines 11-40). The node includes a list of `RequirementResult` nodes. In this particular example, there are 8 nodes in the list, because every `Requirement` from the input requirements specification is evaluated with the 4 types of workload combinations (single, double, triple, and quadruple) we measure. Each `RequirementResult` node has 3 attributes and an internal node. The `_input_requirement` attribute identifies the requirement to which this result belongs to. The `_overall_result` attribute specifies whether the referred requirements are met for this particular set of workload combinations. The `_probability` attribute expresses the percentage of combinations that met the time limit. The attributes are followed by the `_predicted_combination_result` node. The node includes a list of `PredictedCombinationResult` nodes that represent the result of the prediction for that particular workload combination. Each `PredictedCombinationResult` consists of 5 attributes. The `_average_runtime_ratio` expresses the execution time ratio introduced in the previous section. Attributes `_new_microservice_id` and `_closest_microservice_id` show the ID of the currently submitted and its most similar microservice. The `_background_combination` attribute shows the background workloads of that particular workload combination. If the value is set to `null`, there are no background workloads. The `_predicted_combination_avg_runtime` attribute shows the predicted runtime of the workload combination in milliseconds.

Predicted workload combinations: Beyond the general overview presented by the `RequirementResult` nodes, the prediction result includes data about every predicted workload combination. These are divided into the following four nodes, according to the arity of the combination:

- Node `_single_predicted_combination_result`.
- Node `_double_predicted_combination_result`.
- Node `_triple_predicted_combination_result`.
- Node `_quadruple_predicted_combination_result`.

Each node contains a list of `PredictedCombinationResult` nodes, which describe the details of the predicted workload combination.

```yaml
!yamlable/result.requirement.MultiRequirementResult
_input_requirements:
- &id001 !!python/object:requirements.requirement.Requirement
  _microservice_id: AVL
  _probability: 100
  _runtime: 10000
- &id003 !!python/object:requirements.requirement.Requirement
  _microservice_id: AVL
  _probability: 70
  _runtime: 9500
_output_requirement_results:
- !yamlable/result.requirement.RequirementResult
  _input_requirement: *id001
  _overall_result: true
  _predicted_combination_results: &id004
  - !yamlable/result.predictor.PredictedCombinationResult # single combination
    _average_runtime_ratio: 1.0
    _background_combination: null
    _closest_microservice_id: AVL
    _new_microservice_id: AVL
    _predicted_combination_avg_runtime: 9320.548
  _probability: 100
- !yamlable/result.requirement.RequirementResult
  _input_requirement: *id001
  _overall_result: false
  _predicted_combination_results: *id002
  _probability: 29
- !yamlable/result.requirement.RequirementResult
  _input_requirement: *id001
  _overall_result: false
  _predicted_combination_results: &id005
  - !yamlable/result.predictor.PredictedCombinationResult # triple combination
    _average_runtime_ratio: 1.1393649242893078
    _background_combination: A-A
    _closest_microservice_id: RB
    _new_microservice_id: AVL
    _predicted_combination_avg_runtime: 9876.024278659113
   # ... additional PredictedCombinationResults for triples
  _probability: 7
# ... additional RequirementResults for the remaining combinations
_single_predicted_combination_result: *id004
_double_predicted_combination_result: &id002
- !yamlable/result.predictor.PredictedCombinationResult # double combination
  _average_runtime_ratio: 1.1393649242893078
  _background_combination: A
  _closest_microservice_id: RB
  _new_microservice_id: AVL
  _predicted_combination_avg_runtime: 9611.226755334885
  # ... additional PredictedCombinationResults for doubles
_triple_predicted_combination_result: *id005
_quadruple_predicted_combination_result: *id006
```

Listing 7.2: Example of prediction result.

8. Implementation

In this chapter, we focus on the architecture and the implementation details of the final system. We highlight the main components, and the possibilities to extend the current solution.

First, we show the system architecture diagram and enumerate its main components. Then, we describe every component in a separate section with its UML class diagram, that visualizes the associated important classes, methods, and attributes. A more detailed description of each class and method can be found in the generated documentation.

8.1 Architecture

This section presents the overall architecture and introduces the main components of the system. As it is clear from Figure 8.1, the architecture consists of multiple components that can be further divided into sub-components. The system is a combination of applications written in Python and Java programming languages. The components running on different machines communicate via REST API. Since the communication between the components is rare and requires to transfer only a small amount of structured data, therefore REST is an ideal choice. Here, we list main components that we will describe in more detail in the next sections.

- Orchestrator: It is written in Python language and is considered as a central component of the architecture. It enables the microservice developer to submit the microservice into the deployment framework and manages the microservice deployment process. A dedicated server is used for its deployment.

- Agent: It is responsible for the microservice measurement process. There can be multiple instances of agents in the deployment framework, each running on a separated server. The agent communicates with the orchestrator via REST API, and is written in Python language.

- Measuring framework: The most important component of the system. It is enabled to run multiple measuring framework instances inside one agent, however, the instances should be executed in separated Docker containers. The measuring framework is written in Java language.

- Communication interface: It enables the measuring framework to load, execute, and measure the submitted microservices. It also provides an API for communication between the measurement framework and the microservice. The communication interface is written in Java language.

- Microservice: These are the concrete implementations written in Java language and represent real-life workloads. Each microservice is executed by the measuring framework inside the Docker container. The microservice communicates directly with its measuring framework because both implement the custom communication interface.

- Predictor: It analyzes the collected data and predicts the microservice execution time in various workload combinations. The predictor is written in Python language.

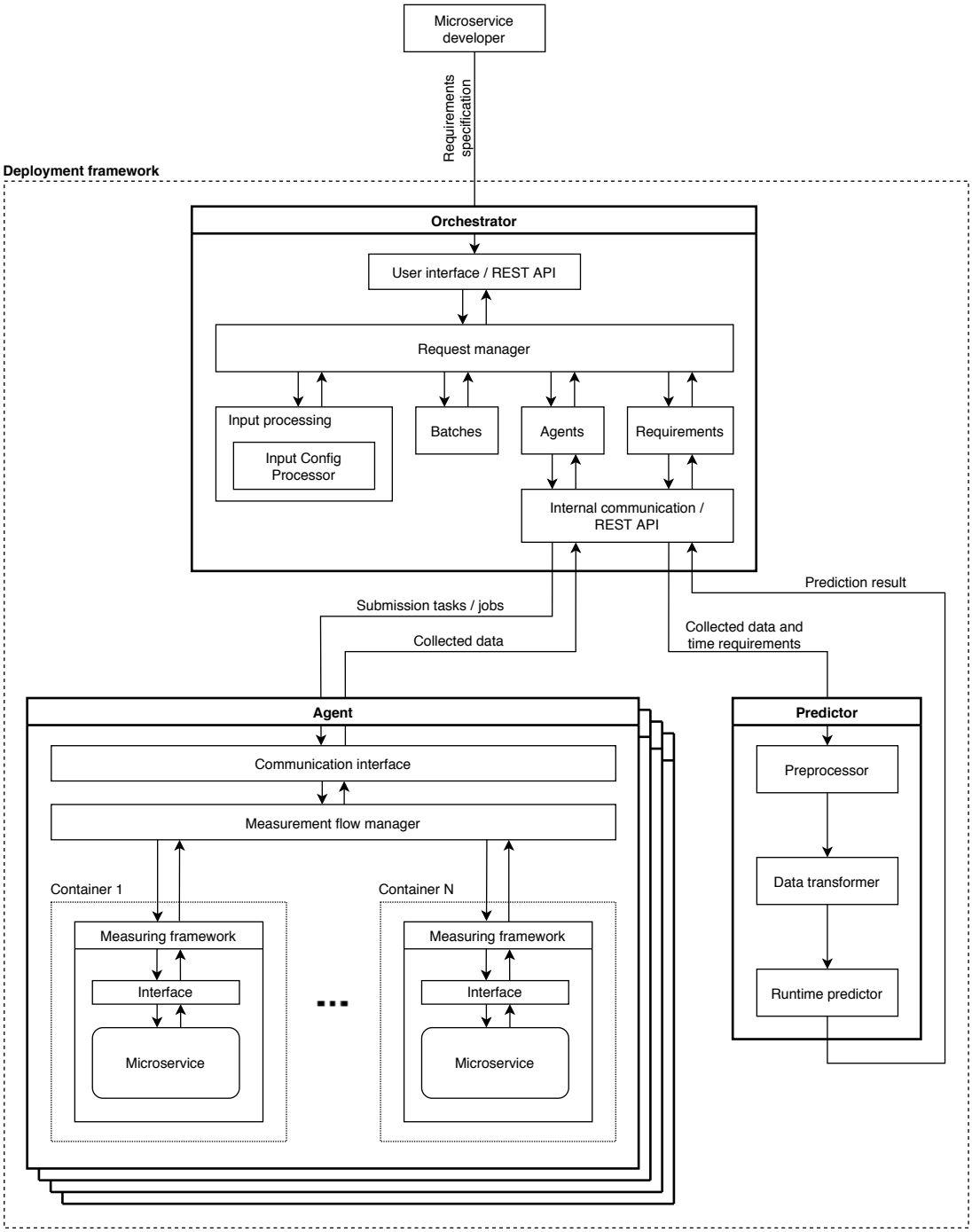


Figure 8.1: System architecture.

