CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Automatic container profiling and resource prediction using container orchestration

Predicting containerized cloud application's resource allocation using automatic container profiling and container orchestration

Master's thesis in Computer Science and Engineering -
Software Engineering and Technology

JOEL SANDERÖD ROXELL
MATTIAS LUNDELL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Automatic container profiling and resource prediction using container orchestration

Predicting containerized cloud application's resource allocation using automatic container profiling and container orchestration

JOEL SANDERÖD ROXELL
MATTIAS LUNDELL

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Automatic container profiling and resource prediction using container orchestration
Predicting containerized cloud application's resource allocation using automatic container profiling and container orchestration
JOEL SANDERÖD ROXELL
MATTIAS LUNDELL

Supervisor: Philipp Leitner, Computer Science and Engineering
Advisor: Vito Cusumano, Ericsson
Examiner: Jennifer Horkoff, Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Automatic container profiling and resource prediction using container orchestration
Predicting containerized cloud application's resource allocation using automatic container profiling and container orchestration
JOEL SANDERÖD ROXELL
MATTIAS LUNDELL
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As more organizations are moving to a cloud-based infrastructure, it is becoming increasingly important to consider efficient resource management. Cloud technologies such as container orchestrators offer simplified solutions that may help an organization scale its clusters based on pre-defined thresholds set by the organization. However, there is a risk of over-allocating resources if these threshold values are inaccurate or set in an ad-hoc manner. Previous research has successfully used artificial intelligence and machine learning to dynamically adjust the size of the cluster by predicting the future resource needs on a virtual machine. However, while it may provide a good indication on which direction the cluster is changing, the underlying cause of the change is lost. Therefore, this thesis aims to explore, evaluate, and implement an automatic profiler for containerized applications so that organizations may track and estimate both their containers' and the total cluster's resource needs. Furthermore, this thesis gives answers to the research questions: how can containers be profiled to describe their actual usage of resources accurately and whether or not a profile-based cluster lead to less resource consumption. The study was done as a case study in collaboration with Ericsson. By involving an industrial partner, live data from both a test and a production environment hosted by Ericsson was used in the analysis. The outcome of the study was a system comprised of three submodules. First, a data collector that automatically registers containers and their historical resource usage in a database by querying the metrics API of the container orchestrator. Next, a system that queries the database and, using K-means clustering, assigns labels to each container based on their relative resource usage. Finally, a predictor, based on a recurrent neural network and long short-term memory cells, that predicts a container's future resource needs. The results show that historical records of a container's resource usage are an effective way of profiling individual containers. By aggregating all containers', the total resource usage of the cluster is easily obtained without losing track of individual containers. The total resource usage may also be transformed to represent the actual node count needed at any given time, which may help an organization adjust its node count to more optimally fit the actual need. Furthermore, by dynamically adjusting the node count according to the predictions, the analysis shows that there are financial savings to be made when comparing against a constant node count.

Keywords: computer science, machine learning, cloud computing, container orchestration, container profiling, container labeling, resource predictions, apcera, kubernetes.

Acknowledgements

We want to thank the CVC team at Ericsson for being welcoming and helpful during our stay. The team shared valuable knowledge and provided us with constructive insights, which helped us finish our study. A special thanks to Vito Cusumano and Per-Johan Wiberg, our study would not have been possible without their help and feedback.

We would also like to thank Philipp Leitner for always challenging us with his spot-on and constructive feedback as well as guiding us in becoming better researchers. The outcome of this study would not have been half of what it is without him and his support.

Joel Sanderöd Roxell & Mattias Lundell, Gothenburg, August 2019

Contents

List of Figures

List of Tables

1
Introduction

Most cloud platforms provide the possibility to run containerized applications (containers) in clustered environments which allows companies to move to more scalable infrastructure patterns, such as the microservice architecture. At its core, a container is a standalone executable package of software where all necessary runtime components of an application are bundled together [1]. In contrast to virtual machines (VMs), which also can be used to isolate software, containers are an abstraction at the application layer while VMs are an abstraction at the hardware level. To illustrate, if three applications are isolated into VMs, each VM would consist of a full copy of the operating system (OS) as well as the application and all necessary runtime components. In contrast, if the same applications instead were containerized, they would share the same OS kernel while still being run as isolated processes, making them much more lightweight and resource efficient.

The convenience and simplicity of running isolated applications inside containers produce less notable drawbacks, e.g., the issue of over-allocating resources, i.e., provision more resources to the comprised cluster than the applications need to operate efficiently. Ericsson, the industrial partner of this thesis, describes a normal behavior which indicates that developers do not spend much thought on determining containers' resource requirements, such as CPU or memory, due to the absence of tooling. This behavior may lead to the risk of over-provisioning of resources if an arbitrary number of nodes (VMs) are attached to the cluster to run the containers.

A deployed container without any resource constraints is allowed to use as much of a given resource as the host kernel will allow, if there is not a hard boundary, all available resources may eventually be consumed. Besides, if the host kernel identifies that there is not enough memory left on the device to perform critical system functions, it will start to discharge out of memory-exceptions and begin to clear memory by halting arbitrary non-critical processes [2]. The process of resource exception handling in combination with the probability of placing containers with similar volatile resource demand on the same host may lead to individual starvation and finally cause an abrupt termination of the specific container.

A worst-case scenario would, e.g., be a container that has a significant setup and warmup time. The rate of traffic increases four-fold yet predictable due to typical user behavior. An auto-scaled system would be too slow to adapt to the increase, initially. Furthermore, the container orchestrator, which autonomously manages the deployment and scaling, may put two different containers onto the same node. Both with similar memory profiles, eventually one is forced to terminate due to starvation, further extending the downtime of that particular service. Without an accurate baseline or container profile, it is hard to determine what the actual

size of the cluster should be at a specific point in time. The main issue, according to Ericsson, is that a single deployment usually specifies a static node count, which is never changed, or at least not empirically determined.

Existing solutions to date allow clusters to reactively auto-scale using thresholds, i.e., they increase or decrease in node count based on computing resources such as CPU or memory. Threshold-based auto-scaling is enough if the number of nodes match the current rate of traffic and oscillates moderately. However, this assumes that the correct resource thresholds already are determined for the cluster. Due to the uncertainty, organizations configure clusters to run at a setting that is considered sufficient and these configurations seldom have a basis in any empirical evaluation. Consequently, the organization will commonly procure additional resources, increase cost, and consume more energy [3].

Over time, a container usually exerts a distinct behavior which can be seen as its profile. In other words, a container's profile is the aggregation of its resource usage over time, e.g., broadly speaking a system might be used more during the day compared to after-hours. Previous research shows that the application of techniques such as reinforcement learning may produce sound results, with regards to similar time-series [4]. However, most research focus on resource allocations on the level of VMs. While this is a good predictor for the size of the total cluster, the technical aspect of which container is consuming what is lost. Therefore, one can assume in what direction the cluster is going to grow and toward which thresholds, though have no trace of the actual origin of the change.

A cluster's total resource requirements may be determined using the allocation of each container as the footprint of Docker is negligible [5]. This thesis presents a method of tracking each deployed container in the cluster, collect its profile, and later predict the comprised allocation of the entire cluster without losing the crucial information of each container's resource allocation. The produced information may be used to strategize deployments or fundamentally restrict specific containers from running on the same node if they consume the same type of resource at specific hours.

The default resource allocation depends on the container orchestrator and the container engine. We will focus on Docker containers, specifically in this thesis as it is considered to be the standard way of containerizing applications. As container orchestrator, we will make use of both Apcera and Kubernetes, the former being what Ericsson is using and the latter being an integral part of the open-source community and maintained by the Cloud Native Computing Foundation. [6], [7].

1.1 Goal

The purpose of this study is to explore, evaluate, and implement an automatic profiler for containerized applications so that organizations may track and estimate their containers' resource requirements.

Firstly, a collector service will gather data and create profiles for each specific container's resource requirements. Next, the collected data will be consumed by another service, which will assign labels to the containers. Labels indicate what type of resources the container utilizes most and how volatile it is in relation to others.

In essence, these services determine the internal application's default requirements and their deviations. Subsequently, the containers will have an appropriate profile, or baseline, which may be used to prevent applications from competing for the same resources of the host system, by repelling specific applications from each other to separate nodes. Lastly, previous findings in this field of research will be applied to predict individual services' increase or decrease in utilized resources, based on their individual profile.

The process should yield an adequate method for maintaining the underlying assets that define the clusters' resource availability since the aggregate of all container resources will reflect the total size of the cluster. Ultimately, giving an organization the possibility to proactively increase or decrease the number of actual nodes connected to the cluster, yet keeping the knowledge of which container(s) contribute to the result. In either case, it will provide the organization with an empirical evaluation and adjustment-proposal of their current cluster-based infrastructure.

1.2 Research Questions

Each container running in the cluster has an underlying resource history which defines its fundamental resource needs to operate efficiently. The idea is to provide a solution for collecting this internal information and derive profiles. These profiles may define both upper and lower bounds, in terms of resource usage. Nevertheless, the container should be able to grow if needed using conventional threshold techniques implemented by the orchestrator. Still, the sum of the containers' profiles affects the size of the cluster.

Developers usually set restrictions based on experience rather than on empirical data [3]. Hence, wasted resources may be present due to a misconfiguration of one or more resources, alternatively a poor node count configuration for the cluster. Based on these issues and their current intractability, this thesis aims to find answers to the following research questions.

RQ 1: How can containers be profiled to describe their actual usage of resources accurately?

The goal of this research question is to identify a way to describe the profile of containers based on their actual usage of resources. The profiles may be used to calculate the actual resource requirements of the cluster, thus having the potential to determine the presence of wasted resources by comparing the result to the cumulative resource tally of the cluster's connected nodes. Furthermore, if accurate profiles can be established, it would also open up the possibility of labeling containers based on their resource usage. Labeling containers this way would allow for a more intelligent deployment where, e.g., two memory-heavy containers can be separated from each other to increase the robustness of the cluster.

RQ 2: Does a profile-based cluster lead to less resource consumption while retaining or improving its availability?

As an extension of the first research question, the aim is to determine if the predicted resources would yield a significant difference in resource allocation compared to pre-existing methods. The hypothesis is that empirically defined container profiles would reduce the size of the cluster to a near optimal size and thus reduce the unnecessary waste of resources.

1.3 Approach

To answer the research questions and achieve the overarching goals, the study is done as a case study in collaboration with Ericsson as an industrial partner. Ericsson provides expert knowledge in the field as well as a test and production environment from which data about containers' resource usage can be gathered and analyzed. The primary outcome of this study is a containerized system that may be deployed in a container-based cluster. When deployed, the system will autonomously gather data and register all deployed containers in a database. Next, a sub-system will query the database and label each container based on their relative resource usage. Finally, another sub-system will predict future resource requirements of individual containers based on the gathered data and the assigned labels.

The labeling process is based on the K-Means clustering algorithm. That is, the system will group containers with similar resource usage into clusters and assign a corresponding label. How well the system performs is evaluated in terms of the silhouette score, outlined in Section 5.2.

When making predictions, the system uses a recurrent neural network (RNN) with long short-term memory cells (LSTM) trained separately for each container. The predictions are evaluated in terms of mean absolute error (MAE), mean absolute percentage error (MAPE), and r^2. Additionally, the results are also compared against previous research.

Furthermore, the gathered data is analyzed and compared to the current configuration of the cluster. By summing the data, the total size of the cluster can be determined in terms of actual resource usage. By comparing the actual usage with the allocated resources, any potential waste, as well as financial and environmental savings, may be determined.

1.4 Outline

The thesis is outlined as follows. A background description of the industrial context, cloud technologies used, and machine learning techniques applied is given in Section 2. This is followed by a review of related work to this study in Section 3. A description of the research methodology and the system developed is found in Section 4 and Section 5. The result from running the system in both a test and production environment is presented in Section 6, followed by a discussion of the lessons learned in Section 7. Finally, in Section 8, our conclusions of the study are drawn by following

up the overaarching objective and the research questions as well as proposing future work.

2
Background

This section aims to provide the necessary background knowledge of the industrial context, Section 2.1, the cloud technologies used, Section 2.2, and the machine learning techniques applied, Section 2.3, for thesis thesis.

2.1 Industrial Context

Ericsson is a leading Swedish organization in the information and communications technology (ICT) sector with a history that dates back to the year of 1876 [8]. Today, the organization is present on all continents of the world and offer their customers a wide range of solutions, products, and services in four different business areas: Networks, Digital Services, Managed Services, and Internet of Things (IoT) and New Business [9]. This study was done in collaboration with the Connected Vehicle Cloud (CVC) department, which is part of the IoT and New Business area, and the Research and Development (R&D) department.

CVC is an IoT solution offered to original equipment manufacturers (OEMs) in the automotive industry. The primary objectives with the solution is to support their customers' increasing needs for scalability, security, and flexibility as the automotive industry and vehicles become more digitalized and connected. A key feature of CVC is a cloud platform which OEMs may use to deploy containerized applications and services used by their customers. That is, Ericsson is hosting and maintaining the cloud to ensure that OEMs can focus on the development and innovation of new applications and services. [10]

In order to fulfill the objectives and to answer the research questions, the study was done on-premise at Ericsson's office at Lindholmen, Gothenburg between February and July 2019. Other than expert knowledge from their long history in the ICT sector, Ericsson also provided both a test environment in which the development could take place, and a live production environment from which data could be gathered and analyzed. Further details about the research methodology and the developed system can be found in Sections 4 and 5.

2.2 Cloud Technologies

The purpose of this section is to provide an overview of the essential cloud technologies used and discussed in this thesis. The first section, Section 2.2.1, introduces cloud providers in general and, in particular, Amazon Web Services (AWS). The concept of clusters and containers are then presented in Sections 2.2.2 and 2.2.3.

Finally, In Section 2.2.4, container orchestrators such as Kubernetes and Apcera are described.

2.2.1 Cloud Providers

Cloud providers are companies that offer different cloud computing (CC) services, usually in a pay-as-you-go fashion. The specific CC service relevant for this thesis, known as Infrastructure as a Service (IaaS) [11], allows customers to procure on-demand computing capacity in the cloud instead of making a significant up-front investment in a traditional IT infrastructure. This study is using Amazon Elastic Compute Cloud (Amazon EC2) as the underlying infrastructure. While alternatives exist, this thesis is limited to Amazon EC2 as that is the provider Ericsson was using at the time of the study.

Amazon EC2 is accessed using either a graphical user interface (GUI) or a command-line interface (CLI) and provide a range of different functionalities, ranging from security configurations, resource-groups, auto-scaling, and volume management [12], [13]. Either interface is used to create a set of VMs of some specific configuration and later accessed using secure shell (SSH). System administrators access these VMs, configures them, and finally joins them to form a combined computational resource known as a cluster.

One of the perks of using a cloud provider is that one quickly can spin up a new VM (node) and attach it to the cluster to increase the cluster's computing capacity. Similarly, one can detach a node and terminate it to release allocated cloud resources, hence reduce cost. However, while Amazon EC2 provides customers with more flexibility in their IT infrastructure by allowing features such as auto-scaling, it does not solve the problem of over-allocation of resources if the underlying container profiles are not accurately determined and tracked.

2.2.2 Clusters

Clusters are used to improve performance and availability in contrast to a single computer, while typically being more cost-effective than single computers of comparable speed or availability [14]. Generally, organizations use either AWS or similar services such as Google Cloud [15] or Microsoft Azure [16] to create VMs. Next, the allocated VMs are joined to form the cluster, where a single VM is referenced as a node once it is attached to the cluster. Finally, the cluster orchestrator can run distributed tasks on the comprised computing resources.

Users of the system may send jobs and receive results using the employed orchestrator which in turn dispatch jobs onto the comprised resource [17], [18]. However, they need not necessarily know how and where jobs get executed in the cluster. Nevertheless, tools such as service discovery [19], [20] are provided by the orchestrator that may be used to communicate with the internal services. More specifically regarding this thesis, a job is fundamentally a running container which has been dispatched to run on the cluster by an orchestrator.

The orchestrator is used to abstract the internal logistics of maintaining the executed containers, which otherwise is a quite complex and tedious task. The or-

chestrator place containers where they fit best depending on an internal deployment strategy and give the organization the guarantee that the deployed containers are executed somewhere in the cluster, indefinitely if that is the case, assuming they do not crash or gets terminated. In such a case, the container is typically rebooted.

In Amazon EC2, the cloud provider used in this study, nodes that make up the cluster are referred to as instances. While there are several different instance types, this thesis assumes that each node in the cluster is either an Amazon EC2 m5.large instance with the default configuration of 8 GB memory and two vCPU (two CPU cores times two processes per CPU core) or a m5.xlarge instance 16 GB memory and four vCPU when estimating the cost and size of the cluster [21]. There are two reasons for this: (1) to ensure some degree of confidentially concerning Ericsson's cluster configuration, and (2) the fact that the M5 instances are the latest generation of what Amazon refers to as general-purpose instances.

2.2.3 Container

A container is a lightweight, standalone, unit of software that packages source code and all its dependencies so that the application may operate reliably from one computing environment to another [1].

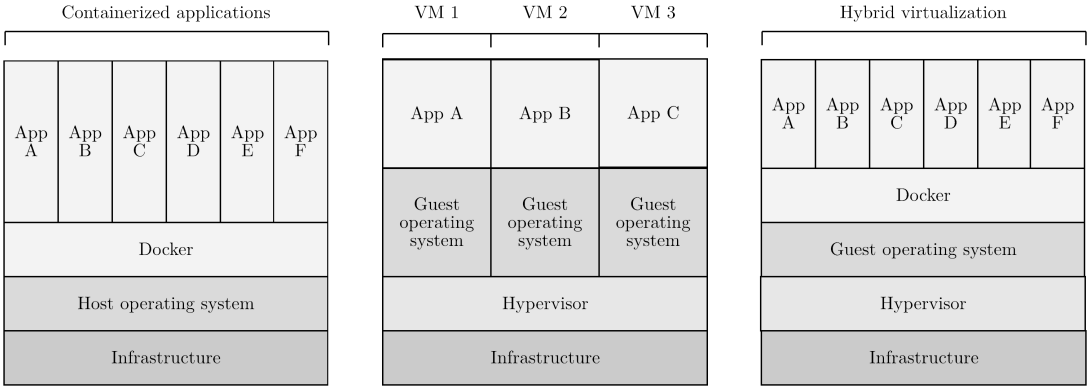

Figure 2.1: A high-level illustration of the fundamental difference between containers and VMs.

The container paradigm was derived as a remedy for the problematic layout of VM-based applications which are more prone to over-allocation and dreadful administration, as illustrated in Figure 2.1. Previously, either a single VM was used to run a single application or packed with as many applications as possible. The first case is prone to waste since a VM allocates a static amount of resources. Additionally, there is significant work to determine what size of a VM best fit the different applications resource requirements. The following problem is the embodiment of an administrator worst nightmare. How many applications are running on the machine? Which applications depend on which dependencies? Which applications depend on other applications running on the same machine? How does one quickly scale horizontally using those configurations? Are applications inadvertently

affecting each other? The Docker platform is, in many ways, a response to mitigate these problems by providing a simple method for sharing and utilizing resources more efficiently compared to that of a VM.

Since containers do not need the additional load of a hypervisor, each container runs directly within the host machine's kernel. As a result, Docker containers may run on any given hardware combination in contrast to VMs, and may even run inside a VM.

