ACIT5900

Master Thesis

in

Applied Computer and Information Technology (ACIT)

May 2023

Cloud-Based Services and Operations

Towards Green Container Management:

A novel approach for container resource allocation through statistical modeling

Håkon Borgersen Ay

Department of Computer Science
Faculty of Technology, Art and Design

Towards Green Container Management: A novel approach for container resource allocation through statistical modeling
Supervisor: Kyrre Begnum

https://www.oslomet.no

Printed: Oslo Metropolitan University

Abstract

In this exploratory study, a comprehensive framework is presented that leverages ARIMA and Facebook's Prophet time series forecasting models for predicting container resource usage in a Kubernetes environment. The framework includes data collection and processing, as well as the development of an algorithm for suggesting resource allocation, which is combined with the Prophet model. Multiple KPIs were devised to evaluate the performance of the resource allocation algorithm, the statistical models, and a combined method. The proposed framework offers a successful solution for suggesting resource requests and limits. When applied to a sample Kubernetes node, the framework resulted in a 39.3% reduction in allocated resources and a 60% increase in memory usage coverage. This approach significantly benefits Kubernetes environments, promoting greener resource management and reduced electricity costs.

Preface

It all began when Intility presented a thesis proposal titled "Kubernetes for Sustainability" to Oslomet. After conducting my own research and discussing the thesis with Intility and my supervisor, I decided to focus on resource usage in Kubernetes projects. My team at Intility, who possess extensive experience with Kubernetes, assured me from the outset that the project was relevant and would provide valuable insights and benefits for them. At the same time, my supervisor Kyrre and I ensured that the thesis could stand independently.

During the literature review phase in early December 2022, having agreed upon and signed off on the project but not yet officially commenced, I quickly realized that I needed to deepen my understanding of Kubernetes. To facilitate better analysis and modeling of resource usage, I took a course on Kubernetes administration. By February, I had obtained the Certified Kubernetes Administration (CKA) certificate from the Linux Foundation.

A brief introduction to Intility: Intility is a comprehensive platform service for multi-cloud IT environments. As a fully managed platform service, it is currently utilized by over 600 companies across 2000 locations worldwide. In addition to providing the platform service, Intility places great emphasis on sustainability, considering it a prerequisite rather than an alternative mindset. Through numerous initiatives over the years, they have demonstrated their commitment to this field.

To maintain anonymity, all endpoints, node names, pod names, and container names have been anonymized.

Acknowledgments

I would like to express my gratitude to Oslo Metropolitan University for two amazing years. The courses provided were highly relevant to my chosen career path. My deepest appreciation goes to my tutor and supervisor, Kyrre Begnum, for his outstanding lectures and guidance throughout the thesis. Thank you for diligently reviewing my work numerous times and offering valuable feedback. I hope our collaboration was as rewarding for you as it was for me. Thank you for volunteering to be my mentor; I would gladly repeat this experience.

I am grateful to my employer, Intility, for providing this opportunity and to the senior engineers who assisted me in navigating multiple platforms (Dynatrace, OpenShift, and Kubernetes), as well as for setting up a dedicated Prometheus server. Special thanks to Dani Wold Kristiansen for his encouragement and trust in the project's scope. Thank you for granting me the opportunity to enroll in the CKA and for covering the associated costs. I look forward to spending more time with you all.

Finally, I want to thank my friends for ensuring that I maintained a balanced life by engaging me in social activities. I am grateful for my family's pride and encouragement throughout this journey, and to my partner for motivating me to persevere and continue writing, even on the most challenging days.

Oslo, January-May 2023

Håkon Borgersen Ay

Contents

Abstract . . . . . **ii**
Preface . . . . . **iii**
Acknowledgments . . . . . **iv**
Contents . . . . . **v**
Tables . . . . . **viii**
Figures . . . . . **x**
Code Listings . . . . . **xiv**
Abbreviations . . . . . **xvi**
1 Introduction **1**
1.1 Research Questions 3
1.2 Objectives of The Paper 3
1.2.1 Results & Outline 4
2 Background **5**
2.1 Motivation 5
2.2 Cloud Computing 8
2.2.1 Vendor Lock-In 9
2.2.2 Skepticism and Concerns Surrounding Cloud Computing . . 11
2.3 Cloud Trends and Challenges 12
2.3.1 The Wasteful Consequences of Idle Resources 14
2.4 Container Orchestration 16
2.5 Kubernetes 18
2.5.1 Kubernetes Pods Resources: Requests 19
2.5.2 Kubernetes Pods Resources: Limits 19
2.5.3 Specifying Resources for a Container 20
2.5.4 CPU Throttling 21
2.6 Green Computing 24
2.6.1 Energy Consumption vs. PUE 27
2.7 Statistical Modeling for Resource Usage 28

2.7.1 Cross-Industry Data Forecasting . . . 31
3 Approach . . . 32
3.1 Phase I: Data Collection, Processing, and Modelling . . . 34
3.1.1 Processing a Single Container . . . 34
3.1.2 Data pre-processing . . . 35
3.1.3 Statistical Modeling . . . 36
3.2 Phase II: Evaluating Efficiency of Models Across Multiple Containers 38
3.3 Scope and Limits . . . 40
4 Results - Phase I:
Data Collection, Processing, and Modelling . . . 41
4.1 Data Collection . . . 42
4.2 Data Processing (PI-2a-c) . . . 46
4.2.1 Analyzing Memory Usage . . . 48
4.2.2 Analyzing CPU Usage . . . 54
4.3 The Statistical Models . . . 57
4.3.1 Exploring Strategies for Resource Optimization . . . 58
4.4 Linear Programming . . . 60
4.4.1 Evaluation of the Linear Programming Model . . . 63
4.5 Auto Regressive Integrated Moving Average (ARIMA) . . . 66
4.5.1 Determining the Differencing (d) . . . 67
4.5.2 Determining the Autoregressive (p) and Moving Average (q) 68
4.5.3 Fitting the ARIMA model . . . 69
4.5.4 Evaluation of the ARIMA Model . . . 72
4.6 Facebook's Prophet Model . . . 74
4.7 Summary: Evaluation of Models (PI-4c) . . . 78
5 Results - Phase II:
Evaluating Efficiency of Models Across Multiple Containers . . . 81
5.1 Model Performance Evaluation for Predictive Models . . . 82
5.2 Prophet + Tuned LP Model (PI-7ab) . . . 87
5.2.1 Introducing a Performance Score . . . 91
5.3 Summary: Addressing Research Question 2 . . . 97
6 Discussion . . . 99
6.1 Evaluating The Predictive Models . . . 100
6.1.1 Moving Forward with the KPIs . . . 102
6.2 The Impact and Limitation of the Datasets . . . 103
6.2.1 Training/Prediction of Datasets . . . 104

6.3 The Challenges of Predicting Containers . . . 105
6.4 Suggested Workflow for the Framework . . . 106
6.4.1 The Lack of Incentives for Reducing Resources . . . 107
7 Conclusion . . . 109
7.1 Future Work . . . 110
Bibliography . . . 112
A Survey Sent to Developers at Intility . . . 118
B Python Code . . . 124
B.1 Pre-processing Code . . . 124
B.1.1 Reading and pre-processing .csv files . . . 124
B.1.2 Converting memory to float . . . 125
B.1.3 Calculating skewness . . . 126
B.1.4 Script for showing daily memory usage during a 2 week period 127
B.2 Linear programming . . . 128
B.2.1 Calculating The MREI . . . 130
B.3 ARIMA . . . 130
B.3.1 ACF and PACF . . . 131
B.3.2 Fitting the ARIMA model . . . 131
B.3.3 Test Data Underprediction Rate (TDUR) . . . 132
B.3.4 MAE, MAPE and RMSE . . . 133
B.4 Facebook's Prophet Model . . . 133
B.5 P+LP . . . 134
B.6 Performance Score . . . 136

Tables

3.1 Implementation steps and activities during Phases I and II. The same abbreviations will be used as check markers throughout the results chapters 39

4.1 Comparison of initial and updated request/limit for c1-c3 memory usage (values in MB). The emphasized values demonstrate a significant increase in MREI for c2 and c3, while c1 experiences a slight decrease but also exhibits a substantial reduction in resource allocation. All percentages are rounded to the nearest whole number 65
4.2 Evaluation of ARIMA model performance for c1-c3 using MAE, MAPE, RMSE, and TDUR 73
4.3 Evaluation of Facebook's Prophet model performance for c1-c3 using MAE, MAPE, RMSE, and TDUR 78

5.1 Comparison of ARIMA and Prophet model performance in terms of TDUR and MAPE. Percentages are rounded to the nearest whole percentage. On each row, the best result for each evaluation method out of the two models is followed by an arrow pointing upwards. The table shows that the results are varying between the two models and that there is no correlation between the best result for TDUR and MAPE 83
5.2 Comparison of Original and New MREI for LP and P+LP. Each line represents the results of the models being run on the corresponding container. The table shows that P+LP is the model with the best MREI value while the LP algorithm has the highest reduction of memory request and limit. The upwards arrow display the best value for MREI and resource request between the LP and the P+LP 90

5.3 Comparison of performance scores across containers using original MREI values, LP method, and combined P+LP method. This table illustrates the improvement in performance scores after applying the LP and P+LP methods, demonstrating their effectiveness in creating a more sustainable Kubernetes environment by addressing both overallocated and underallocated containers 93

Figures

2.1 Survey Sent to Developers at Intility: A depiction of the responses to a question about Kubernetes resource allocation methods 7
2.2 The relationship between power consumption and CPU load percentage, ranging from 0% to 100% 14
2.3 Traditional deployment, virtualized deployment, and container deployment 17
2.4 Simplified Kubernetes Architecture 19
2.5 A depiction of a single-threaded application requiring 200ms of processing time to complete a request without any imposed limits 22
2.6 An illustration of a single-threaded application with a CPU limit of 0.4 23
2.7 Data center average annual power effectiveness (PUE) worldwide (2007-2022) 27

4.1 Visualization of memory request variations for 38 operational containers on Node01, with 14 distinct memory request lines displayed. All the lines demonstrate a constant memory request 44
4.2 Memory usage over time for c1, showcasing periods with missing data that account for small percentages (1-2%) 48
4.3 Memory usage plot for c1, displaying the data after incorporating mean values for missing data points 50
4.4 Comparison of c1's actual memory usage and the set resource request/limit, the distance between the solid red line and the solid blue line highlights the significant overallocation of resources 51
4.5 Memory usage distribution for c1, revealing slightly positive skewness and the decision against using standard deviation for evaluation PI-3d 52

4.6 Overlapping 24-hour memory usage segments for container01, the variations in each line illustrate the absence of clear hourly or daily patterns in resource consumption 54
4.7 CPU usage plot for c1, displaying the data after incorporating mean values for missing data points 54
4.8 CPU usage for c1 with set CPU resource request indicated by the dotted red line. The figure showcases overallocation since the requested resources are high compared to the actual CPU usage. The plot in the top right corner zooms in on the orange area and shows the erratic behavior of the c1's CPU usage 55
4.9 The graphic displays the potential increase in container capacity on node01 after optimizing the memory request for c1 from 736MB to 180MB. The diagram indicates that by applying a similar reduction to multiple containers like c1, the capacity for accommodating containers on the same node could potentially quadruple, expanding from 65 containers to 266 59
4.10 Visualization of c1's original and new memory request and limit. The zoomed-in plot shows that at one point the resource usage goes over the requested memory, but still stays comfortably under the limit 60
4.11 Visualization of c2's original and new memory request and limit. Notice the jagged memory usage and exceedingly low original memory request 62
4.12 Display of c3's original and new memory request and limit. In terms of fluctuations, c3 is more erratic than c1, but less than c2. Pay attention to the heavy overallocation on the original limit as well as the underallocation of the original request in the zoomed-in plot, much like c2 63
4.13 Order of differencing determination using autocorrelation plots and ADF test. Each column features the original data after differencing (top) and its corresponding autocorrelation (bottom). The font of the axes are small but the most important point is the display of the patterns 68
4.14 ACF and PACF plots for c1's memory usage 69

4.15 ARIMA plot for c1 illustrating the challenges faced due to an unfavorable train/test split and the model's struggle with upward trending memory usage and peaks 70
4.16 ARIMA plot for C2 highlighting the model's significant deviation from actual values, difficulty in capturing trends, and inability to predict the final peak, resulting in poorer performance compared to C1 71
4.17 ARIMA plot for C3 showing a decrease in low points frequency in the training set, leading to higher usage predictions 72
4.18 Prophet model's forecast for c1. The solid blue line indicates Prophet's forecast, black dots represent observations, and purple denotes test data. Prophets uncertainty is observed as the light blue shade. The prediction has a slight downward trend in contrast with the upwards trend of the test data. However, the uncertainty interval nearly captures the highest observations of test data 75
4.19 Prophet model's forecast for c2. The plot shows a cautious prediction line amidst split observations 76
4.20 Prophet model's forecast for c3. A dense appearance is observed due to outliers from start-up phase, with average fluctuation of around 5MB (between 60MB and 65MB) amidst erratic observations 77

5.1 Accurate ARIMA model prediction for c3's usage, demonstrating rare yet effective pattern recognition and resource allocation. TDUR of 66.4% and MAPE of 0.4% indicate satisfactory performance, despite minor deviations from the 95% target of TDUR 84
5.2 Significant deviation of ARIMA model prediction from test data due to a 25% drop, resulting in exceptionally poor performance (TDUR: 0%, MAPE: 4.2%). Illustrates the challenge in predicting and accommodating such usage patterns 85
5.3 Prophet model's accurate prediction for container c1, closely tracking memory usage with a TDUR of 59.5% and MAPE of 0.6%. Uncertainty interval captures most peak deviations, highlighting the model's effectiveness in predicting c1's usage 85

5.4 Sudden spike in training data contrasted with Figure 5.2, showcasing Prophet model's response through larger forecast fluctuations and expanded uncertainty intervals. Emphasizes the importance of considering such patterns when fine-tuning models, with TDUR at 100% and MAPE at 15.9% 86
5.5 A comparison of suggested resource allocation between the LP and P+LP models for container c8. The P+LP model effectively addresses the underallocation issue by setting a higher resource request based on Prophet's upward trend prediction. LP's request line is below all of the test data (solid red line), while P+LP's request is comfortably over c8's memory usage (solid green line) 89
5.6 Visual representation of Prophet's downward prediction of memory usage versus the upward trend of actual data for c18, and the successful adaptation of the LP and P+LP request limit to handle increased memory usage despite initial performance score discrepancies for P+LP 94

A.1 Survey Sent to Developers at Intility: Question 1 119
A.2 Survey Sent to Developers at Intility: Question 2 119
A.3 Survey Sent to Developers at Intility: Question 3 120
A.4 Survey Sent to Developers at Intility: Question 4 120
A.5 Survey Sent to Developers at Intility: Question 5 121
A.6 Survey Sent to Developers at Intility: Question 6 121
A.7 Survey Sent to Developers at Intility: Question 7 122
A.8 Survey Sent to Developers at Intility: Question 8 122
A.9 Survey Sent to Developers at Intility: Question 9 123
A.10 Survey Sent to Developers at Intility: Question 10 123

Code Listings

2.1 An example of a basic .yaml file for initiating an Nginx container in a Kubernetes environment, with specified resource requests and limits. The container requests 64 MiB of memory and 250 millicores of CPU, while the limits are set at 128 MiB and 500 millicores . . . 21
4.1 Prometheus setup for resource data retrieval with an example query for a single container on node01 . . . 44
4.2 Base plot for container resource data visualization with customizable elements . . . 49
4.3 Code snippet illustrating the 'plot_24_hour_segments' function, used to split and plot 24-hour segments for memory usage data . . . 53
4.4 Code snippet illustrating the addition of an inset zoomed-in view (orange square) to the CPU usage plot . . . 56
4.5 Implementation of linear programming to optimize memory request and limit values . . . 61
4.6 Code snippet for running the ARIMA model . . . 69
4.7 Request to Prometheus server . . . 74
5.1 Function that calculates the performance score Using the MREI value, resource request difference, and set ideal target, buffer and weights . . . 97
B.1 Reading and appending .csv files . . . 124
B.2 Converting memory to float . . . 125
B.3 Calculating skewness in container . . . 126
B.4 Script showing daily memory usage for a 2 week period . . . 127
B.5 The Linear Programming Model . . . 128
B.6 Calculating the MREI . . . 130
B.7 Determining differencing for the ARIMA model . . . 130
B.8 Calculating ACF and PACF for ARIMA . . . 131
B.9 Fitting the Arima model . . . 131

B.10 Calculating TDUR 132
B.11 Calculating MAE, MAPE, and RMSE 133
B.12 Facebook's Prophet Model 133
B.13 Finalized P+LP model, included plotting 134
B.14 Calculating Performance Score 136

Abbreviations

ACF Autocorrelation Function

ARIMA Autoregressive Integrated Moving Average

BH Black Hole

COVID-19 Coronavirus Disease of 2019

CSP Cloud Solution Provider

CPU Central Processing Unit

ELM Extreme Machine Learning

ESG Environmental, social, and governance

GCP Google Cloud Provider

KPI Key Performance Indicator

LSTM Long Short-Term Memory

MAPE Mean Absolute Percent Error

mCores Millicores

ms Milliseconds

MSE Mean Squared Error

MREI Memory Request Efficiency Index

OOM Out of Memory

OOME Out of Memory Exception

PACF Partial Autocorrelation Function

PUE Power Usage Effectiveness

P+LP Prophet Linear Programming

RMSE Root Mean Squared Error

SARIMA-KF Seasonal ARIMA with Kalman Filter

SDWF Self-Directed Workload Forecasting

SVR Support Vector Regression

TDNN Time Delay Neural Network

TDUR Test Data Underprediction Rate

VM Virtual Machine

Chapter 1

Introduction

We are progressively accepting that rapid changes in technology, industries, societal patterns, and new processes have come to stay in our society while expecting even more changes to come. As a result, it is crucial for individuals and organizations to adapt to these changes in order to stay relevant and competitive in today's ever-evolving landscape. Thanks to fast-moving technological concepts we were able to keep millions of people at work during the pandemic, deliver technology, and give access to the internet to people all over the world, thus making knowledge and worldwide communication even more accessible.

One key aspect of our society's digital transformation is the growing use of cloud computing. Cloud computing refers to delivering computing services such as storage, processing, and software over the internet (from data centers) rather than through local servers or personal computers. It allows users to access and utilize these services on demand, with the added benefits of scalability and flexibility, however, the ease of access leaves room for more responsibility in terms of handling and managing the resources given, and especially the requesting of resources.

Before cloud computing, organizations typically relied on in-house IT infrastructure, such as physical servers and storage devices to run their applications and store their data. This was known as the traditional or on-premise model. The major challenge with this model was the static investments of hardware combined with the over-allocation of resources to ensure that their systems could handle sudden spikes in demand. This often meant substantial upfront investments in hardware and infrastructure, and carefully calculated resource estimation.

Although cloud computing created a gold rush and largely solved these problems by allowing organizations to rent and use only the resources they need, when they need them, we are experiencing a lack of ownership of the resources at play. Now organizations can afford to over-allocate since they can scale down at any time without any consequences other than paying an increased price, which is low-effort to deal with compared with getting rid of on-premise resources. The increased resource usage is clearly noticeable when we examine the resource usage of data centers, which comes as a result of the swift cloud adoption of many organizations.

Today it is common knowledge that data centers consume a substantial amount of electrical energy. Data centers in the EU are using roughly 3% [1] of all electricity currently in demand. In other words, smarter and greener use of digital technologies will play a *key* part in limiting the carbon footprint of data centers.

The increasing demand for computing resources due to the digitalization of our society and the corresponding energy consumption of data centers has led to a growing interest in green computing. On one hand, we need to make use of technology that assists in deploying and managing more resources to keep up with advancements on the digital front. On the other hand, we need to reduce the environmental impact the data centers bring by proposing solutions and strategies that effectively utilize the resources in play. Among the many ways of reducing the energy consumption of data centers, one of them is to better estimate resource usage and be more precise with resource allocation, which will be how we approach this problem further in this paper.

So far, computing resources are talked about in terms of virtual machines (VMs), but we will investigate the resource usage of containers running on top of these virtual machines. The standard way of delivering software is through containers since it makes deploying and running applications more convenient. Containers request resources from the underlying VM, in other words, the VM acts as the subservient and the containers dictate the resource usage of the VM. Even though developers create and deliver the containers, thus specifying the resource requests, it still stands as a matter for the underlying infrastructure, which will normally be handled by the infrastructure developers. To manage the infrastructure consisting of multiple containers running on multiple nodes we are making use of container orchestration management tools such as *Apache Mesos*, *Docker*

Swarm, and *Kubernetes*. These tools play a crucial role in modern software development, making it possible to deploy, scale, and manage large-scale applications and services with ease.

1.1 Research Questions

By using historical data and machine learning we can anticipate the resources a deployment will be needed, thus hitting the nail on the head when allocating resources to workloads. From this, two problem statements emerge:

Research Question 1 *How can time series forecasting models be applied to predict resource usage for individual containers in Kubernetes environments, and what are the challenges and benefits associated with using these models to ensure prediction accuracy across various container usage patterns?*

Research Question 2 *How can we effectively evaluate the impact of predictive models on resource allocation strategies in Kubernetes environments?*

Time series forecasting models refer to well-established statistical approaches that will be explored in the background chapter. Determining how these models can be applied to data gathered from containers within a Kubernetes environment remains to be seen. It is anticipated that predicting container behavior may present challenges, and analyzing the advantages and drawbacks of various models will be intriguing to investigate. Further fine-tuning of these models to ensure improved predictions will be a subsequent step.

Evaluation will constitute a significant portion of the thesis, requiring a thorough assessment of the forecasting models. As part of this process, appropriate evaluation strategies must be deliberated. Besides evaluating the models themselves, it will be crucial to assess the implications of implementing the proposed strategies within a Kubernetes environment.

1.2 Objectives of The Paper

Investigating and optimizing resource usage in Kubernetes environments is essential for improving the sustainability and efficiency of cloud computing infrastructure. The problem this thesis aims to address is the development of a systematic approach to analyze, model, and predict resource usage in Kubernetes projects,

with a focus on CPU and memory utilization. This will involve acquiring and preprocessing relevant data, identifying patterns and anomalies, and implementing statistical models to predict resource usage accurately. The ultimate goal is to enhance resource allocation policies and reduce over-provisioning, resulting in more sustainable and efficient Kubernetes environments.

1.2.1 Results & Outline

In this study, we conduct a comprehensive analysis of real data collected from containers operating in a real-world production environment. We utilize well-established algorithms to examine and forecast resource usage and present an experimental design accompanied by evaluation strategies to assess the effectiveness of our approach. Although predicting memory usage proves challenging, our findings demonstrate the merit of combining predictive model outcomes with other algorithms to recommend resource allocation for containers.

Chapter 2

Background

The field of computing has undergone significant transformations in recent years, with the advent of new technologies and trends shaping the industry and academia. The purpose of this background chapter is to provide a comprehensive overview of the relevant concepts and technologies that form the foundation of our research. We will discuss the motivation behind this study, the basics of cloud computing, the significance of green computing, the latest trends in the industry, the importance of container orchestration, the key features of the popular container orchestration system, Kubernetes, and the role of statistical modeling.

This chapter serves as a primer for the reader, providing a broad understanding of the context and relevant technologies that will be discussed throughout the rest of the report. By clearly understanding the background information, the reader will be able to appreciate the significance and impact of the research.

2.1 Motivation

Before beginning the literature review and detailed descriptions of concepts, tools, and technologies used in the project, it is appropriate to give a brief glimpse of the road taken before this project started.

Docker and Kubernetes got introduced to us during the first year of the ACIT master's program. Container technology and orchestration management systems were hard to grasp at first but after spending weeks debugging a Kubernetes cluster, the technology and using Kubernetes as an orchestrator became more transparent. Getting configuration management systems to work as intended is highly satisfying. After applying for Intility's master thesis proposal on *awareness of Kubernetes resources*, the die was cast.

Several prominent yet typical issues within Intility's Kubernetes cluster centered around the fact that some pods consumed excessive memory before shutting down and restarting. A Kubernetes pod represents the smallest deployable unit in the Kubernetes system, composed of one or more containers that share the same network namespace and storage volumes. Each container requests CPU and RAM, either a default value or a specified one. Although pod restarting is standard practice for Kubernetes, it raised the question as to why this occurred so frequently, which led to seeking permission to examine metrics related to CPU and RAM usage. During this investigation, it was observed that many pods appeared to be over-provisioned, utilizing only a small portion of the requested resources. In summary, the findings were as follows: on the one side, there were pods restarting due to insufficient resource requests, while on the other side, there were pods using merely 10-20% of the allocated resources and not restarting. When inquiring with colleagues about this phenomenon, the general response provided was along the lines of:

"*It is difficult to accurately predict the resources a deployment will require. To ensure the smooth operation of the application, it is better to allocate more resources than needed. In the event that our current node reaches its limit, we simply request a new one from the cloud infrastructure team. Although performing our job should be our main focus, it would be advantageous to develop the expertise to request and allocate appropriate resources in a Kubernetes environment to optimize performance and achieve cost savings*".

The initial statement suggests that accurate resource allocation has been relegated to be a desirable but unattainable feature due to the intricate nature of resource prediction and the lack of opportunity from a developer's standpoint. Furthermore, their response suggests that it is challenging to accurately forecast the resources required for a deployment, but at the same time, they understand the advantages and the value of doing so if it were possible.

Fascinated by the response, a survey was created using Google Forms with anonymous replies and sent to 90 developers at Intility through an email group. Refer to Appendix A for the survey description, along with all questions and responses. This survey was designed and employed as a means to gain a deeper understanding of the company developers' perspectives on these issues. With a

response rate of 10%, or 9 developers, the turnout was relatively low. However, it still provided valuable insights, as some of the answers to the questions displayed noticeable similarities.

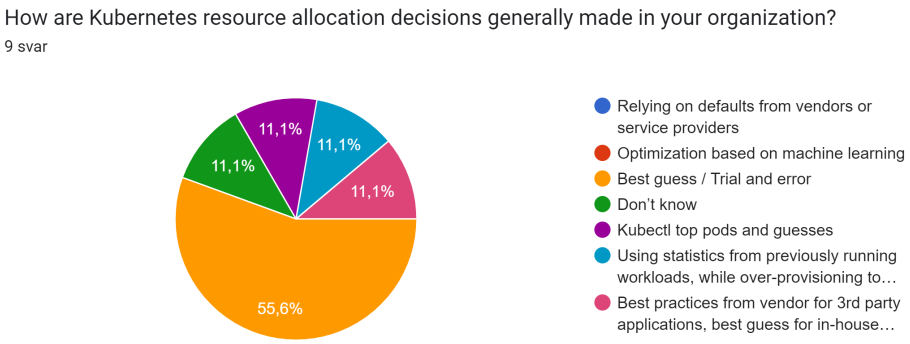

Figure 2.1: A depiction of the responses to a question about Kubernetes resource allocation methods, revealing that 5 out of 9 participants rely on "best guess" or "trial and error" approaches, indicating the challenge of determining the appropriate amount of resources required for optimal performance

One of the more interesting findings was a question about how the Kubernetes resource allocation decisions were made. In Figure 2.1 we see a question asking how Kubernetes resources usually are allocated, and 5 out of 9 stated that they are using *best guess* or *trial and error* as main approaches either personally or have the impression that it is done this way throughout the organization. In other words, the difficulty lies in figuring out how much resources are needed. If they knew this, we can assume with some confidence that more correct resource requests would take place. The survey conducted during the project, aimed at mapping the resource allocation trends in Kubernetes environments, is located in the Appendix A.

The need for a more efficient approach to specifying requested resources prompted an investigation into workload profiling and statistical modeling techniques, aimed at determining the optimal resource requirements for containerized environments. However, before heading straight to the root of this problem, some further research to get a broader picture of why this is a problem, how common it is, and how to estimate the usage of CPU and RAM better, had to be examined. With that, we conclude the starting point of our project and move on to the next part of the chapter.

2.2 Cloud Computing

The cloud era and the technological advancements within cloud computing are well accepted, and as we briefly mentioned in the introduction chapter, the advancements created a *gold rush* for organizations and largely solved the problems that an in-house IT infrastructure brought with it. The proverbial *buffet had opened*, and companies who had the capacity to and were comfortable with transitioning to using cloud solutions did so [2]. The idea promoted by cloud solution providers was that businesses should invest in new technology, specifically in the cloud. Slogans such as "*Accelerate your digital transformation*" and "*Transforming Businesses and Shaping Innovation*" were, and are still, popular among cloud solution providers [3–5].

Cloud computing fundamentally changed how computing resources were acquired. Before the cloud era, one could defend buying powerful and expensive servers by making sure that the unused resources of the hardware got used by other applications or services, and thus the organizations often overestimated their needs to make sure they were set for the years to come [6]. The over-allocation was done for multiple sets of reasons. They had to have enough resources for the future to make sure their applications would have sufficient resources with their intended growth taken into consideration. Ordering more frequently would be too time-consuming, because each time they had to get approval from management, thus providing detailed reports and numbers explaining why resources needed to be expanded. The same mindset followed the organizations that had been thinking like this for many years into the cloud computing paradigm. In other words, they were buying cloud resources as if they were buying physical hardware, thus over-allocating resources when there was no real need to do so.

While companies continue to acquire more resources than necessary, the degree of excess has been substantially diminished, leading to better energy efficiency in data centers. The consolidation of computing power within data centers has contributed to increased sustainability, as these facilities are highly proficient in managing servers. Not shifting the computer power to the cloud would have led to a very large portion of global electricity being consumed by data centers [7].

The use of cloud-native technologies in application development presented new opportunities. By building cloud-native applications, with the cloud in mind,

it became possible to implement dynamic scaling and disposable VMs or servers as a design principle. This approach ensured that cloud-based services were used with a focus on flexibility and cost-effectiveness, rather than vendor lock-ins and long-term commitments.

2.2.1 Vendor Lock-In

In the context of the video gaming industry, there is a historically prevalent pattern where consumers invest significantly in a particular console through the purchase of an extensive collection of games and accessories specifically compatible with that platform. The unique experiences each console provides, including high scores and multiplayer battles, foster a sense of attachment and loyalty. Consequently, when another console model with distinct features is introduced into the market, consumers often find the prospect of abandoning their accumulated collection and adapting to a new platform daunting. This scenario is a representation of vendor lock-in, a sophisticated and enduring business strategy deeply rooted in both financial and emotional considerations.

This same strategy is observable in the contemporary era of cloud solution providers (CSPs). Businesses, analogous to gamers and their consoles, tend to invest heavily in a specific CSP, becoming accustomed to its system and deeply integrating it into their operational processes. The concept of transitioning to a different provider, even if it offers unique features or advantages, often presents an intimidating prospect, much like the transition to a new gaming platform. Therefore, businesses often remain tethered to their initial provider not due to a preference for their services, but because the perceived challenges and costs associated with change are considered too formidable to undertake. This underscores the effectiveness of vendor lock-in as a strategic tool in the digital age, transcending industries, and technological advancements.

This has been a concern in cloud computing since the early days, and as the popularity of the technology increased among businesses, the discussion about vendor lock-ins grew [6]. Cloud solution providers (CSPs) offer a wide range of services that allow businesses to run their applications and store data in the cloud. However, each provider has its own set of tools, APIs, and technologies which can make it difficult to switch to another provider without significant effort and cost.

For example, Google Cloud Platform (GCP) is known for its strength in data analytics and machine learning, with services such as BigQuery, Cloud Dataflow, and TensorFlow. GCP is also known for its focus on sustainability, with a commitment to achieving carbon neutrality for its data centers and operations, as we will see more of in Section 2.6. If your business is using many of these technologies, and their ESG[1] rating is dependent on pursuing carbon neutrality and they are expected to provide clear and concise data of their total energy consumption, it could prove to be hard to find another CSP that matches the tools and their vision. Another example is Microsoft Azure, which is known for its strong integration with Microsoft's other products and services, such as Office 365 and Active Directory. For developers, this means a shared codebase and APIs which leads to easier development. For other employees, this seamless integration brings *single sign-on* with the same user credentials for all Microsoft services, which simplifies the user experience and improves security. This is all managed from a service called identity management that Azure Active Directory provides. Removing this seamless integration by switching providers would result in quite the migration for the developers, and the user experience would be sure to decrease.

As a result, businesses that adopt a particular cloud provider's services may find themselves locked into that provider's ecosystem, making it difficult to switch to another provider or move their applications and data to an on-premises environment. This can be a significant risk for businesses, as it can limit their flexibility and increase their costs.

To mitigate the risk of vendor lock-ins, businesses can invest in multi-cloud strategies to reduce the dependency on any single provider's technology, pricing, or service-level agreements. By using multiple CSPs the organization should be more flexible and have more experience with different solutions and generally have more expertise in the field since they have to translate knowledge between the applications residing in different environments. This approach naturally includes emphasizing portability along with keeping data separate from each provider or by using third-party providers.

Another approach to reducing the risks of vendor lock-ins along with increasing scalability, resilience, and flexibility is to introduce cloud-native technology to

[1] An ESG rating evaluates a company's commitment when it comes to environmental, social, and governance (ESG) issues, and how proactively the company manages ESG issues that are most relevant to its business

the stack. Cloud-native technology is a set of principles, practices, and technologies that enable organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. These cloud-native projects are created with the cloud in mind and make it seamless to work within multiple cloud environments. Some examples of this are Infrastructure as Code (IAC), Containers, and Kubernetes. We will come back to these in Section 2.4. While using cloud-native technology can avoid vendor lock-ins, it also eases development for software developers since the same principles and code translates well into other environments, opening up possibilities for using multiple providers. Containers help with environment configurations and dependencies, while Kubernetes creates a layer of abstraction between the cloud and their application. In conclusion, organizations use cloud native technologies to ease development between environments and different CSPs, thus reducing the impacts of vendor lock-in and creating abstraction layers between the clouds and the applications.

2.2.2 Skepticism and Concerns Surrounding Cloud Computing

While cloud computing has gained widespread acceptance and adoption, there are people and organizations that remain skeptical or wary of its potential benefits. Their concerns often revolve around security, privacy, reliability, and the loss of control over data and infrastructure.

One of the primary reasons for skepticism towards cloud computing is the potential risk to security and privacy. Storing sensitive data on remote servers managed by third parties can raise concerns about unauthorized access, data breaches, and cyberattacks. High-profile incidents like the 2017 Equifax breach [8] or the more recent AWS data breach in 2022 [9] have fueled these concerns and generated doubts about the security of cloud-based systems.

Another common concern is the loss of control over data and infrastructure when migrating to the cloud. Organizations may worry about the reliability and performance of their applications, as they become dependent on the cloud provider's infrastructure and uptime. Outages affecting major cloud providers, such as the Amazon Web Services (AWS) outage in 2017 [10], have underscored these concerns and highlighted the potential risks of relying on third-party infrastructure.

