Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Hardware Dimensioning for Microservice-based Cyber-Physical Systems: A Profiling and Performance Prediction Method

Author: Marijn Vollaard (2726662/13573187)

1st supervisor:	Benny Akesson	
daily supervisor:	Ben Pronk	(TNO-ESI)
2nd reader:	Ana Oprescu	

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

December 18, 2023

Abstract

With the rise of large-scale web applications, microservices emerged as a viable alternative to monolithic systems. Offering scalability, agile development, and customizable deployment, microservices were adopted not only in a web environment but also in cyber-physical systems (CPS). However, this adoption introduced a new challenge: ensuring adequate hardware allocation and microservice deployment for performance requirement satisfaction in CPSs. When little is known about application performance upfront, this may involve constructing a prototype with all envisioned compute nodes and trying out all relevant deployment configurations, which becomes a very costly and time-consuming endeavor. To mitigate these challenges, a need arose for a means to predict the system performance for varying deployment configurations before construction. Hence, the development of a structured methodology for hardware dimensioning, which involves determining the required number of compute nodes to meet performance requirements supported by performance prediction, becomes imperative.

The goal of this thesis is, therefore, to provide a structured hardware dimensioning methodology comprising a profiling method and a performance prediction method. We do so by introducing four novel contributions: 1) A component-based profiling method, 2) a performance prediction method, 3) a structured hardware dimensioning methodology, and 4) validation of the approach, using a case study that represents a prototype of a CPS. Through this approach, we have come to two prediction models, for which the predictions differ by at most 20%. Altogether, the proposed methodology provides a solid basis for hardware dimensioning of microservice-based CPSs.

Contents

List of Figures

List of Tables

1

Introduction

In recent years, there has been a growing demand for scalability in the context of large-scale web services. Traditional monolithic applications running across multiple servers have proven inadequate in handling the escalating workload resulting from an expanding user base. In response to this challenge, the concept of microservice architectures has been introduced [1, 2]. Microservices offer a solution by affording scalability, facilitating agile development, and enabling the customization of service deployment within an application [3].

Microservices have found widespread adoption not only within the domain of large-scale web services, but also in applications for cyber-physical systems (CPS) [4]. A CPS is characterized by the integration of both computational and physical components [5, 6]. In the case of a CPS, scalability remains a valuable feature, but the driving factor behind the adoption of a microservice architecture is the increased possibility of deployment customization and agile development. A CPS is often an instance of a product, undergoing iterative development cycles in response to varying user and stakeholder demands, which may entail varying features and capabilities, thus impacting performance and latency requirements. Consequently, instances of a product may vary heavily. A microservice architecture proves advantageous in this context, as it is highly customizable and therefore allows for the utilization and deployment of microservices tailored to the specific needs of the customer [7].

The variability among product instances presents challenges in ensuring performance guarantees. If there is little known upfront about the application performance, a prototype needs to be built. Building such a prototype of a CPS is often costly and takes time. To avoid this, there is a need for a means to predict system performance across different

configurations before construction. This becomes particularly critical because system performance depends on the number of compute nodes and their specification. To ensure that performance requirements are met, an adequate amount of resources in terms of compute nodes and network must be allocated. This process of hardware allocation to an application is referred to as "hardware dimensioning".

In cloud computing environments, the problem of hardware dimensioning is typically addressed by deploying applications randomly and adapting them to their specific requirements while they are operational [8, 9]. However, this approach is not feasible for CPSs, as these systems are typically presented as fully integrated products, complete with the hardware on which they run. Adding hardware once the system is operational is often unfeasible due to constraints like limited energy supply or physical space. In such cases, guaranteeing performance in advance necessitates the ability to predict how an application will perform with a given hardware configuration.

1.1 Problem Description

Since different instances of a product may exhibit distinct functionalities, the allocation of hardware resources to each instance may also differ. In cases where a product has undergone numerous iterations, experienced system experts may allocate hardware based on past knowledge. Proper hardware dimensioning then relies on their availability. Additionally, when dealing with new products or instances that possess markedly different functionalities, this experience does not exist yet. A structured approach is missing in this context, as further supported by Chapter 3. Furthermore, a complete hardware system may not yet be available in the dimensioning stage, except for certain testing infrastructure. Thus, there is a need for performance prediction based on limited available hardware.

The inherent flexibility of a microservice system presents an opportunity for optimization through strategic deployment. However, it also gives rise to questions regarding the selection of services to run on specific hardware and the optimal configuration of these services. Combining or distributing microservices across various nodes may have a significant impact on an application's performance. To fully harness this flexibility for hardware dimensioning, a comprehensive understanding of an application's performance and the implications of microservice deployment and placement on hardware is necessary.

Thus, there is a need for a structured hardware dimensioning approach that involves the prediction of the performance of an application given certain hardware and a configuration of microservices over nodes in the context of CPSs that employ a microservice architecture.

While hardware dimensioning may include both network dimensioning as well as compute node dimensioning in a CPS, this thesis only considers compute node dimensioning and assumes there is always sufficient network hardware available.

1.1.1 Proposed Approach

In this thesis, we propose a hardware dimensioning approach that aims to solve the problem described in the previous section. It includes three steps as shown in Figure 1.1: 1) Application Profiling, 2) Performance Prediction, and 3) Hardware Dimensioning.

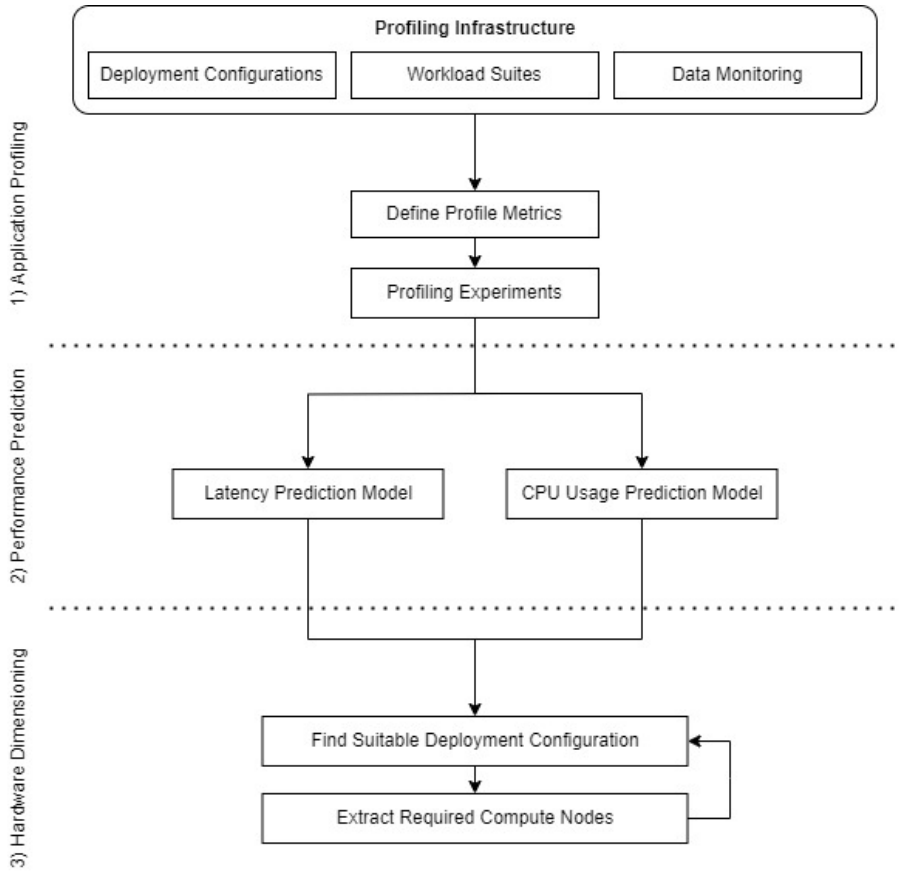


Figure 1.1: Overview of our proposed hardware dimensioning approach.

The application profiling step consists of defining the profiling infrastructure needed to perform the profiling of an application. The profiling infrastructure comprises 1) defining profiling deployment configurations, 2) designing a workload suite, and 3) creating data monitoring and collection infrastructure. Once the profiling infrastructure is defined, the profile metrics need to be determined, after which the profiling experiments are run to

gather those metrics. In this thesis, we aim to come to a profiling approach that applies to any general microservice application, is time-efficient, can be performed on a limited amount of hardware, and forms the basis for performance prediction.

The performance prediction in our approach consists of a latency prediction model and a CPU usage prediction model, that leverage the application profile and are given a deployment configuration to predict either the performance in terms of system workflow latencies or the CPU usage of an application. In this thesis, we will look for a performance prediction approach that balances the accuracy of the model with its simplicity. We consider the accuracy to be the difference between our predictions and our validations of the model. With the simplicity of our model, we mean that the model includes as few performance aspects as possible while it is still complete in its prediction.

Then the performance prediction is used in the last step, hardware dimensioning, which consists of finding a suitable deployment configuration of microservices for a number of nodes, based on the performance requirements and performance prediction models. We repeat the process of finding a suitable deployment configuration with a varying number of nodes, so that we come to the minimum number of nodes for which we can find such a deployment configuration. We can then use that minimum number of compute nodes as the answer to our hardware dimensioning problem.

1.1.2 Research Questions

Based on the problem description and proposed approach, the following research questions were formulated:

RQ1: What is a suitable profiling method and framework for a microservice-based CPS for the purpose of hardware dimensioning?

RQ2: What is a suitable performance prediction method that predicts whether the performance requirements of a microservice-based CPS are met, with a given deployment configuration of microservices over a set of homogeneous nodes and an application profile?

RQ3: How can we use the profiling and performance prediction methods to come to a structured approach to hardware dimensioning in the context of a microservice-based CPS?

1.2 Contributions

In this thesis, we look at how to perform hardware dimensioning in the context of microservice-based applications in CPSs. The goal is to provide a structured approach to hardware dimensioning that can be used by practitioners as a tool in assigning compute nodes to a microservice application in a CPS. We provide the following contributions that are a part of the hardware dimensioning approach:

1. Automated profiling framework and a profiling method for a microservice application in a CPS that can be used as the basis for performance prediction models.
2. A performance prediction method, shown with two prediction models, one that predicts latencies of system workflows and one for the CPU usage of a node. They consider the profile of a microservice application in a CPS and its deployment configuration.
3. A demonstration of how to leverage the profiling and performance prediction methods to come to a structured approach to hardware dimensioning in the context of a microservice-based CPS.
4. A case study that is used 1) to validate both the profiling and performance prediction methods described in this thesis and 2) as an example of how to perform hardware dimensioning.

1.3 Outline

The outline of the rest of this thesis is as follows: Chapter 2 explains the background necessary to understand the rest of the thesis. Chapter 3 discusses related work, to put the thesis into its context. Chapter 4 introduces the Meal Delivery Service (MDS), which is the case study application by which the methods proposed in this thesis are validated. Chapter 5 and Chapter 6 describe our application profiling method and performance prediction method respectively. Chapter 7 goes into detail on how those methods are used in our structured hardware dimensioning approach. Chapter 8 discusses the method and results of our experiments, and Chapter 9 concludes this thesis, combined with some remarks on future work.

2

Background

This chapter aims to give an understanding of the fundamental concepts of the material discussed in this thesis. Section 2.1 describes cyber-physical systems and explains their characteristics, Section 2.2 defines what microservices are and why they are increasingly used, Section 2.3 explains how observability can be used in microservice applications, and Section 2.4 discusses the concept of hardware dimensioning.

2.1 Cyber-Physical Systems (CPS)

Cyber-physical systems (CPS) are systems comprising both computational and physical components. For example, a CPS may be a smart thermostat adjusting room temperature based on occupancy, or an autonomous drone that uses sensors to navigate and avoid obstacles in real-time. These systems commonly use feedback loops, wherein physical processes influence computational operations and vice versa [10]. CPSs are generally held to a higher standard of reliability and predictability than general-purpose applications [6]. Many industrial CPSs incorporate mission-critical functionalities that must operate correctly while meeting specific performance requirements.

Consequently, CPSs frequently face rigid resource utilization and latency requirements. At the same time, the inherent unpredictability of the real world can lead to unexpected behavior in the physical components of a CPS. For instance, consider an autonomous drone. The drone relies on a network to communicate with a central control system for receiving commands. If there is a delay or packet loss in the communication network, the drone's control system may receive outdated information or experience delays in executing commands, which could cause the drone to fly into a moving object, for example. The unpredictability of outside components, like the moving object, require the system to

be more reliable, while dealing with unpredictable outside factors. The combination of elevated reliability standards and unpredictable physical behavior makes CPS performance prediction more critical than general-application performance prediction.

Often, multiple instances with different configurations of a CPS are developed. To address this variability, microservices are increasingly adopted within these systems. Their versatility in development, deployment, and hardware placement offers a practical solution to the intricate demands of a CPS [4]

2.2 Microservices and Microservice Deployment

A microservice is a small, independent, loosely coupled service that has a specific functionality within a system [2, 11]. Microservices in an application work together, engaging in communication and data exchange to collectively form an application or microservices architecture. A notable attribute of microservices is their autonomy, enabling independent development, deployment, and scaling without the need to fully redeploy or rebuild the whole system of services. Additionally, microservices within an application are generally not fixed in terms of their deployment over nodes in a system, so they can be deployed on any suitable node, also at run-time.

Both the independent nature and the broad deployment potential of microservices make it an attractive software architecture to use in the context of CPSs. However, the high potential for customization in the deployment and redeployment of microservices necessitates a more thorough insight into the effect of this deployment customization on the performance of a microservice system. Due to the lack of insight into the effect of various deployment configurations on the performance of an application, we speak of a lack of predictability of performance in this context.

Performance variance between deployment configurations is mainly influenced by communication overhead and microservice interference. Generally, a trade-off in deployment can be found between 1) placing microservices on the same node, which will result in higher interference, and 2) further distribution, which may lead to a higher communication overhead [12]. The preferable microservice placement depends on the priorities in terms of performance requirements, the application, and the CPS in question.

2.3 Observability in Microservice Applications

To get insight into the performance of a microservice system, there is a need for observability [13]. Observability is a measure of how well the state of a service can be known from external outputs [14]. Three common external outputs form the three pillars of observability. They are (i) traces, (ii) logging, and (iii) metrics [15]. A trace is a record of activities or interactions that occur within a service or application that processes requests or performs tasks, logs are records of events, actions, or messages within a service or application, and metrics are measurements that quantify the behavior and performance of a service or application over time, such as CPU and memory usage. In this thesis, metrics and traces are used in our profiling and performance prediction methods and are therefore discussed in further detail.

2.3.1 Time-Series Metrics

Time-series metrics refer to quantitative measures used for analyzing and interpreting data that is collected over time [16]. This is crucial in various applications, including system performance monitoring, network analysis, and cyber security. Time-series metrics enable the assessment of how computational systems evolve over intervals, offering insights into trends, patterns, and potential anomalies. Common metrics in this context include average response times, throughput, error rates, and resource utilization.

The Gartner report [17] gives an overview of many available commercial tools that offer the collection and analysis of metrics, like Paessler PRTG [18] and OpManager [19]. Additionally, open-source tools are available, like Prometheus [20], which is currently the most prominent tool of its kind. Prometheus offers a query language through which resource utilization metrics can be retrieved.

By leveraging time-series metrics, it is possible to gain a deep understanding of the dynamics of software and hardware performance. This information is vital for optimizing system efficiency, predicting resource needs, detecting irregularities, and ensuring the robustness and reliability of computer systems.

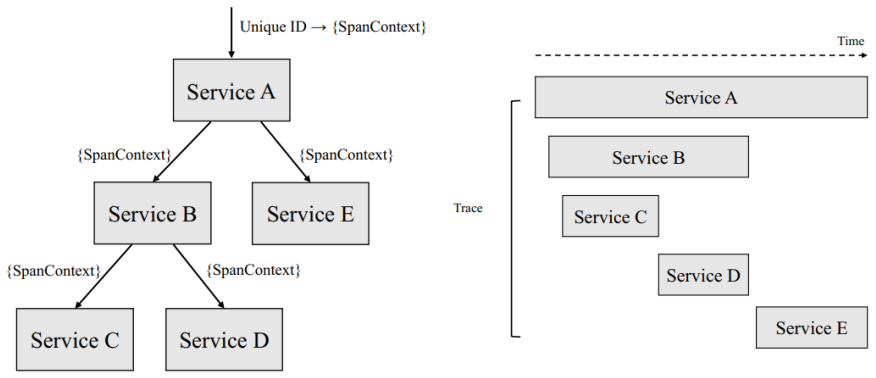
In this thesis, we will use Prometheus to retrieve time-series metrics in the context of resource utilization, specifically CPU utilization of both compute nodes and microservices.

2.3.2 Traces, Spans, and System Workflows

In microservice architecture, spans and traces are essential concepts for understanding and monitoring the flow of requests within a complex distributed system. A span represents a

unit of work, typically associated with a specific operation, function, or path through one microservice, and a set of spans collectively forms a trace, capturing the end-to-end path of a request as it traverses through various microservices. For example, in Figure 2.1a [13], a span may represent a path through service E that starts and stops at a specified event, where the trace it is a part of consists of the paths from service A through E as shown in Figure 2.1b. Spans provide detailed information about the duration, context, and events associated with a particular task, while traces offer an interconnected view of the entire request's life cycle. This is a crucial insight for diagnosing performance issues, identifying bottlenecks, and ensuring the reliability of microservice applications. Spans and traces represent an instance of a system workflow. A system workflow in this context is a specified path through an application, which a trace may take. An example of a system workflow in Figure 2.1a may be a path from service A through service B that ends in service C. A trace may be an instance of a system workflow, if it follows that exact path. By examining spans and traces, it is possible to gain insights into how requests propagate through the microservices in a system, aiding in the optimization of response times, troubleshooting of errors, and overall enhancement of the system's efficiency and resilience. Traces can be gathered by monitoring tools, such as those presented in the Gartner report [17]. Additionally, open-source tools are available, of which Jaeger [21] is currently prominently used. Jaeger can be deployed to collect and store spans and traces within an application.

In this thesis, the traces for a specified system workflow are gathered with Jaeger after which their latencies are extracted.



(a) Trace Topology

(b) Trace and Span

Figure 2.1: Example of a trace that consists of spans for a simple system workflow.

2.3.3 Tail Latency

One characteristic of traces and spans is the time, or latency, they take to complete. By gathering many traces that match one system workflow, a distribution of latencies of that workflow can be made based on the latency of each trace. Such a distribution may look as shown in Figure 2.2.

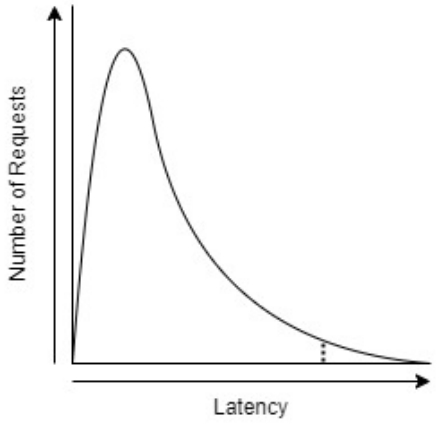


Figure 2.2: Example of 95% tail latency of a system workflow.

A yth percentile tail latency of such a distribution denotes the latency for which y percent of the latencies are lower. In the figure, the 95% tail latency is denoted by the dotted line. With the yth percentile tail latency, we claim that y percent of the traces will finish within a certain time. It can thus be used on latency guarantees of system workflows.

Tail latency aims to represent a worst-case measure while still focusing on relatively common cases. With lower y values, fewer outliers are considered, but the higher the y is, the higher the focus lies on the worst-case latencies. Therefore, picking the y in the context of hardware dimensioning is a matter of making a trade-off between being conservative with a higher y, when we take into account the latency outliers, resulting in over-dimensioning and extra costs, and a weaker performance guarantee, with a lower y.

2.4 Hardware Dimensioning

To ensure the performance of a CPS, sufficient hardware needs to be provided for its application. This process is called hardware dimensioning. It is an important part of the development of CPSs, and is often done without much scientific backing. The complexity arises from the fact that a product may have multiple variants, each characterized by significantly diverse functionalities, so a distinct hardware configuration is required for each

instance. While experienced professionals may draw on prior iterations for hardware allocation in established products, this knowledge is unavailable for new products or variants that have very different functionalities.

During the dimensioning stage, the resources for a prototype may not be fully available, except for limited testing infrastructure. Consequently, the prediction of system performance has to be made based on the available resources. In monolithic applications similar challenges persist [22]. Measurements can be done on the testing infrastructure, but a prediction needs to be made on the effect of upscaling more servers with the same piece of software on it.

With the introduction of microservices, performance satisfaction may be achieved differently: by varying the placement of microservices over nodes. The inherent flexibility of microservices allows for performance enhancement through strategic deployment. However, this flexibility poses questions regarding the optimal assignment of services to hardware. Combining or distributing microservices across nodes can significantly influence application performance, particularly in the context of heterogeneous hardware environments.

The primary driving force behind hardware dimensioning is economic efficiency. The overarching objective is to meet performance requirements with minimal resource utilization, thereby mitigating costs. Striking a balance is crucial, as excessive hardware deployment, or over-dimensioning, leads to unwarranted expenses. Conversely, inadequate resource allocation, or under-dimensioning, risks falling short of performance requirements. Notably, over-dimensioning is often preferred over under-dimensioning, as adding hardware once the system is operational is often impractical due to constraints such as limited energy supply or physical space.

3

Related Work

In this chapter, we put this thesis into the context of various state-of-the-art related work. We start with a discussion on related hardware dimensioning approaches in Section 3.1, then we look at various profiling practices in Section 3.2, and lastly, we show some performance prediction methods in Section 3.3.

The dissection of hardware dimensioning into profiling and performance prediction is based on a literature study that looks into the current state-of-the-art and research directions of hardware dimensioning [12].

A comparison of the related work to this thesis is shown in Table 3.1, Table 3.2, and Table 3.3.

3.1 Hardware Dimensioning

In this section, we introduce two distinct methods of hardware dimensioning in different fields. Both methods perform online hardware dimensioning, where potential solutions can be evaluated by running them on complete hardware configurations.

De Filippo et al. [23] aim to satisfy quality-of-life constraints or budgets of AI applications. The idea is to integrate the domain knowledge of experts with data-driven models to learn the relationships between hardware resource consumption and AI algorithm performance. The approach involves evaluating various AI algorithms across diverse hardware resources, and creating data to train machine learning models. Then, optimization techniques are used to identify the optimal hardware configuration that complies with budget, time, or solution quality requirements. The work differs from this thesis as it solely considers AI applications, and uses machine learning with online hardware dimensioning, as an AI application is running on various complete hardware configurations. The goal of

this thesis, on the other hand, is to determine the required hardware resources and initial deployment configuration before we have all compute nodes available.

Purnaprajna et al. [24] present a scheme for time and power-efficient embedded system design for hardware-software partitioning and hardware allocation. They use genetic algorithms to come to an optimal hardware configuration in terms of power usage and execution time of tasks for embedded system design. An objective function is created that aims to minimize both aspects. Various hardware configurations are then evaluated by running the hardware and software components in full. This work considers embedded systems with hardware and software components, however, it evaluates its hardware configurations through online performance measurements to come to an optimal solution. This means that it addresses hardware dimensioning by employing automated design space exploration and then measuring the results of a fully developed system. In contrast, we perform manual design space exploration by picking a few relevant deployments and then predicting performance instead of measuring.

As these approaches do not consider hardware dimensioning before application deployment, nor do they consider microservice-based applications, we need to look further than the current hardware dimensioning approaches to come closer to solutions for our problem described in Section 1.1. Thus, we introduce work that regards component-based profiling and performance prediction approaches, to provide insight into the state-of-the-art in those areas as well.

3.2 Profiling

Profiling is the process of monitoring and analyzing the behavior, performance, and resource utilization of an application or system. It can be used to gain valuable insights into the functioning of an application or system and may pinpoint areas that can be improved. It is instrumental for optimizing performance, ensuring efficient resource utilization, and identifying potential areas for improvement in the overall functionality of the software or system.

There is a distinction between application profiling and system profiling. Application profiling relies on design knowledge and can consider specific metrics tailored to the application or its components. For instance, Do et al. [25] focus on application-specific metrics like HTTP requests processed per second for Apache or frames processed per second for X264. Alternatively, more general metrics, such as CPU and memory usage, can be employed for profiling various applications, as demonstrated by Han et al. [26]. These broader

metrics offer utility beyond individual applications and can be applied to profile any general application.

System profiling generally encompasses available resources in hardware, or resource utilization. For example, Bao et al. [27] use indicators like CPU cores, RAM, and storage to define a system's profile. Profiling metrics may vary broadly based on the purpose, development stage, and target hardware and infrastructure [28].

The related work is categorized into online and offline profiling. Online profiling [29, 30, 31] is conducted while an application or component is actively in use. Offline profiling is done before an application fully runs on its target system. The dynamic approach of online profiling allows real-time data gathering and deployment optimization, resulting in a higher quantity of available monitoring data. Online profiling, like its offline counterpart, commonly involves metrics such as memory, CPU, and network utilization, and for both profiling methods, the goal is generally to optimize resource usage or predict Quality of Service satisfaction. Unlike offline profiling, online profiling can adapt to changing environmental situations. However, it is not applicable for hardware dimensioning, where we do not have access to the full system for comprehensive data gathering.