8.2 Orchestrator

The orchestrator is the entry point of the deployment framework and represents a bridge between the microservice developer and the internal parts of the framework. Its main responsibility is to enable the microservice developer to submit the microservice along with its requirements. It also distributes the microservice measurement among the cluster of measuring agents and passes the collected data to the prediction process.

In this section, we show how the orchestrator communicates with the microservice developer and the internal parts of the system. Then, we present each main component separately.

8.2.1 Communication Interface

REST API

The default communication method between the orchestrator and other parties is implemented via REST APIs, which exchange data in JSON format.

The `RequestManager` exposes an endpoint for every main functionality related to submitting the requirements specifications, managing the batches, and enabling the communication with the agents.

The REST APIs are realized using the Flask Framework[1], that is a third-party library used for developing web applications in Python.

Administrator UI

Beyond the REST API, the orchestrator also provides an optional command-like user interface. This is mostly used for the testing purposes and enables a faster and more simple interaction with the application. It is implemented by the `AdministratorUI` that extends the built-in `Cmd`[2] class. The command-like interface offers the same functionalities as the REST API does.

8.2.2 Main Parts

Input Processing

The requirements specification submitted by the microservice developer is passed to the `InputManager` that is responsible for parsing the input files. The `InputManager` processes the single requirements specification file into two separate files: batch configuration and requirements of the microservice. The `InputManager` uses the `InputConifgProcessor` to generate the batch configuration file. The `InputConifgProcessor` is a separate application and its purpose is to generate the batch configuration from the input file based on a predefined pattern. The requirements are simply extracted from the requirements specification file and are stored in a separated requirements file.

[1] `https://flask.palletsprojects.com/`, version 1.0.2
[2] `https://docs.python.org/3.6/library/cmd.html`

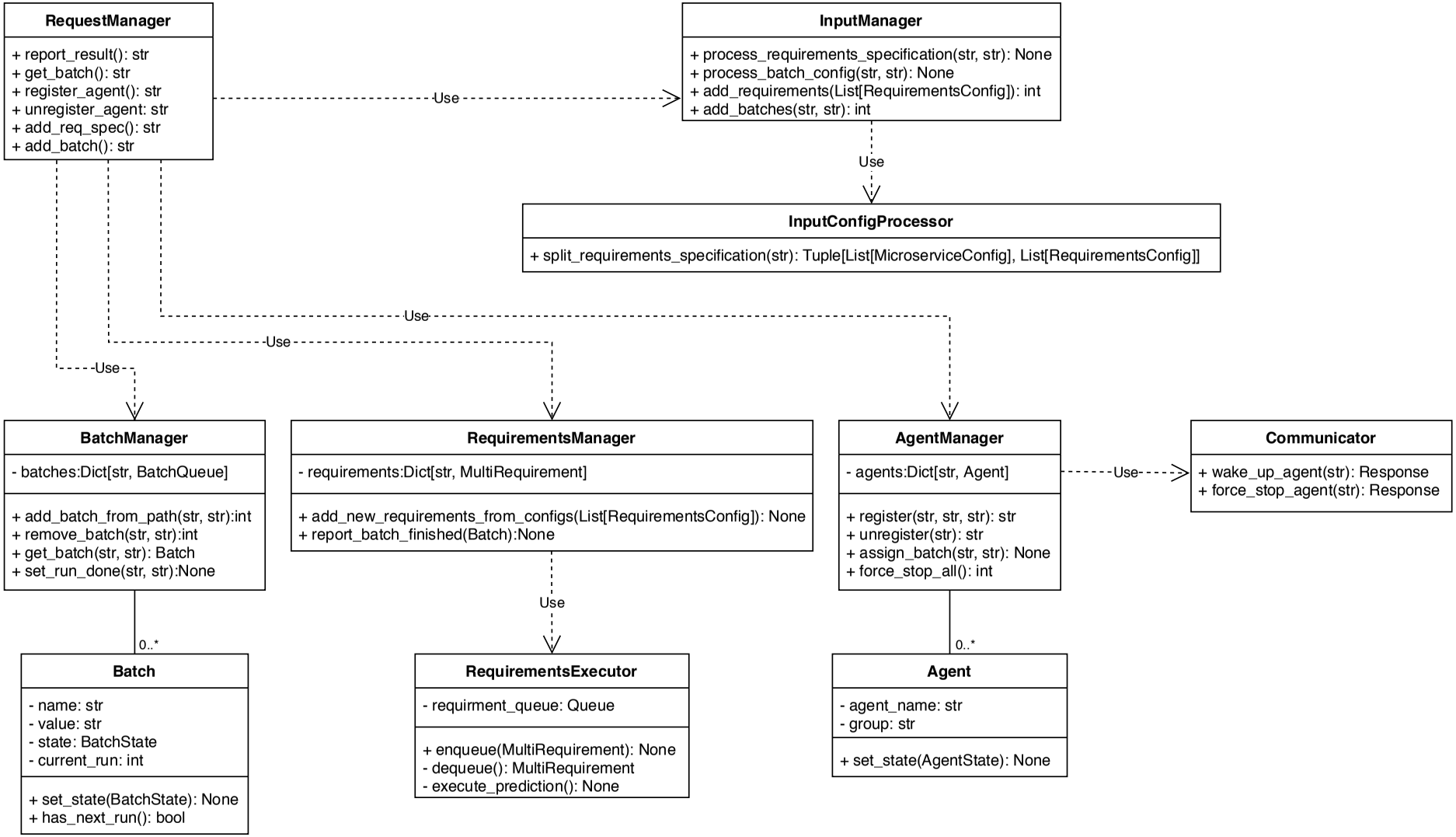


Figure 8.2: UML class diagram of the orchestrator.

Batch Management

The generated batch configurations are represented as instances of `Batch` and are managed by the `BatchManager`, that allows to add and remove batches, or to update their current states.

The `BatchManager` organizes the batches into batch groups. By default, the batches are added into the `default` group. These batch groups allows to gather batches of the same type and to assign them to agents that provide a specific environment. The agents assigned to different groups may differ in their settings, therefore batches from different groups can be measured in different environments.

Furthermore, the `BatchManager` is responsible for distributing the batches among the agents, and managing the state of the batches during the execution. The possible states of the batch are defined in the `BatchState` enum. The various batch states enable the `BatchManager` to distinguish between the batches already executed and the batches waiting for execution.

The batches are stored in `BatchQueue`, a queue-like data structure which holds the batches. There is a separate `BatchQueue` instance for every batch group. When the next batch is requested from the queue, the data structure returns the first batch which is in `BatchState.WAITING` state or returns `None` if no such batch exists.

Requirements Management

The `RequirementsManager` keeps track of the submitted requirements and provides methods for managing them. The requirements submitted by the microservice developer are represented as an instance of `Requirement`. If there are more requirements for a single microservice, they are wrapped into an instance of the `MultiRequirement`.

As it is described in Chapter 7, the requirements are evaluated in the last step of the prediction process. However, the prediction process cannot be started until all the batches of a particular requirement are successfully measured. To be able to start the prediction process and to establish the communication between the orchestrator and the predictor, the `RequirementsExecutor` is used. It enables the orchestrator to communicate with the predictor, and to pass the requirements and the collected data to the predictor. Furthermore, it ensures that the prediction result is returned to the orchestrator as an instance of the `MultiRequirementResult`.

Agent Management

To be able to communicate with the agents and to distribute the batches among them, there is a need for `AgentManager`. Agents are represented by the `Agent` class that includes the most important properties of the agents. For instance, each agent has its unique name, unique address, name of the currently assigned batch, etc. The `AgentManager` manages the agent registration - assigning it to the appropriate batch group, unregistration and monitoring the agent's state. Furthermore, it enables the orchestrator to suspend the agent for a while or to immediately terminate the agent.

8.3 Agent

The agent is responsible for requesting new batches, managing the execution flow of batch tasks included in the particular batch configuration, and returning the measurement result to the orchestrator. As it is described in Chapter 5, the batch configuration includes optional initial and final tasks besides the mandatory microservice tasks. Therefore, the main objective is to design a task executor application and a public interface that enables to execute various tasks specified in the batch configuration.

First, we describe the main concept of the agent application life-cycle. Then we present the architecture in two steps. In the first step, we show the mechanism that accepts the batch configuration file from the orchestrator and parses the task definitions. In the second step, we show the general task execution mechanism that executes the parsed tasks.

8.3.1 Application Life-cycle

The agent application has a predefined life-cycle that helps the application to run without user interaction. This life-cycle defines the crucial states that the application goes through during its execution. The application life-cycle is controlled by the `MainExecutor` class, which implements the appropriate mechanisms to control the transitions between the states during the application life-cycle. The life-cycle consists of multiple states that cover the main execution flow and the corner cases as well. In this section, we describe the most important steps of the life-cycle. The entire life-cycle and the transitions between the states are described in the documentation of the `MainExecutor` class.

- Registered: The agent successfully establishes the communication with the orchestrator and registers itself as an active agent. Registering the agent to orchestrator ensures that every iteration of the batch execution will be executed on the agent that the batch was first assigned to. No other agent can register at the orchestrator using the same agent name.
- Stand-by: The agent is ready to execute the batch. If it does not get force stopped in the meanwhile, it keeps querying the orchestrator for a batch to execute in every 10 minutes until it gets a batch. If a batch is received, the execution continues in the Executing state.
- Executing: The batch is being executed. The execution is done if it is successfully finished, force stopped, or failed to execute the batch. Regardless the result of the batch execution, the application execution continues in the Execution done state.
- Execution done: The batch execution has been finished. In this state, the agent creates the result based on the outcome of the batch execution and reports this result to the orchestrator.
- Restarting: The agent executes the `reboot` command, which restarts the underlying operating system.