As discussed in the previous section about vendor lock-ins, considering availability and having a cloud exit strategy in mind are important concepts to withstand dependency and reduce the potential risks of relying on third-party infras-

tructure.

Lastly, we need to address the environmental impact of large-scale cloud infrastructure and the resource depletion and electronic waste that come with it. While data centers greatly reduce the carbon footprint when compared with a multitude of traditional in-house infrastructures, which we will come back to later in this chapter, we still need to address the environmental impact of their operations.

While these concerns are important, we used this sub-section mainly to include these perspectives to give a better picture of some concerns with cloud computing. These findings will not be further considered in the project as it does not influence the project in any way, since we are not examining security, privacy, the loss of control, and reliability of the data centers. However, the last point, regarding the environmental impact is a crucial part of this project and will be revisited in Section 2.6.

2.3 Cloud Trends and Challenges

Many tech companies are incorporating cloud technology into their strategy to stay competitive in today's market. By leveraging the cloud, businesses believe they can achieve growth, respond quickly to changes, increase revenue, and meet their objectives. Cloud usage has been growing for several years and was accelerated even further by COVID-19 (Coronavirus Disease of 2019). During the pandemic, many businesses were forced to shift to remote work, and CSPs provided a flexible and secure way for employees to access company resources and collaborate with each other. In the UK in April 2020 46.6% of people in employment worked from home, and of those who did some work from home, 86.0% did so as a result of the coronavirus [11]. There were also other factors responsible for the rapid acceleration during the pandemic, such as an increase in demand for online services and e-commerce which also got reflected well in the stock market as investors recognized the growth potential of these businesses [12]. Lastly, the uncertainty and unpredictability made organizations prioritize agility and scalability, which the cloud is designed to provide.

To further examine different trends according to employees, let us take a look at some surveys done in the field.

Flexera, a computer software company that is a part of IBM's partner ecosystem and has been around since 1988, recently published the eleventh edition of their *State of The Cloud Report* (previously known as the RightScale State of The Cloud Report) [13]. The report explores the thinking of 753 respondents from a survey done worldwide in late 2021. The respondents worked in organization sizes between 1-100 to 10,001+ employees, and in a broad range of industries, such as healthcare, education, and retail to name a few. The roles of the respondents were diversified between business, enterprise architect, development, cloud architect, and IT/OPS. The company's cloud usage was split into levels of light, moderate, and heavy, based on the respondents' answers.

This report highlights a massive uptake of cloud usage among small to mid-sized businesses, having increased their spending by about 38% when compared with the previous report from 2021. The article also mentions that some of the top challenges, which were similar to last year, were assessing on-premise vs cloud costs (44%) and rightsizing/selecting the best instance (42%). Both of these challenges shed light on our problem statements, as solving those will make it more clear to assess costs and especially rightsize the instances. Another interesting highlight is that the third-party tools that assist in areas, such as orchestration and container management, are losing ground to native tools from the cloud providers themselves. This information pushes our project towards having more focus on examining the use of native tooling to solve our problem statement if we were to go in that direction.

In a survey done by StormForge in 2021, [14], 134 IT professionals with knowledge of their organization's cloud spending got surveyed. The key findings showcase that on average an estimated 47% of all cloud spending is wasted. Also, cloud spending and cloud waste are increasing for 75% organizations. Finally, Resource optimization and efficiency are high priorities for 75% of organizations. Organizations agree that cloud resources are being wasted, and they want to optimize their resource usage. However, when asked why optimization is not a higher priority, 37% replied that optimization is too difficult. The findings are similar to the findings from the survey conducted on Intility, which can be found in Appendix A. In this survey, 66% responded that cloud complexity makes it hard to estimate how many resources are needed.

2.3.1 The Wasteful Consequences of Idle Resources

From the reports read so far, one new report from March 2023 from sysdig.com [15], and the survey created and conveyed to Intility A, we see that overallocation is a normal thing to do when developers have an urge to scale quickly, little knowledge of the containers to be run, lack of Kubernetes knowledge, and lack of capacity planning. The study from *sysdig* shows that 49% of the reported containers have no memory limits set and 59% have no CPU limits set. For requests on the other hand, we see from the study that as much as 82% of requested CPU and 30% of request RAM may go unused, depending on the number of nodes. The report from sysdig concludes that companies with more than 1000 nodes could reduce their wasted resources by $10M per year.

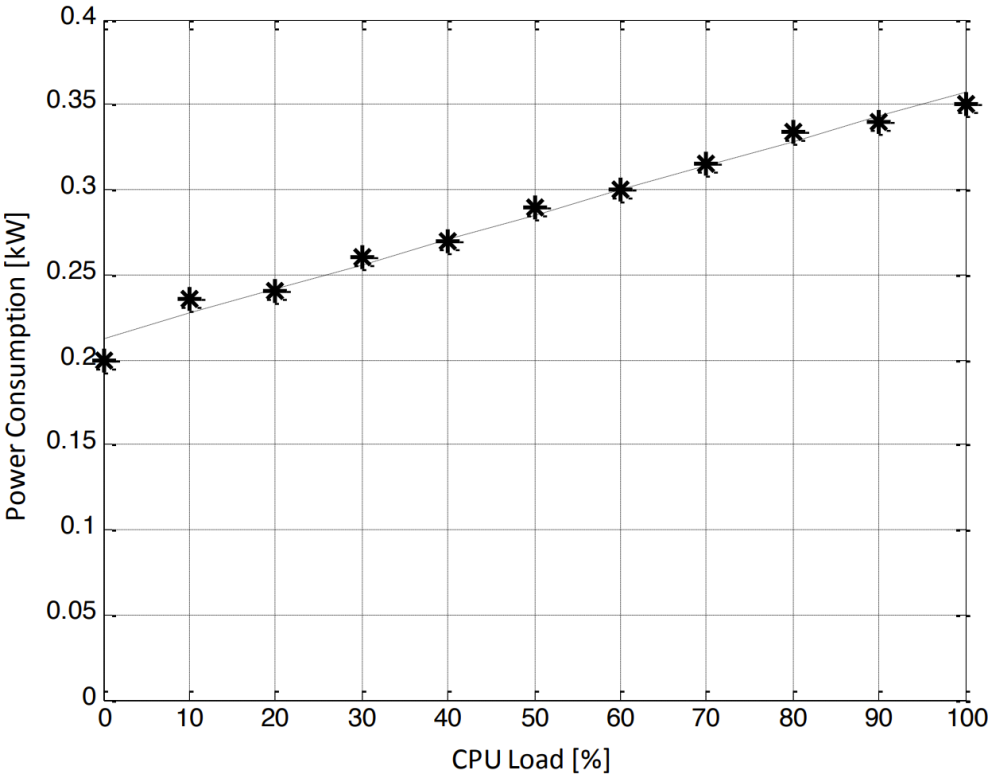

Figure 2.2: A graphical representation of the relationship between power consumption and CPU load percentage, ranging from 0% to 100%. The data demonstrates that even at 0% load, the server consumes 57.14% of its power utilization compared to a 100% load [16]

In a paper from Sarji et al. [16] a server's power utilization was tested under different load percentages. They concluded that with 0% load, the server is still consuming 57.14% of the power when compared to being at 100% load. The pa-

per examines various scenarios in which virtual machines should either be turned off, put into sleep mode, or left running during periods of low CPU utilization. In this scenario, there are multiple parameters that impact the decision to be done for the VM, including but not limited to, how often the server is idle, the time it takes to shut down, and the energy required to turn on the server. Figure 2.2 shows the power consumption together with the CPU load percentage going from 0% to 100%. The measurements were done on a Fujitsu Siemens TX300 S2 server with two cores and 2GB of RAM. Most servers in Intility's data centers are of type HPE DL380 with 36 cores and 1024 GB RAM. These servers have an idle wattage of 290, and a production wattage of 420, which would give a similar curve as seen in Figure 2.2. The former server was released in 2005. It goes without saying that there has not been much change in the electricity consumption of idle hardware. When we consider these numbers, we understand that idle hardware can play a huge factor in total energy consumption.

What happens when applications receive more resources than necessary? Imagine a server with two cores and 2 GB of RAM. There is a critical application that the developers, based on their previous experience with similar containers, expect to use approximately 1000 millicores (mCores) and 1000 MB RAM on average. However, the actual average usage is 500 millicores and 500 MB. The developer's main priority is to ensure high uptime for the application. Although they possess some knowledge of Kubernetes, they may not be experts in capacity planning. Consequently, the container is allocated three times the resources it usually needs, equating to 1.5 cores and 1.5 GB RAM, to accommodate potential surges in demand. From the developer's perspective, this appears to be only a 50% increase in their estimated resource requirement. While this is considered an overallocation of resources from a broader standpoint, the developers decide that it is crucial to maintain high uptime for this essential application.

Assuming a situation where another application with a similar workload is to be deployed, the initial server has only 500 mCores and 500 MB RAM available, necessitating the launch of a new server for resource allocation. With each application using approximately 500 mCores and 500 MB RAM, about 66% of the assigned resources remain unused. Referring to Figure 2.2, it is observed that idle hardware consumes around 57% of the resources compared to a fully utilized server, indicating inefficiency in the current configuration. Allocating double the requested resources, as opposed to triple, could have enabled the accommodation

of the second application on the same server, reducing idle hardware and potentially allowing for a server shutdown. A calculation to estimate power consumption in both scenarios was conducted using data from Figure 2.2. As demonstrated in Equation 2.3, power consumption can be reduced by 44.23% by minimizing idle hardware.

$$Scenario1 : 0.26kW + 0.26kW = 0.52kW \tag{2.1}$$

$$Scenario2 : 0.29kW + 0.00kW = 0.29kW \tag{2.2}$$

$$KwReduction = \frac{(0.52kW - 0.29Kw)}{0.52kW} * 100\% = 44.23\% \tag{2.3}$$

In reality, the servers will often consist of multiple different applications and workloads. Some applications move together and peak together, while others will be completely stale. Critical applications will need to be made sure to have enough resources at all times, while non-critical applications may have some room for high latency at certain times. Unique workloads, criticality, and all that are unknown make making decisions regarding resource allocation a much more complex task than the given example. However, the point stands, it should be possible to reduce power consumption by being more conservative with the allocation of resources.

2.4 Container Orchestration

Using containers and container orchestration tools is a promising approach to reducing the environmental impact of computing. Containers are a solution to reliably running software in any computing environment. The term and technology called containers arrived from cargo ships that transport freight containers from one place to another. Just like cargo ships that have multiple stacks of containers, computing environments can have multiple containers that are treated equally without paying attention to their contents. Containers are transported or *run*, and they operate independently as long as they are on the ship. This is in contrast to deploying software without encapsulating the environment and dependencies, which can be inefficient and wasteful. By using containers, resources can be optimized, and energy efficiency can be improved, much like how cargo containers on a ship are efficiently transported to their destination.

Figure 2.3 shows how we have gone from traditional deployment to container deployment and their underlying features. The containers are similar to Vms, hav-

ing their own filesystem, and share of CPU, memory, and process space. However, they are still considered lightweight, since they share the operating system among the applications. One thing to notice in Figure 2.3 is that due to the operating system and the virtual machine not being a part of the application deployment, see the grey isolated boxes, we can separate the concerns that developers and system administrators have. In other words, the applications are completely decoupled from the infrastructure.

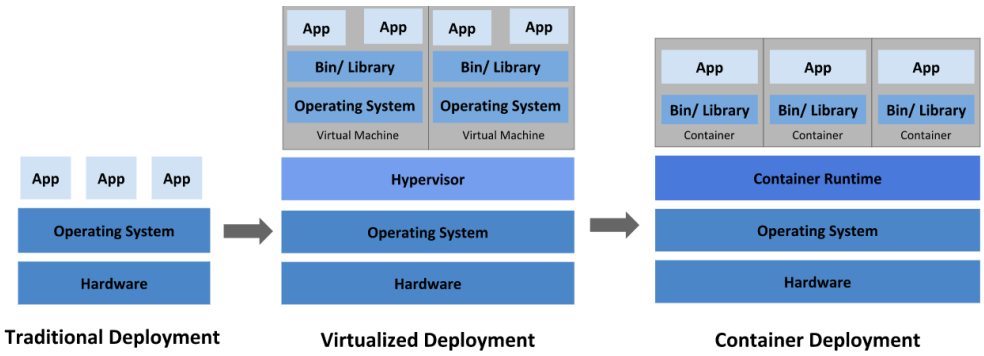

Figure 2.3: Illustration of the transition from traditional deployment to container deployment, highlighting the key features of containerization. Containers, akin to VMs, possess their own filesystem and a share of CPU, memory, and process space, but remain lightweight by sharing the operating system among applications [17]

In light of the new containerization paradigm, the need for automated management, scaling, and maintenance of containers increased in importance. Another word for this is *container orchestration*. The most common orchestrator is Kubernetes, which is the *de facto* standard at the time for managing containers. We will be exploring more of Kubernetes and more of its features later in this chapter's next section.

Containers and orchestration tools offer numerous benefits, but managing them can become challenging when deployments grow rapidly and cluster complexity increases. It can be difficult to track the resource needs of containerized applications due to the sheer number of deployments and their varying services or functions. On top of this, Kubernetes and its pods add another layer to already virtualized environments which were over-committed too. This situation can lead to resource inefficiencies such as resource leaks and over- and under-allocation, which we now should be familiar with. However, to emphasize the importance of those two situations in our paper we will address them once more. Over-and under-allocation can appear when the resource needs or workload of an appli-

cation is not well understood, either due to high complexity or the lack of data, as we have seen reported in several surveys from Section 2.3. This may lead to resources being allocated inappropriately, either providing too much or too little for an application to function efficiently. Resource leaks, or *zombie resources*, are forgotten deployments that continue to run even when they are no longer needed, thus consuming valuable resources.

Managing resources efficiently as applications and services grow in complexity is typically a major challenge for system administrators. There is already enough to do just to ensure the stability and performance of the applications, and adding balancing the need for optimal resource usage on top, could be challenging. If capacity planning and optimization could be done in an efficient manner, we could more easily identify opportunities to reduce over- or under-allocation and adjust resources accordingly.

2.5 Kubernetes

In this section, we will briefly introduce a more detailed view of Kubernetes and its underlying components. In the subsection, we introduce an important concept regarding resource usage of Kubernetes pods, which aligns well with the project. This information is gathered from the official Kubernetes documentation [17] but compressed to only include the parts of the framework that are needed to further follow the thesis. We will therefore not go into detail on the Kubernetes control plane and its components, such as etcd, kubelet, kube-proxy, scheduler, etc.

Figure 2.4 does a great job explaining a simplified Kubernetes architecture. The nodes are machines or virtual machines that have a set amount of resources available. Each time a pod gets deployed, the scheduler decides what node the pod shall be deployed on if not anything else is specified in the configuration of the pod. The Kubelet, which is responsible for spinning up deployments on each node, only deploys the pod after the info regarding the new pod has been sent and documented to the API server which updates the etcd database. If everything is good so far, the new pod starts up in its designated node.

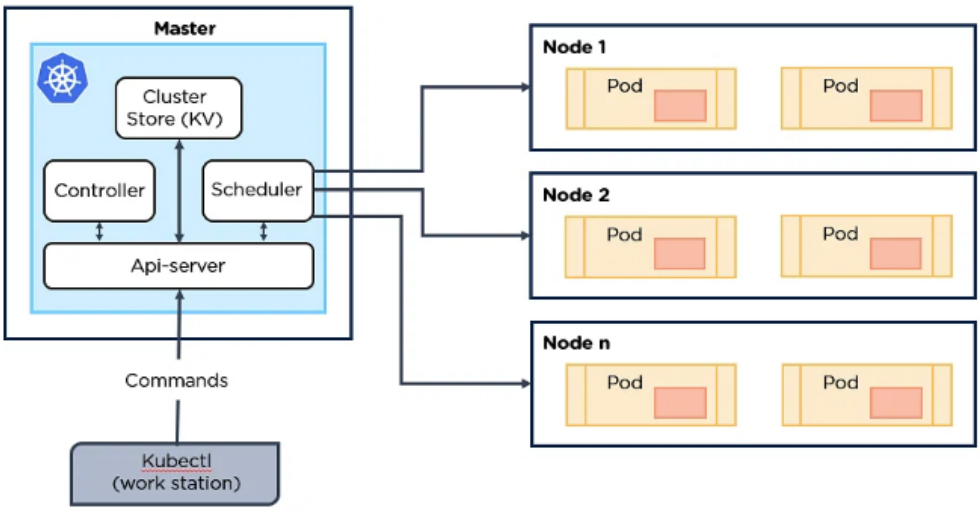

Figure 2.4: A simplified representation of the Kubernetes architecture, illustrating the process of deploying a pod. Nodes, which are machines or virtual machines with allocated resources, are assigned pods by the scheduler that is located on the master node [18]

2.5.1 Kubernetes Pods Resources: Requests

While the node has a set amount of resources, the pods do not necessarily have a static amount of resources. This can be set in the declarative yaml file, also called manifest. In the manifest one can specify the amount of CPU and RAM that should be requested for the containers within the pod, this essentially means the number of resources that will be reserved only for the respective pod's usage. A pod can for example request 1GB of RAM and only use 200MB, thus keeping an additional 800MB of RAM unavailable for other pods in the same node. The same goes for the CPU. It is the Kubernetes Scheduler's job to deploy the pod on a node that can satisfy the request being made by the pod. In this case, if a node does not have 1GB of unrequested RAM, the pod will not be scheduled on that node. It is important to note that the pod's resource usage (while running on the node) may exceed the requested resources and will only be limited if a resource limit is set.

2.5.2 Kubernetes Pods Resources: Limits

In addition to resource requests, Kubernetes also allows the pods to have resource limits. Limits decide how far the pod can stretch out its resource usage and is not to be confused with the resource request. A container that exceeds the memory limit set for the container will be forced to enter a "CrashLoopBackOff" state which

essentially means that it is shut down. This is known as the pod being killed by going Out of Memory (OOM). Depending on the restart policy defined for the deployment or pod, i.e., "Always", "OnFailure" or "Never" the pod will follow perform the actions based on the restart policy. The default value is "Always". The same goes for the CPU, but instead of shutting down the container when the requested limit is reached the processes will start to slow down, lowering the CPU usage and increasing the latency of the running processes. This is called CPU throttling and will be a feature we will need to take into account and discuss during this project. This happens since the set limit determines the maximum number of milliseconds (ms) the container can use the CPU at a given time. To make sure to that the container is within its limits it will throttle the container instead of shutting it down. [19].

2.5.3 Specifying Resources for a Container

In Code Listing 2.1, a basic .yaml file example is presented, which, when executed in a Kubernetes environment, would initiate an nginx container. This container requests 64 Mebibytes[2] of memory and 250 millicores of CPU. Additionally, the .yaml file sets limits at twice the requested values, specifically 128Mi and 500m.

While it is not mandatory to define resource requests or limits in a standard Kubernetes environment, doing so enhances performance, stability, and predictability within the cluster. If not specified otherwise or if restrictions are not imposed on the Kubernetes environment, a developer has the flexibility to remove memory and/or CPU from the *requests* specification, and memory and/or CPU from the *limits* specification, or all four specifications altogether.

In a case where the requests is not specified, Kubernetes assumes default values for the resources. When a pod doesn't have resource requests defined, it is considered as having zero resource requests (i.e., no guaranteed resources). This means the pod can still be scheduled, but it won't have any guarantees about the CPU and memory resources allocated to it.

In such a case, the pod may be scheduled on any node with available resources, but the scheduler won't reserve any resources specifically for the pod. As a result,

[2]In the context of Kubernetes, we use Mebibytes (MiB) rather than Megabytes (MB) due to the binary prefix standard. 1 MiB equals 2^{20} bytes (1,048,576 bytes), while 1 MB equals 10^6 bytes (1,000,000 bytes)

the pod may compete for resources with other pods on the same node, and if the node becomes resource-constrained, it could lead to resource contention, affecting the performance or stability of the pod and the node.

In addition to adding *resources* to the .yaml file, more *specs* can also be added to the container such as labels, ports, and replicas to name a few. For the full list of specs and other configurations that can be added to manifests, visit the Kubernetes documentation [17].

Code listing 2.1: An example of a basic .yaml file for initiating an Nginx container in a Kubernetes environment, with specified resource requests and limits. The container requests 64 MiB of memory and 250 millicores of CPU, while the limits are set at 128 MiB and 500 millicores

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver1
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Although resource requests and limits are not enforced on Kubernetes pods, the concept of when to include requests and limits is an ongoing discussion. However, in most cases, it is suggested that you should include requests and limits on memory usage. While for the CPU it is generally recommended to include requests, but to only use limits in certain circumstances [19–21].

2.5.4 CPU Throttling

Both OOM and CPU throttling are delicate matters to handle and how this will be dealt with for memory and CPU usage will differ. While we generally want to reduce the chance of pods getting killed due to the consumption of too much memory, we still want to keep the requested resources as low as possible to fit more pods into the node.

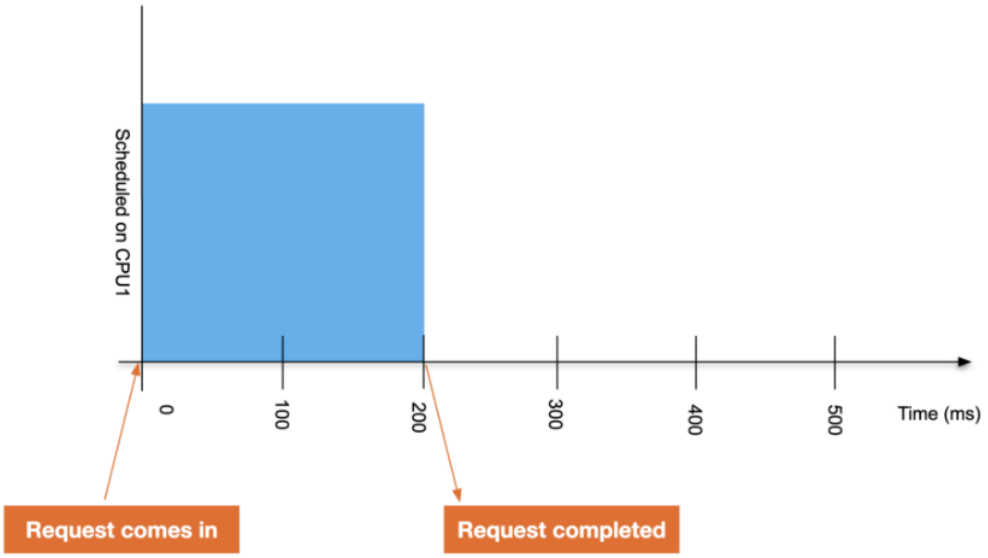

Figure 2.5: A depiction of a single-threaded application requiring 200ms of processing time to complete a request without any imposed limits [22]

With CPU throttling, which is a technique used to manage a system's performance and prevent it from overheating, the processes will slow down instead of restarting the container. Since the CPU limit decides how many ms the container shall have of the available cores on the node, nearing the limit will reduce its share of the CPU, while still processing the application. To further explain this concept we have included two figures that were provided in a blog post on "indeed engineering" [22]. In Figure 2.5 we have a single-threaded application that needs 200ms of processing time to complete a request. Without limits, it would take 200ms for the application to finish its process as one would guess. However, if we set the CPU limit to 0.4 CPU, which would mean that the application will get 40ms of runtime for every 100ms period, the 200ms request would now take 440ms as shown in Figure 2.6. We can calculate this using Equation 2.4, where T = Processing time without limits, R = actual runtime in each period (Derived from CPU limit), P = period duration, and N = number of periods **including a partial period** (T/R).

$$totalProcessingTimeWithLimits = N * P - (P - R) \tag{2.4}$$

$$5 * 100ms - (100ms - 40ms) \approx 440ms \tag{2.5}$$

Equation 2.4 is based on Figure 2.6 where the chosen process is run at the start

of every 100ms period, but in reality, the process can be run whenever during the 100ms time period, and therefore we can use "the fifth period" instead of 440ms. However, since this is highly conceptual, we chose to withdraw the partial period.

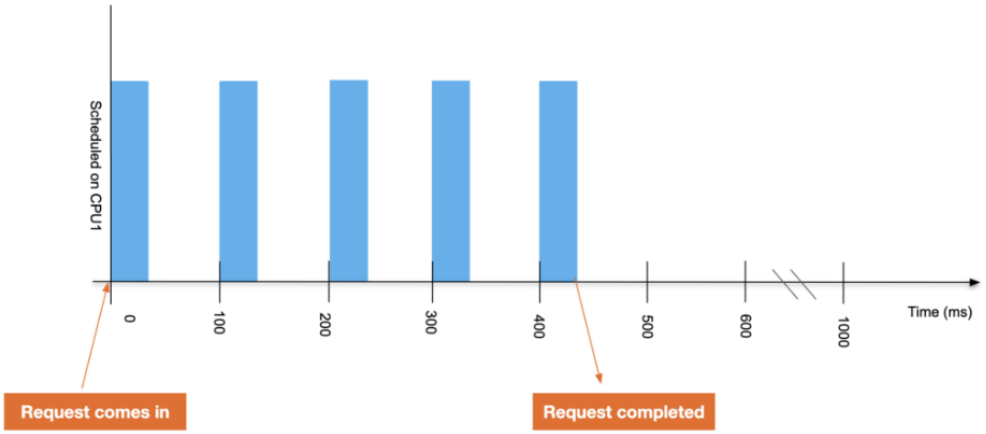

Figure 2.6: An illustration of the same single-threaded application with a CPU limit of 0.4, resulting in 40ms of runtime per 100ms period, causing the 200ms request to take 440ms to complete [22]. The impact of CPU throttling on application processing time can be calculated using Equation 2.4

The blog post from "indeed engineering" [22] presents a bug related to CPU throttling which was introduced by the v4.18 release of the Linux kernel in a commitment that was supposed to fix inadvertent throttling [23]. The issue was that normal CPU usage and high throttling appeared simultaneously, which should not have been possible. The bug has now been fixed.

Another post, originally published on Turbonomic.com [20], show an example of a workload where the CPU usage is consistently around 25%, but as soon as the CPU limit is halved the response time increases by 4 times, even though the CPU usage stayed the same. They blame bad container configuration as the culprit without going further into the details.

CPU throttling on running containers needs to be further investigated during the project, as well as keeping an eye out for any irregularities regarding CPU throttling. Finding out when and how often the different containers throttle, together with evaluating the criticality of the applications will be important aspects to cover during the later stages of the project.

In light of our investigation into trends for determining resource requests, which involved analyzing multiple surveys from various companies and exploring the implications of underallocation discussed earlier, we can draw a conclusion.

It appears that developers are motivated to overallocate resources for Kubernetes containers to avoid potential drawbacks associated with underallocation. By doing so, they hope to ensure optimal performance and stability for their applications within the container ecosystem.

2.6 Green Computing

This section will delve deeper into the topic of Green Computing, focusing specifically on data center energy usage and power usage effectiveness (PUE). We will also examine some measures that can be taken to optimize energy consumption within data centers, with the goal of reducing their environmental impact. The EU's ambitious goal of becoming the first climate-neutral continent by 2050 [24] adds even more urgency to this issue, making it necessary for all of us to think about our energy usage and reduce consumption where possible. In this section, we will explore the challenges and solutions associated with Green Computing, including the latest worldwide statistics on PUE.

An important metric to consider when evaluating energy efficiency is PUE, which is the ratio of energy used by the data center to the amount of energy used by the IT equipment and is boiled down to total energy consumption / IT equipment energy consumption. PUE was first introduced in 2006 by the non-profit organization *The Green Grid Association* and has now become the most commonly used metric for reporting the energy efficiency of data centers [25]. A PUE of 1.0 would indicate that all of the energy used by a data center is consumed by the equipment. That is an unrealistic number since additional energy always will be consumed by cooling systems, lightning, and other non-IT equipment. Organizations that are running their own data centers also need to test run the data centers with an aggregate running on fuel in case of electricity deficiency. This is to ensure uptime during a potential crisis where electricity deliverance is being affected.

Data centers have been employing various techniques to enhance their energy efficiency and establish methods for monitoring power consumption. The list below outlines a number of prevalent approaches adopted by data centers to achieve these goals:

- **Server virtualization:** Allows multiple virtual servers to run on a single physical server, reducing the need for multiple physical servers and the cor-

responding energy consumption.

- **Cooling systems:** Implementing cooling systems that are efficient at removing heat from servers, thus reducing the need for traditional air conditioning systems.
- **Power usage effectiveness (PUE)**: Measuring the overall energy efficiency by comparing the amount of energy used by the IT equipment with the total amount of energy consumed.
- **Artificial intelligence and machine learning:** Optimizing energy consumption by predicting and controlling the power usage of servers.

The general placement of the data centers also affects the PUE. Some areas allow for creative solutions such as using nearby water sources for cooling or air-based economization in case of a generally cold climate. A country's mix of renewable energy is also something to consider. If the electricity used by the data center has a high mixture of renewable energy, it will have a positive impact on the PUE. In Intility's sustainability report [26] we see that they have implemented cold and hot aisle containment, water cooling from a fjord, and free cooling using air-side economization.

As one can imagine, new data centers will likely have more smart solutions to reduce the PUE as compared with older data centers. This was also stated in the description of the report from Statista [27].

As mentioned in the introduction to the thesis, data centers are consuming a substantial amount of electricity. In late 2020 the European Commission (EC[3]) published a report that shows datacentres in the EU used an estimate of 76.8 TWh/a which accounts for using 2.7% of the electricity in demand [1]. We will further on be referring to this number, which is 76.8 Twh/a, as the EU's data center consumption. By 2030 EU expect a 28% increase, which will result in total usage of 3.2% of the EU's electricity demand. It should go without saying that the CSPs have done a tremendous job regarding optimizing energy consumption in recent times. Between 2010 to 2018, the amount of computing done in data centers increased fivefold, but the energy consumed by data centers only grew by 6 percent [28].

The EU's total data center consumption is a huge number, and to put this into

[3]The European Commission is the executive of the European Union. It operates as a cabinet government, with 27 members of the Commission headed by a President.

perspective, a regular household in Europe uses 3.7 MWh per year [29], which is a small fraction compared to the EU's data center usage. If we consider the expected increase in data centers' energy usage we are looking at an annual energy consumption of roughly 25 million households in Europe. That is an incredible amount of energy that has no signs of slowing down and low effort to mitigate since the world is in need of computing resources. Therefore, we have to look for solutions in other areas, such as PUE and optimizing energy consumption within the data centers.

The EU has set an ambitious goal of making Europe the first climate-neutral continent by 2050. In other words, using energy efficiently and reducing consumption where it is possible, thus relying less on non-renewable is something we *all* need to think about. Making sure data centers are efficient with their electrical usage will therefore play a huge part in the EU reaching this goal in the decades to come.

So far we have provided some numbers and discussed power usage in Europe's 1500+ datacentres [30, 31]. Other countries in other regions are building new data centers as well, with the USA being the leading country with over 2700 data active data centers [32]. Further on, we will be investigating worldwide statistics on PUE.

According to statista.com the average data center annual PUE was 1.55 in 2022 with a downward trend since 2007 with 2019 as an exception as seen in Figure 2.7.

There are a couple of key takes from this graph, which consists of responses from 669 respondents. First, we see that followed by the downward trend, the graph has stabilized, only going down 0.2 points each year from 2020. Is this due to the data centers not being able to do anything more in regard to the PUE? Have they reached their limit? Google claims to be forward-thinking on power usage efficiency as mentioned briefly in the section about vendor lock-ins, thus being transparent with their energy usage. On their website, they list their continuous PUE improvement that shows the PUE of all their data centers [33]. They reached a PUE of 1.10 in 2020 with no new downward movements after that, just like the worldwide average it seems to have stagnated. However, there is still a long way from 1.55 to 1.10, so there is more to be done to lower the PUE on data centers worldwide.

Secondly, it would be interesting to see the distribution of the data centers and

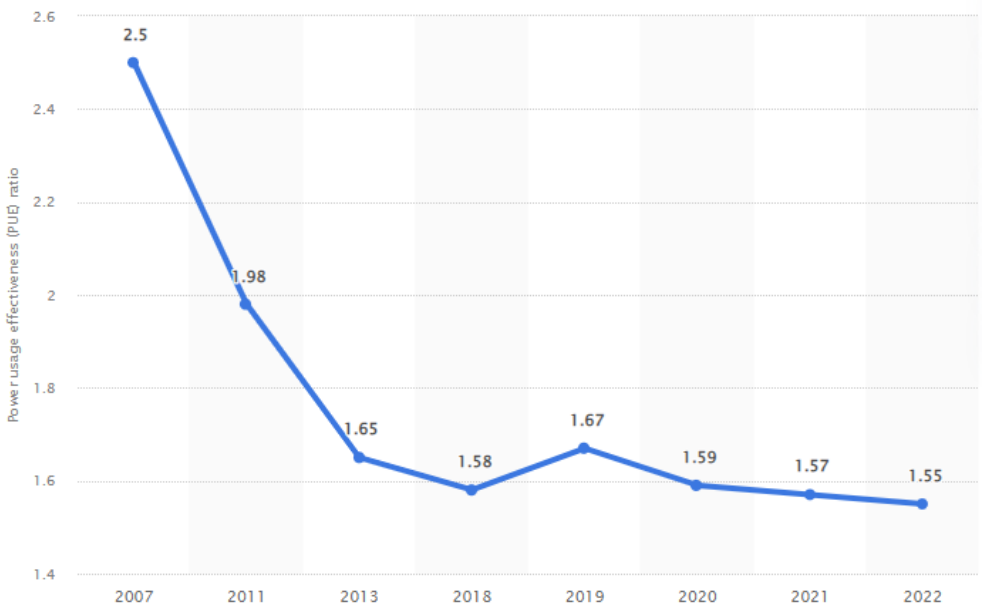

Figure 2.7: Data center average annual power effectiveness (PUE) worldwide (2007-2022) [27]

not only the average, as the average can be quite deceiving in cases such as this one. Are the recent data polluted due to new energy-efficient data centers and are the more mature ones with older technologies still going on as they were? And if this is the case, why would the statistics be presented in a way that showcases a positive downward trend? This opens up more questions and should be further inspected when resource data is acquired. That is however out of scope for this project.

2.6.1 Energy Consumption vs. PUE

While PUE is important, it says more about the efficiency of the data center and not necessarily anything about the efficiency of the running servers. EU's goal of reducing power usage does not only consider the data center's PUE but also the total consumed energy, which can be reduced by allocating resources to deployments more effectively.

Depending on the setup of the data center, the running servers may have a good PUE ratio, and having partly idle servers may even add up to an even better PUE since the power source into the data center is being used efficiently. However, by using a stricter policy, fitting more containers in the servers, and thus reducing

the idle hardware, we may get empty servers that can be completely shut down. While reducing the number of running servers will reduce the needed energy by a fair amount, this may surprisingly enough have a negative impact on the PUE since it only describes the *ratio* of the IT equipment's energy usage versus the cooling. Fewer servers would paradoxically lead to a higher PUE unless the cooling system is adjusted also.