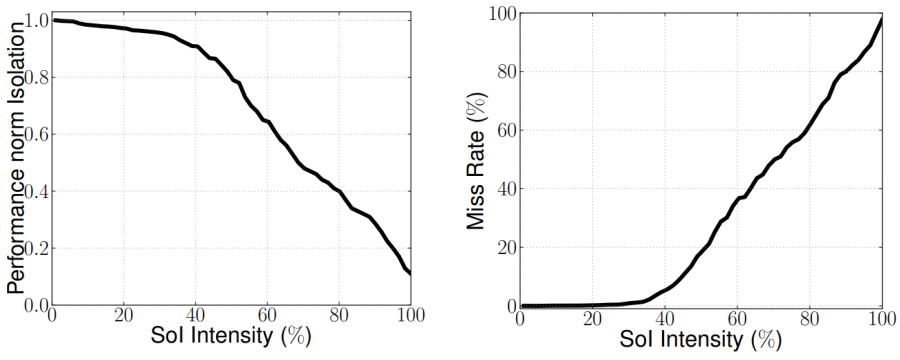
Several papers explore offline profiling [32, 33, 34]. This can be performed on early available hardware without requiring the full context of its final deployment. These approaches typically focus on key aspects affecting microservice-based application performance in varying deployment configurations: communication overhead and performance interference.

For example, Joseph et al. [34] look at communication overhead to enhance Kubernetes' [35] random microservices placement by introducing IntMA, a profile represented as an interaction graph. This graph quantifies communication between microservices through edge weights. Each microservice is represented by a weighted vertex containing its processing and memory requirements. The application profile is then used to optimize microservices deployment across nodes, minimizing inter-service communication. IntMA differs from the approach of this thesis as it does not predict performance, but optimizes deployment of microservices based on a communication overhead profile. The method does not indicate whether performance requirements are satisfied or how much performance may increase through the deployment optimization approach, but rather works under the assumption that minimizing communication overhead will increase performance regardless of the application.

Adeppady et al. [36] employ the concept of identifying interference patterns by running microservices individually and then in pairs on a node. Performance degradation occurs when two microservices compete for resources compared to their isolated execution. This

variance is incorporated into a microservice's profile as a contentiousness vector, where each value represents a pair of microservices. The collective set of vectors forms the application profile. The profile is then used to predict the request throughput of a microservice for dynamic redeployment of microservices in the cloud. It differs from this thesis as its main focus of profiling is on interference, which we do not yet do. Additionally, the profiling approach and performance model are validated at runtime, as the goal is the optimization of the deployment configuration through dynamical redeployment, whereas this work solely considers an initial deployment configuration and the prediction of performance requirement satisfaction.

Delimitrou et al. [33] devise an application profile using interference workloads, employing a contention benchmark suite with 15 Sources of Interference (SoI). These SoIs highlight interference among co-scheduled applications in data centers or on the same server. Data centers, central to cloud computing, offer hosting and management services for servers and diverse applications. An example of an SoI is the LLC-SoI (Last Level Cache Source of Interference). It ranges from 0% to 100% intensity, affecting the LLC occupancy on its node. Figure 3.1 demonstrates that the mfc application from SPECCPU2006 [37] experiences performance degradation with higher LLC occupancy by the SoI. The LLC miss rate of mfc also increases with SoI intensity, indicating sensitivity to LLC interference. An application is profiled based on all SoIs that influence the performance of an application. This profile aids in scheduling applications on servers, avoiding pairing those with similar profiles to minimize performance degradation from interference.



(a) **(b)**

Figure 3.1: The mfc application performance from SPECCPU2006 benchmark suite, influenced by increasing intensity of an LLC-SoI (Last Level Cache Source of Interference). 3.1a shows the normalized performance of mfc with increasing intensity of the LLC-SoI. 3.1b shows the LLC miss rate of mfc with increasing intensity of the LLC-SoI

3.3 Performance Prediction

Based on the profile of an application and system, a performance prediction model of an application can be constructed. The approach to performance prediction may vary based on the purpose and the definition of performance.

Some related work uses machine learning [29, 30, 38, 39, 40] to dynamically predict and improve the performance of an application in a cloud environment. The work of Rahman et al. [11] also falls under this category. It aims to predict end-to-end tail latency for microservices based on their CPU utilization in VMs and pods, a directed acyclic graph (DAG) of the application, and last-level cache usage as inputs for a machine learning model. The model, a deep neural network approach [41], demonstrates the highest accuracy among various machine learning methods. The prediction, aligned with application SLOs, informs recommendations for upscaling the number of instances per microservice. The objective is to meet SLOs, dynamically predict short-term resource upscaling needs, and minimize resource utilization. Since the primary goal is the prediction of the need for upscaling in the short term, a machine learning model is suitable in this context, as much monitoring data is available for an application. The purpose of the performance prediction model differs from ours. The method of performance prediction is also heavily based on online profiling, which is not possible in our hardware dimensioning approach.

Generally, machine learning models appear to be suitable for online performance prediction, where much data can be gathered to fit the model [12]. In cases where not much data is available, performance prediction is done with an analytical model. Chen et al. [42] and Bao et al. [27] use similar analytical models to predict the response time per application component. Although the purpose and environment of the works are different, the performance definition in both works is defined as the response time or end-to-end latency of an application. The prediction is made by analyzing the response time of the components in an application and some form of the sum of the response times is then used as the application response time.

Chen et al. [42] use a model that forecasts the response time per component based on each component's profile and the number of threads employed by the application. The objective is to allocate the optimal number of threads to minimize response time. The profile encompasses the overhead ratio per concurrent request to access the container, the average processing time for a single thread, and the processing time on back-end resources like databases. The analytical model, represented by Equation (3.1), uses the number of concurrent requests, x, and the number of threads used by the application y. By varying

thread counts and comparing results, the model identifies the optimal thread count for a component. While this work also profiles an application based on its components, it aims to optimize the number of threads within each component of an application, where the core method of performance prediction is restricted to components of an application, rather than the application as a whole. Additionally, deployment configurations of components of an application are not taken into account in the model. The purpose of the performance prediction is therefore different, as well as the scope of the model.

$$T = ax + \frac{bx}{y} + cy \tag{3.1}$$

Bao et al. [27] focus on predicting response time for microservices in a cloud environment to minimize costs in public cloud scheduling. VMs, deployed with one or more microservices, incur assigned costs. The model aims to predict both microservices' execution time and the associated monetary costs on VMs in a cloud environment, where microservices are dynamically deployed and upscaled on demand. The execution time of microservice i at time t is decomposed in Equation (3.2), where $ET_i(t)$ is the microservice's execution time, $IT_i(t)$ represents initialization time on the cloud infrastructure, $PT_i(t)$ is the processing time dependent on concurrent microservices, VM, and node, and $RT_i(t)$ indicates the time for microservice i to transfer output data to succeeding microservices. While this work also decomposes an application into microservice components and analyzes the behavior of each microservice, the purpose of this model is different from the prediction model of this thesis. The performance prediction fuels a decision on deployment or upscaling. Based on expected user demand, microservices are deployed, and the decisions are based on monitored data in a fully developed cloud environment. In this thesis, we start with the notion that we do not have a full environment available yet, and we want to predict a suitable hardware configuration.

$$ET_i(t) = IT_i(t) + PT_i(t) + RT_i(t) \tag{3.2}$$

Adeppady et al. [36] show a prediction model that does not predict the response time of an application, but rather the throughput per microservice. As described in Section 3.2, they build a profile of an application based on the contentiousness of each microservice. Its prediction model consists of two components: An offline component and an online component. The offline component leverages the contentiousness of microservices. This contentiousness is computed by running microservices isolated and then in pairs, where

the contentiousness is calculated from the difference in performance between those two setups, with increasing load for the microservices. The contentiousness of each microservice is represented by a vector, where each value in the vector represents the interference of each pair of microservices. The second component of the prediction model is based on the expected interference, or contention, for microservice X. The throughput of X can be predicted based on the contentiousness vector of microservices Y and Z with an online regression model [43]. The full prediction model is then used for dynamic microservice placement over nodes in the cloud, where minimizing the throughput of each microservice is the goal. The purpose of this model differs from hardware dimensioning, and is fueled by run-time monitoring. In this sense, it differs from the work in this thesis.

In conclusion, as shown in Table 3.1, no related work provides a method where a component-based performance prediction model is made to find a suitable initial deployment configuration for hardware dimensioning. In this thesis, we aim to provide a structured approach to hardware dimensioning, through an application profiling and performance prediction method, that fills this gap in the research.

Table 3.1: A characterization of the profile and prediction methods and context of the related work

Related works	Context		Profile
	Environment	Purpose	Application/System Profiling
Do et al. [25]	VMs in the cloud	Optimize resource usage	Both
Han et al. [26]	MSs in the cloud	Optimize resource usage	Application
Bao et al. [27]	MSs in the cloud	Optimize cloud costs	Both
Aksakalli et al. [32]	MSs in the cloud	Predict QoS satisfaction	Both
Delimitrou et al. [33]	Applications in datacenters	Minimize interference	Application
Joseph et al. [34]	MSs in the cloud	Minimize application response time	Both
Chen et al. [42]	Applications in datacenters	Predict QoS satisfaction	Both
Adeppady et al. [36]	MSs in the cloud	Optimize resource usage	Both
Ma et al. [39]	MSs in distributed resource centers	Minimize MS idle rate	Both
Lv et al. [30]	MSs in edge computing	Optimize resource usage	Application
Grohmann et al. [29]	MSs in the cloud	Prevent performance degradation	Application
Rahman et al. [11]	MSs in the cloud	Predict end-to-end latency	Both
Zhang et al. [40]	MSs in the cloud	Predict QoS satisfaction	Both
Fu et al. [31]	MSs in cloud-edge continuum	Optimize resource usage	Both
This thesis	MSs in CPS	Hardware Dimensioning	Application

Table 3.2: A characterization of the profile and prediction methods and context of the related work

Related works	Profile	Profile metrics						
	Offline/Online	Communication overhead	Contention	Application-specific metrics	CPU	Memory	Network	Trace Latencies
Do et al. [25]	Offline			X	X	X		
Han et al. [26]	Offline	X			X	X	X	
Bao et al. [27]	Offline				X	X		X
Aksakalli et al. [32]	Offline	X			X	X	X	
Delimitrou et al. [33]	Offline		X		X	X	X	
Joseph et al. [34]	Offline	X			X	X	X	
Chen et al. [42]	Offline				X	X		X
Adeppady et al. [36]	Both		X		X	X		
Ma et al. [39]	Online				X	X		
Lv et al. [30]	Online	X			X	X	X	
Grohmann et al. [29]	Online	X			X	X	X	
Rahman et al. [11]	Online		X			X	X	
Zhang et al. [40]	Online	X	X		X		X	
Fu et al. [31]	Both	X	X		X	X	X	
This thesis	Offline	X			X			X

Table 3.3: A characterization of the profile and prediction methods and context of the related work

Related works	Prediction model	
	Model type	Predicted metric
Do et al. [25]	Analytical	Performance difference
Han et al. [26]	None	None
Bao et al. [27]	Analytical	Processing time/cost
Aksakalli et al. [32]	None	None
Delimitrou et al. [33]	None	None
Joseph et al. [34]	None	None
Chen et al. [42]	Analytical	Processing time
Adeppady et al. [36]	Analytical + ML	MS throughput
Ma et al. [39]	None	MS idle rate
Lv et al. [30]	None	Application response time/load balance
Grohmann et al. [29]	Analytical + ML	Performance degradation
Rahman et al. [11]	ML	Tail latency
Zhang et al. [40]	ML	QoS satisfaction
Fu et al. [31]	ML	Latency/throughput
This thesis	Analytical	CPU usage/Tail latency

4

Case Study

The profiling and performance prediction methods introduced in this thesis are validated against an example case study: The Meal Delivery Service (MDS). This chapter presents that case study, which includes the MDS, its performance requirements, and the TNO cluster, on which validation experiments are executed.

4.1 The Meal Delivery Service (MDS)

The MDS is an application that is specially created for research into performance of microservices-based CPSs. It is a prototype that is based on an existing industrial CPS application.

The application consists of four core microservices and one API gateway. The microservices are: 1) the PlannerService, 2) the MealDispatchingService, 3) the MealDeliveringService, and 4) the MealPreparingService. The overview of the interaction between the services is shown in Figure 4.1. In the following sections, we discuss the purpose of the MDS and the functionality of each microservice. In this prototype, each microservice implements one core functionality to keep the microservices small. Generally, this is good practice in microservice systems [44].

The MDS schedules and delivers meal requests based on their urgency. Every second, the PlannerService plans the incoming requests to kitchens and bikes, while impatient customers walk toward the restaurant. The PlannerService schedules the meals so that each meal is cooked and delivered to the corresponding customer, if such a schedule is possible. If meals do not arrive on time, the customer will walk towards the restaurant and complain, and the application has failed.

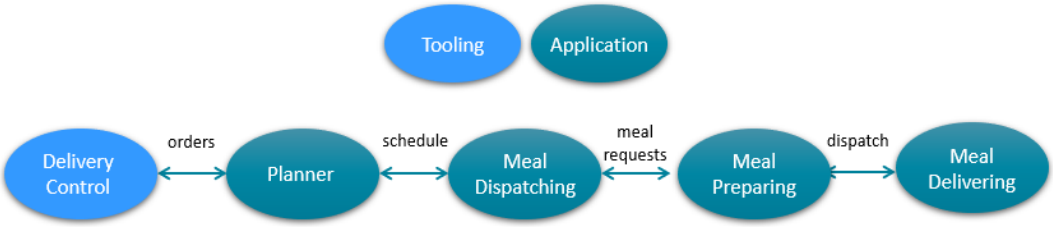


Figure 4.1: Overview of microservices of the MDS prototype

DeliveryControlService

The DeliveryControlService is the API gateway of the MDS. It simply forwards the meal requests it receives to the PlannerService. We will not use this service in our profiling performance prediction examples, as it simple and does not contribute much to the MDS case study.

PlannerService

The PlannerService is the service that schedules the incoming meal requests. It is responsible for receiving meal requests and the transformation into the best-fitting schedule. The schedule is a list of to-be-executed meals, ordered on the delivery time, with the desired kitchen and bike type. The schedule is recalculated every second based on the current and newly incoming meal requests and system status, such as the available kitchens and bikes.

MealDispatchingService

The MealDispatchingService is responsible for executing the schedule to the best of its abilities, considering the current status of the system. Given the latest schedule, it dispatches meals to the MealPreparingService at the scheduled time.

MealPreparingService

The MealPreparingService includes kitchens that can prepare the meals according to the incoming requests of the MealDispatchingService. It may have multiple kitchens and a kitchen may prepare multiple meals at the same time. In the experiments done in this thesis, we assume there are always enough meal preparation slots in kitchens available.

MealDeliveringService

The MealDeliveringService is responsible for the bikes that may deliver the meals to the customers. It serves as a resource manager for the available bikes and hands these out to the MealPreparingService on request, when a meal is prepared.

4.1.1 Request Types

There are three distinct request types for the MDS: Meal Request, Cancel-Meal Request, and Reset-Meals Request.

Meal Request

The MDS processes meal requests and sends them to customers. They form the basis for the load on the MDS. Each request has a delivery time, which indicates when a meal should be delivered, a customer speed, which indicates how fast a customer moves, and a customer distance, which indicates from where the customer walks. Based on these variables, a priority ordering can be made, where the most urgent meals are placed at the front of the list. In our experiments, we keep each of these variables constant, while varying the number of simultaneous meal requests that are requested to provide varying workloads to the application, which is further described in Section 5.5.2. Additionally, a distinction between standard meal requests, and platinum meal requests can be made. Platinum meal requests will always have priority over standard meal requests.

Cancel-Meal Request

The cancel-meal request can be used to cancel a meal, which indicates that it does not need to be delivered to a customer anymore.

Reset-Meals Request

A reset-meals request resets the whole application. The schedule is emptied, the customers disappear, and any meal, that is in the process of being delivered, is canceled.

4.1.2 Artificial Load

As the effect of increasing load on the CPU usage of each microservice turns out to be fairly low, we have introduced some artificial load in the PlannerService. The artificial load aims to capture the load expected from a planner with industrial complexity. Additionally, the goal of this load is to see whether we can predict the artificial trends accurately with

our profiling and prediction methods. It is introduced with a function, *spendTime*, which increments an integer by one in a while loop for the amount of time, t, in milliseconds, which is calculated as shown in Equation (4.1). Here, s is the list of incoming meal requests in the PlannerService, and $|s|$ represents the number of meal requests in the list.

$$t = 10 \cdot |s| \tag{4.1}$$

This artificial load introduces extra CPU load in the PlannerService that is linear to the number of simultaneous requests in the schedule of the PlannerService. In addition, the latency of each trace that includes this function will be dependent on the number of simultaneous meal requests in the schedule. For example, if the schedule includes 50 meal requests, the trace will indicate a latency of at least 500 ms.

4.2 Performance Requirements

Alongside the MDS functionality, performance requirements have been specified. They have been constructed so that they mimic the performance requirements of the existing industrial CPS application.

With this, we identify two distinct categories of requirements: 1) Resource requirements, which state that the application should not use more than x% of an available resource, like CPU or memory usage, and 2) latency requirements, which define that certain sequences within an application should take no longer than a specified amount of time. The former has an identifier of the form RCR-X, and the latter one of the form LR-X.

We specify one resource requirement as follows:
RCR-1: A node should never use more than 60% of the available CPU computational power over 15 seconds.

We specify the following latency requirements:
LR-1: A new standard meal request is added to the PlannerService queue and offered to the MealPreparingService $<$ 2000 ms
LR-2: A new platinum meal request is added to the PlannerService queue and offered to the MealPreparingService $<$ 500 ms
LR-3: The time from meal delivery until the reporting to PlannerService $<$ 100 ms
LR-4: If a user submits a reset-meals request then a reset response should be completed $<$ 1000 ms

LR-5: If a user submits a meal-cancel request then a meal-cancel response should be completed $<$ 1000 ms
LR-6: The meal request schedule is planned in the PlannerService once requested $<$ 480 ms
LR-7: Once a meal can be dispatched, it is sent to the MealPreparingService and dispatched $<$ 2500 ms
LR-8: The preparation status of the MealPreparingService is updated, once requested $<$ 100 ms
LR-9: The delivering status of the MealDeliveringService is updated, once requested $<$ 100 ms

In this thesis, the approach is validated for four latency requirements, because the goal of the case study in this thesis is not to fully analyze the MDS application, but rather to illustrate how to profile an application and predict its performance. The requirements that are taken into account are **LR-1**, **LR-3**, **LR-6**, and **LR-7**.

The four latency requirements are chosen for the validation, since, they together, span the four core microservices in the MDS. Additionally, their requirements range from 100 ms to 2500 ms, and the latencies of spans may largely contribute to the latencies in the system workflow for some, while the communication may play a larger part in others. With this, we validate whether the model can be applied to a diverse set of latency requirements spanning any microservice in an application.

The picked requirements are a part of the meal request system workflow, shown in Figure 4.2. Both the sequence diagram and the spans that make up a latency requirement have been identified with application knowledge in the design phase of the application.

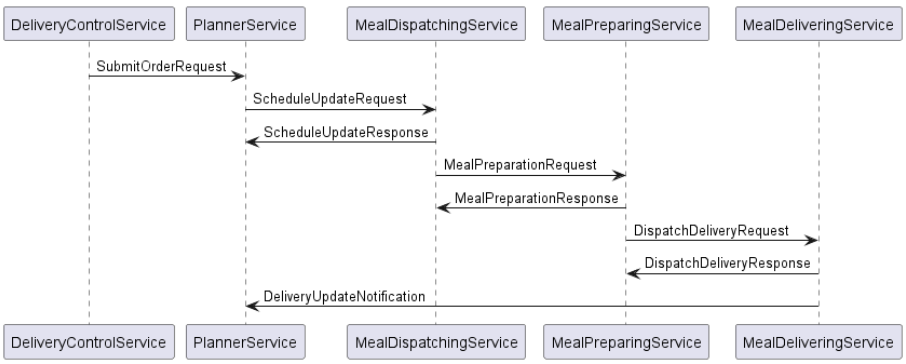


Figure 4.2: Sequence diagram of the meal request to delivery-workflow of a meal request in the MDS.

The sequence diagram indicates that the meal request starts at the DeliveryControlService, which forwards the SubmitOrderRequest message to the PlannerService. Then the PlannerService sends the schedule with all meal requests to the MealDispatchingService, through the ScheduleUpdateRequest. It then submits a MealPreparationRequest to the MealPreparingService to signal that a meal can be prepared. Once that is the case, a DispatchDeliveryRequest is propagated to the MealDeliveringService to indicate that a meal can be delivered. Once the meal is delivered, the DeliveryUpdateNotification is sent, as an acknowledgment to the PlannerService that a meal has been delivered. Note that the ScheduleUpdateRequest, MealPrepartionRequest, and DispatchDeliveryRequest have a response variant, which functions as an acknowledgment that the corresponding message has been delivered and processed correctly.

The system workflow of each latency requirement in the context of this sequence diagram is shown in Figure 4.3, Figure 4.4, Figure 4.5, and Figure 4.6. The workflows are indicated by their distinct color.

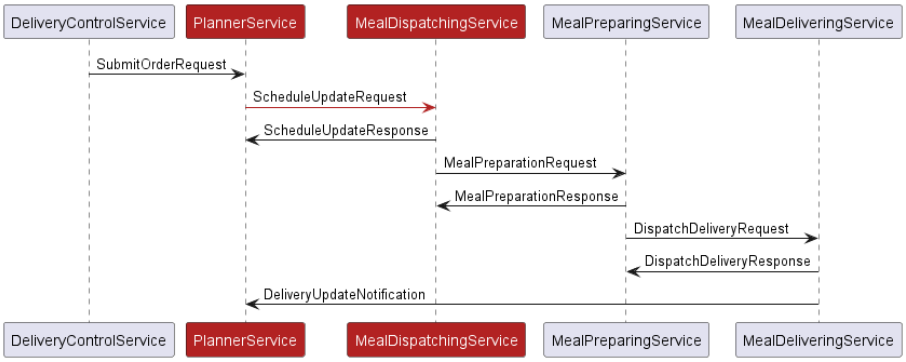


Figure 4.3: The workflow of latency requirement 1, LR-1

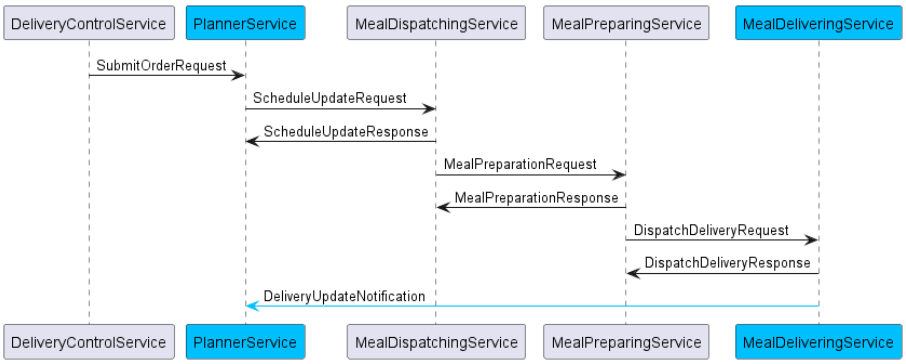


Figure 4.4: The workflow of latency requirement 3, LR-3

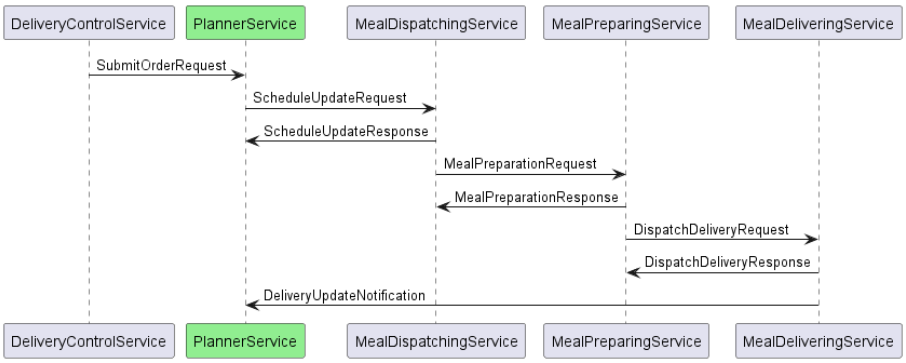


Figure 4.5: The workflow of latency requirement 6, LR-6

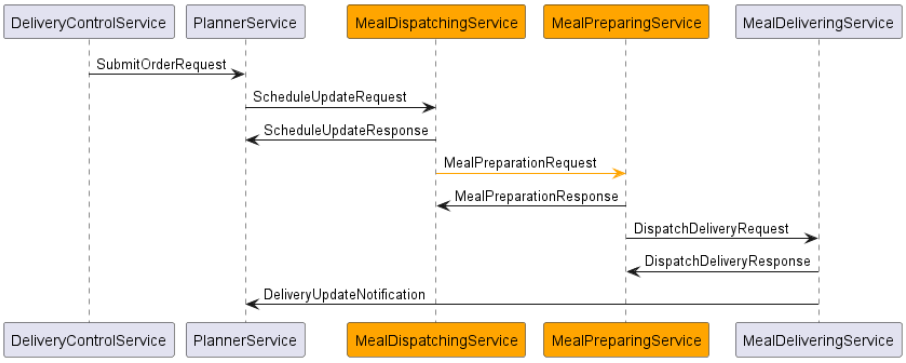


Figure 4.6: The workflow of latency requirement 7, LR-7

4.3 Hardware and Experiments Environment

The application profiling experiments and the performance prediction validation experiments are performed in a Kubernetes environment [35]. Each service in the MDS is deployed in a container in its distinct pod, where each container resembles a virtual machine with allocated resources. We have not limited the resources a container may use, except for the specified resources per node.