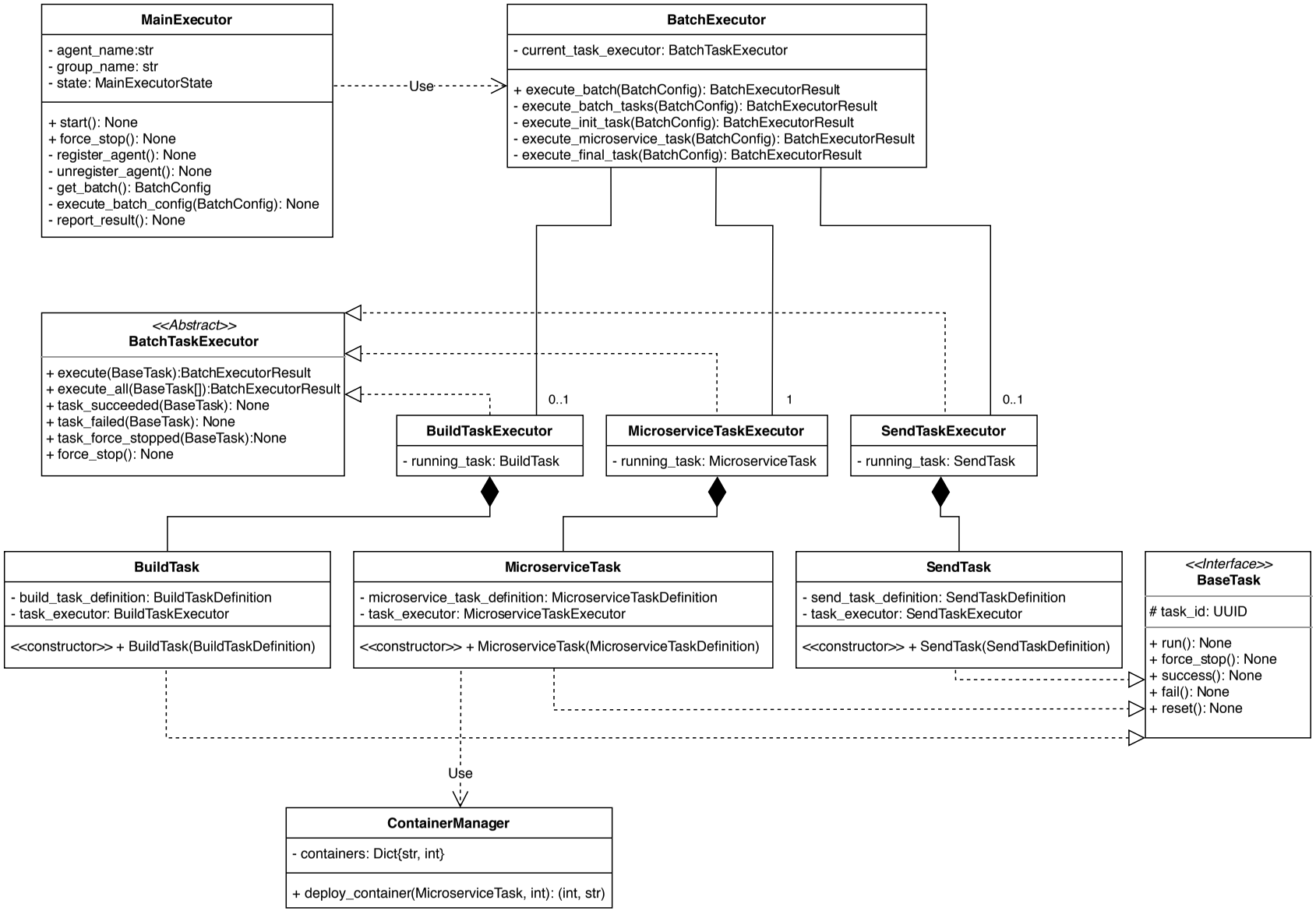


Figure 8.3: UML class diagram of the agent.

- Unregistered: The agent is unregistered from the orchestrator and will not query for new batches. This state can be reached only when the agent execution is force stopped by the orchestrator.
- Failed: There was a critical failure during the life-cycle e.g., failed to register, sign in, sign out, report result or unregister. The agent tries to report the failure to the orchestrator and terminates the application.

8.3.2 Input Processing

The input processing is a sequence of steps that parses the received batch configuration into executable tasks. In the following, we use Figure 8.4 to describe the architecture of input processing and to present the main classes involved in each step. In order to show every case of the process, we assume that the batch configuration file contains concrete initial and final tasks besides the mandatory microservice task.

The parsing process starts by receiving the batch configuration file in JSON format from the orchestrator. Then the entire parsing process is managed by the `BatchParser` class.

Step 1. The `BatchParser` splits the JSON file into smaller sections: main attributes of the batch configuration and task definitions. Then, the `BatchParser` passes each task definition to the corresponding task definition parser.

Step 2. Besides the required attributes, every task definition contains unique attributes. Therefore, it is required to implement a custom parser that parses the attributes of that particular task definition. In our solution, the *build task*, *microservice task* and *send task* definitions (see Section 5.1.2) are parsed by the following custom parsers: `BuildTaskDefinitionParser`, `MicroserviceTaskDefinitionParser` and `SendTaskDefinitionParser`.

The results of the task definition parsers are returned as concrete implementations of the abstract `TaskDefiniton` class, that store the attributes of the corresponding task definition. The concrete implementations are the following: `BuildTaskDefinition`, `MicroserviceTaskDefinition` and `SendTaskDefinition`.

Step 3. The parsed task definitions are returned to the `BatchParser`, that passes them to the `TaskCreator`. The `TaskCreator` is responsible for converting the task definitions into executable tasks depending on the type of the task definition. The executable tasks must extend the `BaseTask` abstract class that defines the necessary methods for tasks execution e.g., start, stop, reset, change state, etc. In our solution, the concrete implementations of the `BaseTask` are the following: `BuildTask`, `MicroserviceTask`, and `SendTask`.

Step 4. The `TaskCreator` returns the executable tasks to the `BatchParser`. Then, the `BatchParser` creates an instance of the `BatchConfig` class that is responsible for representing the parsed batch configuration file. In the last step, the `BatchParser` adds the already parsed main attributes of the batch configuration and the executable tasks to the `BatchConfig` instance.

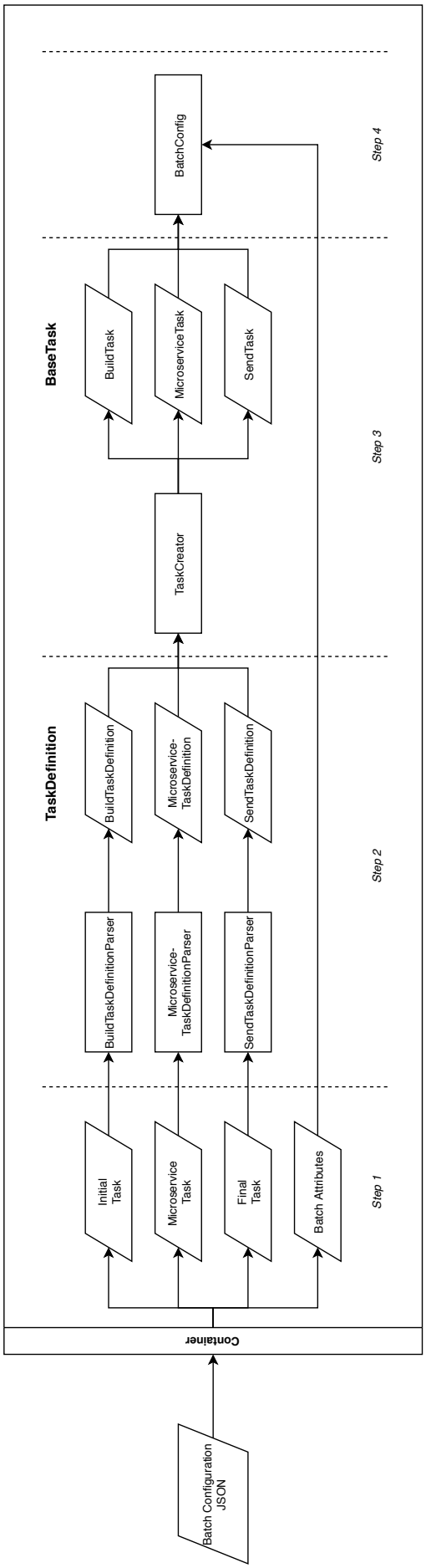


Figure 8.4: Parsing the batch configuration file.

8.3.3 Task Execution

The goal is to design a general task executor that is able to execute mandatory microservice tasks and optional user-defined tasks defined in the batch configuration file. Since, the optional tasks can be implemented and added by the microservice developer, we designed the `BaseTask` interface that enables the task executor to execute them. By implementing the `BaseTask` interface, the tasks are split into main steps that represent critical parts of the task execution process. In order to successfully execute the steps, and to handle their possible failures without letting to fail the entire measuring process, the concrete `BaseTask` instances are executed by particular task executors. The `BatchTaskExecutor` defines an interface that allows to safely execute the tasks. Therefore, for each concrete `BaseTask` there must be implemented a concrete `BatchTaskExecutor`, that executes the particular `BaseTask` step-by-step. During the execution, each step reports its result to the executor. Therefore, the execution is always supervised and the odd events can be immediately handled.