It is important to consider a data center's fixed energy overhead, such as cooling systems, power distribution, and lighting, which continue to consume energy even when servers are turned off. If a significant number of servers are turned off, the fixed energy overhead may represent a larger portion of the total energy consumption, leading to an increase in PUE. Additionally, when servers are turned off, the heat distribution within the data center can change, potentially leading to inefficient cooling. This can cause cooling systems to work harder to maintain the desired temperature, resulting in increased energy consumption and a higher PUE.

By understanding this we can conclude that PUE not necessarily is the correct way to evaluate the energy efficiency of the containers within the nodes/servers. We need to evaluate this in another way by examining the actual resource usage of the containers or the nodes in totality, thus calculating the power consumption of the running idle hardware as briefly shown in *The wasteful consequences of idle resources* in Section 2.3.1.

2.7 Statistical Modeling for Resource Usage

Having introduced cloud computing, the trends in the industry, and lastly green computing, I will now move towards investigating the literature available on modeling workloads on Kubernetes pods. Since we are going to forecast resource usage and use the credibility of our predictions as part of our argument for lowering resource requests, the state-of-the-art within forecasting in the industry should be examined. Statistical modeling can be used to predict resource usage by analyzing data on resource consumption over time. Our goal is to build a model that can accurately predict future resource usage based on past data and other relevant factors. This involves collecting data on resource usage and other relevant variables, such as day and time of day, type of resource, and size of the resource. The data is then used to fit a mathematical model that describes the relationships

between resource usage and the relevant variables. This model will then be used to make predictions about future resource usage.

Statistical modeling of resource usage by containers is not well represented in the literature, especially not in the light of green computing. What is meant by that, is that we have not come across any articles that examine the CPU and memory usage of active containers, their resource requests and limits, and attempt to optimize them to minimize the environmental impact of unused resources. However, some articles are to be found on similar topics. One article suggests creating a proactive autoscaling framework that uses time series analysis to predict the number of pods needed in the future [34]. By predicting the request rate they can remove a number of surplus pods, thus reducing over-allocation. To do this they are using a bidirectional long short-term memory model (Bi-LSTM) on real trace workload datasets from NASA and FIFA. They are also comparing their algorithm to autoregressive integrated moving average (ARIMA) and long short-term memory (LSTM), with a 1-step approach and a 5-step approach. Their proposed system design shows better performance in accuracy and speed than the default Kubernetes horizontal pod autoscaler (HPA) when provisioning and de-provisioning resources. Finally, to evaluate their results they use mean square error (MSE), root means square error (RMSE), mean absolute error (MAE), and coefficient of determination (R^2). Their evaluation is based on two real web server logs: two months of NASA web servers and three months of FIFA World Cup 98 web servers.

While the data analyzed here are a number of HTTP workloads and not the CPU and RAM usage of containers, the predicted data still help to determine the number of pods that should be run. Their goal is similar to ours, but they have a horizontal approach which often is the case for scaling applications in Kubernetes since Kubernetes excels in that kind of scaling. We will strive to use their findings to better estimate the required resources for containers, which essentially is a vertical approach to the problem.

Another article [35] proposes a self-directed workload forecasting (SDWF) method to estimate the future of workload on cloud servers. Their primary contributions to the work are twofold. First, the forecast error feedback is introduced which enables the model to learn from its recent forecasting pattern. Second, a population-based metaheuristic optimization algorithm, i.e., black hole (BH) is improved for better learning of network weights to achieve more accurate fore-

casts. The black hole algorithm got introduced in [36], and is a nature-inspired metaheuristic algorithm inspired by the black hole phenomenon. In that paper, the method is applied to solve the clustering problems, while in [35] it is used to estimate future workloads by first organizing the population into multiple clusters or sub-populations, including both the local and global best information to the process of generating the new solutions. The forecast accuracy of the proposed method is evaluated using MSE, MAE, and mean absolute percent error (MAPE), on different Prediction Window Sizes (PWS) which is the time interval between two consecutive forecasts. For their method, they are using 6 different datasets: HTTP weblogs from three different worldwide web servers, Google cluster traces (CPU and memory), and CPU utilization traces from PlanetLab.

Facebook's Prophet framework, which was first released in February 2017 [37], has proven successful in various recent studies. These studies range from forecasting air temperature and COVID-19 outbreaks to predicting financial markets and Microsoft Azure virtual machine workloads [38–41]. As stated on its website [42], Prophet works best with time series that exhibit strong seasonal patterns and have several seasons' worth of historical data. The results from these studies show that the Prophet model can produce different results depending on the time series being analyzed. For example, in a study using five-year daily time series data [38], Prophet performed better at forecasting maximum temperature, while LSTM performed better at estimating minimum temperature. However, the difference was not substantial when evaluated using RMSE, putting the two methods at similar performance levels. In a study of COVID-19 cases in Saudi Arabia from March to August 2020, Prophet had low accuracy in forecasting recovered cases but high accuracy in forecasting deaths [39]. In a study of the price trend of the Morgan Taiwan Index, using training data from January 2014 to December 2018, Prophet showed advantages in predicting future price trends [42]. In a study of virtual machine demands [41], the authors used one month of resource utilization data, such as maximum CPU utilization and total resource utilization, to predict the next seven days and plotted daily and weekly patterns. The predictions were evaluated using MAPE with a 10% rolling window size. This study used a much shorter time period and produced positive results, which is promising for our project involving short-lived containers, given Prophet's ability to incorporate daily or weekly seasonality.

2.7.1 Cross-Industry Data Forecasting

When discussing Facebook's Prophet, various industries were touched upon. Numerous articles discuss using different models to forecast future activities in diverse fields such as supermarkets, agriculture, oil prices, and stocks. Season-based modeling is quite common in these industries, particularly when seasonal patterns can be easily identified, such as harvest seasons. This enables the model to be more accurate in its forecasts as it learns the seasons more effectively. Prophet tends to favor STL-based modeling, but other methods like the Seasonal Trend Decomposition (STL) [43] method and Extreme Learning Machine (ELM) [44] have been used for a longer period with much success.

One notable paper, published in 2017 with over 110 citations, suggests using a hybrid STL-ELM approach for forecasting agricultural commodity prices [45]. The paper highlights the excellent descriptions of the work conducted and the data utilized, which makes it easy to follow for anyone new to the field of statistical modeling. It sets the seasonality to a 12-month cycle and uses data from January 2002 to April 2014 to predict the prices of various vegetables in the following month. When compared to standalone ELM, Seasonal ARIMA with Kalman Filter (SARIMA-KF), Time Delay Neural Network (TDNN), and Support Vector Regression (SVR), the hybrid STL-ELM method outperforms its competitors in short, medium, and long-term forecasting. As a result, this approach is acknowledged as an effective tool for predicting the prices of highly seasonal vegetables.

After examining various models across multiple industries and diverse data sets, a more comprehensive understanding of statistical modeling, evaluation techniques, and the outcomes of predictions has been gained, including insights into what works and what does not. In the context of the STL, it may be possible to incorporate seasonal factors such as weekdays, time of day, or any other recurring patterns that can be identified. While the applicability of these suggestions remains uncertain, it is intriguing to observe that many similar algorithms with varying parameters are employed across different data in various industries. At this stage, we are prepared to develop an approach to address the problem statements, obtain and further investigate the data, and subsequently explore the application of statistical models on the data.

Chapter 3

Approach

The background chapter highlighted the required knowledge we would need to be able to go forward with an approach. The choices taken in this approach are established from the knowledge acquired in the previous chapter. We now revisit the problem statements once more and provide a short introduction on what kind of research we will be doing for this project. After that we continue by splitting the approach into two phases; Data collection, processing, and modeling (Phase I) and evaluating the efficiency of models across multiple containers (Phase II). In accordance with the Introduction chapter, two research questions were established:

Research Question 1 *How can time series forecasting models be applied to predict resource usage for individual containers in Kubernetes environments, and what are the challenges and benefits associated with using these models to ensure prediction accuracy across various container usage patterns?*

Research Question 2 *How can we effectively evaluate the impact of predictive models on resource allocation strategies in Kubernetes environments?*

The first research question is an exploratory task and will point this project toward exploratory research. This means that we will have a flexible approach to the research process, opening up for modifying the problem statements and methodology as unexpected or exciting findings appear. Since we will not be bound by a specific hypothesis, the project will allow for more creativity in the research process. This type of research does not necessarily reach a conclusion as the goal can keep changing based on what is discovered along the way. The second statement is more about how well we are able to use statistical modeling to predict resource

usage based on the investigation done in problem statement one. The project's end goal is to lower resource requests and limits and measure the benefits of doing so. An appropriate measurement could in this case be the reduction of electricity or costs.

While exploratory research often is time-consuming as it involves gathering and analyzing a large amount of data, it is recommended as an approach for a longer project, where the exploration often will be a part of the literature review or pre-research. In some way, exploratory research can continue for all time, with no end. It should be emphasized that utilizing such an approach for a short project does not necessarily entail unfavorable outcomes. However, we will need to be aware of certain risks. There will be a risk of getting "lost" in the research process and losing sight of the original research questions since there can be a lack of structure and focus if new findings appear frequently. Making crucial decisions regarding the appropriate paths to discontinue researching, as well as which results to utilize, will be critical judgments to make throughout the process. One also needs to keep in mind that our sample size, which we will decide upon, will limit the generalizability of the outcome. This means that the findings may not apply to a broader population or context and only to our very specific situation.

This research approach opposed to confirmatory research or comparative research is a much more flexible and creative approach. The confirmatory approach is designed to test a specific hypothesis or theory using a pre-determined research design, sample size, and data analysis plan. In contrast, comparative research has the goal of comparing the research of others to discover similarities, differences, advantages, and disadvantages. This could for example be to compare multiple statistical modeling algorithms that were determined beforehand to examine which one that gave the best result. Both of the two approaches have a road map that will not change based on their findings since it is more or less defined in their approach. This makes them well structured and easy to follow and stay on track as compared to exploratory research. For the next section, we examine phase I and phase II of our approach. A table summarizing the planned steps and activities is found in the last section of phase II (3.2).

3.1 Phase I: Data Collection, Processing, and Modelling

Before being able to do any computation we will need to acquire data. There are several ways we could acquire data to use for this project. We could for instance use datasets provided by Google or other companies. By doing this we could possibly get clean pre-processed data, which we could start working on immediately. Additionally, the dataset may already have been used and processed by several people, making it possible to get inspired for how to further process the data, or stand on their shoulders to make further contributions. A similar approach would be to use conclusions from other papers in similar fields and use their findings to build this project. An example of that would be to immediately decide upon an algorithm based on the success of that algorithm that was found in another study on predicting resource usage. This approach could open up more time being spent on the more practical approach, on how to implement the findings in a production environment, and on the benefits of applying that exact model.

Given that this project is a collaborative endeavor with Intility, the findings will be of greater relevance if we employ data provided by Intility. In circumstances where it becomes necessary to request supplementary meta information or additional data, it is possible that certain colleagues may offer assistance by furnishing the required data or providing guidance throughout the process. This also applies to elucidating intricate workloads, pods, or the environment. Intility has ensured that a majority of the data from their Kubernetes environment is accessible across various platforms. It will be necessary to examine this environment in order to determine the subsequent direction of the project. Nevertheless, it is anticipated that the majority of Intility's Kubernetes Nodes or namespaces are under surveillance, and as such, the project's phases will be planned accordingly, taking this factor into account.

3.1.1 Processing a Single Container

We will start by investigating one single pod or container on a single node, before investigating multiple pods or containers. This is to ease the data processing and create a slow and steady approach. Investigating multiple containers in multiple nodes from the start may prove to be less time-consuming if the path is set, but in an exploratory project like this, it could be of high risk to do a lot of time-consuming data gathering in case some findings change the direction of the

project.

Python will be used to import, process, and evaluate the data, due to the familiarity with the programming language and its popularity of it among researchers and data scientists. While we assume that we have the data available, we still need to export it to our local machine and import it to our Python environment. Depending on the output of the different datasets acquired from possibly different platforms, we expect to spend some time finding relevant data for our project.

Further on, we will need to consider how far back in time we will gather data and the time interval of the outputted data. Both of these factors will depend on configurations set on the monitoring solutions. Another interesting aspect is to see the aliveness of the containers. We may expect restarts or shutdowns during the time interval for certain containers will need to be addressed. Also, handling peaks and lows in memory and CPU usage will be important. We will need to be able to find and pinpoint restarts, shutdowns, peaks, and lows in the acquired dataset to handle the data accurately.

3.1.2 Data pre-processing

Since we are talking about containers we will expect to see gaps in the stream of data. The gaps could indicate that the pod was either temporarily unavailable or it may have been shut down or restarted. This could be due to a number of factors, including network connectivity issues, system failures, resource constraints, or planned maintenance. We need to explore whether and how to calculate resource usage while accounting for downtime. If we only assess resource usage without considering periods when the pod is down, the resulting resource usage prediction may be inaccurate. Therefore, it is essential to factor in downtime during the calculation process. This will be heavily dependent on how the monitoring platform presents these restarts or shutdowns.

It might be valuable to explore further, particularly in differentiating between out-of-memory (OOM) kills and abnormal restarts, by employing linear regression to examine if the resource usage curve was continuously rising before abruptly stopping. This could be contrasted with a stable resource usage line over a significant period. The former visualization might potentially indicate an OOM kill rather than a regular restart.

To determine the root cause of the data gaps, one may need to examine the logs and events for the pod and the cluster or check for any relevant notifications

or alerts. With that said, we will not examine the logs and events behind the gaps, nor be evaluating and considering redundancy for this thesis. This could, however, be interesting to assess in the future in combination with creating statistical models for the same dataset. Dealing with missing data will vary based on the proportion of data that is absent. Several approaches can be used to address missing data, such as data imputation, data exclusion, or data augmentation. Each method has its own set of benefits and drawbacks to consider. Additional irregularities must also be identified and evaluated individually.

A final consideration is addressing changes made to pods. We have previously discussed the scenario when a container simply restarts. However, when modifications are made to a pod or container's configuration file, the running pod and its container will receive a new name and id, in contrast to when a container just restarts. This could result in the loss of extended data unless the monitoring platform manages that data intelligently. Nonetheless, it might be appropriate to treat a modified container as a new entity.

Other than gaps in data, restarted pods, and pods with new names or ids, we will expect to notice short-lived pods and thus will need to decide if or how to include or when to exclude these pods during the statistical modeling.

Moreover, depending on what meta information we have access to we may need to do some normalization on the values, to make sure they are treated as equal in the models to be used. This can be found by performing an exploratory data analysis, which will give us a broader understanding of patterns, trends, and anomalies in the dataset. This will be done by using plotting libraries like *Matplotlib* or *Seaborn*, to create charts, such as line plots, histograms, or box plots.

3.1.3 Statistical Modeling

In this part of phase 1, we will research and implement different statistical models in Python for predicting a time series dataset. The chosen models will rely on the type of acquired data and the available meta information. When choosing and configuring the models we will need to decide upon a strategy, either being conservative or more liberal with assigning resources. Depending on the uptime and availability expectations of the containers and their criticality we will make important decisions regarding request policies. These choices will also be influenced

by the type of environment they are deployed in, such as test, development, or production environments.

The choice of statistical model and the specific metrics used will depend on the type of resource being analyzed and the goals of the analysis. For example, if resource usage varies significantly over time, a time series model may be used, while if resource usage is influenced by several different factors, a regression model may be used. The final model can then be used to make predictions about future resource usage and inform resource allocation decisions. However, it is also important to consider the complexity and interpretability of the model when making a final decision.

We must also determine how the algorithm should balance over-predictions and under-predictions in relation to the container's original resource usage. Specifically, we need to decide if the algorithm should be penalized when it under or over-predicts resource usage. The overall goal is to explore resource usage, learn how to predict it, and lastly, reduce the allocated resources where applicable.

A less complex model will be the first one to be tested and evaluated. We start off with a simple algorithm since we expect to build and learn from the first one tested. As is the nature of our project. By doing this we possibly also save time by not spending too much time on a complex algorithm that may be taking us in the wrong direction.

The planned number of models is three for this project. This could be one algorithm with three different variations or three completely different algorithms. The choice will depend on the data acquired, the meta information available, and the patterns of the data. We assume that some tweaks may be done to certain algorithms during the modeling, without necessarily calling it a new model. Due to the unfamiliarity of this kind of statistical modeling, we expect an approach consisting of repeatedly trying, failing, and learning.

Now so far we have only mentioned processing a single container, but at this point, we will evaluate the chosen models on multiple containers to get a better view of the generally best algorithm before continuing over to phase 2. It exists numerous ways to evaluate the results of the different models, and we will set out to use multiple evaluations strategies to determine the best-fitting model.

Based on the predictions of the model we are hopefully able to set resource

requests and limits accordingly in a more efficient manner. These requests and limitations should take into account the container's workload which includes peaks and bottoms of resource usage over a longer period. This would be done differently for CPU and memory, as these two behave differently when reaching their set limit as we thoroughly explained in the background chapter.

3.2 Phase II: Evaluating Efficiency of Models Across Multiple Containers

Based on the findings from Phase 1, we will use the best-fitted models and apply them on a larger scale. Depending on the available data and on how applied Intility's monitoring solution is we may be able to apply the pre-processing strategy, processing, and modeling on multiple containers, a namespace, or multiple nodes.

We assume that this phase will be time-consuming, as the repeated process of exporting, importing, and stitching data will take a while, in addition to the steps of data processing. We may be able to create queries or scripts that ease this process. This should be possible within Python in regard to data processing, but it is less known at this stage, how this could be done on the monitoring platforms. The best case scenario is that we have access to all nodes and the format is the same. If we have enough time we could analyze and apply the algorithm to everything running in production, which could give us the potential savings for the entire cluster.

Different nodes and different containers may include other data, new features, or missing data, which we did not face during phase 1. This could eventually lead to changes that need to be done to the chosen algorithm. By clustering the containers that have similar workloads, and creating one master container that could resemble that container with similar workloads, we could possibly reduce the time spent on using the model on all containers by a good amount of time. Clustering could also help with handling the nodes or containers that have unfamiliar data that is not found on container01 on node01 (the first investigated container) since then we can cluster up those containers as well and treat them as one unit. This can be especially useful for containers that have very transparent usage such as bursting over the same intervals over the same points of time, applications only running at day/night time, or static applications using the same amount of resources over a long period of time.

Table 3.1: Implementation steps and activities during Phases I and II. The same abbreviations will be used as check markers throughout the results chapters

Steps	Activities	Abbr.
	Phase I: Data Collection, Processing, and Modelling	
1. Data Acquisition	a. Work together with Intility to obtain pertinent data b. Draw insights from related research studies c. Get familiar with the data available on multiple platforms	**PI-1:** a, b, c
2. Investigating Data	a. Investigate the cluster and decide upon the node to explore for phase I b. Choose containers to analyze and export the data to our local environment c. Use Python to import and pre-process data	**PI-2:** a, b, c
3. Data processing	a. Address data gaps, restarts, and anomalies. b. Examine resource requests and limits in conjunction with resource usage. c. Investigate data distribution, correlations, and patterns. d. Present findings using plots and figures.	**PI-3:** a, b, c, d
4. Statistical Modeling	a. Research and implement models in Python. b. Create a simple model for setting requests and limits. c. Test, apply, and evaluate multiple prediction-based models.	**PI-4:** a, b, c
	Phase II: Evaluating Efficiency of Models Across Multiple Containers	
5. Processing All Containers	a. Acquire data from all containers on node01. b. Pre-process and process all containers.	**PII-5:** a, b
6. Model Performance Evaluation	a. Compare the performance of three different models on all containers b. Determine the best model for predicting resource usage	**PII-6:** a, b
7. Fine-tuning	a. Refine models based on the evaluation. b. Optimize model parameters for better performance.	**PII-7:** a, b
8. Resource Allocation	a. Implement the best model and propose new resource limits and requests. b. Evaluate the performance and the reductions/savings from using the model.	**PII-8:** a, b

3.3 Scope and Limits

Due to the nature of the project, i.e., being exploratory, all new paths opening up along the project within the explored field cannot be explored. This is further enforced by this being a short thesis. With more time we would have liked to research a couple of more things such as investigating multiple nodes in the cluster, different scheduling algorithms, and increasing the resource usage of a container. Especially resource optimization for containers, with having spent a lot of time on mainly the over-allocating, we would benefit from this research, thus using it for both up and down-scaling. By investigating more nodes in the cluster, including dedicated nodes where requests and limits are more carefully assigned, we would get more data to strengthen the support and confidence between the resource usage across workload profiles throughout multiple nodes. We will also not investigate the possibilities of rescheduling pods between nodes to better utilize the available resources tied up with the requested resources.

In some cases one may want to change the configuration for a running service to let the service run on more threads, thus being able to utilize more of the available resources instead of shrinking the over-allocated resources. These scenarios and where they can be applicable would be interesting to further examine in another project or as a continuation of this project.

Chapter 4

Results - Phase I: Data Collection, Processing, and Modelling

In this chapter, an attempt is made to predict future CPU and memory usage by employing a combination of time series forecasting methods and machine learning techniques. A straightforward approach would be to examine the resource usage, identify the maximum, set the request equal to the maximum, and limit a few points above that. However, the goal is to set the request lower while maintaining the same limit as previously suggested. This approach allows for the scheduling of more pods on the same node while still accommodating peaks in resource usage. It is hoped that an algorithm can be developed to predict future resource usage, thereby potentially lowering CPU and Memory requests without negatively impacting performance. This chapter begins with data gathering and pre-processing to prepare the information for further analysis and meaningful feature extraction. Several forecasting methods mentioned in the Background chapter will be further explored, tested, and evaluated using appropriate metrics. Once resource usage predictions are made, an assessment will be conducted to determine if it is possible to lower the CPU and memory requests for a number of pods on a node, based on workload type and policy. The benefits of this approach will also be measured.

In the subsequent chapters, all diagrams and charts were produced using Python scripts that utilized the *matplotlib.pyplot* library, abbreviated as plt. This library facilitated the development of visually engaging graphs that clearly con-

veyed the findings. Despite considerable effort devoted to crafting and enhancing these visualizations, the code for plotting the graphs shares significant similarities throughout the result chapter. As a result, not all plotting of code will be included, but a few examples will be provided at the beginning of this chapter. The emphasis will be placed on presenting the actual algorithms and models employed, showcasing both their visual representations and base code. The plots will mostly consist of the blue line that represents resource usage, black lines for original resource request/limit, red lines for new request/limit, and orange lines for predictions. Some graphs will also have a zoomed-in time frame, which is represented as an orange transparent box on a given time frame. We will come back to all of this at several points during this chapter.

Considering that Phase II would involve running the majority of the algorithms multiple times, efforts were made to ensure that the Python scripts and plotting were created in a way that automated importing, data processing, running the algorithms, plotting, and saving of figures by only changing input variables at the beginning of the Jupyter Notebook. The table from the approach chapter including steps and activities will be referred to throughout the beginning or the completion of steps. This is done by providing the respective abbreviation, for example, Phase I, step 1, activity a, will be referred to as PI-1a.

4.1 Data Collection

In the approach chapter, it was mentioned that data would be made accessible to a certain extent. Throughout the project, access to a Dynatrace dashboard was obtained, which included multiple nodes and their corresponding pods. Initially, time was dedicated to exploring the data available on the Dynatrace platform. Dynatrace is a software intelligence platform offering performance monitoring and management solutions for cloud-based applications. Dynatrace agents can be installed on nodes or directly injected into containers within a pod. Although the resource usage of containers, such as CPU and RAM, was accessible, the set resource requests and limits for the containers were not. This was true for the majority of containers, necessitating the retrieval of this data from alternative sources at a later stage.

The second challenge involved dealing with two distinct metric graphs for CPU and memory. One graph illustrated CPU usage in terms of mCores (millicores) per minute, while the other depicted the percentage of the node's CPU utilized by the

container. Although these values appeared fundamentally similar, it was necessary to ensure that the graphs were not measuring anything additional or distinct. To confirm this, the percentage of usage was multiplied by the node's CPU, resulting in the correct number of mCores shown on the first graph. Additionally, the graph incorporated CPU throttling in mCores, which is assumed to be a valuable metric when recommending CPU requests. Generally, CPU usage can be further divided into system usage and user usage. In this study, any mention of CPU usage refers to the combined usage of both system and user usage. In other words, they will not be analyzed or treated separately, as the primary focus is on the resource consumption of each container at a more comprehensive level.

In the case of memory usage, there are two graphs: container memory usage and process memory usage. These two graphs do not exhibit similar trends or workloads. It is essential to differentiate between them and understand the significance of each metric. The former metric, container memory usage, likely represents the total memory consumption of the container and is the metric considered by Kubernetes when managing resource requests and limits. Consequently, this is the metric that will be taken into account when evaluating memory usage in relation to requested resources. On the other hand, the latter metric, process memory usage, has more dependencies and may encompass heap memory, stack memory, and other process-specific memory allocations. In other words, process memory usage might appear to consume more memory than the container's total memory usage, even if that particular process is only running on that single container.

In order to address the absence of displayed resource requests and limits on the Dynatrace platform, a senior architect was contacted to obtain the relevant data (PI-1a). A Prometheus server was set up and began scraping resource requests and limits from pods in an internal production environment – the same environment accessible through Dynatrace. While connected to the company network, the endpoint could be accessed with added arguments to retrieve limits and requests. By examining the entries and labels in the Prometheus graphical user interface (GUI), a query was formulated to acquire the necessary data for the project's continuation. Code listing 4.1 illustrates the endpoint and the query used to obtain the resource limits for a single container in node01 over the past 12 weeks. The query employed in the code listing is identical to the one used in the Prometheus GUI. The first node analyzed will be referred to as node01, which

is a node in a production environment, and the first container in node01 will be denoted as *container01* or *c1*. To obtain requests, the word "limits" would be replaced with "requests", and the same principle applies to swapping the resource, such as CPU, with memory. This concludes.

Code listing 4.1: Prometheus setup for resource data retrieval with an example query for a single container on node01

```
curl -g 'https://intiliy-url.no/api/v1/query?query=
    kube_pod_container_resource_limits{node=~"node01",container=~"container01",
    resource="cpu"}[12w] > node01_cpu_resource_limit.txt'
```

At present, a server displays requests and limits for most containers; however, not every container has assigned requests and limits, as this assignment is not strictly enforced.[1]. On March 26th, the Prometheus server was reset, resulting in the loss of request and limit data gathered from the Kubernetes environment since January 31st. Although this may appear to be a significant setback, the requests and limits set rarely change, so it can be assumed that if the history of the same container name is found in Dynatrace, the current request and limit number can still be used when analyzing the container.

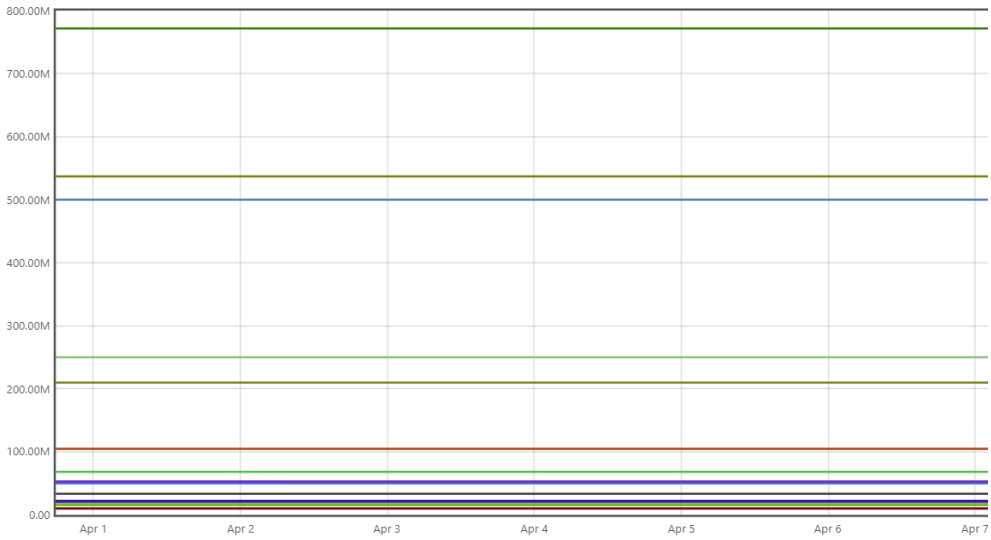

Figure 4.1: Visualization of memory request variations for 38 operational containers on Node01, with 14 distinct memory request lines displayed. All the lines demonstrate a constant memory request

[1]During this project, Intility implemented a new policy, mandating developers to set requests and limits on *new* deployments

In the past ten days, Node01 has had 38 operational containers, according to Dynatrace metrics. The Prometheus server also displays 38 entries for CPU and memory requests. It is worth noting that despite the large number of entries (38), only a few lines are seen in Figure 4.1, since only memory requests and not CPU requests were included in the query shown in the figure. In that figure, 14 colored lines represent 14 different variations in memory request. When executing the same query for CPU requests and memory limits, similar results are obtained, but with even fewer variations of set resource requests. This is particularly true for CPU limits, where there are 12 entries, with 3 different limits set: 1mCores, 0.5mCores, and 0.3mCores.

The lines appearing on top of one another in Figure 4.1 indicate identical requests, rather than differing values. The findings suggest that the developers rely on a specific set of numbers to estimate memory and CPU usage. This trend is more pronounced in CPU than in memory, where variations are more extensive, supporting our hypothesis that requests and limits are determined using numbers from other similar deployments within the cluster, or by trial and error. Figure 4.1 also displays the lack of change of the requested resources for all containers. These findings also tell us that the request and limits are constant and do not change during the runtime for any container, at least on node01, and is most likely the case for most nodes if not for the entire Kubernetes cluster. Lastly, CPU limits are only configured for 59% of the containers, while memory limits are configured for 62% of entries.

Observation 1:

Neither memory nor CPU requests and limits are altered during a container's runtime, and the established resource requests and limits are strikingly similar, despite the containers handling distinct workloads.

Data was also transferred from Dynatrace to the local environment, employing a similar approach to that used with the Prometheus server. By exporting query results from the Dynatrace GUI, access to a node and any container running on that node was possible. Initially, the focus was on analyzing three distinct containers within a single node. However, while developing Python scripts for importing and processing container resource usage, the code was designed in a way that adding more containers to the existing script would necessitate minimal changes. This method ensures code reusability for future research stages.

When accessing a container in Dynatrace, dashboards with existing queries are available, and it is possible to open one of the visualizations and export it to a .csv file to access the file locally and import it into the Python environment. Before exporting the data, time intervals need to be addressed. As discussed in the approach, either short or long intervals must be chosen. Due to the project's nature, high-resolution intervals are not deemed necessary. Higher intervals are better suited for real-time monitoring and short-term fluctuations. Dynatrace offers a minimum 1m interval, with the next option being 5m. Therefore, a 5m interval will be used, which is believed to be suitable for identifying trends and patterns while reducing storage and processing requirements. To better detect resource usage peaks and maintain a conservative approach regarding overestimation, data points will concentrate on the maximum value for the 5m interval instead of the average value, which is usually Dynatrace's default.

For the NASA and FIFA datasets from the paper on bidirectional short-term model (Bi-LSTM [34] in the Background chapter, Section 2.7, a two and three-month basis were used. While on the paper using the time series forecasting method from Facebook (Prophet) [41], the time series data points are within a one-month time period, which they classify as short-term load (PI-1b). Containers are typically short-lived, as discussed in the background chapter, with a few exceptions. Dynatrace permits 5m intervals up to 21 days back in time; however, when switching further back than 21 days the time interval for the entire time series, the time interval gets capped at 1h. For containers with more than 21 days of data, one can either use the last 21 days with a 5m interval and ignore the rest of the available data, or use the entire data set from the last month.

Having worked together with Intility to obtain pertinent data, drawn insights from related research studies, and gotten familiar with the data available on the multiple platforms, we finish off step PI-1a-c.

4.2 Data Processing (PI-2a-c)

Before applying statistical models to the collected data, it is essential to perform data processing. This section will address converting objects to float values, managing missing information, standardizing the time format, addressing spikes, and incorporating constant values such as request and limit. Data distribution, correlation, and pattern identification will also be examined. Some of these preprocessing

steps are taken to fill data gaps and make the data suitable for algorithmic applications.

Data was collected from three containers on node01 (n1), referred to as container01 (c1), container02 (c2), and container03 (c3). These containers and the node were chosen randomly, ensuring there was some recent activity in terms of CPU and memory usage for the selected container. At this stage, the types of containers running on the node and the workload of the given container are not considered. This aspect will be revisited when examining workload profiles. Initially, these three containers are considered before analyzing the entire node. This approach helps simplify the processing and ensures the framework works as expected before investing significant time in collecting and pre-processing large amounts of data. To further emphasize taking one step at a time, memory usage is processed first, with the CPU being used as arguments or for finding correlations during processing.

Data processing and modeling were coded in an interactive web-based environment called Jupyter Notebook, popular among data scientists and researchers. The notebook supports Python, among other libraries, and is the chosen programming language for the remainder of the project due to the author's familiarity with the language and Python's status as a scientific language. By employing 5-minute intervals with the peak value, increased volatility can be anticipated in containers that fluctuate due to processing workload in brief time frames, as opposed to examining the 5-minute mean. However, using the maximum value allows for better and a more realistic comparison to the container's limit, helping determine Out of Memory (OOM) issues and explain potential CPU throttling.

Although data are available for when a container's CPU throttles and how many millicores it throttled, the Linux kernel's throttling algorithm is quite complex, making it difficult to predict the exact moment of throttling and the amount it throttles.

After importing 14 days of data for c1, 15 days for c2, and 16 days for c3, the pre-processing of .csv files for CPU and memory usage began. An in-depth explanation of the pre-processing will not be provided here, but relevant code with comments can be found in Appendix B.1.

The data processing will be divided into two sections: one for memory usage and another for CPU usage, starting with memory usage. The pre-processing de-

scribed in the following two sections will be applied to all three containers, but only the plots from c1 will be displayed in the first two sections. However, findings from c2 and c3 will be discussed. In the section where the statistical models are applied to containers c1-c3 (Section 4.3), the accompanying graphs and explanatory text for c1-c3 will be provided. This approach allows for the presentation of resource usage for c2 and c3 without overloading the paper with excessive figures.

4.2.1 Analyzing Memory Usage

In Figure 4.2 we have the memory usage before processing the data for container01. In the figure, it is observed that the missing data accounts for small percentages (between 1-2%). The period of missing values can be due to an external restart of the pod or a restart performed by the container itself. External restarts could be initiated by an administrator, the monitoring platform, or the node itself if it needed to quickly shut down a container to keep another container with higher priority alive. Determining the exact cause would require further examination of event logs. It is unlikely that these gaps are due to the memory limit being reached, as there is no sudden peak in memory consumption. Moreover, if the container were redeployed with changes in its configuration file, it would receive a new ID connected to its name.