The containers run on the TNO cluster, which consists of three nodes: A control plane, Archview01, and two worker nodes, Transact01, and Transact02. The MDS microservices are only deployed on the worker nodes. Specifications of each node are shown in Table 4.1.

Table 4.1: Specifications for each node in the TNO cluster

Node	CPU cores	CPU name	CPU base frequency (GHz)
Archview01	4	Intel(R) Xeon(R) Gold 5115	2.10
Transact01	16	Intel(R) Xeon(R) Gold 6442Y	2.60
Transact02	16	Intel(R) Xeon(R) Gold 6152	2.40

5

Application Profiling

In this chapter, a general method of application profiling is introduced. The profile is used to calibrate the prediction models introduced in Chapter 6 and forms the basis for our hardware dimensioning approach for a microservice-based CPS. The profile is based on the profile of each of its microservices. The method is applied within an automated profiling framework and alongside an example case study of the MDS as discussed in Chapter 4.

The profiling approach consists of three distinct steps: 1) constructing profiling infrastructure by defining deployment configurations, creating a workload suite, and setting up data monitoring services, 2) defining the profiling metrics, and 3) running profiling experiments, where we repeat the experiment a sufficient number of times. Each step will be explained in detail in this Chapter.

The high-level workflow of the automated profiling framework consists of 1) automated deployment, 2) serving requests, 3) data collection, and 4) creating a profile, as shown in Figure 5.1.

Figure 5.1: High-level workflow of automated profiling framework of an application

5.1 Profiling Infrastructure

In this section, we discuss the infrastructure and preparation needed to run profiling experiments.

5.1.1 Deployment Configurations

We use a microservice profile that is as little influenced by other microservices in the application as possible. Therefore, each microservice of an application will need to be profiled in isolation, to avoid interference of other microservices on its profile, and needs its own deployment configuration, as shown in Figure 5.2. An isolation deployment configuration separates the profiled microservices, MS A in Figure 5.2, from the other microservices in an application, MS B through M. Additionally, cluster infrastructure and monitoring services should be placed on different nodes from the application microservices, to minimize any performance interference on the application. An isolation deployment configuration therefore requires one node on which the isolated microservice is profiled, a set of nodes that holds the other microservices within the application, and a set of nodes that deploys any infrastructure and monitoring services.

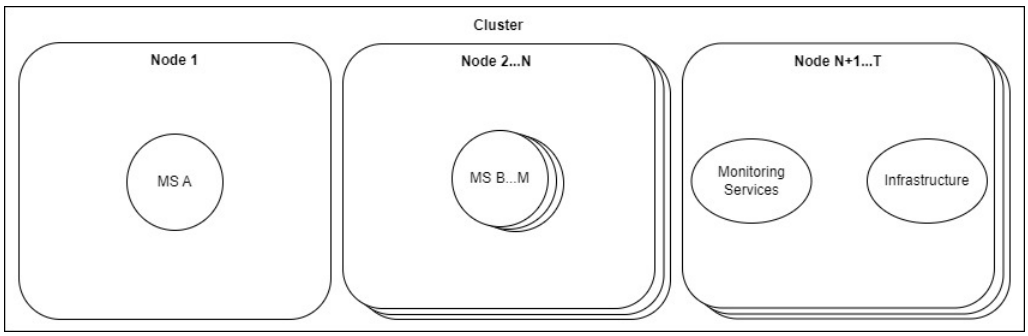


Figure 5.2: Form of an isolation deployment configuration

The set of nodes that hold the rest of the application microservices can be kept as small as possible, provided the microservices on a node fit, and those nodes do not need to match the target system hardware. One arbitrary deployment configuration should be picked over the minimum number of nodes so that all microservices have sufficient resources to function and microservices should only be swapped between nodes based on whether they are profiled or not. The specific deployment configuration of the microservices could result in some effect on the application profile, and the degree of the effect on the profile is application and system-specific, so depending on the context it may still need to be taken into account, but we regard it as outside the scope of this thesis.

The profiling node should be as similar as possible to the hardware of the target system, the system on which the application will be used eventually. The more the profiling hardware differs from the target system, the less we can say about the accuracy of the

performance prediction. The assumption in this work is that the target system will consist of homogeneous nodes, or similar nodes, that match the profiled node, therefore only one node needs to be provided that is similar to the target system. Determining how similar a profiling node has to be is application-specific and may be determined through experiments, as shown in Appendix C.3.

When a deployment configuration is created for each microservice, they can be deployed in turn, ensuring that we profile each microservice one by one. This process should be automated, so that the profile can be made without any further human intervention. Its role is displayed in high-level workflow as shown in Figure 5.1.

Aside from the isolation deployment configurations, we need one more set of additional deployment configurations to retrieve the communication latency of messages within a node. The goal of these deployment configurations is to measure the latency for local communication, between microservices within a node. Remote communication is measured already in the isolation deployment configurations, as each microservice is isolated once. There are three trivial options to retrieve the local communication metrics:

- If possible, deploy all microservices on the profiled node. This may result in interference in the communication latencies of the microservices.
- If we have two similar profiling nodes, we can use the communication latencies of the microservices on the other node in the isolation deployment configuration as a local communication profile. The advantage is that we only need the isolation deployment configurations to set up the whole profile.
- Run pairs of microservices on the profiling node that communicate with each other.

The last option could result in many deployment configurations, which is not scalable for large applications. However, it minimizes interference of other microservices in the results, which may increase prediction accuracy, and guarantees that the microservices will fit on the node. The choice of which deployment configurations are used in this context depends on the available time to profile, the number of microservices in the application, and the available hardware.

5.1.2 Workload Suite

To determine the behavior and characteristics of an application, a workload suite needs to be created. It consists of one or more request sets, where a request set is a number

of requests to an application. The goal of the workload suite is to measure the effect of different loads on an application, so that the characteristics of an application can be profiled. It can be built 1) upon prior knowledge of an application's behavior, 2) on insight into what the expected user profiles are, or 3) the load can grow until a certain metric limit is met, like CPU or memory usage, or if a latency requirement is violated. 4) A workload suite can also be built based on agreements between various users and stakeholders of a system on how much load an application should be able to handle. It is thus heavily application and system-dependent. A suitable workload suite for hardware dimensioning will show the effect of load applied on the performance requirements that are specified for an application. Additionally, if specific workflows within an application need to be included in the performance prediction the workload suite should include requests that are expected to follow those workflows.

To profile through the automated framework, an automated method of serving requests needs to be made as well, as shown in Figure 5.1. After a deployment configuration is employed, the workload suite can be served to the application, representing experiments from which the application profile can be built.

5.1.3 Data Monitoring and Collection

The data collection for the profile is done by observability tools that can extract resource usage metrics of individual microservices. Additionally, they should be able to collect the latencies of all relevant traces, where the relevant traces include all workflows that will be analyzed in the performance prediction as described in Chapter 6. Observability tools as described in Section 2.3 can be used in this process.

The choice of tools and their configuration in this approach may drastically influence profiling results, as monitoring infrastructure may use a significant part of a node's resources, for example. This could cause a profile to include the interference patterns of a microservice with the monitoring services, which can cause performance predictions to be off.

In Section 5.5.2, we provide an example pipeline to extract CPU usage metrics and an example pipeline to extract latency metrics.

Once the data monitoring and collection are done, we can either use the next deployment configuration to gather data for the next microservice in the application, or, if the experiment on each deployment configuration has been completed, we can start creating the application profile, as shown in Figure 5.1 and explained in Section 5.3.

5.2 Defining Metrics

The data that is gathered forms the basis for the application profile. A profile can consist of various metrics, based on the profiled application and system and its requirements. Most application performance requirements consist of both resource requirements and latency requirements. This profiling method therefore considers both resource metrics, in the form of CPU usage, and traces, from which latencies within an application can be extracted. Which metrics and traces are extracted depends on the specified performance requirements and are therefore also application and system-dependent. If a resource requirement sets a limit to the CPU usage of microservices on a node, then CPU usage metrics should be extracted as a part of the application profile. The necessary infrastructure to extract the metrics and traces depends on the type of metrics and traces as well.

5.3 Creating a Profile

Once the required traces and metrics are gathered for all microservices in an application, the profile can be built. The application profile is split up into resource usage metrics and latency metrics, according to the two requirement types that are generally regarded in microservice applications: Resource requirements and latency requirements.

5.3.1 CPU usage Metrics

The resource usage component of the profile consists of two parts: Application CPU usage, consisting of the CPU usage of each microservice in the application, and the system overhead. The profile is based on the monitoring data gathered from the automated profiling framework.

5.3.1.1 Microservice CPU Usage

For each experiment, we use the highest observed CPU usage value of an isolated profiled microservice for a given request set in the workload suite. We collect the average CPU usage of the microservice over intervals of T seconds, the peak CPU usage is picked as shown in Figure 5.3.

In this example, the CPU usage of a microservice in an application is shown against the time for one experiment. The observed CPU usage data consists of four values: A, B, C, and D. They represent CPU usage averages over their corresponding interval. For the

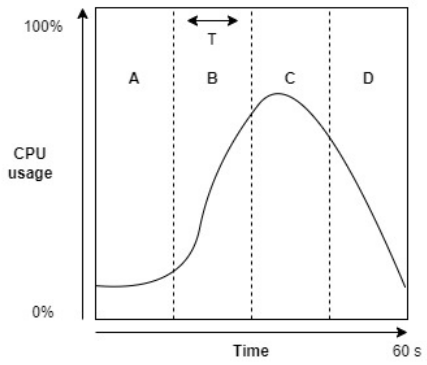


Figure 5.3: An example of how to retrieve the CPU peak of a microservice in a profiling experiment.

profile of a microservice, we pick the highest average value, or the peak value of the four values, which in this example would be C.

We choose to store the peak CPU usage value of an experiment in the profile, because we aim to catch the worst-case scenario. The profile will form the basis of the performance prediction, which will influence the hardware dimensioning. If we take into account the worst-case situation of a microservice, the profile is kept conservative and the hardware dimensioning is likely to be on the conservative side as well. This means that over-dimensioning is more likely than under-dimensioning. As discussed in Section 2.4, under-dimensioning is generally less harmful than under-dimensioning.

5.3.1.2 Overhead

An important aspect of the performance prediction of the CPU usage of an application is the overhead that an application may generate. The overhead is heavily dependent on the system and environment the application runs in. It is calculated for isolated microservice as the difference between the peak node CPU usage and the peak microservice CPU usage on that node for each distinct load defined by the workload suite. The overhead thus encompasses anything that causes extra CPU usage, aside from the microservice's CPU usage. This involves application infrastructure, the communication overhead generated by the application framework, and monitoring services that poll from the application.

We apply fitting to capture the trend of the overhead of each microservice in an application for varying load. The form of the fitting is heavily application-specific. In this method, we discuss a linear fitting as shown in Equation (5.1), so we can include two aspects of the overhead: 1) b, The overhead offset, which is a constant for each microservice and

represents the minimum amount of overhead a microservice, and the node it is running on, generate, and 2) a, the overhead slope, which is dependent on a microservice and its load. x represents the load based on the workload suite. The overhead offset likely includes some load from the system infrastructure and part of the monitoring services on the node, while the slope includes the communication overhead generated by the application framework. It is possible that the trend in the overhead is not linear with the load. In that case, a different fitting may need to be identified and applied based on the application.

$$ax + b \tag{5.1}$$

A separate experiment has to be conducted to calculate the idle offset of each microservice. In this experiment, each isolation deployment configuration is used, but no load is put on the application. This means that each isolated microservice will be idle, and the overhead generated represents the overhead offset of a microservice. This experiment can be conducted for any amount of time, where the longer it is conducted the more conservative the offset values will be. In the examples mentioned in this thesis, a time of 10 minutes is used from which the peak CPU usage of each idle microservice is taken. If the general minimal overhead of microservices on a system is already known, that can also be used as value for b. Additionally, if it is known that the overhead offset is fairly similar for all microservices, it is possible to choose one average value of b for all microservices, which may simplify the application profile if an application includes many microservices.

If the variability in the overhead of a microservice for varying loads is high, it may be beneficial for the hardware dimensioning to increase the value for the overhead offset so that the overhead prediction will stay on the conservative side. A possibility is to increase b, so that all profiled overhead for each load stays under the fitting. If the variability is so high that a relation between the load and overhead is barely visible, it might indicate that there is a need for additional profiling, and the experiments have to be run more often to gather more data.

5.3.2 Traces and Latency Metrics

The latency profile consists of two parts: The profile of the spans within microservices, and the communication profile. A profile is made for each relevant system workflow of an application, rather than for each microservice.

We assume that a latency requirement that represents a system workflow of an application generally has the form: "Once started, a system workflow should complete within a

certain time frame". Note that we may not want to guarantee that 100% of all instances of a system workflow are completed within a specified time. Therefore, we can use the yth percentile tail latency as a representation of y% of instances of a system workflow completing within the specified time frame.

In this context, y has to be chosen depending on how conservative we want to be in satisfying timing requirements. The higher the y, the higher the guarantee of requirement satisfaction, but picking a high y may also result in over-dimensioning, as outliers will dictate the performance prediction. The percentile tail latency may also differ per requirement if the criticality of requirements differs, but cannot differ within a requirement, as this will complicate the performance prediction. The exact y for the percentile latency to be profiled should therefore be carefully considered.

5.3.2.1 Microservice Spans

Each system workflow is built up by one or more spans within at least one microservice. Therefore for the profile, we need to collect the latency of the spans within a microservice. Just as for the CPU usage profile component, each microservice included in the profile is isolated on a node when it is profiled. The effect of varying loads is then measured in the profile as the difference in tail latency between them.

Note that the spans within a microservice related to one system workflow can differ from the spans within the same microservice for another system workflow. Two aspects can differ: 1) The events that mark the start and stop of a span within a microservice, and 2) the request type and system workflow may follow different spans within a microservice. Therefore, the behavior and role of a microservice can differ heavily per system workflow that it is a part of.

5.3.2.2 Communication

The latency profile not only consists of the spans within microservices, but also includes the communication between microservices. An important parameter for the prediction of the latencies of the traces is the difference between the communication latency between microservices that are running together on a node, "local communication", versus the communication latency between microservices on separate nodes, "remote communication". The profile should include the latency of both communication types for each message that is included in the traces for the relevant system workflows.

The communication latency may be heavily influenced by the load on microservices, for example, if its message queue fills up. If that is the case a profile should gather the percentile tail latency for each load separately. However, if it is known that the communication latency is not influenced by load, there is no need to measure the latencies separately for each message. An average communication latency for each message may suffice for the profile, which can help simplify the profiling method.

5.4 Iterative Profiling

The profiling method described in this chapter so far discusses one iteration of profiling. However, it is possible that multiple profile iterations need to be conducted to collect sufficient data, as is shown by the arrow that creates a loop in Figure 5.1. The results of multiple iterations should then be aggregated into average values of the profiles. If results show a lot of noisy behavior, this could be an indication that there is a need to upscale the number of iterations for the profile. In general, the more iterations a profile has, the more reliable the performance prediction will be, since it is based on a bigger data pool. Of course, running a profile experiment may be time-consuming, especially if an application contains many microservices.

The amount of profiling iterations needed to ensure a reliable performance prediction is context-specific and depends on the profiled application, the available time, and the expectations of the stakeholders involved. Thus, for any distinct application, a feeling should be created for how many times an experiment has to be repeated to ensure that noise and variations are evened out.

5.5 The MDS Profile

This section displays the MDS case study as an example to show how the profiling framework can be implemented. The full MDS profile is shown in Appendix A.

5.5.1 Experimental Setup

The profiling experiments are performed on the TNO cluster described in Section 4.3. The measurements will be done on Transact01. The MDS as described in Section 4.1 is used to show an example of how to use the profiling method, and consists of four distinct microservices: the PlannerService, MealDispatchingService, MealPreparingService,

and MealDeliveringService. All experiments and results shown in this chapter have been performed in this context.

5.5.2 Profiling Infrastructure

We will first show the profiling infrastructure that is needed to construct the MDS profile.

Deployment Configurations

For the MDS, the number of isolation deployment configurations is four, since it consists of four microservices. Each service is deployed in isolation on Transact01. The other three microservices in the MDS are then deployed on node Transact02 and the remaining infrastructure and monitoring services run on Archview01. An example of a deployment configuration where the PlannerService is isolated is shown in Figure 5.4.

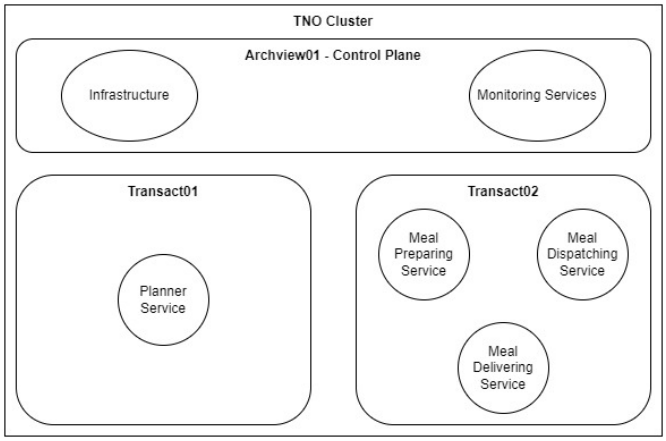


Figure 5.4: An example in the MDS context of an isolation deployment configuration for application profiling.

The automated deployment step requires each microservice in the MDS to be automatically deployed on the node that is specified in the deployment configuration. This is accomplished by dynamically editing the YAML file of each microservice by specifying the node on which it should be deployed. Then the deployment is performed with a script that uses Kubectl, a command tool of Kubernetes [35]. There is no need to redeploy the infrastructure services, since those are always deployed on the same node, Archview01.

For the local communication measurements, we can use one deployment configuration that places all microservices on one node, since it fits on Transact01. This deployment configuration is shown in Figure 5.5. We have chosen this configuration, since it simplifies the data processing for the communication profile, and the expectation is that the difference

caused by the contention between microservices is minimal in the MDS context, further supported by the experiment in Appendix C.2.

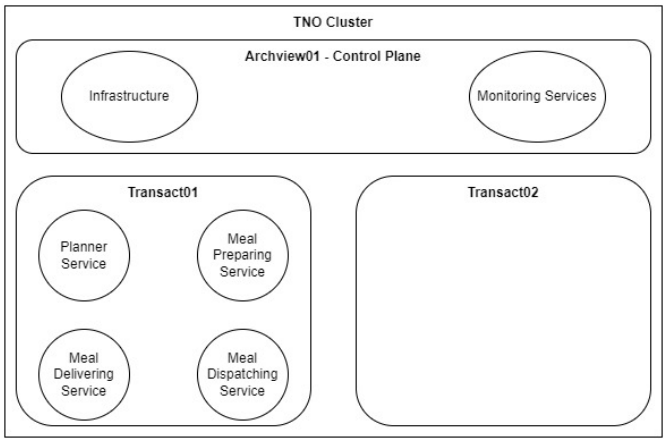


Figure 5.5: Deployment configuration for profiling local communication latencies for the MDS

Workload Suite

In the context of the MDS, the workload suite consists of meal order requests. The application will be profiled based on an increasing number of simultaneous requests, so that the behavior of the application with an increasing load is shown. In this case study, the number of simultaneous requests served goes from 0 to 100 with incremental steps of 10. The number of simultaneous requests could be chosen based on the artificial latency pressure in the PlannerService, and the performance requirements that concern that service, **PR-1**, **PR-3**, and **PR-6**. Since the artificial latency in the PlannerService is 10ms per order request as discussed in Section 4.1, the order latency is expected to grow with the number of simultaneous requests. The expectation is then that latency requirement **PR-6** will be violated for 50 simultaneous orders and above, as we assume that requests cannot be processed concurrently in the PlannerService. Although the workload suite could, thus, be limited to 50 simultaneous orders, this case study will try to push the limits of the MDS application, and we set the number of simultaneous orders to 100. Any load above this number proves to be a point of the MDS application where orders show unexpected behavior, and order requests may not arrive on time. We have automated the process of serving requests in the MDS.

Each set of requests in the workload suite represents an experiment. At the start and end of each experiment, timestamps are recorded. These timestamps indicate what experiment

ran at what point in time, and can be used for the data processing of the profile later on.

Data Monitoring: CPU usage Pipeline

For the case study, we have constructed a pipeline that extracts CPU usage metrics for an application in a Kubernetes cluster, as shown in Figure 5.6.

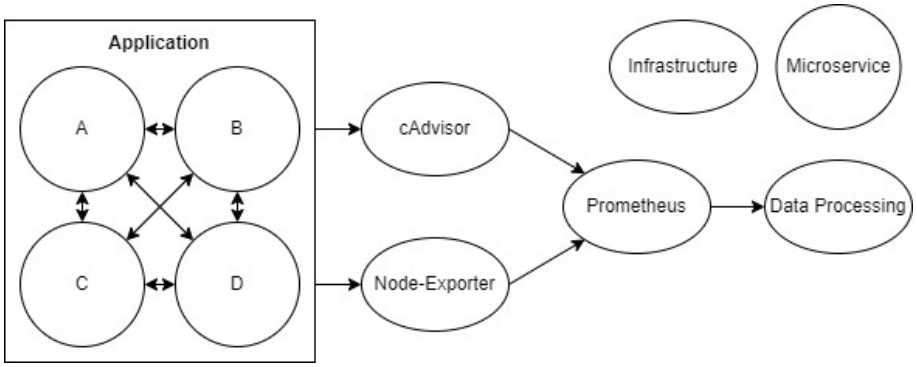


Figure 5.6: Metrics extraction pipeline in a Kubernetes cluster. From application to data processing.

With this infrastructure, the CPU usage data per microservice may be extracted with cAdvisor [45]. It collects CPU usage metrics per microservice in a cluster. Node Exporter [46] gathers CPU usage metrics per node in a cluster. The metrics are then delivered to Prometheus [20], from which the data can be stored in a database or JSON object. Examples of CPU usage metrics are CPU usage average or minimum over a time interval, either per CPU on a node or on average per node.

For each microservice in the MDS, the extracted metrics will include its CPU usage for each workload suite request set. The average CPU usage over an interval is taken per microservice through cAdvisor. Preferably this interval matches the resource requirement.

Note that we use two distinct infrastructure services to record the CPU usage of containers and nodes. This is not ideal, since intervals may not match up in the CPU average interval, causing peaks to be distributed differently over the interval, therefore resulting in different averages for roughly the same interval. An example is shown in Figure 5.7. Here the blue line is the node CPU usage and the red line is the CPU usage of a container that runs on the node.

The example shows that the average container CPU usage over an interval may be calculated as higher than the average node CPU usage that is measured by a different tool when sample timing is not synchronized, even though the node CPU usage is always higher than the container CPU usage. This could especially prove problematic in our calculation

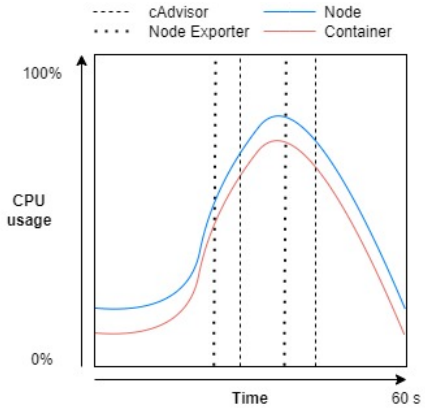


Figure 5.7: An example of how using two tools with different sample timing can result in a skewed comparison.

of overhead, as it requires both tools. In this thesis we still use two tools in this context, since it is what is available in our infrastructure, however, ideally, infrastructure should be built so that one tool gathers both the container CPU usage and node CPU usage. We try to mitigate the problem in this work by creating many profiles and aggregating the results as discussed in Section 5.4, which is why we feel that this will not influence the results significantly.

Also, note that both cAdvisor and Node Exporter extract the CPU usage per core on a node. We take the average over the CPU usage per core to keep the profile simple, but various other metrics can be extracted to form a profile, including a per-core analysis and per-core profile.

Data Monitoring: Latencies Pipeline

We have also constructed a pipeline to gather relevant latencies to include profiling for the system workflows. OpenTelemetry [47] collects all spans from an application, and forwards them to Jaeger [21], which groups the spans into traces. Jaeger then stores these traces in a database tool called ElasticSearch [48]. At this point, the traces are ready for preprocessing for the eventual profile. The pipeline is shown in Figure 5.8. Both pipelines are used in the MDS context.

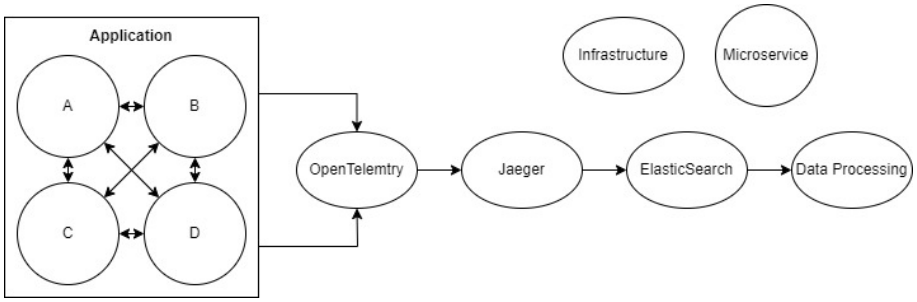


Figure 5.8: Trace extraction pipeline in a Kubernetes cluster. From application to data processing.