The `BatchExecutor` manages the execution of batch tasks within the main execution process. The batch execution process is divided into three phases, where each phase executes a particular type of task. The tasks are executed sequentially in the following order: initial, microservice, and final task. The `BatchExecutor` executes every concrete instance of the `BaseTask` by using task's concrete `BatchTaskExecutor`. The concrete batch task executors for the initial, microservice and final tasks are the following: `BuildTaskExecutor`, `MicroserviceTaskExecutor`, and `SendTaskExecutor`. The `BatchExecutor` ensures that the tasks are executed in this order, and reports the result of each task execution to the `MainExecutor`.

8.4 Measuring Framework

The measuring framework is executed in the second step of the batch execution process. The purpose of the measuring framework is to provide an environment that enables to iteratively execute the microservice and measure its resource usage. The measuring framework is executed inside a Docker container, therefore there can be multiple instances running in one agent in parallel, each in a separated Docker container. Even though, the resource usage of a particular microservice is not measured i.e., the microservice represents a background workload, the measuring framework is required to manage the microservice execution.

In order to be able to run the measuring framework, the following preconditions should be met during the initialization step of the batch execution process: the appropriate Docker image is built and includes the assets required for executing and measuring the microservice. These assets are: the measuring framework, the submitted microservice, and the dependencies that will be required during the execution.

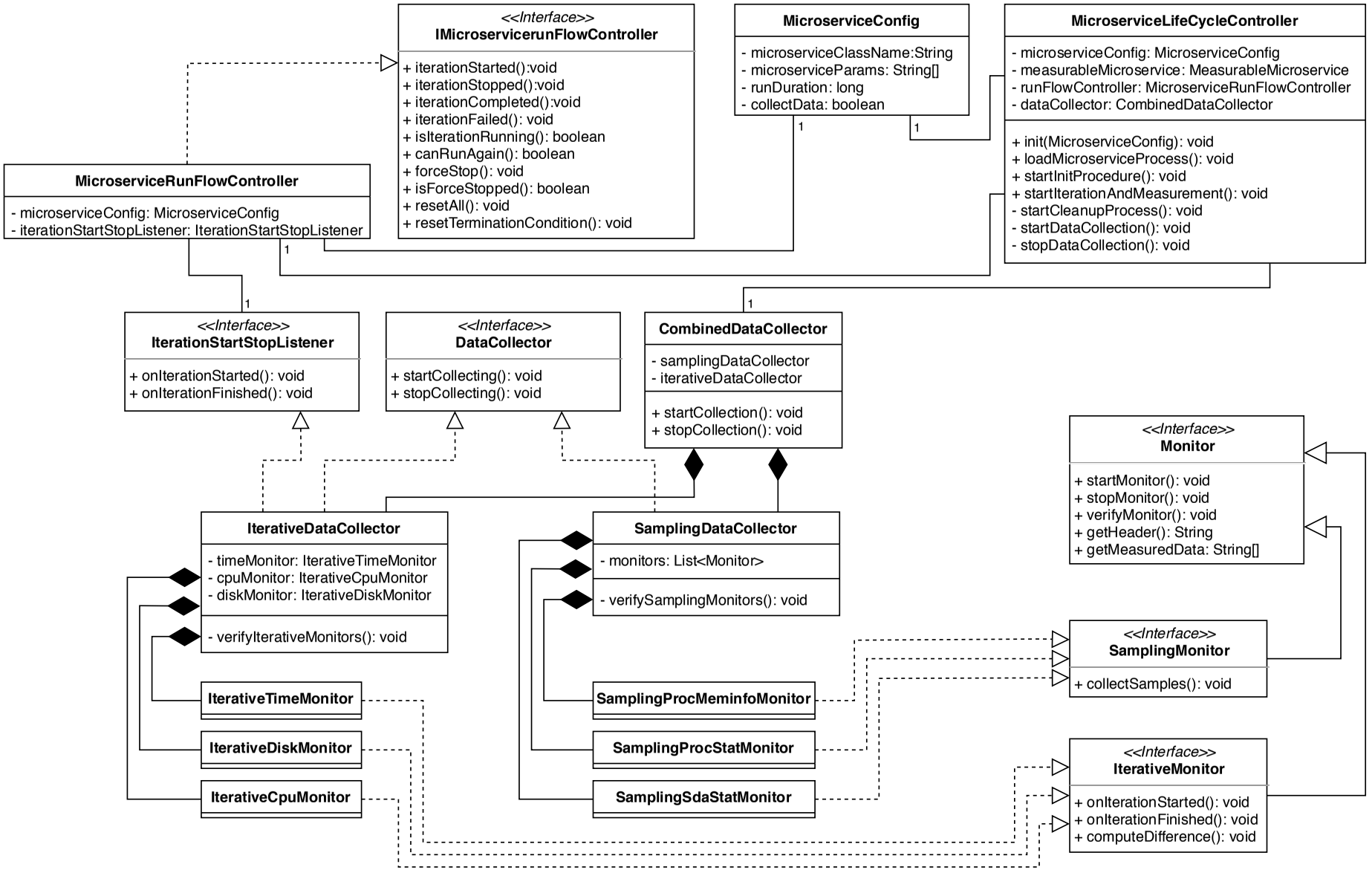


Figure 8.5: UML class diagram of the measuring framework.

8.4.1 Microservice Execution

Main Controller

As described in Chapter 6 and in Section 8.5, the second step of the batch execution process can be further divided into three phases of microservice execution. To be able to control the execution of each microservice phase, we use the `MicroserviceLifeCycleController` (`MLCC`). The `MLCC` is responsible for managing the microservice execution flow, handling the success, and failure for each phase.

Initialization

However, before the `MLCC` can control the execution of phases, the measuring framework should be initialized. The properties of the microservice execution and measurement are passed by command-line parameters to the Docker container, that uses them to start the measuring framework. The measuring framework parses the parameters and creates an instance of the `MicroserviceConfig` class. The `MicroserviceConfig` includes the most important properties of the microservice: class-path of the microservice test-point, configuration of the microservice, etc. Then the measuring framework initializes the `MLCC` by passing this `MicroserviceConfig` instance to it.

Class Loading

The `MLCC` initialization process is followed by loading and initializing the microservice. The `MLCC` uses the class-path from the given `MicroserviceConfig` instance to load the concrete implementation of the `MeasurableMicroservice` interface that represents the microservice test-point. By calling the public methods of this interface, the `MLCC` will able to execute the phases of the microservice.

Controlling the Microservice Execution

The *initialization* and *cleanup* phases usually consist of a few simple steps. However, the *iterate and measure* phase is more complicated and requires an intensive communication between the measuring framework and the microservice. To manage this phase and to ensure the communication between the parties, the `MLCC` uses a concrete implementation of the `IMicroserviceRunFlowController`, the `MicroserviceRunFlowController`. This run flow controller enables the communication between the measuring framework and the microservice: reporting when the iteration starts or finishes, succeeds or fails. Furthermore, it guarantees that the microservice execution does not exceed the specified measurement time (12 minutes in our case).

Reporting Sub-results.

Beyond the management of the microservice execution, the `MLCC` maintains the communication with the agent. The `MLCC` reports the result of each phase to the agent that processes the reported result and notifies the orchestrator if necessary. The communication between the measuring framework and the agent is based on HTTP requests.

8.4.2 Data Collection

The most important part of the measuring framework is data collection that includes measuring the microservice resource usage and behavior during its iterative execution.

We have implemented two different data collection approaches: iterative and sampling data collection. The former collects the microservice resource usage within each microservice iteration, while the latter uses an iteration-independent data collection scheduled at a predefined frequency.

Data Collectors

Even though the approaches significantly differ in their implementation details, they are required to execute the same actions: start and stop the data collection process and return the collected data. Therefore, we designed the `DataCollector` interface that the concrete data collectors should implement in order to satisfy these requirements. By implementing the `DataCollector` interface, one is not limited to use only the existing approaches but can extend the data collection process with additional data collector implementations.

However, the high amount of concrete data collectors significantly increases the complexity of their management and coordination with the microservice execution. To simplify their management, we decided to add an additional abstraction layer above the concrete data controllers. This is realized by the `CombinedDataCollector`, that gathers the concrete data controllers implementing the `DataCollector` interface, and ensures their management.

In our solution, the `CombinedDataCollector` manages two concrete implementations of the `DataCollector` interface: the `IterativeDataCollector` and the `SamplingDataCollector`.

The `IterativeDataCollector` manages the data collection at the beginning and at the end of each microservice iteration. This concrete data collector implements the `IterationStartStopListener` interface as well, that enables the microservice to notify the `IterativeDataCollector` when the iteration starts and finishes. At the end of each iteration, the `IterativeDataCollector` calculates the difference of the collected data to get the amount of resources used within one microservice iteration.

The `SamplingDataCollector` manages periodical data collection regardless of the current iteration of the microservice. The `SamplingDataCollector` samples the microservice resource usage until the overall data collection is enabled by the `MicroserviceLifeCycleController`.

The `IterativeDataCollector` and the `SamplingDataCollector` do not implement concrete data collection algorithms by themselves, but they use specific *data monitors*, which are designed for this purpose.

