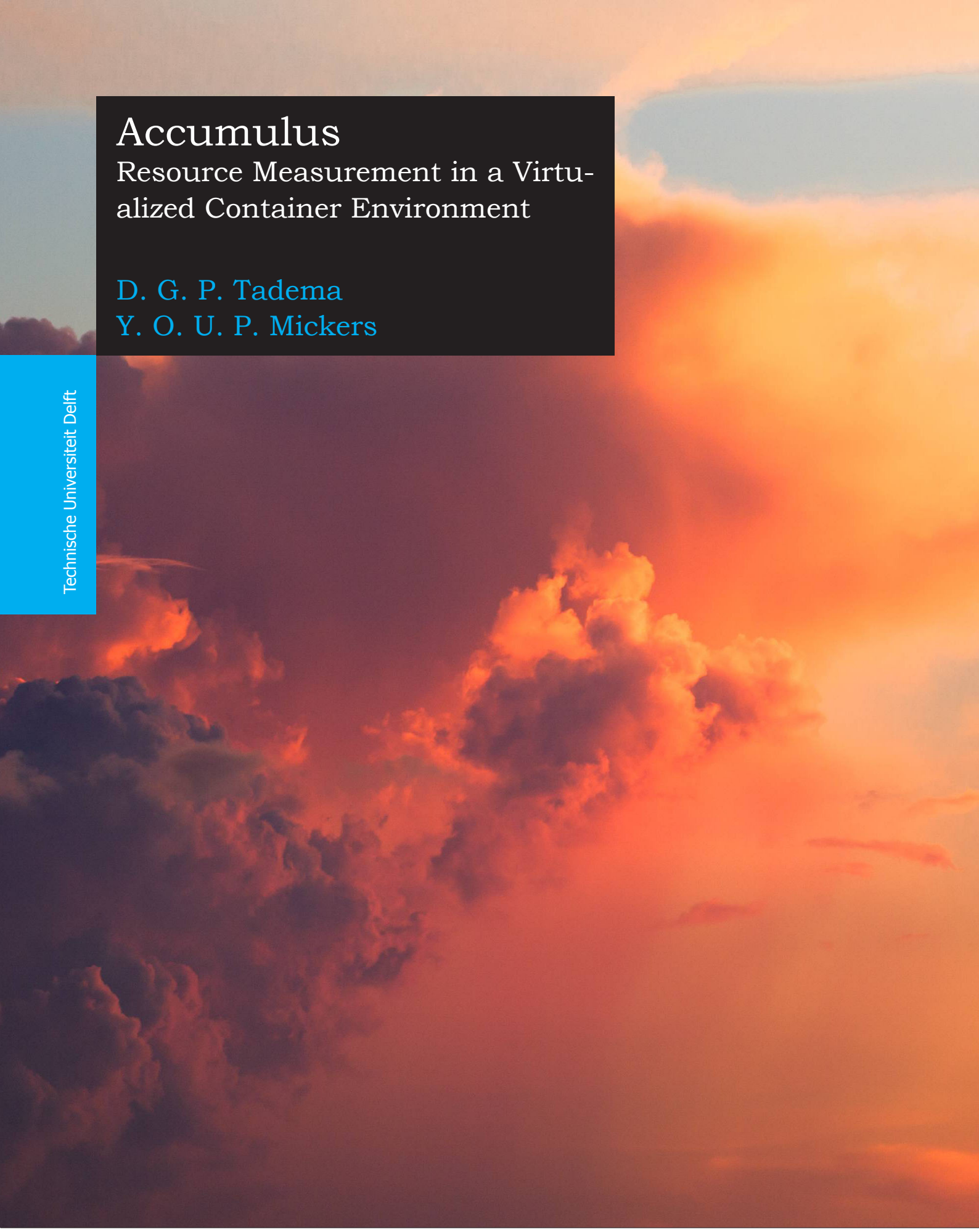


Accumulus

Resource Measurement in a Virtualized Container Environment

D. G. P. Tadema
Y. O. U. P. Mickers

Technische Universiteit Delft

TU Delft Delft University of Technology

Challenge the future

Version	Date	Reviewers	Remarks
0.1	22 December 2016	Authors	First draft (no complete report yet).
0.2	5 January 2017	Authors	Version created for Nerdalize internal use.
0.3	11 January 2017	Authors	Version handed in for revision.
0.4	11 February 2017	dr. ir. A. Iosup	Incorporated first feedback by dr. ir. A. Iosup.
0.5	25 February 2017	dr. ir. A. Iosup	Second hand-in for revision.
1.0	6 March 2017	dr. ir. A. Iosup	Final report.
1.1	6 March 2017	dr. ir. A. Iosup	Final report (minor improvements).
1.2	7 March 2017	dr. ir. A. Iosup	Final report (minor improvements).

Table 1: Versioning.

Accumulus

Resource Measurement in a Virtualized Container Environment

by

D. G. P. Tadema
Y. O. U. P. Mickers

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Friday March 10th, 2017 at 12:00 AM.

Supervisor: dr. ir. A. Iosup TU Delft
dr. M. de Meijer, Nerdalize

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

Acknowledgements

This thesis would not have been possible without the help of many people. A few of those people we would like to mention in particular, to show our appreciation of their support throughout the process. First of all Alexandru Iosup for his guidance and feedback on matters both conceptual and otherwise. We thank Mathijs de Meijer for his excellent mentorship and helping us deal with hurdles along the way. Ad van der Veer for his help with the Go programming language. All employees at Nerdalize for creating an inspiring environment conducing creativity. A final thanks to everyone at the Distributed Systems group and the Nerdalize employees for their suggestions after the trial presentations.

D. G. P. Tadema
Y. O. U. P. Mickers
Delft, March 2017

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Problem Statement . . . 1
- 1.3 Approach . . . 2
- 1.4 Main Contributions . . . 2
- 1.5 Structure . . . 3

2 Background **4**
- 2.1 Concepts . . . 4
- 2.2 Technologies . . . 5

3 Problem Analysis **9**
- 3.1 Problem Definition . . . 9
- 3.2 User Stories . . . 10
- 3.3 Requirements . . . 10

4 The Research Process **13**
- 4.1 Research . . . 13
- 4.2 Development Tools . . . 14
- 4.3 Systems . . . 16
- 4.4 Chosen Development Tools . . . 18
- 4.5 Chosen Systems . . . 19

5 The Development Process **21**
- 5.1 SCRUM . . . 21
- 5.2 Go . . . 22
- 5.3 Continuous Integration . . . 22
- 5.4 Unit Testing . . . 22
- 5.5 SIG Code Review . . . 22

6 Design of Accumulus, a Monitoring and Cluster Analysis System **24**
- 6.1 Architecture Overview . . . 24
- 6.2 Accumulus . . . 25
- 6.3 SQL Mechanism . . . 26
- 6.4 Heapster . . . 28
- 6.5 Influx . . . 28
- 6.6 Processors . . . 29
- 6.7 Grafana . . . 29
- 6.8 Possible Extensions . . . 30

7 Experimental Testing and Validation **32**
- 7.1 Accuracy Evaluation . . . 32
- 7.2 Basic Validation . . . 35
- 7.3 Overhead . . . 38
- 7.4 Scalability Evaluation . . . 39

8 Discussion & Future Work **47**
- 8.1 Discussion . . . 47
- 8.2 Future Work . . . 48

9 Summary & Conclusion **51**
- 9.1 Summary . . . 51
- 9.2 Conclusion . . . 51

A Sprint Plans **53**
A.1 Sprint 1 54
A.2 Sprint 2 55
A.3 Sprint 3 56

B Research Report **57**

C Project Proposal **101**

D Requirements **106**

Bibliography **108**

1

Introduction

1.1. Context

Computers are used almost everywhere in today's world, and since the 2000's we have seen a shift from computations done on a local level to computations done in centralized data centers. A recent development is the shift from grids and data centers to clusters, where thousands of computers are connected via a network and can work together on a task [1].

Computers have certainly enabled us to solve problems previously deemed infeasible and improved our lives in countless ways, but their increased usage comes at a cost. At the moment, estimates hold data centers accountable for 2-3% of the worldwide energy consumption. A number that is expected to grow significantly in the coming years, with data centers surpassing the airline industry in CO2 production [2]. As such, it is vital to make the use of computation power more sustainable.

One of the companies that are trying to achieve this is Nerdalize, who uses the heat produced by servers to provide houses with warm water and central heating. In participating households, the boiler will be replaced by a computing unit developed by Nerdalize. Such a decentralized cloud has lower operational cost than centralized competitors as the high cost of housing and cooling are mitigated [3].

Nerdalize is also trying to solve other issues in cloud computing. One of these problems is vendor lock-in, which it wants to solve by combining their cloud with other clouds to create a so-called multi-cloud, a single heterogeneous architecture that uses multiple infrastructure providers. A multi-cloud solves the issue of vendor lock-in, while also addressing geo-diversity [4].

While doing this Nerdalize is also coming up with solutions for the relative intransparent pricing models cloud providers currently offer. Cloud costs are hard to estimate beforehand, which is potentially scaring away customers. Earlier research has partly alleviated this problem [5], but billing is currently still done on a per-instance basis, instead of on the actual resources used.

1.2. Problem Statement

Currently, most cloud providers bill their customers for the resources and instances they reserve or based on the number of times their function is called and the time taken by each called function [6]. These pricing models, however, vary between providers and machines as shown in Figure 1.1, and it is hard to forecast performance on a specific instance. As a result, the market for cloud computing is far from transparent. Nerdalize aims to make cloud computing more of a commodity. One aspect of this is how customers are charged. In the future, Nerdalize wants to shift from a per instance to a per job billing model. New problems arise in such a billing model, resource usage needs to be accurately tracked and be available to the user to get insight into the performance and cost of their job.

To charge customers on a per job basis a system is needed that accurately measures the resources used by a job with minimal overhead. While this task is relatively easy on a single computer, determining

these metrics for a job which is split across multiple computers (each of which might also be working on several other jobs at the same time) brings several complications. Some research has already been done in this field [7–10]. This has lead to some great systems for resource monitoring, but they do not meet Nerdalize's requirements. Most of them are not customer oriented, nor do they take into account the power used by the servers performing the jobs. The developed system should also work across multiple clusters and should be able to specify on which hardware the resources were used. Finally, the system should also be scaleable, as it will eventually work with multiple clusters, each containing many nodes.

A system that

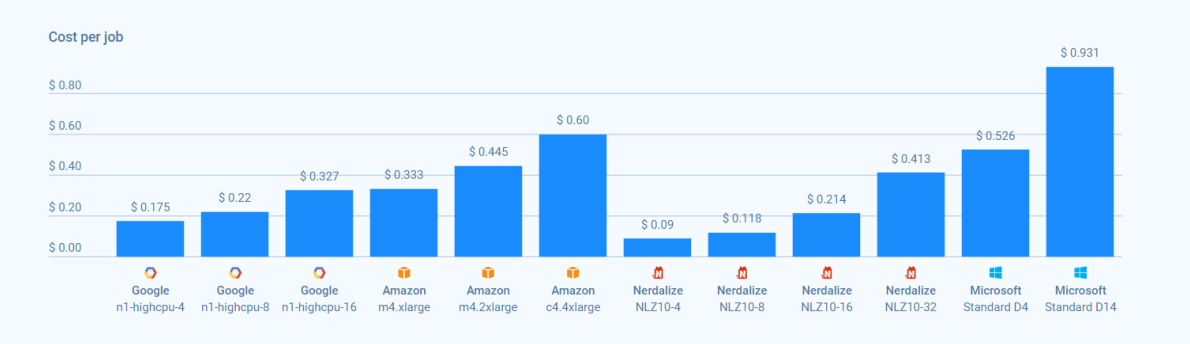


Figure 1.1: Pricing of different providers [11].

1.3. Approach

The goal of this project is to create a system that can accurately measure resource use and estimate the accompanying costs in a container-based cloud system and make this information available to both the user and the cluster service provider. The development will start with two weeks of research, during which will be researched how cloud providers calculate costs and how resource usage can be accurately measured, especially in set-ups relying on Docker and Kubernetes. During the research, a preliminary design for a system will also be created. The five weeks after the research period will be used to design and implement a system that can help Nerdalize and others to accurately calculate the resource usage and thus the costs of jobs on a cluster infrastructure. The last three weeks of the project are used to test and validate the built system and write a thesis report.

One of the resource monitoring systems that was found during the research phase, Heapster, is already focused on Kubernetes clusters and will therefore be taken as a starting point. It will be extended with the desired functionality, such as the monitoring of resource consumption across multiple clusters and adding tags to the consumed resources specifying the hardware on which they were used.

1.4. Main Contributions

In this thesis, we present Accumulus, a system for cluster resource usage monitoring. Accumulus consists of multiple components, The main component that collects information from all the clusters, and a software stack to collect the metrics within the individual clusters.

1. A dynamic tagger for Heapster that allows additional tags to be added to measurements via an SQL database.
2. A package written in Go () to work with time series database that offers concurrent querying, writing, and syncing between different instances of a time series database.
3. Accumulus, a system to do multi-cluster monitoring of Kubernetes clusters.
4. Processors that can combine metrics and be used to calculate combined metrics such as power consumption and operational cost.

5. An evaluation of Heapsters accuracy and the scaling capabilities of Accumulus. Experimental work validates the Accumulus system and shows it can scale to clusters containing over a thousand nodes.

1.5. Structure

This Report is structured as follows: Relevant concepts found in cloud computing are explained in Section 2.1, Section 2.2 gives an overview of technologies used in the field of cloud computing and. Chapter 3 presents the main problem and high-level goals of the project. Section 3.3 defines a set of requirements created from the obtained user stories, which a system solving this problem has to fulfill. Chapter 4 describes our research process. The research done during the project is described in Section 4.1. Section 4.2 lists considered development strategies, programming languages and tools. The choices are explained and supported in Section 4.4. many existing solutions for resource monitoring already exist; Section 4.3 goes into detail and Section 4.5 explains which system was chosen as a starting point, also explaining this choice. Chapter 5 then goes into detail on how the development strategies and tools selected in Section 4.4 were applied during the implementation. Chapter 6 covers the design and implementation of our system. And explains the role of each element. Chapter 7 shows how the system will be tested and validated, and the results of those experiments. The implications of our system and future work are discussed in Chapter 8. Lastly, Chapter 9 summarizes this thesis and gives a final conclusion.

2

Background

This chapter is used to present background information related to the Accumulus project. In particular desired prior knowledge about concepts and technologies is described in Section 2.1 and Section 2.2 respectively.

2.1. Concepts

This section defines some of the concepts in the field of distributed computing.

2.1.1. Virtual Machine

A virtual machine is a machine level virtualization method that provides an isolated environment that simulates a different system or architecture. This allows a single machine to run multiple different operating systems [12].

2.1.2. Container

A container is an operating-system level virtualisation method that provides an isolated environment, simulating a closed system running on a single host. It gives the user the ability to have an environment to run applications with the necessary resources and environment configuration.

Containers are comparable to VMs (Virtual Machines), in that they are meant to decrease the problems often associated with deploying software on different servers or computers. They both do this by packaging all dependencies in a single environment, which can then be deployed across several computers.

The main difference is that VMs simulate a complete OS (operating system), with each VM having its own kernel, filesystem, and virtual hardware. Containers, on the other hand, run on a shared OS and Kernel, as seen in Figure 2.1. Each container has its own isolated userspace but they share system resources. As a result, containers are often much smaller (MB's compared to GB's) and much faster to start-up [13].

2.1.3. Cloud

A cloud is a group of computers that work together connected by a network, usually acting as a single system or offering a single service. Users do not directly control the hardware, but instead rely on third-parties to fulfill their computation needs. The complex back-end of a cloud is often hidden from users and managed by cluster orchestration software such as Kubernetes or Openstack. [15]

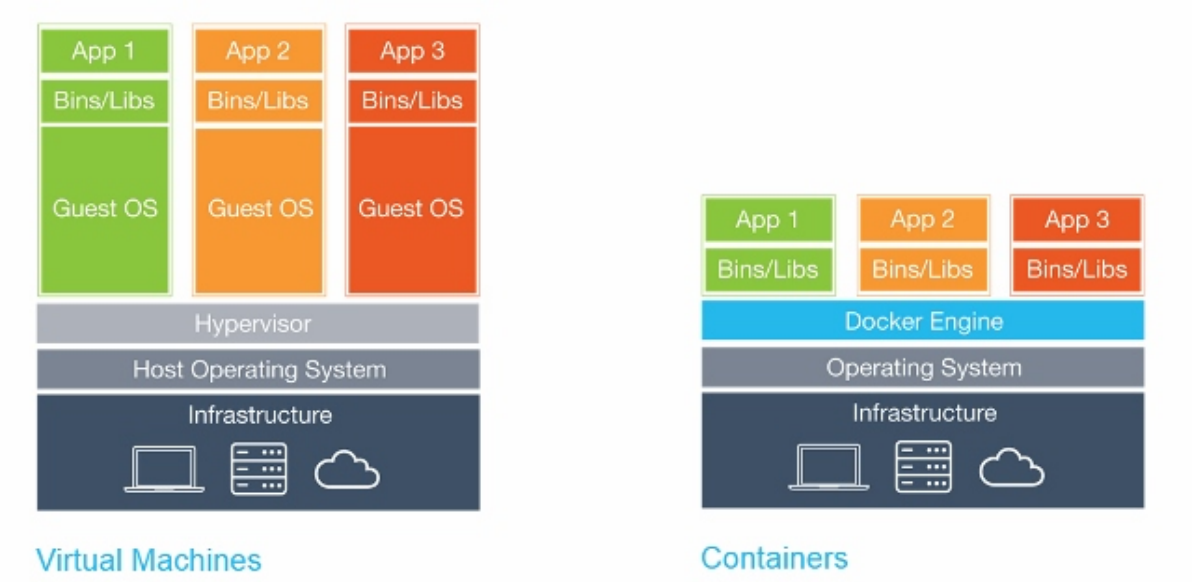


Figure 2.1: VM's versus Containers [14].

2.1.4. IaaS

Infrastructure as a Service is one of the service models used in cloud computing. Just like other cloud service models, IaaS offers access to computing resource in a virtualised environment. In the case of IaaS, the computing resource provided is specifically that of virtualised hardware, in other words, computing infrastructure. This includes virtual server space, network connections, bandwidth, IP addresses and load balancers. [16]

The cloud provider is responsible for maintaining the hardware infrastructure. The client, on the other hand, is given access to the virtualized components in order to build their own IT platforms.

2.2. Technologies

In this section technologies used in cloud computing are explained.

2.2.1. Docker

Docker is an open-source container implementation. It offers both Linux (CoreOS) and Windows containers. Docker utilizes resource isolation via Linux kernel isolation technologies (cgroup and namespaces). They provide a great isolation for many applications. However, they do pose some risks in the case of multi-tenant environments. All containers on the same host share the same Linux kernel with that host [17].

Nerdalize will tackle this problem by creating a cluster for each customer, this way customers will never be able to access data belonging to other customers, thus ensuring security.

2.2.2. Kubernetes

Kubernetes is an open source system for automating deployment, scaling, and management of containerized applications or jobs on multiple hosts [18]. Kubernetes offers scheduling, replication control, and load balancing. It uses a state aware replication controller to handle failing machines or applications.

Nerdalize offers a cloud service for running compute-intensive workflows, with Kubernetes being used to manage the containers that make up such a workflow. Their in-house built workflow scheduler Flower [19] is used to ensure the workflow is executed in the right order and minimal heat is wasted.

A short description of several important Kubernetes concepts follows now and an overview of how these work together is shown in Figure 2.2.

Node

Nodes are the workers in a Kubernetes cluster, as they execute the computations assigned to them. A node cannot consist of several computers, although a computer can host several nodes if desired.

Pod

Kubernetes works with pods, with a pod being an atomic unit containing one or more tightly coupled containers. Pods cannot be split over multiple motherboards as the containers in a pod is run on a single node.

Service

A pod is mortal, as they can be terminated by the scheduler if deemed necessary. As such, they cannot be relied on by external processes. The solution is found in a service, which is offered together by a collection of pods, thereby providing a reliable interface for communication [20].

Job

A job is the terminating counterpart of a service. A job creates one or more pods and ensures that a specified number of them successfully run. As pods successfully complete, the job tracks the successful completions. When the specified number of completions is reached, the job itself is complete. Jobs are useful for running workloads and batch jobs.

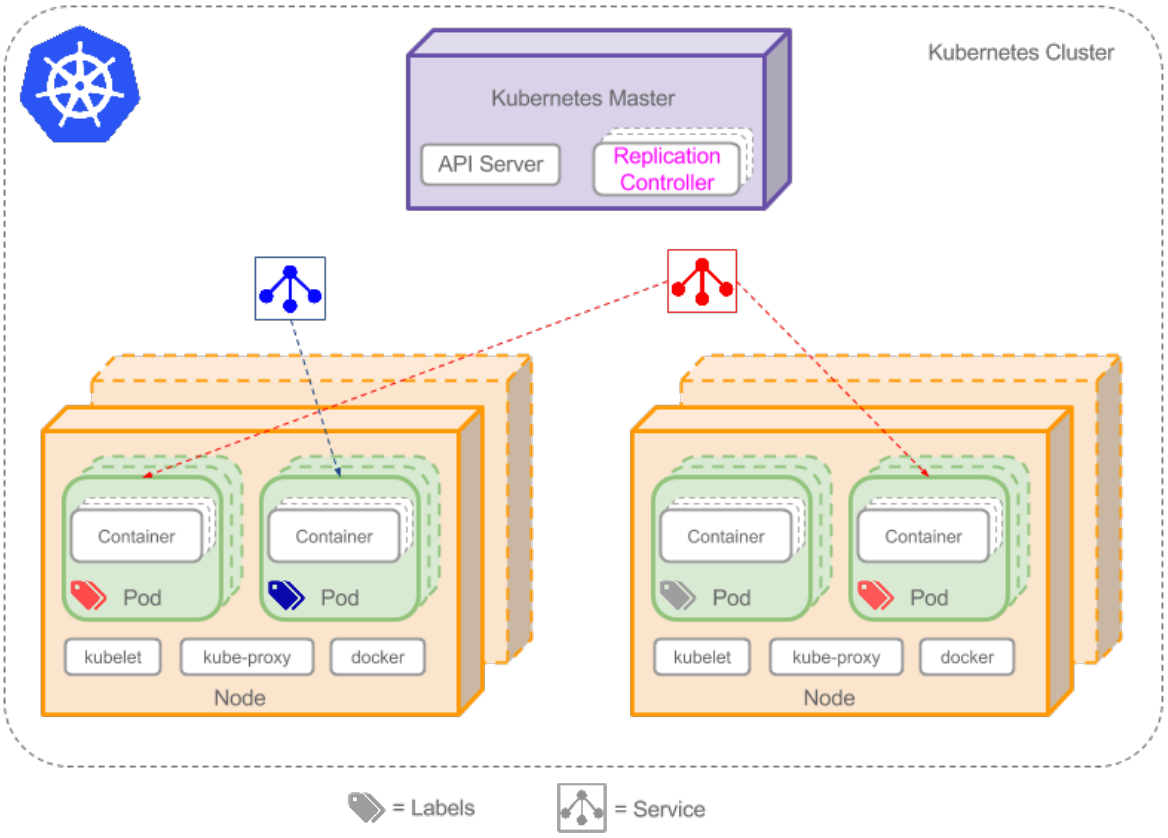


Figure 2.2: A Kubernetes cluster.

Resource Requests and Limits

Kubernetes' scheduling is affected by two metrics, being the amount of CPU and memory. Aside from these metrics, cluster owners can use "Opaque Integer Resources" to define custom metrics. This is still in alpha as of yet. Kubernetes has notions of limits and requests for all metrics, with limits and requests being set per container. The resource limits and requests of a pod are easily determined, by simply summing those of all underlying containers.

Resource requests come into play before a pod is run and determine where the pod will be run. Whenever a pod is created, the scheduler will look for a suitable node for the pod to run on. Suitable here is defined as having the amount of available CPU and memory exceed the amount requested.

Resource limits, on the other hand, are important during a pod's execution. Whenever the memory used by a container exceeds the limit set, the container will be terminated. Whether the container will be restarted is determined by the "restartable" flag of the container. A container may or not be allowed to exceed the CPU limit set for extended periods of time, but will not be terminated for exceeding it. It follows naturally that a container's resource limit must exceed its resource request [21].

(Persistent) Volumes and Persistent Volume Requests

On-disk files within a container cannot be shared between containers and are not durable. As such, they are lost whenever the container crashes. A solution to both these problems is found within Volumes, whose lifetime is equal to the pod enclosing them. A Volume subsystem provides storage to a single pod, independent of the underlying storage systems used, which are often vendor specific (e.g. Azure's file system or AWS elastic block store).

Volumes still seize to exist whenever their owning pod ends. As such, they are not suitable for sharing data within a job or even to save the output data. For this, a PersistentVolume is needed, which again provides storage independent of the underlying storage system used.

A PersistentVolumeClaim is the storage equivalent of a pod. Where a pod requests and consumes computing resources from the available nodes, PersistentVolumeClaims consume storage on PersistentVolumes [22].

2.2.3. Go

Go is a programming language developed by Google in 2009 [23]. It is chosen as the programming language for this project mainly because both Docker and Kubernetes are written in it and because several employees of Nerdalize are already well-versed in it. For a more extensive motivation, we would like to refer you to Section 4.4.

2.2.4. Cloudbox

The Cloudbox is developed by Nerdalize and serves as a housing for the computers, offering both physical security and ease of maintainability. It will be placed in people's homes and will provide hot water to the homeowner while providing compute for Nerdalize's customers. Inside the Cloudbox is space for 3 compute modules (initially one slot will be kept empty), each featuring 2 motherboards. The motherboards have 2 CPUs with 10 cores each, giving the Cloudbox a maximum of 120 cores.

2.2.5. NCE

The Nerdalize Cloud Engine is the main product Nerdalize is building, it will be a platform that allows engineers and users to communicate with Nerdalize's cloud infrastructure. It will eventually offer three ways to communicate with the Nerdalize cloud, a command line interface, an API, and a graphical user interface. With this Nerdalize aims to make running jobs as simple as possible. The NCE will handle authentication, data set storage, queueing of jobs, and control over workflows.