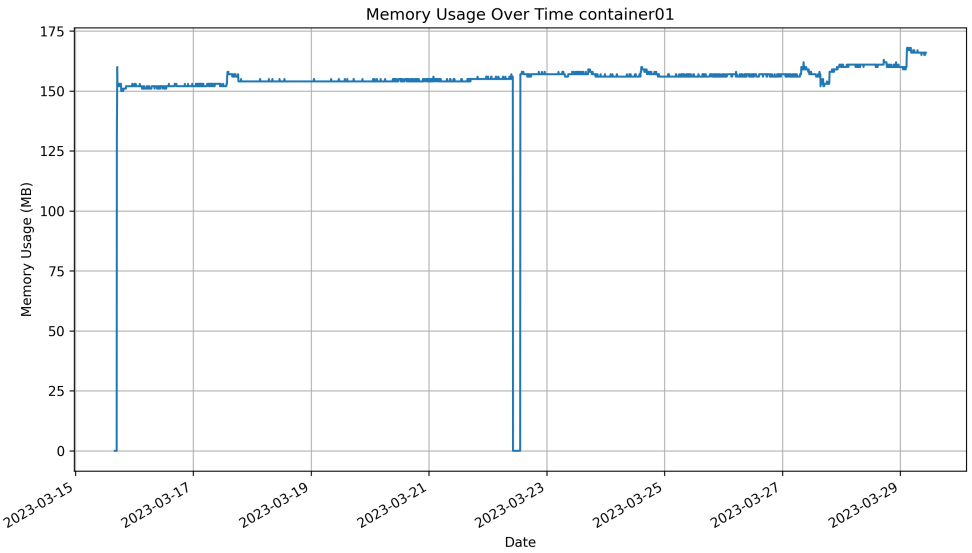

Figure 4.2: Memory usage over time for c1, showcasing periods with missing data that account for small percentages (1-2%)

Since this container retains its ID, that possibility can be ruled out. While the

gaps do not necessarily indicate an anomaly, handling the missing data is crucial to ensure the proper functioning of the algorithms.

For this first figure, we included the coding for the plot, subplot, and labels, as seen in Code Listing 4.2. This is to show how the rest of the figures will be plotted, with some adjustments. New lines, horizontal lines, and labels will be added, however, the main frame will look like this code. In addition to plotting the data, captions, and labels, this script saves the figure in our local environment which makes it accessible in the exact same size as the rest of the containers will be when re-running the script with a different .csv file. For the code regarding importing and cleaning the .csv file, see Appendix B.1.

Code listing 4.2: Base plot for container resource data visualization with customizable elements

```
import matplotlib.dates as mdates
import matplotlib.pyplot as plt

# PLOT
fig, ax = plt.subplots(figsize=(12, 6))
ax.set_xlabel('Date')
ax.set_ylabel('Memory Usage (MB)')
ax.set_title(f'Memory Usage Over Time container0{container}')
ax.grid()

# Customize date ticks and format
ax.xaxis.set_major_locator(mdates.AutoDateLocator())
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
plt.setp(ax.get_xticklabels(), rotation=30, ha='right')

ax.plot(df1['Date'], df1['Containers: Memory usage'],label=f'Memory Usage c{
    container}')
ax.legend()

# Save and show the plot
plt.savefig(f'memory_usage_c{container}.png', dpi=300, bbox_inches='tight')
plt.show()
```

We did not find any missing values on c2 and c3, however, this could be due to that those data were collected at different times. We decided to use the mean value of the total usage to contemplate the missing values. Figure 4.3 shows the plot of c1's memory usage after this had been done. What we notice in both of these figures is that they both use a lot of resources at startup, but the usage afterward differs. The memory usage goes down about 25% then keeps steady

with a slow rise. We will refer to this symptom as the *L* model, which resembles the high startup phase followed by the immediate drop in resource usage. We do not see the *L* model as clearly in c2 due to erratic behavior, but it is present in c3. Nonetheless, a modest upward trend in memory usage is observed across all three containers, and it is expected that the proposed models will account for this trend.

Observation 2:

The memory usage exhibits an *L* model behavior which is represented by high initial usage followed by a steady slow rise.

In order to address the missing data, we initially processed the imported .csv file containing date and memory usage information. We then filled the gaps in the data using the mean resource usage value. The date was standardized, and the resource usage, represented as a string in GB, MB, or Bytes, was converted to a float and adjusted to display in MB. Finally, we employed the mean value to fill any missing data points (PI-3a).

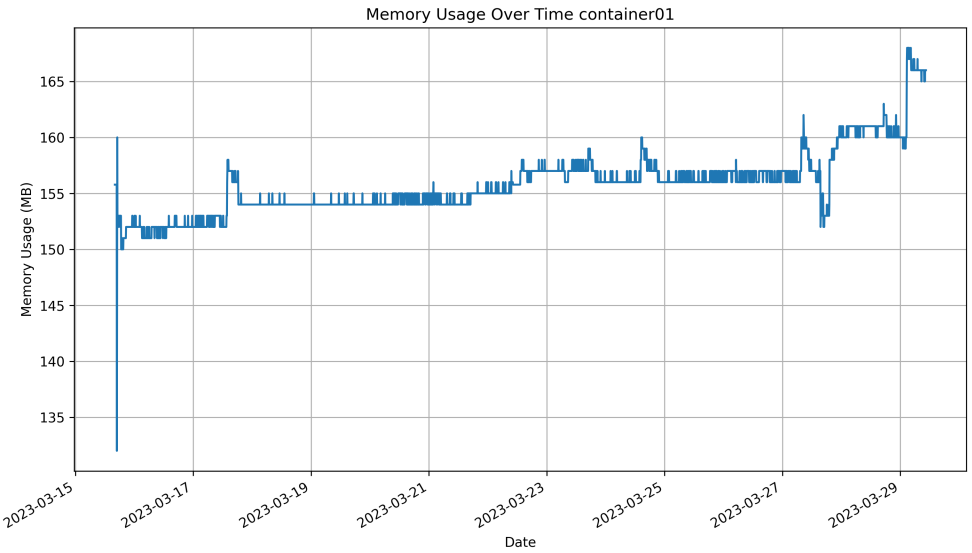

Figure 4.3: Memory usage plot for c1, displaying the data after incorporating mean values for missing data points

In Figure 4.4 we added the request and limit set on the container in addition to custom thresholds. It is in this figure we for the first time see how big the difference is between the requested resources and the actual usage, which we conclude

with is *Heavily Overallocated* (PI-3b). The resource request and limit in this case is set to the same number, that is 736MB, and is represented by the red line. The blue line represents the same memory usage from Figure 4.3 but is seen with respect to the resource request/limit.

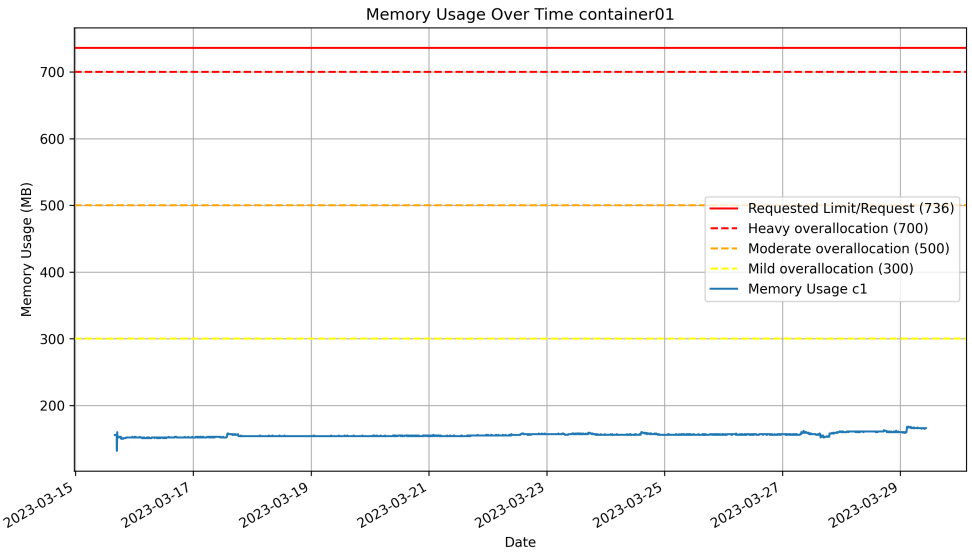

Figure 4.4: Comparison of c1's actual memory usage and the set resource request/limit, the distance between the solid red line and the solid blue line highlights the significant overallocation of resources

Before beginning with the optimization models we examined the distribution of the recorded memory usage put into bins. Figure 4.5 shows the distribution fit into 20 bins. The dataset has a skewness value of 1.01 which indicates a slightly positive skewed distribution. In other words, the data has a longer tail on the right side of the clustered distribution, and most of the values are clustered towards the left. We also see that the median and mean values are almost identical, with the mode being a bit more to the left. This tells us that the most common values are lower than the average and median values. For c2 the median and mode have similar values, while the mean is a bit more to the left. The skewness value is -0.68 which indicates that the distribution is slightly skewed to the left, meaning that there are more values below the mean, than above. Lastly, on c3 we got a negative skewness of -1.62, which indicates that the distribution is substantially skewed to the left. Based on these distributions and the nature of our data, we can conclude that standard deviation will not be considered as an evaluation method.

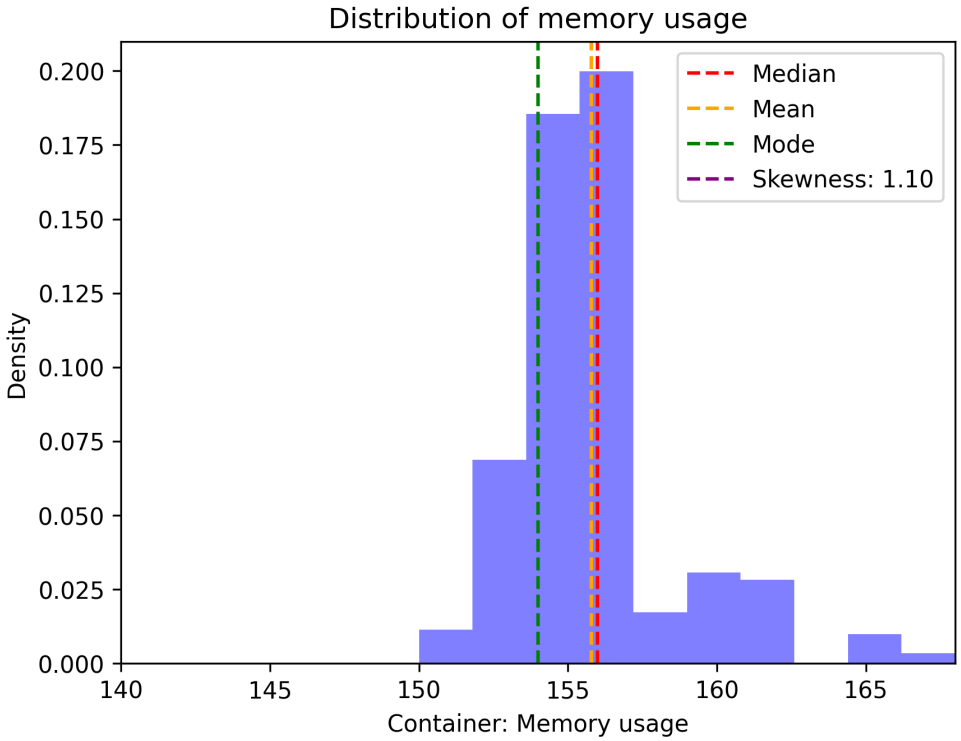

Figure 4.5: Memory usage distribution for c1, revealing slightly positive skewness and the decision against using standard deviation for evaluation PI-3d

Something we did to investigate the possibilities of any hourly or daily patterns in the memory resource usage was to split up the dataset into 24h splits, then lay them on top of each other (PI-3c. This is displayed in 4.6, where each line represents a day between 2 weeks of data from container01. By the visualization itself, we can say that there is no immediate pattern to be found on either of the containers. Similar output was also gotten from c2 and c3. With no patterns being observed we will not be considering seasonal decomposition methods per now, and therefore stray away from methods found in Section 2.7.1, where we discuss findings of statistical methods from other industries, that focus more on seasonality or trends. However, in phase II of the project, we do open up the possibilities that more cyclic usage from certain containers may be found.

Observation 3:

Memory usage distributions show varying degrees of skewness, and no clear patterns were found in the hourly or daily memory usage.

A code snippet from the function that splits and plots the 24-hour segments is shown in Code Listing 4.3. The *plot_24_hour_segments* function takes in the data frame and the resource usage as columns as arguments. The unique days are stored in *days*, which we later loop through, extract data, and plot with a label representing the day. The last full day is carefully calculated in regard to the first timestamp. The full code along with code for plotting which is also in the same function just described, can be found in Appendix B.1.4.

Code listing 4.3: Code snippet illustrating the 'plot_24_hour_segments' function, used to split and plot 24-hour segments for memory usage data

```
def plot_24_hour_segments(df, column, window_size=12):
    first_timestamp = df.index[0]
    days = df.index.normalize().unique()

    # Calculate end date
    last_full_day = df.index[-1].normalize() - pd.Timedelta('1D')
    end_date = last_full_day + pd.Timedelta(hours=first_timestamp.hour, minutes=
        first_timestamp.minute)

    # Apply rolling average
    df_smooth = df.rolling(window=window_size, center=True).mean()

    plt.figure(figsize=(12, 6))

    for day in days:
        start_time = day + pd.Timedelta(hours=first_timestamp.hour, minutes=
            first_timestamp.minute)
        end_time = start_time + pd.Timedelta('1D') - pd.Timedelta('5m')

        if end_time <= end_date:
            day_data = df_smooth.loc[start_time:end_time, column]

            if not day_data.empty:
                times = [t.hour * 60 + t.minute for t in day_data.index.time]
                plt.plot(times, day_data.values, label=day.date())
```

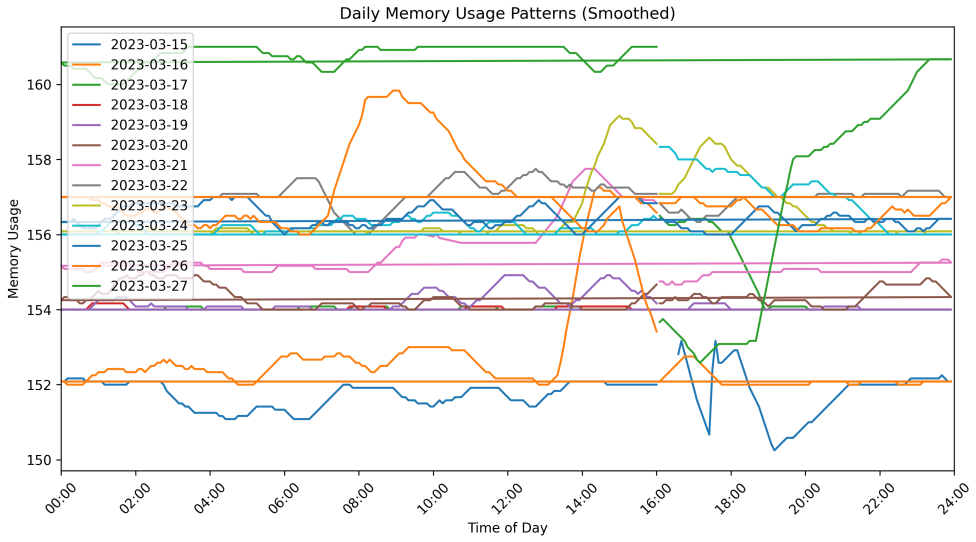

Figure 4.6: Overlapping 24-hour memory usage segments for container01, the variations in each line illustrate the absence of clear hourly or daily patterns in resource consumption

4.2.2 Analyzing CPU Usage

For the CPU for c1 (see Figure 4.7) we see a peak of almost 1000mCores followed by an immediate drop to a relatively low value. Following the drop it may seem like the usage goes on a steady course, with one small peak towards the end.

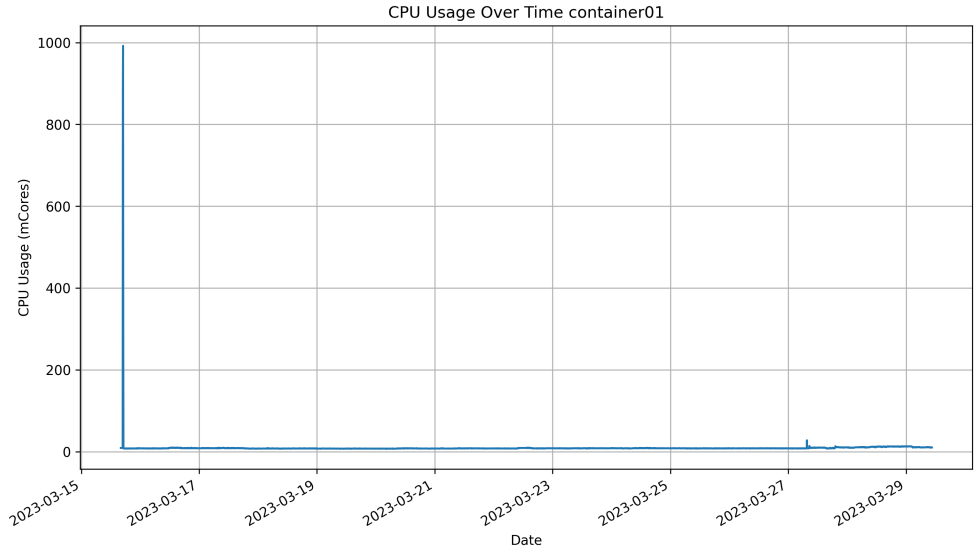

Figure 4.7: CPU usage plot for c1, displaying the data after incorporating mean values for missing data points

For this plot, we have already done similar pre-processing that was done for the plots of c1's memory. The plot strengthens the theory of the *L* model, showing high resource usage from the first seconds of runtime.

The decision was then made to incorporate and visualize the CPU request and limit settings for c1, with the aim of assessing the difference between resource requests and actual CPU usage. In Figure 4.8, we notice that the CPU request is set to 100mCores, significantly higher than the average CPU usage of around 5mCores but much lower than the peak, which is close to 1000mCores. This container does not have a set resource limit for CPU usage. We also included a zoomed-in plot for the 27th of March in the same figure to closely examine the output (indicated by the orange transparent rectangle). In the zoomed-in frame, we observe highly fluctuating usage, including a peak reaching nearly 30mCores. Based on these graph outputs, we start to think that predicting these values using any algorithm may be challenging. In c2 and c3, we observe similar erratic behavior, with more frequent peaks like the one that jumps from around 5mCores to a sudden 30mCores usage, as shown in the zoomed-in figure. However, unlike c1, they do not exhibit extremely high consumption at the start of runtime. Additionally, none of the containers appear to display a gradual increase in CPU usage, in contrast to memory usage.

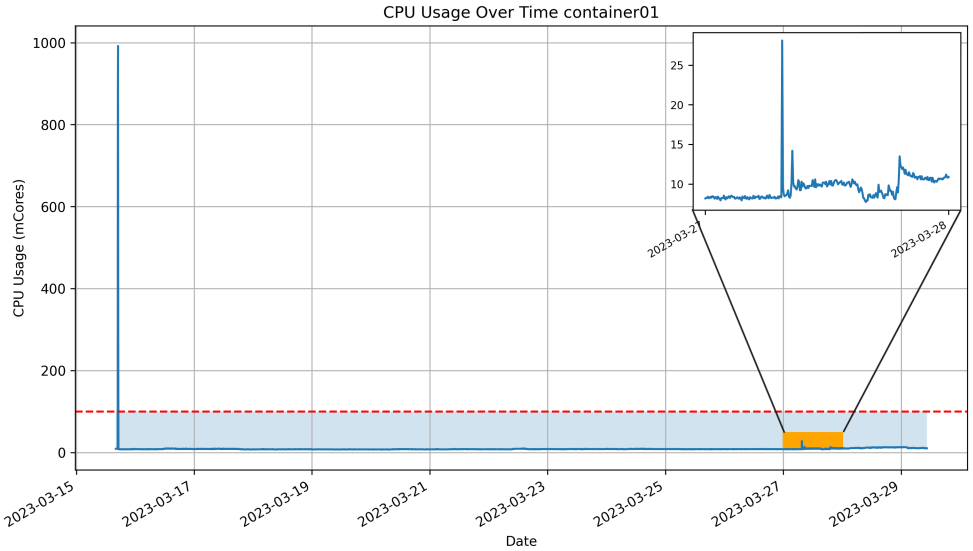

Figure 4.8: CPU usage for c1 with set CPU resource request indicated by the dotted red line. The figure showcases overallocation since the requested resources are high compared to the actual CPU usage. The plot in the top right corner zooms in on the orange area and shows the erratic behavior of the c1's CPU usage

While the code for this plot is more or less like the plots for memory usage in the previous section, we see the inset plot with the orange square for the first time. In Code Listing 4.4 we see the additions to our normal plotting. The start and end dates are specified before we filter the data to only include the data between those dates. Further on we fit the inset plot in the top right corner before we draw the orange rectangle using the *ax.add_patch* with arguments based on start date, end date, and height.

Code listing 4.4: Code snippet illustrating the addition of an inset zoomed-in view (orange square) to the CPU usage plot

```
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
# Create the inset plot
start_date = pd.Timestamp('2023-03-27')
end_date = pd.Timestamp('2023-03-28')

filtered_data = df2[(df2['Date'] >= start_date) & (df2['Date'] <= end_date)]

axins = inset_axes(ax, width="30%", height="40%", loc="upper right")
axins.plot(filtered_data['Date'], filtered_data['cpu'])

axins.xaxis.set_major_locator(mdates.DayLocator())  # Show only one date
axins.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
plt.setp(axins.get_xticklabels(), rotation=30, ha='right', fontsize=8)
plt.setp(axins.get_yticklabels(), fontsize=8)

# Draw a rectangle on the main plot to indicate the zoomed-in region
rect_x1 = start_date
rect_x2 = end_date
rect_y1 = min(filtered_data['cpu'])
rect_y2 = max(filtered_data['cpu'])

new_height = (rect_y2 - rect_y1) * 2

ax.add_patch(plt.Rectangle((rect_x1, rect_y1), rect_x2 - rect_x1, new_height, fill=
    True, color='orange', linestyle='-', linewidth=1, facecolor=(1, 1, 0, 0.3)))
```

Investigate Correlation (PI-3c)

Lastly, we calculated the correlation between the CPU and Memory usage as this can prove to be important to consider when moving forward to statistical models. We used the Pearson correlation coefficient (see Equation 4.1), which is the most common way to measure a linear correlation, to measure the relationship

between the CPU and Memory usage for this container. When using this formula with x variable being the CPU and y being the memory usage. The correlation coefficient we got was -0.09 which is a weak negative correlation between CPU and memory usage. This means that as CPU usage increases, memory usage tends to slightly decrease, but the relationship is not very strong. For c2 we got a correlation coefficient of 0.37 and 0.02 for c3, which are both considered as relatively weak coefficients in our field.

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{4.1}$$

Based on this correlation coefficient, it may still be beneficial to include the CPU as an independent variable in a predictive model, as even a weak negative correlation can provide some additional information to help predict memory usage. However, in this case, we decided not to move forward with using either CPU or Memory as predictor values for calculating resource usage. One could also consider other factors, such as the significance of the coefficient and the Variance Inflation Factor (VIF) values when deciding whether to include CPU in your model, but neither of the two will be used further on in the project.

4.3 The Statistical Models

We have discussed various statistical modeling methods that could potentially be applied to address the problems at hand, and in the following sections, we will delve into the selected statistical model and explain why it is suitable for our dataset. Based on the findings from the previous section, we have decided not to include CPU usage due to its unpredictable nature and the presence of unforeseen peaks, making it difficult to analyze. Based on research so far we do not see any correlation between the two metrics either. By focusing solely on memory usage, we can dedicate more time to refining the models, tuning, and evaluating before commencing phase 2. By employing a statistical model in conjunction with workload profiles, which are averages of clusters containing similar services, we may be able to predict the future resource usage for RAM. However, we must still take into account edge cases and factors such as out-of-memory (OOM) errors, penalizing under-allocation, and allowing some room for sudden spikes, which we will discuss further in this chapter.

4.3.1 Exploring Strategies for Resource Optimization

As our approach is not proactive, wherein we would constantly adjust request limits and requests using the model, we cannot be certain if resource usage patterns will change significantly over time. Considering the usage of container01, it may be more effective to utilize percentiles like the 95th percentile of historical usage, or alternative heuristic approaches such as optimization models like linear programming and other linear algorithms. Therefore, we will begin with a simple linear programming approach. During Phase I, the models will not undergo fine-tuning as the emphasis is on general characteristics. As the process advances through multiple containers in Phase II, it becomes more appropriate and well-founded to fine-tune the models.

Before diving into this, let's restate our primary objective. By reducing resource requests, we can eventually fit more deployments into their respective nodes, and if we can accommodate enough deployments, we may be able to shut down a node entirely. It's important to see the bigger picture when evaluating these models. If multiple developers each request 500MB more than necessary for their containers, and there are tens of nodes with thousands of containers, these figures will accumulate into a significant amount. Considering the hundreds of servers under Intility's ownership, this could result in a considerable amount of saved electricity.

To further illustrate this example, let's consider node01 and container01. By examining the memory usage and request for c1 on node01, we can identify opportunities to improve efficiency and increase the number of containers that can be hosted on the node.

In this case, we find that the memory request for c1 is set to 736MB, while its actual usage is only around 170MB. By reducing the memory request to 180MB, we could potentially decrease the resource allocation by 75.5%, freeing up more memory for additional containers on node01. If we could reduce the memory request by 50%, we could effectively double the number of containers running on the node.

Taking this approach and reducing the memory request for c1 from 736MB to 180MB, we could potentially fit four times as many containers (266 containers) on the same node, as we see in Figure 4.9. This increased efficiency could have significant benefits in terms of cost savings, resource utilization, and overall sys-

tem performance. The other containers may vary significantly from container01's findings, and we do not expect to see a 556MB decrease in memory allocation for other containers. Yet we wanted to provide a figure and finalize the concept of over-allocation in a grander format.

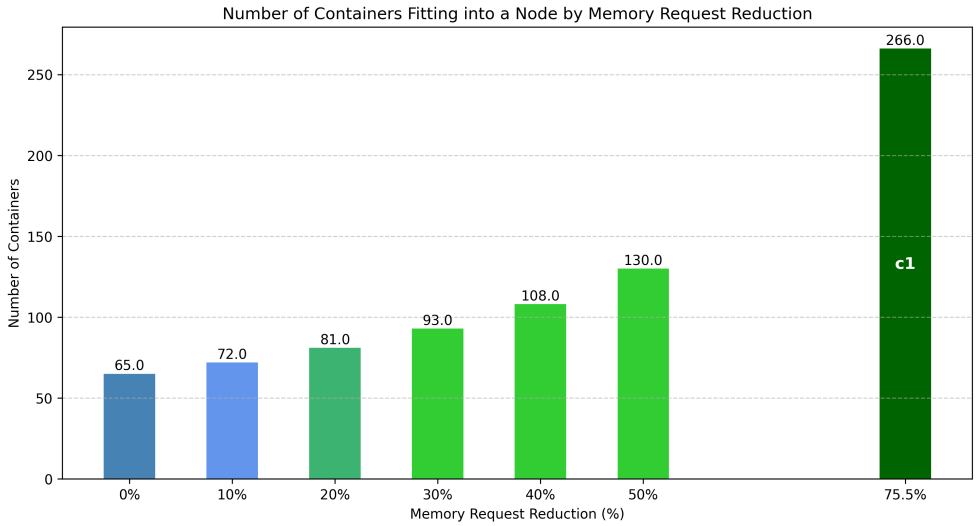

Figure 4.9: The graphic displays the potential increase in container capacity on node01 after optimizing the memory request for c1 from 736MB to 180MB. The diagram indicates that by applying a similar reduction to multiple containers like c1, the capacity for accommodating containers on the same node could potentially quadruple, expanding from 65 containers to 266

It is important to note that this analysis focuses solely on memory allocation and does not consider other factors such as CPU usage and its request/limits. To ensure that the increased number of containers would still function effectively on node01, we would also need to evaluate and optimize CPU requests and usage accordingly. By doing so, we can create a more efficient and well-optimized Kubernetes cluster that can host a larger number of containers without overprovisioning resources.

From the next section and onward we will compare and discuss memory requests and limits frequently. While the figures should be clear enough to explain what lines represent the memory usage, previous request/limit, and new suggested resource allocations. It is important to underline that the solid red lines will always represent the suggested memory *requests*, and the red dotted lines will represent the suggested memory *limits* unless else is specified. Both of these types of suggestions will be based on the particular model which is applied. The

solid black lines will represent the original memory *request* and black dotted lines will represent the original *limit*. All four lines will be drawn as constant lines going across the entire plot. Resource request and limit will sometimes be referred to as R (request) and L (limit), and R/L, but the context of their usage will then be strictly specified. An example would be if the resource request and limit are set to 100MB and 200MB we could say R/L = 100MB/200MB.

Definition 1:

Solid black lines and dotted black lines represent the original resource request and original resource limit. While solid red lines and dotted red lines represent new resource requests and new resource limits, as suggested by the applied model. Resource usage will be represented as blue lines.

4.4 Linear Programming

For our first optimization model, we went for linear programming, which is a solution that does not base the suggestions on any predictive analysis. This model will be a part of PI-4a since it is the first model to be implemented in Python, while also covering step PI-4b since resource allocation will be suggested after getting the results from the model. We use the *pulp* library for this solution.

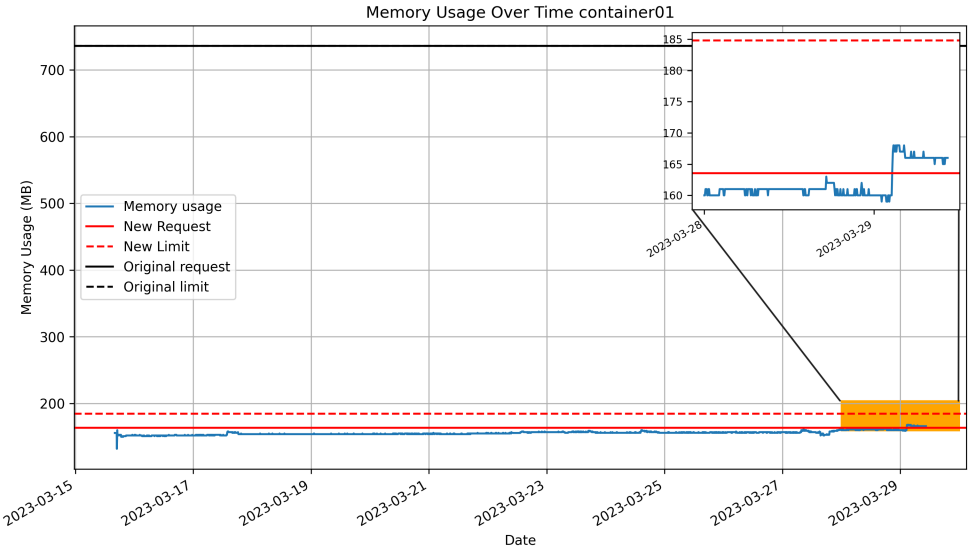

Figure 4.10: Visualization of c1's original and new memory request and limit. The zoomed-in plot shows that at one point the resource usage goes over the requested memory, but still stays comfortably under the limit

The first step is to set a safety margin, because without a safety margin the proposed resource request would be equal to the average or the median of the data. In our case, it would be average, since we chose to use the average as a plausible representation for our data. By only using the average to set the resource requests we limit ourselves to a static trend. Realistically we would want a safety margin, especially for containers running in a production environment. In our case we set the safety margin, that is the *resource limit*, to be 10 percent over the maximum memory usage observed in the 2-week period acquired. By doing this we ensure that the high usage at the start of the container's lifetime is considered if we were to assign this limit to another container with a similar workload. This is shown in lines 4 and 5 in Code Listing B.8. For the *resource request*, we used the typical memory usage and increased the value with 5%. This model has been inspired by the case study called *Set Partitioning Problem* which is found on *PuLP's* website [46].

The function *pulp.LpVariable()* takes the new values and sets them as lower bounds for the variables for the problem. Further on we add constraints to the problem, where the requests must be less than or equal to the memory limit. Finally, The *prob.solve()* function retrieves the optimal values for the variables, that is request and limit, that respects the boundaries. The set memory request and limit based on this model can be seen in Figure 4.10. To begin with, c1 had the request and limit set to 736MB as we saw in 4.4. With the model applied we have R/L = 164/185. That is a 572MB reduction in request and a 547MB reduction in limit.

Code listing 4.5: Implementation of linear programming to optimize memory request and limit values

```
1  # Linear programming model
2  import pulp
3
4  safety_margin_percentage = 10  # Add. memory allocated as a percentage of max usage
5  safety_margin = max(df1['Containers: Memory usage'])*(safety_margin_percentage/100)
6
7  # Calculate the typical memory usage (e.g., the mean or median)
8  typical_memory_usage = np.mean(df1['Containers: Memory usage']) * 1.05
9
10 # Defining the problem, and specifying that the objective is to minimize a function
11 prob = pulp.LpProblem("ContainerResourceAllocation", pulp.LpMinimize)
12 # Variables
13 x_request = pulp.LpVariable("x_request", lowBound=typical_memory_usage)
14 x_limit = pulp.LpVariable("x_limit", lowBound=max(df1['Containers: Memory usage'])
```

```
    + safety_margin)
# Objective function
prob += x_request

# Constraints & solving the problem
prob += x_request <= x_limit
status = prob.solve()
```

For c2 the output is quite different then from c1. For c2 the set memory request is way down from the actual resource usage which makes the service run purely due to its high limit. Having a container and resource request set like c2 could be damaging for the cluster as the container occupy memory that normally should only be accessed when containers are experiencing extra load. In this case, we have an underallocation rather than an overallocation, and therefore we will not be able to consider the saved memory for this container. Here we actually increase the memory request, but it is done to increase the healthiness of the cluster rather than saving memory. However, we will still include the before and after values. Before the model the R/L was set to 500MB/2GB, and after we set the R/L to 768MB/1017MB, which is a 53.6% increase in resource request and a 49.15% decrease in limit.