Despite many benefits, the container paradigm does not solve the problem of over-allocation. Containers are by default allowed to use as much of a given resource as the host kernel will allow, given that no resource constraints are defined. Even if constraints are applied, Ericsson explains that they are rarely derived from empirical data and infrequently updated. Consequently, if no constraints are configured, there is a risk of having one container starving other containers. Besides, if poorly defined constraints are set, there is a risk of over-allocating resources and thus accumulate unnecessary costs.

2.2.4 Container Orchestrators

A container orchestrator manages the lifecycles of containers in large and dynamic environments. It is used to manage and automate tasks such as deployments, redundancy, scaling, resource allocation, external exposure, network configuration, service-discovery, health-monitoring and more [18], [22]. While different orchestrators exist, this thesis is limited to two: (1) Apcera, due to it being the tool Ericsson is using as their foremost orchestrator, and (2) Kubernetes, due to it being used during our local development and to increase the generalizability of the system being developed. Both orchestrators are systems that run in clustered environments as described in Section 2.2.2. Additionally, more details about the system's interfaces and integrations can be found in Section 5.

Apcera encapsulates different workloads that are deployed as a job, abstracting the complexities of each application. It ensures that the executed code runs with the correct environment and packages. The jobs are essentially deployed either by using functions in the CLI or by specifying a manifest file. A job is a workload that runs, manages a container, and enables administrators to update and oversee the system as a whole. A workload may be anything ranging from simple code to complex systems. In the case of this thesis, a workload will be synonymous with a Docker container. The process of isolating everything as jobs allows Apcera to apply the same orchestration and governance pattern to all deployed variants [18].

For Kubernetes, one uses defined Kubernetes API objects to construct the desired state of the cluster similar to manifest files in Apcera. The state is constructed by the aggregate of such files, which have been applied to the cluster using the Kubernetes API. A deployment file describes what container to run in the cluster, with what type of environment configuration, and what dependencies. A configuration is composed of attributes such as replica count, resource limitations, requirements, and service exposure. Once a desired state is defined, Kubernetes work to make the cluster's current state match the desired. In order to do that, Kubernetes uses a range of tools and tasks to modify the attributes mentioned above, i.e., stopping,

starting, restarting containers, or scaling container replica counts [17].

Regardless of both orchestrators ability to automate much of the management aspect of running containers in a cluster, poorly configured deployments or manifests lead to an increased risk of running over-allocating clusters. However, both orchestrators have a metrics API which can be utilized to collect information about both the actual nodes connected to the cluster and the internal containers running on each node. This thesis presents a method for efficiently collecting that information so it can be used to (1) construct container profiles based on their actual usage, (2) label containers based on their relative resource usage, (3) predict future resource needs to preemptively scale the cluster rather than being reactive, and (4) ensure traceability of individual containers.

2.3 Machine Learning Techniques

The purpose of this section is to give a general introduction to the machine learning algorithms and techniques applied in this study. First, the clustering algorithm, K-Means clustering, used to label containers is presented in Section 2.3.1. Next, the general approach of artificial neural networks is described in Section 2.3.2 followed by the more specific approach of recurrent neural networks and long short-term memory cells in Section 2.3.3

2.3.1 K-Means Clustering

Unsupervised learning is a branch of machine learning where the objective is to explore the underlying structure of the data. The name, unsupervised learning, is derived from the fact that, in contrast to supervised learning, the analyzed data does not have any pre-defined labels. The algorithm has to figure out relationships on its own, without any guidance from the data. In this thesis, one of the goals stated in Section 1.1 is to label individual containers based on their relative resource usage at specific points in time. Therefore, applying an unsupervised learning algorithm is fitting as there are no pre-defined thresholds. A simple yet computationally efficient algorithm of that type is the K-means clustering algorithm.

The objective of the K-means algorithm is to partition the data into a pre-defined number of groups or clusters, based on similarities in the features of the members [23]–[25]. The first step of the algorithm, seen in Algorithm 1, is to define the number of clusters, k, the algorithm should find. This step is also one of the drawbacks of this algorithm. Given that the input data is unlabeled, it might be difficult to know how many clusters could, or should, be found in the data. In this thesis, the algorithm will be used to label the relative resource usage of containers. That is, we want the algorithm to cluster containers into groups of different levels of resource usage. From a practical viewpoint, it seems sensible to look for groups of containers representing low, medium, and high resource usage, hence, in this thesis we assume $k = 3$. Step 2 is to randomly initialize the centroids, c_k, for each cluster, k. The next step is to assign each observation, x, in the data set, X, to the nearest centroid using the euclidean distance: $d = \sqrt{(x - c_k)^2}$. Once all observations have been assigned to a cluster x_k, the centroids are updated. First,

the means of the newly constructed clusters, μ_k, are computed by dividing the sum of the observations, x_k, with the number of observations in that cluster, N_k. Each centroid is then updated to be the mean of its cluster. Finally, the assignment is evaluated in terms of whether or not any re-assignments were made. That is, if any of the centroids were updated after the assignment. If true, the algorithm will return to step 3 and re-assign each observation to the updated centroids. Otherwise the algorithm has converged [26]–[28].

Algorithm 1 K-Means Clustering

Step 1: Define the number of clusters
1: $k \leftarrow [0, \dots]$
Step 2: Initialize centroids
2: $c_k \leftarrow$ random value
End For
Step 3: Assignment
For each $x \in X$
3: Assign x to the nearest c_k
4: $x_k \leftarrow x$
End For
Step 4: Update centroids
For each c_k
5: $\mu_k \leftarrow \frac{1}{N_k} \sum x_k$
6: $c_k \leftarrow \mu_k$
Step 5: Evaluation
7: **if** any c_k was updated **then**
8: Go to step 3
9: **else**
10: The algorithm has converged
11: **end if**

2.3.2 Artifical Neural Networks

Previous research, Section 3, has successfully used Artificial neural networks (ANNs) to predict future resource needs from historical records. Therefore, the core of the prediction model presented in this thesis is based on the fundamental concepts of a feed-forward ANN. On a high-level, ANNs can be described as a collection of supervised and unsupervised learning algorithms. The inspiration of the human brain and its learning process is found in two specific areas: (1) that knowledge is gathered from the environment through a learning process, and (2) that the knowledge is stored in inter-neuron connections [28]. For supervised ANNs similar to the prediction model presented in this thesis, the knowledge gathering and learning process is done by iteratively mapping inputs to output while storing the knowledge in weight vectors representing the inter-neuron connections.

In general, a supervised ANN consists of three types of layers as depicted in Figure 2.2. The first type, known as the input layer, is a vector of features that

represents one observation in the data set. In this figure, the input vector consist of two features X_1 and X_2. The rightmost layer is known as the output layer, and this is where the final result of the ANN is presented. Depending on the problem, the resulting output, $\hat{Y}$ may be a classification label or, as in this thesis, a predicted value. The hidden layer, on the other hand, can be seen as (1) an output layer for another hidden layer or, as in Figure 2.2, the starting input layer, and (2) an input layer to another hidden layer or, as in Figure 2.2, directly to the output layer. The three types of layers are linked together by sets of weight vectors that represents the inter-neuron connections. The network illustrated in Figure 2.2 has two sets of weights: the first set, W^1, maps the input to the hidden layer and the second, W^2 maps the hidden layer to the output layer. Here, W^1 consists of six weight vectors, $w_{i,j}$ where i refers to the input feature and j a neuron in the hidden layer while there are only three in W^2, $w_{j,k}$ where k is the output.

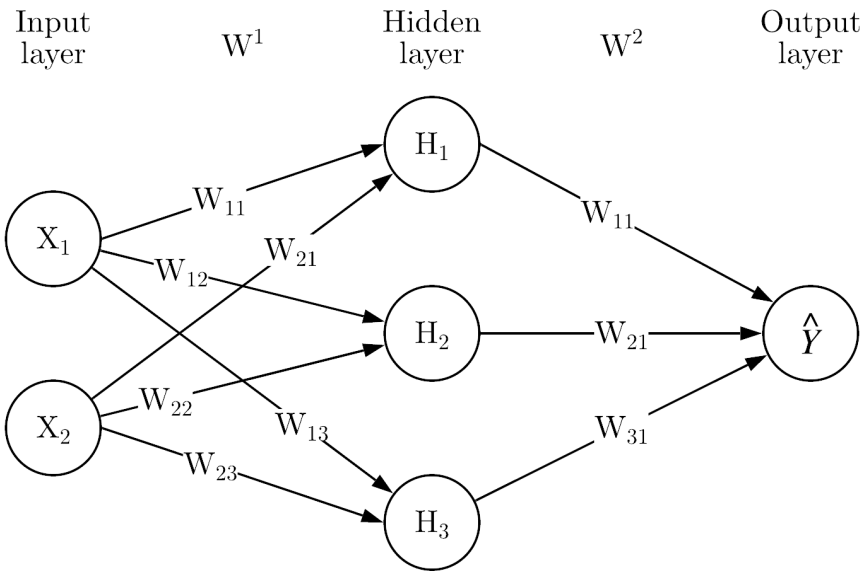

Figure 2.2: A one-layered feed-forward artifical neural network.

The first step in the learning algorithm of ANNs, Algorithm 2, is to randomly initialize all weight vectors, $w_{i,j}$ and $w_{j,k}$, to a small value. Next, for each observation, $(\mathbf{x}, y)$ in the training set, $(\mathbf{X}, \mathbf{Y})$, there are three necessary steps: (1) feed-forward information, (2) compute the prediction error, and (3) back-propagate the error to update the weights. The input values are fed forward through the network from the input layer, via the hidden layer to the output layer. Formally, this is done by first summing the result from applying an activation function, $A(\mathbf{x}, w_{i,j})$, to each incoming input value and its connecting weight vector. The result, h_j where j refers to a specific neuron in the hidden layer, is then used as input value to the output layer. There are several activation functions available, however, the most commonly used are hyberbolic tangent (tanh), logistic (sigmoid) and rectified linear unit (ReLU). The predicted value for the current training instance, $\hat{y}$, is then computed similarly by applying an output function, $O(h_j, w_{j,k})$. The next step is then

to compute the prediction error, ϵ_n, by applying an error or loss function, $L(\hat{y}, y)$, to the predicted output and the actual value. As with activation and output functions, there are several alternatives available depending on the problem at hand. Given that the ANN will be used to predict future resource needs in this thesis, the problem can be defined as a regression problem, hence, the mean squared error (MSE) is a suitable loss function to use. To update the weights, the error is then propagated back throughout the network using stochastic gradient descent (SGD). Once all instances in the training set has passed through the network, the performance may be evaluated. A typical evaluation process is to compare the total loss, i.e. the sum of each prediction error ϵ, with a pre-defined stopping criterion, δ. If the performance is satisfying, the training may be concluded, otherwise the training is repeated from step two, as seen in Algorithm 2. [29]

Algorithm 2 Feed-forward ANN with SGD

Step 1: Initialize weight vectors
For each: $w_{i,j}$ and $w_{j,k}$
1: $w_{i,j} \leftarrow [0, 1]$
2: $w_{j,k} \leftarrow [0, 1]$
End for:
For each $(\mathbf{x}, y) \in (\mathbf{X}, \mathbf{Y})$
Step 2: Feed-forward information
3: $h_j \leftarrow \sum A(\mathbf{x}, w_{i,j})$
4: $\hat{y} \leftarrow \sum O(h_j, w_{j,k})$
Step 3: Compute error
5: $\epsilon_n \leftarrow L(\hat{y}, y)$
Step 4: Back-propagate error and update weights
6: $w_{j,k} \leftarrow w_{j,k} - \eta \nabla w_{j,k}$
7: $w_{i,j} \leftarrow w_{i,j} - \eta \nabla w_{i,j}$
End for
Step 5 Evaluate performance
8: $\epsilon \leftarrow \sum \epsilon_n$
9: **if** $\epsilon < \delta$ **then**
10: Finished training
11: **else**
12: Go to step 2
13: **end if**

The feed-forward of computed results and backpropagation of error are two fundamental parts of any ANN. However, the actual design of the network, the type of inputs and what the model outputs are dependent on what problem it is trying to solve. This fact has lead to the development of ANNs specialized in solving a particular type of problem. The problem at hand in this thesis is to predict future resource needs based on historical data. Therefore, the specific type of ANN used is a recurrent neural network (RNN) presented in Section 2.3.3.

2.3.3 Recurrent Neural Networks

RNNs are well-suited for predicting future resource needs based on historical data. In general, this type of ANN is designed to solve problems related to sequential data such as speech recognition, word translation or time series. The reason for this is that RNNs use a state vector to keep track of what previously happened in the sequence and use that when trying to predict the new state. In the context of this thesis, the state vector would keep track of how the resource usage for each container in the cluster changes over time.

The fundamental concept of ANNs described in Section 2.3.2 still applies for RNNs. That is, the network is feeding information forward throughout the network until it reaches the output layer where the prediction finally is made. Then, the error of the prediction is calculated and propagated back to update each weight vector in the network, using SGD. What makes RNNs different from a general ANN, displayed in Figure 2.2, is that knowledge gained from previous predictions is used to predicting future outcomes. This knowledge is stored in weight vectors that are connecting a neuron in the hidden layer at time $t-1$ with a neuron in the hidden layer at time t. This difference is illustrated in Figure 2.3 by the different types of weight vectors: $W_{X,H}$ connects the input layer X and the hidden layer H, $W_{H,Y}$ H and the output layer Y, and finally $W_{H,H}$. When the RNN makes its prediction, $\hat{Y}_t$, at time t, the input values, X_t, is fed forward to the hidden layers, H_t. However, given that the network already did the same procedure for time $t-1$, the knowledge stored in $W_{H,H}$ is also fed to the current hidden layer H_T. That is, when applying the activation function in step 2 from Algorithm 2, it has three inputs instead of two: $A(X_t, W_{X,H}, W_{H,H})$.

There is, however, a significant drawback with RNNs caused during the learning process. When the error is propagated back, it gets smaller each step through the network until it more or less vanishes due to the many consecutive derivations needed. Consequently, weights at the beginning of the network barely get updated, leaving a large portion of the network untrained. This is known as the problem of vanishing gradients [28], [30]. Vanishing gradients can, however, be mitigated by employing long short-term memory, LSTM, cells [31]. Therefore, to avoid the risk of vanishing gradients, this thesis will utilize LSTM cells in the prediction model.

On a high level, a LSTM has the same repeating structure as a RNN. The difference is what is going on inside the modules [32], [33]. The LSTM cell can be described as having four different gates that control how much of the new information the cell should store and how much of the old information the cell should keep. The output of each cell is then passed forward to the next cell, and the same process starts over. The problem of vanishing gradients is dealt with by letting error gradients flow back unchanged.

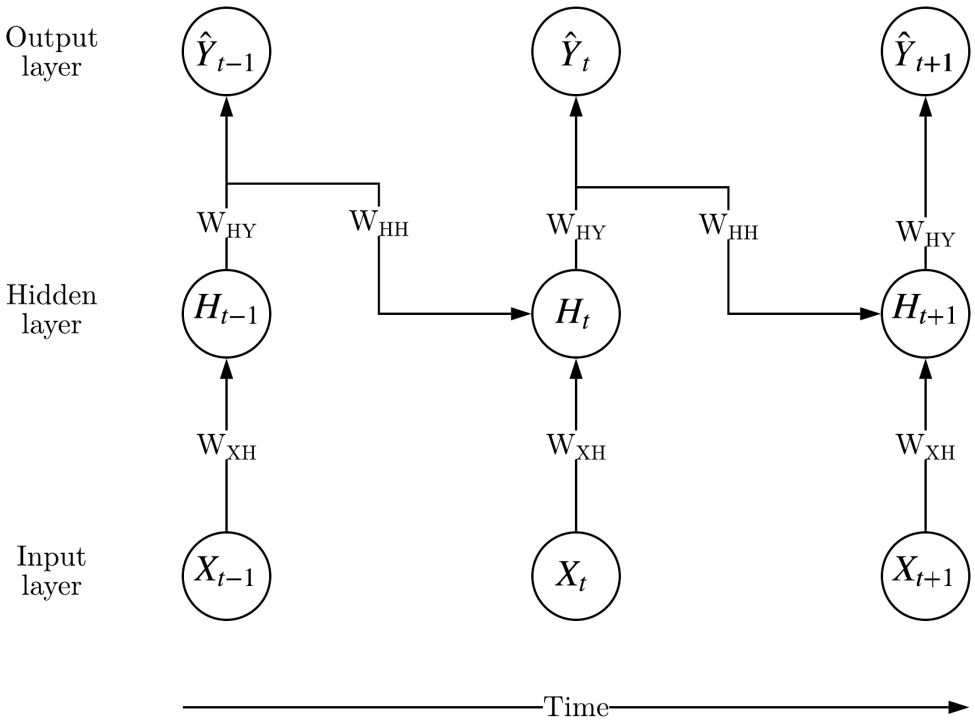

Figure 2.3: A one-layered feed-forward recurrent neural network.

3
Related Work

The purpose of this section is to position this thesis in relation to previous research. In general, much of the related research considers VMs rather than containers, which is the main focus of this thesis. However, as discussed in Section 2.2.3, both VMs and containers can be used to virtualize applications, and both utilize the same types of resources, e.g., CPU and memory. Hence, methods used to predict e.g., the memory needs of a VM is still applicable when considering containers. Due to the multifaceted problem, this section will be divided into two parts. First, related work on the topic of container profiling will be presented before moving on to predicting future resource needs.

3.1 Container Profiling

Much research on virtualized application profiles focuses on the relationship between performance and resource usage. Do et al. [34] claim that an application typically displays a specific behavior and resource usage pattern over time. Furthermore, they explain how applications share resources in a cloud environment, which means that applications located on the same host may affect each other. By varying the workload, the relationship between an application's performance and the utilization of the underlying system resources could be determined using canonical correlation analysis (CCA). The output from the analysis, one vector representing system resource usage, one vector representing application performance, and the correlation between the two, was used as a profile for a specific application. While Do et al. used CCA to identify a linear relationship between performance and resource allocation, Kundu et al. [35] rejected conventional linear regression models. Their study confirmed that the non-linear dependence of performance on resource levels and the complex influence of contention could not be represented using such simple functions. Instead, in a later study [36], the authors used machine learning models to model the relationship between resource allocation to virtual machines and their performance. They determined the near-optimal resource configuration for a VM using file input/output (I/O) per second and requests per second as a performance metric to evaluate both ANNs and support vector machines (SVMs). While the previously mentioned studies aimed to profile applications in virtualized or cloud -environments, Wood et al. [37] investigated how virtualization overheads affect an application's requirements. That is, how the requirements should be adjusted before transferring an application to a virtualized environment such as the cloud. First, the authors used a trace-based method to profile an application running natively and

virtualized. This method assumes that historical resource usage patterns are representative of future application behavior. Next, they used stepwise linear regression to define a model that translates a resource usage profile from one environment to the other.

While both Do et al. and Kundu et al. offer sophisticated solutions to model or profile virtualized applications such as containers in terms of the relationship between performance and resource utilization, this thesis will not consider this relationship. In contrast to the mentioned studies, this thesis aims to profile live containers using data of their actual resource usage rather than data generated from synthetic benchmarks. Additionally, this thesis will use a trace-based method to profile containers similar to what Wood et al. described. However, the traces will be collected from live containers running in the cloud. Consequently, there is no need to transform the trace between a native and a virtual environment. Moreover, since a container's resource usage trace is collected from a live cluster, any effects from other containers on the same node will be captured in the profile similar to what Do et al. described.