5.5.3 Defining the Metrics and Creating a Profile

In the MDS case study, we have defined the metrics we extract as the peak CPU usage of a microservice over 15-second intervals during an experiment, and a 90th percentile tail latency for all system workflows. We take the 90th percentile tail latency as we feel it represents a fairly conservative prediction indicator. We have picked the polling interval through experiments on the case study, although we had some restrictions as discussed in Section 8.3.4.

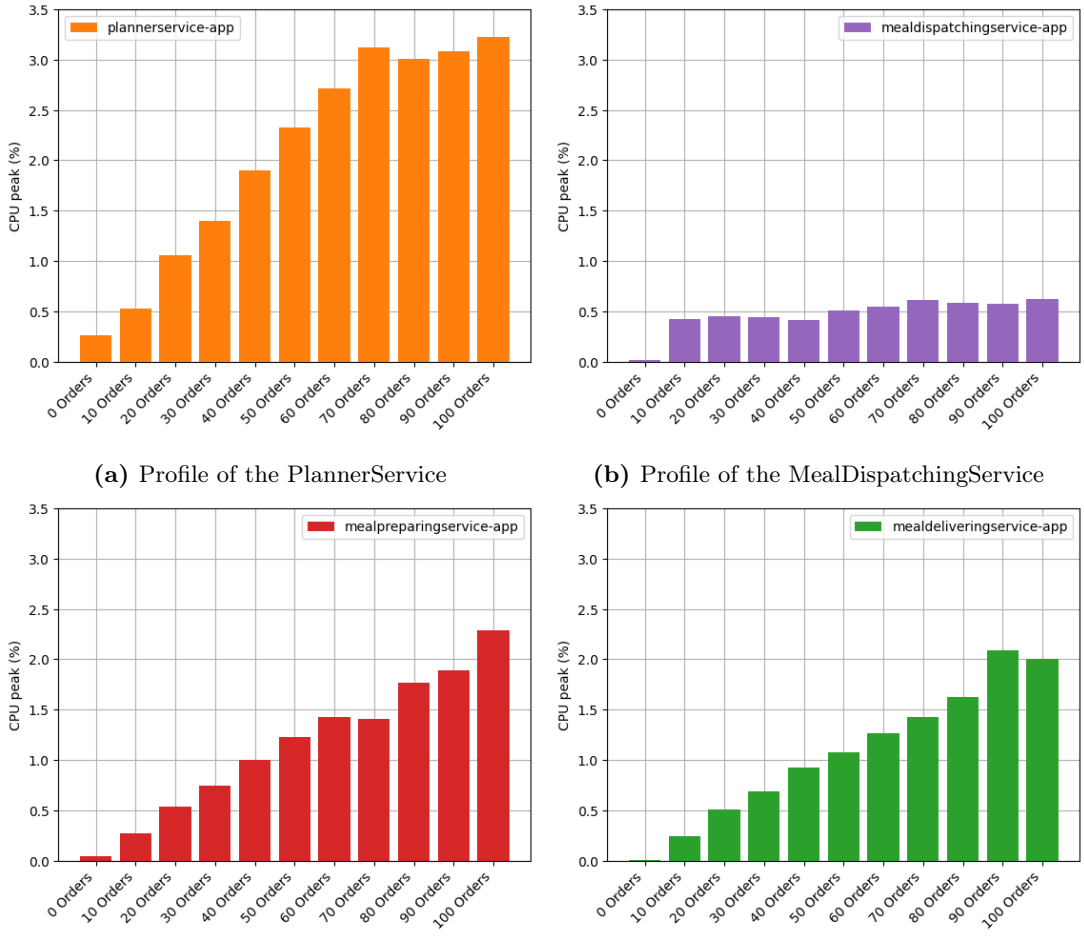
CPU Usage Profile

The microservice CPU usage results of the MDS profile are shown in Figure 5.9.

For each microservice, the influence of the increasing load can immediately be seen. The PlannerService, MealPreparingService, and MealDeliveringService all grow linearly in CPU usage with the load. The MealDispatchingService in Figure 5.9b shows that the peak CPU usage goes to 0.5% as soon as there are meal orders to process, but further growth is very minimal with increasing load.

The peak CPU usage of each microservice stays under 3.5% mainly because each microservice in the application has very minimal functionality and does not employ very CPU-intensive structures or algorithms.

The PlannerService does plateau after 70 orders, as shown in Figure 5.9a. Examining the exact reason for the characteristics of the MDS falls outside the scope of this work and will not be discussed, although this does bring up that this type of profiling can also help gain insight into how an application works. It can point out unexpected behavior and even pinpoint areas of improvement in an application. If a profile shows a lot of unexpected behavior that can be attributed to noise, it could be an indication that more data needs to be gathered to form a reliable profile.



(a) Profile of the PlannerService

(b) Profile of the MealDispatchingService

(c) Profile of the MealPreparingService

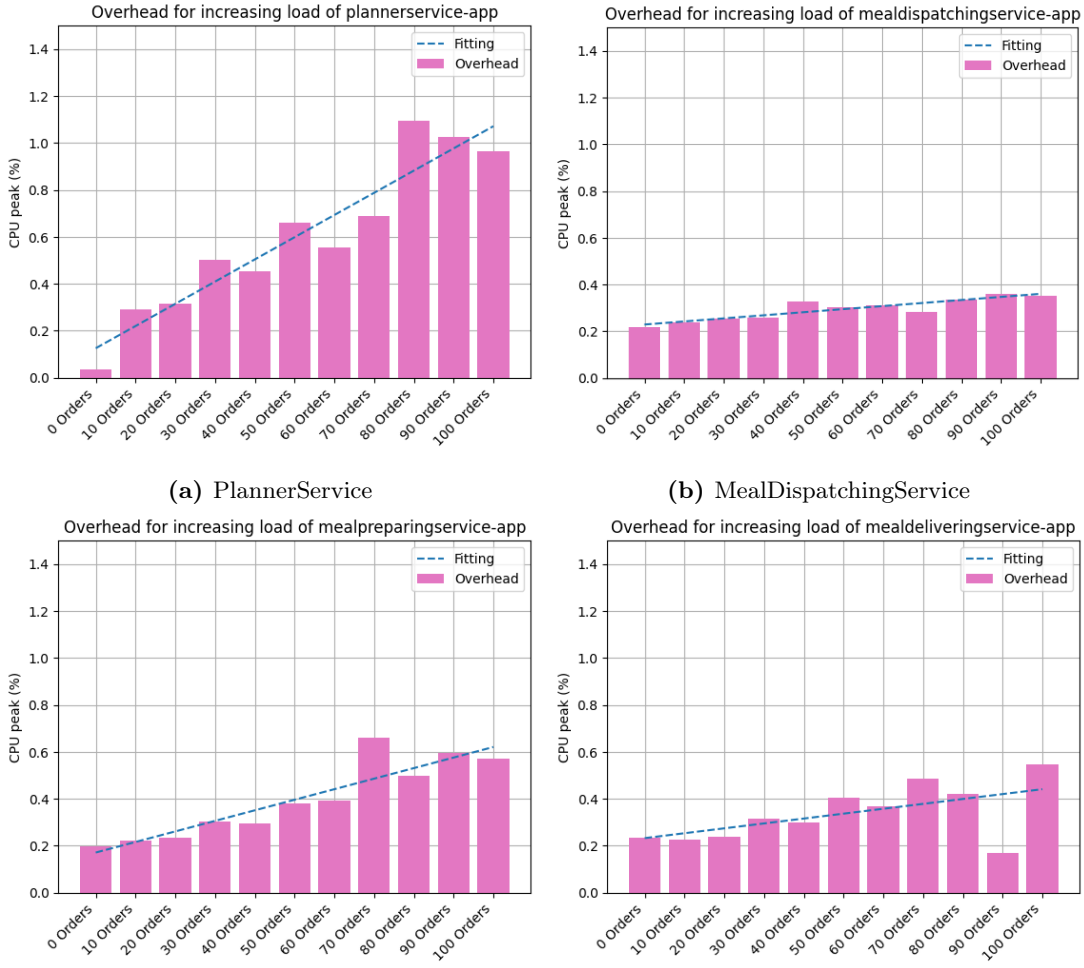
(d) Profile of the MealDeliveringService

Figure 5.9: The CPU usage profile for each microservice in the MDS. The peak CPU usage is shown against increasing request load

Overhead

The overhead fitting of each microservice in the MDS case study is shown in Figure 5.10.

First, the overhead ranges from around 0.2% to less than 1.2%. It is also shown that the overhead per microservice grows to different degrees per microservice with the load, and is therefore at least to some extent dependent on the microservice. A performance prediction model should thus also consider this aspect in its prediction. The variability in the overhead is quite high over different loads. This is especially the case for the PlannerService overhead, shown in Figure 5.10a, where the overhead for 50 orders is much higher than that of 60 orders. Two causes can be identified. 1) Since the CPU usage is so low, and the difference between overhead with 0 orders and 100 orders is only 1%, noise or



(a) PlannerService

(b) MealDispatchingService

(c) MealPreparingService

(d) MealDeliveringService

Figure 5.10: The CPU overhead profile for each microservice in the MDS. The overhead is shown against increasing request load.

small events may have a large influence on the measurements, and 2) the tool that gathers the node CPU usage, Node Exporter, and the tool that gathers the microservice CPU usage, cAdvisor, differ as discussed in Section 5.1.3. This may also be the cause of the dip in overhead in Figure 5.10d, for 90 orders, although further investigation is necessary to conclude this definitively.

The results of the overhead offset experiment and the overhead slope of each microservice in the context of the MDS are shown in Table 5.1. Further details of the idle overhead experiments in the MDS context, to acquire b, are shown in Appendix C.1.

We should note that while the overhead offset of each microservice is fairly similar, the slope differs quite a lot between microservices and appears to be microservice-dependent.

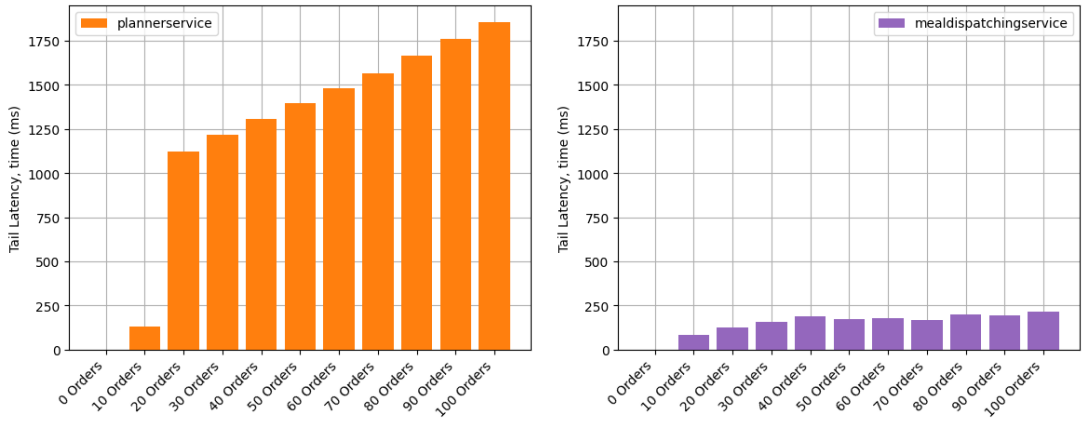
In the performance prediction, it is therefore likely necessary to include some sum of the slopes of the overhead of microservices if they are deployed on the same node, while the offset should stay as some constant value that is similar to the average offset of the microservices involved.

Table 5.1: Linear fitting of the overhead for each microservice in the MDS, based on increasing load. a represents the slope and b the offset of the linear fitting, which has the form $ax+b$.

Service	a	b
PlannerService	0.017	0.217
MealDispatchingService	0.001	0.249
MealPreparingService	0.004	0.287
MealDeliveringService	0.002	0.250

Traces and Latency Metrics

We show an example of the profile of a system workflow through **LR-1**. It includes both a span in the PlannerService as well as in the MealDispatchingService. Therefore both are included in the application profile. The results are shown in Figure 5.11.



(a) The latency within the PlannerService.

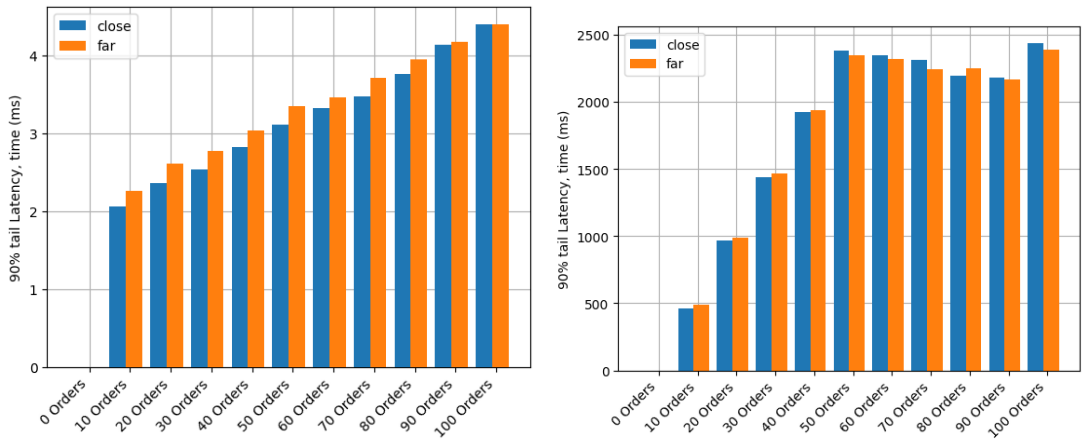
(b) The latency within the MealDispatchingService.

Figure 5.11: The 90th tail latency of the spans within microservices involved in **LR-1**.

Based on this profile, we can determine that the PlannerService has the highest impact on latency in this system workflow, overall its latencies are an order of magnitude higher than the MealDispatchingService.

Communication

We show the local and remote communication latencies for both **LR-1** and **LR-7** in the MDS context. The results are shown in Figure 5.12. Note that the y-axis scale for both graphs is different.



(a) 90th percentile tail latency of the ScheduleUpdateRequest message for increasing load on the MDS. **LR-1**

(b) 90th percentile tail latency of the MealPreparationRequest message for increasing load on the MDS. **LR-7**

Figure 5.12: Communication profile of messages for **LR-1** and **LR-7**. Both for local and remote communication.

The general trend for all messages is that the tail latency of the communication goes up with the load. This is likely due to increased congestion in the send and receive of the messages with increased load, or is caused by a higher CPU load because of increased communication. Figure 5.12a shows expected behavior, where the latency of the ScheduleUpdateRequest varies from just above 2 ms to just above 4 ms from 10 to 100 simultaneous order requests. Additionally, the remote latencies are slightly higher than the local latencies. This makes sense, since the communication between different nodes is likely to take longer than the communication within a node, due to the extra communication overhead.

With these profiling results and Figure 5.11a, we can also show that **LR-1** is mainly determined by the latency of the span in the PlannerService, as the communication has a latency of at most 5 ms, while the span takes at least 200 ms.

Figure 5.12b shows a growth in communication latency for the MealPreparationRequest message. The remote messages do not necessarily take longer than the local messages in these results, which hints that the distance between the microservices does not play a big role in the unexpected growth in communication latency, but rather the computational

time of the sender and receiver encapsulates most of the communication time. Further research is necessary to definitively conclude this.

Iterative Profiling and Profiling Time

Through experimentation, we have acquired a feeling of how many iterations were necessary to run the MDS to minimize noise in the profiling results. Ultimately, we came to 50 iterations for the MDS profile. Each iteration takes roughly 30 minutes. With 50 iterations, the variability in results showed to be fairly minimal. We estimate that it took roughly 25 hours to profile the whole application. We further discuss profiling time in Section 8.3.1.

6

Performance Prediction

In this chapter, we discuss performance prediction in the context of hardware dimensioning. We describe a performance prediction method through two performance prediction models, for which the profiling method of Chapter 5 forms the input. The method leverages the idea that a performance prediction for compositions of microservices can be made based on the individually profiled microservices.

The performance prediction method follows the structure of the profile, where it makes a distinction between CPU usage requirements and latency requirements. For both types of requirements, the goal of the prediction is to foretell whether requirements are violated for a given deployment configuration of microservices over nodes and a given load.

To validate the proposed approach, the models are validated in the MDS context with five distinct deployment configurations.

6.1 CPU Usage Prediction Model

We first present the CPU utilization prediction model, which aims to predict the CPU utilization of a node given the microservices that are deployed on it. The prediction model consists of two components, a microservice component and an overhead component. These are then added to predict the peak node CPU usage as shown in Equation (6.1). Here, $R_n(x)$ is the expected peak CPU utilization on a node, n, given the load, x, $A_n(x)$ represents the microservice CPU usage component, and $O_n(x)$ represents the overhead component. All variable descriptions are displayed in Table 6.1. Note that the load may be defined in different ways, as described in Section 5.1.2, and a mapping needs to be made from increasing load to x in the model.

$$R_n(x) = A_n(x) + O_n(x) \tag{6.1}$$

This model does not take the effect of microservice interference on a single node into account, which is further discussed in Section 8.3.2.

Table 6.1: Overview of the variables in the CPU usage model

Variable	Description
$R_n(x)$	The expected peak CPU usage on a node, n, given the load, x
$A_n(x)$	Microservice CPU usage on a node, n, given the load, x
$O_n(x)$	Overhead CPU usage on a node, n, given the load, x
M_n	The set of microservices on a node, n
p_{i_x}	The peak CPU usage of microservice, i, given the load, x
b_n	The set of overhead offsets for all microservices on M_n
a_i	The overhead slope of microservice i

6.1.1 Microservice resource model

The microservice component of the CPU usage prediction model includes the CPU usage profile of each microservice that runs on a node in isolation, as described in Section 5.3.1.1. This is the basis of the prediction model. The core assumption here is that the resource usage of an application of a node can be indicated by the sum of the resource usage of microservices on that node. This component of the prediction model is shown in Equation 6.2). p_{i_x} represents the peak CPU usage of microservice i with a given load x, as described in Section 5.3.1.1. The set M_n represents all microservices that are running on node n.

$$A_n(x) = \sum_{i \in M_n} p_{i_x} \tag{6.2}$$

The model uses the peak CPU usage to predict the overall application performance because the prediction is a part of hardware dimensioning, where the goal is not to be as accurate as possible, but rather to arrive at a conservative estimate. By taking the peak CPU usage of a microservice for a given load, the expectation is that the microservice will generally not use more CPU usage than that peak and the prediction is therefore conservative.

6.1.2 Overhead model

The overhead component of the CPU usage prediction model represents a missing aspect in the sum of the CPU usage of isolated microservices. It encapsulates the CPU usage created by communication overhead, and the infrastructure services. In Section 5.10, we describe a linear fitting that makes up the overhead of an application. The fitting shown in that section is an example of a fitting that can be used in this model, but the overhead model should apply the trend found in the profiling stage. The overhead model is shown in Equation (6.3), where M_n is the set of microservices that run on node n, b_n is the set of overhead offsets of all microservices in M_n, a_i is the overhead slope of microservice i, and x represents the given load.

$$O_n(x) = max(b_n) + \sum_{i \in M_n} a_i \cdot x \tag{6.3}$$

To keep the prediction conservative, we define the expected overhead offset to be the highest overhead offset of any microservice in M_n, as we assume, from the profile, that the offsets of the varying microservices are similar. If this is not the case, we could take the average offset, or minimum offset, depending on the purpose of the prediction model as well. We do not sum the offset overhead of each microservice, as it largely includes the CPU usage of infrastructure and monitoring services, which we have experimentally determined to remain fairly constant regardless of how many microservices are running on a node.

Note that the overhead prediction always includes some monitoring services that may not be running on the target system for which the prediction is made. We do not see this as a problem, as the expectation is that the monitoring services only form a small factor in the overhead. Additionally, it would only cause a more conservative performance prediction, which is preferable in the context of hardware dimensioning. However, the degree to which monitoring services impact CPU usage is system and application-dependent, which should be evaluated or tested before using the CPU usage prediction model.

To calculate the expected overhead slope of concurrently running microservices, the overhead slope of each microservice, running in isolation on a node, is added together. The overhead on a node is then predicted by multiplying the slope with the load, e.g. the number of orders in the MDS, together with the overhead offset. The slopes are added in such a way, since we expect the overhead to grow with the number of services that run on a node according to their individual slopes.

6.1.3 MDS Performance Prediction

In this section, we show how to apply the CPU usage prediction model. This example is a part of the MDS case study introduced in Chapter 4. The MDS case study includes four microservices and two similar nodes on which the validation will take place. We use the EvenSplit 1 deployment configuration, as shown in Figure 6.1.

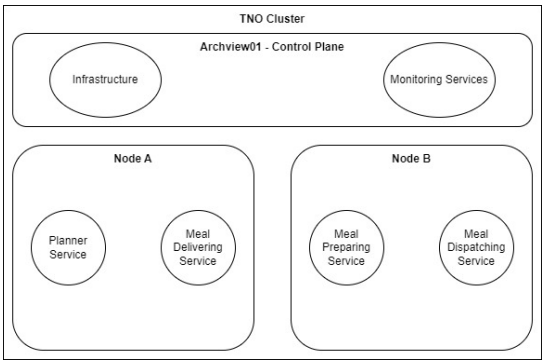
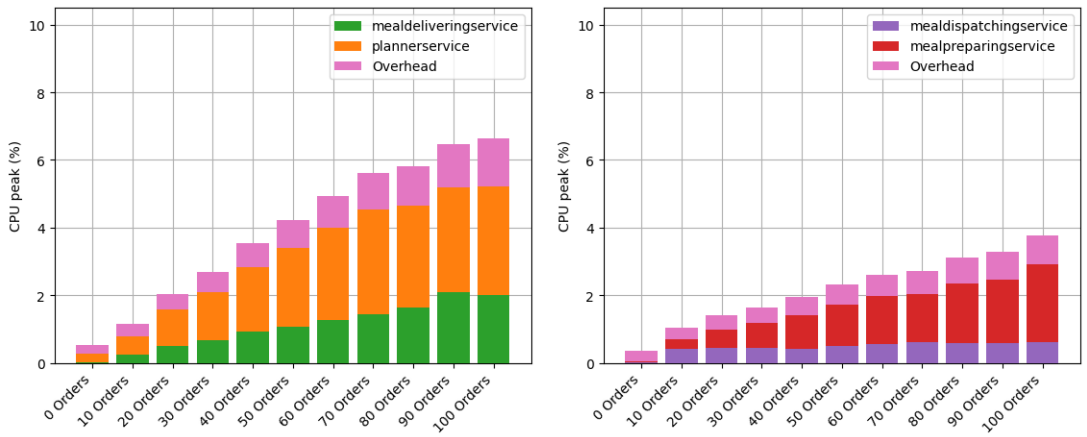


Figure 6.1: An example of a deployment configuration of the MDS on the TNO cluster: EvenSplit 1.

The predictions of the complete CPU usage model of Equation(6.1), in the context of the MDS for deployment configuration EvenSplit 1, are shown in Figure 6.2.



(a) The prediction of CPU usage for node A **(b)** The prediction of CPU usage for node B

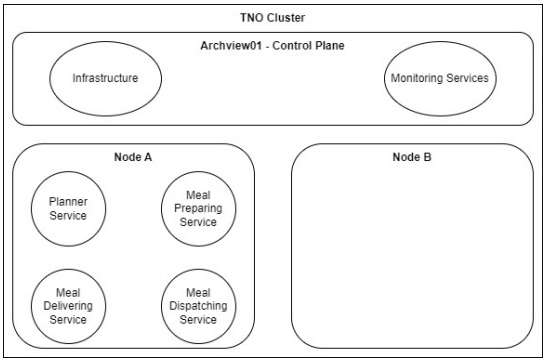
Figure 6.2: Prediction of the CPU usage model for the EvenSplit 1 deployment configuration.

The results indicate the expected CPU utilization of the nodes, given the microservices that run on them for varying loads. The prediction follows the trends that are shown in the application profile, as it is based on it. In this example, the model predicts that the CPU usage of each node stays under 7% for any load of up to 100 simultaneous requests.

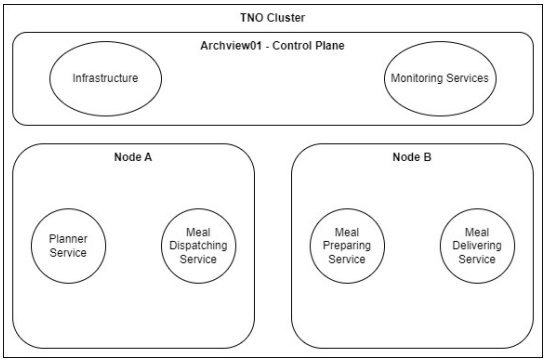
Requirement **RCR-1** is therefore satisfied for at least up to 100 simultaneous requests with this given deployment setup, as the CPU usage stays well under 60% for each node.

6.1.4 Model Validation

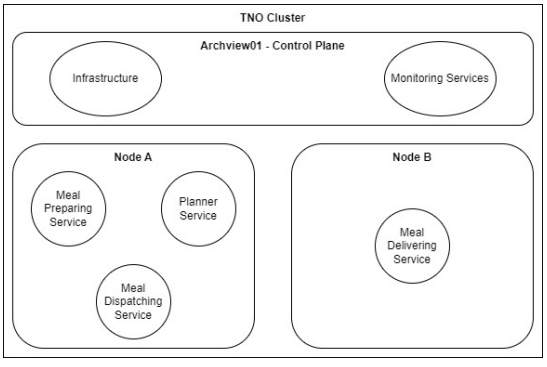
Now that we have shown how to use the CPU usage prediction model, we will validate whether the model predicts the CPU usage of a node accurately in the MDS context. It is validated by computing its results based on the EvenSplit 1 deployment configuration and four additional deployment configurations. These deployment configurations are shown in Figure 6.3. The predicted CPU usage is compared to the measured CPU usage on Transact01.