2.2.6. Heapster

Heapster is an open source tool for Container Cluster Monitoring and Performance Analysis [24]. It is written in Go and is compatible with Kubernetes version 1.0.6 and up. Heapster measures various resource statistics on Container and Pod level and provides aggregate statistics for node and cluster levels. Those statistics can be stored in various back-ends, with the default being InfluxDB.

3

Problem Analysis

This chapter creates a definition of the problem Nerdalize has. From this definition, a problem statement is derived and the main question which will be answered during the project is formulated. Section 3.2 lists user stories which give a description of the system from the end users perspective. In Section 3.3, requirements are defined, categorized, and ranked according to the amount of priority they deserve.

3.1. Problem Definition

Nerdalize is building its main product, the Nerdalize Cloud Engine (NCE). Their Cloud strives to be competitively priced, which they want to accomplish by placing their servers in homes with central heating. By cooling the server and reusing the heat to heat the home there is no need for expensive air-conditioned server space.

They also strive to make cloud computing more of a commodity market. To reach this goal they want to be more transparent in how they charge customers, this is done is two ways. First, Nerdalize wants to give the customer more insight in their resource usage, second, they want to charge their customers on a per job basis. In order to do this, they need a system that accurately measures resource use in a cluster. Resource measurement on a single system is trivial, but jobs on cloud services are run on a great variety of machines. Nerdalize's case has the additional difficulty of their machines being installed in houses where they are not easily accessible.

The high-level goals of this project are:

1. Research how to measure resource usage on a Kubernetes cluster.
2. Research the current resource measurement systems Kubernetes offers (Heapster).
3. Build a system to measure and aggregate the resource use of an NCE job.
4. Extend the system to include the heat and power usage statistics of Cloudboxes.
5. Design an API to make statistics available to users and the Nerdalize system.
6. Design a simple GUI that shows these statistics.
7. Validate the built system against its requirements

From this problem definition, we are able to derive a problem statement. The statement is based on the problem definition by Nerdalize and the academic requirements specified by TU Delft. The derived statements differ from the original definition in that they are more concise and more abstract, omitting implementation details.

1. How can the resource usage of a job running on a container-based cloud be measured?
2. Can power consumption and heat production be measured on container or pod level?
3. Can charging users on a per job basis make pricing more transparent?

3.2. User Stories

An interview with Nerdalize uncovered three actors

- **Nerdalize Compute Billing**
  Financial department that needs billing information to send invoices to customers.
- **Nerdalize Compute Sales**
  Sales department that charges customers for compute resources with a particular pricing model.
- **NCE User**
  Users of the Nerdalize Compute Engine

For each actor several user stories are given, for consistency they are all in the form of As <actor>, I want to <function or requirement> in order to <reason>

1. As the **Nerdalize Billing Dept.**, I want to **have data on resource usage** in order to **bill the customers**.
2. As the **Nerdalize Sales Dept.**, I want to **have data on resource usage** in order to **implement my billing models**.
3. As the **NCE User**, I want to **read accurate resource consumption per cluster, namespace, label, node, and container** in order to **see my resource usage (and thereby expenses)**.
4. As the **NCE User**, I want to **read accurate resource consumption per cluster, namespace, label, node, and container** in order to **have feedback on the performance of my jobs thus allowing me to optimize for runtime or cost**.
5. As the **NCE User**, I want to **read accurate predicted resource consumption per cluster, namespace, label, node, and container** in order to **see my expected resource usage (and thereby expenses)**.

3.3. Requirements

In this section, the requirements distilled from the user stories and the interviews are enumerated and prioritized.

3.3.1. Functional Requirements

Based on the collected user stories we created the following functional requirements. They are prioritized using the MoSCoW model

Must Haves

1. The system measures accurately and with an overhead that is 10% or less.
2. The system can run without any manual interference needed.
3. The system measures the usage, request, and limit of all required metrics.
4. The system records the machine and job metrics belong to.
5. Measurements are retrievable via an API.

Should Haves

6. The system measures the usage, request, and limit of all optional metrics.
7. The system differentiates between internal peering and outward internet network traffic.
8. Local measurements are sent at short intervals in order to analyze resource usage on failing nodes.
9. The system uses retention policies to reduce storage usage.
10. Collected measurements are stored centrally for historic and billing purposes.

Could Haves

11. The system also measures network traffic between Cloudboxes.
12. The system can make predictions on resource use of a job that is running.
13. Resource usage is not only accessible via the API but can also be viewed in a GUI.
14. The system generates a report when a job is finished.

Won't Haves

15. The system is integrated with the Nerdalize Cloud Engine's user interface.

3.3.2. Metrics

For the purpose of clarity, the used metrics are explained in-depth here instead of in the MoSCoW-model.

For every pod, it is also important to register the instance on which it was used. This is to be able to determine for example whether RAM usage was DDR3 or DDR4 and on what type of CPU the CPU time was used. For more advanced cost analysis nodes could even indicate whether the energy they received was useful to them (i.e. was the heated water likely to be used soon). This draws interesting parallels with an earlier research project done at Nerdalize [19].

CPU Time Usage

It should be known how much CPU time is spent on a certain job. A distinction should be made between CPU time reserved, and actual time used (as a reserved CPU does not consume energy and therefore has a different cost profile).

RAM Usage

Both the reserved RAM and the total amount of RAM used should also be measured. Contrary to the CPU use, reserved RAM is equivalent to RAM actually used, as reserved RAM is unusable by other jobs and does not differ in any way.

Disk I/O and Space

For disk usage, it is important to know both the total amount written and read from the disk (I/O) and the total amount of space reserved/used on the hard drive by a job. A separation should be made between local disk usage and central disk usage.

Network Traffic

In network traffic, three separations can be made. First of all, there is the traffic inside the Nerdalize network (e.g. Cloudboxes' local drives receiving their data from the central server or Cloudboxes peering data to each other). Secondly, there is the traffic through the Internet Exchange, where internet companies can send their data directly to each other by using the Border Gateway Protocol instead of the "regular" internet (e.g. exchanging traffic with selected cloud providers). Finally, there is the normal internet traffic, which would be the most expensive form.

Uptime

It is also important to know the total uptime of a machine, this allows users to see when machines are added or removed from their cluster and allows Nerdalize to see when Cloudboxes go offline. Measuring this metric also allows Nerdalize to refund resource usage on a failed Cloudbox.

Heat (optional)

Nerdalizes Cloudbox produces heat that can be used to warm houses. Although currently not planned it is a possibility that in the future Nerdalize may want to charge households for generated heat. In this case, it is important to have statistics on how much heat a job produced and how much of this heat is used. Nerdalize may then chose to discount jobs which created useful heat.

Privacy-related issues require special care here. From the test setup in Nerdalize's office building, it already became apparent that possibly sensitive information could be retrieved from the temperature measurements.

Power (Optional)

Like heat, power is another real-world factor participating and should be taken into account in the cost analysis. Privacy is less of an issue here, as power consumption is dictated by computation need and to a lesser extent by the homeowner. However, power consumption could still reveal heat consumption to some extent, therefore requiring some care.

3.3.3. Non-functional Requirements

Apart from the functional requirements, the interviews have uncovered several non-functional requirements

Performance

The developed system may only impose a very small overhead on the jobs that need to be run in the cloud. If the overhead is too large, the system effectively becomes useless, as it will introduce a computational, and therefore monetary burden far outweighing the benefits.

Accuracy

A measuring system is only useful if its measurements are accurate. In Nerdalize's case, this is defined as having measured values deviate less than 10 percent from the actual value. For a more in-depth discussion about accuracy, we would like to refer you to chapter 6.

Open Source

The developed system or parts of it could be available as open source. This is required to collaborate with (experienced) people from the Kubernetes community who may help out with difficult design decisions and provide valuable feedback. It also allows other people to report bugs in the system and collaborate code or documentation speeding up development. The most interesting part for this approach will be the resource measuring part of our system as it will most likely be based on the open source software Heapster.

Maintainability

Maintainability measures the ease with which developers are able to make changes to the system. This is important because when the project is finished, less time is available to maintain the system. If it is easily maintainable and open source, other developers are able to take over without having to extensively study the code. This is beneficial to Nerdalize and other users of the system as it is more likely kept up to date.

Flexibility

As the field in which Nerdalize is operating is relatively new, it is bound to change quickly. Both Kubernetes and Docker are under heavy development, in which the direction that will be taken is not always clear yet. As such, the entire project needs to be setup in such a way that any changes can be conducted easily and without requiring major revisions to the code.

4

The Research Process

This chapter is used to describe the research process. In particular, it discusses how the the research was performed in Section 4.1, which development tools were considered in Section 4.2, and which already existing systems were considered in Section 4.3. Section 4.4 presents the chosen development tools, while Section 4.5 presents the chosen systems.

4.1. Research

In this Section, the research processes that were used to gather necessary information for the design and implementation of our system are described. Subsection 4.1.1 and Subsection 4.1.2, which respectively describe the processes before the project started and the processes during the project.

4.1.1. Before the Project

Two weeks before the official start of the project a meeting with Nerdalize took place in which the subject of the project was decided. Mathijs from Nerdalize explained several of the technologies in use at Nerdalize and that the would need to get familiar with. None of the project members had any prior knowledge of the container ecosystem and distributed systems in general. To get familiar with Docker two books were read.

Kubernetes is the system Nerdalize uses to manage their clusters, to get familiar with Kubernetes books were read and a tutorial provided on the Kubernetes website was followed.

Go is the language Docker and Kubernetes are written in and is also used by a lot of Nerdalize's systems. Videos were watched about Go and tutorials were done to get a better understanding of Go and its concepts.

4.1.2. During the Research Phase

Most of the research was done in the first two weeks of the project as a clear view of the project was needed. In this research phase and throughout the project multiple types of research were used: interviews, literature studies, digital media, and running test systems.

Interviews

During the project multiple interviews with different people from Nerdalize took place. Those interviews helped to get a better view of the requirements for the system and getting familiar with how Nerdalize organizes their cloud infrastructure. An overview is shown in Table 4.1.

In an interview with Mathijs, the planned structure and architecture of the Nerdalize Cloud Engine was discussed and all systems in the NCE were explained. This showed where our system would be and how it should integrate.

A second interview with Thirza from the Nerdalize billing department was more focused on what the billing department of Nerdalize needs and how our system can provide this information. This helped with selecting a set of metrics for the system.

The third interview with Boaz from Nerdalize's sales department gave insight in how Nerdalize wants to sell cloud computing and which data is necessary for doing so. This interview also sheds light on how usage date could be shown to the end user which is useful when designing a GUI for the system.

In another interview, Nerdalizes network engineer helped define the network metrics that are available and how they could be measured. The conclusion of this interview was that not all initially selected network metrics can be measured, thus a smaller set of metrics was selected for the project.

A second interview with Mathijs showed the preliminary architecture design. This allowed flaws to be discovered and improvements to be made. The improved preliminary architecture design addresses issues with data security and moved more functionality to a central core.

Name	Specialization within Nerdalize	Experience
Mathijs	CTO	15 years of experience with software engineering 4 years with cloud computing
Thirza	Billing	2 years of billing experience
Boaz	Sales	5 year
Tim	Network	10 years of networking experience 8 of which in data centers

Table 4.1: People interviewed throughout the project.

Literature Study

A literature study was done for two main reasons. Background knowledge on container infrastructure and clusters had to be acquired. Sources for this information were scientific papers and books. Furthermore, related work had to be surveyed to find ways in which problems have been solved before by existing resource measuring systems. Sources for the related work were project homepages and repositories.

Digital Media

The featured videos on Go's homepage were used to get familiar with some of Go's concepts such as subroutines and reflection. Other forms of digital media used were slide shows and online articles.

Running Test Systems

During the last week of the research phase of this project, a local Kubernetes cluster was set up with Minikube [25]. This cluster was used to get a better view of how Kubernetes works and which features are already offered by Heapster. This was extremely useful as it helped us understand how Heapster and InfluxDB work together (InfluxDB working as a sink for Heapster with Heapster pushing the metrics), which metrics are collected and which are missing from our system. Later in the project a Kubernetes cluster will be set up on Google cloud this cluster was used for deployment, testing and debugging the system

4.2. Development Tools

In this section, several development tools are described and compared. The eventual choices are detailed in Section 4.4.

4.2.1. Development Strategies

In this section, different development strategies that were considered for the project are explained. For the development strategy, there are two requirements. The development strategy needs to be

able to cope with requirements volatility as Nerdalize is still developing their main product and not all requirements are clear yet. Second, the development strategy needs to work well for a system that consists of many decoupled software products.

Waterfall

The waterfall model is a sequential design process, used in software development processes, in which progress is seen as flowing steadily downwards through the phases of conception, initiation, analysis, design, construction, testing, production, implementation and maintenance. Despite the development of new software development process models, the Waterfall method is still a dominant process model.

SCRUM

Scrum is both an iterative and incremental software development framework. It defines a flexible development strategy.

A key principle of SCRUM is requirements volatility. Scrum recognizes that the customers can change their minds about what they want and need, and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an evidence-based empirical approach. It accepts that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly, to respond to emerging requirements and to adapt to evolving technologies and changes in market conditions.

XP

Extreme programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity, and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers. The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels.

4.2.2. Programming Languages

In this section, some of the programming languages that were considered for the project are outlined. As a monitoring system needs to run in real time, one requirement for the programming language is performance. A second requirement is that the language has to offer tools to run concurrent, as the system will need to process data of multiple clusters at the same time.

C(++)

Ever since its appearance in 1983, C(++) has been used in an incredible number of applications. Its low-level nature with corresponding memory management makes it ideal for performance-critical and real-time purposes. However, despite several revised standards, the language is starting to show its age and can be considered verbose, meaning it the amount of code needed is often high compared to implementations in other languages.

Go

Go is an open source programming language developed by Google. It offers a modern language with interesting concepts such as channels and concurrency and an innovative approach to interfaces and reflection. It can be considered a good fit for this project due to the fact that both Kubernetes and Docker are written in it. Grafana, Prometheus (partly), InfluxDB and Chronograf are also written in Go, enabling easier integration.

Python

Python is a high-level interpreted dynamic programming language. While most implementations tend to be slow compared to compiled programming languages, developing applications is often much faster, making it ideal for rapid prototyping. Being one of the most popular languages available today, there is a very large and active community, with a wealth of libraries available. Graphite and the Graphite webapp are both written in Python.

4.2.3. Repository

In order to collaborate on the code, a repository with version control is needed to ensure functions that are developed separately can be integrated with minimal effort. Multiple solutions were considered.

GitHub

GitHub is the most well-known repository service with over 14 million users. It offers a Git based distributed version control and has several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

GitLab

GitLab is an open source repository manager, like Github it offers collaborative features but unlike GitHub, it has an integrated build system, offers free private repositories, and can be self-hosted.

4.3. Systems

There are already many systems to provide some of the functionality a resources monitoring system needs. In this section, the commonly used systems in distributed computing are outlined.

4.3.1. Data Retrieval and Storage

Data retrieval and storage are considered together, as most systems offer them in an integrated fashion.

Graphite

Graphite is a monitoring tool to store numeric time-series data, offering an SQL-like language for querying the stored data. It consists of Carbon, a service for collecting time-series data, Graphite-web, offering a user interface and a possibility to render graphs and Whisper for storing data. Besides Whisper (a local file-based time-series database), data can also be stored in Ceres (a distributable time-series database) or even InfluxDB.

InfluxDB

InfluxDB is an open-source time series database developed by InfluxData and written in Go [26]. Data is grouped as measurements, which are comparable to tables in traditional relational databases. A measurement consists of a range of timestamps, each holding multiple key-value pairs. Some remarks can be made about the relative non-scalability of the system (horizontal scaling is only possible in the commercial version), something that could cause problems in the future. However, as InfluxDB is currently one of the fastest-growing databases, we expect that solutions have emerged from the community by that time.

Prometheus

Prometheus is an open-source time series database originally developed at SoundCloud, largely written in Go. It "works well for recording any purely numeric time series. It fits both machine-centric monitoring as well as monitoring of highly dynamic service-oriented architectures". However, "If you need 100% accuracy, such as for per-request billing, Prometheus is not a good choice as the collected data will likely not be detailed and complete enough." [27].

Ganglia

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters. It leverages widely used technologies such as XML for data representation and RRDtool for data storage and visualization. It is currently in use on thousands of clusters around the world and can scale to handle clusters with 2000 nodes [28]. Ganglia is however not meant as a user-facing system, and even though it is aimed at clusters, it does not work well in a containerized environment.

Nagios

Nagios is open source software for monitoring systems, networks, and infrastructure. It saw its first release in 1999 and is still actively developed. Nagios is however not aimed at clusters [8] and as such the setup process for Kubernetes would be complex. For this reason, other options are preferred.

MonALICA

MonALICA is a globally scalable framework of services to monitor and help manage and optimize the operational performance of Grids, networks and running applications in real-time. However, since it is aimed more at operational performance than resources utilization measurement, other options are preferred [29].

Relational Database Management Systems (RDBMS)

Besides time series databases, several RDBMS with SQL-support (MySQL, SQLite, and PostgreSQL) were also considered. Despite not being tailored to time series, which will form the majority of the data and therefore dictate most of our needs, they are true and tested systems that have been applied to solve largely divergent problems.

4.3.2. Data Visualization

Having a GUI (and therefore data visualization) is a could have of the system and it is still unclear in the research phase whether the project will eventually feature one. Nevertheless, it is good practice to already take into account possible techniques and frameworks in order to simplify eventual implementation, either during the project or in the future.

Grafana

Grafana is an open-source time series metric analytics and visualization suite. It offers built-in support for various time series databases, including Graphite, InfluxDB, and Prometheus. It is written in Go.

Chronograf

Chronograf is an open-source time series visualization application, developed by InfluxData (the company behind InfluxDB). It is part of the TICK stack shown in Figure 4.1, which is meant to manage time series data [30]. It consists of Telegraf (data collection from different sources), InfluxDB (storage of the data), Chronograf (visualization) and Kapacitor (monitoring and detecting anomalies in the data, alerting based on triggers). Due to the tight integration (in the figure it can be seen that every element directly relies on InfluxDB), it is only viable as an option if InfluxDB is chosen (using Graphite is possible, but only through the use of an InfluxDB, introducing unnecessary complexity/work).

Graphite Web App

The Graphite web app is part of the Graphite suite, dealing with data visualization. It is implemented as a Django webapp, using Cairo for vector rendering. Analogous to Chronograf, considering it is only interesting if Graphite is chosen for data retention.

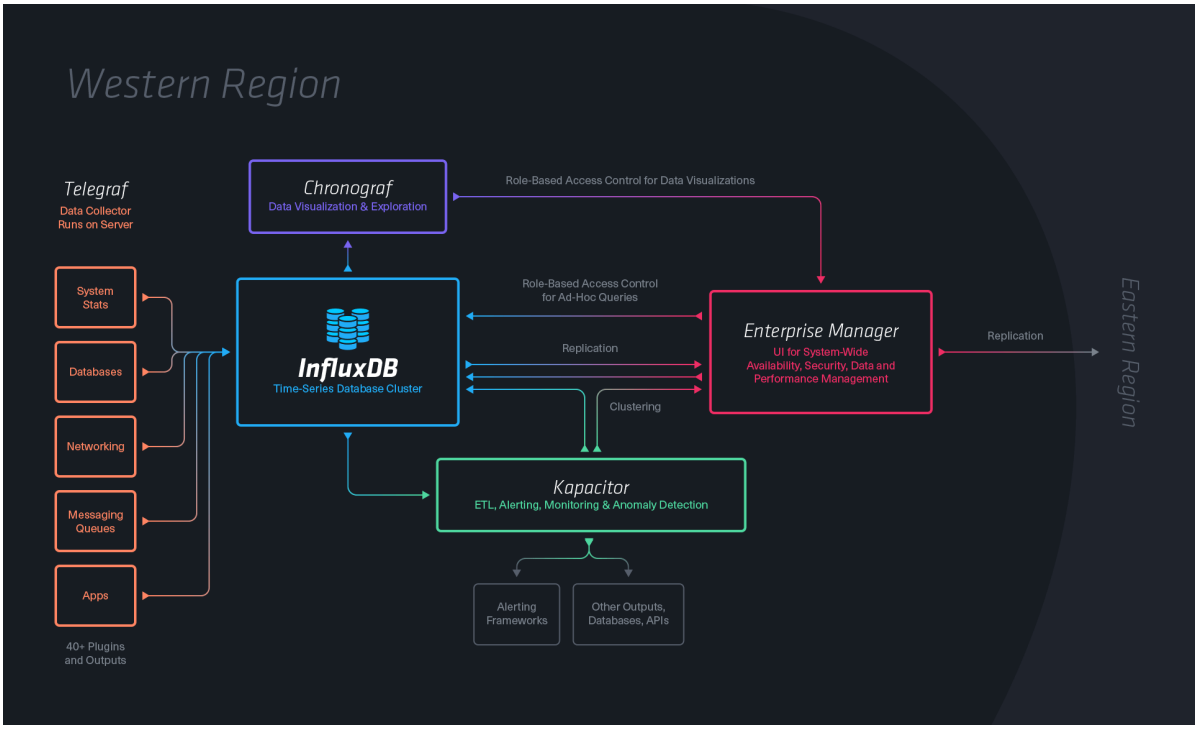


Figure 4.1: The TICK stack.

4.4. Chosen Development Tools

It this section the chosen tools for development such as the development strategy, programming language, and testing tools are explained and the choice for them is supported by arguments and the requirements set in Section 4.2.

4.4.1. Development Strategy: SCRUM

For the development strategy, Waterfall was deemed inadequate for this project as some of the requirements might change during the project, due to the relatively new and flexible nature of the company and its software stack.

Furthermore, SCRUM was chosen over Extreme Programming. While both strategies have similar philosophies, both parties have more experience with SCRUM. Several concepts of Extreme Programming are still considered useful, though, such as pair programming. The sprint duration was set at 2 weeks, meaning there will be 3 sprint reviews (20 January, 3 February, and 17 February). Furthermore, there will also be a sprint meeting on every Friday in order to monitor the progress and to react timely in the event of problems. The sprint plans for the sprints are shown in Appendix A and an overview of all sprints is given in Table 4.2. SCRUM's short sprints also allow for concurrent work to be done on multiple decoupled software products and thus fulfills the requirements set for a development strategy.

#	Weeks	Main activity
0	1-2	Acquire background information, gather requirements, create preliminary design
1	3-4	Development
2	5-6	Development
3	7-8	Development and Validation
4	9-10	Writing final report and prepare for demo and presentation

Table 4.2: Overview of sprints.

4.4.2. Programming Language: Go

Go was chosen over both C++ and Python, due to the ease of integration with Kubernetes and Docker and its abundance within Nerdalize. Apart from this, its harmony with the rest of the stack can be considered a key factor in the choice. Go also offers good performance and great tools for concurrency, one of requirements set in Section 4.2.2.

4.4.3. Development Environment

Repository: GitLab

Since both GitHub and GitLab offer almost the same features our choice for GitLab was made because GitLab was the preferred tool within the Nerdalize company. This means that experienced colleagues can help with setting up the repository.

Test System: Minikube

To test code on a running system Minikube was used. Minikube sets up a local Kubernetes cluster which allowed to deploy code locally making it easier to debug.

4.4.4. Code Quality: GitLab CI

To ensure each build has good code quality we will use GitLab CI. Continues integration will test the code both functionally as non-functionally. The choice for GitLab CI was made because it is an integrated part of GitLab and it has features that other CI tools don't offer, such as continuous deployment. An overview of how this was used in the project is available in section 5.3.

Testing: Unit Tests

Go offers an integrated testing package that offers unit testing, benchmarking and code coverage [31]. it will be used to ensure the written code behaves the way it is designed to. Section 5.4 further explains how this is done in the project.

Formatting: GoFmt

GoFmt automatically formats Go code, this ensures that all Go code has the same format making it easier to write, read and maintain [32]. As a result of this, there is no need for automated testing code style.

Documentation: GoDoc

GoDoc automatically generates documentation on Go code based on comments in packages and function.

4.5. Chosen Systems

This section goes in to detail on which systems where chosen as a starting point and the reasons they where chosen.

4.5.1. Data Collection: Heapster

Heapster was chosen for data collection as it offers a good starting point. Heapster uses cAdvisor, a tool that measures the resource usage and performance of containers, and runs it within a Kubernetes cluster collecting measurements from all containers. Heapster also collects additional statistics from Kubernetes itself. It aggregates this data and makes it available via REST endpoints. Heapster, however, does not support all the necessary statistics as all network traffic is treated equal and monitoring of disk I/O also seems to be lacking.

4.5.2. Data Retrieval and Storage: InfluxDB (and SQL)

As we tend to steer away from reinventing the wheel, RDBMS was not be considered for the primary data due to the relatively convoluted set-up that would be required. Prometheus, InfluxDB and to a lesser extend Graphite seem to offer solutions to many of our problems and promise to do so right out-of-the-box, while also enabling a better representation of our data. Ganglia, and MonALISA were also considered, but they are outdated and do not integrate with Kubernetes out of the box.

InfluxDB was also chosen due to the experience already obtained by the employees and because integration would be easier with Nerdalize's existing systems, which also rely on InfluxDB. As at this moment it cannot be accurately predicted whether InfluxDB will suffice for Nerdalize's needs in the future, the system will be built as modular as possible, enabling a relatively easy switch to another database.

However, a RDBMS will still be used, mostly for the secondary data (E.g. client information, relating jobs to clients or storing specifications of all systems). Because most modern languages (including Go) allow communication with RDBMS's supporting SQL through a uniform interface, choosing a specific RDBMS is not necessary at this point in time, as they can be changed with very little effort.

4.5.3. GUI: To Be Determined

The GUI would ideally be integrated with the rest of Nerdalize's GUI. However, as Nerdalize recently attracted a new employee to enhance their GUI experience, the used frameworks and set-up might change in the near future. This means that, while not ideal, the exact set-up of a (possible) GUI will fall out of the context of this project.