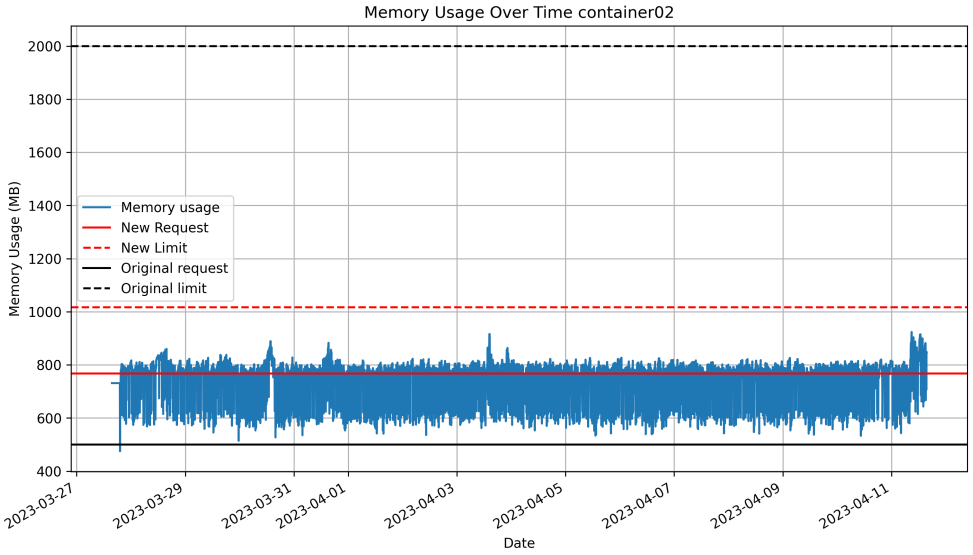

Figure 4.11: Visualization of c2's original and new memory request and limit. Notice the jagged memory usage and exceedingly low original memory request

In a manner similar to c2, c3 is also underallocated, which leads our model to recommend increasing its memory request. C3's original request line is closer to

the actual allocation compared to c2, though it remains slightly underallocated. Nonetheless, the original request limit is approximately 2000% higher than the memory usage, which indicates an overestimation of the set limit. It's worth noting that setting a high limit by itself does not impact the cluster; however, having multiple containers with limits close to 2000% peaking simultaneously could result in the shutdown of random pods to conserve memory, as the usage would far exceed the requested memory. C3's behavior is less erratic than c2's and does not exhibit an increasing pattern like c1's. This makes c3 easier to model, and as a result, the new request consistently hovers just above the request line at all times.

The original R/L was set to 50MB and 1.5 GB, and after running our LP model the suggested R/L is 65MB/71MB. This is a 30% increase in requested memory and a 95.26% decrease in requested limit.

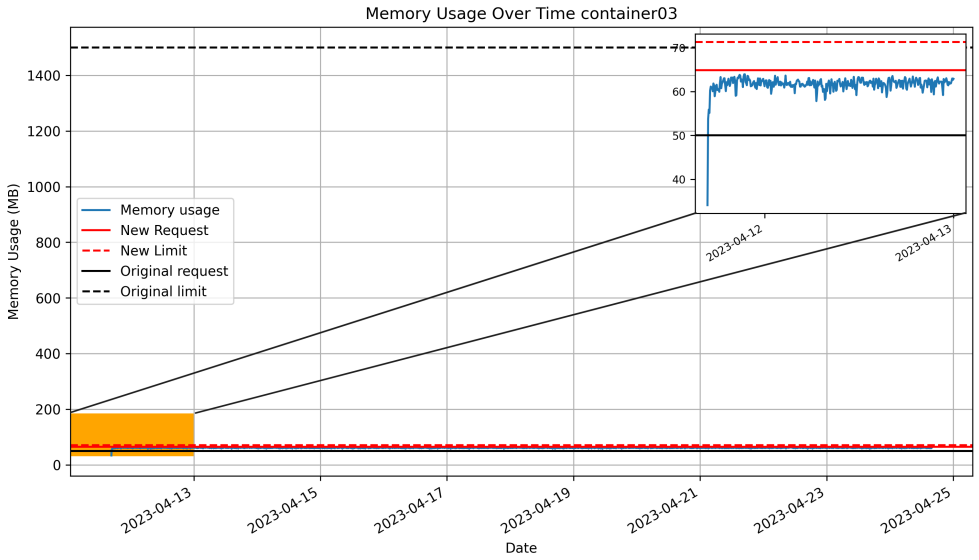

Figure 4.12: Display of c3's original and new memory request and limit. In terms of fluctuations, c3 is more erratic than c1, but less than c2. Pay attention to the heavy overallocation on the original limit as well as the underallocation of the original request in the zoomed-in plot, much like c2

4.4.1 Evaluation of the Linear Programming Model

To evaluate performance, we've created a key performance indicator (KPI) called the Memory Request Efficiency Index (MREI), which measures the percentage of time data points (memory usage) are below the request line. A higher MREI is desirable, but an overly high value may indicate overallocation during the given period. The ideal target is set at 95%, with a margin of +/- 4%, which implies that

we allow some resource usage to be above the request line. The idea behind this is that having the resource usage being above the request line 95% of the time should account for a couple of containers peaking from time to time. This target accounts for occasional memory usage spikes since we assume that peaks will not happen across multiple containers at the exact same time while saving 10% of resources compared to a higher target.

A memory request consistently above resource usage yields a 100% MREI, indicating overallocation. One drawback of this KPI is that it only takes into account whether the value is above or below the resource request, not the extent of the deviation. For instance, a 100% score means the resource request is always above actual memory usage, but MREI doesn't reflect if the request line averages 1MB or 100MB above resource usage. In other words, MREI is more effective in revealing a poor memory request rather than *precisely* how good it is.

However, the short-lived peaks and troughs we have observed thus far help balance this out. The MREI allows us to discern performance differences, even in instances of prior underallocation. If original requests were set lower than actual usage, our model may appear to be increasing requests and limits, contrary to our goal of reducing them. In cases of underallocated containers, the MREI will be low, so when we increase the request and limit, the MREI will rise.

We will use a similar KPI for the predictive models, though the evaluation method will vary slightly for those models.

Definition 2:

The Memory Request Efficiency Index (MREI) measures the percentage of time the memory request line is above memory usage. The ideal MREI is 95%, with a margin of +/- 4%.

In addition to the MREI, we also include the original request/limit, the new R/L, the difference between the old and new, going from old to new, and the difference in percentage. The combined evaluation of our linear programming model for these 3 containers is shown in Table 4.1. A negative difference in the model indicates resources saved, or a reduction of allocated memory, which is what we want to achieve in this process unless the container is originally underallocated. A positive difference would indicate that our model suggests an *increase* of resources. Memory limits were evaluated as a supplementary KPI. However, reducing memory limits does not directly influence the Kubernetes cluster or decrease

allocated resources, making it a lower priority for this project.

The results in Table 4.1 offer a number of insights based on our analysis thus far. This table does not encompass memory usage for individual containers. To review memory usage, please refer to the plots for c1-c3 and the accompanying text above. For c1, the MREI is reduced by 2%, bringing it closer to the optimal MREI value of 95%. In this case, allocated memory is decreased by 572MB which is a great feat. Both c2 and c3 require additional memory resources due to being heavily and mildly underallocated, respectively. C2's MREI is capped at 44%, indicating that the LP model may not be suitable for all container types, particularly those with significant fluctuations like c2. For such containers, increasing the safety margin may be necessary to achieve a higher MREI.

The model appears to perform well for cases that exhibit a slight upward trend and mild fluctuations, as it either touches or slightly surpasses the request line by the end of the dataset, getting the MREI closer to 95% rather than 100%. For mildly fluctuating containers without any trends, usage remains entirely below the request line. In all instances, the requested limit was reduced by at least 49%. Finally, the total MREI increased from 33% to 80%, demonstrating a significant improvement in resource allocation as well as reducing the allocated resources by a total of 289MB, even when handling underallocation for c2 and c3.

Table 4.1: Comparison of initial and updated request/limit for c1-c3 memory usage (values in MB). The emphasized values demonstrate a significant increase in MREI for c2 and c3, while c1 experiences a slight decrease but also exhibits a substantial reduction in resource allocation. All percentages are rounded to the nearest whole number

	Orig. MREI	**New MREI**	**Orig. R/L**	**New R/L**	**Diff. R/L**	**% Diff. R/L**
c1	100%	**98%**	736/736	164/185	**-572/-547**	**-78/-75(%)**
c2	<0.1%	**44%**	500/2000	768/1017	+268/-983	+54/-49(%)
c3	<0.1%	**100%**	50/1500	65/71	+15/-1429	+30/-95(%)
Total	33%	***80%***	1286/4236	997/1273	-289/-2951	-23/-70(%)

The linear programming model does not try to predict any values and bases itself only on historical data using conservative margins to make sure the service has enough resources. The LP model will even handle erratic behavior since the limits should be able to handle the fluctuations while the average is handled by the set resource request. The main weakness of this model is containers with an exponential increase in usage since the average of the data points will not be able

to keep up with the increase, even though we have safety margins for the limit. While the result from this model is static, it can be run at intervals such as every other week, to re-calibrate the recommended request and limit to be set. In our case, with 2 weeks of data, we could consider using the values gotten from the model for the next 2 weeks, before re-running the model and getting variables to use for another 2 weeks.

The safety margins set for this model are seen as quite conservative with a lot of wiggle room. One could spend more time to make the model stricter by lowering the safety margins, but that could go at the expense of the containers' performance. It is important to keep in mind that lowering the allocated resources should not affect performance to a high degree. The safety margin for limits is based on the peaks in the data, and for containers that have high, but not frequent peaks it would leave a lot of space or area between the limit line and the actual usage. Therefore calculating the area between usage and the new set of requests and limits should be used when estimating variations of this model, and should be considered during the evaluation. However, at this point in the project, especially with the data we have, that is not required since the resource allocation is already quite overallocated, which would make lowering the safety margin even further only achieve a few additional percentages, which in this case would be high risk with low reward.

4.5 Auto Regressive Integrated Moving Average (ARIMA)

Having created a basic algorithm with safety margins set to certain percentages shows promise, and reduces the allocation of plenty of MB of memory. However, we want to be more dynamic with the set safety margins, and one approach to this is to predict future usage, and set requests and limits accordingly. For this, we discussed whether to use linear regression or ARIMA for our approach after having analyzed the containers. This model and the following prediction models as well as the evaluation of these statistical models will be fulfilling step PI-4c and with that conclude Phase I.

While linear regression assumes a linear relationship between the dependent variable and one or more independent variables, the ARIMA model does not rely on the existence of independent variables. ARIMA only uses the time series data itself (the past values and forecast errors) to make predictions, thus making it a

fitting model to predict our data.

ARIMA models consist of three components: autoregressive (AR), differencing (I), and moving average (MA). The model is represented as ARIMA(p, d, q), where p is the order of the autoregressive component, d is the degree of differencing, and q is the order of the moving average component. Before building the model we need to take care of the assumptions about the data and determine the parameters. For this approach, a guide for building an ARIMA model was followed [47], with adjustments being made throughout. Even though a guide was followed and the ARIMA library was used, not everything worked out of the box, which resulted in changes to the code. Edits were also made to how the figures were plotted, to stay compliant with our plotting so far. Further on, *p,d,q* is discussed and plotted for c1. However, for c2 and c3, we only include plots of when the ARIMA model is applied.

4.5.1 Determining the Differencing (d)

To determine the order of differencing we checked the autocorrelation plot and found out that the data was non-stationary and differencing had to be done. We also did a mathematical test using *The Augmented Dickey-Fuller* (ADF) test which aims to reject the null hypothesis that the given time-series data is stationary. It calculates the p-value and compares it with a threshold value or significance value of 0.05. Figure 4.13 shows our data with differencing going from 0-2 with the ADF value on top of the top figures.

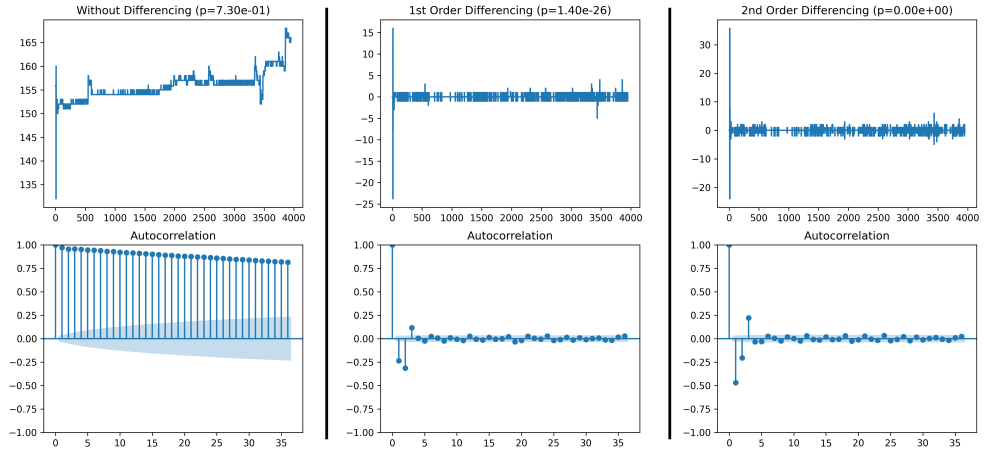

Figure 4.13: Order of differencing determination using autocorrelation plots and ADF test. Each column features the original data after differencing (top) and its corresponding autocorrelation (bottom). The font of the axes are small but the most important point is the display of the patterns

The 1st-order differencing and the 2nd-order differencing are very similar, and since our goal is to make the data stationary with the minimum amount of differencing required we went for the 1st-order differencing. By using the 2nd order of differencing we remove more information about the underlying data, introduce additional noise, and make the model slower. Since the 1st and 2nd order is so close in p-values, we stay conservative and conclude that there is more value by using the 1st order and see how the model will perform.

4.5.2 Determining the Autoregressive (p) and Moving Average (q)

To decide upon these values we use the autocorrelation function (ACF) from the previous section, as well as the partial autocorrelation function (PACF). These functions provide us with an initial estimate for p and q. In Figure 4.14 we have plotted the ACF and PACF for container01's memory usage. From the figure, we see that both the ACF and PACF plots show a significant spike at lag 1, after which the values drop and mostly remain within the blue-shaded confidence interval. This suggests that an initial value of "1" would be a good starting point for both p and q. Based on the observations so far, we decided to start with an ARIMA model with the following parameters $p=1$, $n=2$, $q=1$. Next, we will fit the ARIMA model.

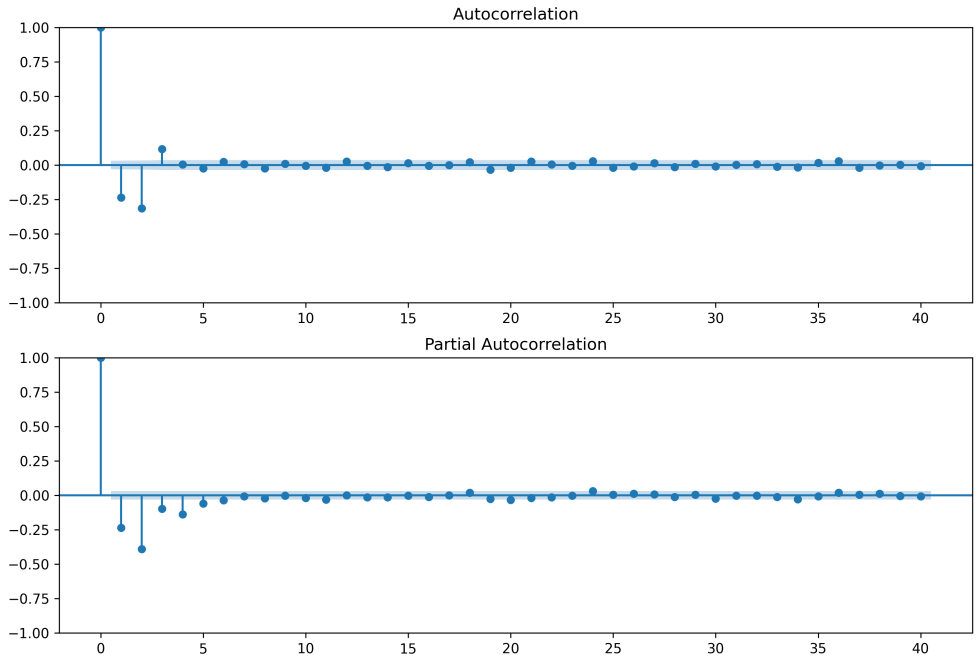

Figure 4.14: ACF and PACF plots for c1's memory usage

4.5.3 Fitting the ARIMA model

We applied the model to c1-c3 using the chosen parameters, resulting in varying outcomes. The algorithm can be found in Code Listing 4.6, while the complete coding example is provided in Appendix B.3.2. We opted for an 80/20 training/test split, maintaining a consistent approach across all containers rather than customizing splits for better alignment. This simple and blind approach has its advantages in an exploratory context, such as this study.

Another take could be to first analyze each container's usage before selecting the train/test split to optimize a selection that accounts for individual container characteristics. Expanding on this, multiple clusters with similar attributes could share the same split. However, we did not pursue this approach, as optimizing splits is beyond our scope. While creating clusters based on workload is within our scope, time constraints prevented us from doing so in this case.

Further on c1-c3 is fitted into the ARIMA model and the results will be discussed after each plot and lastly evaluated together, much like what was done for the LP model.

Code listing 4.6: Code snippet for running the ARIMA model

```
1 train_size = int(len(memory_usage) * 0.80)  # 80% of the data for training
```

```
train_data = memory_usage[:train_size]
test_data = memory_usage[train_size:]

p,d,q = 1,2,1

model = ARIMA(memory_usage, order=(p, d, q))
results = model.fit()

predictions = results.predict(start=train_size, end=len(memory_usage) - 1, dynamic=
    True)
```

For C1 (see Figure 4.15), the train/test split is unfortunate, as two peaks occur immediately after the training period, leaving the model unprepared. In this specific case, a 90/10 split would yield better predictions. The forecast exhibits a slight upward trend, which seems reasonable when looking at the training data. However, the prediction is deemed inadequate since it does not closely match the test data for the majority of the time. Without using constant re-calibrations throughout the fitting we see that ARIMA struggles with memory usage with an upwards trend with peaks.

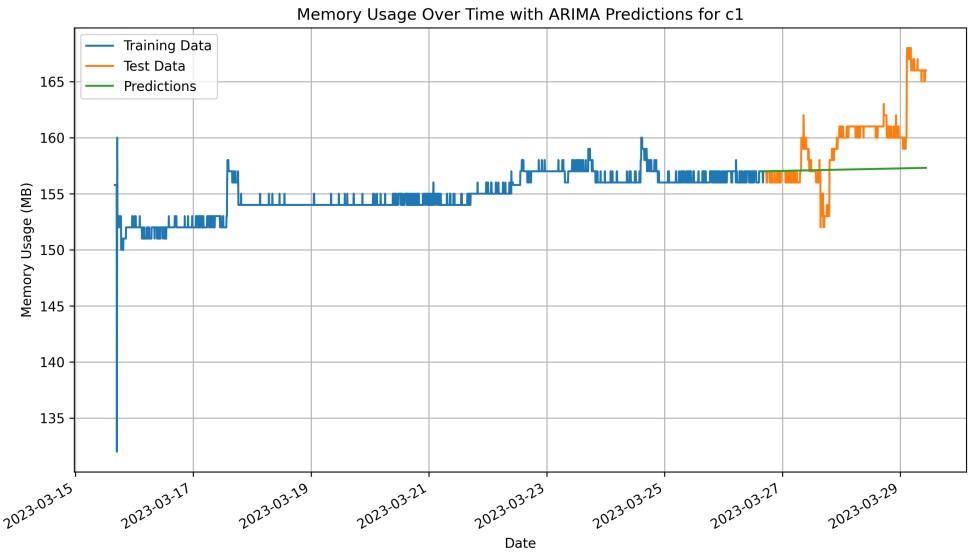

Figure 4.15: ARIMA plot for c1 illustrating the challenges faced due to an unfavorable train/test split and the model's struggle with upward trending memory usage and peaks

C2's predictions deviate more significantly from the actual values compared to those of C1. According to Figure 4.16 our model forecasts a downward trend in the coming days.

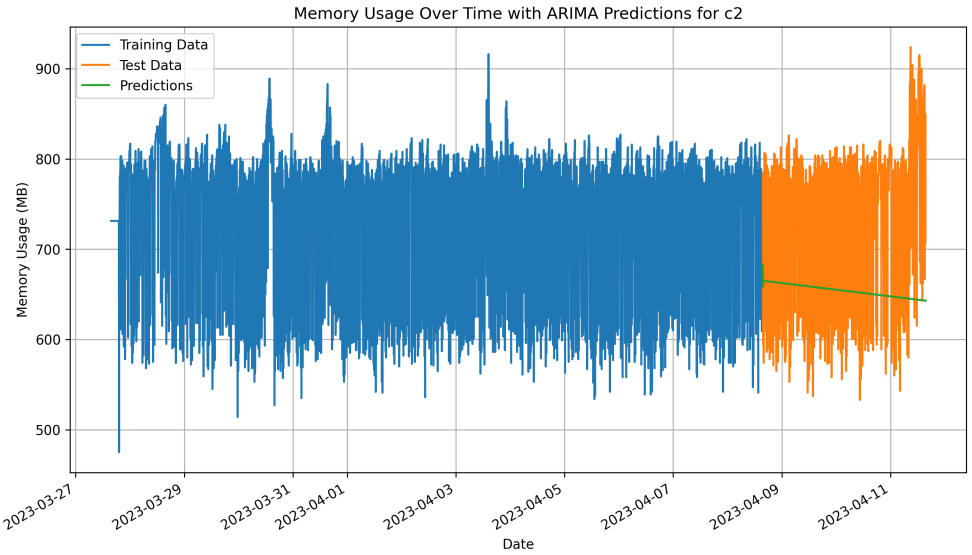

Figure 4.16: ARIMA plot for C2 highlighting the model's significant deviation from actual values, difficulty in capturing trends, and inability to predict the final peak, resulting in poorer performance compared to C1

Although it might be difficult to discern from the plot, both the average and density exhibit a downward trend. The model reflects this trend, as its predictions also show a slight decrease. The model also entirely fails to capture the final peak that emerges towards the end. Ultimately, similar to C1, our model struggles to predict C2 accurately, and in this case, the performance is even poorer.

In the case of c3, seen in Figure 4.17, towards the end of the training set, there is a decrease in the frequency of low points, which leads the model to predict higher usage for the upcoming days. However, as observed in the test data, this is not the case, as the low points start to appear more frequently again. Comparing C3 with C1 and C2, the predictions for C3 are the most accurate, even though they are still not ideal. In terms of resource allocation, basing it on C3's prediction would result in overallocation for the container, whereas relying on C1 and C2's predictions would lead to underallocation.

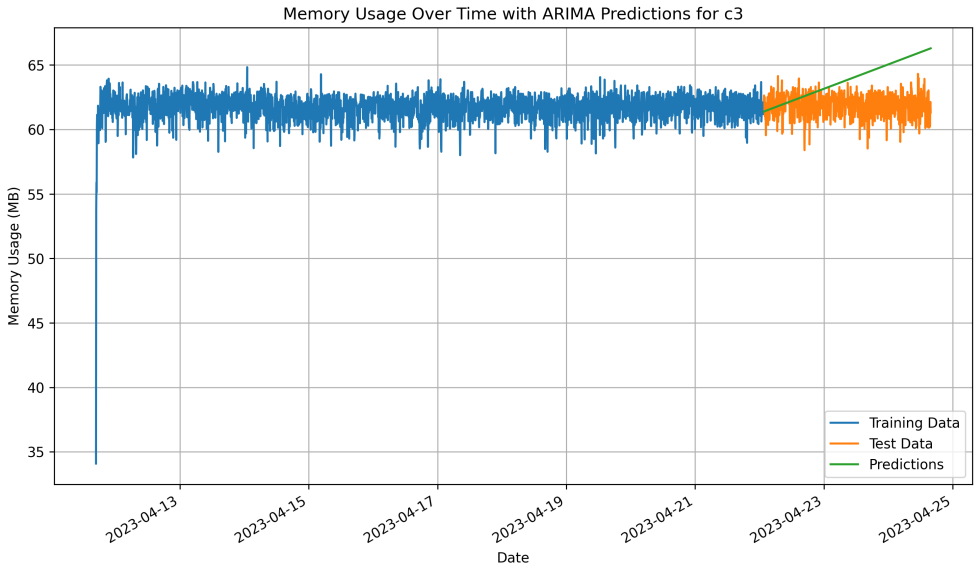

Figure 4.17: ARIMA plot for C3 showing a decrease in low points frequency in the training set, leading to higher usage predictions

4.5.4 Evaluation of the ARIMA Model

To assess the performance of this model, we will use MAE, MAPE, RMSE, and the Test Data Underprediction Rate (TDUR) as evaluation metrics, with TDUR and MAPE serving as our main KPIs. MAPE is chosen as a co-KPI because it is well-suited for datasets with varying scales, as it calculates the mean absolute error as a percentage. Table 4.2 shows that the ARIMA model did not perform exceptionally well, with an average TDUR of 49.1% and a MAPE of 6.7%. However, these numbers present a somewhat better picture than the plots, as a MAPE of 6.7% might be deemed acceptable in some cases.

For c1, a MAPE of 1.9% can be considered satisfactory, especially when taking into account that it missed two consecutive peaks from the training data. This result might provide a foundation for further improvement, although the prediction value is still above the test data only 34.6% of the time, according to the TDUR.

C2 is evidently the most challenging container to model, both for the LP model and the ARIMA model, with a MAPE of 14.8% and a TDUR of 27.6%. This indicates that the predictions frequently underestimate the actual resource usage, and the prediction error is considerably higher compared to c1 and c3. C3's TDUR of 85% is the highest among the three containers, making it the most suitable candidate for suggesting resource allocations. This is because its predictions are above

the actual resource usage most of the time, which ensures that the allocated resources are sufficient to meet the requirements of the container.

In the case of c1-c3, we observe that the models perform quite well for the first predicted day before becoming too static, going too high, or too low. Reducing the test duration could yield better results for the ARIMA method, making it applicable if the algorithm were run daily to predict the next day. This may be an acceptable routine to adopt, although we would still prefer to extend the time periods, as setting new requests and limits requires a container restart, which could potentially lead to availability issues. Nevertheless, initiating daily analyses could be a good starting point.

Additionally, examining c1-c3 in this model provided us with a clearer understanding, as request and limit lines that are either over or under the resource usage do not interfere with the plot.

Table 4.2: Evaluation of ARIMA model performance for c1-c3 using MAE, MAPE, RMSE, and TDUR

	TDUR	**MAPE**	**MAE**	**RMSE**
c1	34.6%	**1.9%**	3.1	4.0
c2	27.6%	**14.8%**	112.7	124.4
c3	**85.1%**	3.5%	2.1	2.6
Average	49.1%	6.7%	39.3	43.7

Unlike the LP model, we have chosen not to recommend request or limit values for this predictive model, as the outcomes are quite diverse, and it is not feasible to establish practical resource requests or limits based solely on the predictions. Combining the predicted values with another model, such as our LP model, might yield a suitable resource allocation, but devising a solution based on these three containers appears to be somewhat of a stretch. We will reevaluate this approach in Phase II after examining the results in a broader context.

Definition 3:

The Test Data Underprediction Rate (TDUR) measures the percentage of time the prediction is above the test data and will act as a KPI for evaluating the predictive models. This KPI has the same limitations as MREI which was discussed during the LP model.

4.6 Facebook's Prophet Model

The original plan involved the analysis of three predictive models. However, due to time constraints, the fitting of a fourth model, including the LP model, could not be completed during Phase I. To gain a deeper understanding of the Prophet model beyond the knowledge acquired from the papers introduced in the background chapter [38–40, 42], a guide found on kaggle.com was used as a starting point for working with the model [48].

Without incorporating further adjustments to the dataset, such as seasonality, weekends, nighttime, and other regressors, a basic model was employed, as demonstrated in Code Listing 4.7. Prophet also allows for features that alter the uncertainty intervals and outliers. Further adjustments will be discussed in Phase II after having applied and seen the results of the model from multiple containers.

In the coding example, the data is initially transformed to ensure compatibility with the model, using both date and memory usage. Subsequently, the same train/test split is applied, and the model is trained on the training set. Finally, the prediction is displayed using a 5-minute frequency, as the existing datapoints are separated by this duration in terms of minutes between each datapoint.

Code listing 4.7: Request to Prometheus server

```
model = Prophet(changepoint_prior_scale=0.01)

df = pd.DataFrame({'ds': df1['Date'], 'y': df1['Containers: Memory usage']})
df['ds'] = pd.to_datetime(df['ds'])

# Split the data into train and test sets
train_size = int(len(df) * 0.8)  # 80% of the data for training, 20% for testing
train_df = df[:train_size]
test_df = df[train_size:]

# Fit the model on the training data
model.fit(train_df)

# Make predictions on the test data
future = model.make_future_dataframe(periods=len(test_df), freq='5min',
    include_history=False)
forecast = model.predict(future)
```

For the Prophet model, we modified the visualization slightly for two primary reasons: first, to provide a perspective on the previously examined datasets, in this

case, using observation points (represented by black dots); and second, to accommodate the uncertainty interval (indicated by the light blue shaded area). The purple dots signify the test data, and the solid blue line represents the predicted values. The conventional method of displaying this model is through a scatter plot featuring observation points (the training set), as opposed to a line chart. The complete script, including the plotting code, can be found in Appendix B.12.

The first forecast depicted in Figure 4.18 exhibits a slight downward trend, which contrasts with the ARIMA model that displayed a minor upward trend. This indicates that the two models have differing predictions for the future of c1 in the coming days. The Prophet model is capable of detecting and fitting trend changes, utilizing the frequency and magnitude of observed trend changes in the data's history for future predictions. This is why the uncertainty interval (represented by the blue-shaded area) extends considerably and is able to capture the second peak quite accurately. When only focusing on the forecast, represented by the solid blue line, the model may not appear very effective. However, when considering the uncertainty interval, the model's potential usefulness becomes apparent.

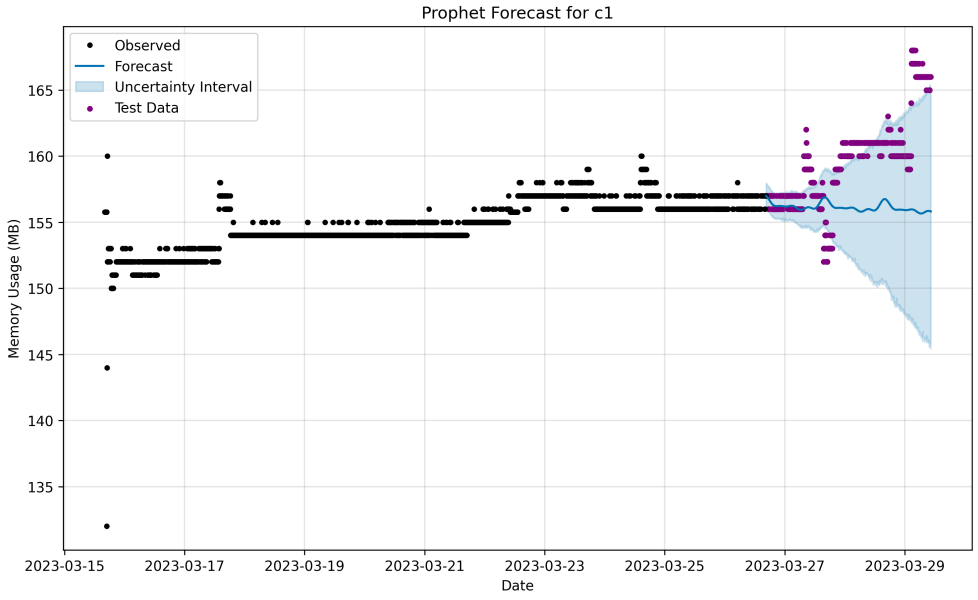

Figure 4.18: Prophet model's forecast for c1. The solid blue line indicates Prophet's forecast, black dots represent observations, and purple denotes test data. Prophets uncertainty is observed as the light blue shade. The prediction has a slight downward trend in contrast with the upwards trend of the test data. However, the uncertainty interval nearly captures the highest observations of test data

The forecast for C2 using the Prophet model presents a well-balanced average line. Instead of leaning more towards the denser areas, it lies in between, taking a cautious approach, which consequently creates a distance from all observations, potentially leading to poorer performance with our evaluation metrics. It could be argued that, in general, a model suggesting average data when the observations are so split may not be suitable for server or container usage. However, in our case, with the implementation of both request and limit parameters, the findings might still be useful, particularly considering our previously established optimal target of 95% +/- 4 ratio.

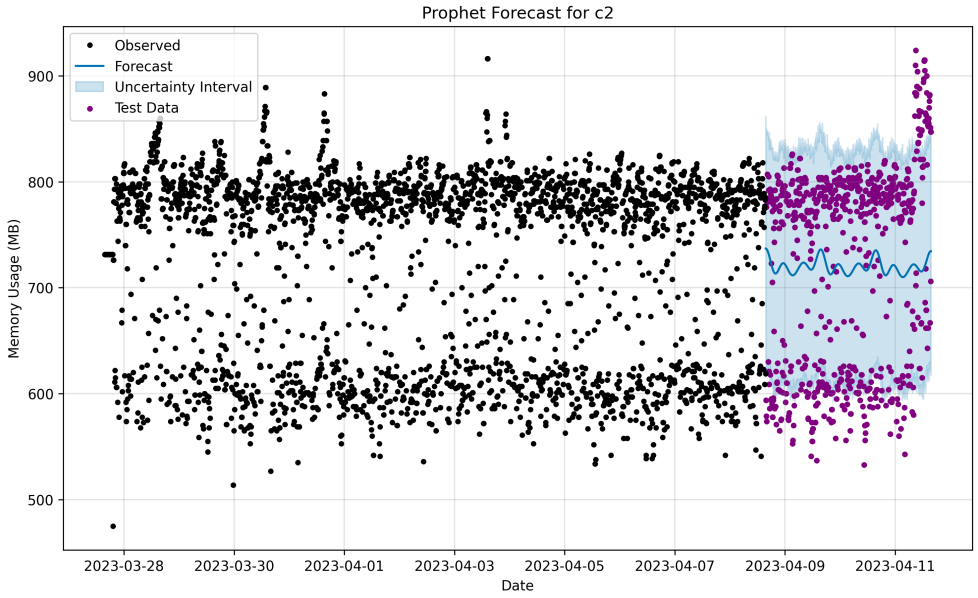

Figure 4.19: Prophet model's forecast for c2. The plot shows a cautious prediction line amidst split observations

Observation 4:

The uncertainty intervals generated by the Prophet model effectively capture the magnitude and frequency of trend changes observed in the training data, allowing it to accurately account for most peaks in the test data

Similar to the ARIMA model, this prediction exhibits a slight downward trend, which is not immediately evident in the figure based solely on the training set observations. The uncertainty interval extends significantly, encompassing most of the highest observations, due to the presence of occasional peaks in the training data. However, it does not quite reach the last peak. The prediction is somewhat

closer to the density of the upper observations but generally lies in the middle of the two dense areas. The uncertainty interval's ability to capture a wide range of possible outcomes is particularly interesting, as it provides valuable insights into the potential variability of the model's predictions and helps to inform decision-making under uncertainty.