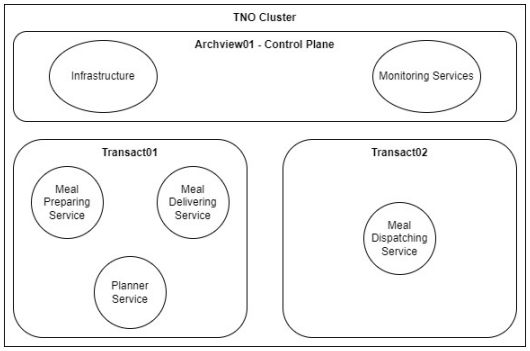
(a) The deployment configuration of all MDS microservices on one node: OneNode



(b) A deployment configuration where each node deploys two MDS microservices, where the PlannerService and MealDispatchingService are placed together: EvenSplit 2



(c) The deployment configuration where the MealDeliveringService runs in isolation and the other three MDS microservices are placed on the other node: TripleSolo 1



(d) The deployment configuration where the MealDispatchingService runs in isolation and the other three MDS microservices are placed on the other node: TripleSolo 2

Figure 6.3: The four additional deployment configurations that will be used in the validation.

We feel that these five deployment configurations showcase the variability in the possible deployment configurations, so we can make a comparison between the various microservice placement options of the MDS, given the two nodes and four microservices. These configurations deploy microservices in two distinct ways in isolation and trios, in two pairs of two, and include the scenario where all microservices are placed together. The different deployment configurations show that the latency prediction model applies to, and holds for, any combination of microservices in the MDS context.

Since the MDS application profile is an aggregation of 50 iterations of a profile, as discussed in Section 5.4, the validation is also run 50 times, to keep the comparison as straightforward as possible.

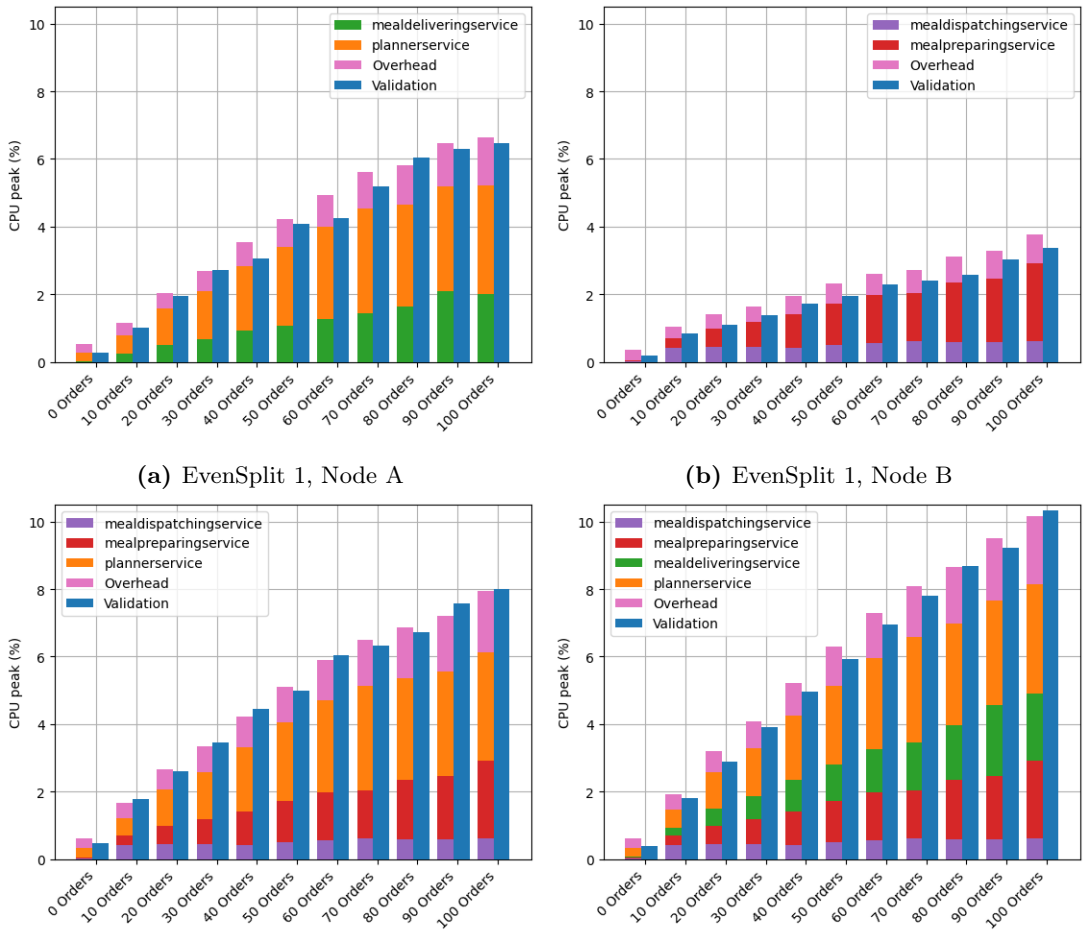
Based on feedback from practitioners, in the context of hardware dimensioning for the case study of the MDS, we deem the models to be sufficiently accurate if the predictions only differ by at most 20% to the validation. The prediction preferably has higher CPU usage values than the validation, as such a prediction prevents under-dimensioning, which is less desirable than over-dimensioning in the context of hardware dimensioning, as discussed in Section 2.4.

Note that nodes Transact01 and Transact02 have slightly different specifications. Since we have only done profiling on Transact01, we omit validation on Transact02 in these experiments. This means that we run each deployment configuration twice on the TNO cluster. For example, we deploy EvenSplit 1 and the mirrored version of it, where the microservices on Transact01 and Transact02 are swapped, so that we can validate both the prediction for node A, as well as node B. Then, only the measurements on Transact01 are used in the validation. We have an additional experiment on prediction and validation on Transact02 in Appendix C.3.1.

The validation results for both nodes of EvenSplit 1, node B of TripleSolo 1, and the node of OneNode are shown in Figure 6.4. The rest of the results can be found in Appendix B.1.

The results show that all predictions fall within 20% of the validation for any deployment configurations and all amounts of load. Additionally, the biggest difference is shown when the prediction is higher than the validation, like in Figure 6.4a, which is sufficient for hardware dimensioning, since our model aims to be conservative in its prediction. The trend of a linear increase in peak CPU usage for increasing load is reflected in the validation for any deployment configuration as well.

Just as predicted, the CPU usage of each node falls when the MDS microservices are more evenly distributed over the nodes. The trend in the prediction is followed by the



(a) EvenSplit 1, Node A

(b) EvenSplit 1, Node B

(c) TripleSolo 1, Node A

(d) OneNode, Node A

Figure 6.4: The prediction against the validation for both nodes of EvenSplit 1, node B of TripleSolo 2, and the OneNode deployment configuration. Validation is done on Transact01

validation. The assumption that we can add the CPU usage of isolated microservices to retrieve the CPU usage of a node seems to be correct in this context.

Additionally, while some predictions predict a lower CPU usage, as shown in Figure 6.4c for 40 orders, the model undershoots at most by 6.80% compared to the validation, and most predictions overshoot their prediction, as is more desirable.

Thus, overall the model predicts the behavior of the application on the system well, with at most 16.25% overshooting the validation measurements, and only 6.80% undershooting.

6.2 Latency Prediction Model

The second prediction model aims to predict the latency of system workflows based on the load on an application and the deployment configuration of microservices over nodes. The system workflows may be based on the requirements of an application. Chapter 5 discusses the method of extracting these subtraces from an application, and shows the profiling method that forms the basis for this model.

The model is shown in Equation (6.4), and the variable descriptions are displayed in Table 6.2. It consists of the combination of the microservice spans and communication in the system workflows. Here, $L_r(x)$ is the expected yth percentile tail latency for a given system workflow r and load x, $S_r(x)$ is the latency component of the spans within microservices, and $C_r(x)$ is the communication component.

$$L_r(x) = S_r(x) + C_r(x) \tag{6.4}$$

It leverages the idea that we can use the addition of latencies of spans and messages in an application to predict the total end-to-end latency of a system workflow.

Note that we assume that each microservice has one span from beginning to end and does not call any child services. If a microservice calls another microservice during its execution and waits to finish its execution until it gets a response from its child, the time of both the child and parent would be added up. This has to be taken into account if it is known that that occurs in an application, and could also be handled through the profile, by splitting the span of the parent microservice into two spans. The model can then remain the same.

Table 6.2: Overview of the variables in the latency model

Variable	Description
$L_r(x)$	The yth percentile tail latency for a given system workflow, r, and the load, x
$S_r(x)$	Microservice latency of system workflow, r, given the load, x
$C_r(x)$	Communication latency of system workflow, r, given the load, x
M_r	The set of microservices within system workflow r
s_i	The yth percentile tail latency of the span in microservice, i
I_r	The set of messages included in system workflow r
$p_m(x)$	The communication latency of message m in I_r, given the load, x
l_m	The local communication latency as the yth percentile tail latency
d_m	The remote communication latency as the yth percentile tail latency

6.2.1 Spans in Microservices

The first component of the model considers the spans in the microservices involved in a system workflow. These spans are heavily application-dependent, and may include one to all microservices, depending on the system workflow. Based on the application profile, we will use the yth percentile tail latency per span in the performance prediction.

In order to predict the total latency of a system workflow through multiple microservices, the measured tail latency of the span in each individual microservice, that is included in the system workflow, is added up. This component of the model is thus defined as in Equation (6.5). Here, $S_r(x)$ is the latency component of the spans within microservices, based on the system workflow, r, and load x, M_r is the set of microservices within the system workflow r, and s_i is the yth percentile tail latency of the span in microservice i.

$$S_r(x) = \sum_{i \in M_r} s_i \tag{6.5}$$

Note that there is no distinction yet in this model between deployment configurations. Regardless of the microservice placement, the latencies of the spans of the microservices are added up based on the measurements for isolated microservices. This means that the effect of contention of microservices on trace latencies, as in the CPU model, is not considered in this model, as discussed in Section 8.3.2. We assume this still works for services that use a low percentage of system resources. An additional experiment on the effect of interference of microservices on latencies in the MDS context to support this is discussed in Appendix C.2.

6.2.2 Communication

Traces may not only consist of spans in microservices, but also of the communication between the microservices. Therefore we need to add the latencies of these messages, which are acquired through the application profile, to the model. This also introduces the dependency of the latencies on the deployment configuration, as a distinction is made between local and remote communication. By local communication, we mean messages that are sent between microservices that are running on the same node. By remote communication, we mean messages that are sent between microservices on separate nodes. We do not make a distinction between remote communication between nodes that are placed close to each other, and remote communication between nodes that are far apart in terms of physical space between them. Additionally, the model is built under the assumption that all nodes are directly connected, and do not include hops over other nodes.

We assume that remote communication generally has higher percentile latency results than local communication for a general application, as it includes a greater travel time. However, if this is not the case, it will show up in the application profile and the communication model will still be applicable. The communication model is shown in Equation (6.6) and Equation (6.7). Here, $C_r(x)$ is the sum of message latencies within system workflow r for load x, I_r is the set of messages included in system workflow r, and $p_m(x)$ is the yth percentile tail latency of message m. $p_m(x)$ either is defined by the local communication latency, $l_m(x)$ or the remote communication latency, $d_m(x)$.

$$C_r(x) = \sum_{m \in I_r} p_m(x) \tag{6.6}$$

$$p_m(x) = \begin{cases} l_m(x), & \text{if m is sent over local communication.} \\ d_m(x), & \text{if m is sent over remote communication.} \end{cases} \tag{6.7}$$

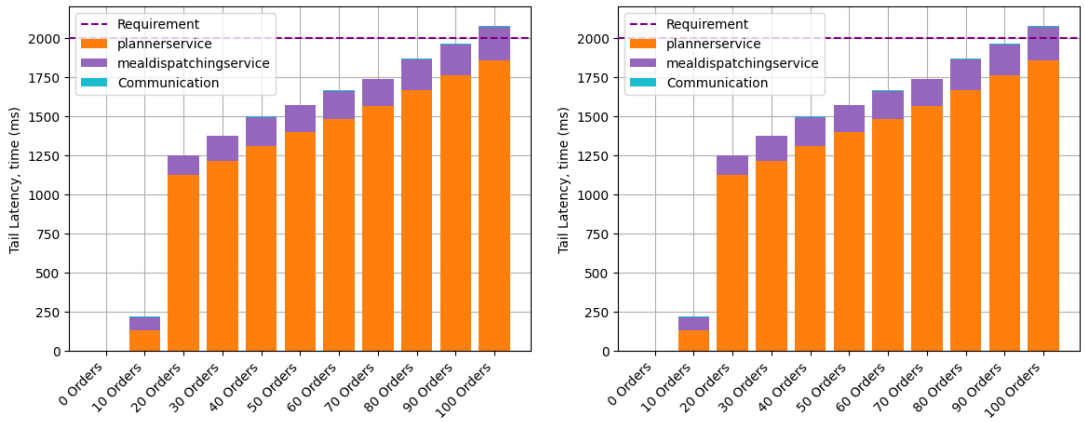
6.2.3 MDS Performance Prediction

We now show how to use the latency prediction model within the MDS. We predict the 90th percentile tail latency of the system workflow that is based on **LR-1**. The latency predictions for a system workflow are split up into a prediction for local communication and remote communication. Since the system workflow of **LR-1** includes the PlannerService, MealDispatchingService, and a message between them, we can use EvenSplit 1 and EvenSplit 2 to represent both a trace flow through two nodes, and within one respectively. We use the 90th percentile tail latency as explained in Section 5.5.3 on the MDS profile.

The 90th percentile tail latency predictions are shown in Figure 6.5 for both local and remote communication.

The results give insight into the most significant component in the latency of a system workflow. For example, **LR-1** is mostly defined by the PlannerService span, and the communication latency is only a small part of the prediction. A consequence is that in this example, the communication distance does not influence the prediction, which also means that there is little difference in latency prediction for different deployment configurations: The predictions for EvenSplit 1 and EvenSplit 2 differ at most by 0.24 ms, which means the local communication differs at most by 0.02% for this latency requirement.

Additionally, this prediction indicates that the latencies of 10 simultaneous requests generally lay much lower than those of 20 and up. Any exact reason for trend-breaking behavior in results may be further investigated, although that is very application-specific, and falls outside the scope of this thesis.



(a) EvenSplit 2, local

(b) EvenSplit 1, remote

Figure 6.5: The 90th percentile tail latency for a workflow based on **LR-1** for local and remote communication, EvenSplit 1 and EvenSplit 2.

Note also that the prediction shows that the requirement is violated with 100 simultaneous order requests, indicated by the dashed line.

6.2.4 Model Validation

Now that we have shown the latency prediction model and we have given an example of how to use it, we will validate whether it predicts latencies with some accuracy in the MDS context. We validate the predicted latencies of the model against the measured latencies of the application running on the TNO cluster. The deployment configurations EvenSplit 1 and EvenSplit 2 are used for the validation, where we compare the predictions of the latency model to the measured latency of request traces through the MDS application. We use these two deployment configurations, since these represent both a system workflow within one node, as well as a system workflow spanning two nodes for the latency requirements that concern two microservices: **LR-1**, **LR-3**, and **LR-7**. Their specific workflows are discussed in Section 4.2.

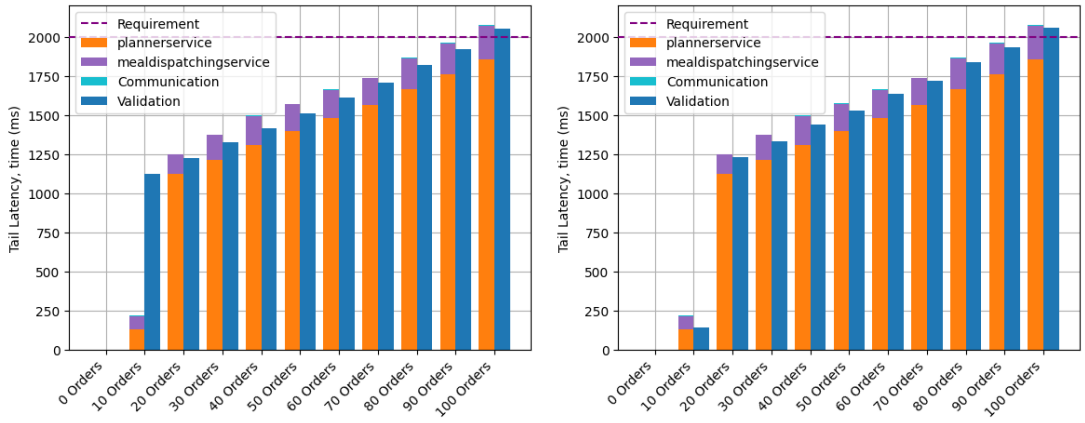
Note that the remote latency predictions are validated partially on Transact02, as we did not have two identical nodes at our disposal. We do provide a short experiment in Appendix C.3.2, to show that the validation on the two nodes does not differ much, and therefore we can still claim these validation results are valid.

The results of **LR-6** are shown in Appendix B.2. The prediction only differs by at most 2% for the EvenSplit 1. This latency requirement is predicted as the most accurate. Just as the prediction, the validation shows that the latency requirement is violated with

50 simultaneous orders and up, as expected through the artificial latency described in Section 4.1.2.

LR-1

The validation of the prediction for **LR-1** is shown in Figure 6.6. It is split up into validation for the PlannerService and MealDispatchingService running concurrently on a node, and them running on separate nodes according to EvenSplit 1 and EvenSplit 2.



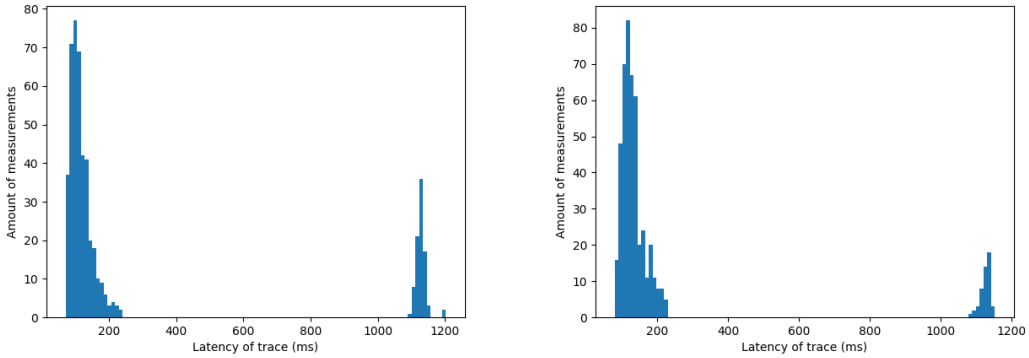
(a) EvenSplit 2, local

(b) EvenSplit 1, remote

Figure 6.6: Prediction of 90th percentile tail latency of the system workflow based on **LR-1** compared to its validation for varying load.

Overall, the trend of the prediction is shown in the validation results as well. The 90-percentile tail latency grows linearly with the load. Furthermore, the prediction generally is more or less than 5% of the validation, which means that we feel that the prediction is sufficiently accurate, and that the addition assumption of our model holds. The prediction is higher than the validation in any case, which is desirable. This is likely due to the sum of percentiles, adding up the 90th percentile latencies for each component in the trace, against the 90th percentile of all traces. The difference between EvenSplit 1 and EvenSplit 2 is minimal, indicating that communication only plays a small role in this latency requirement. The requirement is violated for both cases when 100 simultaneous orders are issued.

There is one outlier. The 90-percentile tail latency for a load of 10 meal orders differs by 1000 ms between the two different validation setups. To explain this difference, a deep dive is necessary into the tail latency computation of this particular trace. The distribution of latencies for the workflow request for both validation setups is shown in Figure 6.7.



(a) EvenSplit 2, local

(b) EvenSplit 1, remote

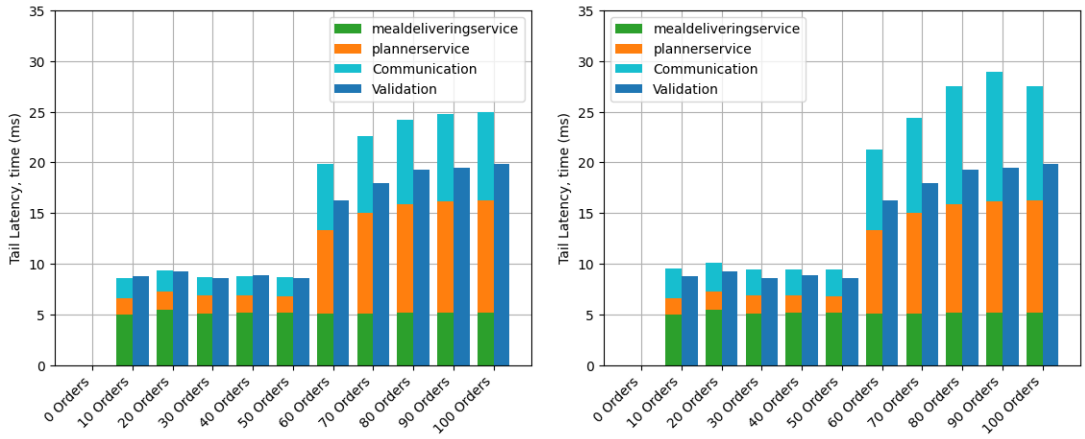
Figure 6.7: Deep dive into the distribution of latencies of 10 meal orders for the full workflow request that is based on **LR-1** of the validation for EvenSplit 1 and EvenSplit 2.

The latencies of the trace are divided into two distinct peaks, one peak is under 200 ms, and the other is above 1000 ms. The different results per the validation setups for 10 meal orders may be explained by these two peaks. It so happens to be that the 90th percentile tail latency of the validation in Figure 6.6a just falls within the second peak, and the tail latency of Figure 6.6b falls within the first peak. The 90th percentile tail latency of the PlannerService in the profile just falls within the first peak as well, so the prediction indicates a latency of under 250ms. The exact explanation of why these two peaks occur has to be found in the implementation MDS application and falls outside of the scope of this thesis. However, this example displays how complex a performance prediction can be in this context, as it can be influenced by many factors.

A wrong prediction like this could have been indicated beforehand by noting that the model predicts that the linear trend in latency growth from 20 to 100 simultaneous order requests does not hold for 10 requests. It could then be looked further into why it is the case in this specific application, which may result in more accurate predictions, through an incorporation of the cause of these peaks in the prediction model in some way. It may therefore be beneficial to look into unexpected behavior in the prediction results, to iron out potential wrong predictions, although this process could be very application-specific.

LR-3

The validation of the prediction for **LR-3** is shown in Figure 6.8. It is split up into validation for the MealDeliveringService and PlannerService on one node, as in EvenSplit 1, and on separate nodes, as in EvenSplit 2.



(a) EvenSplit 1, local **(b)** EvenSplit 2, remote

Figure 6.8: Prediction of 90th percentile tail latency of the system workflow based on **LR-3** compared to its validation for varying load.

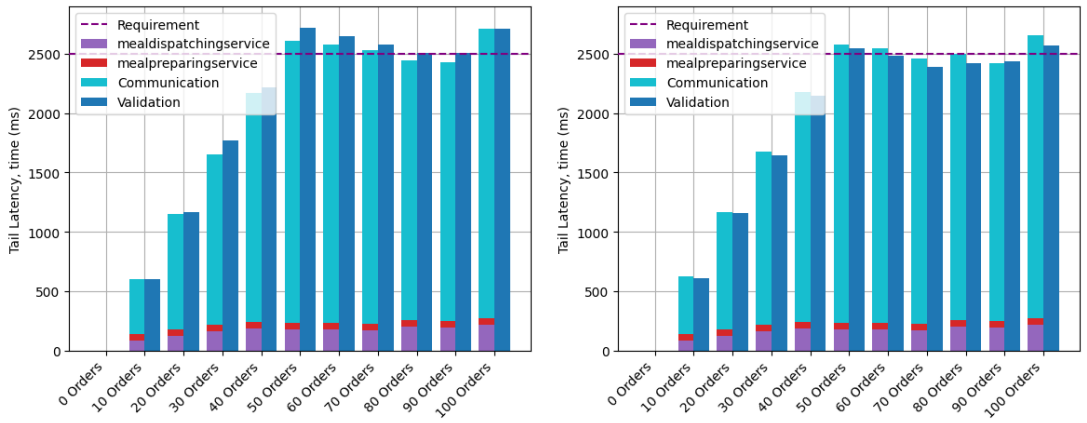
The prediction results show that the results fall within the 20% prediction threshold for 50 simultaneous orders and under, as the maximum difference is 18.34%. However, with a higher load, the prediction overshoots the measured latencies up to 34%. In this example, the difference could be caused by the process of summing up tail latencies. In the prediction, we sum up the latency outliers of each separate component, while in the validation, some longer latencies per component may be canceled out by the other spans and messages in the same request trace. Note that this is not the case in the other latency requirement predictions. We therefore believe that this latency requirement is more sensitive to the conservative nature of adding percentile tail latencies, and more sensitive to noise, because its values are smaller.