4.5.4. Data Visualization: Grafana or Chronograf

By choosing InfluxDB we are able to stay partly agnostic in our data visualization, giving us a fallback in case things go south. Having several options can be considered good practice, further emphasizing InfluxDB as a good choice. Apart from this, the choice also depends on the details of the GUI, which are not yet available at the time of writing.

5

The Development Process

This chapter is used to describe the development of the system. In particular, it discusses how the chosen development strategy was used in Section 5.1, the way several of Go's concepts were used in Section 5.2, how continuous integration was used in Section 5.3, and how unit testing was done in Section 5.4. Lastly, Section 5.5 explains why no SIG review took place for this project.

5.1. SCRUM

During the project, the SCRUM design strategy was used. The project started with two weeks of research, after which the first sprint plan and a backlog were made. During our sprints we didn't follow a strict schedule for sprint reviewing and planning, this resulted in sprint meetings that were postponed and a backlog that was not used to its full potential. The small team size, however, made it possible to discuss plans quickly and stay on track. In a larger team the problems would have been worse, so in SCRUM was not used optimally.

5.1.1. Sprint 1

During the first sprint an architecture design was created, halfway into the sprint Nerdalize made some large decisions and part of the architecture design had to be changed, something that interfered with the sprints planned. Nerdalize also started a new sprint in their software team. However, this helped us as the daily sprint meetings were a good moment to discuss progress and ask for help when problems were encountered. At the end of the first sprint, a working system was not realized yet. However, the final architecture design was presented, which was met with constructive criticism.

5.1.2. Sprint 2

In the second sprint, the implementation of all the parts of the system could be started. The choice was made to work on two different components and integrate them at the end of the sprint, which was a good decision as both components required much prior knowledge to get started on. During this sprint, the choice for Helm [33] to manage application deployments on Kubernetes was made. At the end of the sprint, a demo was given at Nerdalize.

5.1.3. Sprint 3

The last sprint consisted of integrating all separate parts of the system and solving all problems that arose when the system was deployed on a Google Kubernetes cluster. In this sprint, the focus was more on the user facing side of the system, such as the dashboards and API. Problems were encountered with the testability of our code.

5.2. Go

One of the reasons Go was chosen as the programming language was the way Go handles the concept of concurrency. Within Accumulus this was used at multiple places in the code. Deploying clusters happens in a Go subroutine, as does the syncing of Influx databases and the processors. In order to ensure everything happens in the right order waitgroups were used, while thread safety is guaranteed by using mutexes. An example of both is shown in Figure 5.1, in lines 73-75 we see how a mutex is used to claim writing rights on a set of points, line 79 calls functions in concurrent subroutines and uses a waitgroup (lines 78 and 84) to ensure all of them are finished before returning.

```
71 func processBox(mx *sync.Mutex, bp influxdb.BatchPoints, box Cloudbox, id stri
72     pt, _ := influxdb.NewPoint("power", tag, val, ts)
73     mx.Lock()
74     bp.AddPoint(pt)
75     mx.Unlock()
76     for _, node := range box.nodes {
77         n := node
78         wait.Add(1)
79         go func() {
80             defer wait.Done()
81             processNode(mx, bp, n, len(box.nodes), box.total, base, load, ts)
82         }()
83     }
84     wait.Wait()
85 }
```

Figure 5.1: Concurrency example.

5.3. Continuous Integration

In order to ensure high code quality, GitLab was chosen for continuous integration. However, due to the complicated setup of the system and the difficulties of running a Kubernetes cluster within a CI environment this was time-consuming. In the end, the choice was mode to not do integration tests as setting up a cluster and deploying the code every time commits were made took over 20 minutes, while the complicated state of the systems was hard to capture in automated testing.

Future work could improve on this by using GitLabs continuous deployments to automatically build and deploy new builds on test clusters in order to monitor them, possibly using dedicated Kubernetes CI solution such as Jenkins or Fabric8.

5.4. Unit Testing

Since problems with the CI already arose and no prior experience with testing in Go was present, the code was designed without testing in mind. This choice resulted in code that could not effectively be tested by Go's testing suite, which was partially solved by writing end-to-end test. While this was not necessarily the best way to test the code, they at least cover large parts of the code and offer some way to test the code's behaviour.

Future work could improve on this by introducing interfaces so functions are easily mocked and unit tests can be created without relying on dependencies such as Influx or Heapster.

5.5. SIG Code Review

In order to get feedback on the structure our code was send to the Software Improvement Group [34]. Due to the nature of our project (which was largely in development operations) and the separate software components, the SIG determined that a review of the code at that time was not feasible. This was due to the absence of automated Go analysis at their end, and having our system spread across

a modified Heapster instance, a central Accumulus application, some shell scripting and several Helm deployment scripts for both.

6

Design of Accumulus, a Monitoring and Cluster Analysis System

This chapter describes each part of Accumulus in detail, roughly in chronological order. Section 6.1 gives an architectural overview of Accumulus as a whole and describes how the Accumulus ecosystem is deployed, both in the client and the central cluster. In Section 6.2, Accumulus Core and how it relates to other components is described in more detail. The SQL database and the database syncing mechanism between core and client cluster is explained in Section 6.3. Section 6.4 then goes into detail on how Heapster was extended to attach extra tags to all measurements. Next, Section 6.5 describes the design and implementation of the Influx syncing mechanism that was implemented for collecting data from the client clusters. Section 6.6 explains how processors are used on the InfluxDB in the core cluster to calculate certain metrics. How the data collected by Accumulus is visualized is explained in Section 6.7 Lastly, Section 6.8 details possible extensions to Accumulus and alternative setups.

6.1. Architecture Overview

The designed system will consist of different components, which are:

- An extended version of Heapster
- InfluxDB
- Grafana
- Accumulus Core
- An SQL database

A complete overview of the architecture is shown in Figure 6.1. This system is split into two parts, an instance of the client part will run in every client cluster, while the core part runs in a central location and is responsible for deploying the client part in each cluster. The central part connects to all user clusters to collect compute resource data, while it pushes data from processors (such as power consumption and the cost of the resources used so far) to the client clusters. This set-up is chosen in order to enforce that no sensitive data is stored in the client cluster, while also ensuring the client cluster is still monitored if the connection with the core is lost. An added benefit is that the client has access to high-resolution compute resource data if needed, without putting any stress on the centralized application or requiring a very large central storage capacity.

6.1.1. Central Cluster Ecosystem

In the leader cluster, 3 pods are running. The central Accumulus instance, an InfluxDatabase and a Grafana pod (the latter is not required for a correct working of the system, but included for data visualization). Services are also running, exposing the InfluxDatabase, Grafana, and the Accumulus

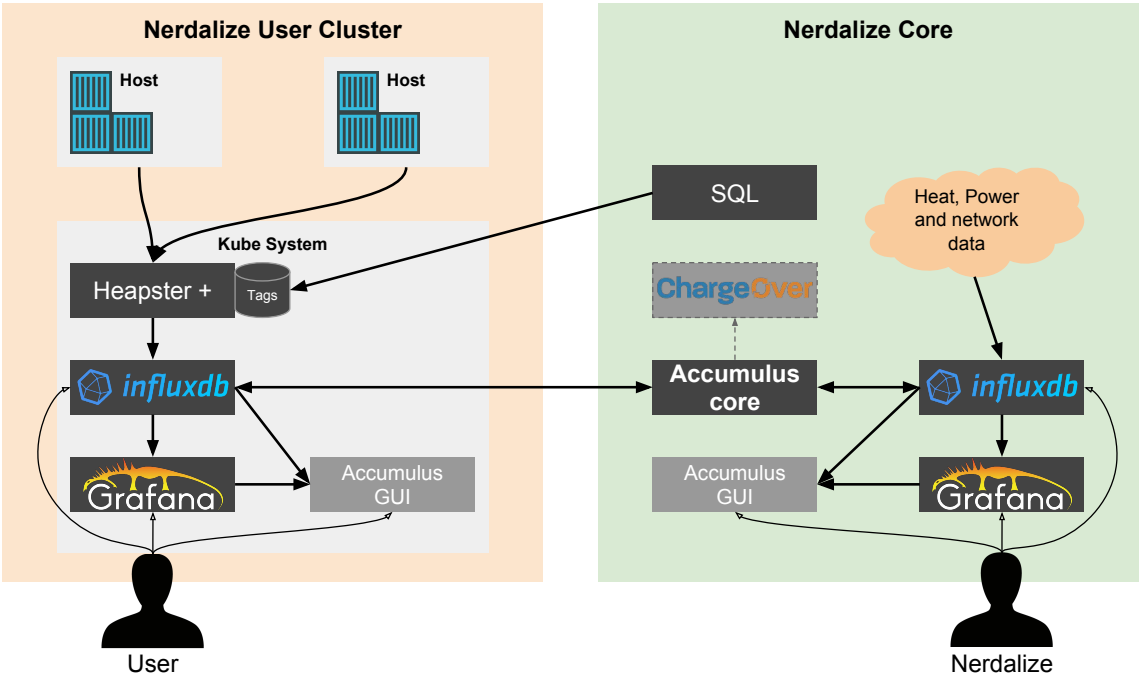


Figure 6.1: An overview of the system architecture.

API. A service account and a secret (containing a Docker key) on which it relies are also deployed for access to the containers hosted in a private Docker repository.

6.1.2. Client Cluster Ecosystem

In each follower cluster three pods will be deployed; A modified instance of Heapster, an InfluxDatabase and a Grafana pod. Again services are set up to expose Influx and Grafana to the user. The same set-up relying on a secret and a service account is used here to retrieve the images from a private docker repository. Finally, a configmap is deployed, carrying the text files which will be converted to the database used by the modified Heapster.

6.1.3. Installation

The deployments themselves rely on Helm, a Kubernetes package-manager. In order to prepare the leader cluster for installation, a Tiller-pod (Helm's client-side) should be installed. This can be done by calling "helm init" on a CLI with access to the desired cluster. Once a Tiller pod is ready, the central Accumulus instance can be installed from the "Charts" found under "etc/Helm". The central instance takes care of the installation on the follower clusters, also relying on Helm. New client clusters are added by calling <IP of the Accumulus-pod>:8080/accumulus/clusters/add/<clustername>. For the client installation, the central Accumulus instance first tries to install Tiller, after which the other "Chart" under "etc/Helm" is used.

6.2. Accumulus

An instance of Accumulus Core will run in the central Nerdalize cluster. Accumulus Core is responsible for the connection between the customer cluster and the Nerdalize core cluster and fulfills three main functions; deployment, collection, and processing.

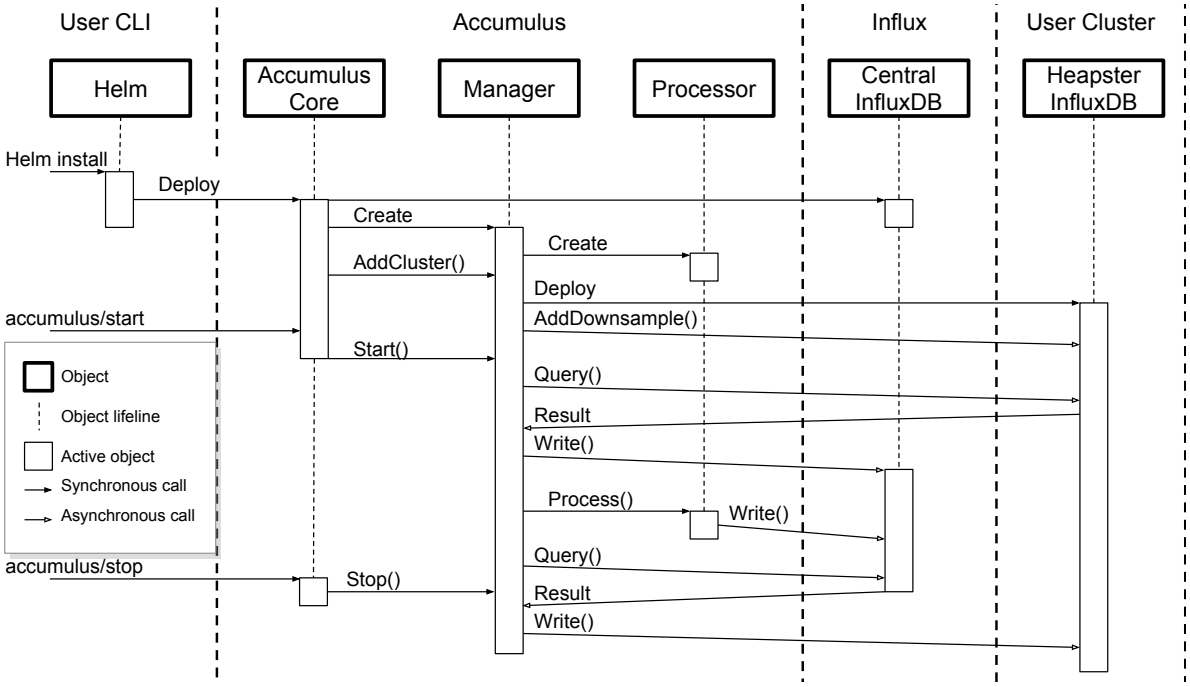


Figure 6.2: Sequence diagram of the deployment and functions of the Accumulus core.

6.2.1. API

The API is the main way to interact with Accumulus, with it clusters can be added and removed and the system can be stopped and started. The endpoints supported by the API are listed in Table 6.1

Endpoint	**function**
/accumulus	Shows the current status of the system
/accumulus/start	Starts the manager
/accumulus/stop	Stops the manager
/accumulus/clusters	gives back a list of clusters
/accumulus/clusters/add/[name]	Adds the cluster from the Kubectl file to accumulus
/accumulus/clusters/remove/[name]	Removes the cluster from accumulus
/accumulus/clear	Clears all data in accumulus

Table 6.1: The Accumulus API.

6.2.2. Manager

The manager is responsible for collecting and processing all the data from follower clusters. On regular intervals, it queries all user clusters for new measurements which it then stores in the central Influx-Database. This database is then used by processors that are added to the manager, those processors generate extra metrics such as power usage and cost by correlating the data to external metrics. Last the manager syncs the generated metrics back to the right follower clusters. Figure 6.2 shows how the Accumulus is deployed and uses a manager.

The manager is built in such a way that if something goes wrong it will be tried again in the next tick, as such it doesn't matter if the network drops or a query doesn't return a result.

6.3. SQL Mechanism

The SQL database running in the core cluster currently contains information on all Cloudboxes (and in the future possibly also about jobs, clusters, and customers). While in theory, it's not possible for

clients to access nodes or information outside their assigned namespace(s), Kubernetes is not built for multi-tenancy, so this security can not (yet) be considered rigid. As such, and because the data can be considered sensitive (the locations and the total number of Cloudboxes, customer information etc.), it is distributed on a need-to-know basis to the customer clusters on start-up.

The distribution is done by exporting the schema of the master database and its contents to text files using the SQLite3 CLI. The content is exported using queries, allowing for flexible rules regarding the exact information exported. The text files are then sent to the cluster using a Configmap, as Kubernetes does not allow any other way to initialize a pod with different files.

During start-up, the modified Heapster will use the text-files and rebuild an SQLite3 database from them. Once again, the SQLite3 CLI is used for this. The SQL database is used by Heapster and the usage of the standard Go SQL interface makes the database back-end completely interchangeable.

In Heapster, a simple cache is included, in which every metric requested for a host is cached, significantly reducing the traffic to the SQL database. When the cache is invalidated, it is completely purged and will be rebuilt by the following queries. While this is not the most efficient cache invalidation method, it is deemed sufficient for this project as a complete cache rebuild (for large clusters having 1000's of nodes) takes less than a second on modern machines and cache rebuilds are expected to be very rare. In fact, they should only happen if information for a node is entered incorrectly in the central database, or when a node receives different hardware. Node removal or addition will not lead to an invalid cache, as in the first case the cached data is simply no longer requested, while in the latter case the value will be cached the next time it gets queried.

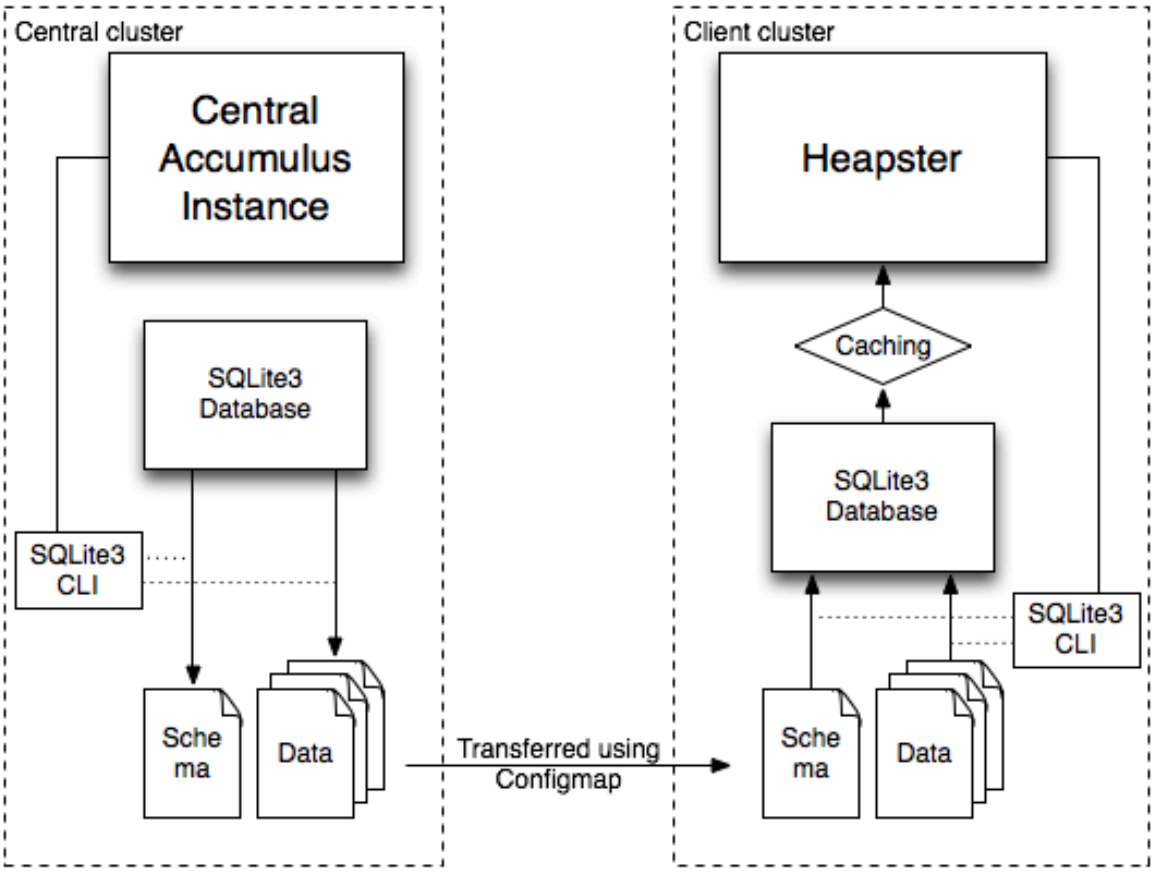


Figure 6.3: Overview of the SQL database initialization mechanism.

6.4. Heapster

For this project, Heapster is extended so additional tags can be added to metrics. An overview of extra tags is shown in Table 6.2. Heapster will use the SQL Database to look up the right data, for example relating resources to the hardware they were used on, and adding the Cloudbox id.

Tag	Metrics	Reason
Cloudbox id	All metrics	Needed to separate measurements for different clusters and to estimate power use and heat production
Cluster id	All metrics	Needed to link metrics back to a customer
Cpu type	Pod and node	Gives the user insight into his performance on the type of cpu And gives Nerdalize data on Cpu usage
Ram type	Pod and node	Gives Nerdalize data on ram usage
Disk type	Pod and node	Gives the user insight into his performance on the type of disk And gives Nerdalize data on ram usage

Table 6.2: Tags on metrics and reason for these tags.

To add this functionality to Heapster a new package was created, in this package two files are present; the tagger handles tagging and caching, and the SQLSyncer builds the database from files. Thanks to this approach only 3 lines had to be added to Heapsters original code, making this extension easily maintainable.

6.5. Influx

InfluxDB will be used for storing time series data. There are multiple instances of InfluxDB, one in each customer cluster and a central instance for long term storage and historic searching. The instances in the customer clusters will keep precise data. InfluxDB down-samples this precise data to a lower resolution, which will be collected by the central Accumulus instance. InfluxDB can also use retention policies to reduce disk usage by only keeping data for a certain time.

In order for Accumulus to sync between and operate with Influx databases, the InfluxSync package was built. InfluxSync offers a comprehensive set of functions seen in Figure 6.4 to query, write and sync influx and uses its own InfluxPoint datatype for this.

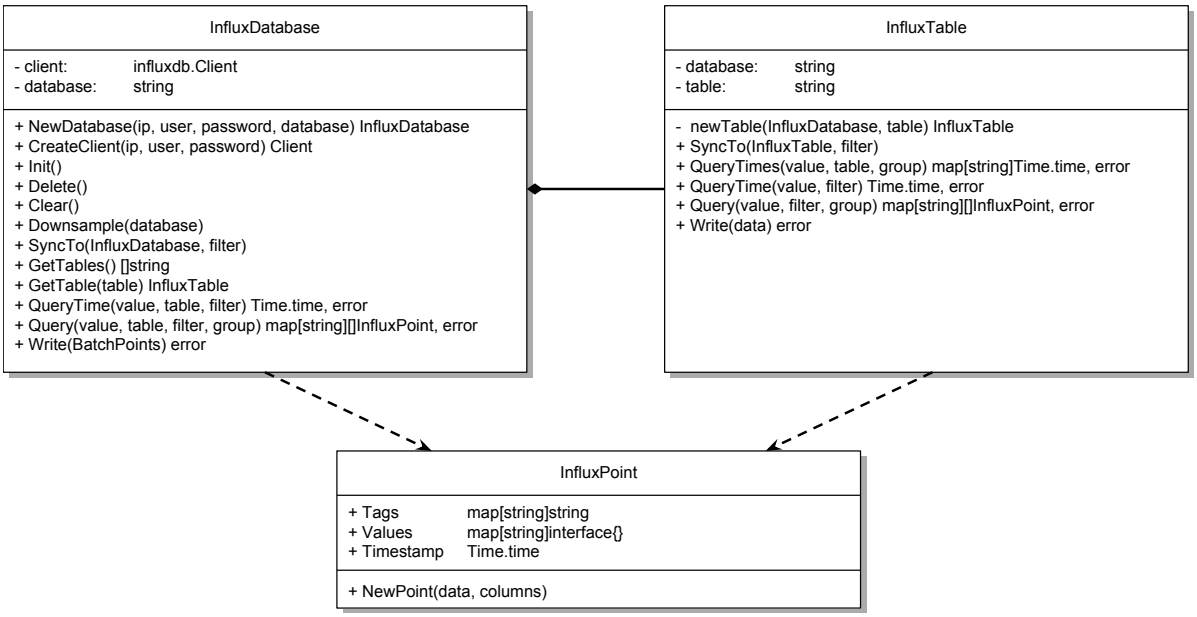


Figure 6.4: Diagram of the core database classes in our system.

6.5.1. Databases and Tables

InfluxSync has a datatype for both databases and tables, a database object holds an Influx Client to communicate with the Influx instance and a string that contains the name of its database within that Influx instance. By calling GetTables() the names of all tables in the database can be retrieved, an InfluxTable object can then be retrieved by calling GetTable(name).

The InfluxDatabase object also has functions to clear its content or add a downsample query to it. On its creation, it will also create its database in Influx if it does not exist yet.

6.5.2. Querying and Writing

InfluxSync offers two main forms for querying, a regular query and a time query. The regular query can be invoked by calling Query(select, where, group) on a table object, with select containing the values you want to query and where the filter, it returns the result in a map, with the results of grouping as keys.

6.5.3. Syncing

Syncing happens per table with the assurance that the tables will be equal after a successful sync. A sync first queries the last synced data point then queries all points after that point from the source table and writes them to the sink table. InfluxSync does this concurrently to reduce the network delay. A sync for an entire database is also done per table, however, there is the SyncToFull() function that does it for the entire database in a single query.

6.6. Processors

Accumulus has support for flexible processors, which get called with the core database as a parameter. They can then perform operations on this database and create or change metrics. 2 processors are developed for Accumulus, a power processor, and a cost processor. Processors run in the central cluster, as they handle sensitive data which is not allowed to exist in the client cluster.

6.6.1. Power

The power processor gets the power consumption from Cloudboxes from Nerdalizes system, then it calculates a distribution over nodes. Every node gets a split of the total power consumption based on a known idle consumption and the node utilization. The processor then splits the power consumption over each pod running on the node, all pods get an equal part of the nodes idle consumption and the remainder is distributed based on CPU time used by pods, a common metric used to approximate power draw [35]. A more detailed analysis of the model the power processor uses is given in Section 7.1.3

6.6.2. Cost

The cost processor is a simple mapping processor, it has a defined ”realistic” cost for each resource and calculates the operational cost for each pod, node, namespace, and cluster. By customizing this function to Nerdalize’s billing model when they switch to per resource billing, the user can get a clear estimate of the total cost of his project, sub-projects, and pods.

6.7. Grafana

For Accumulus we created custom Grafana dashboards shown in Table 6.3, those offer a clear view of the data collected and generated by Accumulus, different dashboards are created for the user and administrators. If users want even more insight they can also create their own dashboards. An example of one of the dashboards is shown in Figure 6.5

Grafana is a dashboard application for time series, it visualizes time series data. Heapster comes with its own version of Grafana, which has pre-configured dashboards for both cluster and pod performance.

Grafana will be used for visual representation of the data, both the customer and Nerdalize will have their own Grafana interface and the possible GUI that may be developed will most likely also use Grafana to render graphs.