Data Monitors

Data monitors are responsible for collecting the usage of a particular hardware resource. Every data monitor implements a specific data collection algorithm, designed for that particular hardware that the data monitor focuses on. The data monitors implement the `Monitor` interface. The interface enables to start

and stop the data collection and to return the collected data. To ensure the compatibility with the iterative and sampling data collectors, the `Monitor` interface is extended by the `IterativeMonitor` and `SamplingMonitor` interfaces.

The `IterativeMonitor` interface enables to implement concrete iterative data monitor, that focuses on a particular hardware resource and collects its usage in every iteration of the microservice. To collect data about multiple resources, we have implemented the following iterative data monitors that are managed by the `IterativeDataCollector`: `IterativeCpuMonitor` monitors CPU usage, `IterativeDiskMonitor` monitors disk I/O usage, and `IterativeTimeMonitor` monitors the microservice execution time.

The `SamplingMonitor` interface enables to implement concrete sampling data monitor that repeatedly collects the usage of a particular hardware resource using a predefined sampling period. To collect data about multiple resources, we have implemented the following sampling data monitors that are managed by the `SamplingDataCollector`: `SamplingProcStatMonitor` monitors CPU usage, `SamplingSdaStatMonitor` monitors disk I/O usage, and `SamplingProcMemInfoMonitor` monitors memory usage.

8.5 Communication Interface

The communication interface library enables the communication between the measuring framework and the microservice. The library includes the following two interfaces: `MeasurableMicroservice` and `IMicroserviceRunFlowController`. See Figure 8.6 for the UML class diagram.

The `MeasurableMicroservice` interface enables the measuring framework to load the microservice and manage the microservice execution flow that includes following main phases: initialize the working environment and the microservice, iteratively execute the microservice, measure its resource usage, and clean up the working environment. The more detailed analysis of the phases is presented in Chapter 6. The `MeasurableMicroservice` interface assigns one particular method for each of the main microservice phases, therefore they can be easily executed from the measuring framework. At the end of each phase, the methods return the execution result as one constant of the `MicroserviceResult` enumeration.

The `IMicroserviceRunFlowController` interface enables the microservice to execute callbacks to the measuring framework. The callbacks include reporting the start and stop of execution of current microservice iteration or checking whether the next iteration can be executed.

Listing 8.1 shows an example implementation of the `iterateAndMeasure()` method of the `MeasurableMicroservice` interface. This method is called to execute the second phase, in which the microservice is executed iteratively and its resource usage is measured until the `IMicroserviceRunFlowController` enables the next iteration.

The execution of the microservice should be bundled into a loop, which enables to execute the microservice repeatedly. The condition of while-loop must include a call of `canRunAgain()` to ensure that the microservice iterations do not exceed the total length of execution. Moreover, it can be also used to terminate the iterative execution if the measurement was force stopped by the orchestrator.

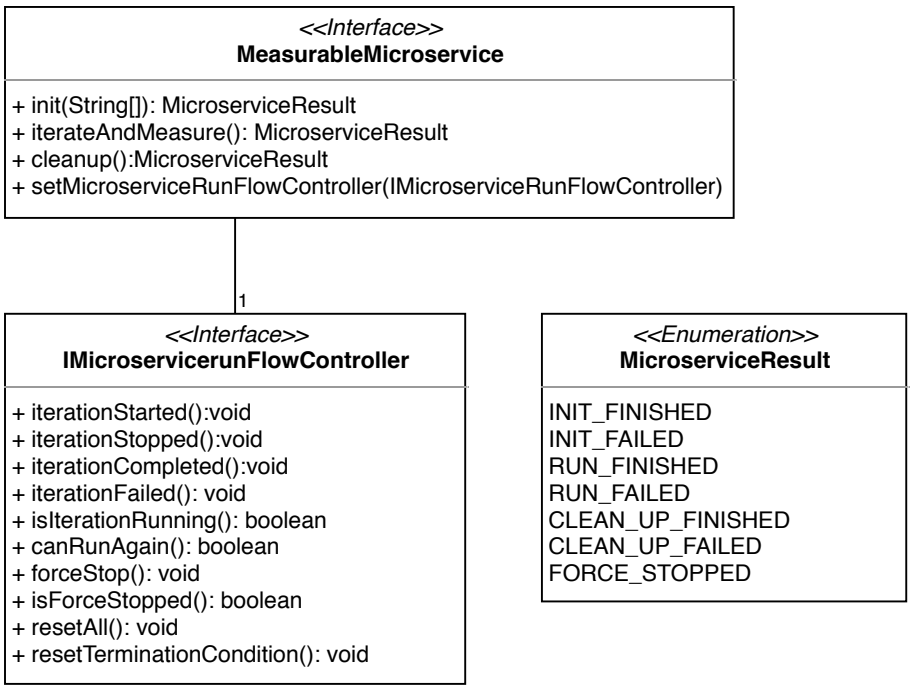


Figure 8.6: UML class diagram of the communication interface.

To report the run flow controller that the microservice execution started and the data collection should be started, call the `iterationStarted()` method. When the microservice is finished, calling the `iterationStopped()` reports the run flow controller that the microservice is finished and the data collection for that particular iteration should be stopped. To increase the counter of the iterations passed, call the `iterationCompleted()` method. If the microservice fails for any reason, the fail must be reported to the run flow controller by calling `iterationFailed()` method. Otherwise, the corresponding result should be returned.

```
1  @Override
2  public MicroserviceResult iterateAndMeasure() {
3      try {
4          while (runFlowController.canRunAgain()) {
5              runFlowController.iterationStarted();
6
7              microservice.run();
8
9              runFlowController.iterationStopped();
10             runFlowController.iterationCompleted();
11         }
12     } catch (TaskFailedException e) {
13         if (runFlowController.isIterationRunning()) {
14             runFlowController.iterationFailed();
15         }
16         return MicroserviceResult.RUN_FAILED;
17     }
18
19     if (runFlowController.isForceStopped()) {
20         return MicroserviceResult.FORCE_STOPPED;
21     } else {
22         return MicroserviceResult.RUN_FINISHED;
23     }
24 }
```

Listing 8.1: Example implementation of the `iterateAndMeasure()` method.

8.6 Microservices

In our solution, the microservices are implemented as Java applications. This fulfills the requirements, which expect real-life microservices implemented in high-level programming languages. Furthermore, using the same programming language for the measuring framework and the microservices enables to simply connect the two components.

Implementing the Communication Interface

Even though, the 17 selected microservices represent different problem domains, they are built on the same principle. As depicted in the UML class diagram in Figure 8.7, every microservice must implement the `MeasurableMicroservice` interface presented in the previous section. Thus, the measuring framework can load the microservices and manage their execution flow including the main phases. Although, it is not mandatory for the other microservice developers, every microservice implemented in our solution contains a class called `Microservice`. This is the main class that serves as an entry-point of microservices that implement the `MeasurableMicroservice` interface.

Test-points

In our project, some microservices implement multiple algorithms from the same application domain. Each of the implemented algorithms extend the microservice with additional functionality, therefore they can be considered as separate test-points of a single microservice. For instance, microservices AVL and RB both

```
public class Microservice implements MeasurableMicroservice {
    private IMicroserviceRunFlowController runFlowController;
    private AbstractTask task;

    @Override
    public MicroserviceResult init(String[] args) {
        if (args.length >= 1) {
            // the first cmd-line argument determines the test-point
            String taskType = args[0];
            switch (taskType) {
                case Task.AVL:
                    task = new AVLTask();
                    break;
                case Task.RB:
                    task = new RBTask();
                    break;
                default:
                    return MicroserviceResult.INIT_FAILED;
            }
            try {
                // using the task object to initialize
                // the test-point and its working environment
                task.prepareTask();
            } catch (TaskFailedException e) {...}
        } else {
            return MicroserviceResult.INIT_FAILED;
        }
    }

    @Override
    public MicroserviceResult iterateAndMeasure() {
        try {
            while (runFlowController.canRunAgain()) {
                runFlowController.iterationStarted();

                // using the task object to execute the test-point
                task.performTask();

                runFlowController.iterationStopped();
                runFlowController.iterationCompleted();
            }
        } catch (TaskFailedException e) {...}
        ...
    }

    @Override
    public MicroserviceResult cleanup() {
        try {
            // using the task object to remove
            //the test-point working environment
            task.cleanupTask();
        } catch (Exception e) {...}
        ...
    }

    // setting the instance of the IMicroserviceRunFlowController ...
```

Listing 8.2: Example usage of microservice test-points.