For C3, including outliers in the plot creates a dense appearance, although the observations are similar to those seen for C2, with two highly dense areas and lower fluctuations, as the memory usage generally remains between 60MB and 65MB. The uncertainty interval is somewhat obscured by the test data; however, it appears to capture the high values for this container as well. In contrast to the ARIMA model for C3, which predicted a moderate increase in memory usage over the next few days, the Prophet model's prediction does not seem to indicate any significant movement in either direction.

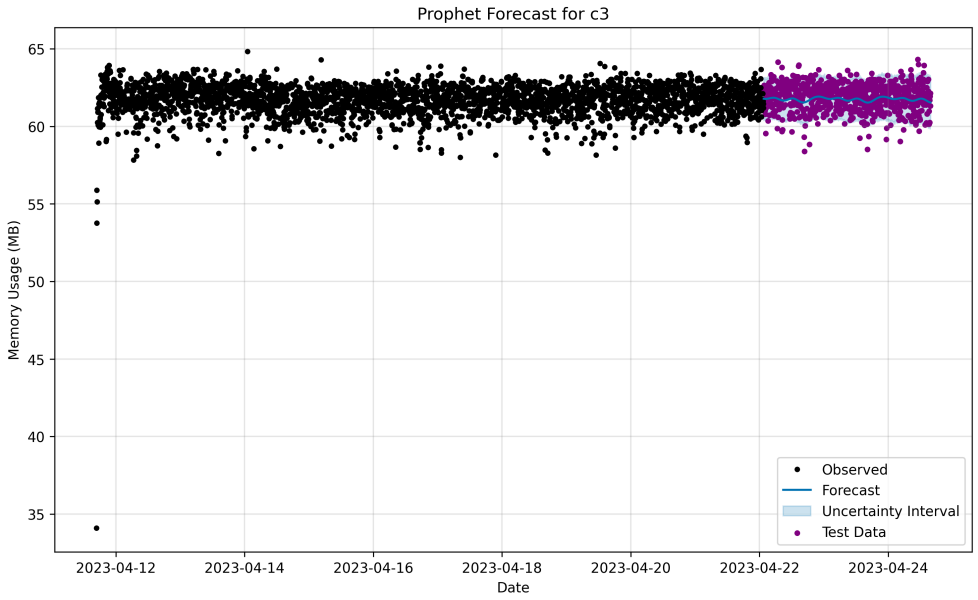

Figure 4.20: Prophet model's forecast for c3. A dense appearance is observed due to outliers from start-up phase, with average fluctuation of around 5MB (between 60MB and 65MB) amidst erratic observations

Overall, Facebook's Prophet model revealed some intriguing insights, particularly when considering the combination of the uncertainty interval and predicted values. Prophet typically avoids leaning heavily towards increases or decreases, instead maintaining a stable position based on more recent values, while using the

uncertainty interval to account for potential peaks and troughs. With the plain model and without making too many adjustments, it would be interesting to base the memory request on slightly altered prediction numbers and set the request limit based on the peaks of the uncertainty interval, with appropriate safety margins. This combination works effectively, but our evaluation will focus primarily on the actual prediction for this model. Nevertheless, the uncertainty interval will be revisited as a compelling argument in favor of using the Prophet model.

he evaluation model employed for both ARIMA and Prophet will remain consistent in order to maintain the sequence, utilizing TDUR and MAPE as key performance indicators, as displayed in Table 4.3. In general, Prophet's performance for c1 was weaker, showing a 24.08% rise in MAPE (lower is preferred) and a 53.6% reduction in TDUR (higher is preferred). Conversely, for c2, Prophet outperformed in all aspects, with MAPE decreasing from 14.8% (ARIMA) to 12.1% (Prophet) and TDUR increasing from 27.6% (ARIMA) to 32.3% (Prophet). As for c3, the lowest MAPE was observed across both models at 1.2% and an MAE of 0.7, which is deemed highly satisfactory. However, the TDUR experienced a 50% reduction from the ARIMA model, diminishing the overall performance on c3.

In summary, ARIMA delivers a superior average in TDUR, while Prophet demonstrates better results in MAPE. Each model excels with distinct usage patterns: ARIMA is effective for containers with discernible trends regarding usage direction, while Prophet is better suited for more unpredictable containers where seasonal variations are difficult to forecast.

Table 4.3: Evaluation of Facebook's Prophet model performance for c1-c3 using MAE, MAPE, RMSE, and TDUR

	TDUR	**MAPE**	**MAE**	**RMSE**
c1	16.1%	2.4%	3.8	4.9
c2	32.3%	12.1%	85.9	93.3
c3	**39.2%**	**1.2%**	0.7	0.9
Average	29.2%	5.3%	30.1	33.0

4.7 Summary: Evaluation of Models (PI-4c)

When evaluating the linear programming model, we use a KPI that measures the distance between the original resource request/limit and the new suggested re-

source requests. For the Linear Programming (LP) model, we consider both requests and limits. As a result, the KPI represents the decrease in resource request achieved by the corresponding model. A decrease, denoted by a negative percentage, indicates that memory resources are being conserved, while a positive percentage signifies an increase in the resource request, allocating more resources for the container. For our first model, the LP, we do not measure MAE or MAPE, as it does not involve forecasting and has no other metrics to evaluate besides the mentioned KPI.

Basing resource allocation solely on the LP model worked impressively well, suggesting requests and limits that effectively managed underallocation and overallocation while providing a safety margin. *However, it is essential to note that this model appears advantageous from a perspective where resources were already either under or overprovisioned.* An already established resource allocation based on average usage with simple safety margins would not benefit from this model. Therefore, either the model needs to become more advanced, thereby approaching the actual resource usage and decreasing the gap between the request lines and the usage, or it can be used in combination with a predictive model. For the predictive models, we assess prediction accuracy using another KPI and the MAPE.

For the statistical models, we adopt a different approach and measure how well they predicted the usage of the test data. One of the KPIs for predictive models is the number of times the prediction is above the resource usage, and we aim for as high a number as possible. The higher the count, the better the model performed on the container. Nevertheless, based on the performance of the models on c1-c3, we have not yet decided to recommend resource allocation, as their performance varies significantly and requires further evaluation. So far, both models show potential. If we could recalculate and rerun the ARIMA model each day, which seems to perform well on the first predicted day, it may provide sufficient predictions with good performance for most containers, except for the most erratic one, c2. Prophet, with its uncertainty interval, is also worth further investigation, and using it in combination with our LP could be valuable for suggesting resource requests for multiple containers.

Our findings throughout Phase I address our first research question: "*How can time series forecasting models be applied to predict resource usage for individual containers in Kubernetes environments, and what are the challenges and benefits*

associated with using these models to ensure prediction accuracy across various container usage patterns?" We have explored two forecasting algorithms, Prophet and ARIMA, and one algorithm that suggests resource allocation based on historical data and safety margins.

The performance of both Prophet and ARIMA varies significantly when predicting memory usage, with some containers showing excellent results while others do not. This variation makes it challenging to recommend a single statistical model for all situations. However, in a selective scenario, both models can provide valuable insights.

Applying these models to multiple random containers may yield average or poor results overall. Fine-tuning the models is also challenging since different tuning approaches work better for specific types of containers. A potential solution could be to cluster containers based on their resource usage patterns and apply tailored models to each cluster, thereby ensuring prediction accuracy across various containers.

Chapter 5

Results - Phase II: Evaluating Efficiency of Models Across Multiple Containers

Phase II will build upon the knowledge and setup established during Phase I, expanding it to a larger environment. In accordance with the table presented in the approach chapter, we will process all containers with available resource requests for a container on a single node within one month (PI-5ab), as long as the memory usage is at least one week old. Containers that restart periodically, with each runtime lasting less than a week, will not be included in this case.

For containers that intermittently run, we do not check for potential changes that may have occurred; therefore, in many cases, we cannot guarantee that the same container has the same tasks for the new runtime period. Short-lived containers (less than one week of aliveness) that may be job-based or run only once and never again during the one-month period will not be evaluated. The primary reason for this exclusion is the limited value of evaluating such containers, as they could be test runs or one-time jobs, which could differ significantly from other containers running for more than a week.

Both ARIMA and Prophet have demonstrated various strengths and weaknesses so far. In this phase, we will evaluate these models using the Test Data Underprediction Rate (TDUR) and MAPE. By assessing the models on all containers within a specific time interval on a single node, we will complete step PI-6a, and then determine the best model for predicting resource usage for given con-

tainers, thus fulfilling activity PI-6b.

5.1 Model Performance Evaluation for Predictive Models

The evaluation period for all containers spans from April 1st to April 30th. The containers C1-C3 used in Phase I differ from those used in this Phase I. Table 5.1 demonstrates the performance of Prophet and ARIMA concerning the evaluation metrics MAPE and TDUR. Throughout this month, most containers operated for more than 21 days, surpassing the time frame that Dynatrace permits for 5-minute data intervals. Consequently, data were aggregated into 1-hour intervals. Although the maximum value for the interval can still be obtained, this may cause the plots to appear different and less erratic. Alternatively, we could focus on new containers that have been running for a maximum of 21 days. However, this approach would significantly reduce the number of available containers, resulting in the loss of valuable data. Out of the 21 containers from which data was gathered, only two containers, c4, and c23 provided 5-minute intervals, and both are marked with an asterisk (*) in Table 5.1.

Table 5.1 presents mixed results, indicating that consistently using either method will not achieve the desired outcomes. The average TDUR is not too poor, with 48.1% for ARIMA and 50.0% for Prophet. Nevertheless, while the average is decent, and a few containers show models closely aligned with our ideal target, several containers exhibit poor TDUR performance (c2, c7, c13, c15, c16, c17, c18, c20). For MAPE, the overall results are satisfactory, although some containers stand out: c6 has over 20% for both models, and c18 and c20 have average MAPE values of 12.2% and 8.1%, respectively.

Table 5.1: Comparison of ARIMA and Prophet model performance in terms of TDUR and MAPE. Percentages are rounded to the nearest whole percentage. On each row, the best result for each evaluation method out of the two models is followed by an arrow pointing upwards. The table shows that the results are varying between the two models and that there is no correlation between the best result for TDUR and MAPE

	TDUR		**MAPE**	
	ARIMA	**Prophet**	**ARIMA**	**Prophet**
c1	45.5%	59.5%↑	0.5%↑	0.6%
c2	3.6%	52.9%↑	2.3%	1.4%↑
c3	66.4%↑	58.6%	0.4%↑	0.8%
c4*	100%↑	77.9%	*0.1%*	*0.1%*
c5	18.4%	57.1%↑	3.4%	3.0%↑
c6	*93.6%*	*93.6%*	29.5%	24.0%↑
c7	*0.0%*	*0.0%*	*0.0%*	*0.0%*
c8	95.7%↑	65.0%	1.6%↑	2.1%
c9	93.6%↑	80.7%	*0.3%*	*0.3%*
c10	77.1%↑	60.0%	0.9%	0.8%↑
c11	92.3%↑	80.7%	2.4%	1.9%↑
c12	53.1%↑	49.0%	*1.4%*	*1.4%*
c13	33.6%↑	7.9%	0.4%↑	0.9%
c14	78.6%	93.6%↑	1.1%↑	1.6%
c15	7.1%	60.0%↑	4.4%	2.6%↑
c16	5.0%	74.3%↑	1.3%	0.9%↑
c17	68.6%↑	24.3%	*0.2%*	*0.2%*
c18	*0.0%*	*0.0%*	4.2%↑	20.1%
c19	51.4%↑	7.1%	0.5%↑	1.2%
c20	0.0%	100%↑	0.3%↑	15.9%
c21*	100%↑	72.4%	+0.1%↑	0.1%
Average	48.2%	50.1%↑	2.5%↑	2.6%

The models themselves exhibited only a 1.9% difference in TDUR and a 0.1% difference in MAPE, making their average performance for predicting container usage quite similar. As previously mentioned, they excel in different workloads, and based on the table, a model may have a poor TDUR but still maintain a good MAPE, or vice versa. We will further investigate certain containers that performed well according to our evaluation metrics, as well as those that did not.

* Containers having a time interval of 5 minutes instead of 60 minutes

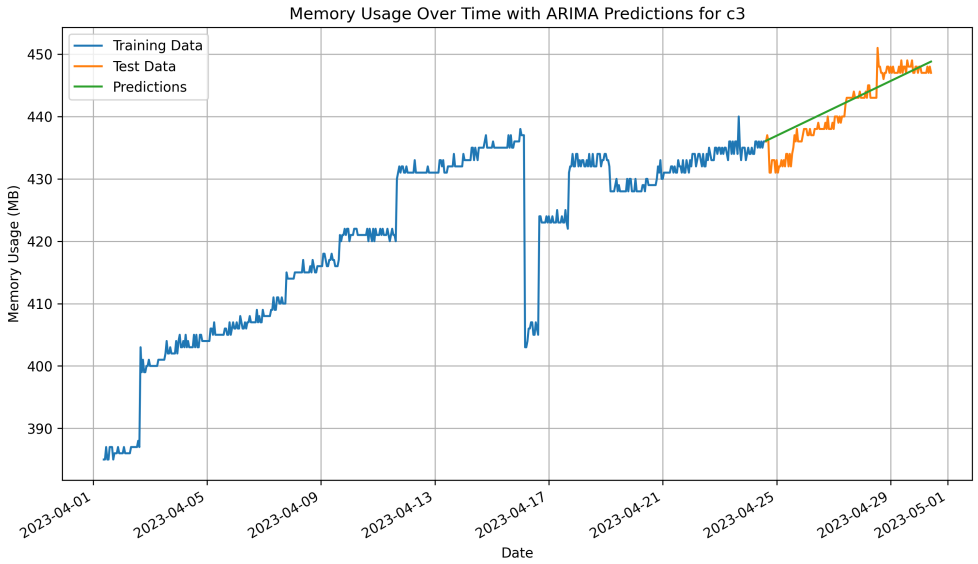

Figure 5.1: Accurate ARIMA model prediction for c3's usage, demonstrating rare yet effective pattern recognition and resource allocation. TDUR of 66.4% and MAPE of 0.4% indicate satisfactory performance, despite minor deviations from the 95% target of TDUR

Figure 5.1 showcases a successful prediction using the ARIMA model. It accurately forecasts c3's usage and lays a solid foundation for resource allocation. Usage patterns like the one in this figure are rare but appear intermittently among the more unpredictable containers. Table 5.1 indicates that c3 has a TDUR of 66.4% and a MAPE of 0.4%, which are considered very satisfactory values. The TDUR is somewhat lower than our 95% target due to some spikes at the end of the testing period, but the overall values are still good, taking into account the safety margins that would be added on top.

Figure 5.2 presents another instance where the ARIMA model's prediction deviates significantly from the test data, primarily due to a 25% drop in the middle of the data set. This pattern seems to confound the ARIMA model entirely. Table 5.1 reflects this poor performance, with a TDUR of 0% and a MAPE of 4.2%, which are considered exceptionally bad, especially for the TDUR. Prophet fared even worse, with an identical TDUR and a MAPE of 20.1%. Predicting this type of usage is extremely challenging. In cases where usage throttles, it is more manageable since we take previous peaks into account when setting resource requests. However, if the usage had increased instead of decreased, we would need to implement high safety margins and set a high request limit to accommodate such patterns.

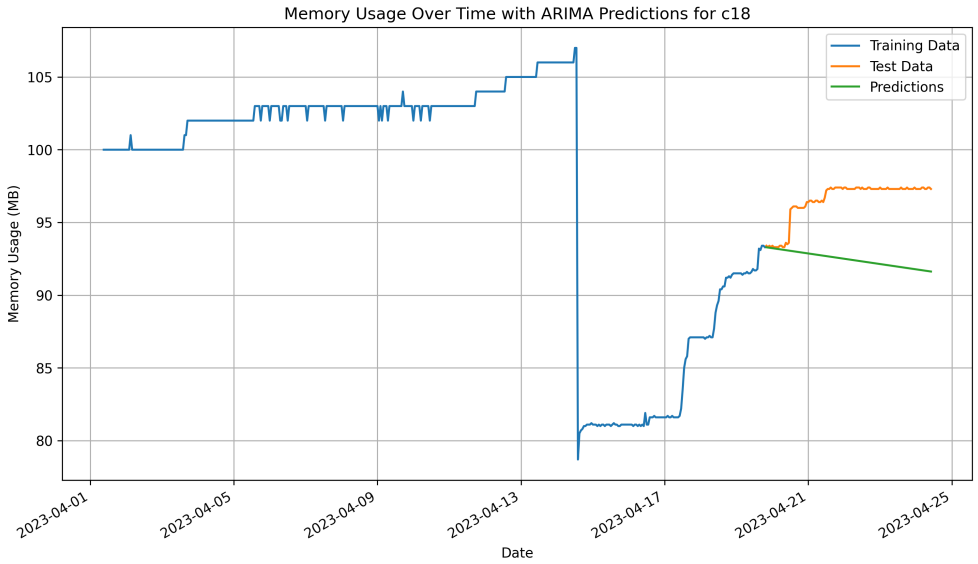

Figure 5.2: Significant deviation of ARIMA model prediction from test data due to a 25% drop, resulting in exceptionally poor performance (TDUR: 0%, MAPE: 4.2%). Illustrates the challenge in predicting and accommodating such usage patterns

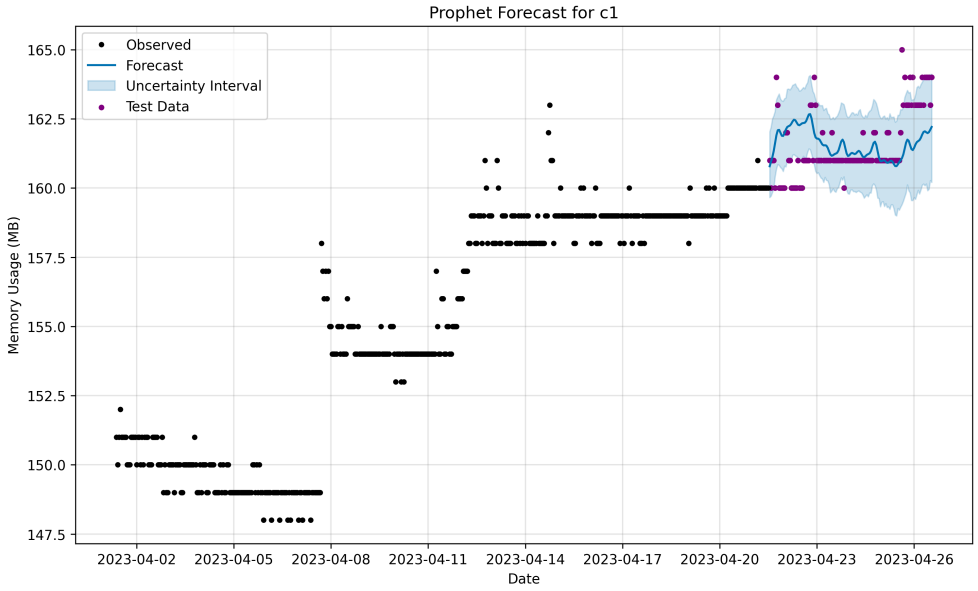

Figure 5.3: Prophet model's accurate prediction for container c1, closely tracking memory usage with a TDUR of 59.5% and MAPE of 0.6%. Uncertainty interval captures most peak deviations, highlighting the model's effectiveness in predicting c1's usage

For container c1, seen in Figure 5.3, the Prophet model closely tracks the memory usage, aligning well with the actual usage. Although it falls slightly short of the peaks in the test dataset by a few megabytes, the uncertainty interval captures most of them. Table 5.1 displays a TDUR of 59.5% and a MAPE of 0.6%, which are considered good values for predicting c1's usage. Again, capturing the highest observation in the test set may be possible by incorporating safety margins.

We have included another example (see Figure 5.4) where the recent data from the training set suddenly spikes, contrasting with the usage drop in Figure 5.2. In this situation, the Prophet model attempts to accommodate the change by exhibiting larger fluctuations in its forecast and expanding the uncertainty interval. While this is not the worst example, with a TDUR remaining at 100%, the MAPE is at 15.9%. It is essential to consider such patterns when fine-tuning models, as they should be manageable since the peaks are often short-lived.

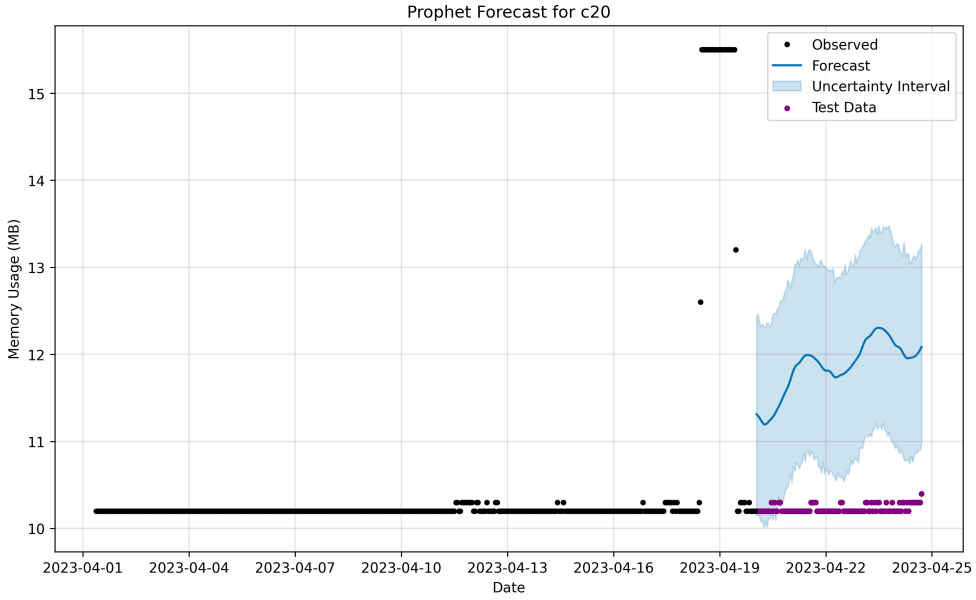

Figure 5.4: Sudden spike in training data contrasted with Figure 5.2, showcasing Prophet model's response through larger forecast fluctuations and expanded uncertainty intervals. Emphasizes the importance of considering such patterns when fine-tuning models, with TDUR at 100% and MAPE at 15.9%

Since the MAPE is reasonably acceptable for both models, we can proceed with either of them. However, we must keep in mind the containers that are challenging to predict, particularly those like c7 and c18, which both have TDUR values of 0. The TDUR, representing the percentage of time the predicted value is above the

actual memory usage, can be managed by adjusting safety margins accordingly. With the Prophet model, we also have uncertainty intervals that are not accounted for in this evaluation table but will be considered during the fine-tuning of the models. These intervals may be useful for adjusting safety margins as needed.

5.2 Prophet + Tuned LP Model (PI-7ab)

We have not yet taken into account the uncertainty interval for the Prophet model in this evaluation, even though we mentioned its importance for determining resource allocation. Based solely on the prediction results, we decided not to proceed with either model alone. However, since the Prophet model showed a 1.9% improvement in TDUR and we have its uncertainty interval, we will assess its performance in contrast to setting resource requests based on the LP model. We would have liked to explore creating uncertainty intervals for the ARIMA model and potentially combining it with LP, similar to the Prophet and LP combination if time allowed.

For this evaluation, we will determine resource requests for the LP model in a similar manner to our container prediction approach, utilizing only the first 80% of data from each container. This allows us to assess the value of predicting containers from node01 and suggesting resource allocation compared to just using the LP model. Based on the results from Table 5.1, we will employ a slightly adjusted LP model capable of utilizing Prophet's predicted values and its uncertainty interval. We will name this new model the Prophet + Linear Programming model (P+LP).

Tuning the model and optimizing its parameters fulfills step 7 PI-7ab, while using the results from the fine-tuned model to propose new resource limits and requests, and evaluating performance completes step 8 *resource allocation* PI-8ab.

The primary enhancements to the new model involve using the average of the predicted value from the Prophet model and adding a 5% safety margin, similar to the LP model, to set the resource request. In terms of memory limits, adjustments have been made for both the LP model and the new P+LP model. For the LP model, the memory limit has been set 15% higher than the highest observed value from the training set. For the P+LP model, we first determine the higher value between the uncertainty interval and the highest recorded value from the training set. For the finalized P+LP including plotting see Appendix B.13.

All evaluations done in Table 5.2 is done on the test data for each container. This table is similar to the table that evaluates the LP model in Results Phase I. The KPI which is the Memory Request Efficiency Index (MREI) evaluates the percentage of time the resource usage is underneath the suggested resource *request*. While a high KPI is good, a KPI that is 100& indicates overallocation. However, we do not examine how much of an overallocation that is in place for an MREI of 100%. The amount of overallocation would also be interesting to examine but is indicated by how much we reduce the original request for the new set request. A minus (-) difference indicates that the new resource request/limit is lowered, which means that we have reduced the resource allocation, which initially is the goal while maintaining a high MREI.

In Table 5.2, we observe the MREI and the difference in allocated resource requests when applying the models to the 21 containers. Both LP and P+LP yield good MREI results, with the exception of container c8 for LP and c18 for P+LP. For c8 (see Figure 5.5), LP has an MREI of 0%, while P+LP has an MREI of 100%. C8 is a heavily underallocated container, requiring a 78.5% increase in resource requests to cover it. The figure shows that, due to the upward trend in Prophet's prediction, the resource request is set much higher than the LP, which uses the average of the training set. We also notice differences in the limit lines: in this case, since the uncertainty interval is higher than the highest observed value from the training set, P+LP uses the maximum value of the uncertainty as a safety margin threshold, compared to LP, which uses the highest observed value. The green lines indicate P+LP's request and limit, while the red indicates LP's request and limit.

For c18, LP achieves a better-set value compared to Prophet's prediction due to Prophet predicting a downward trend. In both edge cases, the high peaks in the test data are confidently covered by the limits set, and with an MREI of 88.9% (including the edge cases), those exceptions will be accommodated.

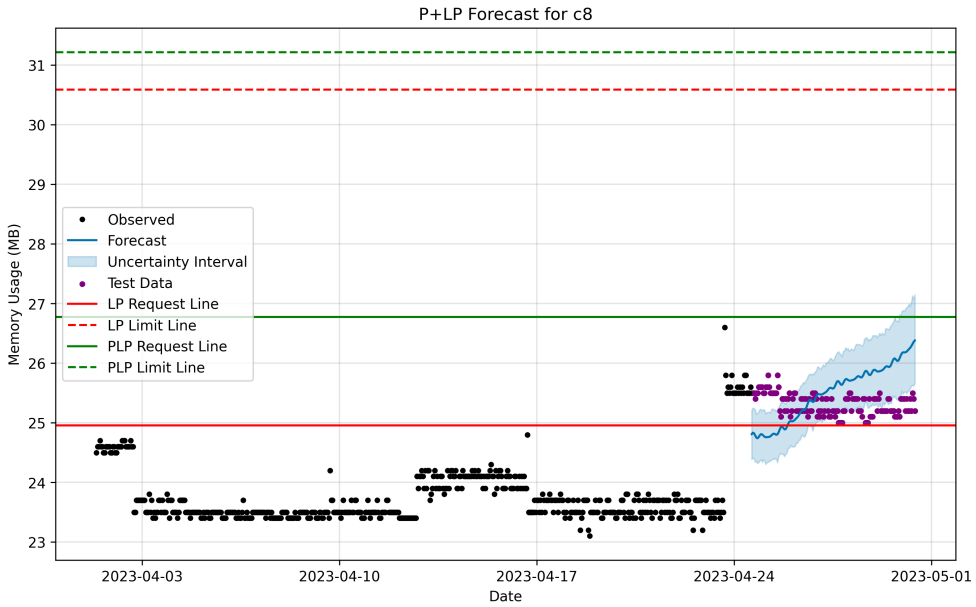

Figure 5.5: A comparison of suggested resource allocation between the LP and P+LP models for container c8. The P+LP model effectively addresses the under-allocation issue by setting a higher resource request based on Prophet's upward trend prediction. LP's request line is below all of the test data (solid red line), while P+LP's request is comfortably over c8's memory usage (solid green line)

We observed that only 8 out of 21 containers have set resource limits (see multiple na's in Table 5.2), meaning that 13 containers can consume as much memory as needed This can be detrimental for a cluster, especially if a container is already underallocated. First, resource usage may be underestimated due to the underallocation, and not having a cap could lead to disastrous consequences, such as consuming the memory of other containers that might require those resources during sudden peaks. It is crucial to ensure that all containers follow the same rules by setting requests and limits appropriately for this system to function effectively.

	Orig.	**LP**		**P+LP**	
	MREI	**MREI**	% **Diff. R/L**	**MREI**	% **Diff. R/L**
c1	100%	76.0%	-77.9/-74.5↑	100%↑	-76.9/-74.3
c2	100%	95.0%	-76.1/na	95.0%	-76.2/na↑
c3	100%	47.9%	-17.6/-68.6↑	100%↑	-13.2/-67.5
c4*	0%	100%	+69.9/-93.8↑	100%	+70.9/-82.8
c5	100%	87.8%↑	-46.5/-97.8	83.7%	-46.7/-97.8↑
c6	100%	93.6%	-93.7/na↑	93.6%	-92.7/na
c7	100%	100%	-98.1/na	100%	-98.1/na
c8	0%	0%	+66.4/na↑	100%↑	+78.5/na
c9	0%	100%	+80.0/na↑	100%	+80.2/na
c10	100%	100%	+0.1/na↑	100%	+0.2/na
c11	0%	97.9%	+54/na	97.9%	+54/na
c12	0%	95.9%	+68.9/na	95.9%	+68.2/na↑
c13	0%	100%	+19.2/na	100%	+19.2/na
c14	100%	100%	-58.1/na↑	100%	-57.6/na
c15	100%	91.4%	-80.6/na	91.4%	-80.6/na
c16	100%	100%	-2.5/na↑	100%	-2.2/na
c17	100%	100%	-6.0/na	100%	-6.1/na↑
c18	0%	100%	+105.1/-96.9	0%	+61.7/-96.9↑
c19	9.3%	82.9%	+2.6/-94.0↑	100%↑	+4.9/-94.0
c20	0%	100%	+10.2/-98.2↑	100%	+24.5/-98.2
c21*	100%	100%	-45.0/-94.0	100%	-45.0/-93.5
Average	48.1%	81.9%	**-43.0/-89.7**↑	**88.9%**↑	-39.3/-88.1

Table 5.2: Comparison of Original and New MREI for LP and P+LP. Each line represents the results of the models being run on the corresponding container. The table shows that P+LP is the model with the best MREI value while the LP algorithm has the highest reduction of memory request and limit. The upwards arrow display the best value for MREI and resource request between the LP and the P+LP

In general, the two algorithms are quite similar, with noticeable differences arising only from a few containers, as indicated by the upward arrow. Based on MREI, the P+LP model is the better choice, but it is also important to consider resource savings. In this regard, LP outperforms P+LP, achieving a 43% reduction in resource requests and a 77% reduction in limits, compared to P+LP's 39.3% and 75.6% reductions, respectively. Our results suggest that by using the P+LP model, we can significantly improve the MREI compared to the original resource R/L before applying any model, increasing it from 48.1% to 88.9%. This improvement should contribute to greater stability for Kubernetes nodes, as running containers

will have access to the required resources most of the time. Simultaneously, we can recommend a 39.3% reduction in allocated resources. While we do not delve into the specifics of how many megabytes are saved on each container, Table 5.2 shows that resource allocations are suggested to decrease for all containers, if we look away from the underallocated ones with a previous MREI close to 0.

5.2.1 Introducing a Performance Score

Even with the addition of Table 5.2, interpreting the results may be challenging, particularly when considering both performance indicators: MREI and the difference between resource requests. To evaluate these values collectively, we introduce a score variable that takes both MREI and the differences into account. This is achieved by calculating the absolute difference between the MREI and the ideal target of 95% +/- 4, which is then normalized to obtain the score factor for MREI. We have determined that a performance score of 100, corresponding to a score factor of 95, represents the highest score. This indicates that MREI is the dominant factor in performance evaluation, with a weighting split of 70/30 between the MREI and different components.

If the difference in resource requests is positive, indicating an increase in allocation, the difference will be disregarded, and the model will only use MREI as the target value. Finally, the buffer, which results from the +/-4 range, is accounted for; any value exceeding a difference of 4 from 95 will be penalized by calculating the MREI factor polynomially, while values within the buffer will decrease linearly. An equation with examples of how the performance score is calculated will be provided after having introduced Equation 5.1. The Python code is also provided at the end of the section in Code Listing 5.1.

In summary, this performance model prioritizes high MREI while still considering the difference in saved resources where applicable. The code can for this can be seen in Code Listing 5.1 and found in Appendix B.14.

In order to thoroughly assess the work completed, we calculate performance metrics across all containers to obtain what we consider the correct performance score. This step enables us to better understand the relationship between the MREI values and the difference in requests. In Table 5.3, we compare the performance scores of the original MREI from the containers prior to any modifications, the containers after applying the LP method, and finally, the containers after implementing the combined P+LP method.

Since the original values from the containers do not involve any increase or decrease in resource requests, our performance score function solely considers the MREI value. Consequently, an MREI of 0 results in a performance score of 0. This table does not include the original MREI values and differences, as they can be found in Table 5.2. We observe that P+LP achieves a performance score of nearly 70%, which we regard as a strong result, particularly in comparison to the original score of 35.6%.

Definition 4:

The performance score is determined based on the Memory Request Efficiency Index (MREI) and the reduction of allocated resources. However, any *increase* in resources is not taken into account when calculating the performance score. A MREI of 95% corresponds to a perfect performance score of 100.

Our findings are somewhat affected by the numerous containers that were initially underallocated, which we have chosen to address in this table, as described with values of *62.8* in multiple rows. If we had overlooked the underallocated containers and concentrated on reducing resource allocation for normally or overallocated containers, our results would have been considerably more favorable. Nevertheless, addressing underallocated containers is a crucial aspect of creating a sustainable environment for a Kubernetes node. By doing so, we ensure that containers are allocated appropriate resources without risking the depletion of resources by other containers due to a significant number of underallocated ones that use more than expected. Considering these factors, our combined algorithms still yield an impressive performance score, highlighting their effectiveness.