Although the prediction is not always accurate in this instance, in both the prediction and the validation, the latency requirement, **LR-3**, is always met, since it is set at 100 ms. It is not shown in the figures since it would not fit on the graph. If the goal of the prediction is to determine whether that is the case, this prediction still is accurate.

It may be possible to come to a more accurate prediction by analyzing why the latencies grow from 60 simultaneous orders onwards. If we know what causes it, we could implement the cause into the model to improve upon it. This is heavily application-dependent.

LR-7

The validation of the prediction for **LR-7** is shown in Figure 6.9. It is split up into validation for the MealDispatchingService and MealPreparingService for EvenSplit 1, local, and EvenSplit 2, remote.



(a) EvenSplit 1, local

(b) EvenSplit 2, remote

Figure 6.9: Prediction of 90-percentile tail latency of trace based on **LR-7** compared to its validation for varying load.

The prediction model differs by at most 6.68% from both validation setups. Therefore, we deem the prediction model to be sufficiently accurate for this latency requirement.

The results of **LR-7** show a prominent role of the communication aspect in the model. They indicate that the communication takes up a significant amount of the total latency of that particular trace. In this case, it may be worth looking into optimizing this particular message between the MealDispatchingService and MealPreparingService to improve performance, in addition to loosening the requirement if that is feasible. A further analysis of the MDS is needed as to why the communication takes up a disproportionate amount of time of the full trace, which falls outside the scope of this thesis.

With both remote and local communication, the prediction falls within the 20% accuracy threshold, and the validation follows the trend in terms of varying load of the prediction. In Figure 6.9b, it is shown that the model predicts a higher latency than is measured for all loads. However, Figure 6.9a shows that the model predicts below the measured latencies for local communication. The reason for the difference is in the communication latencies, since that is the varying factor in these two setups. It is also shown that the latencies for the local deployment configuration appear to be higher than those of the remote deployment configuration. The communication for this requirement shows that messages can take up

to 2 seconds to complete. It could therefore be that the difference in local and remote communication is negligible for this system workflow, and the communication overhead is mainly dictated by some error in the message processing of this component of the MDS application. The prediction model takes this into account through the communication profile, and is therefore still accurate.

7

Hardware Dimensioning

In this chapter, we combine the profiling and performance prediction methods in the context of our proposed hardware dimensioning approach, introduced in Section 1.1.1. We provide an overview of the necessary profiling steps, indicate the role of performance prediction, and explain how to leverage the proposed methods through manual design space exploration. An overview of the whole hardware dimensioning approach was previously shown in Figure 1.1.

7.1 A Step By Step Approach

In the context of hardware dimensioning, profiling forms the basis of the performance prediction, and performance prediction models give insight into how the performance differs for varying deployment configurations over a varying number of nodes. As shown in the results of both models, the resource utilization and system workflow latencies of an application may be influenced by the deployment configuration of microservices over nodes. Thus the performance prediction model can be leveraged as an evaluation in the process of design space exploration [49], to determine the number of nodes and deployment configuration to use in the target system.

Through manual design space exploration, we can pick the minimum number of compute nodes for a suitable deployment configuration as follows:

1. Define the performance requirements of an application that need to be satisfied. We deem a deployment configuration suitable if all relevant performance requirements are satisfied.

2. Profile an application, as discussed in Chapter 5.

- Define the necessary profiling deployment configurations, as described in Section 5.1.1
- Design a workload suite that creates application workload in some way, as explained in Section 5.1.2
- Set up data monitoring and collection infrastructure, as laid out in Section 5.1.3
- Define the profiling metrics, as shown in Section 5.2 and Section 5.3.
- Determine how many times a profile needs to run to cancel out potential noise in the results as described in Section 5.4.
- Run the profiling experiments.

3. Create the performance prediction models as defined by Chapter 6, which require an application profile and deployment configuration as input.
4. Pick a reasonable number of nodes for the target system.
5. Pick potentially suitable deployment configurations to try out. The exact method of picking an initial deployment configuration falls outside of the scope of this thesis, but a good starting point may be to minimize the communication distance within an application, as described in the related work for IntMA, or as in Nautilus [31, 34].
6. Find a suitable deployment configuration for the specified number of nodes where all performance requirements are satisfied, by evaluating each relevant deployment configuration through the proposed performance prediction models. If we can find a suitable configuration, we can downscale the number of nodes by one, if we cannot, we upscale.
7. Pick the lowest number of nodes for which there is a suitable deployment configuration.

7.1.1 The MDS Context

To illustrate how to apply the hardware dimensioning approach, we briefly discuss how we apply it in the context of the MDS. We start by defining how the performance requirements from Section 4.2 should be satisfied:

- A node should never use more than 60% of the available computational resources over 15 seconds.

- Latency requirements **LR-1**, **LR-6**, and **LR-7** are met under a load of 40 simultaneous meal requests.

Since **LR-3** is always met in our deployment configuration examples, we do not consider it here.

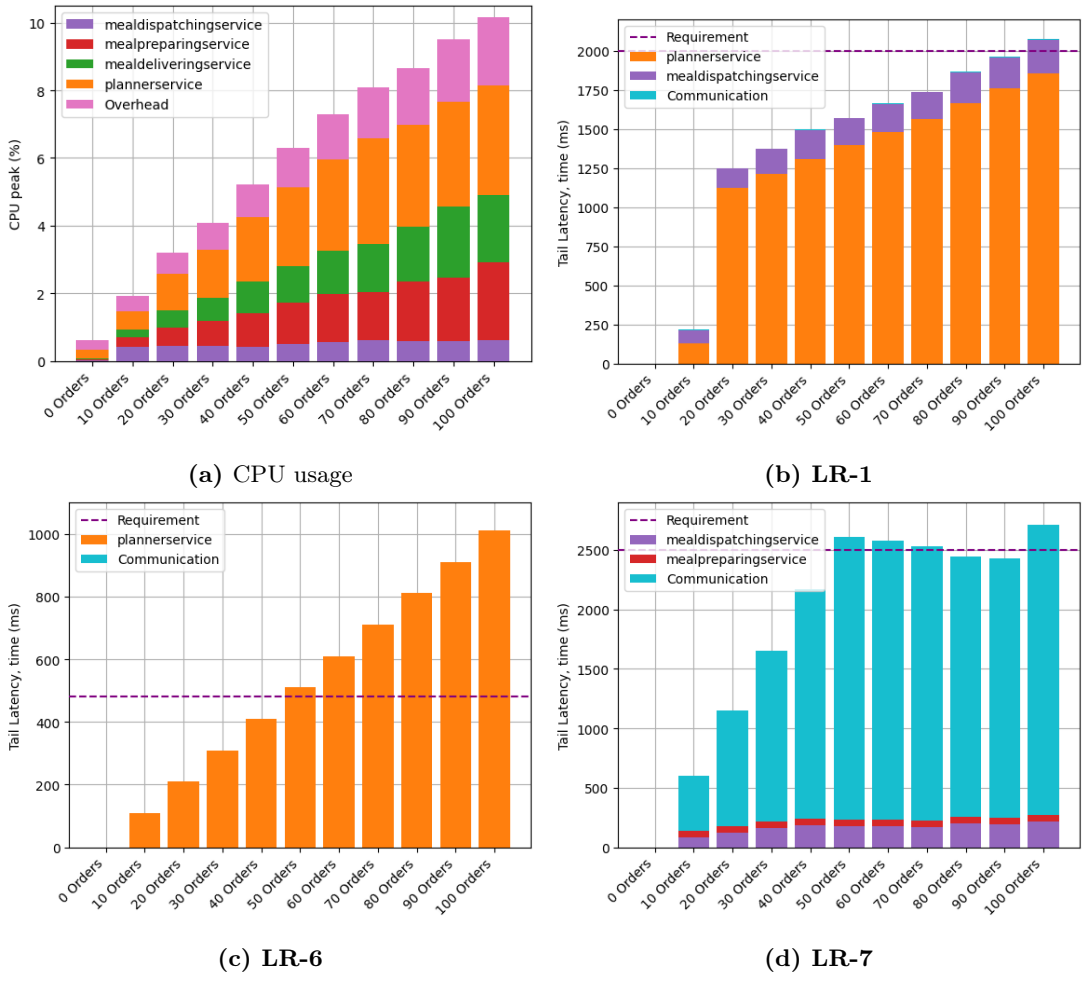
In Section 5.5, we have created the MDS profile, following the steps as described in the manual design space exploration steps. An example of the MDS performance prediction is shown in Section 6.1.3.

Now, we pick a reasonable number of nodes for the target system. Based on the application profile, we chose one node to test our deployment configurations on.

In our example, we pick the only deployment configuration that is possible on one node: OneNode, as all MDS microservices are placed on it. Through our MDS profile and the performance prediction models, we come to the results shown in Figure 7.1.

The results indicate that all requirements we have specified are met, and therefore we have found a suitable deployment configuration on one node. In the case that some requirements were not satisfied, we could upscale to two nodes, and try various deployment configurations in that context. As one node is the minimum amount of nodes we can use, that is the optimal number of compute nodes for the MDS.

Note that the definition of a suitable deployment configuration determines how many compute nodes are sufficient. For example, we can easily satisfy the requirement that a node should never use more than 60% of its available computational resources. However, if we change it to 5% of its available computational resources, we would come to an answer of two nodes, with deployment configuration EvenSplit 1, for example.



(a) CPU usage

(b) LR-1

(c) LR-6

(d) LR-7

Figure 7.1: Performance predictions results for hardware dimensioning on one node with deployment configuration OneNode.

8

Discussion

In this chapter, we discuss the successes and shortcomings of our work, by analyzing the results of the MDS case study for our profiling approach and performance prediction models as part of our hardware dimensioning methodology.

8.1 The Profiling Approach

We have constructed a profiling approach that includes the creation of profiling infrastructure and a definition of a suitable profile for performance prediction. We consider this approach to be a suitable profiling approach as it applies to any general microservice application, is time-efficient, and can be performed on a limited amount of hardware. We feel, therefore, that this profiling approach forms a suitable basis for performance prediction.

We have kept the profiling method general to any microservice application by considering metrics that apply to any application, or by providing a method of deriving application-specific metrics.

A practical aspect of the approach is that minimal hardware is required to create a profile. This is important, as not all hardware is available at the point of profiling, or even no hardware is available for the target system, and solely nodes are available that are as representative as possible to the target system. Therefore, we have limited the necessary hardware to one node that matches the target system as closely as possible, and one node that does not need to match it. Additional nodes may be used if an application does not fit on one node. Furthermore, our aim in the approach has been to keep the number of deployment configurations that are necessary to build a profile to a minimum, to avoid exponential growth with the number of microservices, and to minimize the profiling time. We further discuss profiling time in Section 8.3.1. In our approach, the number of

deployment configurations grows linearly with the number of microservices in an application, as described in Section 5.1.1. Based on feedback from practitioners, we feel that this should be reasonable in terms of the time and effort necessary to perform the profiling proposed in this thesis. Additionally, we show how to construct a profiling workload suite in Section 5.1.2 and a data gathering and monitoring framework for both traces and time-series metrics that can be used as a basis for any distributed microservice application in Section 5.1.3.

Another important aspect of both profiling and validation is the statistical handling of experiments. Where single experiments may show significant variance in CPU usage peaks and tail latencies, depending on the sampling frequency, averaging over many experiments can mediate the results and remove noise. For any application, a few experiments should be performed to identify the variation in the data and the needed amount of profiling iterations. Based on this, a trade-off needs to be made that is application-specific, where too few experiments result in noise negatively influencing the prediction accuracy, while too many experiments may get in the way of obtaining results in a reasonable time. As an example, in our case study, we ran each of our experiments 50 times. The number of experiments may drastically influence the quality of the final results, and should therefore be carefully considered.

8.2 The Performance Prediction Models

We present two performance prediction models that can be used to gain insight into the performance of an application before it is deployed on a target system. These models are built as an aid in hardware dimensioning for microservice-based CPSs in the sense that they indicate an estimate of how much hardware to assign to an application for performance requirements to be met. The two performance prediction models leverage the idea that a performance prediction can be made for compositions of microservices based on the individually profiled microservices.

The profile of components of an application can be mapped upon the performance of an application as a whole. We consider both models to be suitable as they portray a trade-off between: 1) keeping the model simple, so that it may be easily adapted by practitioners and applied to any general microservice application, and 2) making sure the prediction is accurate enough to fulfill their purpose in hardware dimensioning.

We regard both models to be simple as they represent a linear addition of two core performance aspects of microservice-based applications. The CPU usage model considers

the microservice CPU usage and their overhead on a node, and the latency model considers the latencies of spans within microservices and their communication overhead. We have identified the communication overhead as an essential performance aspect in latency predictions through the related work in Chapter 3.

The results of our case study indicate that our method results in a sufficiently accurate performance prediction for both the CPU usage model and the latency model, where we deem a 20% difference in the prediction compared to the validation sufficient, based on the feedback of practitioners.

An important characteristic of the prediction models is that the performance prediction is dependent on the deployment configuration of microservices. It is taken into account in two distinct ways. 1) A CPU usage prediction of a node is made based on the number and types of microservices that run on it, and 2) a latency prediction for latency requirements considers whether communication between services is within or between nodes, as the latency for these cases is different.

Both models include the minimally required parameters that are needed to predict performance, but they may be extended by adding other parameters of a system or application to the prediction. For example, the effect of interference between microservices on the application performance may be a necessary addition, as discussed in Section 8.3.2.

8.2.1 CPU Usage Model

The CPU usage model shows the highest accuracy of the two models, where the predictions at most overshoot the validation by 16.25%, and undershoot by 6.80%, as shown in Figure 6.4. Moreover, the model is more often conservative than optimistic in its prediction, which is desirable in the performance prediction for hardware dimensioning. We can predict the trend in CPU usage accurately for increasing load for varying types and number of microservices on a node. Note that for a limited set of results, the prediction undershoots the validation. The expectation was that the model would overshoot all results, since it is conservative. One cause of the undershooting could be the lack of interference modeling, which may influence particular combinations of microservices. Another cause may be sampling artifacts, although we mitigate this by running our experiments many times.

While this model aims to predict the peak CPU usage on a node, it may also be possible to convert such a model into a prediction on other node resources, like memory usage, or other CPU usage metrics, like average CPU usage. Other related work discusses limiting memory usage with a sum resource usage of individual components of an application in similar ways [27, 42].

One limitation of the CPU usage model is that the overhead component may not be as generally applicable to any microservice application. It only considers a linear trend in overhead with increasing load, but the definition of load may differ per application. We still feel that the model is broadly applicable, as we have explained how to define the load and how to identify the overhead trend.

8.2.2 Latency Model

The latency model results show that for the general case, the accuracy of the prediction model is sufficiently high. Most predictions differ only at most 18.34% from the validation, and most predictions are conservative. Latency requirements **LR-6** and **LR-7** have the most accurate predictions, shown in Figure B.2e and Figure 6.9 respectively. Figure 6.6 show that the predictions for **LR-1** also generally fall within the 20% threshold, except for one outlier. We have explained the cause of this outlier with a deep dive into the distribution of latencies of one workflow. It indicates how complex a latency performance prediction may be. The predictions for **LR-3**, in Figure 6.8, are fairly accurate up until a load of 50 simultaneous orders. Adapting the latency model to completely avoid such inaccurate prediction may be complex, if not impossible, as the performance prediction model will increase in complexity with every edge case that is included. We therefore feel that we have struck the right balance between model simplicity and accuracy based on the results shown in the case study.

The inaccurate predictions seem to occur when the application breaks trends in latencies for increasing load, as shown for **LR-1** and **LR-3**. This could be caused by some resource restriction, or be related to the application design itself. When such behavior is noticed in the profiling stage, it could be investigated and may be incorporated into the prediction model, or be flagged as unexpected behavior to be fixed.

This model cannot only be used to predict requirement satisfaction, but also to generally predict the tail latency of a system workflow. In the context of requirement satisfaction, it makes sense to compute the expected percentile tail latency to guarantee that a percentage of requests will be completed within a certain timeframe, however, it is possible that for other purposes, different metrics are used to represent the system workflow latency, like average latency.

It should be noted that in our latency model, we make a distinction between local and remote communication, but do not distinguish between different distances in remote communication. It could be that the distance between a pair of nodes is much bigger than the distance between another. We do not consider this for two main reasons: 1) We

assume that the distance between nodes in a CPS is fairly short, and nodes are generally placed in the same room or area, and 2) we aim to keep the profiling method simple, to ensure scalability in the approach. This distinction also makes sense if we assume that communication latency mainly consists of the send-off and reception of a message, and the time it takes to travel from one node to another is only a small part of the latency, especially when messages take up a fraction of the bandwidth. This may differ per system and application, and therefore may need to be reconsidered when using this prediction model.

8.3 Threats To Validity

The elements that may compromise the validity of our approach comprise profiling time, microservice interference, the role of the MDS case study, the model variables, and the applicability of our approach to CPSs in general.

8.3.1 Profiling Time

Profiling time may be an issue for larger applications. While the time it takes to profile is heavily application, and workload-dependent, there are some techniques to cut down on that time.

First, the profile of individual microservices does not depend on one another, therefore, it is possible to optimize the time spent on profiling, as the microservices can be profiled in parallel. In larger systems especially, it would be beneficial to use multiple compute nodes, as one additional node could cut the profiling time in half. Do note that, for parallelization, more compute nodes are needed that match the hardware, which is not always an option.

Moreover, it may be useful to optimize an experiment, as the time it takes to run a whole profile heavily depends on the time of each experiment with a distinct deployment configuration and workload. For example, in Section 5.5.3, we explained that the profiling time for our small case study was roughly 25 hours. However, we did not optimize the experimental process, where each experiment with a distinct isolated microservice and load took around 45 seconds, with at least 30 seconds of idle time. This amount of idle time was due to the lack of feedback on when experiments were done running, so we preferred to be on the safe side to ensure to not cut off any experimental data. Optimizing this experiment could include building a feedback system where we could move on to the next experiment as soon as the previous one was done.

Lastly, profiling has to be done once per microservice and a microservice profile can be transferred from other applications if its profile is made on similar hardware. Also, once a profile is created, prediction models hardly take time to compute at all, and any deployment configuration and node can be tried out.

We have further tried to minimize profiling times by minimizing the number of profiling deployment configurations and iterations in iterative profiling.

Considering these mitigation techniques are available, the profiling has to be done only once in the development of a CPS, and it can be automated, we deem the profiling method to have an acceptable time to run the profiling for hardware dimensioning.

8.3.2 Microservice Interference

A key limitation in this work is that while we have identified microservice interference as a potential factor that influences the performance of microservice-based applications [33, 36], neither of the performance prediction models takes the effect of interference on the performance of microservices into account. The models shown in this thesis are the first step in performance prediction modeling in this context. Additionally, an investigation into the effect of interference on application performance may represent a distinct master thesis project. In the validation of the MDS, we have found that the microservices of the MDS have low resource usage, which indicates that interference plays only a small role in this application, as further shown in Appendix C.2. Therefore, for the case study, there is no need to model interference between microservices, and the performance prediction models proposed in this thesis are sufficient in this context.

We expect that microservice interference may more heavily influence the performance of other applications. Therefore interference modeling becomes a logical next step in future research on the performance prediction of microservice-based applications.

8.3.3 The Role of the MDS Case Study

We use the MDS case study as an example of how to perform hardware dimensioning through application profiling and performance prediction, but also in the validation of our proposed methods. It plays a large role in this thesis, as we only validate the concepts of this thesis on the MDS case study, which consists of four microservices that run on at most two nodes. Additionally, all microservices show fairly low CPU usage for the load generated by our workload suite as shown in Figure 7.1, and we validate the latency prediction model,

with latency requirements that all are part of one system workflow, rather than multiple workflows through the system.

Additionally, through the communication latencies profile of the MDS, we can see that the difference in latencies for local and remote communication is not significant on a prediction scale for any of the latency requirements in the MDS context. It seems to have little influence on the outcome of whether a latency requirement is met for a given load. This is very application-specific, and may not test the communication model sufficiently.

We have shown that taking the sum of the performance of individual components works fairly well for a low number of microservices and loads, but it could be that this does not hold for a much higher number of microservices, for example, due to the conservative nature of concatenating CPU usage peaks in our CPU usage model, and the complex nature of summing up tail latencies. Preferably this method is therefore also validated on a variety of applications with different sizes and hardware.

Thus, with only four microservices, the application is quite small, and tens of microservices may have better represented a general, more complex, CPS, that requires more computation, possibly larger data transfers, and more dynamic behavior.

In general, we feel that the MDS case study, despite these validity threats, is relevant in the validation of this hardware dimensioning approach, as this thesis aims to form the basis for a performance prediction method. It serves as an example of how to use the approach in practice and is a prototype that is based on an existing industrial CPS application. The case study application should encapsulate the balance between being simple enough to illustrate a basis for performing profiling, performance prediction, and validation experiments, while not straying too far from existing industrial CPS applications, which may be much more complex. The simplicity and the small number of microservices of this prototype proved to be useful in analyzing the application efficiently, with a focus on the basic principles of microservice-based applications. Nonetheless, further research and validation are necessary to investigate the applicability of the performance prediction methods in a broader context.

8.3.4 Tools, Tail Latency and CPU Usage Peak

The quality of profiling and validation results are highly dependent on the capabilities of the open-source tools that are used, like Prometheus [20], Jaeger [21], and cAdvisor [45]. Therefore, they should be considered carefully. For example, monitoring infrastructure may use a significant part of a node's resources. This may cause performance predictions to be much too conservative. Moreover, we noticed through additional experiments that

monitoring infrastructure could result in performance degradation because of interference with other microservices.

While we have performed some experiments on varying both the percentile tail latency and peak CPU usage, we have set these variables to a 90% tail latency, and CPU usage peak over 15 seconds. A limitation of this work is that we do not vary these variables in our validation. Future research could be conducted to investigate the effect of different CPU peak intervals, as well as different percentile tail latencies.

We partially could not investigate lower intervals to compute the CPU usage peaks, due to limitations of cAdvisor [45]. An open-source tool that monitors and gathers CPU usage data of containers in a Kubernetes [35] environment. cAdvisor appears to take up much of the CPU usage of a node, which significantly influenced the results in our experiments if set to frequent polling intervals of under 15 seconds. Therefore, we were unable to perform profiling and validation experiments with any lower monitoring intervals.

To mitigate the influence of monitoring services on the profiling and performance of the MDS, we have placed the infrastructure on a separate node while profiling. Additionally, we set the polling intervals of any services so that they do not use a significant part of a node's resources.

8.3.5 Applicability to CPSs

While the main goal of this thesis is to provide a structured approach to hardware dimensioning for microservice-based CPSs, we only consider the software components of a CPS.

A CPS can be a complex distribution system featuring a variety of compute nodes [50]. The spectrum of systems addressed in this context spans from large Intel machines to compact embedded boards tailored for real-time control applications. In the realm of CPSs, a notable challenge arises when compute nodes, such as the embedded boards, face resource constraints and must adhere to tight deadlines measured in micro- or milliseconds. Unlike the methodology presented in this thesis, these components of the system often necessitate static analysis for timing and schedulability, rather than relying on measurement-based performance predictions. The more strict latency requirements of a CPS can often be attributed to these compute nodes. The performance prediction of these hardware components falls out of the scope of this thesis, but should be taken into consideration when performing hardware dimensioning in the context of CPSs.

9

Conclusion and Future Work

In this thesis, we have proposed and implemented a structured approach to hardware dimensioning that consists of a profiling and a performance prediction method. Both have been validated through a case study. In this chapter, we demonstrate our key findings and insights derived from our work. Furthermore, we outline potential directions for future research.

9.1 Conclusion

In Chapter 1, we defined three research questions in the context of hardware dimensioning for microservice-based CPSs.

RQ1: What is a suitable profiling method and framework for a microservice-based CPS for the purpose of hardware dimensioning?
In this thesis, we have found that a suitable profiling method in the context of hardware dimensioning should apply to any general microservice application, be time-efficient, be able to be performed on a limited amount of hardware, and form the basis for performance prediction.

Keeping in mind these characteristics, we have created a profiling method that entails the profiling of individually isolated microservices, where the creation of workload suits and profiled metrics per microservice are derived from the performance requirements of the application. The approach requires only one node that represents the target system, in addition to a set of nodes that does not need to match it. We have determined that a suitable profiling framework in this context consists of automated deployment, automated request serving, and data collection and processing infrastructure.

Additionally, we have found that a profile should be built iteratively, where an application-specific balance needs to be found between 1) the time it takes to profile, and 2) iterating the profile enough times to ensure that the profiling results indicate trends between the workload and performance of an application.

RQ2: What is a suitable performance prediction method that predicts whether the performance requirements of a microservice-based CPS are met, with a given deployment configuration of microservices over a set of homogeneous nodes and an application profile?

We have identified that a suitable performance prediction method should strike a balance between being simple, and being accurate in its prediction, with a preference for a conservative prediction. Additionally, the prediction models that are created through this method should be able to distinguish between different deployment configurations. We have determined that a prediction accuracy of 20% is sufficient, based on the feedback of practitioners.

With these characteristics in mind, we have created a cumulative performance prediction method that leverages the idea that a performance prediction can be made about the compositions of microservices based on the individually profiled microservices.

We have created two performance prediction models that both consider two core performance aspects: 1) a CPU usage prediction model that includes CPU usage and overhead of isolated microservices and predicts CPU usage peak of microservices, and 2) a latency prediction model, which considers the latencies of spans within isolated microservices, and the communication latencies between microservices that can predict tail latencies of system workflows. Their predictions are based on a given application profile and deployment configuration. The predictions of both models differ by less than 20% from the validation experiments. Additionally, both prediction models are generally conservative in their predictions.