3.2 Resource Predictions

Huang et al. [38] focus on providing business value to customers that have reserved resources from cloud providers rather than acquiring resources on-demand. The added value is derived from the fact that it is cheaper to reserve resources over time, especially if the reservation is closely matched to the actual need. Since the need for computational resources is dynamic, the authors acknowledge the fact that the reservation has to be adjusted to fit current resource needs to reap the most benefit. To do so, they suggest a time series-based prediction model that considers both historical data and the current state to predict future needs. Similar to Huang et al., this thesis focuses on the user, or customer, of CC services as well as predicting future resource needs. Furthermore, since a reservation of resources can be translated into a cluster, their goal of matching a reservation to the actual need is merely a translation of matching a cluster size to its actual need. However, since Huang et al. make predictions on an aggregated level, information about individual jobs or containers are lost, making the problem of traceability persist.

While Kundu et al. [35] rejected standard regression analysis for their problem, Islam et al. [39] showed success with linear regression when predicting future resource usage patterns. However, when the authors compared the performance of their linear regression model against an ANN, the latter model outperformed the former. The outcome was consistent regardless of the sliding window technique was applied or not. However, the performance was improved for both models when used. The general idea behind the sliding window technique is to map an input sequence to a single output, much like an RNN described in Section 2.3.3. Similar to Huang et al., Islam et al. do not consider individual containers, they only make predictions on an aggregated level. Despite the problem of traceability, their method showed promising results with a MAPE of 0.195 for their default ANN and 0.1772 when using a window size of eight. Therefore, this thesis will build upon the idea of using machine learning techniques such as ANNs to predict future resource needs. In

contrast to Islam et al., however, we will consider RNNs due to them being tailored to solve problems such as ours with sequential data. Furthermore, due to the problem with vanishing gradients, as discussed in Section 2.3.3, our RNN will consist of LSTM cells in an attempt to mitigate this problem. To solve the problem of traceability and still be able to predict the total resource need of the cluster, each container will be considered individually. Then, by aggregating the predictions, the total cluster size can be estimated.

4
Methods

Due to the objective, Section 1.1, and research questions, Section 1.2, the study was conducted as a solution-seekinig case study. Although inherent problems such as generalizability exist, a case study design is typically well-suited for research settings such as ours [40]–[42]. Runeson and Höst [40] argue that, in general, a case study can be broken down into five major process steps, as seen in Figure 4.1. While the complete thesis serves as the reporting step, the rest of this section aims to provide a clear description of the other process steps.

4.1 Case Study Design

The first step of the case study design was to define our objective, or, in other words, what problem or phenomena related to software engineering (SE) we wanted to investigate.

From a global perspective, one of the 17 sustainable development goals published by the United Nations [43] is to ensure sustainable consumption and production patterns. To a large extent, this refers to pursuing a more resource efficient life cycle for all products and services. From a SE perspective, emerging technologies and concepts such as containers, CC, and IaaS, provides a potential solution by offering computational resources on-demand with (an almost) just-in-time delivery. However, it is still up to the customer to define the required amount of resources needed for their applications to run efficiently in the cloud. From our own experience, this is often done in an ad-hoc manner which has the potential of leading to an overallocation of resources and waste. Since previous research and Ericsson supported our experience, the overarching objective became to investigate how this problem could be mitigated, if not solved. Furthermore, when surveying previous research, it became clear that most research on the topic is focused on VMs and

Figure 4.1: The different steps of the research process.

the cloud provider side. The identified gap in research focused on containers and the customer side helped us narrow down the objective and to define our research questions, seen in Sections 1.1 and 1.2 as well as Table 4.1.

Once the objective and research questions were defined, the next step was to define the case and where the necessary data could be gathered. Based on Yin's definitions of case study design [44], both Sjöberg et al. and Runeson and Höst propose two alternatives: a single case or multiple cases [40], [41]. Furthermore, both alternatives may involve a single unit (holistic) or multiple units (embedded) of analysis. In our context, and depending on the level of abstraction, both the case and the unit of analysis may be defined in different ways. However, given that the objective and research questions in this thesis are focused on the actual cluster and its containers, it was obvious to let the cluster be defined as the case and use containers as our unit of analysis. Furthermore, to ensure access to enough quality data, we decided to pursue an industrial partner to collaborate with. This decision was also aligned with our ambition to work in a real-life environment using real data rather than simulated data. Our requirements when searching for a suitable partner were simple given that our primary focus was on the cluster and its containers rather than the actual organization. The organization had to have a live cluster with a large number of containers deployed to it. Furthermore, there had to be high activity in the cluster, e.g., containers experiencing various amount of traffic, containers being deployed or updated, etc. Ericsson fit our requirements, and after a meeting, we could conclude that our objective and research questions were aligned with their interests.

A typical outcome of solution-seeking research, such as ours, is a system or tool [45]. In our case, the outcome is an automatic tool for profiling and labeling containers based on their behavior and resource usage as well as predictions of future resource needs. In light of this, it was inconceivable to have a fixed design process with all parameters determined at the start. Instead, a flexible design process was employed which is characteristic of a case study [40]. The benefit of employing a flexible design process is the possibility of iterating between steps, which is typically the case when developing software artifacts such as a system or tool. In our case, as displayed in Figure 4.1, the majority of iteration was done between preparing the data collection and collecting the evidence. The reason for this is that the sub-modules of the complete system were implemented and tested first individually, then in combination. Additionally, the system was also deployed in three stages: first locally, then in a test environment at Ericsson, and finally in a production environment hosted by Ericsson. Each stage resulted in some changes, either in the data models, the algorithms, or the interfaces between sub-modules. A detailed description of the system and its sub-modules is found in Section 5.

4.2 Data Collection

Several important design decisions concerning data-structures were considered during the data collection preparation. The first question was what type of data we needed. A typical division between studies is whether they use a qualitative or quantitative method where the discrepancy between the two lies in what kind of data

is being collected and analyzed [41]. In general, a case study is typically regarded as a qualitative method, while experiments often serve as an example of the opposite [40]. However, given the more flexible design, in contrast to experiments, case studies may be conducted using either quantitative or qualitative data, or even mix between the two [40], [42]. The next questions were how the data should be gathered and from what source(s). Lethbridge et al. [46] propose three types of techniques contrasted by the degree of human involvement. The first degree includes methods where the researcher is in direct contact with the subjects such as interviews or brainstorming, the second degree is indirect methods where the researcher collects raw data without interacting with the subject, and finally, the third degree refers to the collection of archived data [40]. In the end, the answers to these questions are dependent on the objective of the research, thus, the goal-question-metric approach (GQM) was applied [47].

Goal	Improve the resource allocation of a container-based cluster from a developer's perspective
Question	**What is the current resource allocation?**
Metrics	Allocated memory [MB]
	Allocated CPU [millicore]
	Allocated nodes [# of physical machines allocated]
Question	**What is the current resource usage?**
Metrics	Used memory [MB]
	Used CPU [millicore]
	Needed nodes [Minimum # of physical machines needed]
Question	**How many resources are currently being wasted?**
Metrics	Wasted memory [MB]
	Wasted CPU [millicore]
	Wasted nodes [# of physical machines wasted]
Question	**How many resources will the cluster require in the future?**
Metrics	Predicted memory need [MB]
	Predicted CPU need [millicore]
	Predicted nodes needed [# of physical machines needed]

Table 4.1: The goal-question-metric approach was employed to guide the data collection.

The metrics defined using the GQM approach guided our decision making concerning the questions raised in the previous paragraph. As Table 4.1 shows, all data needed to answer the questions were quantitative. Furthermore, there was no need to apply first degree techniques to gather the data; instead, a mix between second and third-degree techniques was used. Information about individual container's resource usage is readily available by querying the container orchestrator's metrics API. Therefore, a data collector that automatically queried the API and stored the information in a database was implemented. This resulted in increased control and the possibility to align the data collection with our overarching purpose of the case

study. A detailed description of the data collector and the data structures is found in Section 5.1.

4.3 Data Analysis

The choice of analysis method is highly dependent on whether the data collected is of a qualitative or quantitative nature. While qualitative data typically is analyzed using categorization and sorting, quantitative data, such as ours, often includes analysis of descriptive statistics, development of predictive models, and hypothesis testing [40]. Furthermore, in case studies where a new method, tool, or system is developed and analyzed, a comparison of the result against a company baseline is often done [42]. Additionally, to avoid potential bias, the comparison can also be extended to include a sister project [41], [42]. Again, we used the GQM-approach, Table 4.1, to guide our decision making. To analyze if our implemented solution can help an organization improve its resource allocation, comparing the current resource allocation against the actual resource usage was the first step. However, instead of using Ericsson's current resource allocation as a baseline to compare against, a hypothetical company baseline was constructed. The reason for doing so is solely to ensure some degree of confidentiality towards Ericsson. The hypothetical company baseline is created based on the following assumptions: (1) the company is well aware of its cluster's resource requirements, (2) the company employ a static deployment strategy, i.e., they do not dynamically adjust the size of the cluster, and (3) the fixed number of nodes is set to the upper bound of the actual resource need plus two additional nodes. To illustrate with an example: if the maximum number of nodes needed over the time period is 10, the hypothetical company baseline is going to be 12. Additionally, to analyze the predictions using our compiled container profiles, Section 5.3, descriptive statistics such as mean absolute error (MAE), mean absolute percentage error (MAPE), and R^2 were applied. By including MAPE as a performance metric we were also able to compare our predictions against the results presented by Islam et al. [39].

5
System Structure

By reviewing the research questions in Section 1.2 and defined goals in Table 4.1, it is paramount that the following methods are provisioned in some form or another. Initially, a data acquisition process is required, one which may communicate with the cluster orchestrator and gather resource usage for all containers existing in the cluster. The process input is raw data produced by the orchestrator, that data is parsed to a format that is aligned to fulfill the various definitions noted in Table 4.1. In brief, the arrangement must allow data to be used in a survey of resource history utilization for a given container, and then group it by its id and revision. Furthermore, that format should also be able to transmute into a tuple-based structure which may be applied to train a prediction model.

At its core, three separate sub-modules are built to fulfill the four different requirements noted above: Data collector - fetches resource history from the cluster orchestrator, Machine labeler - classify each container in relative terms to all other containers that live in the cluster, Predictor - uses the containers' profiles to predict future resource requirements.

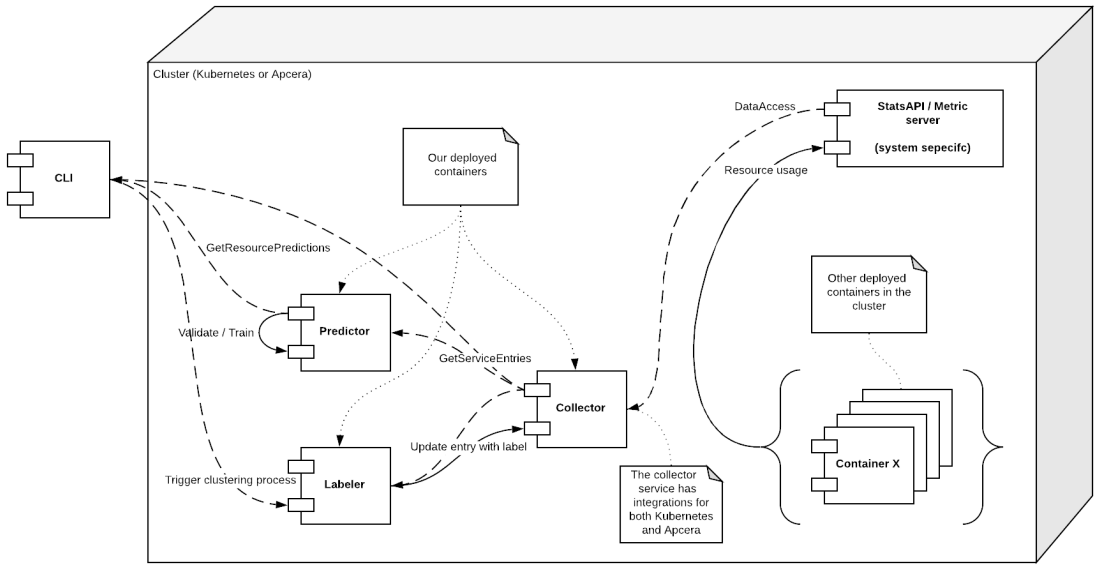

Figure 5.1: Abstract system overview.

All parts of the system reaching from API, data modeling, and data analytics are implemented using python 3.7. Each module is then containerized and managed by Apcera. More detailed information is presented regarding each module in the

following sections.

Figure 5.1 displays how the services finally intercommunicate to collect, update, and finally produce data. Firstly, the collector service queries resource information from the internal metrics API. Secondly, the labeler consumes that data, clustering each entry for each container, and finally updates the record in the collector. Next, the predictor consumes the updated information and trains an RNN internally for each container that has been registered in the cluster. Finally, the created container-models are used to produce future resource predictions, which in turn are aggregated to estimate the consumption of the whole cluster.

5.1 Data Collection and Container Profiling

To consume and even make use of the data which the container orchestrator provides, one must first integrate with its API. From the get-go an orchestrator does not commonly distinguish between container revisions (or versions), a third party must implement that type of functionality. Furthermore, the raw data needs to accommodate the fundamental needs of both the labeling and prediction -process.

The collector service is constructed to gather data from Apcera's or Kubernetes m Metrics API and restructure it to fit the database model shown in Figure 5.2. The diagram displays a structure that allows profiles to be tracked for each container and across different revisions. Furthermore, the architecture enables entries to be joined by container and either clustered or transformed into a structure that mirrors the tuple interface outlined in Section 5.1. In general, all subsequent modules may consume the processed data on their own accord using the standardized interface shown in Appendix B.1 which adhear to the proposed database model.

Apcera supports different time resolutions, which makes it possible to take averages in minutes, hours, days, months, or years over a specific period. In short, one can select a bin size which will average the collected metric-entries (received every 10th minute) over the chosen resolution, then use that configuration to fill in each step over the selected time-span. The collector module uses the average resource consumptions over each day as it is simply not interesting to drill down further. The purpose is to determine the behavior over time and possibly resize our cluster accordingly. The cluster will at most be resized once every day, and therefore, it does not make sense to make use of lower resolutions.

As displayed in Figure 5.1, the implemented sub-module lives in the cluster, consumes data from the orchestrator's internal API using the average resource consumption for each day over six months, and finally exposes it to the labeler described in Section 5.2 and the predictor in Section 5.3.

5.2 Container Labeling

Once data is being collected from the orchestrator in a valid format that adheres to the standard mentioned above and is exposed using the interface shown in Appendix B.1, it is time to determine the relative label for each container and their entries. Remember that the labeling process is needed to create smarter deployment strate-

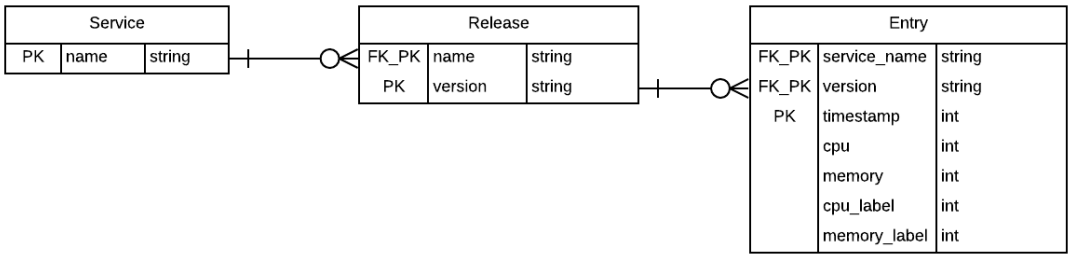

Figure 5.2: Database schema

gies, as mentioned in Section 1.2. The general idea is to use reference points that represent extreme values of each resource type and then find the closest reference point for all entries with regards to each cluster's centroid. The labels may allow an orchestrator to deploy containers with similarly high-value labels on separate nodes in the cluster. More specifically, a model must be trained for each type of resource within each timestamp grouping, hence, a specific service's label may change over time. Nevertheless, the method would counter the problematic issue where services consuming similar resources may starve each other if deployed on the same node in the cluster, as mentioned in Section 1.

Container	Version	Time stamp	CPU usage	Memory usage	CPU label	Memory label	Is labeled
1	1.0	15512	0.7	1,000	3	3	True
2	1.0	15512	0.1	1,000	1	3	True
3	1.0	15512	0.5	200	2	1	True

Table 5.1: Labeling example of three different containers.

The labels are stored as integers, ranging from 0-3, where the value represents the degree of resource usage, i.e., from low to high in ascending order. When a service is registered by the data collector, the labels are defaulted to a value of 0 and the flag *Is_labeled* is set to false. Once a time stamp has been clustered, each service is asssigned its label and the flag is updated to true. Table 5.1, illustrates an example of the information stored in the database. The labels shows that container_1 has been labeled as a heavy cpu and heavy memory, container_2 as low cpu and heavy memory, and container_3 as medium cpu and low memory. As both container_1 and container_2 have been assigned a heavy memory label, a container affinity command may be defined using the container orchestrator to have the containers be deployed on different nodes in the cluster.

The final implementation is constructed as the description above entails and deployed in the cluster. In short, the labeler determines every relative label using K-means clustering as described in Section 2.3.1 and the quality of its groups are determined using the average of all silhouette scores.

Where the silhouette score is a measurement that indicates how much the

produced groupings overlap. The best value is 1, which presents a perfect separation, a value of 0 indicates overlapping clusters, and value of -1 indicates that a sample has been assigned to the wrong group [48].

5.3 Predictor

The predictor is the last module in need of construction to fulfill the final aspects of the defined research questions and GQM. In short, predicting future resource usage for individual containers and then use that result to infer predictions on the entire cluster. As outlined in Section 3, RNN and LSTM is recommended for trace-based analysis and future estimations. Therefore, this thesis bases the fundamental prediction model on those findings. Moreover, the prediction process makes use of the processed data described in Sections 5.1 and 5.2.

When making predictions, the data stored according to the database schema in Figure 5.2 is transformed into a dataset with a data structure as defined in Equation 5.1. container$_i$ represents a container with id $= i$ where i is an integer $\in [1, N]$, and N being the total number of unique containers recorded in the database. Timestamp refers to a specific point in time expressed in UNIX time. However, when used in the following equations, time, t, is an integer that represents a day in a finite time series. CPU and memory refers to container$_i$'s actual resource usage at the specific time stamp while *CPU_label* and *memory_label* are the labels assigned to the container as described in Section 5.2.

$$\langle \text{container}_i, \text{timestamp, CPU, memory, CPU\_label, memory\_label} \rangle \tag{5.1}$$

By using the data structure in Equation 5.1, the problems of aggregating and filtering the data with respect to container ids and/or timestamps are easily solved. The resource usage of a specific container, c_i, at time $= t$ may be defined according to Equation 5.2.

$$\text{r}_{c_i,t} = \langle \text{CPU, memory, cpu\_label, memory\_label} \rangle \tag{5.2}$$

From Equation 5.2, we can easily obtain the amount of memory or CPU that container, c_i used at time $= t$. By including all available timestamps for a given container, we get a finite time series that make up the container's profile and it is these profiles that the model aims to learn.