PERFORMANCE SCORE (Part 1)				PERFORMANCE SCORE (Part 2)			
Container	**Orig.**	**LP**	**P+LP**	**Container**	**Orig.**	**LP**	**P+LP**
c1	62.8	68.2	85.9↑	**c11**	0	67.9	67.9
c2	62.8	100	100	**c12**	0	69.3	69.3
c3	62.8	23.1	66.8↑	**c13**	0	62.8	62.8
c4*	0	62.8	62.8	**c14**	62.8	80.3↑	80.1
c5	62.8	73.7↑	68.3	**c15**	62.8	91.6↑	91.5
c6	62.8	97.1↑	96.8	**c16**	62.8	63.6↑	63.5
c7	62.8	92.3	92.3	**c17**	62.8	64.6	64.7↑
c8	0	0	62.8↑	**c18**	0	62.8↑	0
c9	0	62.8	62.8	**c19**	0.7	53.3	62.8↑
c10	62.8	62.8	62.8	**c20**	0	62.8	62.8
				c21*	62.8	76.3	76.3
Average Orig.: 35.6				**Average LP:** 64.0			**Average P+LP:** 69.7↑

Table 5.3: Comparison of performance scores across containers using original MREI values, LP method, and combined P+LP method. This table illustrates the improvement in performance scores after applying the LP and P+LP methods, demonstrating their effectiveness in creating a more sustainable Kubernetes environment by addressing both overallocated and underallocated containers

In general, the performance score utilizing P+LP for each container was satisfactory. However, an exception was observed for container c18, which was assigned a performance score of 0 as per Table 5.3. Contrastingly, when using LP alone, it achieved a score of 62.8. This phenomenon is illustrated effectively in Figure 5.6.

As depicted in the figure, Prophet forecasts a decrease in memory usage, while the observed and test data trend upwards. This erroneous prediction results in the P+LP request line (represented by the solid green line) falling below the entirety of the test data.

On the other hand, LP, which averages the train data, generates a request line that comfortably surpasses the test data, symbolized by the solid red line. However, given that the observed training data exceeds Prophet's uncertainty interval, our algorithm establishes the request limit (dotted lines) for both LP and P+LP at an identical height, significantly above the test data.

Therefore, despite a performance score of 0, the request limit is still capable of accommodating the increased usage for all containers upon which this P+LP model is implemented.

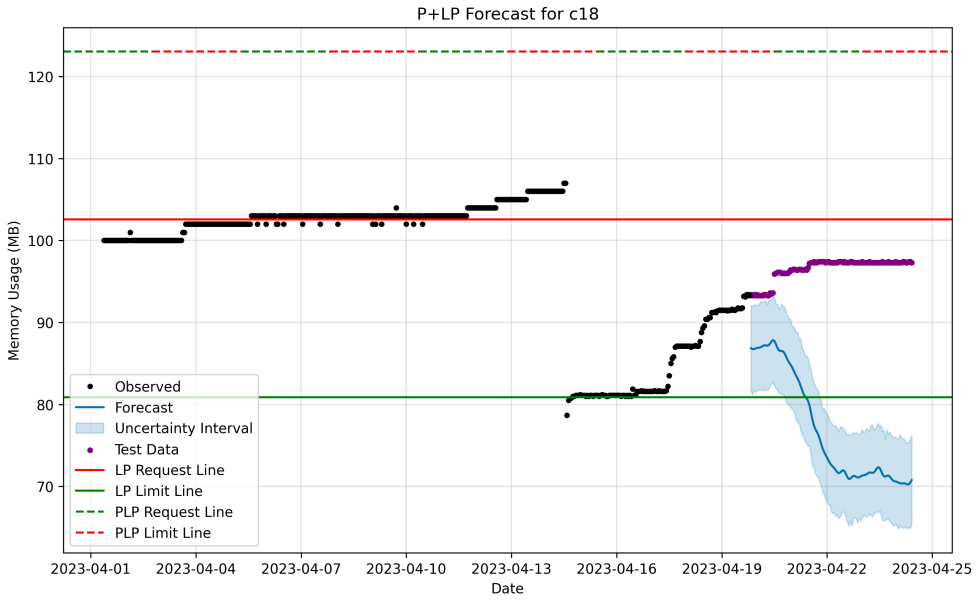

Figure 5.6: Visual representation of Prophet's downward prediction of memory usage versus the upward trend of actual data for c18, and the successful adaptation of the LP and P+LP request limit to handle increased memory usage despite initial performance score discrepancies for P+LP

The formula behind the performance score can be described using a set of equations, as seen in Equations (5.1) to (5.3). The first case in Equation 5.1 checks whether the MREI is equal to the ideal target and sets the performance score to 100. If not, the performance score is calculated using the sum of the *mrei_component* and the *diff_component*. The *mrei_component* is determined in Equation 5.2, where the absolute difference between the MREI and the ideal target is calculated. If this difference is within the buffer, the *mrei_component* is calculated linearly; otherwise, it is calculated polynomially, resulting in a lower value for that component. This value is then normalized and multiplied by the set *mrei_weight*. For the *diff_component* (see Equation 5.3), the value is set to 0 if the resource request difference is positive, and otherwise, the value is set to the absolute value of the difference multiplied by the set variable *diff_weight*.

By using this formula to calculate the performance score, a total of 6 variables are needed. In our case, 4 of the variables were pre-determined before calculating the performance score for all the containers on our running node. These variables are *ideal_target=95*, *mrei_weight=0.7*, *diff_weight=0.3*, and *buffer=4*. The ideal target and the buffer value were chosen early in Phase I based on what was consid-

ered a conservative middle ground for reducing resources while mainly focusing on the quality of service, given the diversity of the containers. The weighting was determined during Phase II, where we leaned more towards the idea of resource reduction being more present, but still keeping MREI as the most important factor. These values set on these variables *should* be challenged, and different values will provide different results. If the ideal MREI target is 100 with 0 buffer, one should set the variables accordingly. Another scenario is if resource reduction is more important, one could change the difference between the weighting of MREI and the difference in resources to 0.5/0.5. The last two variables are the *mrei* and the *resource_request_difference*. These values are obtained from calculating the MREI value of a container, which is done by examining the percentage of time the resource request is above the memory usage of the test data, and the latter variable is found by taking the original resource request and finding the percentage of difference for the new suggested resource allocation.

Consider container 14 with an MREI of 100% and a diff R of -57.6%. Since the MREI does not equal the ideal target, the second line of Equation 5.1 comes into play, requiring the calculation and summation of the MREI component and the diff component. To calculate the MREI component: given that an MREI of 100 is more than 4 units (the buffer) away from 95 (the ideal target), the second case in Equation 5.2is used with the following input: $(1-(\frac{|100-95|}{95}))^2 \times 0.7 \times 100 = 62.8$. Next, the diff component is calculated. In this case, since the reduction of memory is written as a negative percentage, the calculation proceeds as follows: $|\text{-}57.6| \times 0.3 = 17.3$. Otherwise, a positive value would result in the added value being 0, yielding a performance score of 62.8. Finally, referring back to the first equation, these values are added together, resulting in a Performance Score of $62.8 + 17.3 = 80.1$.

$$performance\_score = \begin{cases} 100, & \text{if} \quad mrei = ideal\_target \\ mrei\_component + diff\_component, & \text{otherwise} \end{cases} \tag{5.1}$$

$$mrei\_component = \begin{cases} \left(1 - \frac{|mrei - ideal\_target|}{ideal\_target}\right) \cdot mrei\_weight \cdot 100, \\ \text{if} \quad |mrei - ideal\_target| \leq buffer \\ \\ \left(1 - \frac{|mrei - ideal\_target|}{ideal\_target}\right)^2 \cdot mrei\_weight \cdot 100, \\ \text{otherwise} \end{cases} \tag{5.2}$$

$$diff\_component = \begin{cases} 0, \\ \text{if} \quad resource\_request\_difference > 0 \\ \\ |resource\_request\_difference| \cdot diff\_weight, \\ \text{otherwise} \end{cases} \tag{5.3}$$

We have not extensively discussed or evaluated the reduction of resource limits, which show over an 80% reduction using both LP and P+LP methods, and most likely also ARIMA, although we did not further evaluate this method. These results are important to consider but not as crucial as resource requests. Since we have already significantly reduced the resource limits, they could potentially be set even higher. Our considerable reduction still accounts for all peaks observed in the analyzed containers, but as we have reduced them so much, we could afford to increase them a bit to account for unexpected peaks that may appear surprising based on historical data and catch the uncertainty interval off guard.

However, discussing limits is still important as not having limits or having excessively high limits could potentially consume resources from other containers running concurrently, due to certain containers using many times their expected usage (the memory request). This can happen due to a configuration fault or a memory leak, which causes the container to continually increase its usage over time. This can become dangerous when extremely high limits are set, as seen when our reduction numbers are estimated to be more than 80%. Nevertheless, if setting overly high limits for multiple containers, it can be challenging to accommodate multiple peaks of resource usage taking advantage of those high limits and affecting the entire node in the process.

Code listing 5.1: Function that calculates the performance score Using the MREI value, resource request difference, and set ideal target, buffer and weights

```python
def calculate_score(mrei, resource_request_difference, ideal_target=95, buffer=4,
    mrei_weight=0.7, diff_weight=0.3):
    mrei_difference = abs(mrei - ideal_target)
    score_factor = abs(1 - (mrei_difference / ideal_target))

    if resource_request_difference > 0:
        resource_request_difference = 0

    if mrei == ideal_target:
        performance_score = 100
    elif mrei_difference <= buffer:
        mrei_component = score_factor * mrei_weight * 100
        diff_component = abs(resource_request_difference) * diff_weight
        performance_score = mrei_component + diff_component
    else:
        mrei_component = score_factor **2 * mrei_weight * 100
        print(mrei_component)
        diff_component = abs(resource_request_difference) * diff_weight

        performance_score = mrei_component + diff_component

    # Clip the performance score to be between 0 and 100
    performance_score = max(0, min(100, performance_score))

    return performance_score
```

5.3 Summary: Addressing Research Question 2

During Phase II, we effectively address our second research question: "*How can we effectively evaluate the impact of predictive models on resource allocation strategies in Kubernetes environments?*". First, we utilize the KPI created in Phase I (MREI), which offers insight into the quality of the suggested resource request and limit values by applying them on the test data for each container. This evaluation helps us understand how well the allocation would perform. Furthermore, we calculate the differences in requested resources, indicating a decrease or an increase if a container was underallocated with a suggested memory request below the memory usage for the majority of the time.

Ultimately, we create a performance score that takes into account both the MREI value and the difference in resource requests. By comparing this score to the original values before running any model, we can clearly see the impact that the predictive models, in tandem with our LP algorithm, have on this arbitrary node. Our solution enables us to decrease allocated resources by 39.3% and also increases the amount of time the resource allocation covers the actual usage by 60% which is an exceedingly good result. The combination of the reduction of resources along with the MREI gives us a performance score of 69.7 compared with the original score of 35.6. This comprehensive evaluation approach enables us to effectively assess the influence of predictive models and the LP algorithm on resource allocation strategies within Kubernetes environments.

Chapter 6

Discussion

In this section, we aim to discuss the challenges and potential solutions for resource allocation and prediction. We will focus on aspects such as model selection, data normalization, and the implications of using raw data in our algorithms. We will also discuss the challenges we faced during the project and propose ideal workflows to make use of our findings.

In the approach chapter, we proposed a two-phase strategy, with Phase II essentially being an expanded version of Phase I. Everything in Phase I was carried out with this expansion in mind. Investing time in creating reusable scripts that generate plots with horizontal lines, prediction lines, labels, and legends proved to be highly beneficial in facilitating the transition to a larger scale.

During the project, we realized that focusing on both CPU and memory usage was too much, so we chose to concentrate on multiple models for memory usage instead. Incorporating a model for CPU usage would have added complexities when using multiple models, particularly in phase II. This decision is not seen as a disadvantage; rather, we are shifting the task of applying various algorithms to container CPU usage to a separate project. In the current project, memory is regarded as the most important and easily accessible metric, as applications terminate instead of merely slowing down when they reach their limits. We acknowledge that a slowed-down container could still be critical, but it falls outside our scope, especially when considering CPU throttling as part of the analysis.

6.1 Evaluating The Predictive Models

Before considering different models we specified the environment to investigate. Whether to focus on understanding the resource usage of a specific set of applications or a namespace, or considering the resource usage of the underlying infrastructure, such as hardware resources like CPU and memory utilization. In our case, this would translate to namespace level or node level. Both namespace and node-level modeling can provide valuable insights and the choice of level for the model would depend on the goals and objectives of the analysis. If performed in conjunction with each other it would give a more complete picture of the resource usage across the cluster. However, we chose to investigate the resource usage on the node level to mainly focus on the resource usage of containers from multiple namespaces that run on a single node.

The statistical models employed in our study offered moderate predictions on the memory usage of the containers. We observed four distinct usage patterns among the containers: steady, trending, seasonal, and bursty behavior. The considerable variation in usage patterns makes it challenging to develop a one-size-fits-all model, particularly for random bursty behavior. Both ARIMA and Prophet models performed well in predicting steady, trending, and seasonal usage, as evidenced by the favorable TDUR and MAPE values. Both models are employed with minimal modifications from their default settings. Enhancing these models, particularly Prophet, which offers numerous adjustable parameters, could provide significant benefits. Additionally, clustering containers based on similar workloads and then applying the most suitable configurations for each cluster could greatly improve the models. This could lead to more accurate predictions and better suggestions for resource allocation.

To address more erratic usage patterns, we focused on utilizing uncertainty intervals and safety margins to account for such containers while maintaining resource requests and limits lower than the original values. This approach allowed us to keep the model relatively simple, as well as the algorithm running on top. The simplicity of concepts such as a 10% safety margin based on the highest observed value makes it easy for users to comprehend, use, and improve upon the model Adjustments can then be made easily, and their effects can be observed and assessed accordingly.

During our project, the safety margins we set were relatively high, yet we still achieved an average reduction of around 70% in resource usage with the P+LP model when excluding the originally underallocated containers, for which we increased resource requests. While it is possible to further optimize and reduce resource usage, pushing the numbers down by a few additional percentages would not significantly enhance the values already discovered in this project. Moreover, it would increase the risk of underallocating resources, which is undesirable for containers operating in a production environment.

In situations where high availability is not a priority, adjusting risk settings could potentially lead to further resource reduction. Future work could explore optimizing and fine-tuning resource allocation by building upon existing knowledge and performance indicators. We identified a target MREI of 95% as ideal, and directing the models and resource suggestions toward that target might yield favorable results. Adjusting the ideal target could also be considered based on the user's familiarity with container usage or the prevalence of erratic containers.

Before adopting a more conservative approach to resource requests, engineers responsible for creating containers should prioritize launching containers with fitting resource allocations.

The evaluation table for the P+LP model in Phase II illustrates that resource limits are often set quite high, indicating a lack of alignment with actual container workloads in comparison to resource requests. This pattern suggests that developers tend to ensure their containers have sufficient resources by setting high limits, while resource requests can be either too low or excessive. However, when this scenario occurs on a larger scale, the cluster may face issues such as availability and significant overprovisioning due to numerous containers having excessively high limits. Consequently, the node's available resources might be exceeded if many containers operate beyond their requested resources and closer to their set limits.

This situation likely explains why Intility maintains a considerable safety margin for virtual nodes, which ranges between 12% and 40% across different nodes. By safety margin, we mean that only 12% to 40% of the node's resources have been requested. The infrastructure is presumably aware that some resources might be underallocated, while limits are overallocated, making it difficult for administrators to accurately predict resource allocation at any given moment.

By implementing our solution, we can achieve a significantly higher request

percentage, close to 90%, while still being confident in handling peaks and maintaining a clear overview of the node. This is due to well-established resource requests and limits that accommodate various peak scenarios.

6.1.1 Moving Forward with the KPIs

During the project, multiple KPIs are introduced. While some are well-known and typically used to evaluate the performance of a predictive model, such as MAE, MAPE, and RMSE, we introduce MREI and TDUR, as well as the Performance Score, which considers the reduction of suggested allocated resources. By going a bit further than what was done in this study, MREI can be implemented with percentage margins, for example, being 5% within the margin of resource usage could still considered as data points with the MREI target, or one could even define 5% below resource usage and 20% above resource usage. This approach allows for strict under-allocation policies and targets excessive over-allocation as well. These margins can be further adjusted depending on the desired outcomes. In our case, we kept it simple as we introduced this concept for the first time and proceeded with the MREI-based choices for the Phase II, before making things too complicated. The same applies to the prediction of the models, where we use TDUR, which is the percentage of time the prediction is below the test data. We chose to handle both MREI and TDUR in a similar fashion for simplicity. Going forward, one should test different strategies and margins for MREI and TDUR.

To assess the total impact of our framework on a Kubernetes node, we consider the MREI of the suggested memory allocation along with the change in resource allocation made to achieve this value, and call it the Performance Score. In this process, we decide upon several variables that should be adjusted according to what is deemed important, whether it's quality of service, easy management, or resource reduction. These variables include the ideal target for memory request efficiency, a buffer, and the weighting of MREI and resource reduction. Adjusting these variables can be valuable when dealing with clustered containers based on similar workloads, namespaces, or different environments (development, testing, or production).

It is pertinent to mention that despite meticulous calculations and sometimes multiple iterations to confirm the accuracy of all evaluation methods, including KPIs and performance scores, there could still be minor errors in the numbers

provided during the report. These may pertain to specific values or the summation of columns from tables. This is due to the fact that I solely conducted these evaluations, without any cross-checking or validation from other individuals.

6.2 The Impact and Limitation of the Datasets

Given that containers are designed to operate for a limited duration before being updated or redeployed for various reasons, we anticipated encountering a few containers with datasets spanning several weeks. In addition to containers running for a couple of weeks or days, we saw that many containers were activated to perform a specific task and then shut down, which is one of the numerous advantages of utilizing containers. During phase II, we only investigated and examined containers on node01 that contained over a week's worth of data.

We made a decision to only include containers running between April 1st and April 30th. By doing so, time intervals of 5-minuted, which were used in Phase I, became 60-minute intervals for the most part. This was because containers with resource usage spanning more than 21 days got aggregated, while some containers started in the early-middle of April they still got aggregated since the collection took place early May which introduced the same issue. To address this, one could be more flexible with the time period by using the current day of collection and looking back 4 weeks instead of a specific one-month period (1st to 30th). This would still result in many containers using 60-minute intervals due to their potential 21+ days runtime, but more containers with 5-minute intervals would be included.

However, we do not evaluate the impact of the time interval on our prediction models, either conceptually or mathematically, although it likely has an effect. It is not considered a significant issue since we use the maximum observed value, a value that would not change whether we used 1-minute or 1-day intervals, for setting safety margins. With more homogeneous resource usage, one could consider daily or even weekly data for analysis, which might be a safer option for considering peaks. However, not many containers run for multiple weeks without changes being made or restarts occurring, so these kind of dataset might be limited.

It is worth noting the persistence of containers after they have been reconfigured and assigned a new ID. By tracking the new IDs of a single container,

one could piece together data from several weeks or even months. We initially intended to do this but found that scrutinizing the modifications proved too time-consuming. In instances where changes were as simple as a name alteration, it could be justifiable to track the new container as well. However, when updates involve docker image changes, storage adjustments, network modifications, or other configuration file alterations, it may not be appropriate or may become overly complex to model the combined resource usage.

In numerous projects, normalizing the data to minimize variations or employing a moving average to mitigate highs and lows is a sensible approach. We could have applied these methods to our data, which would have likely led to improved model performance. However, the drawback of this approach is that while the models may seem to perform better, the container still consumes the same amount of resources. Consequently, suggesting request and limit adjustments would be futile, as they would be based on normalized or smoothed data. Although some calculations and analyses can still be conducted on normalized data before reverting to the original values, we did not find it useful in our case. A worst-case scenario would involve a peer, student, or developer at Intility basing themselves on findings from the normalized values to request resources, which might be significantly below the actual usage peaks. By not using normalized data throughout the project we also preserve the research quality and ensure that recommendations are based on accurate and representative information.

A crucial element in our project was to avoid under-allocation in order to maintain uptime and stability for all containers. As a result, we opted to import the maximum values of resource usage rather than the average value over the 5-minute intervals. While utilizing the average of 5-minute intervals in contrary to maximum values, might simplify resource usage prediction for the model, it would be inappropriate to do so, as this approach could potentially truncate peaks, leading to fatal consequences for the containers.

6.2.1 Training/Prediction of Datasets

In many cases, the use of an 80/20 training/test split may not have been ideal for predicting resource usage in containers. Adjusting the training or test sections could potentially yield better results for specific containers. For instance, if we were to use a 50/50 split for container c1, the prediction would closely resemble a

straight line that matches the resource usage well, up until the sudden peaks which would diverge significantly from the prediction. Early in the project, we committed to the 80/20 split rather than exploring alternative approaches or adapting the split for individual containers based on their workloads.

A more flexible approach might have involved using different data splits or even employing adaptive algorithms to determine the optimal split for each container. This could have allowed for more accurate predictions that better account for the variations in resource usage patterns among different containers. In future projects, it may be beneficial to invest more time in identifying the most appropriate data splits, taking into consideration the unique characteristics of each container's workload.

6.3 The Challenges of Predicting Containers

Initially, this project was designed as an exploratory endeavor, and it has successfully evolved in that matter over time. Our goal was to gain a deeper understanding of container resource usage, and we can now effectively describe the data, particularly with regard to memory usage. The ability to comprehend and describe resource usage was always going to be a part of our conclusion as well. We employed well-known and established algorithms, which require data in a specific format. Typically, ARIMA would not anticipate data to begin at an unusually high point, sometimes tens of times greater than the average, and then suddenly drop to zero at another point. We suspect that fluctuations and abrupt peaks negatively impacted the performance, suggesting that our data may not be suitable for such algorithms unless it is exported or obtained differently.

If we had chosen to further examine CPU usage, the correlation between memory and CPU, and investigate daily patterns for all containers in Phase II, the results might have been different, but probably not significantly so. Workload profiling was initially considered as part of Phase II but was eventually omitted. While conducting all these analyses would have been ideal, the scope of the project became too extensive. Nonetheless, our project stands well on its own without these additional analyses. However, it would certainly be interesting to explore these aspects in future work

Kubernetes and similar container management systems like Docker Swarm offer considerable flexibility and features, such as automated rollouts and CI/CD. From a high-level perspective, everything generally runs smoothly and efficiently. However, upon examining container usage, what might be considered anomalies, with regard to resource usage, in other machines or servers are actually normal for these containers. The substantial startup peaks experienced by some containers, the sudden 5x spikes, and random drops to zero usage are all typical, as the environment is designed to manage these situations through quick restarts, launching new containers on the same or another node within seconds, and utilizing replicas to balance the load. While these systems function effectively, the inherently unpredictable nature of each individual container makes developing a predictive model on this level quite challenging.

6.4 Suggested Workflow for the Framework

So far, we have developed a script that, when provided with a dataset from a container, suggests a new resource request and limit for the container. It also calculates the MREI, difference in resources, and performance score before and after the adjustment. The next step involves manually editing the .yaml deployment file with the new values, which restarts the container. This process can be time-consuming, taking 5-10 minutes depending on the developer's familiarity with the platforms.

To streamline the process, the next step is to create a semi-automatic application where the developer or system admin can approve or decline changes. The application can run at specific intervals, suggest adjustments, and log the actions taken by the human. This provides a controlled approach with human input and a paper trail. The application could run as a service on the node it evaluates, allowing for node-specific variables, making it possible to use different allocation strategies and policies for different nodes. The application would need access to the Dynatrace API for resource usage, the Prometheus API for current request and limit, our Python script, write access to all containers on that node.

Determining how often the script should run depends on the desired balance between frequent adjustments and avoiding unnecessary changes. If certain thresholds are reached, the system could alert a team's communication channel, prompting the developer to review the suggestions. Based on our approach, we

predict 20% of the total dataset ahead, which equates to a maximum of 6 days for a 30-day dataset. Since we have 6 days as testing period, a weekly evaluation of certain nodes could be conducted to review the suggested resources and approve or decline significant changes. This should be tested on a non-production node to evaluate the suggestions and weekly performance.

For testing environments, variables can lean more towards green IT as container restarts are generally not a major concern. A monthly review could be introduced where all new suggested allocations are considered, and even minor changes are applied. This serves as a monthly cleanup where historical adjustments are evaluated in collaboration with the deployment owner.

6.4.1 The Lack of Incentives for Reducing Resources

In the early chapters of this study, it was discussed that there is a lack of incentive for reducing resources. Developers are primarily focused on ensuring that applications or services are up and running with high availability. To encourage a shift in mindset, resource allocation should be introduced as a company policy, building on Intility's existing promotion of green computing.

The change could start at an earlier stage, such as during university education, coding courses, or certifications, where resource allocation could be a focal topic. There is currently a lack of collaboration between Kubernetes and sustainability efforts, and it would be beneficial for Kubernetes to promote sustainable practices to a greater extent. Gradually, this will encourage developers to consider resource allocation, even if it requires a small portion of their development time.

Implementing this mindset will take time, but it is essential to start thinking about programming and using Kubernetes more sustainably. In the long run, it could save companies significant resources, especially if processes like these can be automated.

In the midst of the project (mid February), Intility used an admission controller to enforce all namespaces with the exception of certain system crucial applications to specify CPU and memory requests and memory limit, before this a container could be spun up without specifying any resoruces at all. The goal of this is to get a better overview of the resource bindings of the underlying nodes. The change made this project even more relevant since now every developer has to, in some way, set the resource usage of their deployments and pods. It will also be interesting to follow the resource request in the months to come, now that developers

are starting to be a bit more conscious about the requests to be made.

Chapter 7

Conclusion

The objective of this project was to investigate the application of time series forecasting models for predicting individual container resource usage in a Kubernetes environment, as well as to identify the challenges and benefits of employing these models to ensure accurate predictions across diverse usage patterns. Throughout the project, two models, ARIMA and Prophet, were applied to predict resource usage for various containers, and challenges and advantages associated with each model were discussed.

Resource usage data was gathered from Dynatrace and resource requests from Prometheus while preprocessing and processing were carried out using Python prior to implementing the ARIMA and Prophet models. In addition, an algorithm (LP) was developed for suggesting resource allocation based on average usage and safety intervals, which was subsequently combined with the Prophet prediction model. A key performance indicator (KPI) was devised to evaluate the performance of the LP algorithm, measuring the percentage of time the suggested resource request surpassed the actual memory usage (MREI).

Higher-level memory usage in containers that would be more suitable for the algorithms was anticipated. However, it was observed that in most cases, predicting erratic and highly fluctuating memory usage was necessary. The algorithms worked exceptionally well for some containers, while for the more erratic ones, the uncertainty interval from the Prophet model was utilized in combination with safety margins. The predictive models were evaluated similarly to the LP algorithm's KPI, with the TDUR score indicating how often the predicted value was above memory usage.

Additionally, the aim of the project was to effectively evaluate the impact of

predictive models on resource allocation strategies in a Kubernetes environment. Although the models demonstrated average performance across containers, combining the predictions with the algorithm from Phase I (LP) and safety margins resulted in a successful solution for suggesting resource requests and limits. This allowed for accommodating unforeseen peaks while cutting down on allocated resources.

By using the suggested framework, Kubernetes environments without strict policies or gatekeeping for setting resource requests and limits will see significant reductions in allocated resources, leading to a greener environment and reduced electricity costs. The framework can still yield positive results in more conservative environments as well by adjusting variables such as ideal targets, buffers, safety margins, and weighting. The P+LP is scripted in a way that it can be fine-tuned and tested to fit various allocation strategies.

The green impact of the framework is substantial, as it suggests managing containers in a sustainable manner by maintaining control over resource usage and requests for all containers. The framework involves data collection from multiple containers, preprocessing, processing, and running the model with the LP algorithm to recommend resource requests and limits for all containers. The framework, when applied to node01, resulted in an average reduction of 39.3% in allocated resources and a 60% increase in memory usage coverage. An equation that linked the reduction of resource allocation and the MREI value before and after applying the framework to the node showed a 95.8% improvement, highlighting the effectiveness of this approach.

7.1 Future Work

Due to the exploratory nature of this project, it is not possible to investigate all new paths that emerge within the researched field, especially considering the constraints of a short thesis. Given more time, we would have liked to explore additional aspects, such as examining more nodes in the cluster, evaluating different algorithms and fine-tuning the existing ones, and further analyzing CPU usage. Using the framework and the same variables for multiple arbitrary nodes in Kubernetes environments would give great insight to how well the framework performs.

By investigating more nodes in the cluster, including dedicated nodes with

carefully assigned requests and limits, we could obtain more data to strengthen the correlation between resource usage across various workload profiles throughout multiple nodes. Although prediction algorithms struggle to analyze container usage, we believe that fine-tuning these algorithms and using request limits as a safety net could be a promising approach when applying other algorithms and especially during their fine-tuning.

Regarding CPU usage, in some cases, it might be beneficial to modify the configuration of a running service to allow for more threads, enabling the service to utilize available resources more efficiently rather than merely reducing overallocated resources. Exploring these scenarios and their applicability would be an interesting topic for a follow-up project or as an extension of this study.

Additionally, one could investigate the possibility of rescheduling pods between nodes to optimize resource utilization tied to requested resources. However, this approach would need to be compatible with the existing Kubernetes scheduler.

Bibliography

[1] European Commission, Directorate-General for Communications Networks, Content and Technology, F. Montevecchi, T. Stickler, R. Hintemann, and S. Hinterholzer, *Energy-efficient cloud computing technologies and policies for an eco-friendly cloud market : final study report*. Publications Office, 2020. DOI: doi/10.2759/3320.

[2] Zhang, Qi and Cheng, Lu and Boutaba, Raouf, "Cloud computing: State-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010, ISSN: 1869-0238. DOI: 10.1007/s13174-010-0007-6. [Online]. Available: https://doi.org/10.1007/s13174-010-0007-6.

[3] Alibaba Cloud. "Transforming businesses and shaping innovation." (2023), [Online]. Available: https://eu.alibabacloud.com/en?utm_key=se_1012200419%5C&utm_content=se_1012200419. (accessed: 07.03.2023).

[4] Google Cloud. "Why google cloud." (2023), [Online]. Available: https://cloud.google.com/why-google-cloud. (accessed: 07.03.2023).

[5] D. Weiss (neuroflash). "Historic slogans and claims of it and computer brands." (2021), [Online]. Available: https://neuroflash.com/blog/slogans-claims-of-it-computer-brands/. (accessed: 08.03.2023).

[6] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011, pp. 3–10, ISBN: 9780470887998.

[7] Ryan Kap et al. (desosa 2021). "Kubernetes and sustainability." (2023), [Online]. Available: https://2021.desosa.nl/projects/kubernetes/posts/2021-03-29-kubernetes-sustainability-analysis/. (accessed: 14.03.2023).

[8] Federal Trade Comission. “Equifax data breach settlement.” (2022), [Online]. Available: https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement. (accessed: 22.03.2023).

[9] Michael X. H. (Firewall Times). “Amazon web services (aws) data breaches: Full timeline through 2022.” (2022), [Online]. Available: https://firewalltimes.com/amazon-web-services-data-breach-timeline/. (accessed: 22.03.2023).

[10] Y. Sverdlik. “Aws outage that broke the internet caused by mistyped command.” (2017), [Online]. Available: https://www.datacenterknowledge.com/archives/2017/03/02/aws-outage-that-broke-the-internet-caused-by-mistyped-command. (accessed: 22.03.2023).

[11] Office for National Statistics (UK), Labour Market. “Coronavirus and homeworking in the uk: April 2020.” (2020), [Online]. Available: https://www.ons.gov.uk/employmentandlabourmarket/peopleinwork/employmentandemployeetypes/bulletins/coronavirusandhomeworkingintheuk/april2020. (accessed: 11.01.2023).

[12] Fareeha Ali, Digital Commerce 360. “Early estimates: Us ecommerce grows 44.0% in 2020.” (2021), [Online]. Available: https://www.digitalcommerce360.com/2021/01/29/early-estimates-us-ecommerce-grows-44-0-in-2020/. (accessed: 01.02.2023).

[13] Flexera. “State of the cloud report 2022.” (2022), [Online]. Available: https://info.flexera.com/CM-REPORT-State-of-the-Cloud. (accessed: 17.01.2023).

[14] StormForge. “Stormforge 2022 kubernetes & cloud waste survey.” (2022), [Online]. Available: https://www.stormforge.io/survey-report/stormforge-2022-kubernetes-cloud-waste-survey-thank-you/. (accessed: 28.02.2023).

[15] Javier Martinez. “Millions wasted on kubernetes resources.” (2023), [Online]. Available: https://sysdig.com/blog/millions-wasted-kubernetes/. (accessed: 24.01.2023).

[16] I. Sarji, C. Ghali, A. Chehab, and A. Kayssi, “Cloudese: Energy efficiency model for cloud computing environments,” *Energy Aware Computing (ICEAC), 2011 International Conference on*, 2011. DOI: 10.1109/ICEAC.2011.6136680.

[17] Kubernetes. “Overview.” (2023), [Online]. Available: https://kubernetes.io/docs/concepts/overview/. (accessed: 17.03.2023).

[18] S. H. Perveez. “Understanding kubernetes architecture and its use cases.” (2023), [Online]. Available: https://www.simplilearn.com/tutorials/kubernetes-tutorial/kubernetes-architecture. (accessed: 17.03.2023).

[19] E. Khun. “Kubernetes: Make your services faster by removing cpu limits.” (2020), [Online]. Available: https://erickhun.com/posts/kubernetes-faster-services-no-cpu-limits/. (accessed: 18.03.2023).

[20] Dina Henderson (Turbonomic.com/IBM community). “Kubernetes cpu throttling: The silent killer of response time – and what to do about it.” (2022), [Online]. Available: https://community.ibm.com/community/user/aiops/blogs/dina-henderson/2022/06/29/kubernetes-cpu-throttling-the-silent-killer-of-res. (accessed: 24.02.2023).

[21] N. Yellin. “For the love of god, stop using cpu limits on kubernetes.” (2022), [Online]. Available: https://home.robusta.dev/blog/stop-using-cpu-limits. (accessed: 18.03.2023).

[22] D. Chiluk. “Unthrottled: Fixing cpu limits in the cloud.” (2019), [Online]. Available: https://engineering.indeedblog.com/blog/2019/12/unthrottled-fixing-cpu-limits-in-the-cloud/. (accessed: 25.03.2023).

[23] X. Pang. “Sched/fair: Fix bandwidth timer clock drift condition.” (2018), [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=512ac999. (accessed: 26.03.2023).

[24] European Commission. “A european green deal.” (2021), [Online]. Available: https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/european-green-deal_en. (accessed: 11.01.2023).

[25] V. Avelar, D. Azevedo, A. French, “Pue™: A comprehensive examination of the metric,” Report, 2012. [Online]. Available: https://www.thegreengrid.org/en/resources/library-and-tools/237-PUE%5C%3A-A-Comprehensive-Examination-of-the-Metric, (accessed: 06.02.2023).

[26] Intility. “Sustainable platform: Report on sustainability for 2021.” (2021), [Online]. Available: https://intility.no/wp-content/uploads/2021/12/Intility_Sustainable-platform-2021.pdf. (accessed: 12.01.2023).