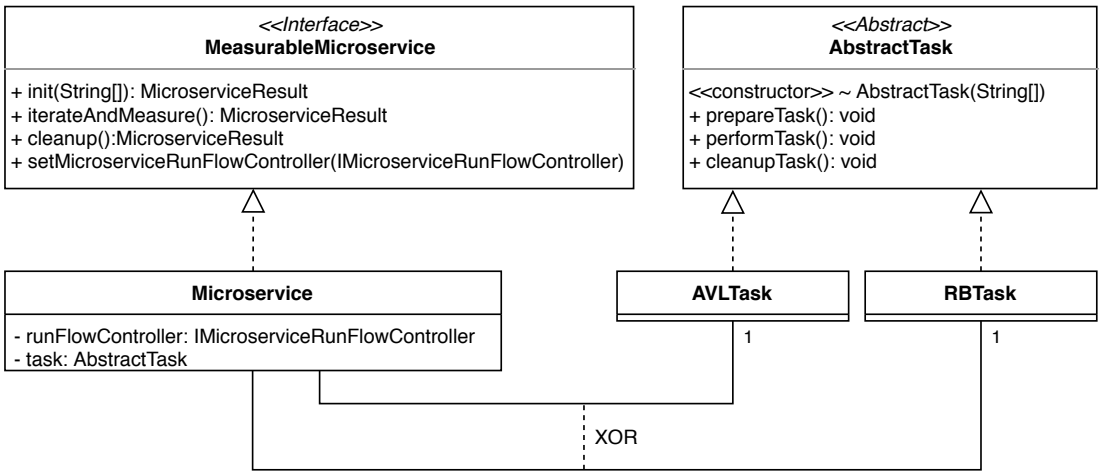


Figure 8.7: UML class diagram of the microservice.

apply the same steps in their algorithms: first they insert nodes into a tree, then they remove the inserted nodes. The different algorithms are implemented within one microservice, however, they are separated into two test-points.

The UML class diagram in Figure 8.7 shows that the test-points extend the `AbstractTask` abstract class. The `AbstractTask` defines three abstract methods that correspond to the three main phases of the microservice.

Listing 8.2 shows an example implementation of the main entry-point, the `Microservice` class. From the implementation is observable, how the appropriate instance of `AbstractTask` is created and how it is used in each phase of the microservice execution. The concrete implementation of the `AbstractTask` is instantiated in the `init(String[])` method. When the measuring framework loads the microservice, the command-line arguments are passed to the `init(String[])` method as an array of String. Based on the command-line arguments, the method determines the appropriate test-point and instantiates it (lines 12 and 15). Then, this instance is used in every microservice phase to propagate the execution (of the particular phase) to the corresponding method of the test-point (lines 23, 37 and 49).

8.7 Predictor

The goal of the predictor is to predict the submitted microservice execution time in various workload combinations and use the prediction result to evaluate the requirements specification.

However, the predictor is considered as a standalone application within the whole solution, it is tightly connected with the orchestrator. The orchestrator starts the prediction process and provides the input parameters after the microservice measurement finishes. Therefore, the predictor should be executed as a sub-module of the orchestrator and should be located on the orchestrator project directory.

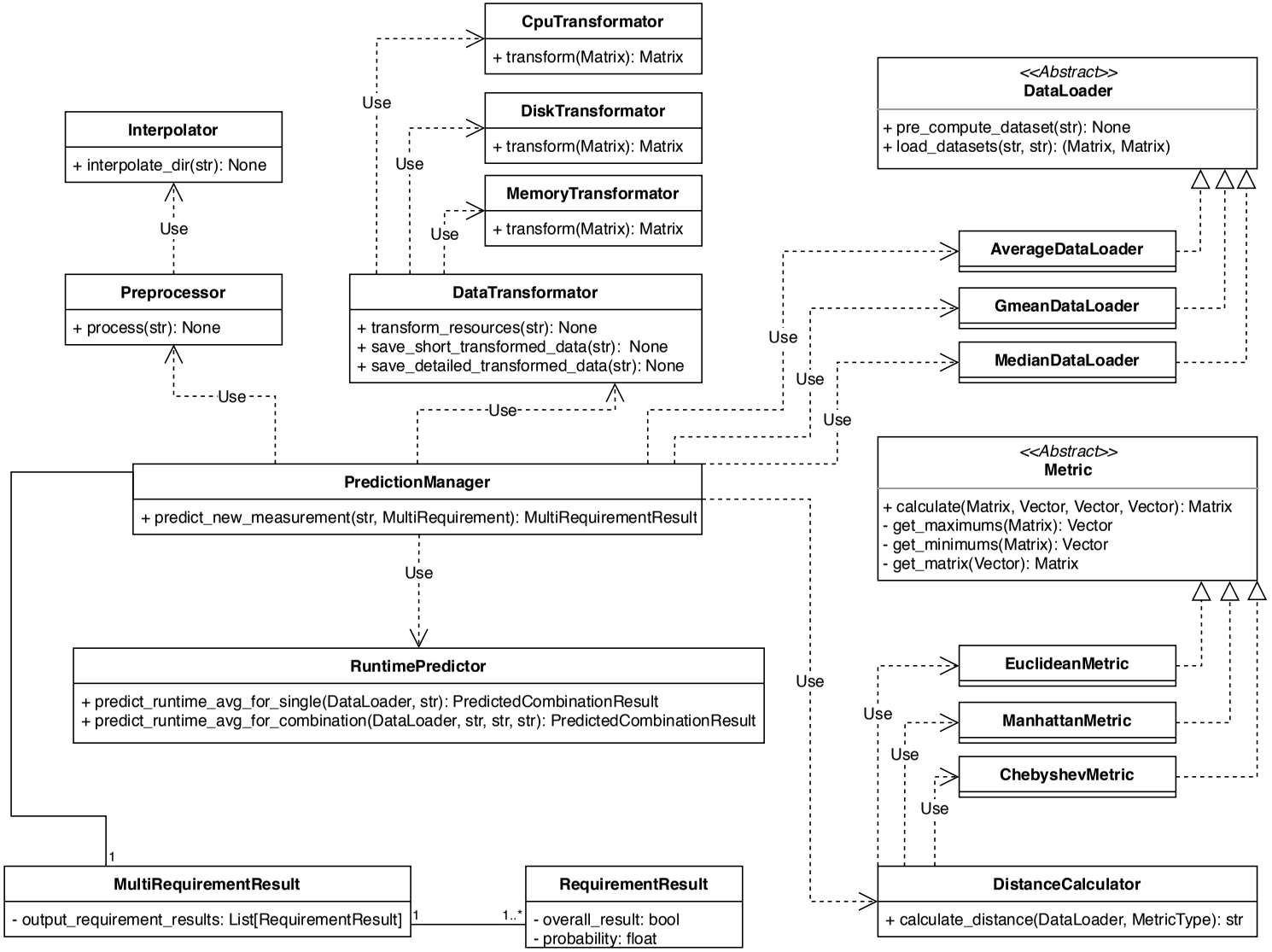


Figure 8.8: UML class diagram of the predictor.

8.7.1 Prediction Process

The prediction process is a sequence of predefined steps. The execution of these steps are controlled by the `PredictionManager` central class that ensures the proper execution of sub-steps and handles the occurring events (step succeeded, failed, etc.). The prediction process is started by the orchestrator that calls the `PredictionManager`. As a part of the call, the orchestrator passes the path to the collected data, and an instance of the `MultiRequirement` class including the microservice execution time and probability requirements.

The prediction process is separated into two big phases: data preparation and prediction.

8.7.2 Data Preparation

The data preparation phase is further divided into data preprocessing and microservice characterization steps.

Data Preprocessing

Data preprocessing is the first main step of the prediction process. It includes three data preprocessing sub-steps shown in Figure 7.1 and Figure 7.2. Their execution is managed by the `Preprocessor` class that contains the implementation of every sub-step, except the data interpolation. The interpolation of sampling results from 2-second-interval to 1-second-interval is implemented by the `Interpolator` class.

Characterization

The data preprocessing includes the microservice characterization, which is described in more detail in Section 7.3. Because the data preprocessing sub-steps are designed to prepare the data for the prediction, additional data transformations steps are required. This transformation process is managed by the `DataTransformator` class. The `DataTransformator` parses the input files and passes the parsed data to the particular transformator. For each monitored hardware resource there exists a particularly implemented data transformator: `CpuTransformator`, `DiskTransformator`, and `MemoryTransformator`. Each data transformator takes the parsed input and transforms into a unique representation that can be later used for characterizing the microservice behavior.

8.7.3 Prediction

The data collection is followed by the second phase of the prediction process, the prediction of microservice execution time. We show the architecture of this phase by following the steps presented in Section 7.4.3 and describe how the `PredictionManager` manages the other components that are involved in the prediction process.

Microservice Representation

Transforming the microservice into a common representation that enables the prediction process to compare it with other microservice is done by the data loaders. The abstract `DataLoader` class implements the basic features of loading the microservice representation, and it is further extended by three concrete data loaders: `AverageDataLoader`, `GmeanDataLoader`, and `MedianDataLoader`. The concrete data loaders load the following microservice representation: average, geometric mean, and median, respectively. The concrete type of data loader used by the `PredictionManager` is specified in the configuration file, and the particular `DataLoader` instance is created by the `DataLoaderFactory`.

Distance Calculation

The second step focuses on finding a microservice from the training dataset that is the most similar to the submitted microservice. This process is managed by the `DistanceCalculator`, which uses a metric to calculate the similarity of two microservices. The abstract `Metric` class implements the basic features of metric calculation, and it is further extended by three concrete metrics: `ChebyshevMetric`, `EuclideanMetric`, and `ManhattanMetric`. The concrete instance used by `DistanceCalculator` is specified in the configuration file, and the particular `Metric` instance is created by the `MetricFactory`.

Prediction

The rest of the prediction process is implemented in the `PredictionManager` and the `RuntimePredictor` classes. The `PredictionManager` generates the double, triple, and quadruple combinations for which the microservice execution time will be predicted. Then, the `RuntimePredictor` computes the execution time ratio of the submitted and its most similar microservice and uses this ratio to predict the microservice execution time in that particular combination. Finally, the `PredictionManager` generates the result of prediction as an instance of the `MultiRequirementResult` class.