5.3.1 Resource Aggregation

To be able to discern the total resource utilization of the cluster using the method outlined in this thesis, one must first determine the usage of its components. Consequently, the overall resource usage for a cluster is summed from each container's resource utilization running inside it. This order of summation is what allows organizations to keep track of allocation deviations or origins, and is the distinct difference that separates this thesis from previous research, as noted in Section 1. In addition, the cluster's connected nodes will determine its total allocation, in other words, its

available resources. The following equations specify how the aggregation process functions logically.

The resource utilization of a specific container at a given point in time, $r_{c_i,t}$ is easily obtained as shown in Equation 5.1 and Equation 5.2. The total resource utilization of the cluster at time = t, R_t, is then found by taking the sum of each individual container's resource utilization as seen in Equation 5.3.

$$R_t = \sum_{i=1}^{N} r_{c_i,t} \tag{5.3}$$

5.3.2 Prediction

The prediction model will examine the collected profile of a specific container and try to predict the future workload of that specific container. The expected load may then be used to discover detailed resource requirements. More specifically by utilizing RNN (LTSM) described in Section 2.3.3, the process will create a model for each container and predict each resource attribute accordingly by referencing at container$_i$'s profile. Finally, aggregating all allocations to a single group.

By aggregating the predicted resource utilization of individual containers at time = t, $\hat{r_{c_i,t}}$, the same way as shown in Equation 5.3, we get the predicted total resource utiliziation of the cluster, $\hat{R}_t$ as seen in Equation 5.4.

$$\hat{R}_t = \sum_{i=1}^{N} \hat{r_{c_i,t}} \tag{5.4}$$

The objective will be to minimize the error in predicted resource utilization compared to the recorded actual utilization. However, as the predictions are made on a container level, the error is first calculated for each container before being aggregated on a cluster level. That is, the error of predicting the resource utilization of a specific container, ϵ_{c_i}, is defined as seen in Equation 5.5

$$\epsilon_{c_i} = \frac{1}{K} \sum_{t=1}^{K} |r_{c_i,t} - \hat{r_{c_i,t}}| \tag{5.5}$$

As Equation 5.2 defined, $r_{c_i,t}$ is the actual resource usage of container c_i at time = t and $\hat{r_{c_i,t}}$ is the predicted resource usage. The sum is taken over a finite time series where K represents the last day in said series.

This definition of an error is more commonly referred to as the mean absolute error (MAE). The value of the error represents how far off the prediction was on average over the time series. Note it is not possible to discern whether or not the predictions were under- or overestimations, only the magnitude of the error may be determined. Overall, the lower the error, the better.

The error on a cluster level, ϵ, is obtained by computing the error on an aggregated level, Equation 5.6.

$$\epsilon = \frac{1}{K} \sum_{t=1}^{K} |R_t - \hat{R}_t| \tag{5.6}$$

Additionally, a separate model configuration is constructed where the model will be able to make use of previous prediction errors to adjust its current guess, using a trivial technique, similar to a moving average, we call "sliding window bias". Each new prediction will be adjusted by $prediction * (1 + averageAdjustment)$, where the average adjustment is determined by previous errors (window size). The purpose of this routine is to bias the model using consistent estimation errors.

5.3.3 Execution

As described in this Section, (1) raw data is collected and processed to workable profiles, (2) each profile-entry is then labeled, next (3) the processed information is passed on to the predictor, which finally starts to train its container-models.

To adhere to Ericsson's policies, we first had to develop the system, deploy it in a test environment, prove that it is working as expected, and then deploy it to the production environment. At its core, both settings are configured in the same manner, except for their respective size.

Six months (182 days) worth of data is collected from the live environment and two months from the test environment by utilizing the defined process flow. The span is limited since the cluster is used less and less since from the time of execution and 182 days back, we wanted to analyze a large cluster that may reflect a real working environment. Only 60 days were collected from the test environment, depending on the fact that the environment is set up during our development phase and is therefore limited. The data is split up into two sets, as seen in Figure 5.3 and 5.4 representing containers in the live environment. The first section includes the training entries which are fed to train each container-model and the later to measure respective performance.

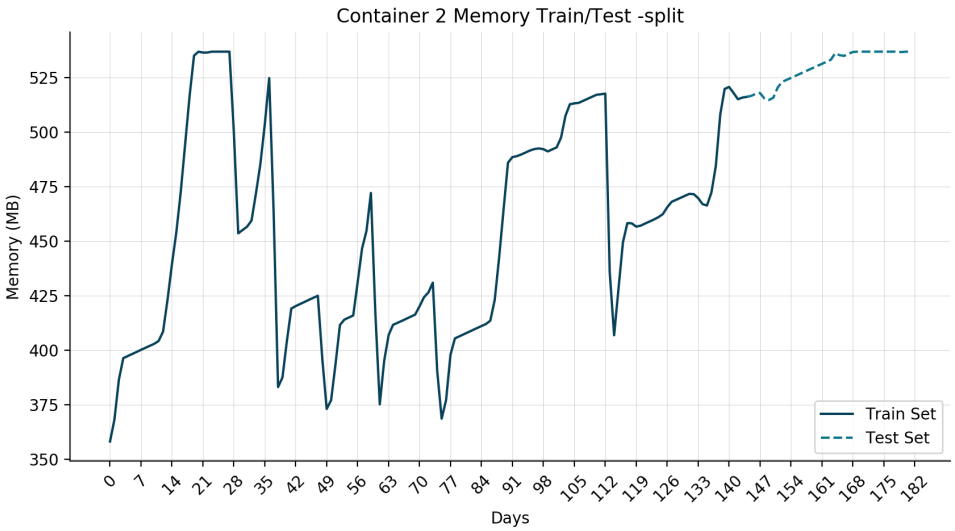

Figure 5.3: Memory allocation profile of container 2 in live cluster.

The train/test-split is set at 0.8, hence, a container-set contains 145 entries in its training set and 37 in its test set. Each container-model uses an individual

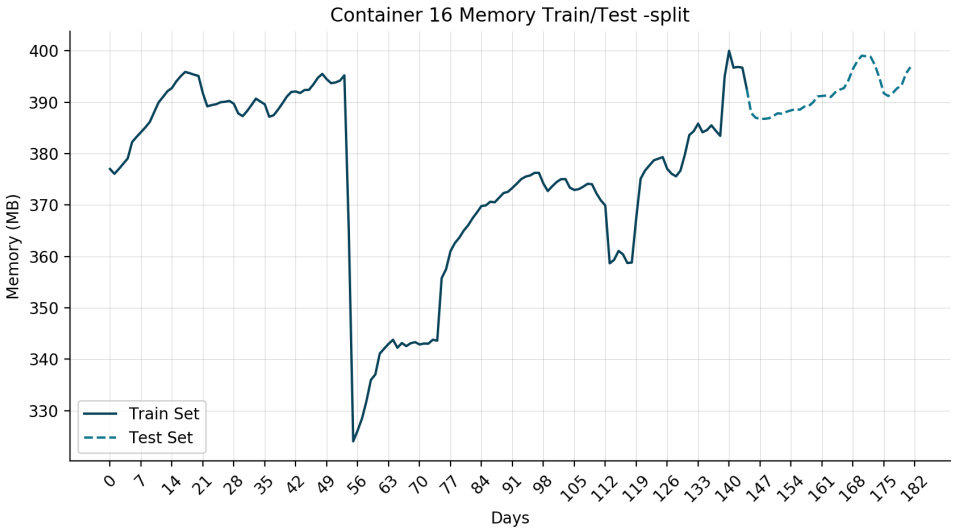

Figure 5.4: Memory allocation profile of container 16 in live cluster.

normalizer to fit and transform its entries accordingly between zero and one, the reason being that RNNs perform better using normalized values, one does not want large numbers to bias the model's learning process.

$$(\text{CPU, memory, day, month, memory}_{\text{label}}, \text{CPU}_{\text{label}}) \tag{5.7}$$

The shape of each container set is currently of shape (145, 6), 145 entries with six features each, shown in the Equation 5.7. Each model is allowed to make use of previous entries when predicting the current by including the previous three entries for every entry. The training sets shape becomes (142, 3, 6) likewise the test set shape (34, 3, 6). The shape indicates that it exists 142 of samples with three sequences and each sequence having six features. Subsequently, the model is compiled, trained, and finally evaluated using the test set. The same sequence of actions was also employed for the test environment, but adjusted to fit its own prameters.

6
Results

This chapter presents the result of the comprised system, its predicted allocations, and errors. There are three sections in total, the first present data from the test environment and the following features data obtained from the production environment. Results from both environments are inccluded as the two exhibit different behaviours that the prediction models were forced to learn. Note that the final model predictions include only memory as the other qualities became insignificant in contrast, with regards to utilization and cost. The last section presents cost estimations and cluster adjustment proposals that are drawn from the prediction models.

6.1 Test Environment

This section presents the results from our experimental evaulation in a test environment hosted by Ericsson. First, the outcome of the labeling process is presented followed by a presentation of the results from predicting future resource needs based on container profiles.

6.1.1 Experimental Evaluation of Labeling

Figure 6.1 presents how much memory containers have allocated in the cluster at a specific point in time. Squares note containers consuming low memory, crosses indicated average memory consumption, and dots indicate a high value. Fundamentally, these groupings lay the foundation of what type of label should be applied to which container at a point in time, so that the orchestrator may perform more intelligent deployments. The cluster's average silhouette score of 0.6172 indicates that the identified groups are separated reasonably since a score near 0 indicates overlapping clusters. Note that the presented Figure 6.1 is one out of the 60 days collected from the test cluster. The entire average silhouette score is 0.6140, which constitutes the same resolution.

6.1.2 Experimental Evaluation of Predictions

The results presented in Figure 6.2, 6.3, and 6.4 show how the algorithm behaves during the test phase, all produced results may be found in Appendix B. The figures display days along the x-axis and amount of allocated memory on the y-axis. The solid lines display actual memory consumption, the dotted line indicates model

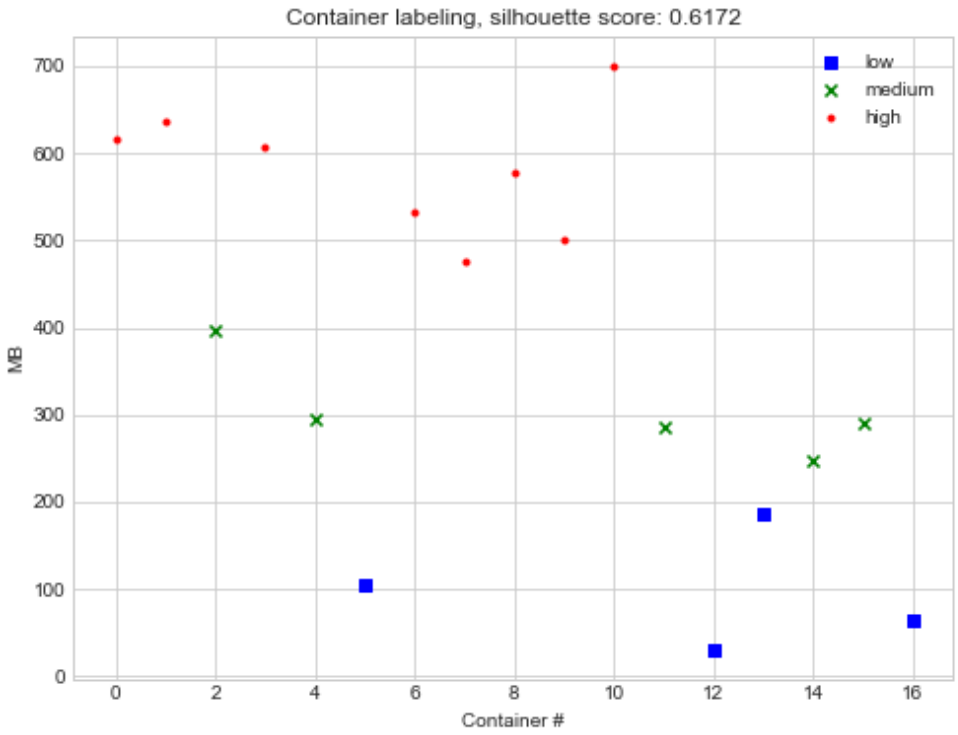

Figure 6.1: Container labeling result at a specific point in time.

predictions, and the dashed-dotted lines present predictions in conjunction with a sliding bias window of two days.

Figures 6.2 and 6.3 displays that the models have captured a repetitive behavior and that the sliding window technique does not impose any improvements. On the contrary, the bias technique is less effective. Furthermore, Figure 6.4 promotes a case where the model does not capture the underlying behavior in an as effective manner, in contrast to the other containers observed in Table 6.1. The result of container 10 presents an example where it is evident that, if the prediction fails, in general, the bias adjustment is likely to be even worse. In short, an adjustment from a model that does not manage to capture the containers underlying profile produce even worse effects.

Table 6.1 displays key metrics for each container running in the cluster and how the models perform in contrast to the technique using sliding bias. From the left: container - numeric identifier, mean absolute error (MAE) - absolute value indicating the average magnitude of each miss, mean absolute percentage error (MAPE) - in effect MAE turned into a percentage value, r^2 - statistical measure that indicates how much of the variance in memory can be explained by the model. The following three columns measure the same metrics in sequence but for the model predictions using sliding bias with window size = 2. In general, Table 6.1 indicates that most models determine specific container behavior, they predict accordingly with a low MAP and produce firm r^2 scores. Furthermore, both Figure 6.5 and Table 6.2 entails

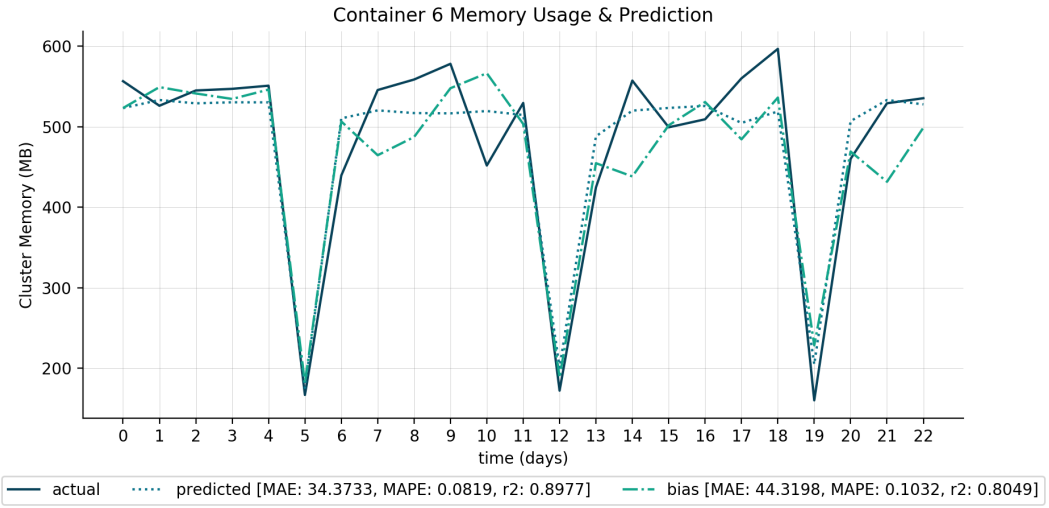

Figure 6.2: Predictions of memory usage for container 6.

the same result, that is, the models produce low magnitude errors (MAE, MAPE), and they can account for much of the variance.

Container	$MAE(MB)$	$MAPE$	r^2	$MAE(MB)_w$	$MAPE_w$	r^2_w
0	22.1532	0.0378	0.9530	20.7665	0.0353	0.9590
1	35.7899	0.0631	0.8883	54.3469	0.0965	0.8036
2	13.6265	0.0356	0.9684	17.4905	0.0450	0.9469
3	36.4033	0.0644	0.7935	45.0337	0.0851	0.7428
4	11.2957	0.0469	0.9629	14.1867	0.0575	0.9346
5	6.3901	0.0678	0.9228	8.6742	0.0918	0.8486
6	34.3733	0.0819	0.8977	44.3198	0.1032	0.8049
7	24.2633	0.0621	0.9173	33.507	0.0859	0.8500
8	23.2149	0.0506	0.9504	29.100	0.0643	0.9266
9	20.0135	0.0562	0.9497	24.9583	0.0697	0.9182
10	61.4608	0.1438	0.6729	100.8524	0.2221	-0.3310
11	22.1729	0.1050	0.6149	34.5312	0.1615	0.4372
12	0.9867	0.0401	0.9660	1.3071	0.0508	0.9502
13	6.4375	0.0387	0.9670	7.7579	0.0480	0.9465
14	7.8983	0.0342	0.9717	7.9762	0.0362	0.9651
15	10.5700	0.0431	0.9558	12.3051	0.0496	0.9470
16	2.4900	0.0451	0.9627	2.7861	0.0485	0.9450

Table 6.1: KPI metrics for each container's memory prediction, both prediction and bias, in test environment.

Figure 6.5 displays the comprised memory usage of all containers in the test cluster. Similarly, the comprised predictions are not far of from the actual values. However, it is still better than the bias technique, which may be more evident with

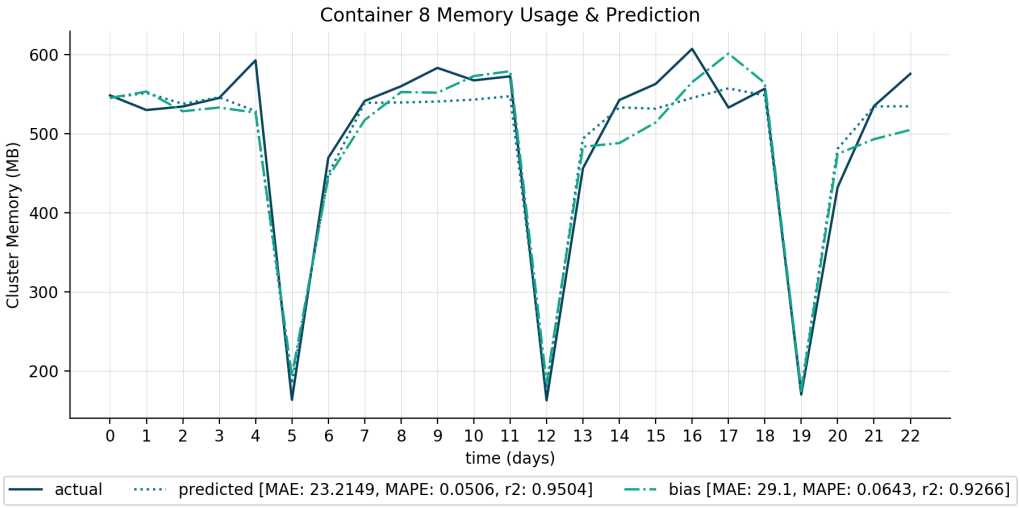

Figure 6.3: Predictions of memory usage for container 8.

regards to Table 6.2. The table shows that the model misses the actual consumption requirements by an average of $\pm 143MB$ each day, that is an average error of $\pm 2.42\%$. The accuracy of the prediction models may be compared against the MAPE of 0.1772 presented by Islam et al. [39].

Window size	MAE(MB)	MAPE	r^2
0	270.0884	0.0513	0.9330
2	392.6138	0.0667	0.8366

Table 6.2: Cluster memory prediction agreggate using window size of zero and two.