User	Dashboard	Usage
Client	Cluster Overview	Shows cost estimate and overall utilisation
Client	Application View	Shows the usage of indiviual applications and is useful for analyzing performance
Nerdalize	Master Overview	Can show the total revenue and profit Nerdalize makes, and the utilisation of their servers
Nerdalize	Client Overview	Gives nerdalize a closer look at a certain client, an be used to analyse their usage patterns and help them improve their application

Table 6.3: Grafana dashboards.

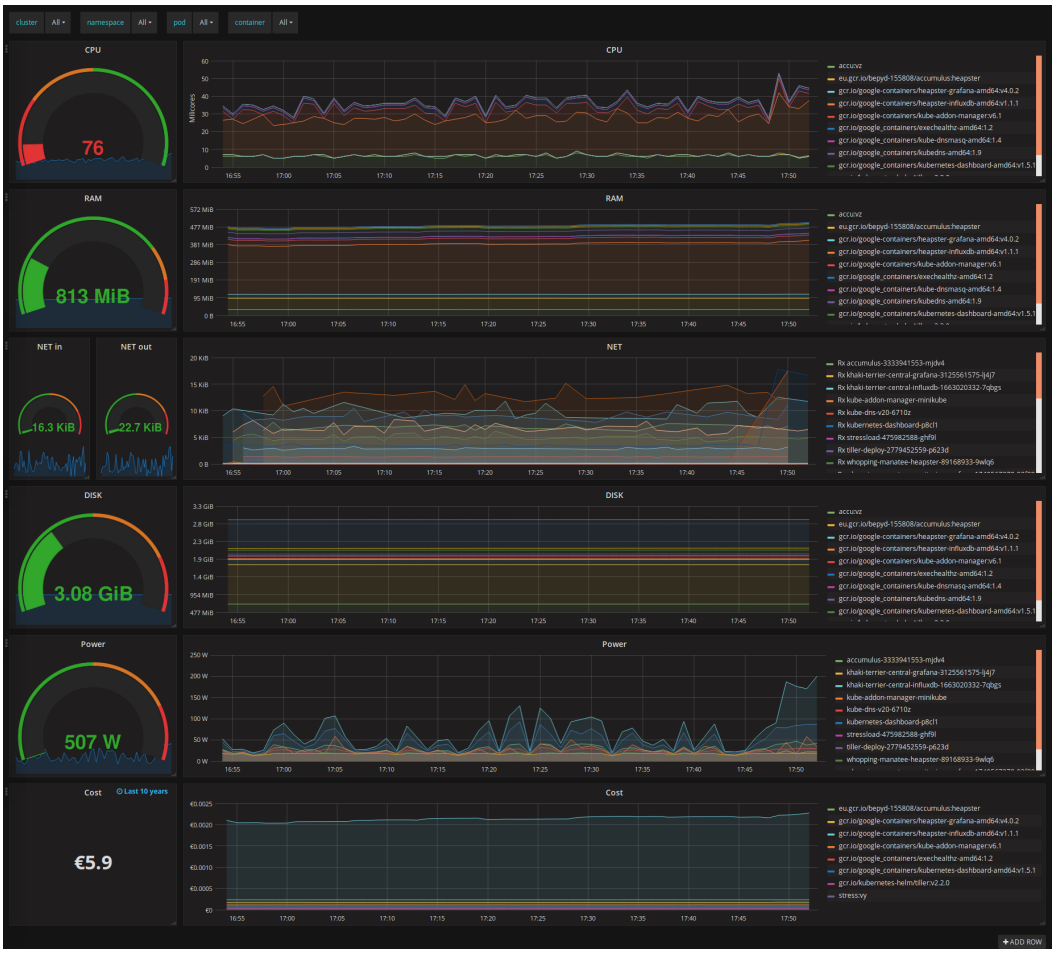


Figure 6.5: The Application view dashboard.

6.8. Possible Extensions

Currently, Accumulus is running in a simple single leader setup. In this section, two alternative architectures are proposed.

6.8.1. Multiple Leaders

Since the central Accumulus instance will check for an existing installation of the modified Heapster before installing one, an architecture having multiple leaders can easily be established. This can be used in high-availability setups and for meeting data replication requirements. The only modification needed for such an architecture is ensuring that the heat and power data is only pushed to the follower cluster by one leader at the same time.

6.8.2. Multilevel Aggregation

In case the number of follower clusters becomes too high for a single leader, a multilevel architecture with several leaders following a super leader can be envisioned. In such a cluster, the usage data propagates upwards until reaching the highest level leader, where it will be stored in the central database. This set-up is easily realized by exposing the leader's database using the same service as used for the follower's database. The super leader will then automatically start treating the leader as if it were one of its followers, pushing power and cost data while pulling usage data. Installation in the follower clusters will still be done by the leaders, although the modular approach would easily allow this to be done by the super leader (remember, a leader will always check whether Heapster has already been installed) or the super leader could defer this to the leaders.

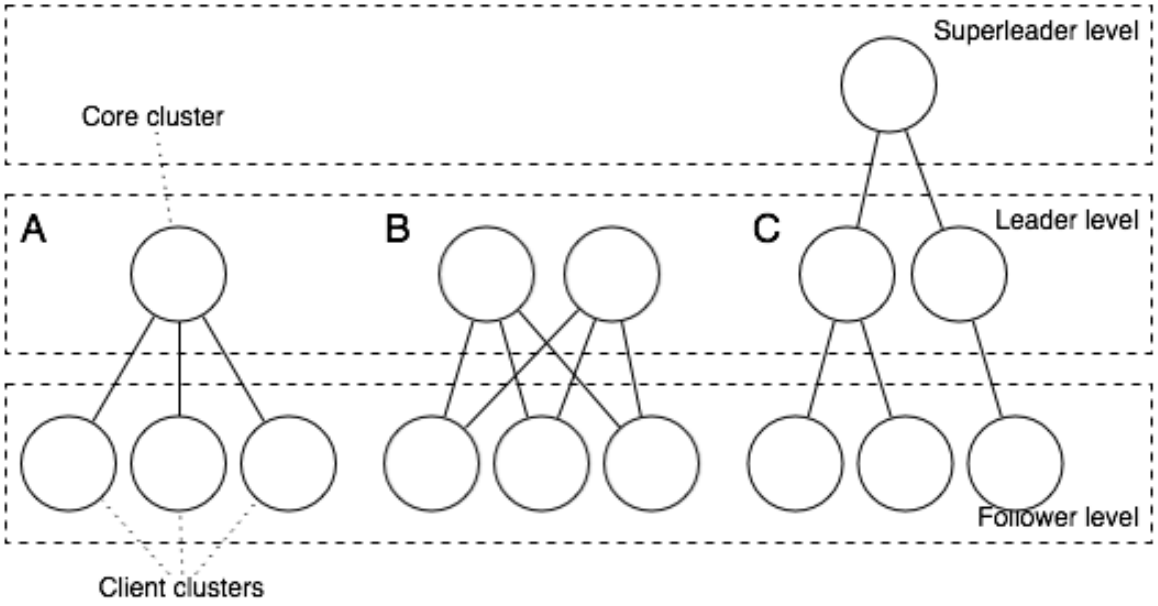


Figure 6.6: Comparison of the current setup (A), a setup having multiple leaders (B), and a multilevel setup (C).

7

Experimental Testing and Validation

A measuring system system is useful only if it satisfies two conditions. The system needs to be accurate, which will be discussed in Section 7.1, and the system needs to have a small overhead, which will be experimentally verified by the experiments proposed in Section 7.3. A additional constraint in this case is scalability, which is discussed in Section 7.4.

7.1. Accuracy Evaluation

A measurement system is not useful if the measurements are inaccurate. Therefore the system needs to be tested for accuracy on all metrics it collects. Section 7.1.1 shortly talks about compute resources, Section 7.1.2 explains how power consumption and heat production of a Cloudbox is measured. Lastly Section 7.1.3 goes into detail on the accuracy of the model for the power processors we built for Accumulus.

7.1.1. Compute Resources

Heapster receives its measurements directly from cAdvisor and as such, Accumulus' accuracy for resource usage depends entirely on that of cAdvisor. cAdvisor collects its measurements directly from the kernel and OS. Because of this these measurements can be assumed accurate enough by doing a basic validation with a known impact on the system. The experiment and results for this validation are covered in Section 7.2.

7.1.2. Power and Heat

The Cloudbox contains a class B electrical energy meter, which is legally required to have a measurement error of less than 2 percent over the temperature range where Nerdalize is operating in [36].

All of the power consumed by the compute modules is transferred to the water, with the exception of the heat lost to the surroundings, meaning an upper bound can be given for the error with which the created (useful) heat is estimated. No useful heat is created if the water in the boiler already has the maximum temperature, something which is easily measured.

The actual heat used by the homeowner is more complicated and can only be inferred by the measurements from two temperature sensors (10K NTC with a B constant of 3435 K), one in front of the heating area and one after. However, for our system the heat used by the home owner is at this moment irrelevant.

7.1.3. Power Model

The power consumption is split over pods and containers in a Cloudbox. To do this the power processor uses a model to distribute the power, this is done by the CPU time used by pods and containers.

In Figure 7.1 the results of putting different loads on an actual Cloudbox are shown. This data was gathered by running stress [37] for 10 minutes. From this data, we can see that CPU power draw is almost linear between a load of 2 to 20 Cores, after which only a slight increase from hyperthreading is seen. The standard deviation in this experiment was low and when hyperthreading the deviation disappears as the meter starts reporting the system to be using a constant amount of power, because of this low deviation we shortened the rest of our experiments on power usage to 2 minutes. In Figure 7.2 the deviation D_r between just CPU load and CPU plus RAM load is displayed. Figure 7.2 also shows that RAM usage only accounts for up to 2.5% of the total power usage of the system. In Figure 7.3 the deviation D_f of the estimated power draw by the processor from the actual power consumption of the Cloudbox is shown. From all this information we can conclude that the deviation depends on the function, RAM use and the deviation of the meter D_m. The maximum total deviation is the given by:

$$1 - ((1 - D_f) * (1 - D_r) * (1 - D_m)) \tag{7.1}$$

When done for all measurements with linear interpolation this gives the graph shown in Figure 7.4, showing that if the load on a machine in a Cloudbox exceeds 12.5% the processor is accurate within 10%.

Other factors such as network traffic and the temperature of the machine also play a role the power usage of the Cloudbox. However, according to [35] the overhead of switches and network traffic falls in a small dynamic range and can therefore be accounted for as base load. For the temperature, this is not the case, but within Nerdalizes application of high compute cloud most machines will be under a constant load and as such their temperature will be constant.

This means that the created power processor is accurate as long as there is a significant load on the system. Furthermore, since the measurements are split over the Cloudbox the total will always add up, so the only case in which the processors accuracy is critical is if multiple users are running on the same Cloudbox.

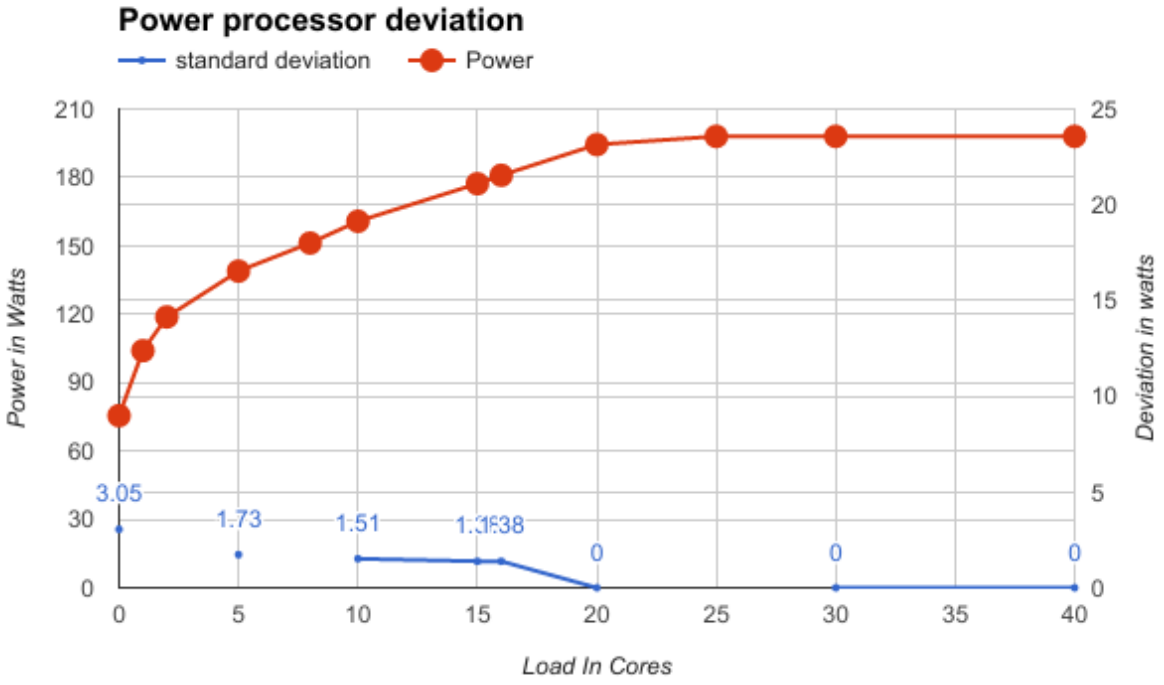


Figure 7.1: Measured Cloudbox power usage.

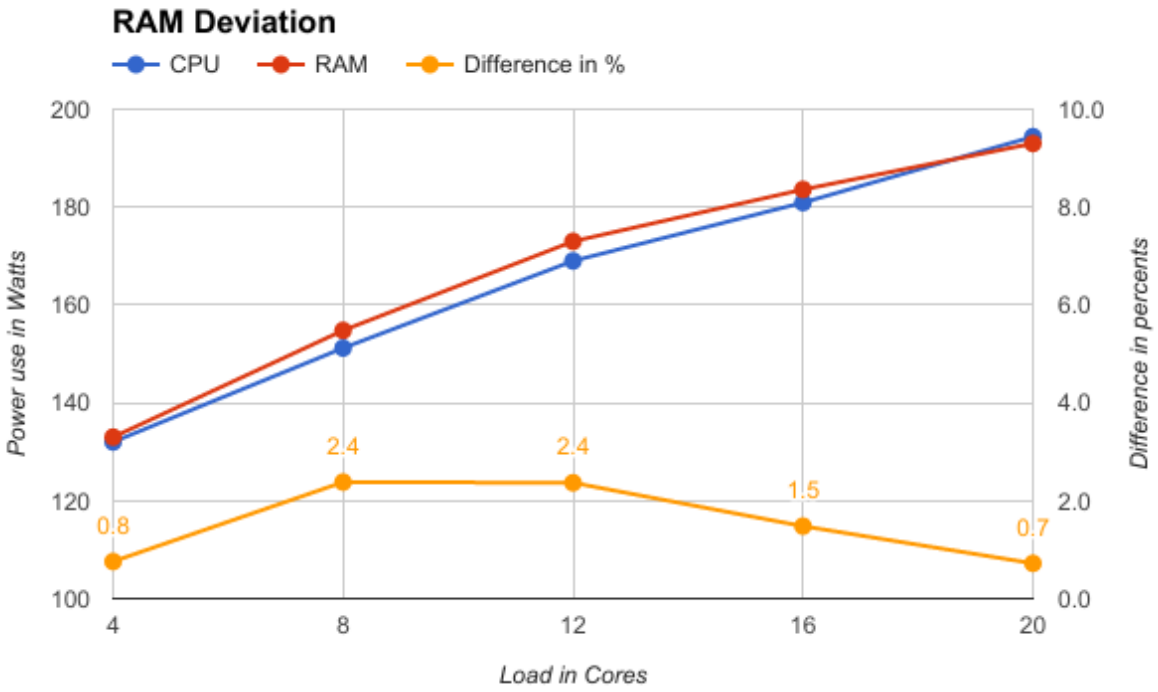


Figure 7.2: Measured Cloudbox power usage when using RAM.

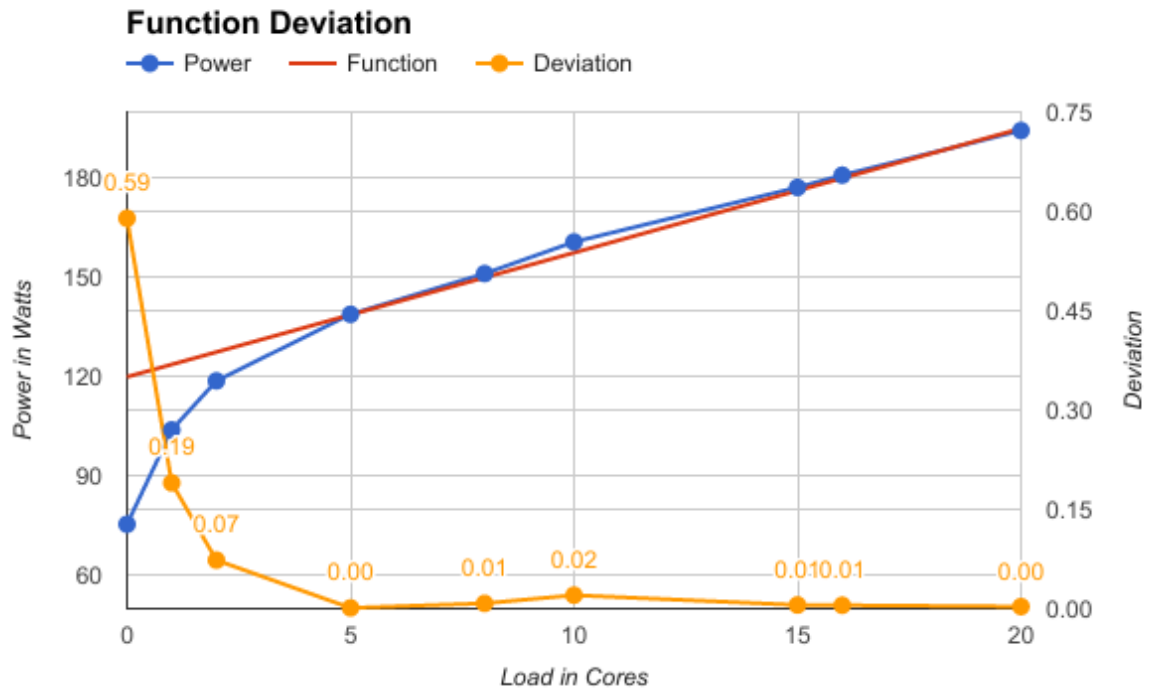


Figure 7.3: The deviation of the processor function against the actual power draw of a Cloudbox.

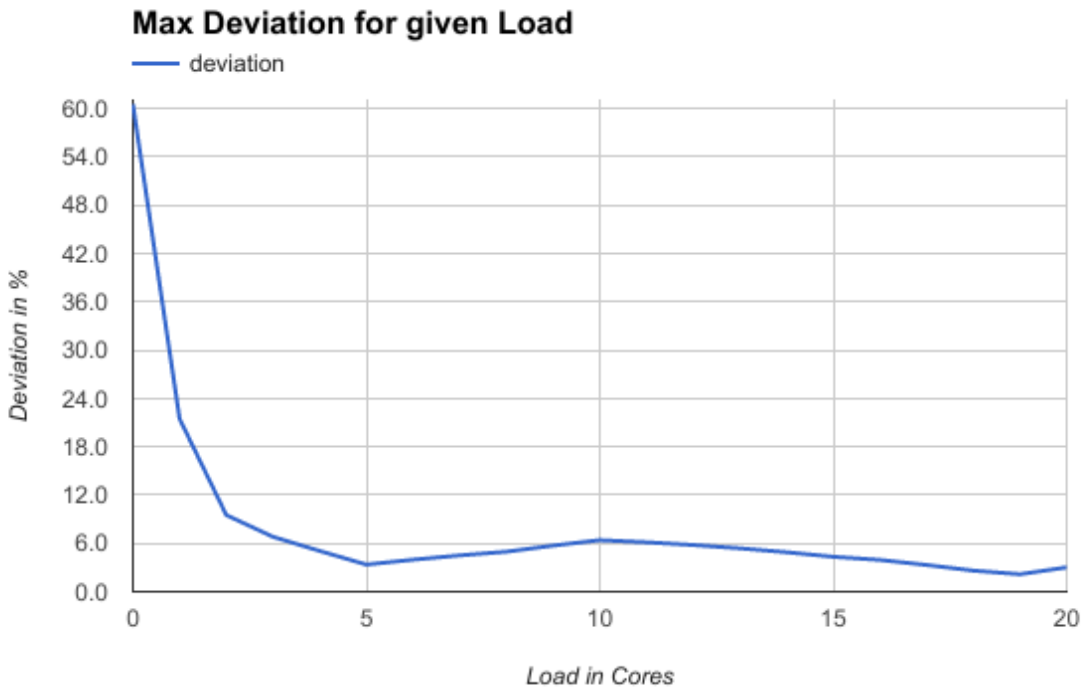


Figure 7.4: The maximal deviation of the processor.

7.2. Basic Validation

To validate the basic working of the system, Nerdalize has requested some smoke tests. In those experiments, stress tests will be used to cause a predictable load on the system. Accumulus should then show the expected values for the load. To do this test a small application with a REST API was created, this application can stress the CPU, RAM, network, and disk. The possible endpoints are shown in Table 7.1.

endpoint	**function**
/stress	shows if the application is running
/stress/CPU/cores	Starts a number of CPU threads calculating sqrt(random).
/stress/RAM/size	Generates a array of the give size in megabytes.
/stress/disk/size	Writes a file of given size to disk.
/stress/net/n	Requests the Go network package page n times.

Table 7.1: Stressload.

Table 7.2 shows the resources that are tested and the expected loads that Accumulus should measure. The test results are shown in Figures 7.5, 7.6, 7.7 and 7.8. From the CPU test, we can see that stressing a single core results in an average of 986 millicores used, a deviation of less than 2% of the expected load. The dual core stress test has a higher deviation but this was expected as the test setup had only 2 cores. In the RAM test, we also see a small overhead, other test showed that this is due to Go and the way Go manages its memory. The disk test shows that the measured use exactly matches the expected use. Lastly, the network test shows the received network traffic from requesting a web page for 5 minutes, the measured value is slightly higher than the expected value due to headers and other overhead not accounted for in the expected value.

So the results of the basic validation test show that Heapster and thereby Accumulus is accurate.

Resource	Test program	Expected result
CPU	Single core stress test	1000 millicores used
CPU	Dual core stress test	almost 2000 millicores used
RAM	Generate 10Mb array	10Mb RAM used
RAM	Generate 20Mb array	20Mb RAM used
RAM	Generate 50Mb array	50Mb RAM used
RAM	Generate 100Mb array	100Mb RAM used
Disk	Write 20Mb file	20Mb disk used
Disk	Write 40Mb file	40Mb disk used
Disk	Write 80Mb file	80Mb disk used
Disk	Write 120Mb file	120Mb disk used
Network	Request Go web page	11Mb network rx

Table 7.2: Basic validation test parameters.

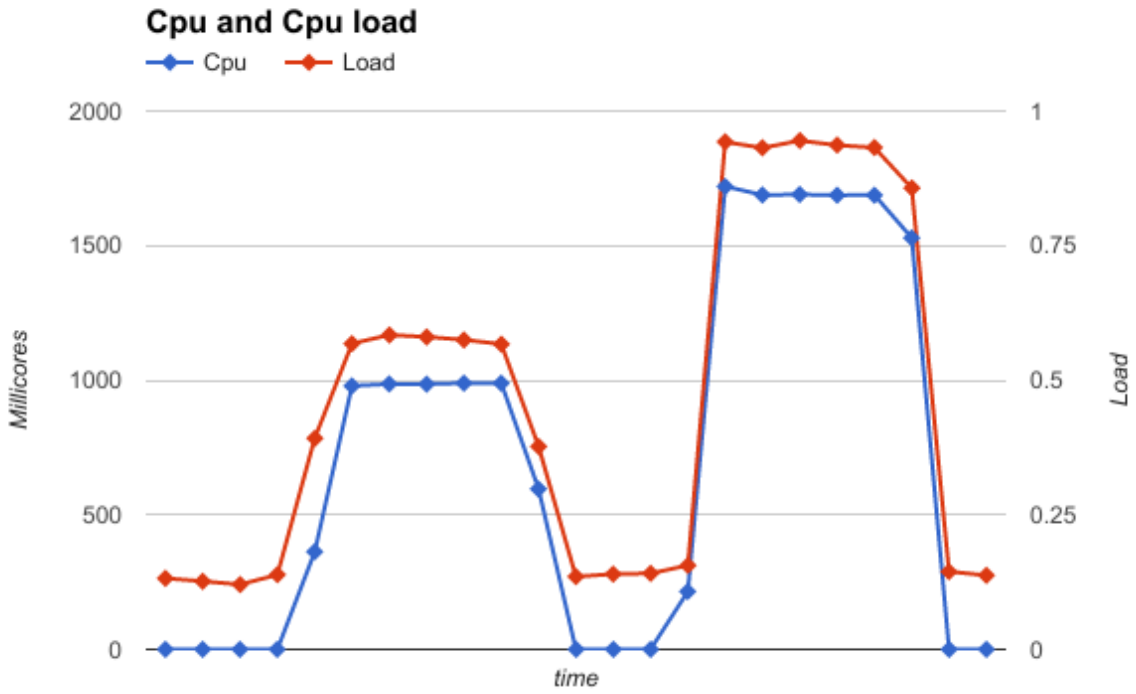


Figure 7.5: Measured CPU usage.