RQ3: How can we use the profiling and performance prediction methods to come to a structured approach to hardware dimensioning in the context of a microservice-based CPS?

We can use the profiling method described in this work as input for our performance prediction models. The predictions of the models then indicate whether relevant performance requirements are met for a given deployment configuration and number of nodes, and therefore, whether the deployment configuration is sufficient.

By determining whether we can find a sufficient deployment configuration for a given number of nodes through manual design space exploration with the performance prediction models as evaluation, we can determine whether we can satisfy the relevant performance requirements for the specified number of nodes. By varying the number of nodes, we may come to a minimum number for which the relevant performance requirements can be satisfied.

Thus, altogether, the profiling method, performance prediction method, and manual design space exploration form a structured approach to hardware dimensioning in this way.

9.2 Future work

We have provided a method of application profiling and performance prediction for hardware dimensioning that may form the basis for various research directions. In this Section, we will lay out some of the more promising directions.

The Purpose of Profiling and Performance Prediction

While the profiling and performance prediction methods are designed for hardware dimensioning, they could be applied for other purposes, one of which is performance optimization through the redistribution of microservices. It could be beneficial to leverage the approaches in this thesis to fit that context, as the environments of systems may change. For example, the number of microservices can be upscaled, or the number of available compute nodes may be diminished. Predicting performance in such a context and adapting to an optimal deployment configuration as quickly as possible when such events occur may be beneficial for an application's performance.

Comprehensive Validation

In our discussion in Chapter 8, we mention that we have only validated our approach on the MDS. While the MDS is made as a prototype of an industrial CPS, it does not represent all CPSs, and does not scale to most. Therefore, future work could investigate the broad applicability of our approach further by validating it on a more comprehensive set of applications, that varies in size and complexity.

Different Metrics

The resource usage metrics that we use in our profiles and prediction models are two of many options that could be considered. In future work, the approach could be extended to include other metrics, such as the average CPU usage of a microservice, average latencies of system workflows, or variations of memory usage or speed. An investigation can be conducted to validate our approach of summing the metrics of components of an application for other metric types.

Performance Aspects

The profiles and performance prediction models could be extended to include a myriad of other performance aspects, like microservice interference or the effect of increased message sizes on application performance. Extending the model could increase the accuracy of predictions and explain anomalies in our results. We feel that the most logical next step in the extension of our models is the inclusion of microservice interference. However, this is a broad topic, where interference can have many causes, therefore we feel that it could span multiple theses.

Design Space Exploration

In our hardware dimensioning approach we perform manual design space exploration dimensioning to find a suitable deployment configuration for a given number of nodes. However, this approach may be more challenging for larger applications, and therefore, future research could look into how to find an optimal deployment configuration for any given number of nodes in a more structured way.

Heterogeneous Nodes

Lastly, this work solely considers systems with homogeneous nodes. In Appendix C.3, we have shown a very simple experiment as an example of the effect of using heterogeneous nodes with our performance prediction method. Future work could conduct such experiments to investigate extending the approach to a heterogeneous methodology by looking at the applicability of predictions on other node types and to what extent it is possible to construct a performance prediction model that is independent of hardware.

Acknowledgements

A special thanks to Prof. Dr. Benny Akesson, Ben Pronk, and others on the TNO-ESI team, for providing guidance and feedback throughout this project. Without their support, this thesis would not have been possible.

References

[1] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. **Microservices: yesterday, today, and tomorrow**. *Present and ulterior software engineering*, pages 195–216, 2017. 1

[2] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. **Microservices: The journey so far and challenges ahead**. *IEEE Software*, **35**(3):24–35, 2018. 1, 7

[3] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, et al. **Service fabric: a distributed platform for building microservices in the cloud**. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018. 1

[4] Robert Harrison, Daniel Vera, and Bilal Ahmad. **Engineering methods and tools for cyber–physical automation systems**. *Proceedings of the IEEE*, **104**(5):973–985, 2016. 1, 7

[5] Carolina Villarreal Lozano and Kavin Kathiresh Vijayan. **Literature review on cyber physical systems design**. *Procedia manufacturing*, **45**:295–300, 2020. 1

[6] Edward A Lee. **Cyber physical systems: Design challenges**. In *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*, pages 363–369. IEEE, 2008. 1, 6

[7] Davide Taibi, Valentina Lenarduzzi, Claus Pahl, and Andrea Janes. **Microservices in agile software development: a workshop-based study into**

issues, advantages, and disadvantages. In *Proceedings of the XP2017 Scientific Workshops*, pages 1–5, 2017. 1

[8] Adalberto R Sampaio, Julia Rubin, Ivan Beschastnikh, and Nelson S Rosa. **Improving microservice-based applications with runtime placement adaptation**. *Journal of Internet Services and Applications*, **10**(1):1–30, 2019. 2

[9] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, Xin Peng, Wenli Zheng, and Minyi Guo. **Qos-aware and resource efficient microservice deployment in cloud-edge continuum**. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 932–941. IEEE, 2021. 2

[10] Radhakisan Baheti and Helen Gill. **Cyber-physical systems**. *The impact of control technology*, **12**(1):161–166, 2011. 6

[11] Joy Rahman and Palden Lama. **Predicting the end-to-end tail latency of containerized microservices in the cloud**. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 200–210. IEEE, 2019. 7, 16, 19, 20

[12] Marijn Vollaard. **Hardware Dimensioning for Microservice Applications in Cyber-Physical Systems: Current Directions and Challenges**. *Literature Study, UvA/VU*. 7, 12, 16

[13] Bowen Li, Xin Peng, Qilin Xiang, Hanzhang Wang, Tao Xie, Jun Sun, and Xuanzhe Liu. **Enjoy your observability: an industrial survey of microservice tracing and analysis**. *Empirical Software Engineering*, **27**:1–28, 2022. 8, 9

[14] Shivakumar R Goniwada and Shivakumar R Goniwada. **Observability**. *Cloud Native Architecture and Design: A Handbook for Modern Day Architecture and Design with Enterprise-Grade Examples*, pages 661–676, 2022. 8

[15] Radoslav Gatev and Radoslav Gatev. **Observability: Logs, metrics, and traces**. *Introducing Distributed Application Runtime (Dapr) Simplifying Microservices Applications Development Through Proven and Reusable Patterns and Practices*, pages 233–252, 2021. 8

[16] Mainak Chakraborty and Ajit Pratap Kundan. **Observability**. In *Monitoring Cloud-Native Applications: Lead Agile Operations Confidently Using Open Source Software*, pages 25–54. Springer, 2021. 8

[17] **Gartner - Peer Insights**, 2023. Accessed: Dec. 5, 2023 [Online]. Available: https://www.gartner.com/peer-insights/home. 8, 9

[18] **Paessler PRTG**, 2023. Accessed: Dec. 5, 2023 [Online]. Available: https://www.gartner.com/reviews/market/infrastructure-monitoring-tools/vendor/paessler/product/paessler-prtg. 8

[19] **OPManager**, 2023. Accessed: Dec. 5, 2023 [Online]. Available: https://www.gartner.com/reviews/market/infrastructure-monitoring-tools/vendor/manageengine/product/opmanager. 8

[20] **Prometheus, From metrics to insight**, 2023. Accessed: Oct. 16, 2023 [Online]. Available: https://prometheus.io/. 8, 39, 73

[21] **Jaeger: open source, distributed tracing platform**, 2023. Accessed: Oct. 16, 2023 [Online]. Available: https://www.jaegertracing.io/. 9, 40, 73

[22] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. **Model-based performance prediction in software development: A survey**. *IEEE Transactions on Software Engineering*, **30**(5):295–310, 2004. 11

[23] Allegra De Filippo, Andrea Borghesi, Andrea Boscarino, and Michela Milano. **HADA: An automated tool for hardware dimensioning of AI applications**. *Knowledge-Based Systems*, **251**:109199, 2022. 12

[24] Madhura Purnaprajna, Marek Reformat, and Witold Pedrycz. **Genetic algorithms for hardware–software partitioning and optimal resource allocation**. *Journal of Systems Architecture*, **53**(7):339–354, 2007. 13

[25] Anh Vu Do, Junliang Chen, Chen Wang, Young Choon Lee, Albert Y Zomaya, and Bing Bing Zhou. **Profiling applications for virtual machine placement in clouds**. In *2011 IEEE 4th international conference on cloud computing*, pages 660–667. IEEE, 2011. 13, 19, 20

[26] Jungsu Han, Yujin Hong, and Jongwon Kim. **Refining microservices placement employing workload profiling over multiple kubernetes clusters**. *IEEE access*, **8**:192543–192556, 2020. 13, 19, 20

[27] Liang Bao, Chase Wu, Xiaoxuan Bu, Nana Ren, and Mengqing Shen. **Performance modeling and workflow scheduling of microservice-based applications in clouds**. *IEEE Transactions on Parallel and Distributed Systems*, **30**(9):2114–2129, 2019. 14, 16, 17, 19, 20, 69

[28] Rafael Weingärtner, Gabriel Beims Bräscher, and Carlos Becker Westphall. **Cloud resource management: A survey on forecasting and profiling models**. *Journal of Network and Computer Applications*, **47**:99–106, 2015. 14

[29] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. **SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications**. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 165–176, 2021. 14, 16, 19, 20

[30] Wenkai Lv, Quan Wang, Pengfei Yang, Yunqing Ding, Bijie Yi, Zhenyi Wang, and Chengmin Lin. **Microservice deployment in edge computing based on deep Q learning**. *IEEE Transactions on Parallel and Distributed Systems*, **33**(11):2968–2978, 2022. 14, 16, 19, 20

[31] Kaihua Fu, Wei Zhang, Quan Chen, Deze Zeng, and Minyi Guo. **Adaptive resource efficient microservice deployment in cloud-edge continuum**. *IEEE Transactions on Parallel and Distributed Systems*, **33**(8):1825–1840, 2021. 14, 19, 20, 64

[32] Isil Karabey Aksakalli, Turgay Celik, Ahmet Burak Can, and Bedir Tekinerdogan. **Systematic approach for generation of feasible deployment alternatives for microservices**. *IEEE Access*, **9**:29505–29529, 2021. 14, 19, 20

[33] Christina Delimitrou and Christos Kozyrakis. **ibench: Quantifying interference for datacenter applications**. In *2013 IEEE international symposium on workload characterization (IISWC)*, pages 23–33. IEEE, 2013. 14, 15, 19, 20, 72

[34] Christina Terese Joseph and K Chandrasekaran. **IntMA: Dynamic Interaction-aware resource allocation for containerized microservices in cloud environments**. *Journal of Systems Architecture*, **111**:101785, 2020. 14, 19, 20, 64

[35] **Production-Grade Container Orchestration, Kubernetes**, 2023. Accessed: Oct. 4, 2023 [Online]. Available: https://kubernetes.io/. 14, 27, 37, 74

[36] Madhura Adeppady, Paolo Giaccone, Holger Karl, and Carla Fabiana Chiasserini. **Reducing Microservices Interference and Deployment Time in Resource-constrained Cloud Systems**. *IEEE Transactions on Network and Service Management*, 2023. 14, 17, 19, 20, 72

[37] **SPEC CPU**, 2006. Accessed: Oct. 2, 2023 [Online]. Available: http://www.spec.org/cpu2006/index.html. 15

[38] Johannes Grohmann, Patrick K Nicholson, Jesus Omana Iglesias, Samuel Kounev, and Diego Lugones. **Monitorless: Predicting performance degradation in cloud applications with machine learning**. In *Proceedings of the 20th international middleware conference*, pages 149–162, 2019. 16

[39] Wubin Ma, Rui Wang, Yuanlin Gu, Qinggang Meng, Hongbin Huang, Su Deng, and Yahui Wu. **Multi-objective microservice deployment optimization via a knowledge-driven evolutionary algorithm**. *Complex & Intelligent Systems*, **7**:1153–1171, 2021. 16, 19, 20

[40] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. **Sinan: ML-based and QoS-aware resource management for cloud microservices**. In *Proceedings of the 26th ACM international conference on architectural support for programming languages and operating systems*, pages 167–181, 2021. 16, 19, 20

[41] Wojciech Samek, Grégoire Montavon, Sebastian Lapuschkin, Christopher J Anders, and Klaus-Robert Müller. **Explaining deep neural networks and beyond: A review of methods and applications**. *Proceedings of the IEEE*, **109**(3):247–278, 2021. 16

[42] Shiping Chen, Yan Liu, Ian Gorton, and Anna Liu. **Performance prediction of component-based applications**. *Journal of Systems and Software*, **74**(1):35–43, 2005. 16, 19, 20, 69

[43] Sanford Weisberg. *Applied linear regression*, **528**. John Wiley & Sons, 2005. 18

[44] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016. 21

[45] **cAdvisor (Container Advisor)**, 2023. Accessed: Oct. 16, 2023 [Online]. Available: https://github.com/google/cadvisor. 39, 73, 74

[46] **Node exporter, Prometheus exporter for hardware and OS metrics**, 2023. Accessed: Oct. 16, 2023 [Online]. Available: https://github.com/prometheus/node_exporter. 39

[47] **OpenTelemtry, High-quality, ubiquitous, and portable telemetry to enable effective observability**, 2023. Accessed: Oct. 16, 2023 [Online]. Available: https://opentelemetry.io/. 40

[48] **Elasticsearch**, 2023. Accessed: Oct. 16, 2023 [Online]. Available: https://github.com/elastic/elasticsearch. 40

[49] Marius Herget, Faezeh Sadat Saadatmand, Martin Bor, Ignacio González Alonso, Todor Stefanov, Benny Akesson, and Andy D Pimentel. **Design Space Exploration for Distributed Cyber-Physical Systems: State-of-the-art, Challenges, and Directions**. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 632–640. IEEE, 2022. 63

[50] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert I Davis. **A comprehensive survey of industry practice in real-time systems**. *Real-Time Systems*, **58**(3):358–398, 2022. 74

Appendix A

The Full MDS Profile

A.1 The Full MDS Profile

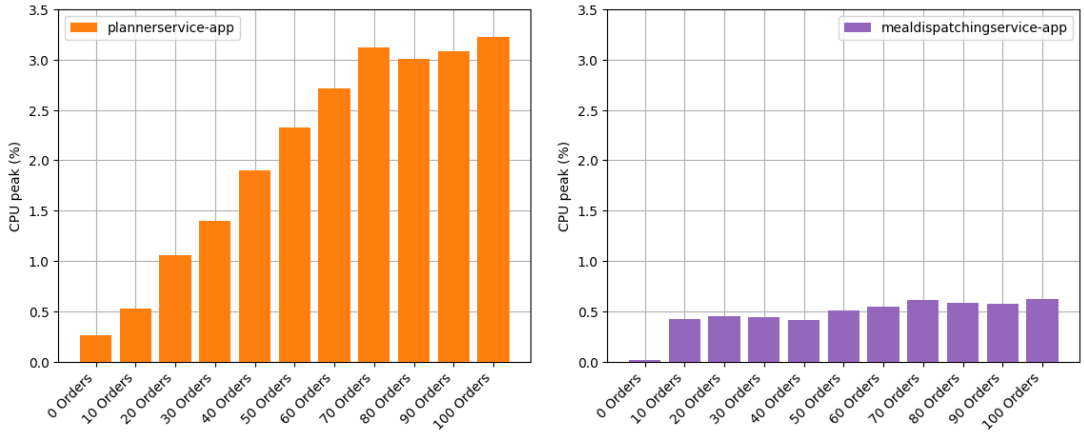
Here, the full MDS profile is shown.

First of all, for each microservice in the MDS, we store the microservice peak CPU usage for varying loads, as shown in Figure A.1. The exact metric of peak CPU usage is explained in Section A.1. We also store the offset, b, and slope, a, for the overhead fitting of each microservice as shown in Table A.1.

For each latency requirement, we store the 90th percentile tail latencies of the spans in microservices that represent its system workflow for varying load, as shown in Figure A.3. Lastly, we store the 90th percentile tail latency communication latencies that represent the latency requirement workflows, where we make a distinction between latencies of local communication and remote communication as shown in Figure A.4.

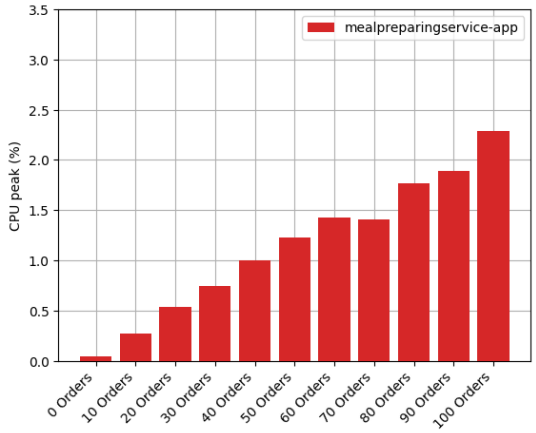
Table A.1: Linear fitting of the overhead for each microservice in the MDS, based on increasing load. a represents the slope and b the offset of the linear fitting, which has the form $ax + b$.

Service	a	b
PlannerService	0.017	0.217
MealDispatchingService	0.001	0.249
MealPreparingService	0.004	0.287
MealDeliveringService	0.002	0.250

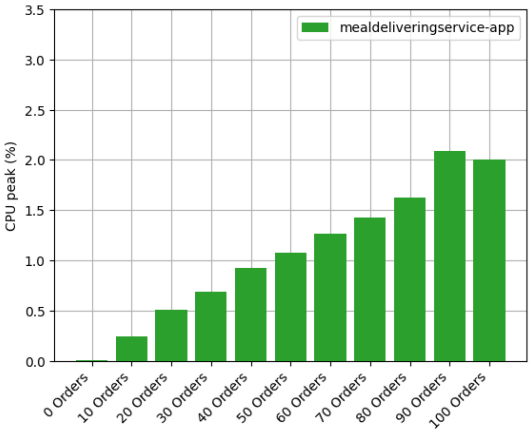


(a) PlannerService

(b) MealDispatchingService

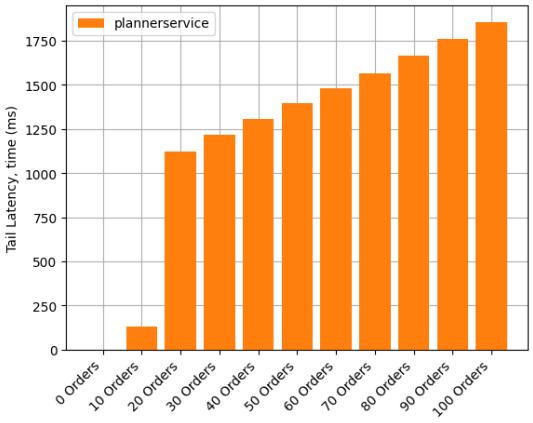


(c) MealPreparingService

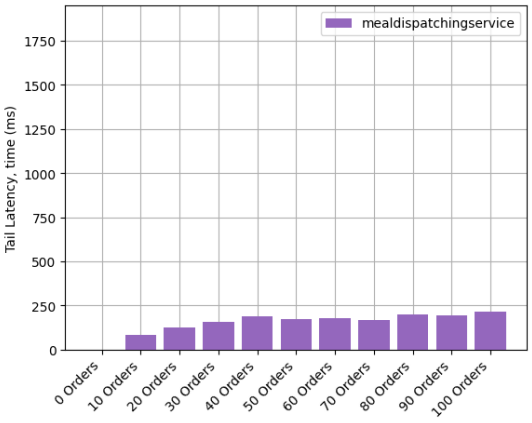


(d) MealDeliveringService

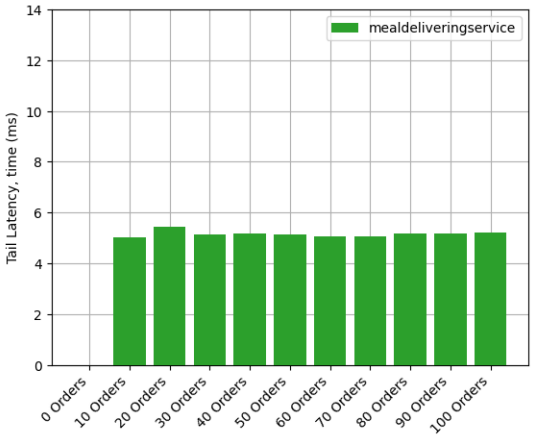
Figure A.1: The CPU usage component of the MDS profile.



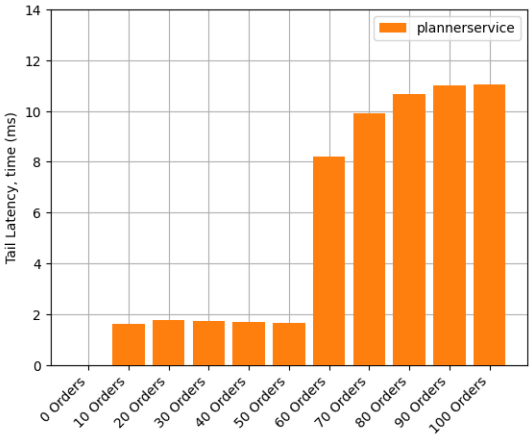
(a) LR-1 - PlannerService.



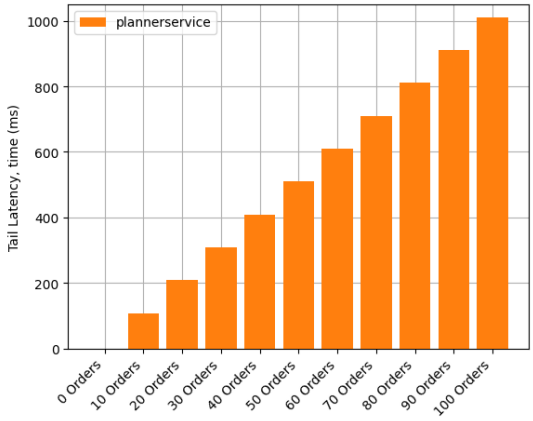
(b) LR-1 - MealDispatchingService.



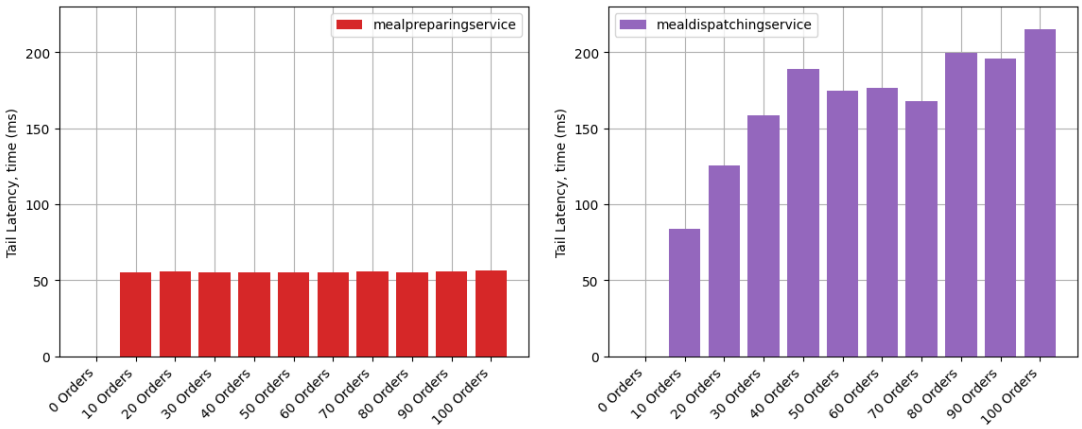
(c) LR-3 - MealDeliveringService



(d) LR-3 - PlannerService

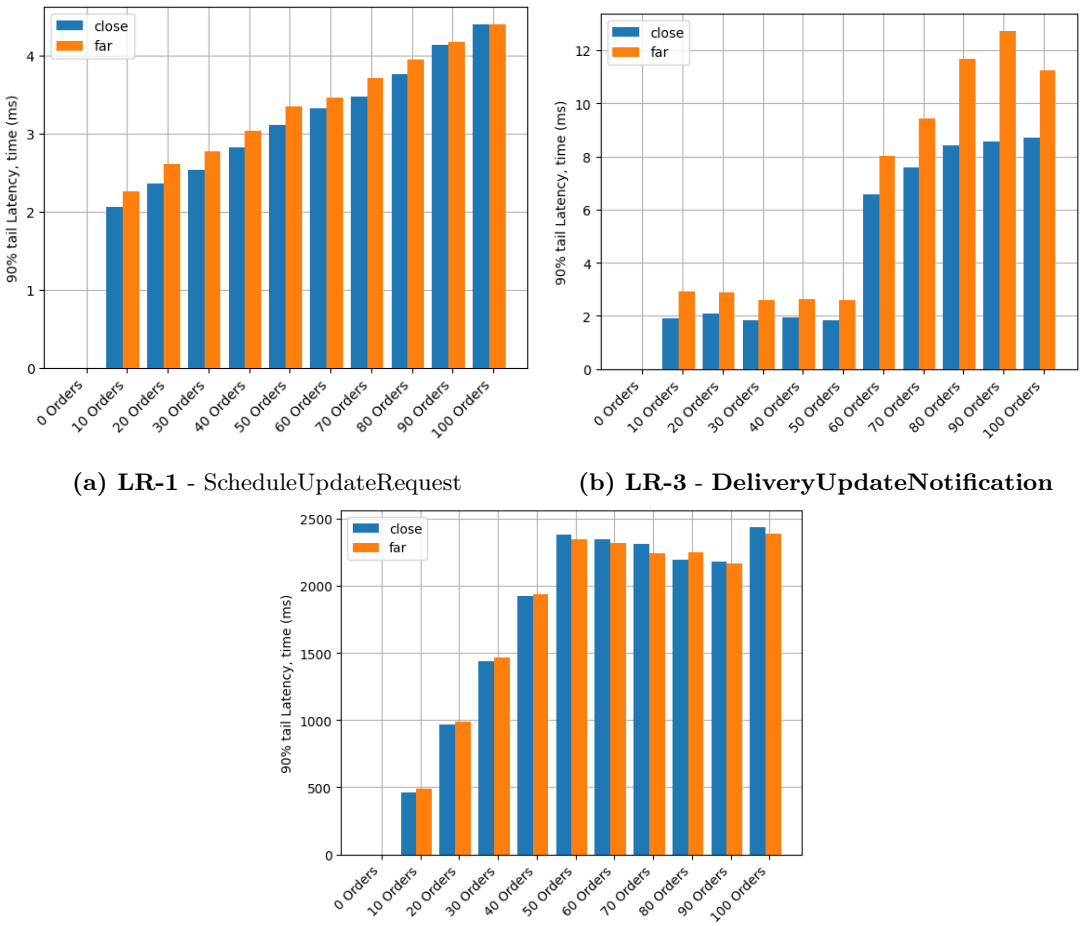


(e) LR-6 - PlannerService



(a) LR-7 - MealPreparingService **(b) LR-7** - MealDispatchingService

Figure A.3: The 90th percentile latencies of the spans for **LR-1, LR-3, LR-6, and LR-7** in the MDS



(a) LR-1 - ScheduleUpdateRequest **(b) LR-3 - DeliveryUpdateNotification**

(c) LR-7 - MealPreparationRequest

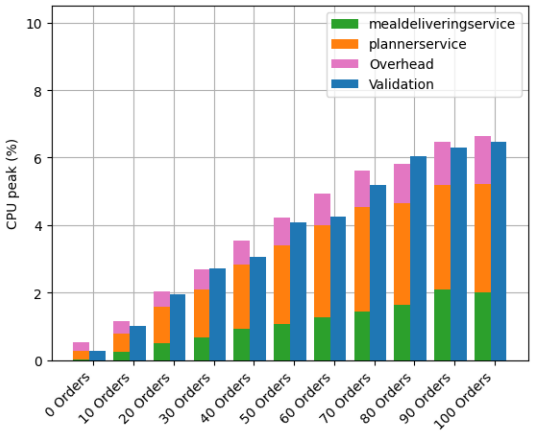
Figure A.4: The Communication component of the MDS profile.