6.2 Production Environment

This section presents the results from running the presented solution in a production environment hosted by Ericsson. The section is split into three different parts. The labeling process is first described, followed by a presentation of the predictions. The final section presents the potential cost savings an organization may gain by dynamically adjust the size of the cluster by complying to the predictions made by the system.

6.2.1 Labeling

Figure 6.6 and Figure 6.7 shows how much memory containers have allocated in the cluster at two specific points in time. Squares note containers consuming low memory, crosses indicated average memory consumption, and dots indicate a high value. Fundamentally, these groupings lay the foundation of what type of label

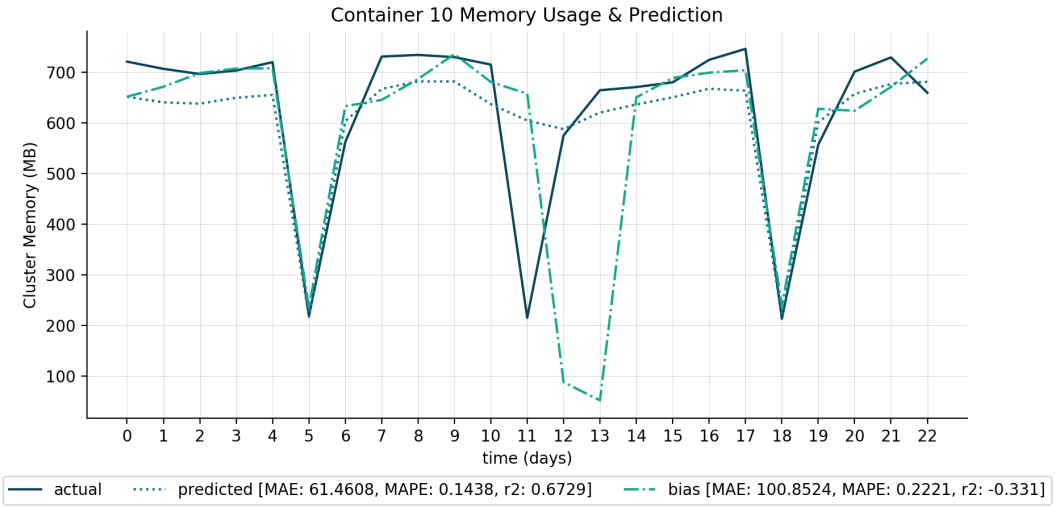

Figure 6.4: Predictions of memory usage for container 10.

should be applied to which container at a point in time, so that the orchestrator may perform more intelligent deployments.

Figure 6.6 displays a silhouette score of 0.8037 and 6.7 a score 0.7073, which indicates a good separation of groups, however similarly as noted in Section 6.1.1 the average silhouette score is used. The cluster's average silhouette score of 0.7707 calculated from 182 days indicates that the identified groups are separated effectively and better than the case of test cluster.

6.2.2 Predictions

Container	$MAE(MB)$	$MAPE$	r^2	$MAE(MB)_w$	$MAPE_w$	r^2_w
2	3.5608	0.0067	0.7149	1.1605	0.0022	0.9260
11	3.8637	0.0050	0.3056	1.5059	0.0020	0.8597
16	1.3267	0.0034	0.7741	1.0246	0.0026	0.8625
25	8.1522	0.0148	0.6854	8.3758	0.0155	0.6684
27	8.0445	0.0310	0.3981	4.295	0.0172	0.7274
37	108.2581	0.0903	0.3793	60.0607	0.0508	0.6787
57	6.9034	0.0123	0.4506	4.7591	0.0086	0.4496
73	22.2765	0.0500	-0.0276	15.5296	0.0369	0.1372
95	1.6334	0.0052	0.5058	1.3778	0.0044	0.5837
207	22.3103	0.0451	0.1436	26.9539	0.0542	-0.4268

Table 6.3: KPI metric table for each container's memory prediction, both prediction and bias (2).

Table 6.3 displays metrics in a similar fashion as described for Table 6.1; however, only a selected subset which represents common characteristics from a set of

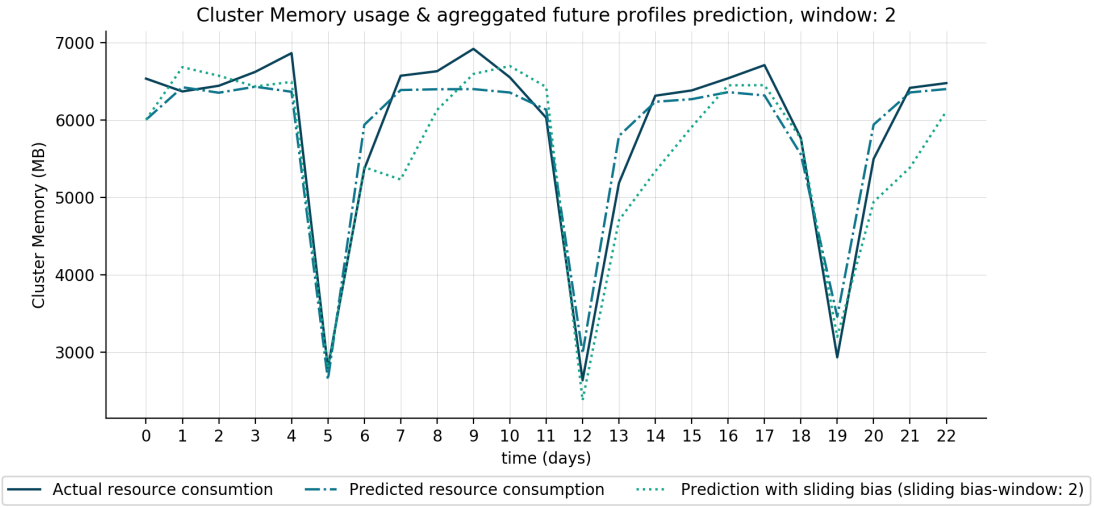

Figure 6.5: Prediction on comprise memory usage.

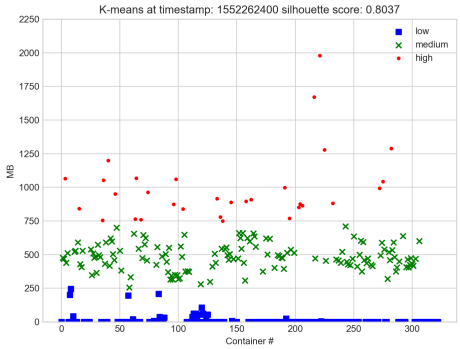

Figure 6.6: Cluster memory labeling 1552262400.

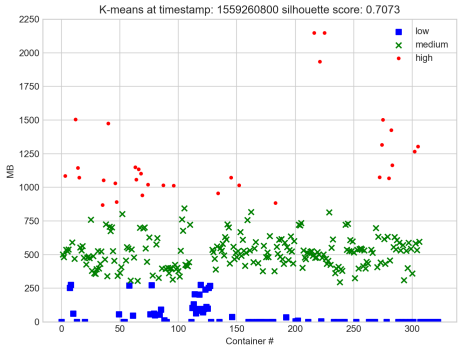

Figure 6.7: Cluster memory labeling 1559260800.

322 containers. The table and the specific example Figure A.3 displays that the models can discover specific containers profiles. Nonetheless, there are exceptions such as containers 73 and 207. Reviewing the two exceptions in Figure A.7 and Figure A.10 one may note that there is a significant lack of any apparent signatures or behaviors, that is, more than randomly spiking. In contrast, the other prediction figures exhibit that most models are capturing the underlying behavior and generates sound predictions. In general, one may note that the dotted line, which indicates non-adjusted predictions underestimate in general. The biased predictions (dashed-dotted), in this case, performs better in general.

Even though there are cases where a model cannot determine the underlying profile of a container, it tends to stay close to the mean allocation as displayed by, i.e., the r^2 value of Figure A.8. In typical cases noted in Table 6.3, they tend to stay between a score of 0.46 and 0.90. In combination with the performance of the compromised models presented in Table 6.4, it shows that a window's size

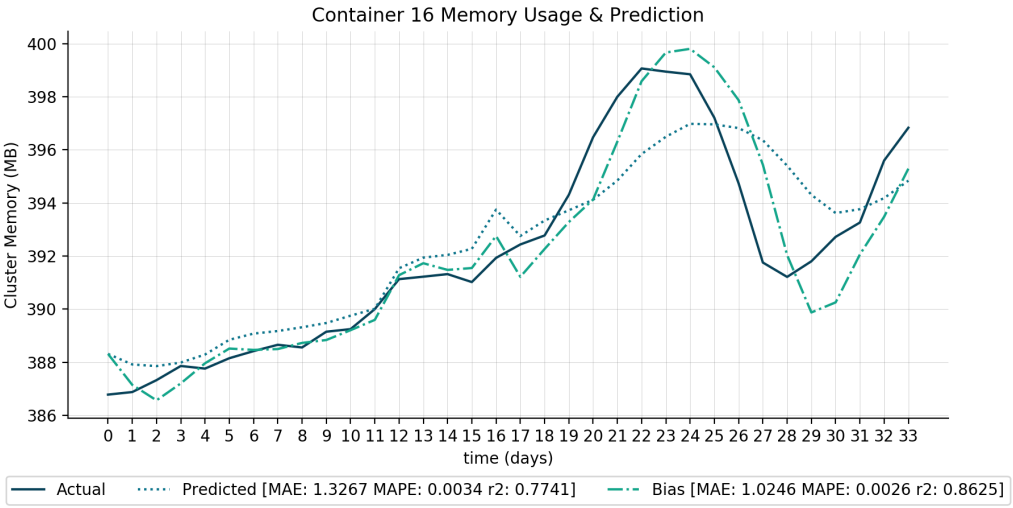

Figure 6.8: Container 16 memory usage prediction.

Window size	MAE(MB)	MAPE	r^2
0	6050.5034	0.0405	0.3331
2	1992.2012	0.0133	0.9026
4	2088.6850	0.0140	0.8943
6	2210.3290	0.0149	0.8852
8	2358.5987	0.0159	0.8753
10	2516.1772	0.0170	0.8627
12	2644.0484	0.0179	0.8516
14	2778.3810	0.0188	0.8423
16	2937.9511	0.0199	0.8304

Table 6.4: Results using different window size configurations for the prediction models.

of two is most effective and that a more significant size consequently brings back the biased prediction to the initial model configuration. In essence, the aggregated model (window size of two) makes an average error of $\pm 2024MB$ ($\pm 1.36\%$) on each day for the entire cluster size and the comprised models may account for 90.33% of the variation in memory. As for the test environment, the prediction models used here also produce lower MAPE compared to the results presented by Islam et al. (MAPE = 0.1772) [39].

6.3 Cost Estimations

The figures and tables provided in this section aim to convey how sound the aggregated prediction models are at determining appropriate cluster size configurations.

Since we need to keep a degree of confidentially with regards to Ericsson, we

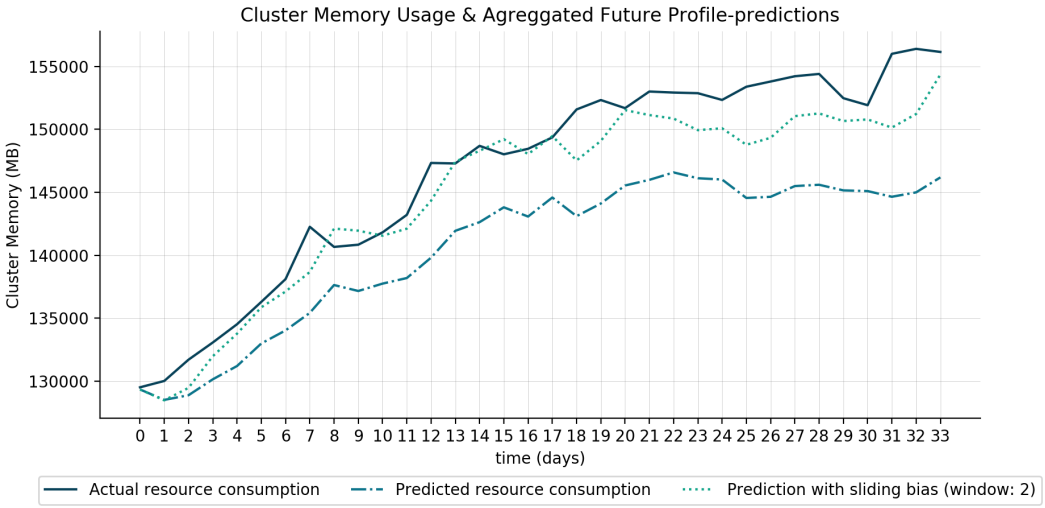

Figure 6.9: Aggregated memory prediction, sliding window: 2.

assume some properties of the cluster while making estimations. Firstly, the measurements use two variants of the "m5-machine". Next, we assume a fixed number of connected nodes to the cluster as the default configuration for the organization, and we assume that those nodes are homogenous in their configuration, for respective estimation variant.

The fixed number is selected by using the upper bound of the actual resource requirements for the cluster and add two extra nodes to illustrate a safety margin employed by the organization. Note that the constant number of nodes is entirely hypothetical and may, therefore, be better or worse in reality. However, in this example, it aims to illustrate a company which is well aware of its resource requirements and addition enforce a fail-safe (+2 nodes), to counter unforeseeable resource spikes.

To extend, during cost estimations, we make use of two types of node configurations provided by AWS: (1) m5.large and (2) m5.xlarge. AWS provides hourly estimate rates for its instances, 0.102 USD and 0.204 USD respectively. The main difference between these instances is that an "xlarge" machine essentially is a multiplied "m5.large" instance with regards to hardware specifications.

Figure 6.10 presents how the system would adjust the number of connected nodes for the cluster. The solid line represents the clusters actual consumption, the dashed - model prediction aggregate, dotted - proposed node count adjustment, dot-dashed - node count adjustment including one additional machine as safety padding, finally, the large dots display what the actual node count would be required to be. Figure 6.11 applies the same arrangement.

Figure 6.10 shows that the model can adjust its node configuration count sensibly, with minor error. More importantly, seen to all days (34) the model is producing a mean machine count error of 0.38, that indicates that model is not even off by a "half" machine in its adjustŋment proposals. Furthermore, by reviewing Table 6.5, one may note that both the dynamic and padded method is generating a small

benefit, 12%, and 7% respectively in comparison to constant node configuration of 21 (maximum required in this example) plus two machines as a maring of safety.

A similar illustration is presented in Figure 6.11, although in this case, the model is entirely spot-on while considering its node adjustment proposals. Moreover, the configuration would also impose some economic benefit, 20%, and 11%, assuming a constant configuration of 10 machines plus two machines for safety padding.

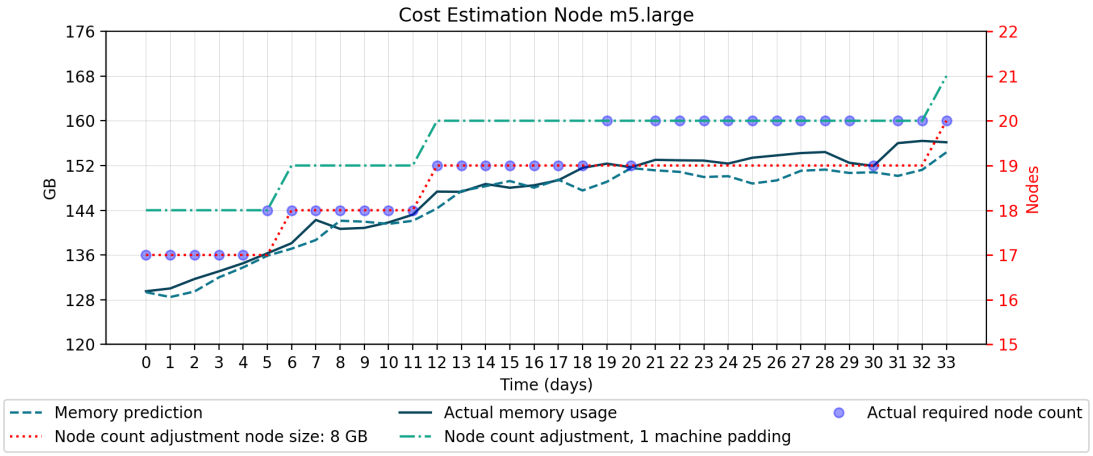

Figure 6.10: Cost estimate using m5.large.

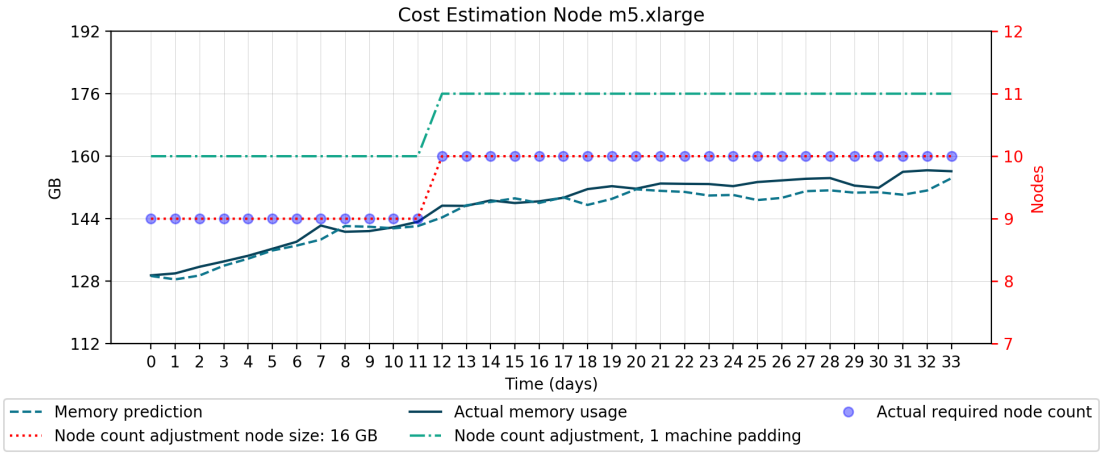

Figure 6.11: Cost estimate using m5.xlarge.

Type	Dynamic	Padding	Const	Dynamic / Const	Padding / Const
m5.large	1539.79	1623.02	1747.87	0.12	0.07
m5.xlarge	1605.89	1772.35	1997.57	0.20	0.11

Table 6.5: Results using different node configurations.

7
Discussion

The purpose of this section is to discuss the study in general and the outcome in particular. The first section highlights the lessons learned by presenting and discussing the key findings from the study. The final section discusses the potential threats to validity identified and how they have been mitigated.

7.1 Lessons Learned

This section provides a discussion of the four most notable lessons learned during the study.

Lesson 1: Trace-based resource profiles is a suitable method when empirically determining resource profiles for containers. By querying the container orchestrator's Metrics API as presented in Section 5.1 and storing the data according to Figure 5.2, trace-based container profiles, similar to Wood et al. [37], are constructed for each container in the cluster. By feeding the profiles into both the labeling and the prediction algorithms and then aggregating to determine the total cluster size, traceability of each container is guaranteed. Additionally, the results prove that it is possible to produce sound predictions for both small, test environment with 17 containers, and large clusters, production environment with 322 containers, using the presented method. In other words, it does not seem to exist a trade-off between traceability of individual containers and predicting future resource needs of a container-based cluster. This fact may sound trivial. However, the majority of previous research has been on a VM-level, which results in a complete loss of traceability in terms of individual containers. Without the traceability ensured by our system, an organization would only be able to predict if the cluster is going to increase or decrease in size. They would not be able to find out if the change is due to a general increase in workload for the entire cluster, or if only a couple of containers are causing the increased need.