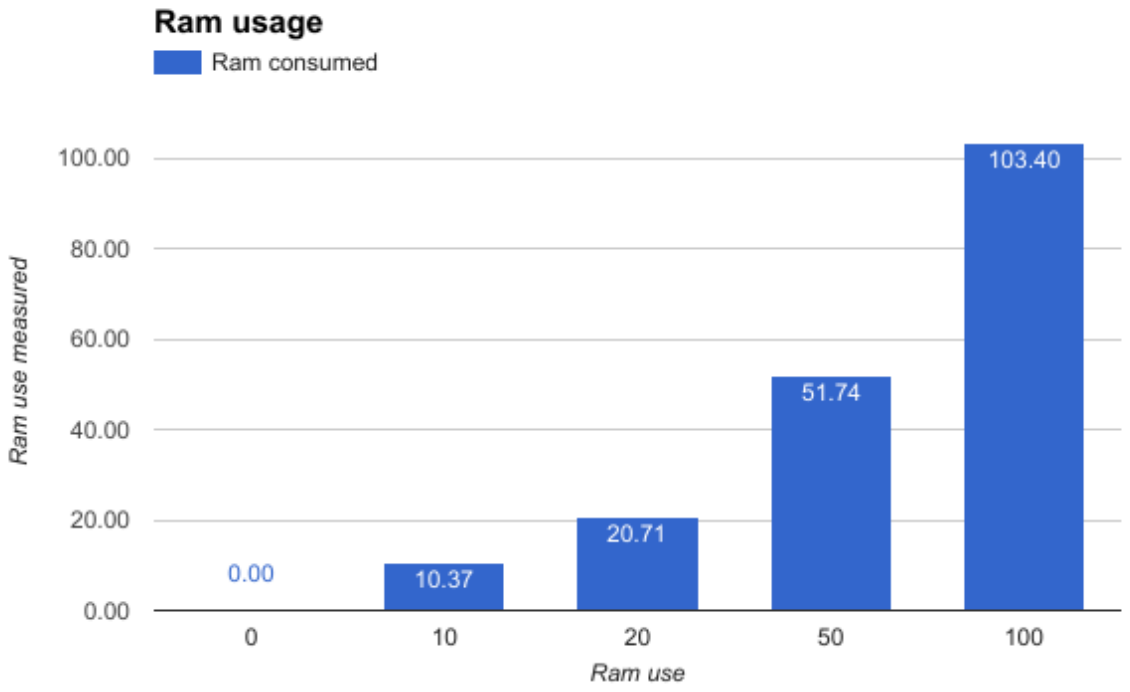


Figure 7.6: Measured RAM usage.

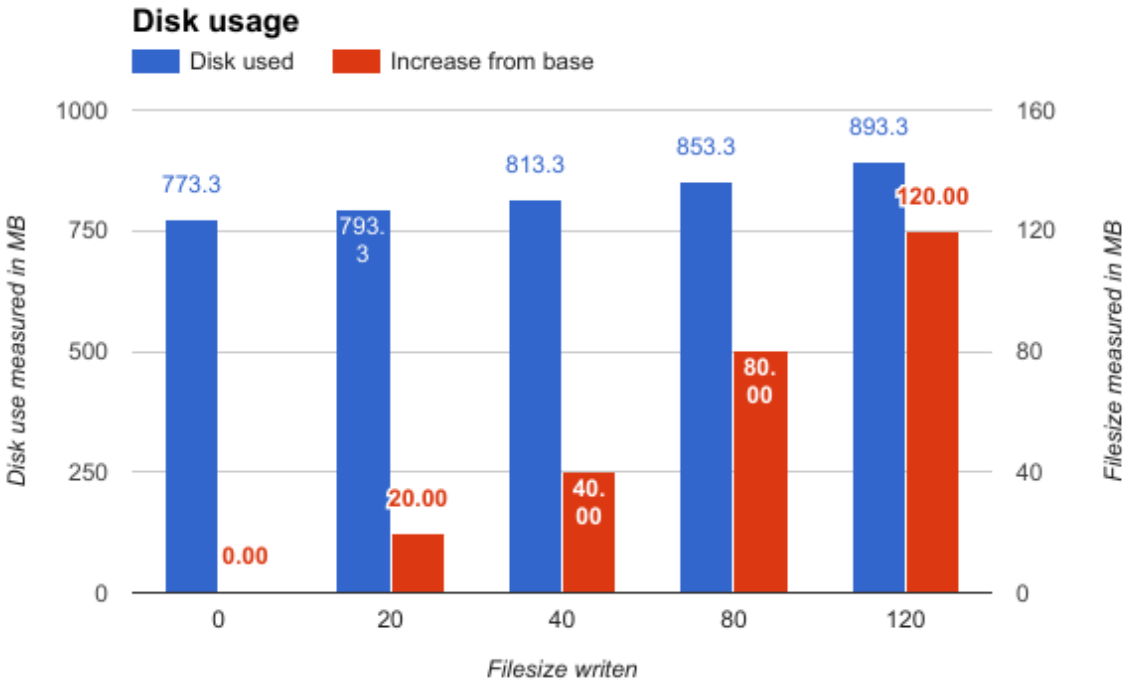


Figure 7.7: Measured disk usage (blue) on left axis, measured increase after writing file (red) on right axis.

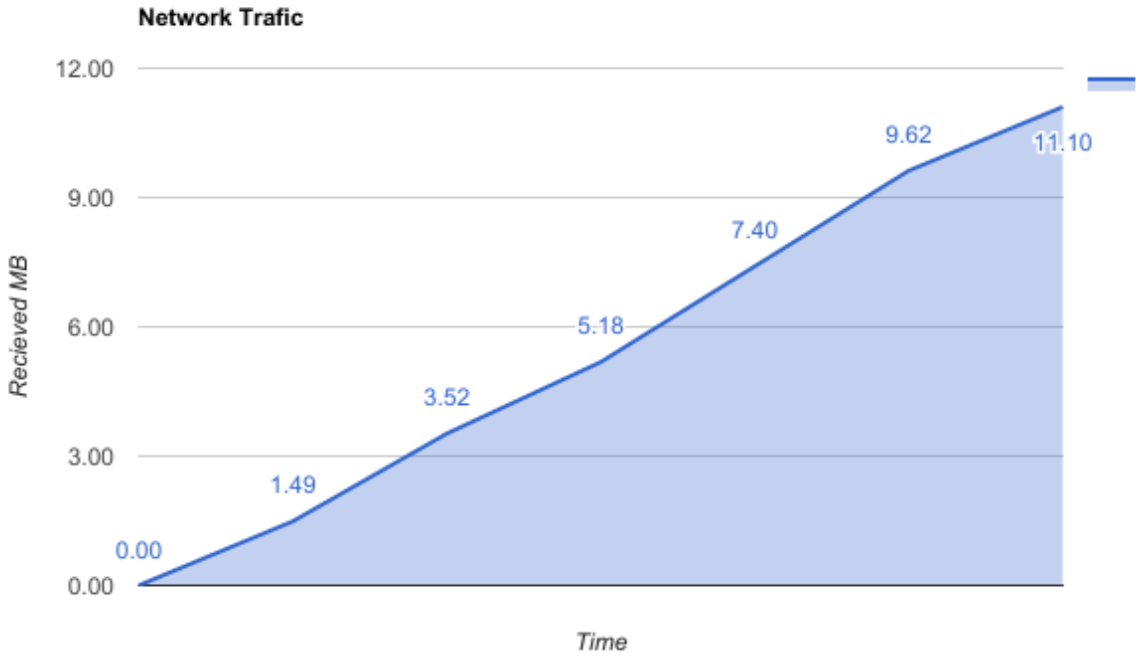


Figure 7.8: Measured network traffic.

7.3. Overhead

Overhead is another important factor in measuring systems. This section explains the experiments testing the overhead in Section 7.3.1, while the results are discussed in Section 7.3.1.

7.3.1. General Setup

In the overhead test, the performance of a cluster with three different setups is measured. Once on a bare Kubernetes cluster, once on a cluster with vanilla Heapster deployed and once on a cluster with our version of Heapster deployed, which syncing to a central Accumulus instance. On each cluster, several benchmarks are run, after which their duration is compared. The slowest 10% of each cluster are not taken into account, in order to minimize outliers. Nerdalize has labeled this experiment interesting, as it is very close to their intended use case.

The test load consisted of running 100 containers containing a ray-tracer. It was run on a cluster consisting of 6 instances of N1-standard-1 in the GKE.

7.3.2. Results

Setup	Average fastest 90 %	Total average	Standard deviation
Bare	00:06:40	00:06:52	00:01:40
Heapster	00:06:34	00:07:00	00:01:48
Accumulus	00:07:00	00:07:15	00:01:32

Table 7.3: Results of the overhead test for a bare Kubernetes cluster, one with only Heapster and one with a modified Heapster syncing to a central instance.

From the results, it is visible that a cluster containing a synced Heapster is only 5% slower than a bare Kubernetes cluster. However, the fact that a cluster containing only Heapster is faster than a

bare cluster, combined with the relatively large standard deviation in all three tests shows that more extensive tests are needed in order to arrive at a solid conclusion.

7.4. Scalability Evaluation

As mentioned the developed system should be scalable, which is experimentally tested in this section. The general setup of the experiment is discussed in Section 7.4.1, the effect of the number of pods/containers on the resource consumption in Section 7.4.2, the effect of the number of nodes on the resource consumption in Section 7.4.3, and the effect of the number of clusters on the resource consumption in Section 7.4.4. The results are then discussed in Section 7.4.5.

7.4.1. General Setup

In the scalability tests, a cluster is monitored by our modified Heapster. Heapster is deployed twice, one instance is synced to the central Accumulus, while the other is not. This is to measure the effect of the syncing on the resource consumption. 3 variables will be varied and their effects on the resource consumption will be measured by Heapster itself. The variables are the number of pods/containers inside a cluster, the number of nodes in a cluster and the number of client clusters that sync to the central Accumulus instance. Due to the duration of the tests and their accompanying costs, each test could only be run once. As such, there is no data available regarding statistical deviations.

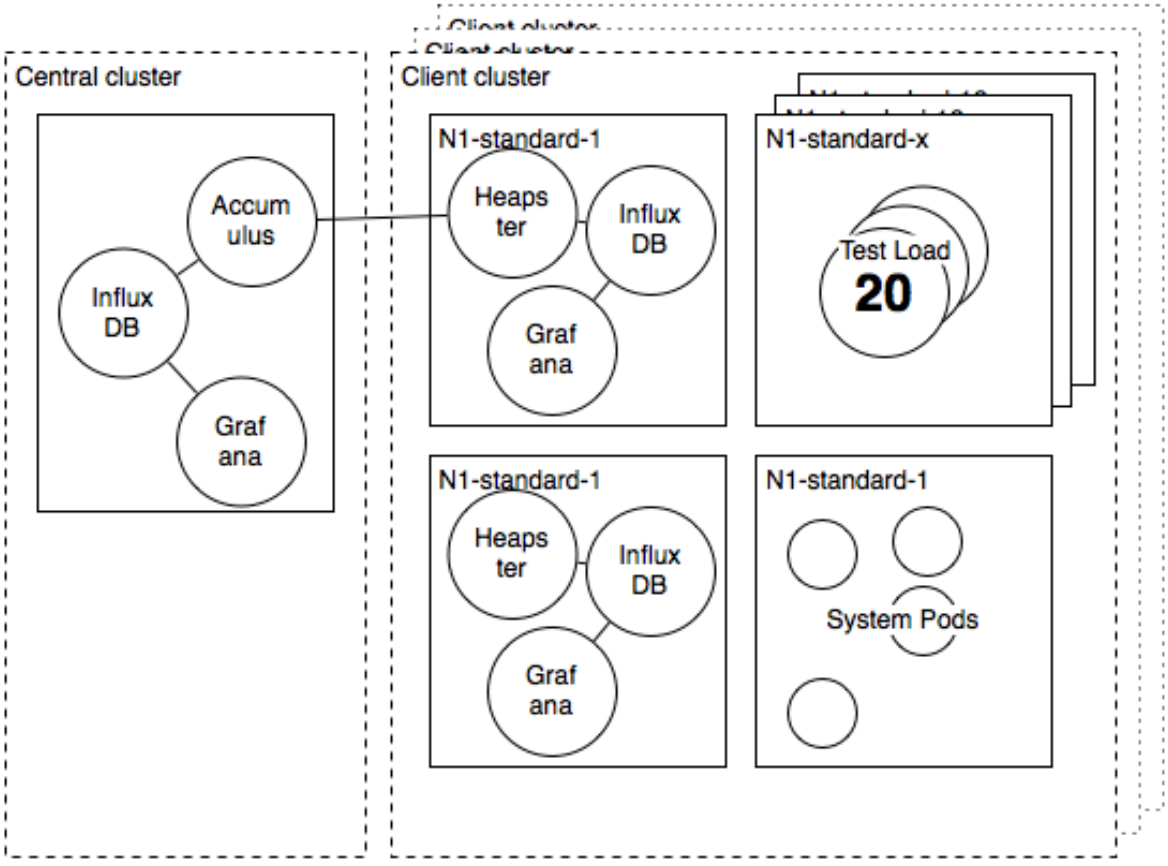


Figure 7.9: Scalability test setup, also showing the three parameters to be varied (pods/containers, nodes, and client clusters).

For each test, we wait until at least 90% of the expected containers is running (in order to account for failing containers), after which it is given 3 additional minutes to completely reach steady-state. The measurement then lasts 10 minutes, where every minute the total usage is measured by Heapster.

To minimize the interference between the different parts of the system, both versions of Heapster run

on their own node, as do the test load and any system pods which run inside the Kubernetes cluster. The only exceptions are system pods which have to run on every node, in our case Kube-proxy and Fluentd Cloud Logging.

All of the scalability measurements were done inside the europe-west-1d availability zone of the Google Container Engine, with a standard cluster consisting of nodes using the Haswell architecture with one virtual CPU each (N1-standard-1). The test pods consisted of 20 containers each (because of limitations in the Kubernetes scheduling, which would not allow more than 110 pods on a single node). The test container would retrieve at random intervals the HTML from one of 10 pre-defined web pages and write it to disk to simulate some network and disk usage.

7.4.2. Containers vs Resource Usage

This is the only test in which another instance than N1-standard-1 was used. This was done because the number of containers needed would not fit on such an instance, so instead an N1-standard-16 was used for the test load. All other instances were kept N1-standard-1.

From the Graphs 7.10 and 7.11, it can be inferred that both the CPU load and the RAM used scale linearly with the number of pods/containers, which is good news for the scalability. It can also be seen that the synced version causes significantly more CPU load and RAM usage, mainly in the Influx database pod, which is also responsible for the majority of the resource usage. Grafana was not used during the tests, and it is good to see its resource use was negligible.

7.4.3. Nodes vs Resource Usage

Test were run for 1, 2, 3, 4, 5, 6, and 8 nodes, each running 10 pods (200 containers). Originally testing with 10 nodes was also planned, but this was eventually deemed unnecessary and thereby cut to save on costs.

From the graphs 7.12 and 7.13, we can see similar results for the nodes as for the pods. RAM and CPU usage scale linearly, with the synced instance using significantly more than the unsynced instance, mainly at the Influx database pod, which is also responsible for the majority of the resource usage. Grafana was not used during the tests, and it is good to see its resource use was negligible.

Interesting to note is the RAM use of the synced Influx instance at 8 nodes, which is actually lower than the RAM use at 6 nodes. We expect this to be due to the fact that the instances used only had 3.75 GB of RAM and therefor either garbage collection, cache purging or some similar mechanism took place in the Influx database pod. This might indicate that the system is able to run on instances having less RAM than what would be expected from the linearly scaling graphs.

7.4.4. Clusters vs Resource Usage

In this test, the focus is on the resource usage behaviour of the central Accumulus instance. While we see a linear scaling pattern for the RAM usage, the CPU usage unfortunately seems to follow a less favourable scaling pattern. This is mainly due to the Influx database pod, as the CPU use of both Grafana and the Accumulus pod seem to scale linearly.

7.4.5. Scalability conclusion

From the container test and the node test it can be seen that the system scales well with the cluster size, while the cluster test shows that this is less so with the number of clusters to be synced. The majority of the resource usage stems from the Influx database pod, both in the client's Heapster part and the central Accumulus instance. As Influx database supports multi-core setups, it is trivial to scale the application further by simply adding more cores. There is even a decentralized version of InfluxDB available, allowing for even larger systems, but the downside is that this is proprietary at the time of writing.

Another remark that can be made is the fact that these tests were run on regular HDD's instead of SSD's. Seeing that the majority of the resource use stems from Influx database, we expect significant

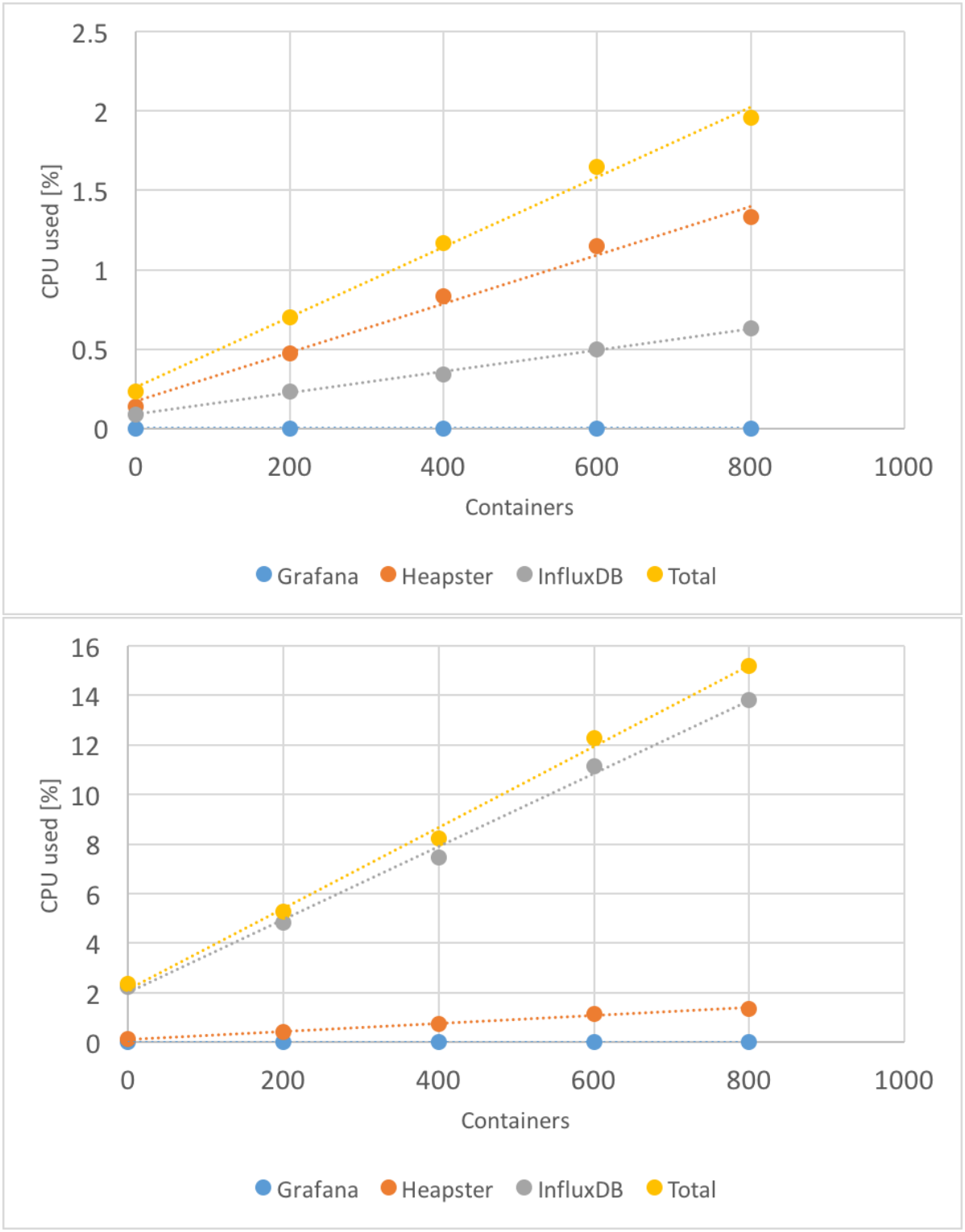


Figure 7.10: Containers in cluster vs CPU utilization for an unsynced (top) and synced (bottom) instance of Heapster.

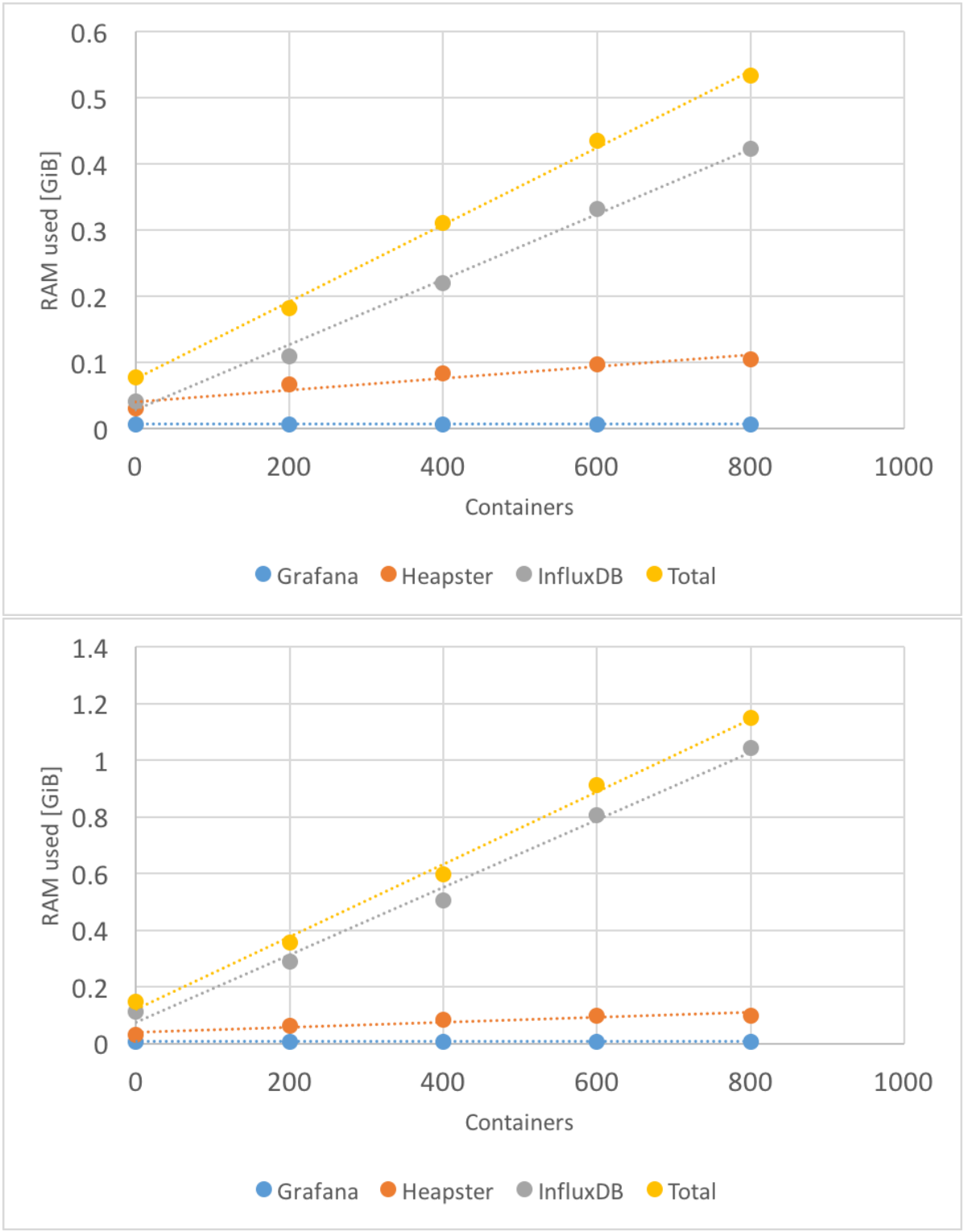


Figure 7.11: Containers in cluster vs RAM usage for an unsynced (top) and synced (bottom) instance of Heapster.

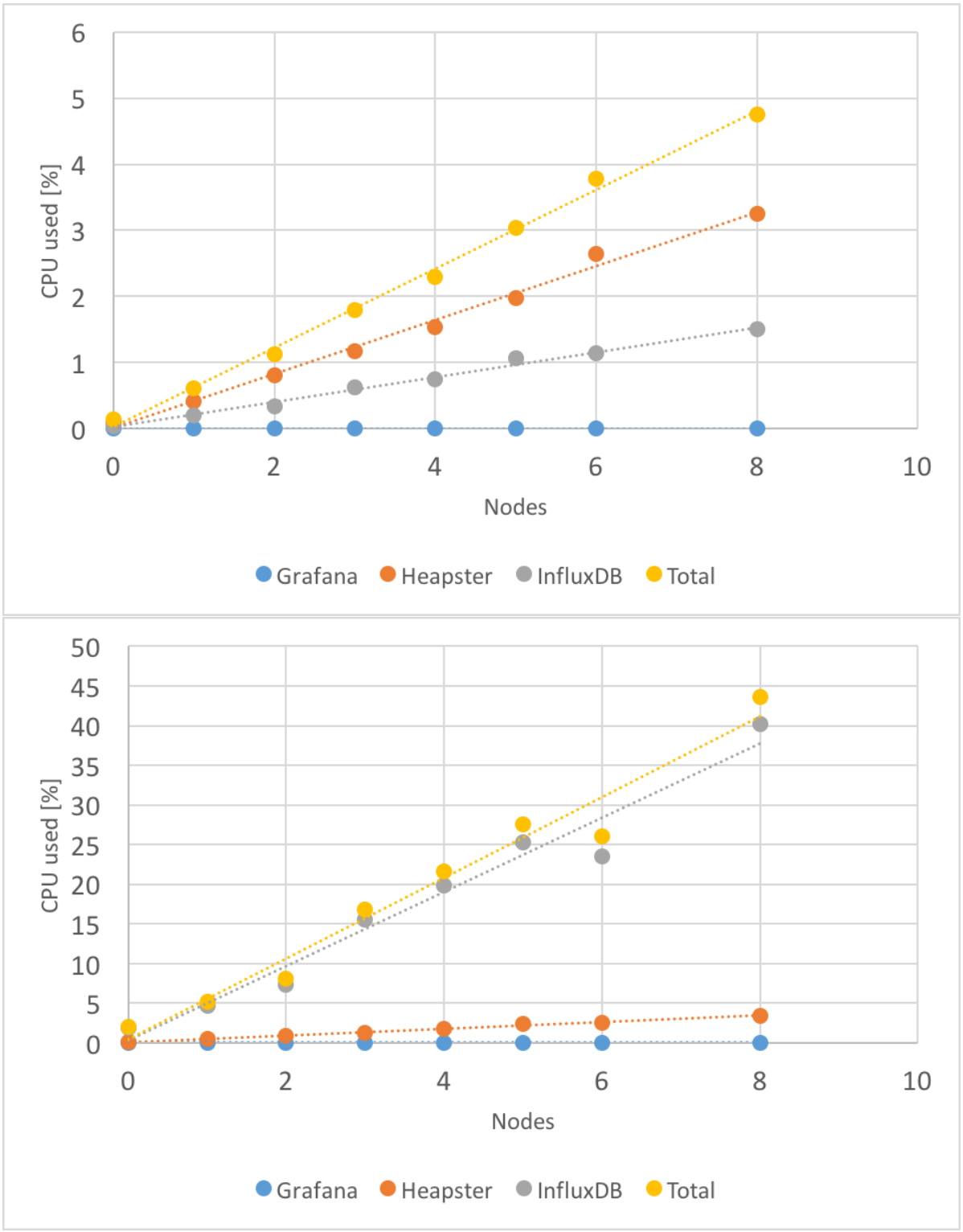


Figure 7.12: Nodes in cluster vs CPU utilization for an unsynced (top) and synced (bottom) instance of Heapster.