8.8 System Setup

This section lists the steps required for setting up the Deployment Framework. The scripts, project resource codes, and other assets referred in this section are located in the `system_setup` directory in the attached media.

8.8.1 General System Requirements

We split the installation of the Deployment Framework into two server computers. We install the Orchestrator and the Prediction application on the first server, and install the the Measuring Agent, including the Measuring Framework, on the second server computer.

We require our server computers to run a clean installation of the `Fedora Sever 28` operating system, and a `Python 3.6.5` interpreter. The additional packages will be installed by the attached installation scripts. Furthermore, we require a 64-bit quad-core CPU, where performance-related features are disabled, 30 GB free space on a dedicated HDD, and 16 GB of physical memory.

8.8.2 Installing Orchestrator and Predictor Applications

Execute the following steps to install the Orchestrator and the Predictor applications. The target installation directory is `/home/df-orchestrator/`.

1. Copy the `/system_setup/orch_pred_install` directory to the home directory of the current user.
2. Open the `orchestrator_and_predictor_src` directory located in the `orch_pred_install` and execute the `install_orchestrator_and_predictor.sh` script.
3. Open the `training_dataset` directory located in the `orch_pred_install` and execute the `copy_training_dataset.sh` script.
4. If the firewall is active in the system, enable ports `60000` and `60011`.

Starting Orchestrator and Predictor Applications

1. Open `/home/df-orchestrator/sources/orchestrator_and_predictor_run/` and activate the virtual environment `venv-orch-pred`.
2. Open `./orchestrator_run` and execute the command shown in Listing 8.3 to start the Orchestrator application.
3. The Orchestrator is running and waiting for the Measuring Agents to connect. Use the `lsb` and `lsa` commands of the administrator UI to list the measurement tasks and the connected agents.

```
1 PYTHONPATH=. python orchestrator/app.py -i y -r ../requirements/rb.yaml
```

Listing 8.3: Starting Orchestrator application.

8.8.3 Measuring Agent

The installation of the Measuring Agent application on the second server computer is divided into two phases. The first phase prepares the system by installing Docker and other required packages. The second phase installs the Measuring Agent application.

Installing Docker

1. Copy the `/system_setup/agent_install` directory to the home directory of the current user.
2. Open the `agent_system_install` directory located in the copied `agent_install` directory and execute the `install_system.sh` script.
3. Accordingly to the Docker Documentation[3], enable to manage Docker as a non-root user.
4. Execute the commands shown in Listing 8.4 to start the Docker service now and to enable to start it automatically when the system boots.

```
1 sudo systemctl start docker
2 sudo systemctl enable docker
```

Listing 8.4: Start Orchestrator.

Installing Measuring Agent Application

Execute the following steps to install the Measuring Agent, and to create a service to start the Measuring Agent after the system booted. The target installation directory is `/home/df-agent/source/measuring_agent_run/`.

1. Open the `agent_client_install` directory located in the copied `agent_install` directory and execute the `install_agent.sh` script.
2. In the target installation directory,
   open the `./constants/config_file.ini` file for editing. Edit the value of the `[agent][OrchestratorAddress]` field so that it contains the address of the Flask server running in the Orchestrator.
3. If the firewall is active in the system, enable port `50000`.
4. Open again the `agent_client_install` directory and execute the `install_agent_service.sh` script.

Starting the Measuring Agent Application

Start the installed `measuring_agent.service` to start the Measuring Agent application. Since, the server computer is restarted multiple times during the measurement process, one must enable to start the microservice automatically after the system re-boots.

[3] `https://docs.docker.com/engine/install/linux-postinstall/`

9. Related work

To be able to position our proposed solution in the context of this research area, we first describe other solutions from the same field of interest. Then, we compare their particular points that define the frame of reference for our research.

Q-Clouds

Q-Clouds [40] is a QoS-aware control framework that tunes resource allocations to mitigate performance interference effects. Q-Clouds analyzes the resource-usage requirements of VMs submitted by the clients by building a multiple-input-multiple-output model that captures performance interference interactions. Then, Q-Clouds manages the interference among VMs by dynamically adapting resource allocation using closed loop resource management.

Cuanta

Cuanta [41] uses active probing to precisely quantify the impact of shared cache interference and memory bandwidth contention for colocated applications. It uses a linear number of measurements to estimate the performance degradation in each application due to contention of shared chip-level resources. The key feature of Cuanta is that it uses a synthetic cache clone of each application to mimic its cache pressure. The cache clones are later used as a proxy for actual applications to predict the interference due to various colocation scenarios.

CloudScope

CloudScope [42] is a model-based approach for diagnosing interference for multi-tenant cloud systems. It employs a discrete-time Markov chain model for the on-line prediction of performance interference of colocated VMs. CloudScope characterizes the performance degradation by probing the system behavior, which includes both the behavior of the hypervisor layer and the hardware characteristics of different workloads. The model is then used to optimally reassign VMs to physical machines and to adjust the hypervisor configuration.

CtrlCloud

CtrlCloud [43] is a feedback-driven resource controlling system that adaptively allocates CPU resources for cloud applications to meet QoS targets. It uses dynamic controllers with a fine-grained resource-share slicing scheme to optimize the resource allocation based on application performance. CtrlCloud maintains an online model to quantify the dynamic non-linear relationship between resource shares allocated to an application and its corresponding performance.

Pythia

Pythia [44] is a colocation manager that predicts the combined contention of multiple batch workloads with a latency sensitive workload on shared resources.

For non-recurring workloads, Pythia executes a sequence of profiling and modeling operations before the workload can be scheduled. First, Pythia performs contention characterization for each batch workload when singly running with a particular latency sensitive workload, and removes the batch workloads that create contention higher than the determined contention score. Then, it uses a small fraction of batch workloads to colocate with a latency sensitive workload, to build the linear regression prediction model for the contention due to multiple batch workloads.

Overview

The common aim of our and the presented approaches is to provide a self-adaptive, performance-aware, and interference-aware system that manages the resource allocation and the deployment of containers in cloud environment. In addition, the system optimizes the overall resource utilization of the cloud while enabling the microservices to meet the required quality of service.

Similarly to Pythia, our approach adds an additional step into the admission process, where it profiles the microservice. However, our approach relies on profiling the microservice via its test points, that enable to determine the microservice resource demands and changes of response time in various workload combinations. We require the developers to implement the test-points so that they mimic the resource demand of the real microservice and to explicitly specify performance objective for them. If the specified objectives are satisfiable by the system, then we admit the microservice. Using dedicated test-points to provide guarantees, we create a suitable contract between the microservice developer and the cloud. It also enables us to treat the microservices as black-boxes.

Mostly presented approaches focus only on the CPU usage when characterizing the microservice and planning the microservice deployment. Contrary to these approaches, our approach uses the CPU, disk, and memory resources to characterize the microservice resource demand and to build a model for prediction.

We target a private edge-cloud environment e.g., mobile network operators, which enables us to apply constrains that would not be possible to implement in public cloud service environments. Unlike other approaches, our approach is based on using containers instead of virtual machines. This allows us fast deployment of the microservices and also fast re-deployment of the containers if the underlying infrastructure lacks of resources. Furthermore, this private cloud enables us to use context-specific details, like providing statistical performance guarantees based on estimation of worst-case instead of other approaches that try to estimate the throughput.

10. Conclusion

In this thesis, we developed a new approach that provides soft real-time guarantees on the response time of microservices running in a container-based cloud. In this context, we implemented a prototype of a deployment framework, which measures the resource usage of the submitted microservice and uses this data to predict the microservice response time in various workload combinations.

In the first part of the thesis, we focused on the requirements analysis and conducted research in two problem domains. The first research area focused on the technical analysis and an overview of the available cloud technologies that fulfill the requirements of the presented motivation application. Afterwards, the research was dedicated to the system performance indicators. It was followed by the analysis of the available resource usage monitoring tools that enable to record the values of the analyzed performance indicators.

In the second part of the thesis, we applied knowledge from the previous research. We designed and implemented a deployment framework, where the developer can submit a microservice and specify its requirements. The deployment framework converts the requirements specification into microservice measurement tasks and distributes them among measuring agents. These agents execute the tasks and assess the microservice resource usage in isolation and in predefined workload combinations. Then, the system uses the collected data to predict the microservice response time in a shared production system.

Finally, we evaluated the prediction process. For this purpose, we implemented artificial microservices that enabled us to simulate various real-life scenarios and therefore to test the measuring framework along with the predictor.

The implemented deployment framework has gone through excessive testing using a cluster of seven physical servers. We have executed more than 10,000 measurements using final version of the deployment framework. During the measurements, the system was stable and free of errors and failures. Furthermore, the system was able to handle the user-related issues e.g., incorrect requirements specification or failing microservices without external intervention and continued the measurements process up to 21 days. This supports the stability of the implementation.

In conclusion, we have successfully implemented and tested a working prototype of the deployment framework, and therefore the goals of this thesis have been completed.

10.1 Future Work

The proposed solution possesses the ability to be extended in the future and the architecture has been designed to support scalability and modifiability according to the future needs.

The data collection process could be improved by implementing additional data collectors to measure the disk and memory usage of a particular process. Thus, the measured data would represent the resource usage of the main microservice only. Also, the prediction accuracy could be increased. It could be either done by extending the training dataset with more microservices, which

would cover a wider application domain, or by implementing a more sophisticated prediction algorithm. For instance, assigning weights to resources based on their impact on the microservice execution time, or utilization of machine learning algorithms.