Lesson 2: K-Means clustering provide valuable insights on how individual containers may be deployed to increase the robustness of a cluster. The silhouette scores noted in Sections 6.1.1 and 6.2.1 produced from clustering containers relatively by their memory usage indicates that K-Means is a reliable tool for the specific task. Hence, it is a suitable method organizations may consult to produce more robust deployments. Both the test and live cluster produce good silhouette scores, which indicate that the model can produce sound groupings in each

environment. The live result of a mean silhouette score of is in numeric reference a better number than that of the test cluster. However, the discrepancy here most likely depends more on the fact that there are 17 containers in the test environment and 322 in the live environment and a notable difference in collected data points. So a direct comparison should not be administered since the silhouette score mainly derives its score from how similar objects are with regards to its cluster compared to other groups, in short, cohesion and separation. It should, however, be noted that there is a trade-off between having a resource-efficient cluster and having a robust cluster. That is, the most robust cluster an organization could have would be if each container were deployed to separate nodes. Doing so would in almost every case also lead to a high degree of wasted resources. Which direction an organization takes is entirely dependent on its deployment strategy. Nevertheless, the presented method may provide valuable insight into how an organization could separate containers with similar resource profiles to increase the robustness of the cluster.

Lesson 3: LSTM-based RNNs are well suited to predict future resource needs from trace-based container profiles. While examining the presented prediction results, we discern that the learned container profiles, their subsequent predictions, and in union with the presented absolute errors, provide evidence that the method is reliable. Reliable, meaning that the models predict individual containers future resource requirements with marginal errors, as displayed in Table 6.4. Additionally, the cluster aggregate, displayed in Figure A.11, entails that individual tracking and prediction of containers may later on accurately express and predict the requirements of the cluster. This is further supported when comparing our prediction models MAPE of 0.0667 (test environment) and 0.0133 (production environment) with the 0.1772 Islam et al. achieved [39]. As learned profiles emit satisfactory predictions, they may be used as an empirically produced tools to scientifically and adequately determine containers future consumption and their lower/ upper bounds. Furthermore, as illustrated in Figure A.11, the models would allow Ericsson to determine if the clusters available resources are configured too high in contrast to its running containers and their resource requirements. In short, they may declare if overallocation exists, and if so, what is causing it. Finally, as the two clusters analyzed in this thesis exhibit completely different behaviours, the test cluster being small and cyclic and the production cluster being large and more linear, the LSTM-based RNN models prove to be highly adaptable.

Lesson 4: There are cost savings to be made by proactively adjusting the cluster size according to future resource needs. As illustrated in Section 6.3, it is evident that the model does not produce an extraordinary result seen to cost reduction. However, that depends entirely on how accurate the clusters previous configurations are. In effect, that depends entirely on how good an organization already is at tracking and adjusting infrastructure accordingly. Aside from potential economic improvements, the model aggregate is giving reasonable adjustment proposals, Figure 6.10 displays a perfect example. Remember that the purpose of this process is to allow organizations to track and maintain their cluster accurately. The tool gives a good indication on exactly how many nodes should be attached

(depending on node size), that is, the baseline of the cluster. Potential spikes or other events excluded from a regular profile may still be countered using reactive auto-scaling. Nonetheless, the organization grasps precisely where resource consumption resides and may adjust or plan their infrastructure using these empirical instruments. Depending on how the company conforms to these finding, they may or may not reduce the amount of wasted resource. Still, that depends on what one defines as waste. If the heap of unused space is considered to be a safety consideration, then one might not think of it as waste but more as a strategic decision. Despite perceptual definitions, an organization may use these tools to determine, with precision, if they are over- or underallocating.

7.2 Threats to Validity

The purpose of this section is to disclose any threats to validity that have been identified and, if possible, how they have been mitigated.

External Validity. Case studies have an inherent problem when it comes to generalizability, which is a threat to external validity [40]. Given that this study was done as a single case study, we cannot be guaranteed that the findings are generalizable across other cases. However, because the case was defined at the cluster level, potential organizational impact is somewhat controlled for. Furthermore, the system is implemented to work for both Apcera- and Kubernetes-based clusters, making it possible to run the same analysis regardless of the cluster's infrastructure, hence, improving the system's generalizability. In the end, however, replications of this study are needed to ensure generalizability.

Construct Validity. Threats to construct validity concerns whether or not metrics used represents what they are intended to measure [40]. This issue is not regarded as a significant threat in this study since no custom metrics were defined or used. All data about containers' resource usage are accessed through the official Metrics API, as presented in section 4.2. Furthermore, MAE and MAPE are commonly used when evaluation prediction models and should not pose a threat. The official scikit-learn documentation [48] suggests using the silhouette coefficient to evaluate how well unsupervised clustering algorithms such as the K-Means algorithm performs. However, it should be noted that unsupervised learning algorithms are inherently troublesome to evaluate since there is no objective truth in what makes a good or bad cluster. Finally, when estimating the total cluster size by aggregating each container, system components used by the container orchestrator are not accounted for, that is, the estimation is always going to be off by a certain amount. However, as Xu et al. [5] stated, the Docker infrastructure does not add much overhead and should therefore not be a notable threat.

Internal Validity. Threats to internal validity refer to the problems of confounding factors as well as selection and researcher bias [40]. The analysis of resource usage and cluster size is limited only to consider memory and CPU. There are other

metrics, such as file I/O and network I/O, available that might be the limiting factor for some containers. However, based on our own experiences, discussions with Ericsson and reviewing previous research, it is reasonable to assume that the overall limiting factor is memory. Excluding other metrics might also be seen as a potential bias on our end. However, the decision is supported by what previous research has done. Finally, concerning selection bias, the cluster analyzed in this thesis was independently selected by Ericsson without any influence from us. This fact should mitigate the risk of selection bias, given that the primary objective for Ericsson was to gain business value by collaborating with us.

Conclusion Validity. Threats to conclusion validity refer to whether or not it is possible to conclude what we claim from our study. It is clear from our findings that the presented method can determine if a cluster has allocated more resources than the containers use. However, it is not possible to determine whether or not this is due to misconfigured containers or poor node configurations. The discrepancy might be a strategic decision from the organization to add extra capacity in case of extraordinary events such as a data center going down. To fully answer why there is a discrepancy, further investigation, such as interviews with developers and management, is necessary. Finally, when comparing our results with the reults presented by Islam et al. [39] it is not possible to determine that our method is superior to theirs, as we use different models and analyse different data. The comparison does, however, serve as useful reality check as we would not expect any worse results than theirs.

8

Conclusion

The purpose of this section is to conclude this thesis by looking back at the initial goal and research questions stated in Section 1. To reiterate, the overarching goal was to explore, evaluate, and implement an automatic profiler for containerized applications so organizations may track and estimate their containers' resource requirements. To achieve this, we decided to conduct a case study in collaboration with Ericsson as an industrial partner. Ericsson provided expert knowledge in the field and the technologies used throughout the study, as well as access to clusters hosted in both a test and a production environment. In other words, the collaboration provided access to real data in contrast to simulated data as previous studies have used. Using real data in combination with ensuring tracability of individual containers are what fundamentally differ this thesis from the related work presented in Section 3. The principal outcome of the study is an automatic system comprised of three submodules. First, a data collector that automatically registers containers and their historical resource usage in a database by querying the metrics API of the container orchestrator. Next, a system that queries the database and, using K-Means clustering, assigns labels to each container based on their relative resource usage. Finally, a predictor, based on a RNN and LSTM cells, that predicts a container's future resource needs. The system was used to answer the following research questions.

RQ 1: How can containers be profiled to describe their actual usage of resources accurately? While reviewing previous research, we found that profiles of containerized applications may be defined in various sophisticated ways. While both Do et al. [34] and Kundu et al. [36] suggested profiles based on the relationship between performance and resource utilization, we adopted a trace-based method similar to [37]. The trace of a container's actual resource usage is first gathered by querying the container orchestrator's Metrics API and then stored in a database according to Figure 5.2. The set of all historical records for a container is then what we define as that particular container's profile. This way of collecting container profiles ensures empirically defined profiles that accurately represents a container's actual usage of resources. Furthermore, as the profiles are used as input to all further operations, tracability of individual containers are always guaranteed. That is, with the presented method it is always possible to trace a individual container's contribution when (1) clustering the relative resource usage of containers, (2) predicting future resource requirements, and/or (3) evaluating the total size of the cluster.

RQ 2: Does a profile-based cluster lead to less resource consumption while retaining or improving its availability? The developed system gives organizations the possibility to examine actual resource behavior and ascertain which container is adding to the difference. However, if a profile-based cluster leads to less resource consumption depends entirely on what type of deployment strategy that is employed. From the findings presented in Section 6.3, it is easy to imagine three different deployment strategies. First, a cautious approach may be to have a static node count, well over the actual requirements in case of an extraordinary event such as a data center going down. Second, a balanced approach might use the predictions as a base but add one or two extra nodes in case of an unforeseen spike in workload. Third, an aggresive approach which dynamically adjusts the node count according to the predictions. Nonetheless, the presented system structure allows organizations to determine the cluster's total and individual consumption adequately.

To conclude, this thesis presents a system for automatic container profiling and resource prediction using container orchestration. The profiles are constructed from containers actual resource usage, gathered by querying the container orchestrator's Metrics API and thus accurately describes the actual resource usage of individual containers. Furthermore, the system guarantees traceability by using individual container profiles as input in all operations. Additionally, the system can accurately predict a clusters future resource requirements. However, a profile-based cluster does not guarantee less resource consumption. Whether or not fewer resources are consumed depends entirely on an organizations deployment strategy. That being said, the system allows an organization to evaluate its cluster configuration empirically.

8.1 Future Works

The purpose of this section is to provide suggestions on future work and potential enhancements of the system presented in this thesis.

8.1.1 Container Complexity

A common practice among software engineers is the process of determining the complexity of an algorithm (big O), which is an indication of how a specific function is performing with regards to computational requirements (memory and CPU) needed to complete its internal instructions. On a more general level, one may see the entire container as a function; what is interesting is to determine the complexity of the container as requests increase or decrease. Commonly, one of its resources will eventually become a bottleneck. Such a definition would allow operational teams to determine when additional replicas of a container should be deployed, to load balance the abundance of requests since the current container eventually becomes limited, seen to physical limitations.

8.1.2 Fault Detection

An automated system would be able to compare previous models with current resource consumption of a container, as long as each version is tagged with a revision id. One could then assume that something is faulty in a container if its current resource behavior starts to deviate too much from previously created model revisions. When the system receives that type of stimuli, it could either alert the operational team or rollback to a properly running revision.

8.1.3 Automatic Infrastructure Redeployments

By utilizing the produced container models as displayed in results, one can estimate the total resource tally of the cluster. The cluster's cumulative resources are dependent on the number of connected machines. Therefore, the predicted estimate can be used to determine what type of machines and how many those should be, to fit future requirements.

If a tool for generating a near-perfect infrastructure schema can be generated, other tools such as terraform or kops may be used to employ that infrastructure. As a result, the organization's cluster would adjust both its node configuration and node count. In essence, the cluster would be entirely dynamic by frequently adapting to its running containers and their respective resource utilization.

Bibliography

[1] Docker Inc. (2019). What is a container?, [Online]. Available: https://www.docker.com/resources/what-container (visited on Mar. 19, 2019).

[2] ——, (2019). Limit a container's resources, [Online]. Available: https://docs.docker.com/config/containers/resource_constraints (visited on Dec. 10, 2018).

[3] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow", in *ICAS 2011, 7th Int. Conf. Autonomic and Autonomous Systems (ICAS)*, Venice, Italy, 2011, pp. 67–74. [Online]. Available: https://www.cl.cam.ac.uk/~ey204/teaching/ACS/R212_2015_2016/papers/dutreilh_icas_2011.pdf (visited on Aug. 6, 2019).

[4] D. Edsinger, "Auto-scaling cloud infrastructure with Reinforcement Learning: A comparison between multiple RL algorithms to auto-scale resources in cloud infrastructure", M.S. thesis, Computer Sci. and Eng., Chalmers Uni. of Technology, Gothenburg, Sweden, 2018. [Online]. Available: http://publications.lib.chalmers.se/records/fulltext/256475/256475.pdf (visited on Aug. 6, 2019).

[5] P. Xu, S. Shi, and X. Chu, "Performance evaluation of deep learning tools in Docker containers", in *2017 3rd Int. Conf. Big Data Computing Communications (BIGCOM)*, Chengdu, China, 2017, pp. 395–403. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8113094 (visited on Aug. 6, 2019).

[6] The Linux Foundation. (2016). Open container initiative: About, [Online]. Available: https://www.opencontainers.org/about (visited on Dec. 10, 2018).

[7] E. Carter. (May 29, 2018). 2018 Docker usage report container initiative, Sysdig, [Online]. Available: https://sysdig.com/blog/2018-docker-usage-report/ (visited on Dec. 10, 2018).

[8] Ericsson. (2019). Company facts, [Online]. Available: https://www.ericsson.com/en/about-us/company-facts (visited on Jul. 18, 2019).

[9] ——, (2019). Portfolio, [Online]. Available: https://www.ericsson.com/en/portfolio (visited on Jul. 18, 2019).

[10] ——, (2019). Connected vehicle cloud, [Online]. Available: https://www.ericsson.com/en/portfolio/iot-and-new-business/iot-solutions/iot-for-automotive/connected-vehicle-cloud (visited on Jul. 18, 2019).

[11] P. M. Mell and T. Grance, "Sp 800-145. the nist definition of cloud computing", Gaithersburg, MD, United States, Tech. Rep., 2011. [Online]. Available:

https://csrc.nist.gov/publications/detail/sp/800-145/final (visited on Aug. 7, 2019).

[12] Amazon Web Services. (2019). Features of amazon EC2, [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html (visited on Mar. 25, 2019).

[13] ——, (2019). AWS CLI command reference, [Online]. Available: https://docs.aws.amazon.com/cli/latest/reference/ec2/index.html (visited on Mar. 25, 2019).

[14] D. A. Bader and R. Pennington, "Applications", *The Int. Jnl. of High Performance Computing Applications*, vol. 15, no. 2, pp. 181–185, May 2001. [Online]. Available: https://journals.sagepub.com/doi/abs/10.1177/109434200101500211?journalCode=hpcc (visited on Aug. 6, 2019).

[15] Google Cloud. (2019). Cloud computing services, [Online]. Available: https://cloud.google.com/ (visited on Aug. 6, 2019).

[16] Microsoft Azure. (2019). Microsoft azure cloud computing platform & services, [Online]. Available: https://azure.microsoft.com/en-us/ (visited on Aug. 6, 2019).

[17] The Kubernetes Authors. (Jul. 11, 2019). Concepts, [Online]. Available: https://kubernetes.io/docs/concepts/ (visited on Aug. 6, 2019).

[18] Apcera Inc. (Sep. 11, 2017). Apcera architecture and components, [Online]. Available: https://docs.apcera.com/introduction/apcera-arch/ (visited on Aug. 6, 2019).

[19] ——, (Oct. 20, 2016). Service bindings, [Online]. Available: https://docs.apcera.com/services/services/#service-bindings (visited on Aug. 7, 2019).

[20] The Kubernetes Authors. (Aug. 1, 2019). Discovering services, [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/#discovering-services (visited on Aug. 7, 2019).

[21] Amazon Web Services. (2019). Amazon EC2 M5 Instances, [Online]. Available: https://aws.amazon.com/ec2/instance-types/m5/ (visited on May 2, 2019).

[22] I. Eldridge. (Jul. 17, 2018). What is container orchestration?, New Relic, [Online]. Available: https://blog.newrelic.com/engineering/container-orchestration-explained/ (visited on Mar. 19, 2019).

[23] V. Maini. (Aug. 19, 2017). Machine learning for humans, part 3: Unsupervised learning: Clustering and dimensionality reduction: K-means clustering, hierarchical clustering, principal component analysis (PCA), singular value decomposition (SVD), [Online]. Available: https://medium.com/machine-learning-for-humans/unsupervised-learning-f45587588294 (visited on Aug. 6, 2019).

[24] S. Mishra. (May 19, 2017). Unsupervised learning and data clustering, [Online]. Available: https://towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a (visited on Aug. 6, 2019).

[25] R. Xu and D. Wunsch, "Survey of clustering algorithms", vol. 16, no. 3, pp. 645–678, May 2005. [Online]. Available: https://ieeexplore.ieee.org/document/1427769 (visited on Aug. 6, 2019).

[26] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm", *Jnl. of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979. [Online]. Available: https://www.jstor.org/stable/2346830 (visited on Aug. 6, 2019).

[27] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithms: Analysis and implementation", *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 24, no. 7, pp. 881–892, Jul. 2002. [Online]. Available: http://ieeexplore.ieee.org/document/1017616/ (visited on Aug. 6, 2019).

[28] S. Haykin, *Neural networks and Learning Machines*, 3rd ed. New York: Prentice Hall, 2009.

[29] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial neural networks: A tutorial", *Computer*, vol. 29, no. 3, pp. 31–44, Mar. 1996. [Online]. Available: http://ieeexplore.ieee.org/document/485891/ (visited on Aug. 6, 2019).

[30] Y. Bengio, P. Simard, and P. Frasconi, "Learning Long-Term Dependencies with Gradient Descent is Difficult", *IEEE Trans. on Neural Networks*, vol. 5, no. 2, pp. 157–166, Mar. 1994. [Online]. Available: https://ieeexplore.ieee.org/document/279181 (visited on Aug. 6, 2019).

[31] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory", *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: https://www.mitpressjournals.org/doi/abs/10.1162/neco.1997.9.8.1735 (visited on Aug. 6, 2019).

[32] C. Olah. (Aug. 27, 2015). Understanding LSTM networks, [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on Mar. 25, 2019).

[33] S. Yan. (Mar. 16, 2016). Understanding LSTM and its diagrams, [Online]. Available: https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714 (visited on Mar. 25, 2019).

[34] A. V. Do, J. Chen, C. Wang, Y. C. Lee, A. Y. Zomaya, and B. B. Zhou, "Profiling Applications for Virtual Machine Placement in Clouds", in *2011 IEEE 4th Int. Conf. Cloud Computing*, Washington, DC, USA, 2011. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6008768 (visited on Aug. 6, 2019).

[35] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao, "Application performance modeling in a virtualized environment", in *The 16th IEEE Int. Symp. High-Performance Computer Architecture (HPCA)*, Bangalore, India, 2010, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/5463058/ (visited on Aug. 6, 2019).

[36] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques", in *The 8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environments (VEE)*, New York, NY, USA, 2012, pp. 3–14. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2151024.2151028 (visited on Aug. 6, 2019).

[37] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy, "Profiling and Modeling Resource Usage of Virtualized Applications", in *the 9th ACM/IFIP/USENIX Int. Conf. Middleware*, Leuven, Belgium, 2008, pp. 366–387. [Online]. Avail-

able: http://dl.acm.org/citation.cfm?id=1496950.1496973 (visited on Aug. 6, 2019).

[38] J. Huang, C. Li, and J. Yu, "Resource prediction based on double exponential smoothing in cloud computing", in *2012 2nd Int. Conf. on Consumer Electronics, Communications and Networks (CECNet)*, Yichang, China, 2012, pp. 2056–2060. [Online]. Available: http://ieeexplore.ieee.org/document/6201461/.