[27] statista. “What is the average annual power usage effectiveness (pue) for your largest data center?” (), [Online]. Available: https://www.statista.com/statistics/1229367/data-center-average-annual-pue-worldwide/. (accessed: 28.02.2023).

[28] Yevgeniy Sverdlik (DataCenter Knowledge). "Study: Data centers responsible for 1 percent of all electricity consumed worldwide." (2020), [Online]. Available: https://www.datacenterknowledge.com/energy/study-data-centers-responsible-1-percent-all-electricity-consumed-worldwide. (accessed: 14.03.2023).

[29] Odyssee Mure. "Sectoral profile - households." (2021), [Online]. Available: https://www.odyssee-mure.eu/publications/efficiency-by-sector/households/electricity-consumption-dwelling.html. (accessed: 01.02.2023).

[30] Cloudscene. "Datacenters in europe." (2023), [Online]. Available: https://cloudscene.com/region/datacenters-in-europe. (accessed: 08.03.2023).

[31] Data Center Map. "Western/eastern/middle europe." (2023), [Online]. Available: https://www.datacentermap.com/western-europe/. (accessed: 08.03.2023).

[32] Statista. "Number of data centers worldwide in 2022, by country." (2022), [Online]. Available: https://www.statista.com/statistics/1228433/data-centers-worldwide-by-country/. (accessed: 08.03.2023).

[33] Google. "Efficiency." (2023), [Online]. Available: https://www.google.com/about/datacenters/efficiency/. (accessed: 11.03.2023).

[34] N.-M. Dang-Quang and M. Yoo, "Deep learning-based autoscaling using bidirectional long short-term memory for kubernetes," *Applied Sciences*, vol. 11, no. 9, p. 3835, 2021, ISSN: 2076-3417. [Online]. Available: https://www.mdpi.com/2076-3417/11/9/3835.

[35] J. Kumar, A. K. Singh, and R. Buyya, "Self directed learning based workload forecasting model for cloud resource management," *Information Sciences*, vol. 543, pp. 345–366, 2021, ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2020.07.012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025520306782.

[36] A. Hatamlou, "Black hole: A new heuristic optimization approach for data clustering," *Information Sciences*, vol. 222, pp. 175–184, 2013, ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2012.08.023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025512005762.

[37] B. Letham, *Prophet: Automatic forecasting procedure*, https://github.com/facebook/prophet, 2017.

[38] T. Toharudin, R. S. Pontoh, R. E. Caraka, S. Zahroh, Y. Lee, and R. C. Chen, "Employing long short-term memory and facebook prophet model in air temperature forecasting," *Communications in Statistics - Simulation and Computation*, vol. 52, no. 2, pp. 279–290, 2023, doi: 10.1080/03610918.2020.1854302, ISSN: 0361-0918. DOI: 10.1080/03610918.2020.1854302. [Online]. Available: https://doi.org/10.1080/03610918.2020.1854302.

[39] M. Khayyat, K. Laabidi, N. Almalki, and M. Al-zahrani, "Time series facebook prophet model and python for covid-19 outbreak prediction," *Computers, Materials & Continua*, vol. 67, no. 3, 2021, ISSN: 1546-2226. DOI: 10.32604/cmc.2021.014918.

[40] W.-X. Fang, P.-C. Lan, W.-R. Lin, H.-C. Chang, H.-Y. Chang, and Y.-H. Wang, "Combine facebook prophet and lstm with bpnn forecasting financial markets: The morgan taiwan index," in *2019 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, IEEE, pp. 1–2, ISBN: 1728130387.

[41] M. Daraghmeh, A. Agarwal, R. Manzano, and M. Zaman, "Time series forecasting using facebook prophet for cloud resource management," in *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6, ISBN: 2694-2941. DOI: 10.1109/ICCWorkshops50388.2021.9473607.

[42] Facebook Opensource). "Prophet: Forecasting at scale." (2023), [Online]. Available: https://facebook.github.io/prophet/. (accessed: 22.03.2023).

[43] R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. Terpenning, "Stl: A seasonal-trend decomposition," *J. Off. Stat*, vol. 6, no. 1, pp. 3–73, 1990.

[44] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: Theory and applications," *Neurocomputing*, vol. 70, no. 1-3, pp. 489–501, 2006, ISSN: 0925-2312.

[45] T. Xiong, C. Li, and Y. Bao, "Seasonal forecasting of agricultural commodity price using a hybrid stl and elm method: Evidence from the vegetable market in china," *Neurocomputing*, vol. 275, pp. 2831–2844, 2018, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2017.11.053. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092523121731812X.

[46] PuLP. “A set partitioning problem.” (2009), [Online]. Available: https://coin-or.github.io/pulp/CaseStudies/a_set_partitioning_problem.html. (accessed: 1.04.2023).

[47] Param Raval. “A set partitioning problem.” (2023), [Online]. Available: https://www.projectpro.io/article/how-to-build-arima-model-in-python/544. (accessed: 5.04.2023).

[48] Prashant Banerjee. “Tutorial: Time series forecasting with prophet.” (2021), [Online]. Available: https://www.kaggle.com/code/prashant111/tutorial-time-series-forecasting-with-prophet. (accessed: 5.04.2023).

Appendix A

Survey Sent to Developers at Intility

Appendix B consists of all questions and answers from the survey sent out to developers at Intility. Below is the description of the survey which was sent out together with the survey

Hello, my name is Håkon Borgersen Ay. I work part-time at the developer infrastructure team and will be starting full-time as a trainee in August 2023. Right now I'm writing my master thesis with the working title: "Optimizing Resource Utilization in a Kubernetes Cluster: Investigating and Automating Resource Allocation using Workload Profiles".

I am conducting an anonymous survey to gather information about resource utilization and cloud waste associated with cloud resources. Your input will help me identify areas of improvement and potential cost savings.

The term Cloud is in this case used to describe Kubernetes clusters. Consider the resource usage vs. the allocated resources of clusters, nodes, and pods when answering. When the word resources are used I'm referring to the allocated CPU and RAM.

Please note that while some of the data collected through this survey may be used in my master thesis, certain information may be kept confidential due to policy constraints. I appreciate your participation.

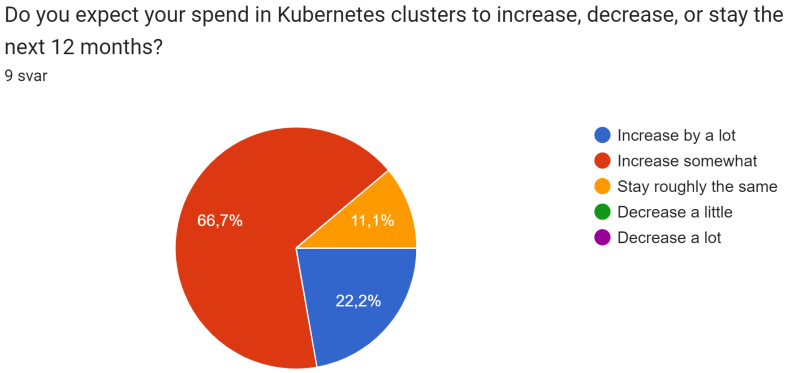

Figure A.1: Survey Sent to Developers at Intility: Question 1

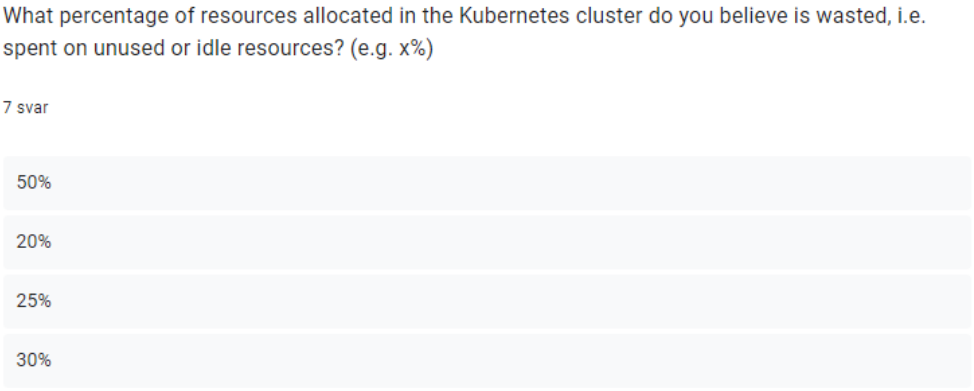

Figure A.2: Survey Sent to Developers at Intility: Question 2

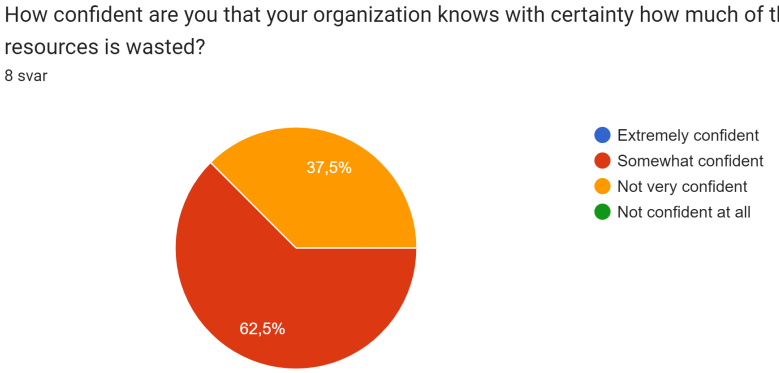

Figure A.3: Survey Sent to Developers at Intility: Question 3

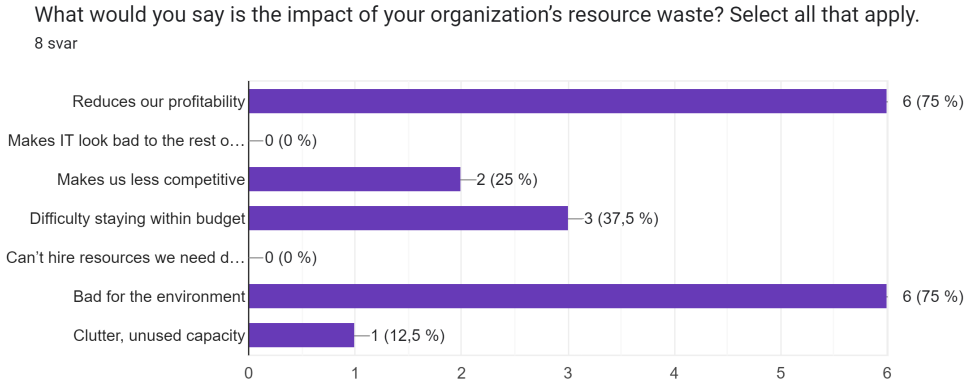

Figure A.4: Survey Sent to Developers at Intility: Question 4

Answers from Figure A.4

Reduces our profitability

Makes IT look bad to the rest of the organization

Makes us less competitive

Difficulty staying within budget

Can't hire resources we need due to budget constraints

Bad for the environment

Clutter, unused capacity

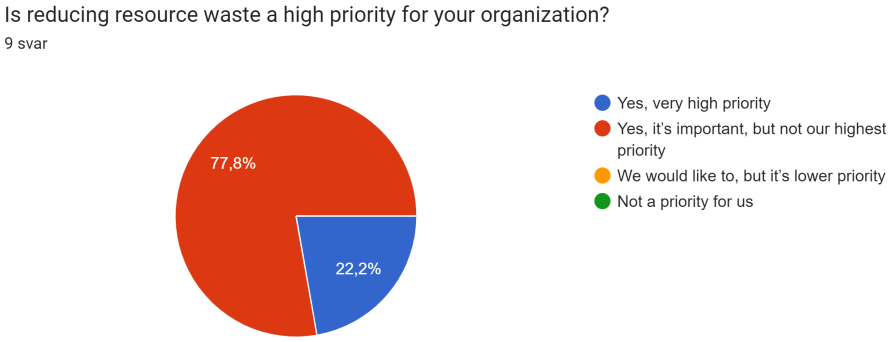

Figure A.5: Survey Sent to Developers at Intility: Question 5

Figure A.6: Survey Sent to Developers at Intility: Question 6

Answers from figure A.6:

Cloud complexity makes it hard to estimate how many resources are actually needed Over-provisioning to ensure applications perform well

Maintaining more environments than needed

Developers have no incentive to run apps efficiently

Not enough attention or management oversight of cloud spend

No visibility into actual resource utilization vs. what we're paying for

Controlling cloud spend is not a priority for us

Ensuring we are ready for a potential surge in demand

Applications are configured for high availability, causing us to run more resources than actually needed

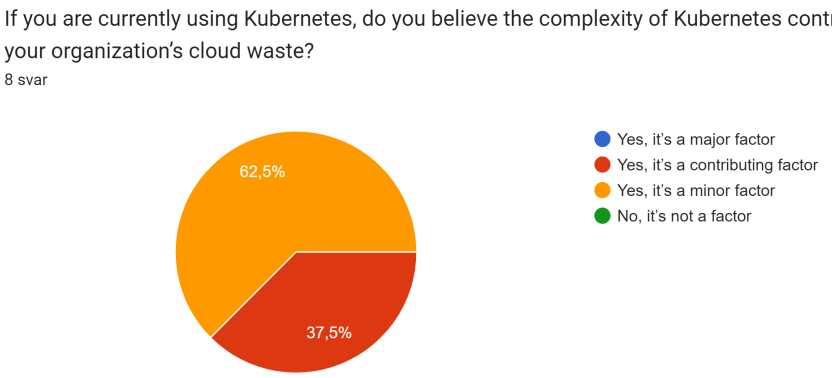

Figure A.7: Survey Sent to Developers at Intility: Question 7

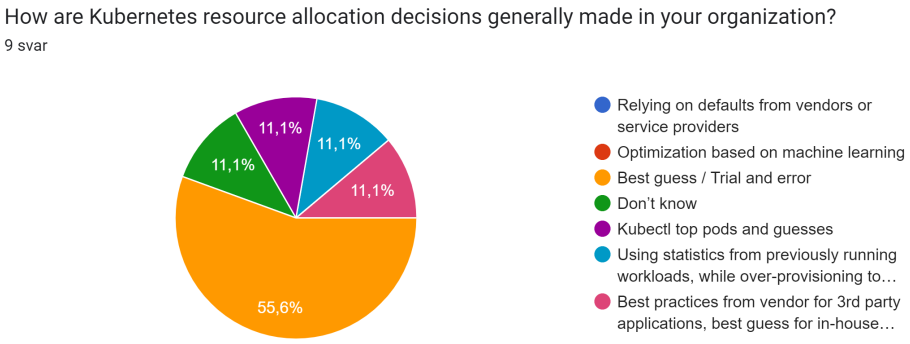

Figure A.8: Survey Sent to Developers at Intility: Question 8

Answers from Figure A.8

Relying on defaults from vendors or service providers

Optimization based on machine learning

Best guess / Trial and error

Don't know

Kubectl top pods and guesses

Using statistics from previously running workloads, while over-provisioning to ensure application performance

Best practices from vendor for 3rd party applications, best guess for in-house developed applications

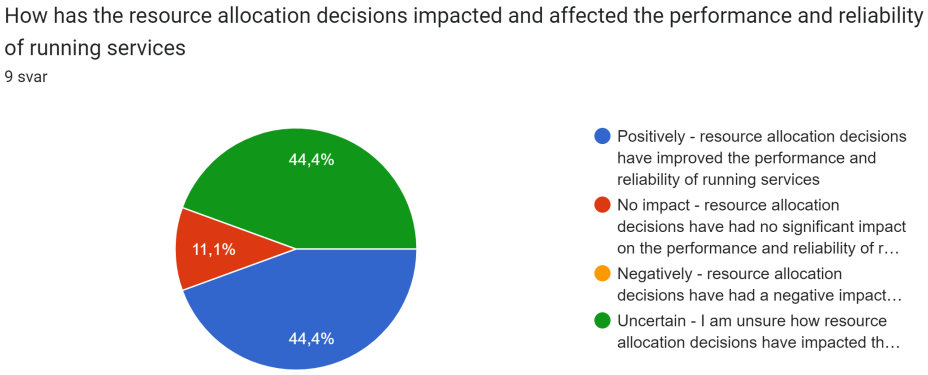

Figure A.9: Survey Sent to Developers at Intility: Question 9

Figure A.10: Survey Sent to Developers at Intility: Question 10

Appendix B

Python Code

B.1 Pre-processing Code

B.1.1 Reading and pre-processing .csv files

Code listing B.1: Reading and appending .csv files

```
# Reading and appending .csv files (ram)

container=1
df1 = pd.read_csv(f'B{container}.csv',
                  sep=',',
                 low_memory=False)

orig_name= df1.columns[1]
df1 = df1.rename(columns={orig_name: 'Containers: Memory usage'})

# Find the index of the last non-null value
last_non_null_index = df1['Containers: Memory usage'].last_valid_index()

# Remove rows with missing values after the last non-null value
df1 = df1.loc[:last_non_null_index]

# Counting missing values and find the percentage
#df1.info()
#print(df1['Containers: Memory usage'].value_counts())
#print(df1[' bytes - collector-xcfsq collector | collector'].value_counts())

# Standardized date format
df1['Date'] = pd.to_datetime(df1['Date'])

```

```
# converting n/a's to mean value
df1['Containers: Memory usage'] = df1['Containers: Memory usage'].apply(
    convert_memory_to_float)
df1['Containers: Memory usage'] = df1['Containers: Memory usage'].fillna(df1['
    Containers: Memory usage'].mean())

fig, ax = plt.subplots(figsize=(12, 6))
ax.set_xlabel('Date')
ax.set_ylabel('Memory Usage (MB)')
ax.set_title('Memory Usage Over Time container01')
ax.grid()

# Customize date ticks and format
ax.xaxis.set_major_locator(mdates.AutoDateLocator())
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
plt.setp(ax.get_xticklabels(), rotation=30, ha='right')

#plt.savefig(f'memory_usage_c{container}.png', dpi=300, bbox_inches='tight')
plt.show()
```

B.1.2 Converting memory to float

Code listing B.2: Converting memory to float

```
def convert_memory_to_float(value):
    # Check if the value is already numeric
    if isinstance(value, (int, float)):
        return float(value)

    # If value is a string, check the unit
    if isinstance(value, str):
        unit = None
        number = value.strip()
        if 'B' in value.upper() and 'MB' not in value.upper() and 'KB' not in value
            .upper():
            unit = 'B'
            number = value[:-1].strip()
        elif 'KB' in value.upper():
            unit = 'KB'
            number = value[:-2].strip()
        elif 'MB' in value.upper():
            unit = 'MB'
            number = value[:-2].strip()
        elif 'GB' in value.upper():
```

```
            unit = 'GB'
            number = value[:-2].strip()

        # Convert the numeric part to a float
        value = float(number)

        # Convert the value to MB based on the unit
        if unit == 'B':
            value = value / (1024 * 1024)
        elif unit == 'KB':
            value = value / 1024
        elif unit == 'GB':
            value = value * 1024

    return value
```

B.1.3 Calculating skewness

Code listing B.3: Calculating skewness in container

```
column_name = "Container: Memory usage"

skewness = df1['Containers: Memory usage'].skew()

print(df1['Containers: Memory usage'].median())
# Create a histogram with 10 bins and a density plot
fig, ax = plt.subplots()
ax.hist(df1['Containers: Memory usage'], bins=20, density=True, alpha=0.5, color="
    blue")
ax.set_xlabel(column_name)
ax.set_ylabel("Density")

ax.set_xlim(100, max(df1['Containers: Memory usage']))

# Add a vertical line at the median, mean, mode, and skewness value
ax.axvline(df1['Containers: Memory usage'].median(), color="red", linestyle="--",
    label="Median")
ax.axvline(df1['Containers: Memory usage'].mean(), color="orange", linestyle="--",
    label="Mean")
ax.axvline(df1['Containers: Memory usage'].mode()[0], color="green", linestyle="--"
    , label="Mode")
ax.axvline(skewness, color="purple", linestyle="--", label="Skewness: {:.2f}".
    format(skewness))

```

```
# Add a legend and title
ax.legend()
ax.set_title("Distribution of memory usage")

# Display the plot
#plt.savefig(f'memory_usage_distribution_c{container}.png', dpi=300, bbox_inches='
    tight')
plt.show()
```

B.1.4 Script for showing daily memory usage during a 2 week period

Code listing B.4: Script showing daily memory usage for a 2 week period

```

  def plot_24_hour_segments(df, column):
    days = df.index.normalize().unique()

    plt.figure(figsize=(12, 6))

    for day in days:
        day_data = df.loc[day:day + pd.Timedelta('1D') - pd.Timedelta('5m'), column
            ]
        plt.plot(day_data.index.time, day_data.values, label=day.date())


df_days=df1
df_days = df_days[['Date','Containers: Memory usage']]
df_days['Date'] = pd.to_datetime(df_days['Date'])
df_days.set_index('Date', inplace=True)



def plot_24_hour_segments(df, column, window_size=12):
    first_timestamp = df.index[0]
    days = df.index.normalize().unique()

    # Calculate end date
    last_full_day = df.index[-1].normalize() - pd.Timedelta('1D')
    end_date = last_full_day + pd.Timedelta(hours=first_timestamp.hour, minutes=
        first_timestamp.minute)

    # Apply rolling average
    df_smooth = df.rolling(window=window_size, center=True).mean()

```

```
    plt.figure(figsize=(12, 6))

    for day in days:
        start_time = day + pd.Timedelta(hours=first_timestamp.hour, minutes=
            first_timestamp.minute)
        end_time = start_time + pd.Timedelta('1D') - pd.Timedelta('5m')

        if end_time <= end_date:
            day_data = df_smooth.loc[start_time:end_time, column]

            if not day_data.empty:
                times = [t.hour * 60 + t.minute for t in day_data.index.time]
                plt.plot(times, day_data.values, label=day.date())

    plt.xlabel('Time of Day')
    plt.ylabel('Memory Usage in MiB')
    plt.legend()
    plt.title('Daily Memory Usage Patterns (Smoothed)')

    one_day_minutes = 24 * 60
    plt.xlim(0, one_day_minutes)

    # Set x-axis ticks and labels
    x_ticks = np.arange(0, one_day_minutes + 1, 2 * 60)  # Every 4th hour in
        minutes
    x_labels = [f'{t // 60:02d}:00' for t in x_ticks]
    plt.xticks(x_ticks, x_labels, rotation=45)

    plt.savefig('daily_memory_usage_patterns.png', dpi=300, bbox_inches='tight')
    plt.show()


plot_24_hour_segments(df_days, 'Containers: Memory usage')
```

B.2 Linear programming

Code listing B.5: The Linear Programming Model

```

# Linear programming modeling
import pulp

safety_margin_percentage = 10  # Additional memory allocated as a percentage of the
    max usage
```

```
safety_margin = max(df1['Containers: Memory usage']) * (safety_margin_percentage / 100)

# Calculate the typical memory usage (e.g., the mean or median)
typical_memory_usage = np.mean(df1['Containers: Memory usage']) * 1.05

# Defining the problem, and specifying that the objective is to minimize a function
prob = pulp.LpProblem("ContainerResourceAllocation", pulp.LpMinimize)

# Variables
x_request = pulp.LpVariable("x_request", lowBound=typical_memory_usage)
x_limit = pulp.LpVariable("x_limit", lowBound=max(df1['Containers: Memory usage']) + safety_margin)

# Objective function
prob += x_request

# Constraints
prob += x_request <= x_limit

# Solve the problem
status = prob.solve()

# Print the results
#print("Status:", pulp.LpStatus[status])
req_value = pulp.value(x_request)
lim_value = pulp.value(x_limit)
print("Request limit for Container 1:", pulp.value(x_request))
print("Memory limit for Container 1:", pulp.value(x_limit))

fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(df1['Date'], df1['Containers: Memory usage'])
ax.set_xlabel('Date')
ax.set_ylabel('Memory Usage (MB)')
ax.set_title(f'Memory Usage Over Time container0{container}')
ax.grid()

# Customizing date ticks and format
ax.xaxis.set_major_locator(mdates.AutoDateLocator())
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
plt.setp(ax.get_xticklabels(), rotation=30, ha='right')


ax.axhline(y=req_value, color='r', linestyle='--', label='Request')
ax.axhline(y=lim_value, color='b', linestyle='--', label='Limit')
```

```
49
50 ax.legend()
51
52 # print plot
53 plt.savefig(f'memory_usage_linear_programming_c{container}', dpi=300, bbox_inches='
       tight')
54 # Show the plot
55 plt.show()
```

B.2.1 Calculating The MREI

Code listing B.6: Calculating the MREI

```
1  def calculate_kpi(df, request_value):
2      """
3      Calculate the percentage of container usage memory datapoints under the set
           request.
4
5      :param df: DataFrame with container memory usage data
6      :param request_value: Request limit for the container
7      :return: KPI as a percentage
8      """
9      under_request = df[df['y'] < request_value]
10     percentage = (len(under_request) / len(df)) * 100
11     return percentage
```

B.3 ARIMA

Determining Differencing

Code listing B.7: Determining differencing for the ARIMA model

```
1
2  adf_p_value_0 = sm.tsa.stattools.adfuller(df1['Containers: Memory usage'])[1]
3  adf_p_value_1 = sm.tsa.stattools.adfuller(df1['Containers: Memory usage'].diff().
       dropna())[1]
4  adf_p_value_2 = sm.tsa.stattools.adfuller(df1['Containers: Memory usage'].diff().
       diff().dropna())[1]
5
6  # Create a single figure with the desired layout
7  fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(18, 8))
8
9  # Without Differencing
```

```
axes[0, 0].set_title(f'Without Differencing (p={adf_p_value_0:.2e})')
axes[0, 0].plot(df1['Containers: Memory usage'])
sm.graphics.tsa.plot_acf(df1['Containers: Memory usage'].dropna(), ax=axes[1, 0])

# 1st Order Differencing
axes[0, 1].set_title(f'1st Order Differencing (p={adf_p_value_1:.2e})')
axes[0, 1].plot(df1['Containers: Memory usage'].diff())
sm.graphics.tsa.plot_acf(df1['Containers: Memory usage'].diff().dropna(), ax=axes
    [1, 1])

# 2nd Order Differencing
axes[0, 2].set_title(f'2nd Order Differencing (p={adf_p_value_2:.2e})')
axes[0, 2].plot(df1['Containers: Memory usage'].diff().diff())
sm.graphics.tsa.plot_acf(df1['Containers: Memory usage'].diff().diff().dropna(), ax
    =axes[1, 2])

plt.savefig(f'memory_usage_plots_with_adf_c{container}.png', dpi=300, bbox_inches='
    tight')

# Show the plots
plt.show()
```

B.3.1 ACF and PACF

Code listing B.8: Calculating ACF and PACF for ARIMA

```
stationary_data = df1['Containers: Memory usage'].diff().dropna()
# Create ACF and PACF plots
fig, axes = plt.subplots(2, 1, figsize=(12, 8))

sm.graphics.tsa.plot_acf(stationary_data, lags=40, ax=axes[0])
sm.graphics.tsa.plot_pacf(stationary_data, lags=40, ax=axes[1])

plt.savefig(f'memory_usage_plots_PACFnACF_c{container}.png', dpi=300, bbox_inches='
    tight')
plt.show()
```

B.3.2 Fitting the ARIMA model

Code listing B.9: Fitting the Arima model

```

memory_usage = df1['Containers: Memory usage']
```

```
3
4  train_size = int(len(memory_usage) * 0.80)  # 80% of the data for training
5  train_data = memory_usage[:train_size]
6  test_data = memory_usage[train_size:]
7
8  p,d,q = 1,2,1
9
10 model = ARIMA(train_data, order=(p, d, q))
11 results = model.fit()
12
13 predictions = results.predict(start=train_size, end=len(memory_usage) - 1, dynamic=
       True)
14
15 # Plot the training data, test data, and predictions
16 fig, ax = plt.subplots(figsize=(12, 6))
17 ax.set_xlabel('Date')
18 ax.set_ylabel('Memory Usage (MB)')
19 ax.set_title(f'Memory Usage Over Time with ARIMA Predictions for c{container}')
20 ax.grid()
21
22 # Customizing date ticks and format
23 ax.xaxis.set_major_locator(mdates.AutoDateLocator())
24 ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
25 plt.setp(ax.get_xticklabels(), rotation=30, ha='right')
26
27 ax.plot(df1['Date'][:train_size], train_data, label='Training Data')
28 ax.plot(df1['Date'][train_size:], test_data, label='Test Data')
29 ax.plot(df1['Date'][train_size:], predictions, label='Predictions')
30
31 ax.legend()
32
33 # Save and show the plot
34 plt.savefig(f'arima_memory_usage_c{container}.png', dpi=300, bbox_inches='tight')
35 plt.show()
```

B.3.3 Test Data Underprediction Rate (TDUR)

Code listing B.10: Calculating TDUR

```
1
2  def calculate_kpi_predictions(df, predictions):
3      """
4      Calculate the number of container usage memory datapoints under the predicted
           values.
```

```
5
6       :param df: DataFrame with container memory usage data
7       :param predictions: Predicted values from the ARIMA model
8       :return: Number of datapoints under the predicted line
9       """
10      test_data = df[-len(predictions):]  # Get the test data corresponding to the
            predictions
11      under_prediction = test_data[test_data['Containers: Memory usage'] <
            predictions]
12      percentage = (len(under_prediction) / len(test_data)) * 100
13      return percentage
14
15  # Assuming the 'predictions' variable contains the ARIMA model predictions
16  kpi = calculate_kpi_predictions(df1, predictions)
17  print(kpi)
```

B.3.4 MAE, MAPE and RMSE

Code listing B.11: Calculating MAE, MAPE, and RMSE

```
1
2   def mean_absolute_percentage_error(y_true, y_pred):
3       y_true, y_pred = np.array(y_true), np.array(y_pred)
4       return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
5
6   mae = np.mean(np.abs(predictions - test_data))
7   mape = mean_absolute_percentage_error(test_data, predictions)
8   rmse = np.sqrt(np.mean((test_data - predictions) ** 2))
```

B.4 Facebook's Prophet Model

Code listing B.12: Facebook's Prophet Model

```
1   model = Prophet()
2
3   df = pd.DataFrame({'ds': df1['Date'], 'y': df1['Containers: Memory usage']})
4   df['ds'] = pd.to_datetime(df['ds'])
5
6   # Split the data into train and test sets
7   train_size = int(len(df) * 0.8)  # 80% of the data for training, 20% for testing
8   train_df = df[:train_size]
9   test_df = df[train_size:]
```

```
10
11
12 # Fit the model on the training data
13 model.fit(train_df)
14
15 # Make predictions on the test data
16 future = model.make_future_dataframe(periods=len(test_df), freq='5min',
       include_history=False)
17 forecast = model.predict(future)
18
19 # Plot the forecast
20
21 fig = model.plot(forecast)
22 ax = fig.gca()
23 ax.set_title(f'Prophet Forecast for c{container}')
24 ax.set_xlabel('Date')
25 ax.set_ylabel('Memory Usage (MB)')
26
27 # Plot the test data
28 ax.scatter(test_df['ds'], test_df['y'], color='purple', label='Test Data',s=10)
29
30 # Add a legend
31 ax.legend(['Observed', 'Forecast', 'Uncertainty Interval', 'Test Data'])
32 plt.savefig(f'prophetC{container}.png', dpi=300, bbox_inches='tight')
33
34 plt.show()
```

B.5 P+LP

Code listing B.13: Finalized P+LP model, included plotting

```
1  model = Prophet(changepoint_prior_scale=0.01)
2
3  df = pd.DataFrame({'ds': df1['Date'], 'y': df1['Containers: Memory usage']})
4  df['ds'] = pd.to_datetime(df['ds'])
5
6  orig_req=15
7  orig_lim=15
8
9  # Split the data into train and test sets
10 train_size = int(len(df) * 0.8)  # 80% of the data for training, 20% for testing
11 train_df = df[:train_size]
12 test_df = df[train_size:]
```

```
13
14 # Fit the model on the training data
15 model.fit(train_df)
16
17 # Make predictions on the test data
18 future = model.make_future_dataframe(periods=len(test_df), freq='60min',
       include_history=False)
19 forecast = model.predict(future)
20
21 # Plot the forecast
22 fig = model.plot(forecast)
23 ax = fig.gca()
24 ax.set_title(f'P+LP Forecast for c{container}')
25 ax.set_xlabel('Date')
26 ax.set_ylabel('Memory Usage (MB)')
27
28 # Plot the test data
29 ax.scatter(test_df['ds'], test_df['y'], color='purple', label='Test Data', s=10)
30
31 # Calculate the new request and limit lines
32 new_request = np.mean(forecast['yhat']) * 1.05  # Adding 10% safety margin
33
34 max_observed = max(train_df['y'])
35 max_predicted = max(forecast['yhat_upper'])
36 higher_value = max(max_observed, max_predicted)
37 new_limit = higher_value * 1.15  # 10% over the higher value
38
39 # Old request and limit lines calculation
40 safety_margin_percentage = 15  # Additional memory allocated as a percentage of the
       max usage
41 safety_margin = max(train_df['y']) * (safety_margin_percentage / 100)
42 typical_memory_usage = np.mean(train_df['y']) * 1.05
43
44 # Define the problem
45 prob = pulp.LpProblem("ContainerResourceAllocation", pulp.LpMinimize)
46
47 # Variables
48 x_request = pulp.LpVariable("x_request", lowBound=typical_memory_usage)
49 x_limit = pulp.LpVariable("x_limit", lowBound=max(train_df['y']) + safety_margin)
50
51 # Objective function
52 prob += x_request
53
54 # Constraints
55 prob += x_request <= x_limit
```

```
56
57 # Solve the problem
58 status = prob.solve()
59
60 # Get the old request and limit values
61 req_value = pulp.value(x_request)
62 lim_value = pulp.value(x_limit)
```

B.6 Performance Score

Code listing B.14: Calculating Performance Score

```
1  def calculate_score(mrei, resource_request_difference, ideal_target=95, wiggle_room
       =4, mrei_weight=0.7, diff_weight=0.3):
2      mrei_difference = abs(mrei - ideal_target)
3      score_factor = abs(1 - (mrei_difference / ideal_target))
4
5      if resource_request_difference > 0:
6          resource_request_difference = 0
7
8      if mrei == ideal_target:
9          performance_score = 100
10     elif mrei_difference <= wiggle_room:
11         mrei_component = score_factor * mrei_weight * 100
12         diff_component = abs(resource_request_difference) * diff_weight
13         performance_score = mrei_component + diff_component
14     else:
15         mrei_component = score_factor **2 * mrei_weight * 100
16         print(mrei_component)
17         diff_component = abs(resource_request_difference) * diff_weight
18
19         performance_score = mrei_component + diff_component
20
21     # Clip the performance score to be between 0 and 100
22     performance_score = max(0, min(100, performance_score))
23
24     return performance_score
25
26 # Example usage
27 mrei = 100
28 resource_request_difference = +80
29 score = calculate_score(mrei, resource_request_difference)
30
```

```
print(f"Performance Score: {score}")
```