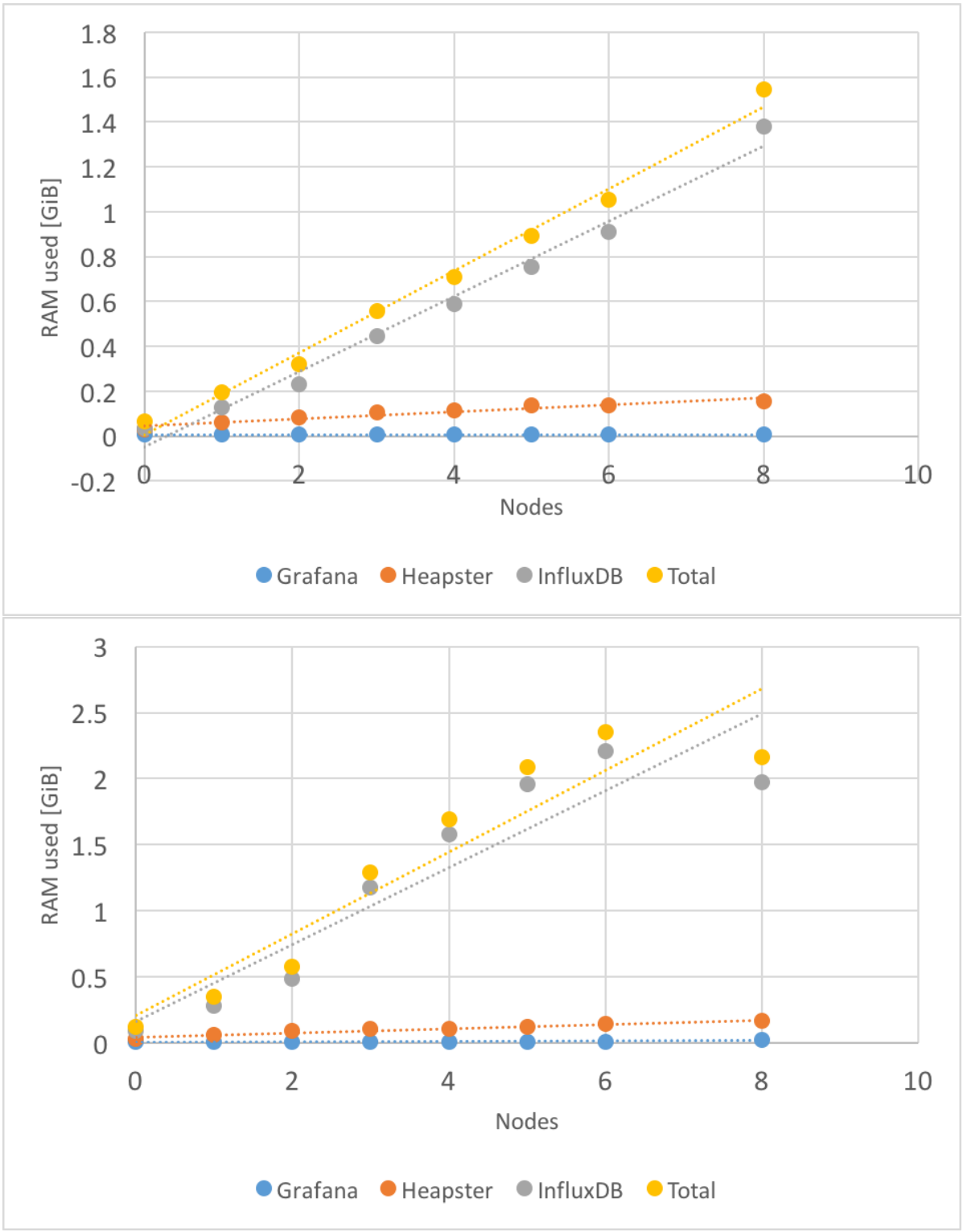


Figure 7.13: Nodes in cluster vs RAM usage for an unsynced (top) and synced (bottom) instance of Heapster.

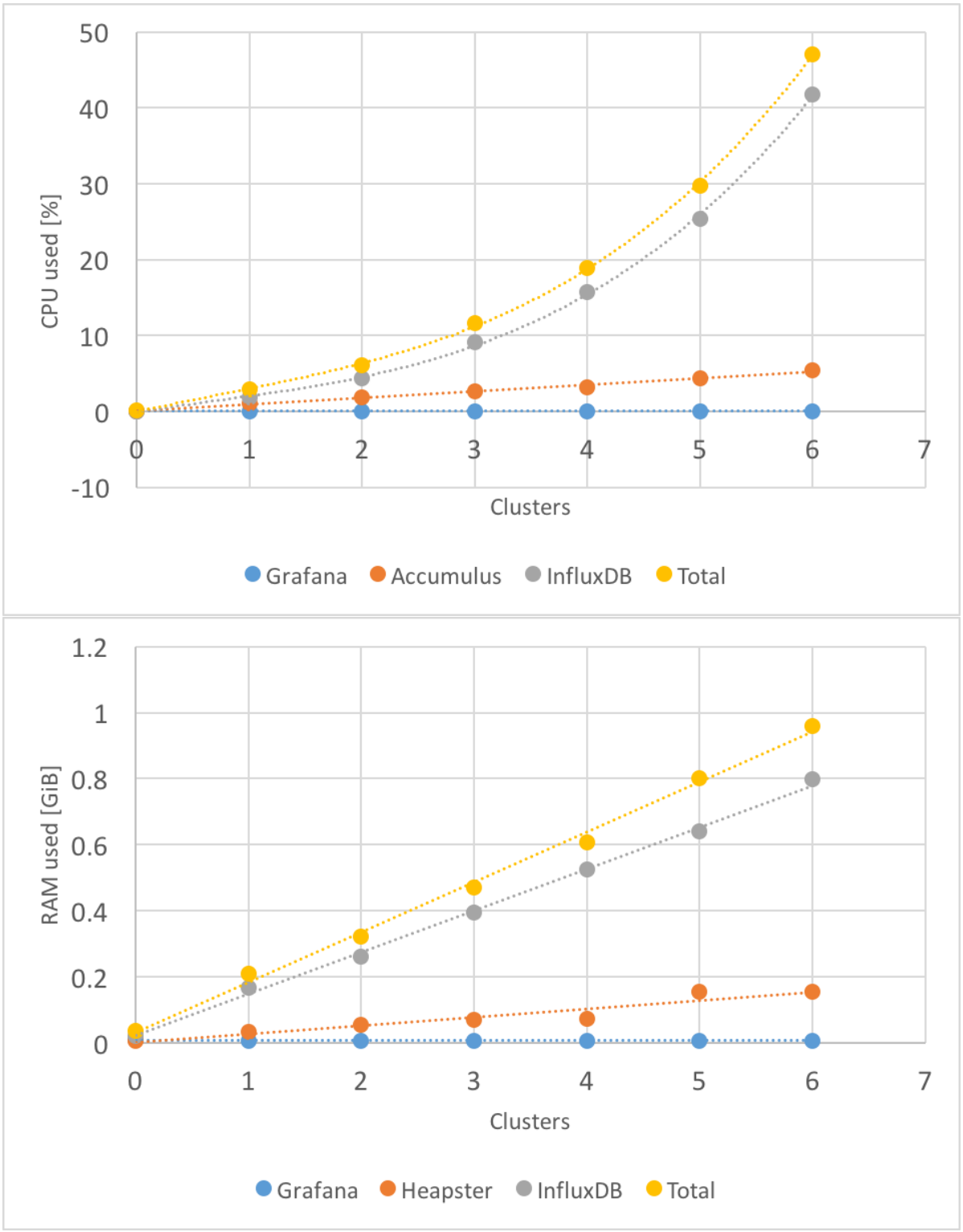


Figure 7.14: Clusters synced vs CPU utilization of Accumulus (top) and clusters synced vs RAM usage of Accumulus (bottom).

speed-ups if SSD's are used.

The scalability tests also showed some issues with the syncing mechanism, as the Influx database of a synced instance used up to 20 times as much CPU as that of an unsynced instance. We expect this penalty to stem from the fact that InfluxDB is optimized for writes, not reads (which are done during the syncing). Seeing that practically the same data is written in both the client database and the central database (excluding a possible downscaling of the data), this performance penalty should be easy to tackle. A naive solution would be to have Heapster write to two sinks, once to the local Influx database and once to the central Influx database.

Another issue that showed was the non-linear scaling with the number of clusters in the central Accumulus instance (or more precisely, its use of the Influx instance). We expect this issue to stem from the implementation details of the use of Influx database, as the amount of data does grow linearly with the number of clusters and should therefore result in linearly scaling resource usage at the InfluxDB pod.

Interesting to note as well is that all these conclusions stemmed from Accumulus system itself. This shows that performance problems and scalability issues are easy to locate in a micro-service architecture monitored by Accumulus. In fact, using Accumulus' cost processor, it should also be possible to calculate the costs of each issue on a yearly basis, thereby aiding the decision of whether development hours should be spent on fixing it.

8

Discussion & Future Work

In this chapter the project itself is discussed in Section 8.1, while Section 8.2 gives some possible directions for future research.

8.1. Discussion

This section discusses our experiences with doing this project and several of the challenges that were encountered.

8.1.1. Main Challenges

During the project, many challenges were encountered, in this section those challenges and how they were dealt with is covered.

Inexperience With Docker

Docker has become the standard container platform, it makes it easy to develop applications that can run anywhere. Packaging Docker containers, however, can be a hassle, the code needs to be compiled for the right operating system (Linux) and dependencies need to be packaged as well. After packaging moving the containers requires using a repository. As we had no prior experience with Docker this did cause issues.

Difficulty of Setting up Clusters

Setting up and having interaction between several clusters proved difficult, mainly due to different authentication mechanisms in use. During week 3 of the development, the decision was made to use Helm instead of the Kubernetes Go client to get the cluster in the desired state, which simplified the process significantly. Testing cluster setup in an automated way turned out to be time-consuming though, also because most CI tools were focused on having a single cluster (which is the intended use case of Kubernetes under normal conditions). It was eventually abandoned, meaning testing had to be done by hand.

8.1.2. Programming Language Experiences

Most of the Accumulus codebase is written in Go. Go is a young programming language developed by Google, dating back to only 2012. This section reflects on Go to give some insight in working with this new language. Writing Go is similar to writing C. Although its syntax is similar to that of C, it has more modern features such as garbage collection, dependency management, and built-in concurrency control. Go has no classes and instead uses structs, interfaces, and methods to organize code.

Working with Go was generally received favourably. It is very easy to create scalable concurrent programs in Go because of the ease of creating threads, called Go routines. Inter-thread communication is convenient using channels. The fact that Go is verbose also helped us as it makes code easily readable. Some other elements of Go were harder to work with, problem were mainly encountered with dependencies and race conditions, with some of these issues taking nearly a day to find and fix.

8.1.3. Changing requirements

During the project Nerdalize made many big decisions on how they want to provide their cloud service, these changes also impacted the project, requiring a redesign of the architecture and a change in requirements. Nerdalize initially wanted a system for billing purposes, however, they later changed the requirements so the system could be used for both monitoring and billing. This system could then give the user insight in his resource consumption and cost.

8.1.4. Ethical Considerations

In this section the ethical implications of Accumulus are discussed. For this, three ethical aspects applicable to Accumulus are considered, being confidentiality, privacy and transparency.

Confidentiality

Resource usage metrics can give insight in the sort of application running, as such this information is sensitive and it would be unethical to reveal this information to others. Accumulus handles this by only storing this information in the user's cluster and the central cluster, ensuring this information is inaccessible to other customers. As these metrics are collected with a time interval of 1 minute, their resolution can be considered to coarse for the cloud provider to infer anything about the specifics of the client's applications. The cloud provider still gets the names of the containers running in the client cluster, but if this is considered an issue it is relatively simple to give them randomized names.

The cloud provider on the other hand also has information they do not want to share but which is still necessary to process resource usage metrics. Accumulus has two ways of dealing with this data. The first is the SQL sync which ensures only data relevant to a cluster is available in a client cluster. The second is the central processing, because metrics are processed centrally the processors can contain company secrets and use sensitive data to process metrics without this data and secrets being at risk.

Privacy

A factor which is specific to Nerdalize's case is the privacy of the home owners. As mentioned under 3.3.2, the temperature measured by the Cloudbox could be used to infer information about the habits of the home owner, posing a privacy concern. As such it must be ensured that this information is inaccessible by Nerdalize's customers, which is done by processing this information in the central cluster.

Transparency

In cloud computing, the customer should always have a certain level of trust in the cloud provider, as it is often impossible to check if the provider delivered what he promised. While this issue is not solved by Accumulus, it should be trivial for the client to see whether or not the provider is altering the measured values. The customer can simply install his own version of Heapster and see if its measurements match those on the bill.

8.2. Future Work

In this section possible future work is discussed.

8.2.1. Testing in Production

The design and implementation described in Chapter 6 and the experimental work discussed in Chapter 7 describe the current state of Accumulus well. Accumulus has proven to be a working system, being able to monitor large clusters. It should however be noted that Accumulus is a proof of concept system and has not yet run in production environments. This section describes how Accumulus can be tested in a more realistic environment.

Since Kubernetes offers control over nodes, a good way to test Accumulus would be to deploy it on an actual cluster in use, while running it in its own namespace. This way it will not impose an additional load on the user's applications and could be left out for billing. This way Nerdalize or other cloud providers could experiment with different billing models and see if Accumulus supports them well. When successfully tested, the system could then be used in production by automatically deploying it on new clusters.

8.2.2. API Usage

As the used compute resources and their costs are available in near real-time via the API, programs could use this to analyze their own behavior and respond to this, enabling real-time optimization. This would be akin to the current shift towards a "smart power grid" (where resources are used much more efficiently),

Relatively simple implementations would be to schedule resource-intensive tasks during off-peak hours, choose a particular implementation based on the relative cost of RAM and CPU, or to increase caching when RAM gets cheaper. Since the database also includes the hardware specifications, systems could calculate the difference between different set-ups, enabling them to pick the most cost or speed efficient one. This would especially be interesting in a more heterogeneous ecosystem (such as a multi-cloud), where different cloud providers provide vastly different set-ups, possibly offering more exotic hardware such as ARM processors and GPUs (or even FPGAs and ASICs), instead of x86 based architecture.

8.2.3. Processors

Currently, there are two processors included in Accumulus, one calculating the costs based on the compute resource usage input, and one calculating the power usage of each process based on the CPU time used by the entire Cloudbox. Due to the flexible set-up, it is possible to add more processors or to replace the current ones with more accurate models.

The cost processor is currently using a very simple unit-pricing model, where every compute resource has a fixed price. A simple extension would be to increase the price during certain peak hours and to start charging different prices for different types of hardware. More rigorous data analysis combined with predictive models could be used to create a pricing model maximizing the profits for Nerdalize. Such a model would most likely also include incentives for clients to do computations during off-peak hours, in order to achieve a greater utilization of the hardware.

The processor calculating the power is currently based on a model consisting of two separate parts, being the fixed base power consumption (which is taken from a Cloudbox with all motherboards idling), and the variable power consumption which is split over processes according to the CPU time used. A slightly more accurate model could have two base consumptions, one per Cloudbox and one per motherboard, as there is no fixed amount of motherboards per Cloudbox. Other models might introduce other variables into the calculation, such as internet usage (which starts using significant power at speeds over 10 GB/s), or the temperature of the Cloudbox (as the PSU's loss increases with its temperature). Another improvement could be made in handling motherboards that are hyper-threading, which uses relatively less power.

A processor which could be added would be the heat actually used by the home owner. For this, extra sensors are most likely necessary within the Cloudbox, as currently only two temperature sensors (one before and one after the heating element) are installed. This processor could be used to bill customers for the warm water they've used if this is desired.

8.2.4. Alarms And Triggers

Grafana includes alert functionality, which can show a notification within the GUI based on certain rules. These rules could be set to trigger on a certain degree of utilization of some resource, if data has not arrived for a certain amount of time, or based on spikes in the internet traffic indicating anomalous behavior (such as (D)DoS attacks). Grafana also supports sending notification via e-mail, Slack, PagerDuty or using custom webhooks. This last option would allow for further automated responses to triggers, such as dynamically rescaling the cluster based on node utilization or rescheduling certain tasks during busy hours.

An option that seems to allow for even more flexible rules, while also promising a wider range of responses is Kapacitor, which is part of the TICK stack mentioned under research. As Accumulus uses InfluxDB as the database, it should be trivial to plug an instance of Kapacitor into the database, enabling its functionality.

9

Summary & Conclusion

In this chapter, a summary of the entire project is given and a conclusion is presented.

9.1. Summary

In this thesis we presented Accumulus, a Monitoring and Cluster Analysis System developed to monitor Kubernetes clusters. The processes used to create Accumulus, the research, design, and implementation were covered, while experimental work was presented to show the scalability and accuracy of Accumulus.

The development of Accumulus started with two weeks of research. During these weeks several of Nerdalize's employees were interviewed. Besides interviews, a literature study was conducted to gather knowledge about Go, Kubernetes, and Docker. The seven weeks after the research period were used to design and implement Accumulus. To manage the product development, the AGILE development methodology Scrum was used. In order to make working as a team easier, GitLab was used for version control. The last week of the project was spent on finishing this thesis and preparing a demo.

The design of Accumulus has led to a fully functional cluster monitoring system with flexible processors that can be used to get insight in the resource usage and the accompanying costs for both the end user and the cloud provider. Accumulus monitors CPU, RAM, disk, and network usage and presents those metrics as visual graphs and numbers to users. The processors can generate additional metrics such as power use, heat generation, CO^2 savings, and cost.

To verify the accuracy and scaling capabilities of Accumulus, a series of experiments was conducted. The basic validation experiment shows that Accumulus is accurate enough for both user and billing purposes. The scaling experiment shows that Accumulus can scale to over 1000 nodes when Accumulus and the InfluxDatabase are run on sufficient hardware and several caveats are taken into account. Overall, the Accumulus project delivers promising results in production-like settings.

9.2. Conclusion

The research questions presented at the beginning of this thesis can now be answered:

1. **How can the resource usage of a job running on a container-based cloud be measured?**
   To measure resource usage on a cluster we designed and implemented Accumulus, Accumulus measures CPU, RAM, disk and network usage via Heapster, and those metrics available in a central InfluxDatabase.
2. **Can power consumption and heat production be measured on container or pod level?**
   From our experimental work, we learned that a function with a good model for power consumption

based on CPU usage can give an accurate approximation of the power used by containers and pods. Accumulus comes with a processor that does this for Nerdalize's Cloudbox.

3. **Can charging users on a per job basis make pricing more transparent?**
   If the pricing model of the cloud provider can be translated into a processor for Accumulus the user can get a real-time view of the cost of his job, making the pricing transparent. Acculumus has a processor with a simple unit-pricing model that calculates the cost, this processor can be adapted to fit other billing models.

In Section 3.3 we listed a set of requirements for our system, Table 9.1 shows the requirements and indicates which requirements are met. Requirement 7 was not met as this would not be possible in an efficient manner with the current setup of Nerdalize's network.

#	Priority	Met	Description
1	must	Yes	The system measures accurately and with an overhead that is 10% or less.
2	must	Yes	The system can run without any manual interference needed.
3	must	Yes	The system measures the usage, request, and limit of all required metrics.
4	must	Yes	The system records the machine and job metrics belong to.
5	must	Yes	Measurements are retrievable via an API.
6	should	Partial	The system measures the usage, request, and limit of all optional metrics.
7	should	No	The system differentiates between internal peering and outward internet network traffic.
8	should	Yes	Local measurements are sent at short intervals in order to analyze resource usage on failing nodes.
9	should	Yes	The system uses retention policies to reduce storage usage.
10	should	Yes	Collected measurements are stored centrally for historic and billing purposes.
11	could	Yes	The system also measures network traffic between Cloudboxes.
12	could	Yes	The system can make predictions on resource use of a job that is running.
13	could	Yes	Resource usage is not only accessible via the API but can also be viewed in a GUI.
14	could	Yes	The system generates a report when a job is finished.
15	wont	No	The system is integrated with the Nerdalize Cloud Engine's user interface.

Table 9.1: Accumulus requirements and indications which requirements are met.

A

Sprint Plans

A.1. Sprint 1

Sprint 1

✓	User Story	Task	Assigned To	Estimated Effort	Priority	Actual Effort	Notes
			161/160 hours			162/160 hours	11/13 completed
x	**As a developer I want a code repository so I can easily collabborate and have version control**	**Set up git repository**	**Diony**	4	A	4	**Two gitlab repositories have been set up**
x	**As a developer I want to do integration testing to ensure all software components integrate well**	**Set up continuous integration**	**Youp**	6	A	24	**Setting up continuous integration took longer than expected due to the complicated environement we need for testing**
x	**As a developer I want a kubernetes cluster to test and run my system on**	**Set up a kubernetes cluster**	**Both**	2	A	6	**Setting up a kubernetes cluster on google cloud took longer as we had to await approval from the company. clusters cost money**
x	**As a developer I want to test my code**	**Set-up unit testing framework and create a unit test**	**Youp**	8	A	10	**We had no experiance with gitlab's integrated CI, because of this setting up unit testing took longer**
x	**As a developer I want a test workload with a predictable load to validate the accuracy of my system**	**Create a test job for personal testing**	**Diony**	4	B	4	**test job is writen in go and has a predictable resource load.**
x	**As a developer I want to be able to quickly depoy my code**	**Set up deployment file for Kubernetes for automatic testing**	**Youp**	14	B	16	**a small issue with authentication took aditional time**
x	**As a developer I want to have a clear model of what I'm going to create**	**Create model and document it**	**Both**	16	A	20	**A lot of important decisions where made at nerdalize during this sprint, as such our architecture design changed multiple times**
x	**As a developer I want to be able to correlate data (e.g. specifications and client data)**	**Set up SQL database and populate it**	**Youp**	8	B	2	**Done, nerdalize already has a system that contains this information**
x	**As a user I want to see the resource use of jobs running on kubernetes**	**Monitor CPU, RAM, disk and network use**	**Diony**	25	B	20	**All metrics are already available in heapster with the exception of Disk IO, we found a solution for this but have problems implemented it.**
x	**As a user i want to be able to select metrics based on tags such as machine id and the job they belong to**	**Split resource usage correctly using tags**	**Diony**	30	A	12	**Most of the tags are in place. Some tags rely on our architecture design and have not been implemented due to the changes in this design**
x	**As a developer I want to know which network statistics are available**	**Investigate possibilities for network usage monitoring**	**Diony**	12	C	8	**In a meeting with nerdalize's network engineer we discoverd that not all network statistics we want are available, a new set of metrics was selected.**
	As a user I want to know how much power my cluster consumed and how much heat this provided	Store power and heat consumption	Both	16	B	4	A change in architecture design changed the way we want to do this, we did however research how this can be done and now have acces to the right database.
	As the billing department, I want my billing software to be able to receive usage data	Design API for querying data	Diony	16	B	10	Changes in our architecture design caused the API part of our project to be deprioritised as both influx and grafana have a good API and will contain all neccesary data in our new design.
	As a student I want to deliver a well written research report	Write research report	Both	-	B	8	Due to late feedback on our first draft we still had to work on the research report

Main Problems Encountered

1	**Changes in architecture design** A lot of important design decisions where made by nerdalize during this sprint, those decisions had a lot of influence on how the environment our system needs to run in looks, as such we had to make changes to our architecture design.	8	We redesigned our architecture design twice and discused it with multiple members of nerdalize
2	**Inexperience with Gitlab CI and Kubernetes authentication mechanisms** Inexperience with the Gitlab CI resulted in a long setup process for our CI, in the end we had to do a step back and do less thorough integration testing as setting up a cluster every time the code has to be tested was not deemed viable.	6	We tried to get CI working on a very complex set up, in the end we shouldn't have invested as much time in this.