[39] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud", *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, Jan. 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X11001129 (visited on Aug. 6, 2019).

[40] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009. [Online]. Available: https://link.springer.com/article/10.1007/s10664-008-9102-8 (visited on Aug. 6, 2019).

[41] D. I. K. Sjoberg, T. Dyba, and M. Jorgensen, "The Future of Empirical Methods in Software Engineering Research", in *Future of Software Engineering (FOSE '07)*, Minneapolis, MN, USA, 2007, pp. 358–378. [Online]. Available: http://ieeexplore.ieee.org/document/4221632/ (visited on Aug. 6, 2019).

[42] C. Wohlin, M. Höst, and K. Henningsson, "Empirical Research Methods in Software Engineering", in *Empirical methods and studies in software engineering*, R. Conradi and A. I. Wang, Eds., Berlin, Heidelberg: Springer, 2003, pp. 7–23. [Online]. Available: http://link.springer.com/10.1007/978-3-540-45143-3_2 (visited on Aug. 6, 2019).

[43] the United Nations. (2015). Goal 12: Ensure sustainable consumption and production patterns, [Online]. Available: https://www.un.org/sustainabledevelopment/sustainable-consumption-production/ (visited on Aug. 7, 2019).

[44] R. K. Yin, "The Case Study as a Serious Research Strategy", *Science Communication*, vol. 3, no. 1, pp. 97–114, 1981. [Online]. Available: https://journals.sagepub.com/doi/10.1177/107554708100300106 (visited on Aug. 6, 2019).

[45] K. Stol and B. Fitzgerald, "The ABC of Software Engineering Research", *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 3, pp. 1–51, 2018. [Online]. Available: https://doi.org/10.1145/3241743 (visited on Aug. 6, 2019).

[46] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying Software Engineers: Data Collection Techniques for Software Field Studies", *Empirical Software Engineering*, vol. 10, no. 3, pp. 311–341, 2005. [Online]. Available: http://link.springer.com/10.1007/s10664-005-1290-x (visited on Aug. 6, 2019).

[47] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach", *Encyclopedia of software engineering*, pp. 528–532, 1994.

[48] Scikit-learn Developers. (2019). 2.3. clustering, [Online]. Available: https://scikit-learn.org/stable/modules/clustering.html (visited on Aug. 6, 2019).

A

Predictions of Memory Usage in Live Cluster

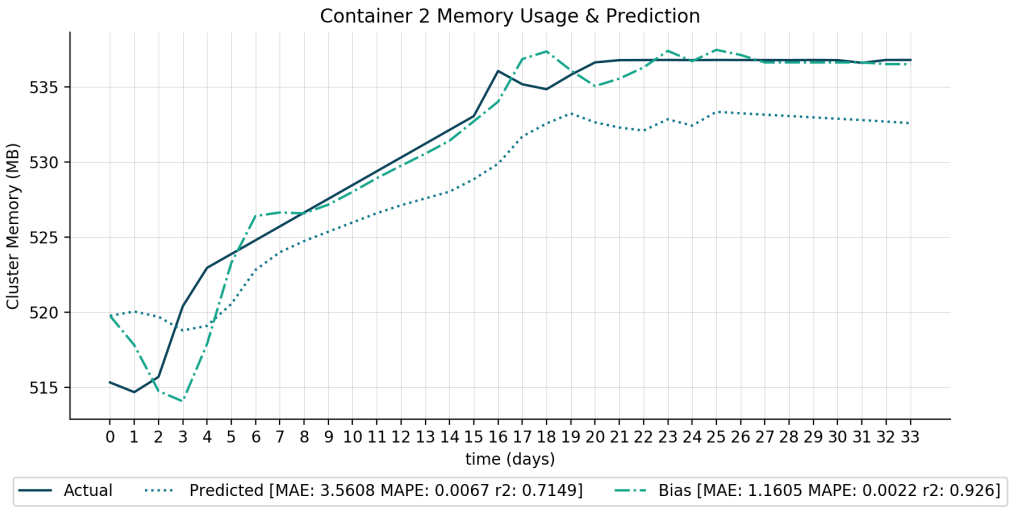

Figure A.1: Container 2 memory usage prediction.

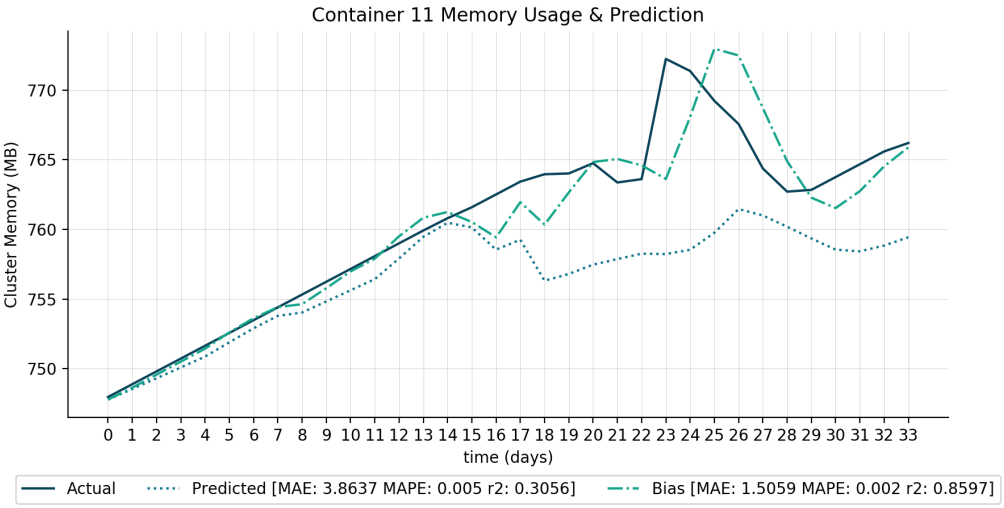

Figure A.2: Container 11 memory usage prediction.

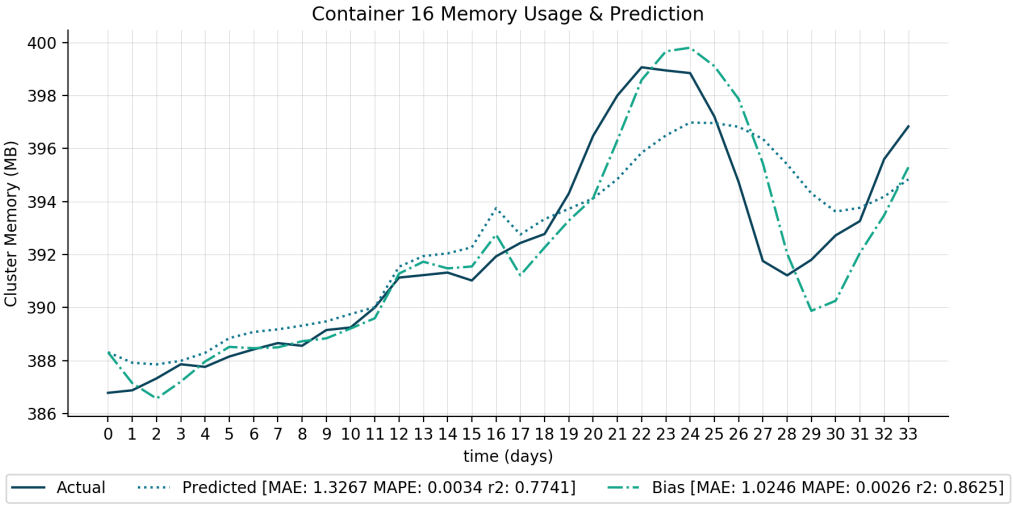

Figure A.3: Container 16 memory usage prediction.

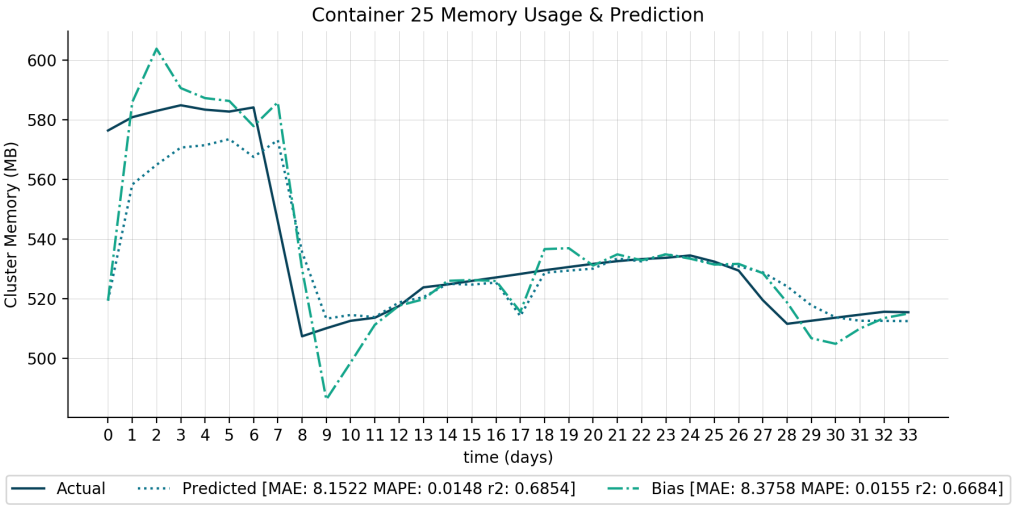

Figure A.4: Container 25 memory usage prediction.

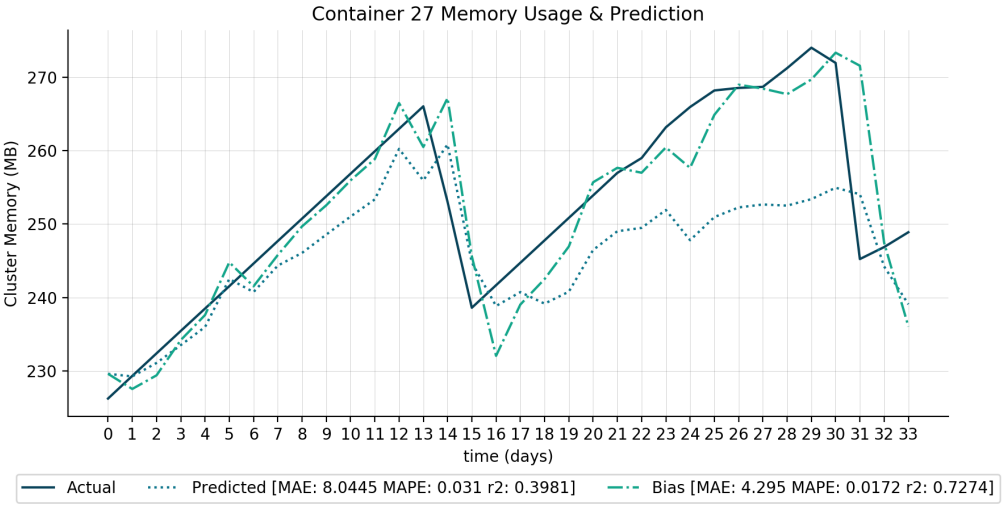

Figure A.5: Container 27 memory usage prediction.

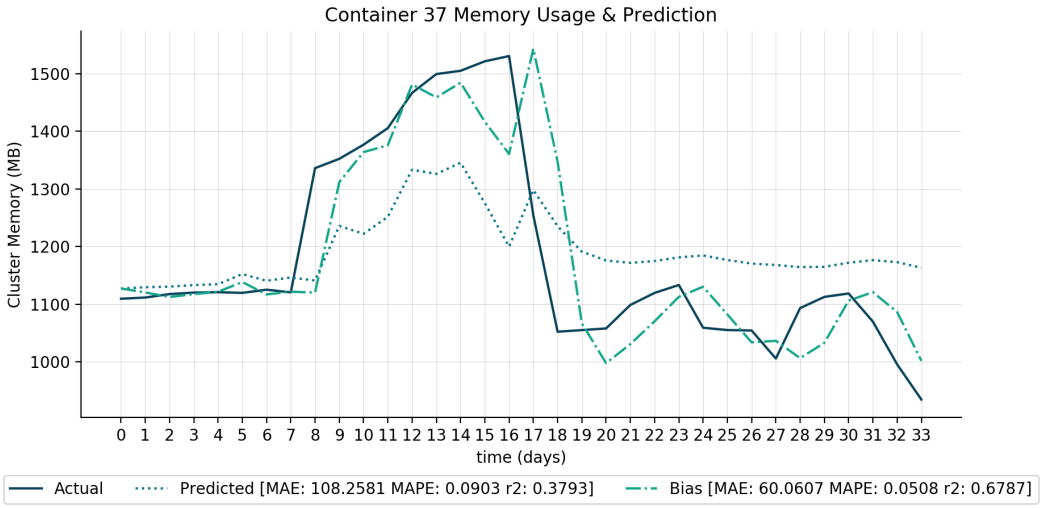

Figure A.6: Container 37 memory usage prediction.

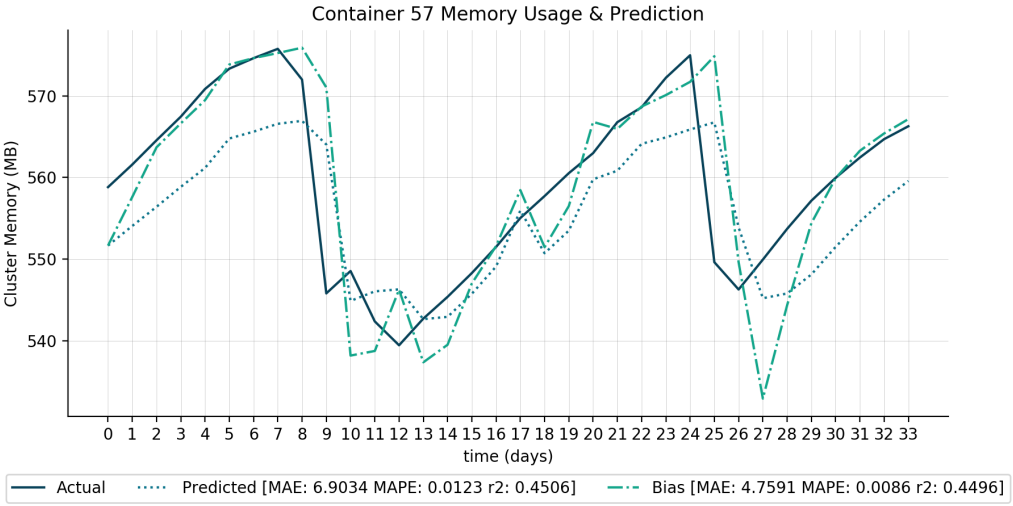

Figure A.7: Container 57 memory usage prediction.

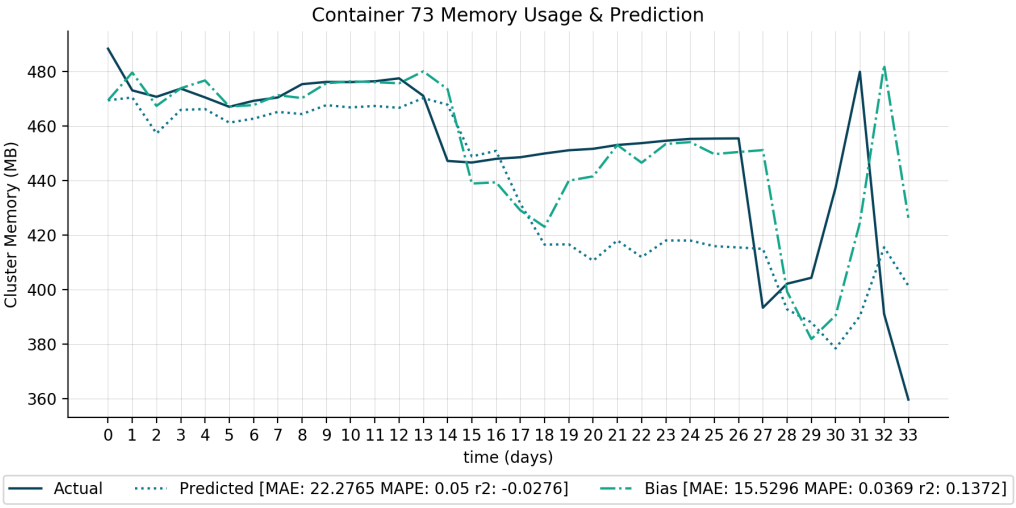

Figure A.8: Container 73 memory usage prediction.

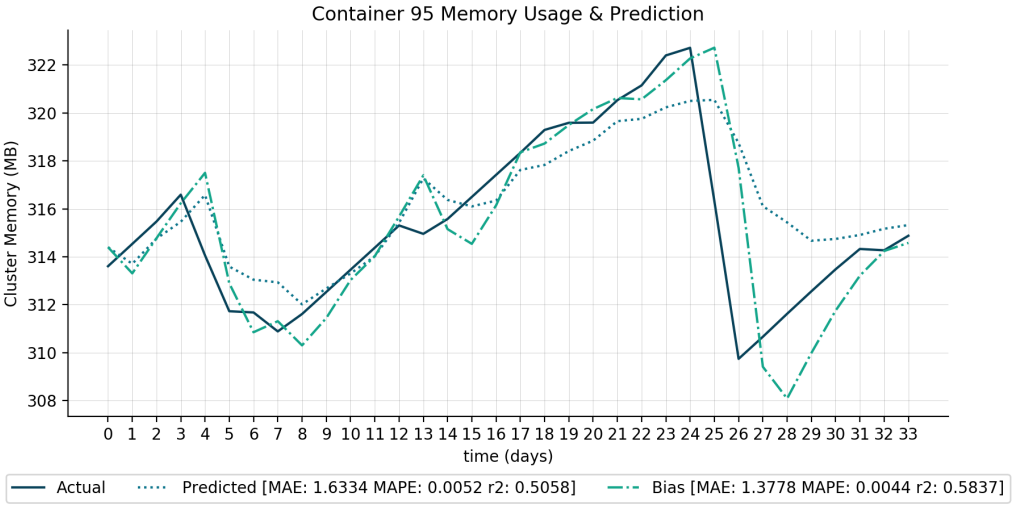

Figure A.9: Container 95 memory usage prediction.

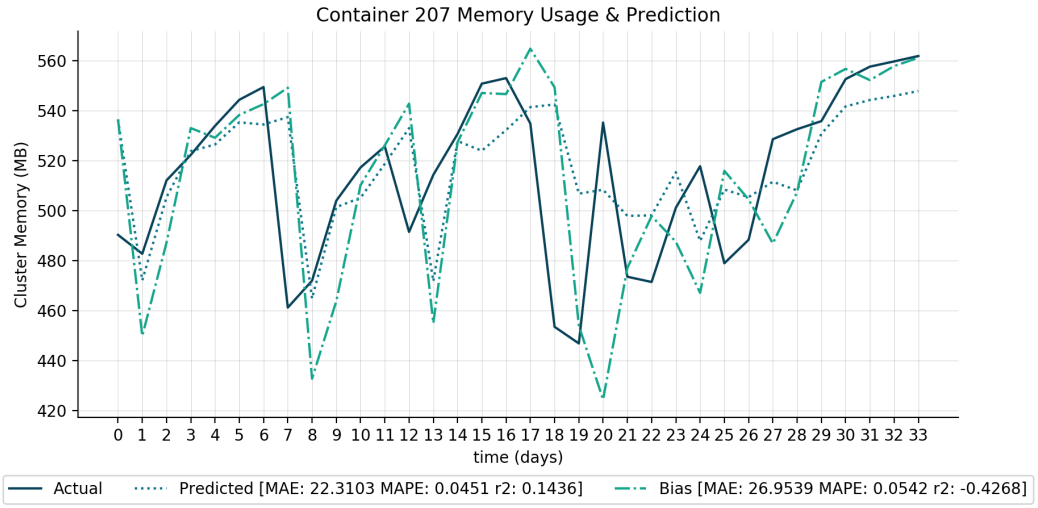

Figure A.10: Container 207 memory usage prediction.