Appendix B

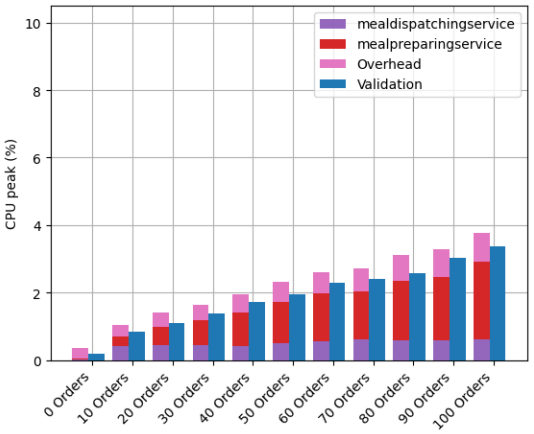
Validation Results

In this Appendix, all validation results of the CPU usage model and latency model are shown.

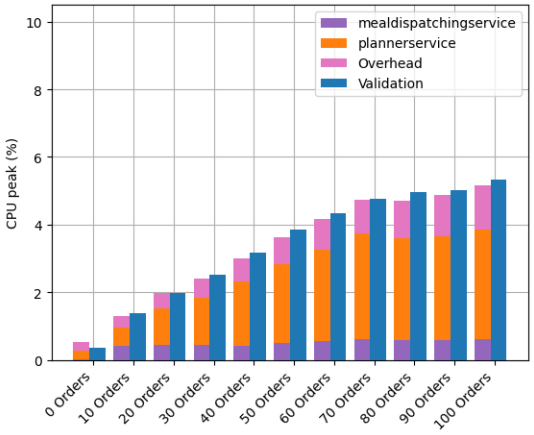
B.1 CPU Model



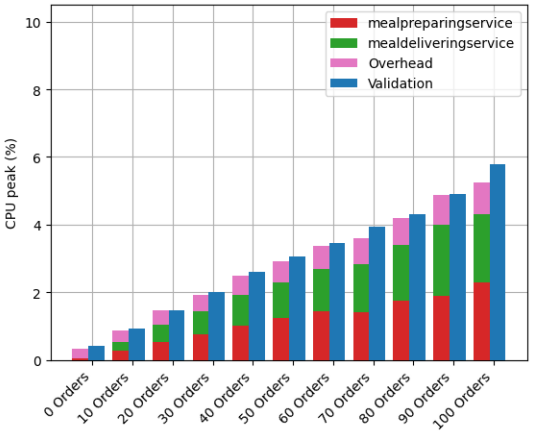
(a) EvenSplit 1, Transact01



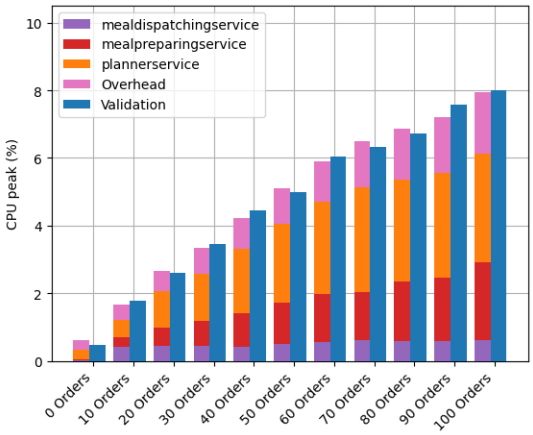
(b) EvenSplit 1, Transact02



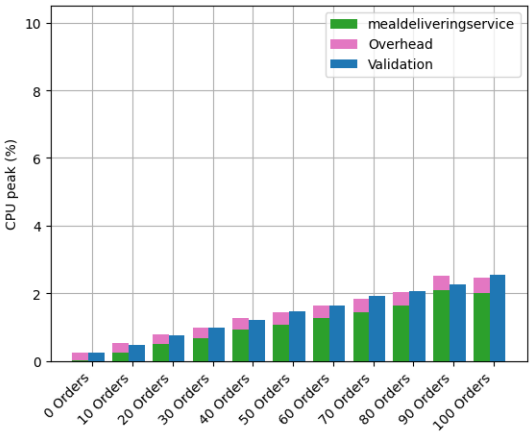
(c) EvenSplit 2, Transact01



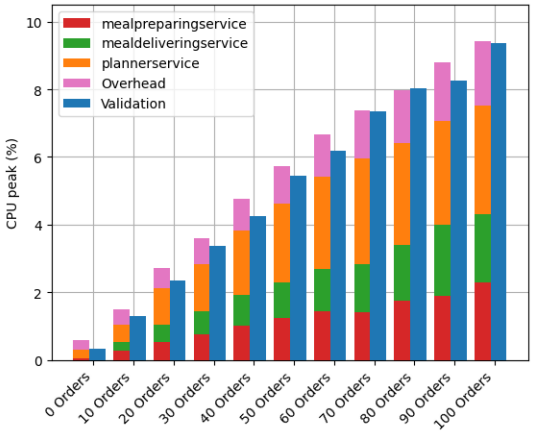
(d) EvenSplit 2, Transact02



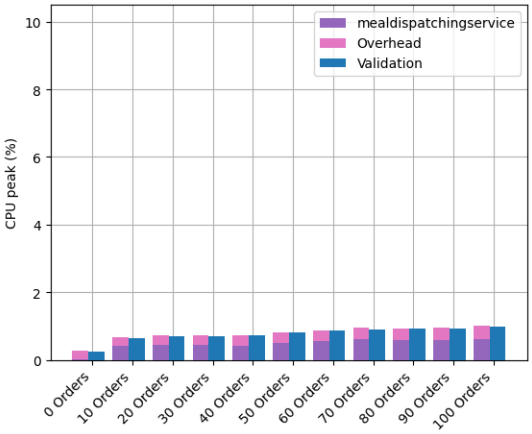
(e) TripleSolo 1, Transact01



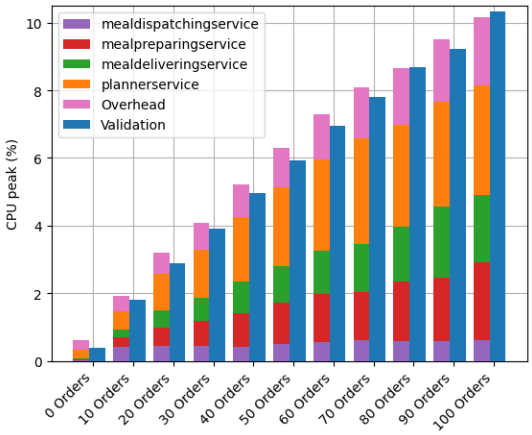
(f) TripleSolo 1, Transact02



(g) TripleSolo 2, Transact01



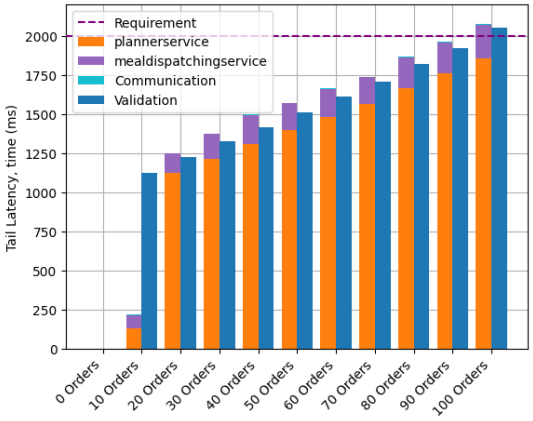
(h) TripleSolo 2, Transact02



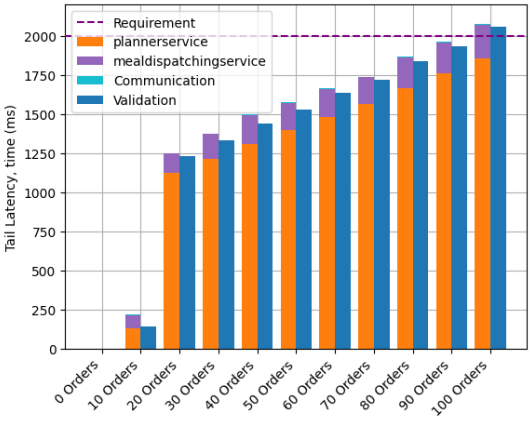
(i) OneNode, Transact01

Figure B.1: The prediction against the validation for the EvenSplit 1, EvenSplit 2, TripleSolo 1, TripleSolo 2, and OneNode deployment configuration.

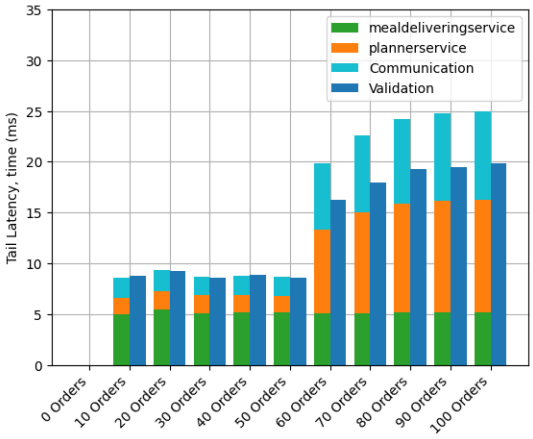
B.2 Latency Model



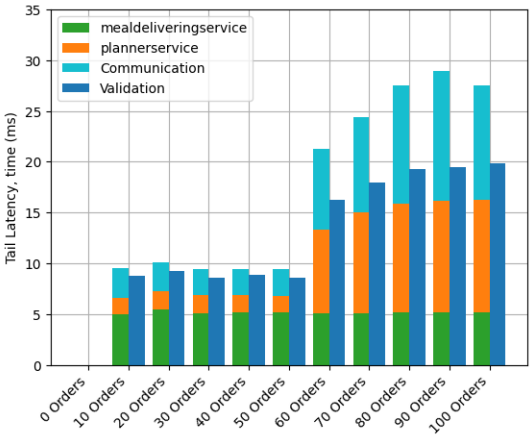
(a) EvenSplit 2, local



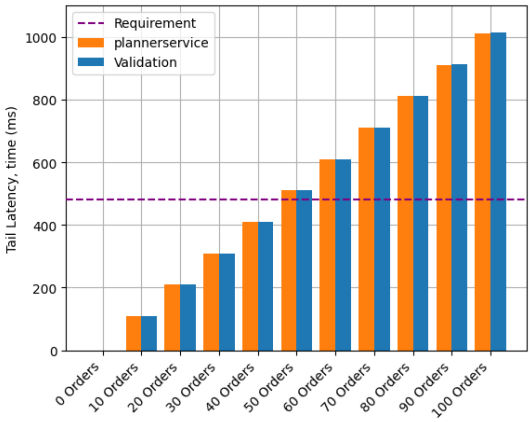
(b) EvenSplit 1, remote



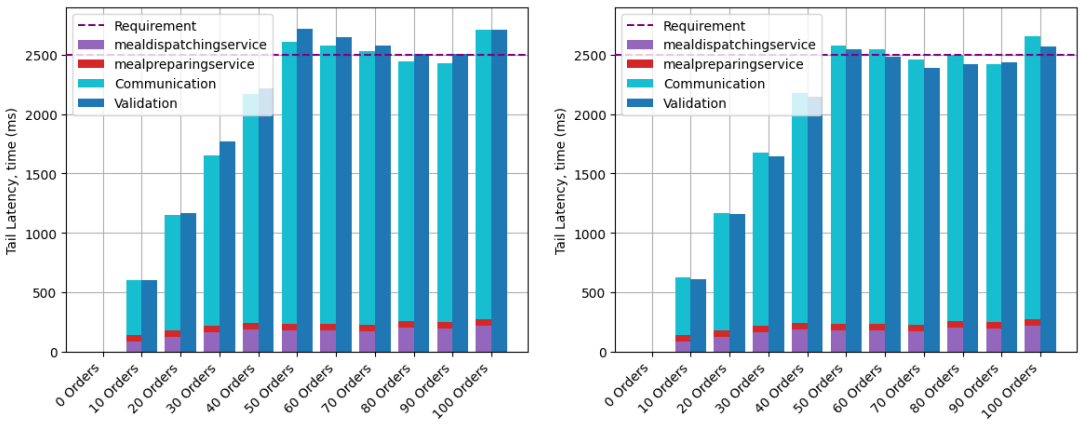
(c) EvenSplit 1, local



(d) EvenSplit 2, remote



(e) EvenSplit 1, local



(f) EvenSplit 1, local

(g) EvenSplit 2, remote

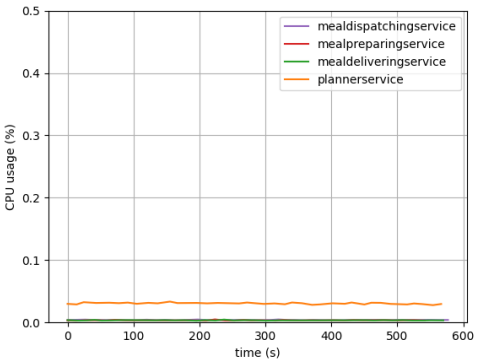
Figure B.2: Prediction of 90th percentile tail latency of the system workflow based on **LR-1**, **LR-3**, **LR-6**, and **LR-7** compared to its validation for varying load.

Appendix C

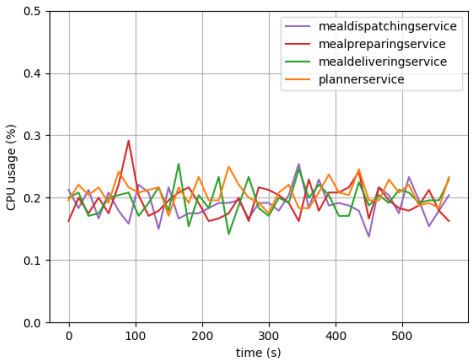
Experiments, Interference and Heterogeneous Nodes

C.1 Idle Overhead Experiments

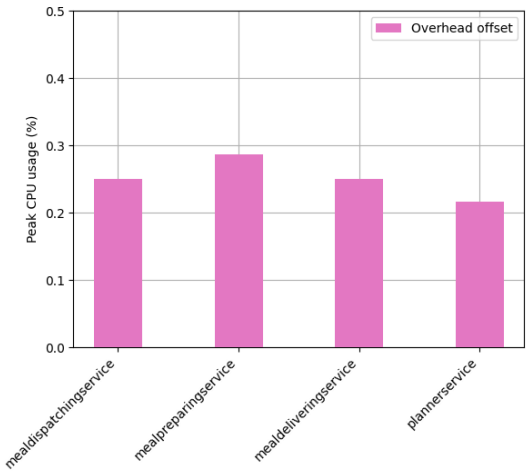
The results of the overhead offset experiment in the context of the MDS are shown in Figure C.1. The idle microservice CPU usage is shown in Figure C.1a and the idle node CPU usage is shown in Figure C.1b. The experiment runs for 600 seconds, or 10 minutes in this example, as we feel that is sufficient time for the MDS to capture the general idle CPU usage of each microservice on the TNO cluster. Figure C.1c shows the idle overhead of each microservice, which is calculated as the difference between the peak node and peak microservice CPU usage over the 10 minutes. Note that the difference in overhead offset is fairly minimal, and at most 0.07%.



(a) The CPU usage of each idle microservice in isolation.



(b) The CPU usage of the node on which each microservice runs in isolation.



(c) The idle overhead of each microservice in isolation.

Figure C.1: Overhead offset results of the CPU usage idle experiment in the MDS context.

C.2 Experiment on the Effect of Interference on Latencies

While it is not included in our model, we briefly investigate whether interference between microservices affects latencies in the MDS context. We do so by comparing the latencies of **LR-3** with the validation of TripleSolo 1, where the MealDeliveringService runs in isolation on Transact01, and OneNode on Transact01, where all services run together. Here we may specifically see the 90th percentile tail latencies for the OneNode validation to be higher if interference contributes significantly to the latency of **LR-3**. If we do not see this difference, we may see that the latencies of the TripleSolo 1 configuration are higher. That could indicate that the communication latency has a more significant part in the latency

than the interference between microservices in the MDS context.

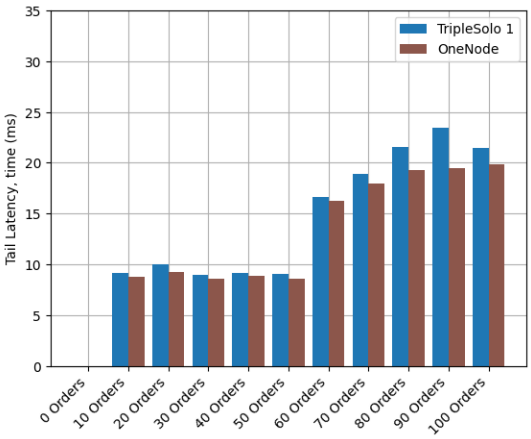


Figure C.2: Comparison of 90th percentile tail latency of **LR-3** for TripleSolo 1 and OneNode deployment configurations.

The results show that the TripleSolo 1 deployment configuration results in higher latencies. Thus the interference between microservices does not seem to have a bigger impact on the latencies than the communication does. Note that this result is MDS-specific.

C.3 Heterogeneous Nodes

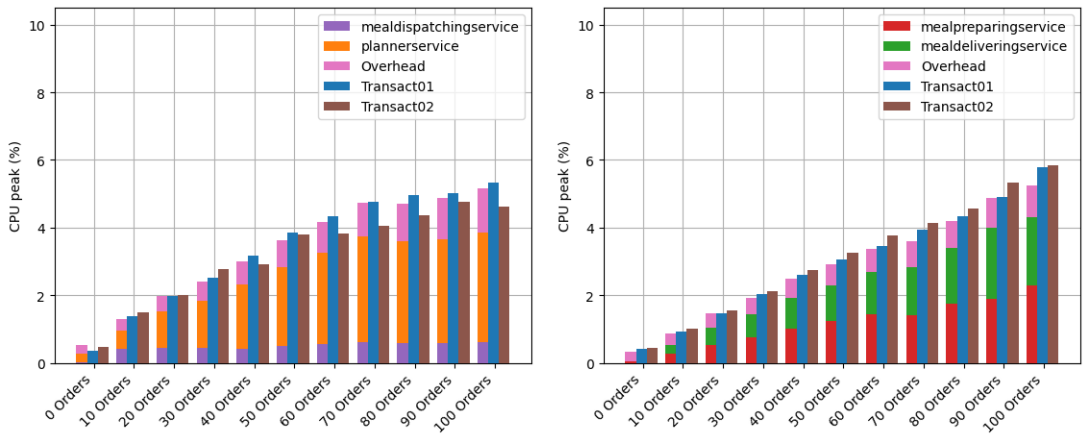
While the model is made for profiling and performance prediction on homogeneous nodes, the profile of one type of node could be applied to a performance prediction for nodes with slightly different specifications as well. In this section, we further investigate this concept by comparing the model results based on the Transact01 profile to validation on the Transact02 node. In the experiments shown in this section, we run each configuration fifty times, and we have taken the average over all runs.

The difference in specification between Transact01 and Transact02 is mainly their CPU base frequency, where Transact01 has a base clock frequency of 2.60 GHz, while Transact02 has a base clock frequency of 2.40 GHz.

C.3.1 CPU Usage Model

We have conducted a short experiment with the EvenSplit 2 deployment configuration, where the CPU usage of each pair of microservices is predicted based on the Transact01 profile. Then we compare the prediction to the scenario where each pair of microservices of EvenSplit 2 are deployed on Transact02 and Transact01. The results are shown in Figure C.3.

The expectation is that the CPU usage of Transact01 will be lower than that of Transact02, since it has a lower clock frequency. Higher clock speeds allow a processor to execute instructions more quickly, potentially leading to faster task completion and lower overall CPU utilization during the specified interval.



(a) First pair of microservices on Transact02 **(b)** Second pair of microservices on Transact02

Figure C.3: Prediction of the CPU usage model for the EvenSplit 2 deployment configuration compared to validation on Transact01 and Transact02.

We can see that in this example, in some cases, Transact01 has a higher CPU usage peak than Transact02 and vice versa. Since the specifications of both nodes are fairly similar, and the CPU usage does not go over 6%, these results may be sensitive to noise. Additionally, other node specifications, such as cache size, or memory speed may influence the results.

Generally, the prediction remains fairly accurate. At most, it overshoots the validation on Transact02 with 17.24% and undershoots with 12.90% overall deployment configurations considered in the Chapter 6, compared to 16.25% and 6.80% respectively for Transact01. It appears that the prediction both overshoots and undershoots more when validating on Transact02. Therefore, we cannot simply say that Transact01 is faster and therefore has higher performance in all cases.

The results show that a prediction could be made on different types of nodes, as long as the nodes do not differ much. Exactly how much the nodes may differ to come to a sufficiently accurate prediction is dependent on the application and system. Looking into how well you can profile nodes and adapt a model fitting heterogeneous nodes should be looked into in future work.

C.3.2 Latency Model

The validation for the latency model for remote communication is not done on two homogeneous nodes, but on two slightly different nodes, as we had no two homogeneous nodes available to us. We still feel that this is not an issue in the validation, and that it would almost exactly match the situation with two homogeneous nodes. To support this claim, we have performed an experiment where we compare the latencies of **LR-3** with local communication on the Transact01 node to the latencies with the same setup on the Transact02 node. The experiment is done using the EvenSplit 1 deployment configuration and its mirrored version, where the MealDeliveringService and PlannerService are put on the same node. The results are shown in Figure C.4.

In general, we would expect the latencies on Transact02 to be slightly higher than those of Transact01, as Transact01 is slightly faster in base clock frequency. However, we have shown in the previous section that this does not hold for the peak CPU usage on both nodes.

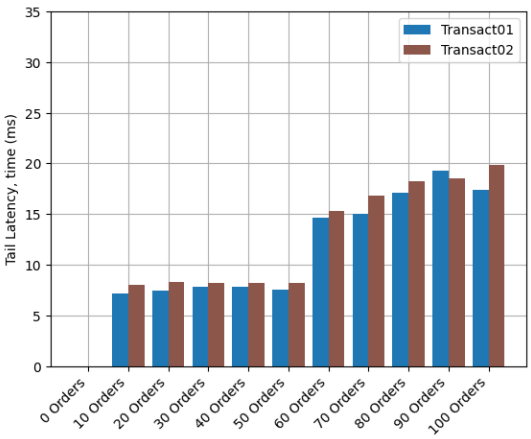


Figure C.4: Comparison of 90th percentile tail latency of **LR-3** on Transact01 and Transact02 with the same microservices deployed on them.

As expected, the results show latencies of Transact01 are slightly lower than those of Transact02. The variability in latencies can be partly explained by noise, since two validation iterations on the same node may have some variability as well. These results are similar throughout all latency requirements discussed in the MDS.

Since the validation is done on two heterogeneous nodes, likely, the latency predictions made in this chapter may even be more accurate than portrayed, as the profile on which the prediction is based, is made for only one type of node.