Figure A.1: Sprint 1

A.2. Sprint 2

Sprint 2			154/160 hours			185/160 hours	14/15 completed
✓	User Story	Task	Assigned To	Estimated Effort	Priority	Actual Effort	Notes
x	As a developer I want to have a clear vision of what I need to do	Sprint review and planning	Both	2	A	2	
x	As a customer I want to be able to automically deploy Heapster on (new) clusters	Make Cluster startup function	Youp	16	A	25	Eventually the decision to use Helm (a Kubernetes package) instead of programming the logic ourselves
x	As a customer, I want Heapster to add tags according to my current ecosystem	Setup central SQLite file	Youp	6	A	4	
x	As a customer, I want Heapster to have information on the nodes the cluster consists, without having to include all data	Make SQL Sync logic	Youp	20	B	30	Various problems were encountered by initializing containers using several files
x	As a customer, I want Heapster to add tags according to my current ecosystem	Create database schema	Youp	4	A	2	
x	As a customer I want the communication with my cluster to be secure	Research authentication mechanisms	Youp	10	A	10	
	As a developer I want to test and deploy my code in an automatic manner	Set up CI	Youp	12	A		The idea turned out to be too time-consuming with minimal benefits. As such, the idea was abandoned
x	As Accumulus I want to be able to pull and push data between databases	Make pushing and pulling of influxdb data possible	Diony	8	A	12	Took longer than expected: The available API did not contain all neccesary fucntions.
x	As Accumulus I want data to be downsampled before it is stored in the core	Add continuous query to downsample data	Diony	4	4	4	We found a way to do everything with a single downsample query
x	As Accumulus I want to have the heat and power metrics of cloudboxes available	Pull heat and power metrics in central Influx	Diony	2	B	4	Took longer, as for our test cluster this data is not available and thus has to be simulated
x	As a developer I want to know how cost and power usage are distributed over the components of a cloudbox	Research the cost and power usage of a cloudbox	Diony	4	B	2	Since a cloudbox was not avaiable at the time we used some random generated values and estimates to test with
x	As a user I want to see how much power my jobs have cost	Create a power processor	Diony	8	B	12	Writing a function to structure the data in a good way took some time as our influx functions were not flexible enough yet
x	As Accumulus I want to have all metrics tagged with the cloudbox and hardware they are measured on	Tag all metrics with the cloudbox and specs	Diony	10	A	8	Done with the exception of the sql connection
x	As Accumulus I want to store metrics for billing and historical purposes on a central DB	Sync influx databases	Diony	10	A	16	Took more time due to a lot of refactors to make the functions more versatile
x	As a Developer I want to show my system to the client so I can collect feedback	Prepare demo	Both	10	C	8	Demo was recieved really well, good feedback was given
x	As a student I want to deliver a well written thesis report	Work on the thesis report	Both	30	B	30	

Main Problems Encountered		
1	**Understanding how Heapster works took a lot of time**	8
	Heapster is a complex piece of software with support for a lot of sinks and sources, as such understanding how it works took a lot of time, in the end we got a clear view of how the parts relevant to our project worked	
2	**The code we were writing for the cluster startup function turned out to look a lot like Helm**	10
	As such, we decided to switch to Helm, a decision which is likely to save us time in the future	

Figure A.2: Sprint 2

A.3. Sprint 3

Sprint 3			146/160 hours			92/160 hours	8/15 completed
✔	User Story	Task	Assigned To	Estimated Effort	Priority	Actual Effort	Notes
x	**As a developer i want to have a clear vision of what i need to do**	**Sprint review and planning**	**Both**	**2**	**A**	**2**	
x	**As a user i want to see the cost of applications and my cluster**	**Create a cost processor**	**Diony**	**8**	**B**	**6**	**For now a simple unit cost model is used based on the current cost of a Cloudbox**
	As a user i find it interesting to see how much co2 emmission i saved by running on a Cloudbox	Create a CO2 processors	Diony	4	C		
x	**As a user I want to be able to deploy Accumulus**	**Create a helm deployment for Accumulus-central**	**Youp**	**4**	**B**	**4**	
x	**As a user I want to have a same codebase**	**Clean-up deployment procedure**	**Youp**	**4**	**A**	**4**	
x	**As a cloud provider i want Accumulus to install everything on user cluster i add**	**Create a deployment function for user clusters**			**B**		
	As a student i want to deliver a well written thesis report	Finish the thesis report	Both	36	A		
x	**As a user i want the calculated power usage tho be accurate**	**Do tests to verify the accuracy of the power processor**	**Diony**	**4**	**B**	**4**	**The test confirmed our hypothesis that CPU is a good measurement for powerusage, and our processor should be accurate enough.**
x	**As a user i want Accumulus to be accurate**	**Do smoketests to verify that heapster is accurate**	**Diony**	**2**	**B**	**3**	**The tests show that heapster and Cadvisor are accurate enough for both monitoring and billing purposes**
	As a developer i want to know the effects of Accumulus on a user cluster	Test the overhead Accumulus has on a cluster	Youp	20	A	15	
x	**As a developer i want to know how well my software scales**	**Test the scalabilty of Accumulus and Heapster**	**Youp**	**20**	**A**	**35**	**Several problems were encountered with the tests, leaving some of their data useless**
	As a developer I want to give a demo	Create demo	Both	12	A	15	
	As a student i want to give a good presentation for my bachelor thesis defence	Prepare a presentation	Both	12	A		
	As a student i want to defend my bachelor thesis	Prepare for the defence	Both	16	A		
		Create all dashboards	Diony	2	B		
x	**As a developer i want my code to be tested well**	**Write tests for our code**	**Both**	**12**	**B**	**4**	**Due to inexperiance with Go and testing code in Go our code is not really testable**

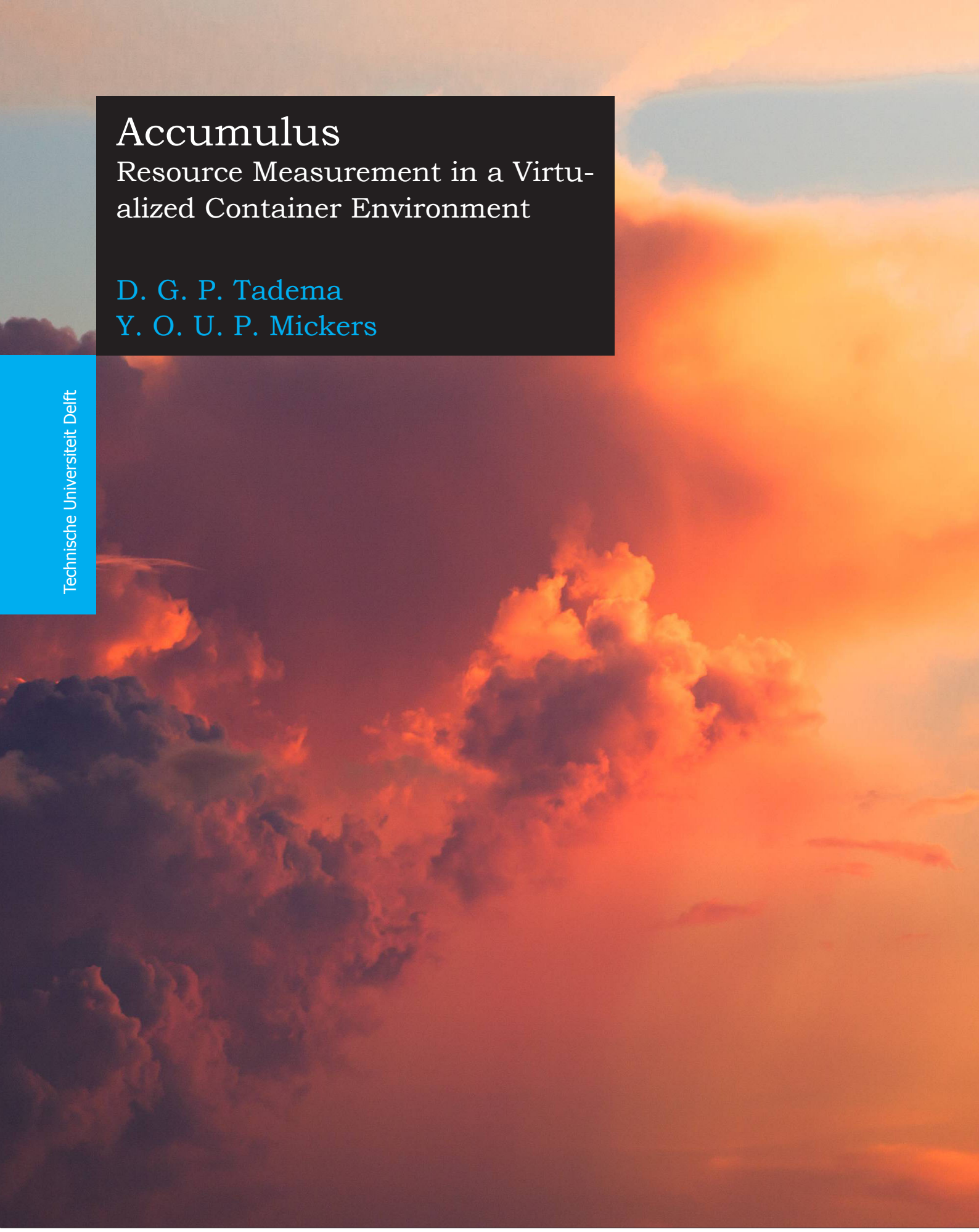
Main Problems Encountered

1. **Testing go code**
   We Delayed testing our code because we where inexperianced with testing in Go, due to this the written code is not testable.

Figure A.3: Sprint 2

B

Research Report



Accumulus

Resource Measurement in a Virtualized Container Environment

D. G. P. Tadema
Y. O. U. P. Mickers

Technische Universiteit Delft

TU Delft Delft University of Technology

Challenge the future

Version	Date	Reviewers	Remarks
0.1	22 December 2016	Authors	First draft (no complete report yet).
0.2	5 January 2017	Authors	Version created for Nerdalize internal use.
0.3	11 January 2017	Authors	Version handed in for revision.
0.4	11 February 2017	dr. ir. A. Iosup	Incorporated first feedback by dr. ir. A. Iosup.

Table 1: Versioning

Accumulus

Resource Measurement in a Virtualized Container Environment

by

D. G. P. Tadema
Y. O. U. P. Mickers

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,

Supervisor: dr. ir. A. Iosup TU Delft
dr. M. de Meijer, Nerdalize

An electronic version of this thesis is available at http://repository.tudelft.nl/. Version 1.2
07-02-2017 (Improvements after first revision by dr. ir. A. Iosup)

TU Delft Delft University of Technology

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Problem Statement . . . 1
1.3 Approach . . . 2
1.4 Structure . . . 2

2 Background **3**
2.1 Concepts . . . 3
2.1.1 Virtual Machine . . . 3
2.1.2 Container . . . 3
2.1.3 Cloud . . . 3
2.1.4 IaaS . . . 3
2.2 Technologies . . . 4
2.2.1 Docker . . . 4
2.2.2 Kubernetes . . . 4
2.2.3 Go . . . 6
2.2.4 Cloudbox . . . 6
2.2.5 NCE . . . 6
2.2.6 Heapster . . . 7

3 Problem Analysis **8**
3.1 Problem Definition . . . 8
3.2 User Stories . . . 9
3.3 Requirements . . . 9
3.3.1 Functional Requirements . . . 9
3.3.2 Metrics . . . 10
3.3.3 Non-functional Requirements . . . 11

4 The research and development process **12**
4.1 Research . . . 12
4.1.1 Before the Project . . . 12
4.1.2 During the Research Phase . . . 12
4.2 Development . . . 13
4.2.1 Development Strategies . . . 13
4.2.2 Programming Languages . . . 14
4.2.3 Repository . . . 15
4.3 Systems . . . 15
4.3.1 Data Retrieval and Storage . . . 15
4.3.2 Data Visualization . . . 16
4.4 Development - Chosen Development Tools . . . 17
4.4.1 Development Strategy: SCRUM . . . 17
4.4.2 Programming Language: Go . . . 17
4.4.3 Development Environment . . . 18
4.4.4 Code Quality: GitLab CI . . . 18
4.5 Development - Chosen Systems . . . 18
4.5.1 Data Collection: Heapster . . . 18
4.5.2 Data Retrieval and Storage: InfluxDB (and SQL) . . . 18
4.5.3 GUI: To Be Determined . . . 19
4.5.4 Data Visualization: Grafana or Chronograf . . . 19

5 Preliminary Design of Accumulus, a Monitoring and Cluster Analysis System 20
5.1 Architecture Overview . . . 20

6 Validation 23
6.1 Accuracy . . . 23
6.1.1 Compute Resources . . . 23
6.1.2 Power and Heat . . . 23
6.2 Basic Validation . . . 23
6.3 Overhead . . . 24
6.3.1 Overhead of Accumulus in a Timed Benchmark . . . 24
6.3.2 Overhead of a Monitored Accumulus Instance . . . 24
6.3.3 Difference in Overhead Based on Interval Length . . . 24
6.4 Scalability . . . 24

A Appendix A: Sprint Plans 26
A.1 Sprint 1 . . . 27
A.2 Sprint 2 . . . 28

B Project Proposal 29

C Requirements 34

Bibliography 36

1

Introduction

1.1. Context

Computers are used almost everywhere in today's world, and since the 2000's we have seen a shift from computations done on a local level to computations done in centralized data centers. A recent development is the shift from grids and data centers to clusters, where thousands of computers are connected via a network and can work together on a task[1].

While computers have certainly enabled us to solve problems previously deemed infeasible and improved our lives in countless ways, their increased usage comes at a cost. At the moment, estimates hold data centers accountable for 1-2% of the worldwide energy consumption. A number that is expected to grow significantly in the coming years, with data centers surpassing the airline industry in CO2 production[2]. As such, it is vital to make the use of computation power more sustainable.

One of the companies that are trying to achieve this is Nerdalize, who uses the heat produced by servers to provide houses with warm water and central heating. In participating households, the boiler will be replaced by a computing unit developed by Nerdalize. Such a decentralized cloud has lower operational cost than centralized competitors as the high cost of housing and cooling are mitigated[3].

Nerdalize is also trying to solve other issues in cloud computing. One of these problems is vendor lock-in, which it wants to solve by combining their cloud with other clouds to create a so-called multi-cloud, a single heterogeneous architecture that uses multiple infrastructure providers. A multi-cloud solves the issue of vendor lock-in, while also addressing geo-diversity [4].

While doing this Nerdalize is also coming up with solutions for the relative intransparent pricing models cloud providers currently offer. Cloud costs are hard to estimate before-hand, which is potentially scaring away customers. Earlier research has partly alleviated this problem [5], but billing is currently still done on a per-instance base, instead of on the actual resources used.

1.2. Problem Statement

Currently, most cloud providers bill their customers for the resources and instances they reserve or based on the number of times their function is called and the time taken by each called function[6]. These pricing models however vary between providers and machines as shown in figure 1.1, and it is hard to forecast performance on a specific instance. As a result, the market for cloud computing is far from transparent. Nerdalize aims to make cloud computing more of a commodity. One aspect of this is how customers are charged. In the future, Nerdalize wants to shift from a per instance to a per job billing model. New problems arise in such a billing model, resource usage needs to be accurately tracked and be available to the user to get insight into the performance and cost of their job. To charge customers on a per job bases a system is needed that accurately measures the resources used by a job with minimal overhead. While this task is relatively easy on a single computer, determining these metrics for a job which is split across multiple computers (each of which might also be working on

several other jobs at the same time) brings several complications. Some research has already been done in this field [7–10]. This has lead to a some great systems for resource monitoring, but these are not customer oriented, nor do they take into account the power used by the server performing the jobs.

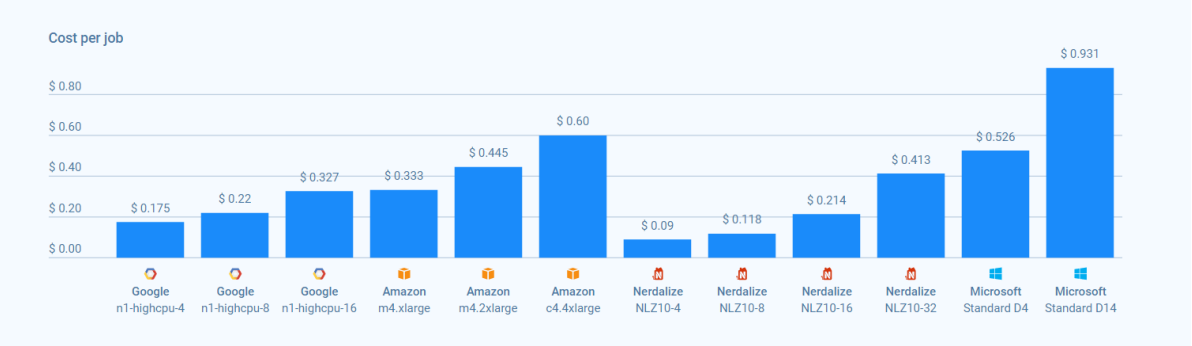


Figure 1.1: Pricing of different providers[11].

1.3. Approach

The goal of this project is to create a system that can accurately measure resource use in a container-based cloud system and make this information available to both the user and the cluster service provider. The development will start with two weeks of research, during which will be researched how cloud providers calculate costs and how resource usage can be accurately measured, especially in set-ups relying on Docker and Kubernetes. During the research, a preliminary design for a system will also be created. The five weeks after the research period will be used to design and implement a system that can help Nerdalize and others to accurately calculate the resource usage and thus the costs of jobs on a cluster infrastructure. The last three weeks of the project are used to test and validate the built system and write a thesis report.

1.4. Structure

This Report is structured as follows: Relevant concepts found in cloud computing are explained in 2.1, section 2.2 gives an overview of technologies used in the field of cloud computing and. Chapter 3 presents the main problem and high-level goals of the project. 3.3 defines a set of requirements created from obtained user stories, which a system solving this problem has to fulfill. Chapter 4 describes our process of research and development. The research done during the project is described in 4.1. Section 4.2 lists considered development strategies, programming languages and tools. The choices are supported in 4.4. A lot of existing solutions for resource monitoring already exist; section 4.3 goes into detail and section 4.5 explains which system was chosen as a starting point, also explaining this choice. Chapter 5 describes a preliminary architecture design of the system and explains the role of each element. Lastly, Chapter 6 shows how the system will be tested and validated.

2

Background

2.1. Concepts

This section defines some of the concepts in the field of distributed computing.

2.1.1. Virtual Machine

A virtual machine is a machine level virtualization method that provides an isolated environment that simulates a different system or architecture. This allows a single machine to run multiple different operating systems. [12]

2.1.2. Container

A container is an operating-system level virtualisation method that provides an isolated environment, simulating a closed system running on a single host. It gives the user the ability to have an environment to run applications with the necessary resources and environment configuration.

Containers are comparable to VMs (Virtual Machines), in that they are meant to decrease the problems often associated with deploying software on different servers or computers. They both do this by packaging all dependencies in a single environment, which can then be deployed across several computers.

The main difference is that VMs simulate a complete OS (operating system), with each VM having its own kernel, filesystem, and virtual hardware. Containers, on the other hand, run on a shared OS and Kernel, as seen in figure 2.1. Each container has its own isolated userspace but they share system resources. As a result, containers are often much smaller (MB's compared to GB's) and much faster to start-up[13].

2.1.3. Cloud

A cloud is a group of computers that work together connected by a network, usually acting as a single system or offering a single service. Users do not directly control the hardware, but instead rely on third-parties to fulfil their computation needs. The complex back-end of a cloud is often hidden from users and managed by cluster orchestration software such as Kubernetes or Openstack.[15]

2.1.4. IaaS

Infrastructure as a Service is one of the service models used in cloud computing. Just like other cloud service models, IaaS offers access to computing resource in a virtualised environment. In the case of IaaS, the computing resource provided is specifically that of virtualised hardware, in other

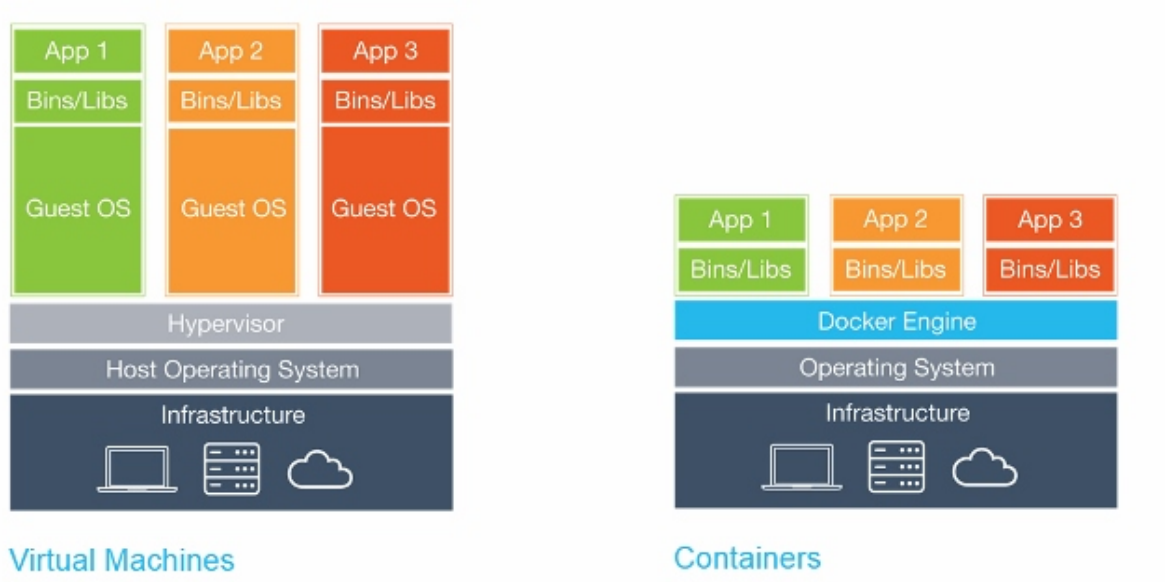


Figure 2.1: VM's versus Containers[14].

words, computing infrastructure. This includes virtual server space, network connections, bandwidth, IP addresses and load balancers. [16]

The cloud provider is responsible for maintaining the hardware infrastructure. The client, on the other hand, is given access to the virtualised components in order to build their own IT platforms.

2.2. Technologies

In this section technologies used in cloud computingg are explained.

2.2.1. Docker

Docker is an open-source container implementation. It offers both Linux (CoreOS) and Windows containers. Docker utilizes resource isolation via Linux kernel isolation technologies (cgroup and namespaces). They provide a great isolation for many applications. However, they do pose some risks in the case of multi-tenant environments. All containers on the same host share the same Linux kernel with that host[17].

Nerdalize will tackle this problem by creating a cluster for each customer, this way customers will never be able to access data belonging to other customers, thus ensuring security.

2.2.2. Kubernetes

Kubernetes is an open source system for automating deployment, scaling, and management of containerized applications or jobs on multiple hosts[18]. Kubernetes offers scheduling, replication control, and load balancing. It uses a state aware replication controller to handle failing machines or applications.

Nerdalize offers a cloud service for running compute-intensive workflows, with Kubernetes being used to manage the containers that make up such a workflow. Their in-house built workflow scheduler Flower[19] is used to ensure the workflow is executed in the right order and minimal heat is wasted.

A short description of several important Kubernetes concepts follows now and an overview of how these work together is shown in figure 2.2.

Node

Nodes are the workers in a Kubernetes cluster, as they execute the computations assigned to them. A node cannot consist of several computers, although a computer can host several nodes if desired.

Pod

Kubernetes works with pods, with a pod being an atomic unit containing one or more tightly coupled containers. Pods cannot be split over multiple motherboards as the containers in a pod is run on a single node.

Service

A pod is mortal, as they can be terminated by the scheduler if deemed necessary. As such, they cannot be relied on by external processes. The solution is found in a service, which is offered together by a collection of pods, thereby providing a reliable interface for communication[20].

Job

A job is the terminating counterpart of a service. A job creates one or more pods and ensures that a specified number of them successfully run. As pods successfully complete, the job tracks the successful completions. When the specified number of completions is reached, the job itself is complete. Jobs are useful for running workloads and batch jobs.

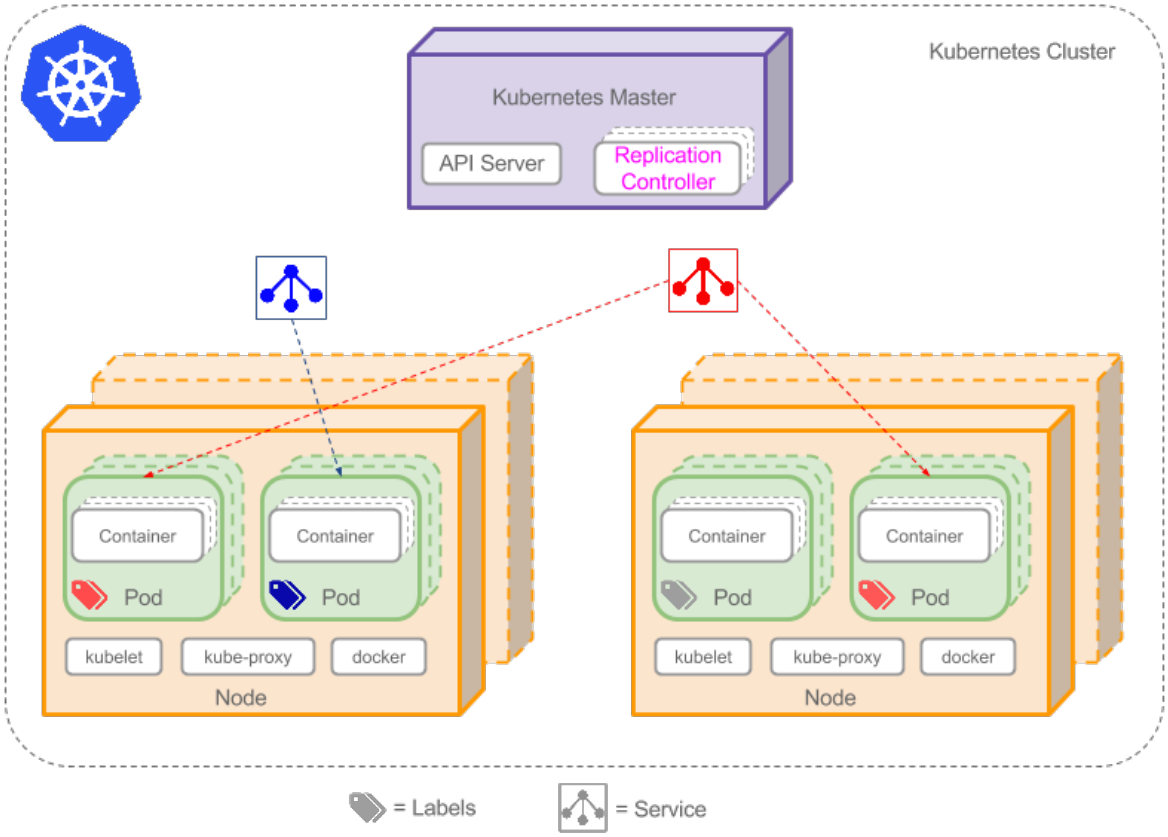


Figure 2.2: A Kubernetes cluster

Resource Requests and Limits

Kubernetes' scheduling is affected by two metrics, being the amount of CPU and memory. Aside from these metrics, cluster owners can use "Opaque Integer Resources" to define custom metrics. This is still in alpha as of yet. Kubernetes has notions of limits and requests for all metrics, with limits and requests being set per container. The resource limits and requests of a pod are easily determined, by simply summing those of all underlying containers.