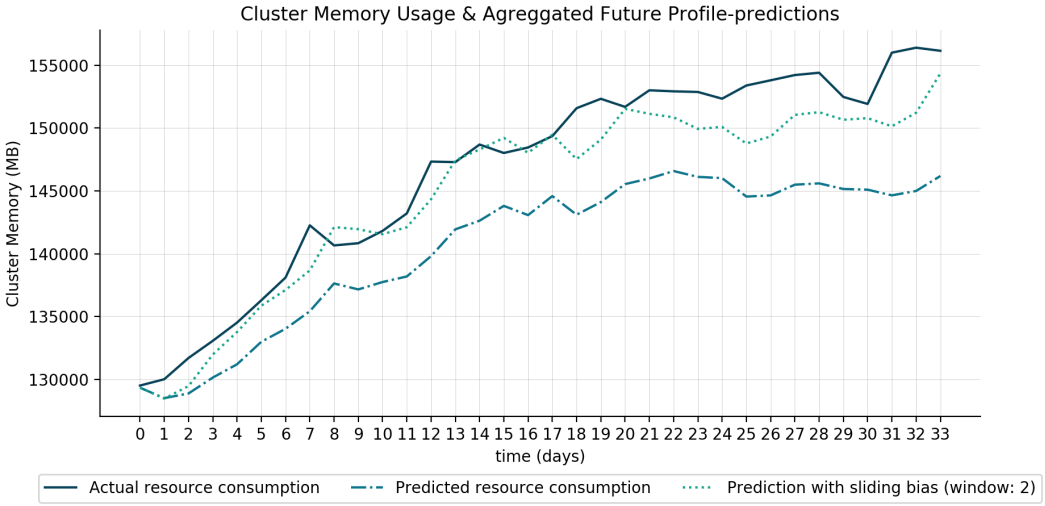

Figure A.11: Aggregated memory prediction, sliding window: 2.

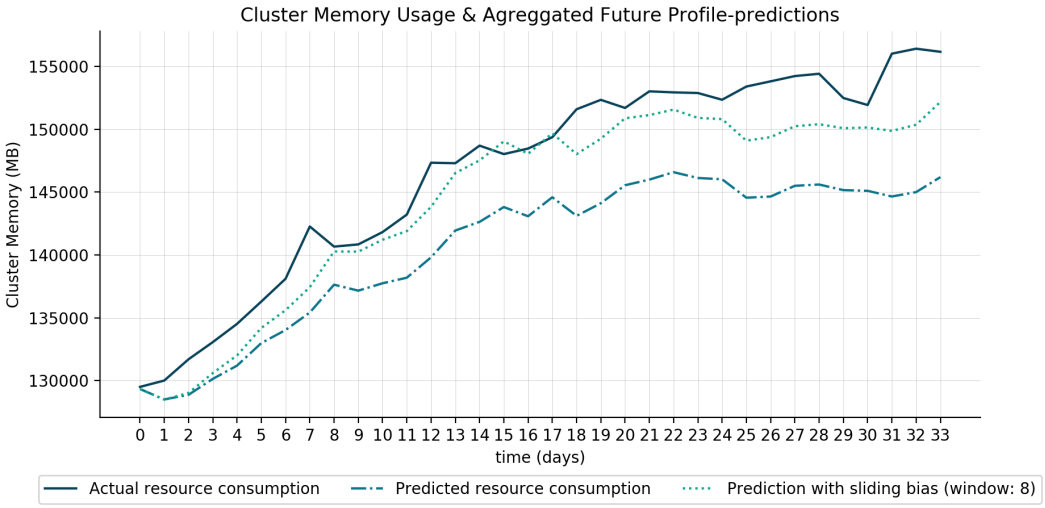

Figure A.12: Aggregated memory prediction, sliding window: 8.

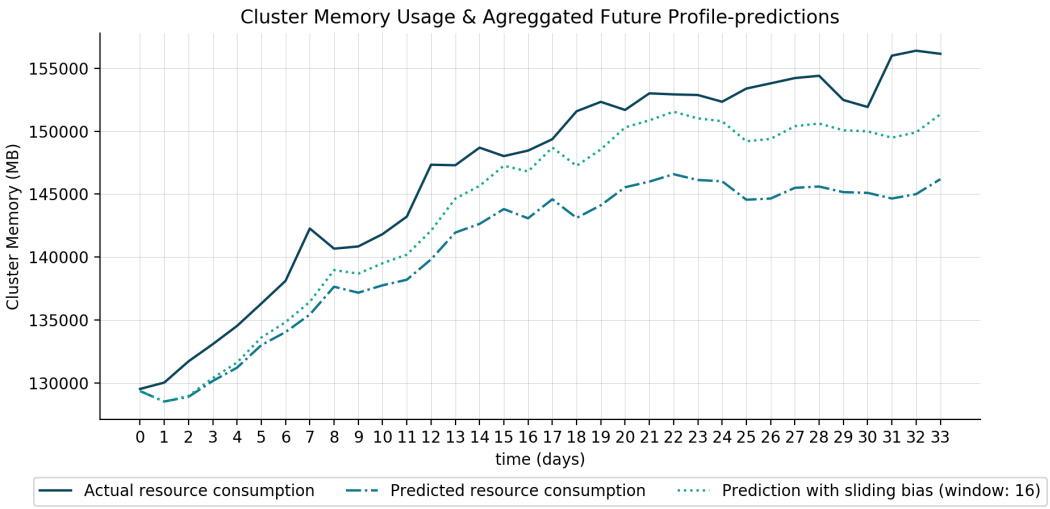

Figure A.13: Aggregated memory prediction, sliding window: 16.

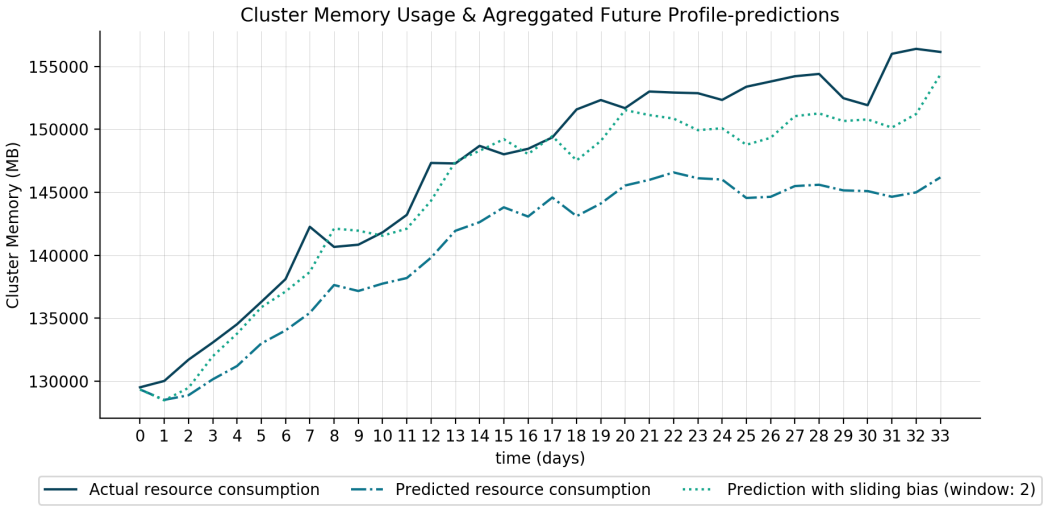

Figure A.14: Aggregated memory usage prediction

B

Predictions of Memory Usage in Test Cluster

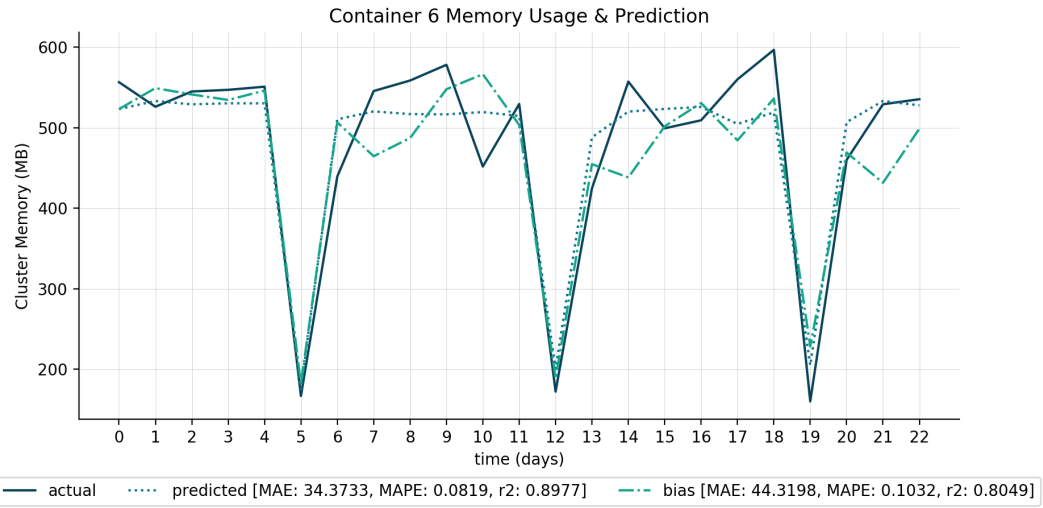

Figure B.1: Prediction container 6.

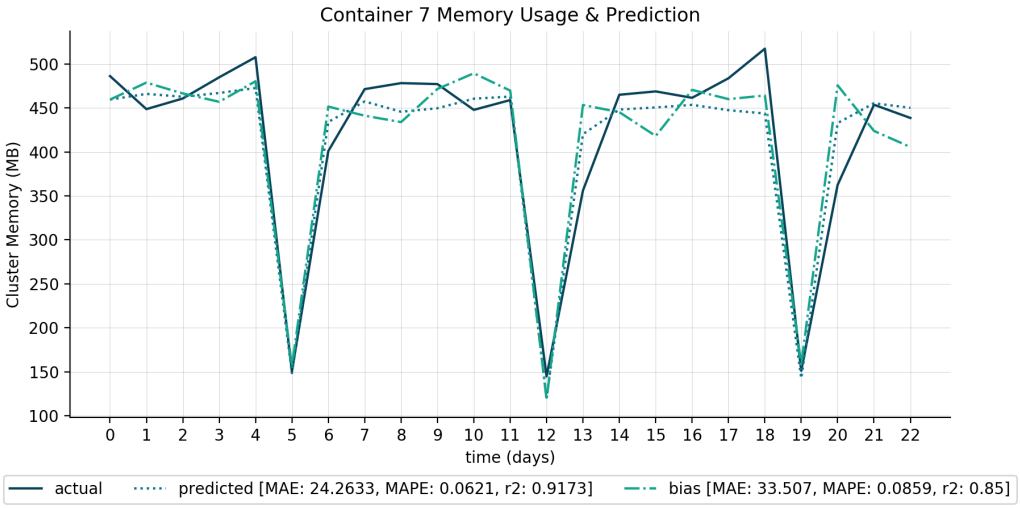

Figure B.2: Prediction container 7.

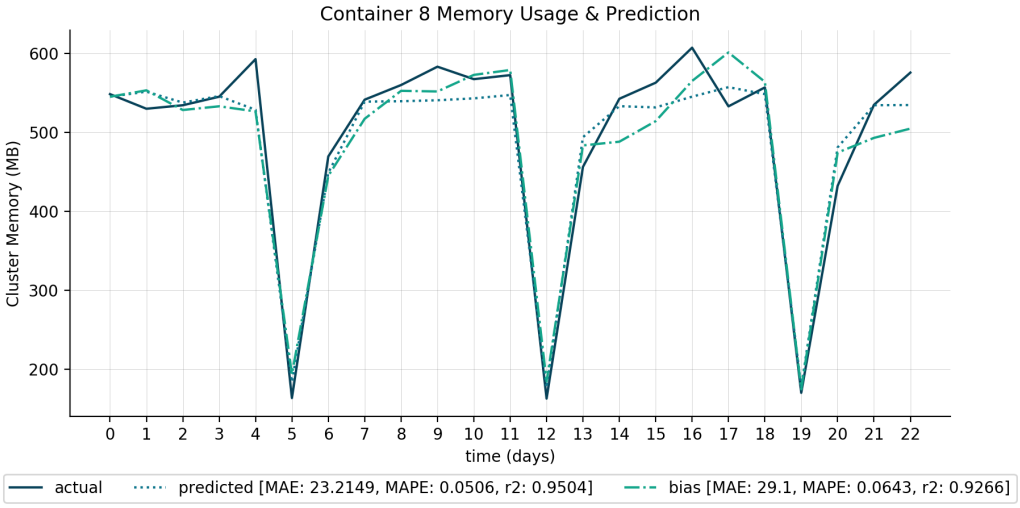

Figure B.3: Prediction container 8.

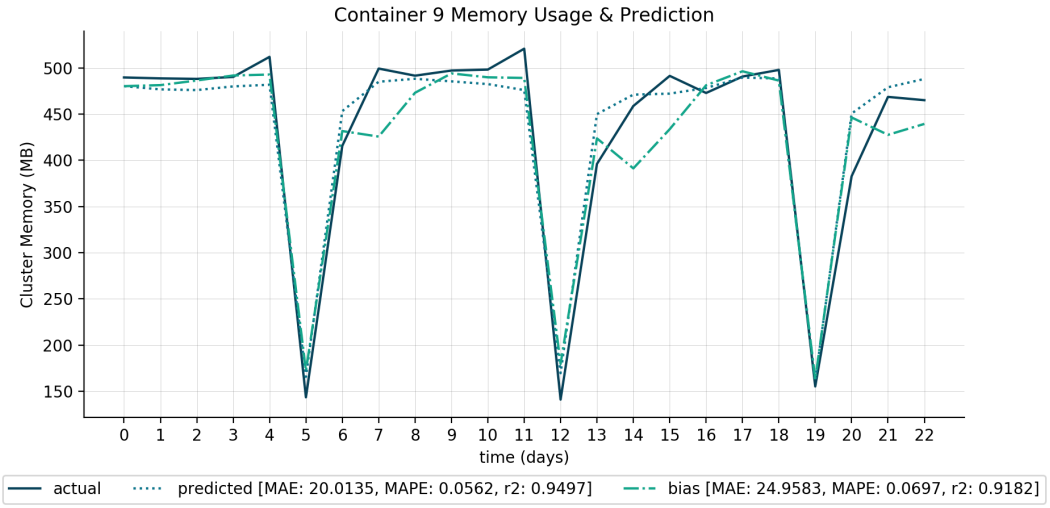

Figure B.4: Prediction container 9.

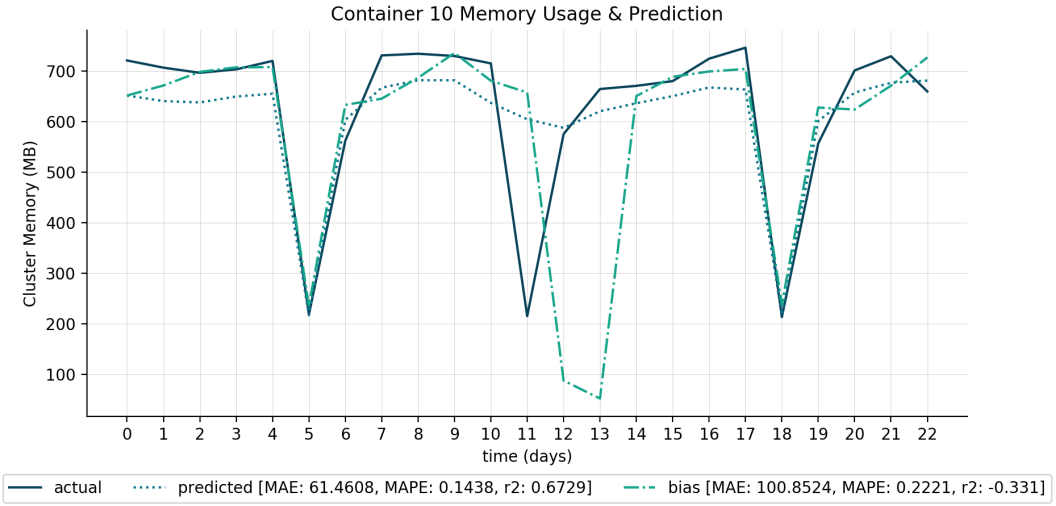

Figure B.5: Prediction container 10.

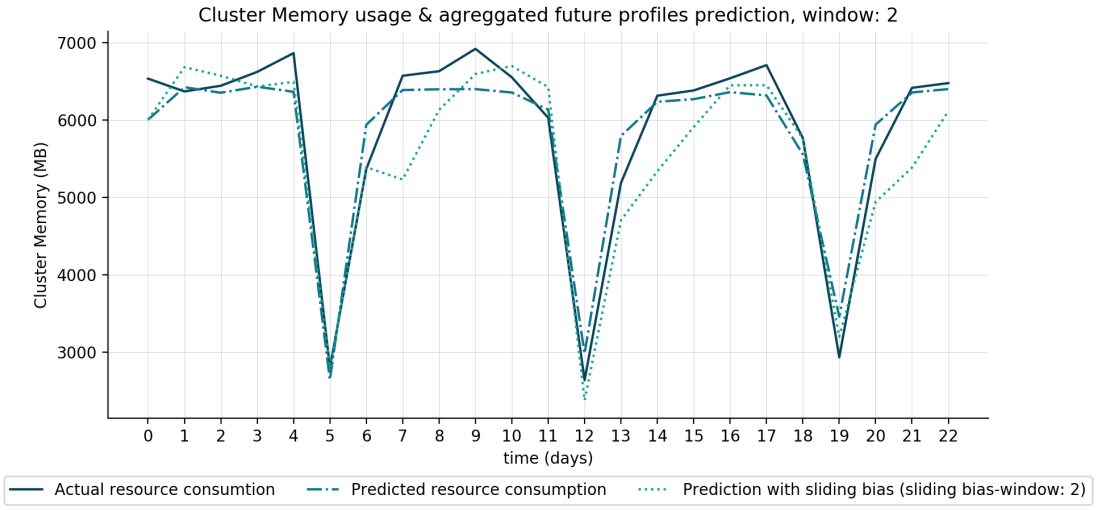

Figure B.6: Prediction Test Cluster.

Listing B.1: API interface definitions

```
IEntry = {
    id: int,
    timestamp: int,
    memory: int,
    cpu: int,
    cpu_label: int,
    memory_label: int
}

IRelease {
    version: string,
    entries: IEntry[]
}

IService {
    serviceId: string,
    releases: IRelease[]
}

ICollectorResponse {
    services: IService[]
}
```