Bibliography

[1] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010.

[2] Wikipedia contributors. Service-level agreement — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Service-level_agreement&oldid=871014823`, 2018. [Online; accessed 3-February-2019].

[3] VMware, Inc. VMware ESXi. `https://www.vmware.com/products/esxi-and-esx.html`, 2019. [Online; accessed 3-February-2019].

[4] Citrix Systems, Inc. XenServer. `https://xenserver.org/`, 2019. [Online; accessed 3-February-2019].

[5] Todd Fredrich. Learn REST: A RESTful Tutorial. `https://www.restapitutorial.com/`, 2017. [Online; accessed 3-February-2019].

[6] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 1–14, New York, NY, USA, 2010. ACM.

[7] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009.

[8] K. Ha, P. Pillai, G. Lewis, S. Simanta, S. Clinch, N. Davies, and M. Satyanarayanan. The impact of mobile multimedia applications on data center consolidation. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*, pages 166–176, March 2013.

[9] C. Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, May 2015.

[10] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, March 2015.

[11] Linux Containers. LXC. `https://linuxcontainers.org/`, 2008. [Online; accessed 6-February-2019].

[12] Docker Inc. Docker. `https://www.docker.com/`, 2013. [Online; accessed 6-February-2019].

[13] The Kubernetes Authors. Kubernetes. `https://kubernetes.io/`, 2003. [Online; accessed 11-February-2019].

[14] Technische Universität Darmstadt. Scala Benchmark Suite. https://kubernetes.io/, 2010. [Online; accessed 03-March-2019].

[15] stress ng. A tool to stress test a computer system in various selectable ways. https://kernel.ubuntu.com/~cking/stress-ng/, 2013. [Online; accessed 03-March-2019].

[16] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Prentice Hall, 2013.

[17] vmstat. Linux man page. https://linux.die.net/man/8/vmstat, 2013. [Online; accessed 15-March-2019].

[18] iostat. Debian Manpages. https://manpages.debian.org/testing/sysstat/iostat.1.en.html, 2018. [Online; accessed 26-March-2019].

[19] Seagate Technology LLC. Constellation.2™ Data Sheet. https://www.seagate.com/files/www-content/product-content/constellation-fam/constellation/constellation-2/en-us/docs/constellation2-fips-ds1719-4-1207us.pdf, 2012. [Online; accessed 20-June-2019].

[20] The kernel development community. The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/, 2019. [Online; accessed 20-June-2019].

[21] Michael Kerrisk. Linux man pages online: proc - process information pseudo-filesystem. http://man7.org/linux/man-pages/man5/proc.5.html, 2019. [Online; accessed 20-June-2019].

[22] Linux Kernel Documentation. The /proc filesystem. https://www.kernel.org/doc/Documentation/filesystems/proc.txt, 2009. [Online; accessed 24-June-2019].

[23] Linux Kernel Documentation. I/O statistics fields. https://www.kernel.org/doc/Documentation/iostats.txt, 2008. [Online; accessed 24-June-2019].

[24] Michael Kerrisk. Linux man pages online: sysfs - a filesystem for exporting kernel objects. http://man7.org/linux/man-pages/man5/sysfs.5.html, 2019. [Online; accessed 20-June-2019].

[25] Linux Kernel Documentation. Block layer statistics in /sys/block/<dev>/stat. https://www.kernel.org/doc/Documentation/block/stat.txt, 2011. [Online; accessed 26-June-2019].

[26] Linux Kernel Documentation. Control groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, 2014. [Online; accessed 30-June-2019].

[27] Michael Kerrisk. Linux man pages online: cgroups - Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html, 2019. [Online; accessed 30-June-2019].

[28] Linux Kernel Documentation. Cpusets. https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt, 2004. [Online; accessed 30-June-2019].

[29] Linux Kernel Documentation. Memory resource controller. https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt, 2008. [Online; accessed 31-June-2019].

[30] cAdvisor GitHub Issue. No stats for blkio. https://github.com/google/cadvisor/issues/1946?fbclid=IwAR06DkzO8C1eOZxy9ECzIxkISj6WcE7N-CI5iUClMyqktB57Kw7-pqTlB3gs, 2018. [Online; accessed 30-June-2019].

[31] Linux Kernel Organization, Inc. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2015. [Online; accessed 26-March-2020].

[32] Department of Distributed and Dependable Systems. Microbenchmarking agent for java. https://github.com/D-iii-S/java-ubench-agent, 2014. [Online; accessed 20-June-2019].

[33] Michael Kerrisk. Linux man pages online: PAPI - Performance Application Programming Interface. https://linux.die.net/man/3/papi, 2019. [Online; accessed 20-June-2019].

[34] pkgs.org Packages Search. Libraries for papi clients. https://pkgs.org/download/papi-libs, 2019. [Online; accessed 20-June-2019].

[35] Vojtěch Horký, Peter Libič, Antonin Steinhauser, and Petr Tůma. Dos and don'ts of conducting performance measurements in java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 337–340, New York, NY, USA, 2015. ACM.

[36] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Benchmark precision and random initial state. In *in Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS 2005*, pages 853–862. SCS, 2005.

[37] DaCapo Project. DaCapo Benchmark. http://dacapobench.sourceforge.net/, 2009. [Online; accessed 23-March-2020].

[38] Docker Docs. Runtime options with Memory, CPUs, and GPUs. https://docs.docker.com/config/containers/resource_constraints/#cpu, 2020. [Online; accessed 23-March-2020].

[39] Apparat. A framework to optmize ABC, SWC and SWF files. https://github.com/joa/apparat, 2013. [Online; accessed 23-March-2020].

[40] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 237–250, New York, NY, USA, 2010. Association for Computing Machinery.

[41] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[42] Xi Chen, Lukas Rupprecht, Rasha Osman, Peter Pietzuch, Felipe Franciosi, and William Knottenbelt. Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds. In *Proceedings of the 2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '15, page 164–173, USA, 2015. IEEE Computer Society.

[43] Omer Adam, Young Choon Lee, and Albert Y. Zomaya. Ctrlcloud: Performance-aware adaptive control for shared resources in clouds. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, page 110–119. IEEE Press, 2017.

[44] Ran Xu, Subrata Mitra, Jason Rahman, Peter Bai, Bowen Zhou, Greg Bronevetsky, and Saurabh Bagchi. Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 146–160, New York, NY, USA, 2018. Association for Computing Machinery.

List of Figures

List of Tables

Listings

A. Attachments

A.1 Batch configuration

The following table describes each attribute of the batch configuration.

Attribute	Description
`configs`	Indicates the beginning of the batch configuration file.
`batch`	Name of the batch definition. Generally, the name is prefixed with `batch--`. Then it is followed by the IDs of the microservices, and enclosed with the IDs of constraints.
`init_task`	Indicates the beginning of the initial task definition. If there is no initial task definition, `null` should be assigned as a value.
`id`	The unique ID of task definitions or constraints. The task definition IDs are conventionally written by upper cases, and are referred in the command definitions.
`def_type`	Type of the task definition. Defining the definition type helps the Agent to choose the appropriate parser that parses the task definition.
`dockerfile_path`	Location of the Dockerfile in the file-system. The resulting Docker image is going to be used by the microservice.
`image_tag`	Name of the Docker image that is going to be used by the microservice.
`microservice_tasks`	Indicates the beginning of the microservice task definitions. This task definition is mandatory.
`microservice_name`	Name of the microservice.
`microservice_class_name`	Class path of the microservice test-point.
`microservice_class_path`	Path to the executable file containing the microservice and its test-points.
`microservice_params`	Command-line parameters required to execute the microservice.
`collect_data`	Set to `true` to enable the data collection at the particular microservice, otherwise set to `false`. The data collection should be enabled only for the main workload.
`main`	Set to `true` to mark the microservice as the main microservice, otherwise set to `false`. There can be only one main microservice in each batch definition.
`run_duration`	Determines the duration of the microservice execution.
`container`	The name of the container where the microservice will be executed.
`image`	The name of the Docker image (created in the initial task) that is going to be used by the Docker container.

`docker`	Indicates the beginning of the Docker-related constraints.
`options`	Beginning of the constraint list.
`option`	A particular constraint for the Docker container.
`java`	Indicates the beginning of the JVM-related constraints.
`final_task`	Indicates the beginning of the final task definition. If there is no final task definition, `null` should be assigned as a value.
`commands`	Indicates the beginning of the command definitions.
`init_command`	Defines the initial command by assigning the ID of the initial task definition.
`microservice_commands`	Defines the microservice commands by assigning the IDs (separated by space) of the microservice task definitions.
`final_command`	Defines the final command by assigning the ID of the final task definition.
`runs`	Defines the amount of iterations the batch and the included tasks will be repeatedly executed.

Table A.1: Description of the attributes in batch configuration.

A.2 The Attached Medium

The medium attached to this thesis contains the following directories:

- **`text`**: Contains the `PDF` version of this text.
- **`sources`**: Contains the source code and the generated documentation for each project. The Python projects include `setup.py` to install them in a separate virtual environment. The Java projects include `build.xml`, therefore the `ant jar` command can be used to build the `JAR` file.
- **`system_setup`**: Contains the prepared resources and installation scripts to help to set up the Deployment Framework.
- **`prediction_resources`**: Contains resources related to the prediction process.