Resource requests come into play before a pod is run and determine where the pod will be run. Whenever a pod is created, the scheduler will look for a suitable node for the pod to run on. Suitable here is defined as having the amount of available CPU and memory exceed the amount requested.

Resource limits, on the other hand, are important during a pod's execution. Whenever the memory used by a container exceeds the limit set, the container will be terminated. Whether the container will be restarted is determined by the "restartable" flag of the container. A container may or not be allowed to exceed the CPU limit set for extended periods of time, but will not be terminated for exceeding it. It follows naturally that a container's resource limit must exceed its resource request. [21]

(Persistent) Volumes and Persistent Volume Requests

On-disk files within a container cannot be shared between containers and are not durable. As such, they are lost whenever the container crashes. A solution to both these problems is found within Volumes, whose lifetime is equal to the pod enclosing them. A Volume subsystem provides storage to a single pod, independent of the underlying storage systems used, which are often vendor specific (e.g. Azure's file system or AWS elastic block store).

Volumes still seize to exist whenever their owning pod ends. As such, they are not suitable for sharing data within a job or even to save the output data. For this, a PersistentVolume is needed, which again provides storage independent of the underlying storage system used.

A PersistentVolumeClaim is the storage equivalent of a pod. Where a pod requests and consumes computing resources from the available nodes, PersistentVolumeClaims consume storage on PersistentVolumes [22].

2.2.3. Go

Go is a programming language developed by Google in 2009. [23] It is chosen as the programming language for this project mainly because both Docker and Kubernetes are written in it and because several employees of Nerdalize are already well-versed in it. For a more extensive motivation, we would like to refer you to section 4.4.

2.2.4. Cloudbox

The Cloudbox is developed by Nerdalize and serves as a housing for the computers, offering both physical security and ease of maintainability. It will be placed in people's homes and will provide hot water to the homeowner while providing compute for Nerdalize's customers. Inside the Cloudbox is space for 3 compute modules (initially one slot will be kept empty), each featuring 2 motherboards. The motherboards have 2 CPUs with 10 cores each, giving the Cloudbox a maximum of 120 cores.

2.2.5. NCE

The Nerdalize Cloud Engine is the main product Nerdalize is building, it will be a platform that allows engineers and users to communicate with Nerdalize's cloud infrastructure. It will eventually offer three ways to communicate with the Nerdalize cloud, a command line interface, an API, and a graphical user interface. With this Nerdalize aims to make running jobs as simple as possible. The NCE will handle authentication, data set storage, queueing of jobs, and control over workflows.

2.2.6. Heapster

Heapster is an open source tool for Container Cluster Monitoring and Performance Analysis. It is written in Go and is compatible with Kubernetes version 1.0.6 and up. Heapster measures various resource statistics on Container and Pod level and provides aggregate statistics for node and cluster levels. Those statistics can be stored in various back-ends, with the default being InfluxDB.

3

Problem Analysis

This chapter creates a definition of the problem Nerdalize has. From this definition, a problem statement is derived and the main question which will be answered during the project is formulated. Section 3.2 lists user stories which give a description of the system from the end users perspective. In section 3.3, requirements are defined, categorized, and ranked according to the amount of priority they deserve.

3.1. Problem Definition

Nerdalize is building its main product, the Nerdalize Cloud Engine (NCE). Their Cloud strives to be competitively priced, which they want to accomplish by placing their servers in homes with central heating. By cooling the server and reusing the heat to heat the home there is no need for expensive air-conditioned server space.

They also strive to make cloud computing more of a commodity market. To reach this goal they want to be more transparent in how they charge customers, this is done is two ways. Firstly Nerdalize wants to give the customer more insight in their resource usage, second, they want to charge their customers on a per job basis. In order to do this, they need a system that accurately measures resource use in a cluster. Resource measurement on a single system is trivial, but jobs on cloud services are run on a great variety of machines. Nerdalize's case has the additional difficulty of their machines being installed in houses where they are not easily accessible.

The high level goals of this project are:

1. Research how to measure resource usage on a Kubernetes cluster.
2. Research the current resource measurement systems Kubernetes offers (Heapster).
3. Build a system to measure and aggregate the resource use of a NCE job.
4. Extend the system to include the heat and power usage statistics of Cloudboxes.
5. Design an API to make statistics available to users and the Nerdalize system.
6. Design a simple GUI that shows these statistics.
7. Validate the built system against its requirements

From this problem definition, we are able to derive a problem statement. The statement is based on the problem definition by Nerdalize as well as the academic requirements specified by TU Delft. The derived statements differ from the original definition in that they are more concise and more abstract, omitting implementation details.

1. How can the resource usage of a job running on a container-based cloud be measured?
2. Can power consumption and heat production be measured on container or pod level
3. Can charging users on a per job basis make pricing more transparent?

3.2. User Stories

An interview with Nerdalize uncovered three actors

- **Nerdalize Compute Billing**
  Financial department that needs billing information to send invoices to customers.
- **Nerdalize Compute Sales**
  Sales department that charges customers for compute resources with a particular pricing model.
- **NCE User**
  Users of the Nerdalize Compute Engine

For each actor several user stories are given, for consistency they are all in the form of As <actor>, I want to <function or requirement> in order to <reason>

1. As the **Nerdalize Billing Dept.**, I want to **have data on resource usage** in order to **bill the customers**.
2. As the **Nerdalize Sales Dept.**, I want to **have data on resource usage** in order to **implement my billing models**.
3. As the **NCE User**, I want to **read accurate resource consumption per cluster, namespace, label, node and container** in order to **see my resource usage (and thereby expenses)**.
4. As the **NCE User**, I want to **read accurate resource consumption per cluster, namespace, label, node and container** in order to **have feedback on the performance of his jobs thus allowing him to optimize for runtime or cost**.
5. As the **NCE User**, I want to **read accurate predicted resource consumption per cluster, namespace, label, node and container** in order to **see my expected resource usage (and thereby expenses)**.

3.3. Requirements

3.3.1. Functional Requirements

Based on the collected user stories we created the following functional requirements. They are prioritized using the MoSCoW model

Must Haves

1. The system measures accurately and with an overhead that is 10% or less
2. The system can run without any manual interference needed
3. The system measures the usage, request and limit of all required metrics.
4. The system records the machine and job metrics belong to.
5. Measurements are retrievable via an API.

Should Haves

6. The system measures the usage, request and limit of all optional metrics..
7. The system differentiates between internal peering and outward internet network traffic.
8. Local measurements are send on short intervals in order to analyse resource usage on failing nodes.
9. The system uses retention policies to reduce storage usage.
10. Collected measurements are stored centrally for historic and billing purposes.

Could Haves

11. The system also measures network traffic between Cloudboxes.
12. The system can make predictions on resource use of a job that is running
13. Resource usage is not only accessible via the API but can also be viewed in a GUI.
14. The system generates a report when a job is finished.

Won't Haves

15. The system is integrated with the Nerdalize Cloud Engine's user interface.

3.3.2. Metrics

For the purpose of clarity, the used metrics are explained in-depth here instead of in the MoSCoW-model.

For every pod, it is also important to register the instance on which it was used. This is to be able to determine for example whether RAM usage was DDR3 or DDR4 and on what type of CPU the CPU time was used. For more advanced cost analysis nodes could even indicate whether the energy they received was useful to them (i.e. was the heated water likely to be used soon). This draws interesting parallels with an earlier research project done at Nerdalize[19].

CPU Time Usage

It should be known how much CPU time is spent on a certain job. A distinction should be made between CPU time reserved, and actual time used (as a reserved CPU does not consume energy and therefore has a different cost profile).

RAM Usage

Both the reserved RAM and the total amount of RAM used should also be measured. Contrary to the CPU use, reserved RAM is equivalent to RAM actually used, as reserved RAM is unusable by other jobs and does not differ in any way.

Disk I/O and Space

For disk usage, it is important to know both the total amount written and read from the disk (I/O) and the total amount of space reserved/used on the hard drive by a job. A separation should be made between local disk usage and central disk usage.

Network Traffic

In network traffic, three separations can be made. First of all, there is the traffic inside the Nerdalize network (e.g. Cloudboxes' local drives receiving their data from the central server or Cloudboxes peering data to each other). Secondly, there is the traffic through the Internet Exchange, where internet companies can send their data directly to each other by using the Border Gateway Protocol instead of the "regular" internet (e.g. exchanging traffic with selected cloud providers). Finally, there is the normal internet traffic, which would be the most expensive form.

Uptime

It is also important to know the total uptime of a machine, this allows users to see when machines are added or removed from their cluster and allows Nerdalize to see when Cloudboxes go offline. Measuring this metric also allows Nerdalize to refund resource usage on a failed Cloudbox.

Heat (optional)

Nerdalizes Cloudbox produces heat that can be used to warm houses. Although currently not planned it is a possibility that in the future Nerdalize may want to charge households for generated heat, in this case, it is important to have statistics of how much heat a job produced and how much of this heat is used. Nerdalize may then chose to discount jobs which created useful heat.

Privacy-related issues require special care here. From the test setup in Nerdalize's office building, it already became apparent that possibly sensible information could be retrieved from the temperature measurements.

Power (Optional)

Like heat, power is another real-world factor participating and should be taken into account in the cost analysis. Privacy is less of an issue here, as power consumption is dictated by computation need and to a lesser extent by the homeowner. However, power consumption could still reveal heat consumption to some extent, therefore requiring some care.

3.3.3. Non-functional Requirements

Apart from the functional requirements, the interviews have uncovered several non-functional requirements

Performance

The developed system may only impose a very small overhead on the jobs that need to be run in the cloud. If the overhead is too large, the system effectively becomes useless, as it will introduce a computational, and therefore monetary burden far outweighing the benefits.

Accuracy

A measuring system is only useful if its measurements are accurate. In Nerdalize's case, this is defined as having measured values deviate less than 10 per cent from the actual value. For a more in-depth discussion about accuracy we would like to refer you to chapter 6.

Open Source

The developed system or parts of it could be available as open source. This is required to collaborate with (experienced) people from the Kubernetes community who may help out with difficult design decisions as well as provide valuable feedback. It also allows other people to report bugs in the system and collaborate code or documentation speeding up development. The most interesting part for this approach will be the resource measuring part of our system as it will most likely be based on the open source software Heapster.

Maintainability

Maintainability measures the ease with which developers are able to make changes to the system. This is important because when the project is finished, less time is available to maintain the system. If it is easily maintainable and open source, other developers are able to take over without having to extensively study the code. This is beneficial to Nerdalize as well as other users of the system as it is more likely kept up to date.

Flexibility

As the field in which Nerdalize is operating is relatively new, it is bound to change quickly. Both Kubernetes and Docker are under heavy development, in which the direction that will be taken is not always clear yet. As such, the entire project needs to be setup in such a way that any changes can be conducted easily and without requiring major revisions to the code.

4

The research and development process

4.1. Research

In this section, the research processes that were used to gather necessary information for the design and implementation of our system are described. This section is split up in Section 4.1.1 and Section 4.1.2, which respectively describe the processes before the project started and the processes during the project

4.1.1. Before the Project

Two weeks before the official start of the project a meeting with Nerdalize took place in which the subject of the project was decided. Mathijs from Nerdalize explained several of the technologies in use at Nerdalize and that the would need to get familiar with. None of the project members had any prior knowledge of the container ecosystem and distributed systems in general. To get familiar with Docker two books were read.

Kubernetes is the system Nerdalize uses to manage their clusters, to get familiar with Kubernetes books were read and a tutorial provided on the Kubernetes website was followed.

Go is the language Docker and Kubernetes are written in and is also used by a lot of Nerdalize's systems. Videos were watched about Go and tutorials were done to get a better understanding of Go and its concepts.

4.1.2. During the Research Phase

Most of the research was done in the first two weeks of the project as a clear view of the project was needed. In this research phase multiple types of research were used: interviews, literature studies, digital media, and running test systems.

Interviews

During the project multiple interviews with different people from Nerdalize took place. Those interviews helped to get a better view of the requirements for the system as well as getting familiar with how Nerdalize organizes their cloud infrastructure. An overview is shown in table 4.1.

In an interview with Mathijs, the planned structure and architecture of the Nerdalize Cloud Engine was discussed and all systems in the NCE were explained. This showed where our system would be and how it should integrate.

A second interview with Thirza from the Nerdalize billing department was more focused on what the billing department of Nerdalize needs and how our system can provide this information. This helped with selecting a set of metrics for the system.

The third interview with Boaz from Nerdalize's sales department gave insight in how Nerdalize wants to sell cloud computing and which data is necessary for doing so. This interview also sheds light on how usage date could be shown to the end user which is useful when designing a GUI for the system.

In another interview, Nerdalizes network engineer helped define the network metrics that are available and how they could be measured. The conclusion of this interview was that not all initially selected network metrics can be measured, thus a smaller set of metrics was selected for the project.

A second interview with Mathijs showed the preliminary architecture design. This allowed flaws to be discovered and improvements to be made. The improved preliminary architecture design addresses issues with data security and moved more functionality to a central core

Name	Specialization within Nerdalize	Experience
Mathijs	CTO	15 years of experience with software engineering 4 years with cloud computing
Thirza	Billing	2 years of billing experience
Boaz	Sales	5 year
Tim	Network	10 years of networking experience 8 of which in data centers

Table 4.1: People interviewed throughout the project

Literature Study

A literature study was done for two main reasons. Background knowledge on container infrastructure and clusters had to be acquired. Sources for this information were scientific papers and books. Second related work had to be surveyed to find ways in which problems have been solved before by existing resource measuring systems. Sources for the related work were project homepages and repositories.

Digital Media

The featured videos on Go's homepage were used to get familiar with some of Go's concepts such as subroutines and reflection. Other forms of digital media used were slide shows and online articles.

Running Test Systems

During the last week of the research phase of this project a local Kubernetes cluster was set up with Minikube[24]. This cluster was used to get a better view of how Kubernetes works and which features are already offered by Heapster. This was extremely useful as it helped us understand how Heapster [25] and InfluxDB work together (InfluxDB working as a sink for Heapster with Heapster pushing the metrics), which metrics are collected and which are missing for our system. Later in the project a Kubernetes cluster will be set up on Google cloud this cluster will be used for deployment, testing and debugging the system

4.2. Development

4.2.1. Development Strategies

In this section, different development strategies that were considered for the project are explained. For the development strategy, there are two requirements. The development strategy needs to be able to cope with requirements volatility as Nerdalize is still developing their main product and not all requirements are clear yet. Second, the development strategy needs to work well for a system that consists of many decoupled software products.

Waterfall

The waterfall model is a sequential design process, used in software development processes, in which progress is seen as flowing steadily downwards through the phases of conception, initiation, analysis, design, construction, testing, production, implementation and maintenance. Despite the development of new software development process models, the Waterfall method is still a dominant process model.

SCRUM

Scrum is both an iterative and incremental software development framework. It defines a flexible development strategy.

A key principle of SCRUM is requirements volatility. Scrum recognizes that the customers can change their minds about what they want and need, and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an evidence-based empirical approach. It accepts that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly, to respond to emerging requirements and to adapt to evolving technologies and changes in market conditions.

XP

Extreme programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity, and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers. The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels.

4.2.2. Programming Languages

In this section, some of the programming languages that were considered for the project are outlined. As a monitoring system needs to run in real time one requirement for the programing language is performance, a Second requirement is that the language has to offer tools to run concurrency as the system will need to process data of multiple clusters at the same time.

C(++)

Ever since its appearance in 1983, C(++) has been used in an incredible number of applications. Its low-level nature with corresponding memory management makes it ideal for performance-critical and real-time purposes. However, despite several revised standards, the language is starting to show its age and can be considered verbose, meaning it the amount of code needed is often high compared to implementations in other languages.

Go

Go is an open source programming language developed by Google. It offers a modern language with interesting concepts such as channels and concurrency and an innovative approach to interfaces and reflection. It can be considered a good fit for this project due to the fact that both Kubernetes and Docker are written in it. Grafana, Prometheus (partly), InfluxDB and Chronograf are also written in Go, enabling easier integration.

Python

Python is a high-level interpreted dynamic programming language. While most implementations tend to be slow compared to compiled programming languages, developing applications is often much faster, making it ideal for rapid prototyping. Being one of the most popular languages available today, there is a very large and active community, with a wealth of libraries available. Graphite and the Graphite webapp are both written in Python.

4.2.3. Repository

In order to collaborate on the code, a repository with version control is needed to ensure functions that are developed separately can be integrated with minimal effort. Multiple solutions were considered.

GitHub

GitHub is the most well-known repository service with over 14 million users. It offers a Git based distributed version control. And has several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

GitLab

GitLab is an open source repository manager, like Github it offers collaborative features but unlike GitHub, it has an integrated Build system and offers free private repositories and can be self-hosted.

4.3. Systems

There are already a lot of systems to provide some of the functionality a resources monitoring system needs. In this section, the commonly used systems in distributed computing are outlined.

4.3.1. Data Retrieval and Storage

Data retrieval and storage are considered together, as most systems offer them in an integrated fashion.

Graphite

Graphite is a monitoring tool to store numeric time-series data, offering an SQL-like language for querying the stored data. It consists of Carbon, a service for collecting time-series data, Graphite-web, offering a user interface and a possibility to render graphs and Whisper for storing data. Besides Whisper (a local file-based time-series database), data can also be stored in Ceres (a distributable time-series database) or even InfluxDB.

InfluxDB

InfluxDB is an open-source time series database developed by InfluxData and written in Go[26]. Data is grouped as measurements, which are comparable to tables in traditional relational databases. A measurement consists of a range of timestamps, each holding multiple key-value pairs. Some remarks can be made about the relative non-scalability of the system (horizontal scaling is only possible in the commercial version), something that could cause problems in the future. However, as InfluxDB is currently one of the fastest-growing databases, we expect that solutions have emerged from the community by that time.

Prometheus

Prometheus is an open-source time series database originally developed at SoundCloud, largely written in Go. It ”works well for recording any purely numeric time series. It fits both machine-centric monitoring as well as monitoring of highly dynamic service-oriented architectures”. However, ”If you need 100% accuracy, such as for per-request billing, Prometheus is not a good choice as the collected data will likely not be detailed and complete enough.”[27]. Whether this also applies in our case still requires further research.

Ganglia

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters. It leverages widely used technologies such as XML for data representation and RRDtool for data storage and visualization. It is currently in use on thousands of clusters around the world and can scale to handle clusters with 2000 nodes.[28]

Nagios

Nagios is open source software for monitoring systems, networks, and infrastructure. It saw its first release in 1999 and is still actively developed. Nagios is however not aimed at clusters[8], as such the setup process for Kubernetes would be complex. For this reason, other options are preferred.

MonALICA

MonALICA is a globally scalable framework of services to monitor and help manage and optimize the operational performance of Grids, networks and running applications in real-time. However, since it is aimed more at operational performance than resources utilization measurement, other options are preferred[29].

Relational Database Management Systems (RDBMS)

Besides time series databases, several RDBMS with SQL-support (MySQL, SQLite, and PostgreSQL) were also considered. Despite not being tailored to time series, which will form the majority of the data and therefore dictate most of our needs, they are true and tested systems that have been applied to solve largely divergent problems.

4.3.2. Data Visualization

Having a GUI (and therefore data visualization) is a could have of the system and it is still unclear in the research phase whether the project will eventually feature one. Nevertheless, it is good practice to already take into account possible techniques and frameworks in order to simplify eventual implementation, either during the project or in the future.

Grafana

Grafana is an open-source time series metric analytics and visualization suite. It offers built-in support for various time series databases, including Graphite, InfluxDB, and Prometheus. It is written in Go.

Chronograf

Chronograf is an open-source time series visualization application, developed by InfluxData (the company behind InfluxDB). It is part of the TICK stack, which is meant to manage time series data [30]. It consists of Telegraf (data collection from different sources), InfluxDB (storage of the data), Chronograf (visualization) and Kapacitor (monitoring and detecting anomalies in the data, alerting based on triggers). Due to the tight integration, it is only viable as an option if InfluxDB is chosen (using Graphite is possible, but only through the use of an InfluxDB, introducing unnecessary complexity/work).

Graphite Web App

The Graphite web app is part of the Graphite suite, dealing with data visualization. It is implemented as a Django webapp, using Cairo for vector rendering. Analogous to Chronograf, considering it is only interesting if Graphite is chosen for data retention.

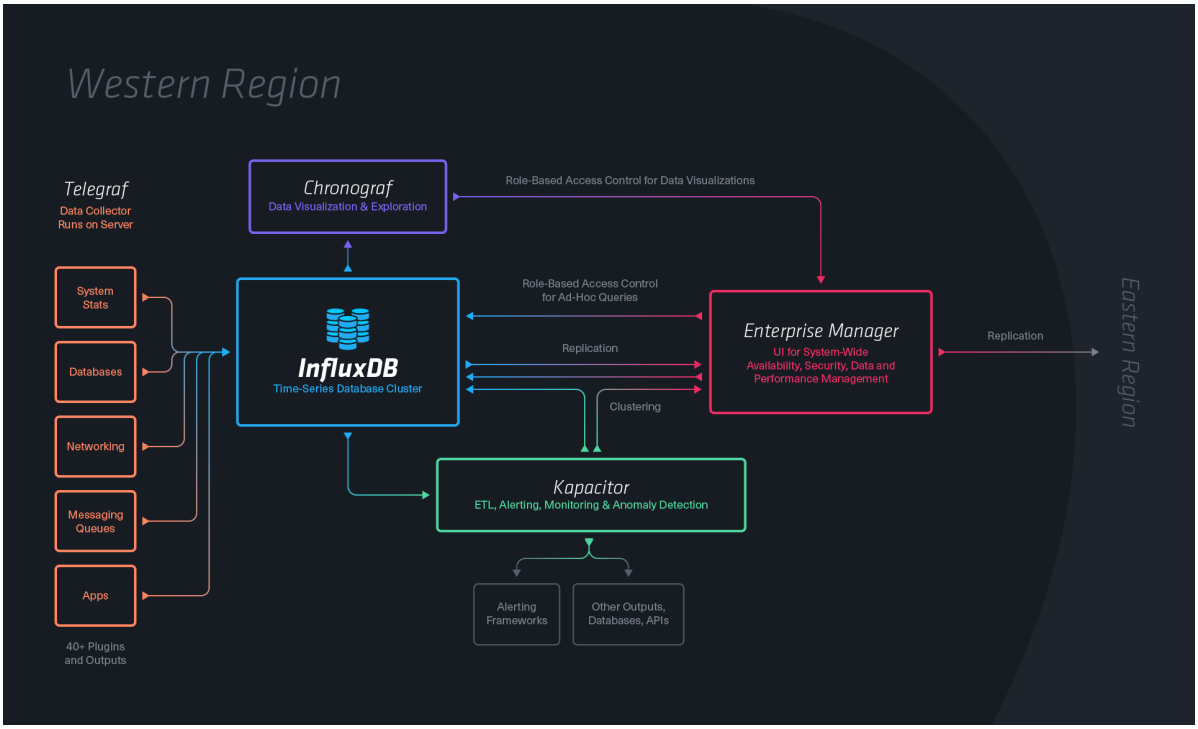


Figure 4.1: The TICK stack

4.4. Development - Chosen Development Tools

4.4.1. Development Strategy: SCRUM

For the development strategy, Waterfall was deemed inadequate for this project as some of the requirements might change during the project, due to the relatively new and flexible nature of the company and its software stack.

Furthermore, SCRUM was chosen over Extreme Programming. While both strategies have similar philosophies, both parties have more experience with SCRUM. Several concepts of Extreme Programming are still considered useful, though, such as pair programming. The sprint duration was set at 2 weeks, meaning there will be 3 sprint reviews (20 January, 3 February, and 17 February). Furthermore, there will also be a sprint meeting on every Friday in order to monitor the progress and to react timely in the event of problems. The sprint plan for the first sprint is shown in appendix A and an overview of all sprints is given in table 4.2

Table 4.2: Overview of sprints

#	Weeks	Main activity
0	1-2	Acquire background information, gather requirements, create preliminary design
1	3-4	Development
2	5-6	Development
3	7-8	Development and Validation
4	9-10	Writing final report and prepare for demo and presentation

4.4.2. Programming Language: Go

Go was chosen over both C++ and Python, due to the ease of integration with Kubernetes and Docker and its abundance within Nerdalize. Apart from this, its harmony with the rest of the stack can be considered a key factor in the choice. Go also offers good performance and great tools for concurrency, requirements set in section 4.2.2 .

4.4.3. Development Environment

Repository: GitLab

Since both GitHub and GitLab offer almost the same features our choice for GitLab was made because GitLab was the preferred tool within the Nerdalize company. This means that experienced colleagues can help us set up our repository.

Test System: Minikube

To test code on a running system Minikube was used. Minikube sets up a local Kubernetes cluster which allowed to deploy code locally making it easier to debug.

4.4.4. Code Quality: GitLab CI

To ensure each build has good code quality we will use GitLab CI. Continues integration will test the code both functionally as non-functionally. The choice for GitLab CI was made because it is an integrated part of GitLab and it has features that other CI tools don't offer, such as continuous deployment.

Testing: Unit Tests

Go offers an integrated testing package that offers unit testing, benchmarking and code coverage[31].

Formatting: GoFmt

GoFmt automatically formats Go code, this ensures that all Go code has the same format making it easier to write, read and maintain[32]. As a result of this, there is no need for automated testing code style.

Documentation: GoDoc

GoDoc automatically generates documentation on Go code based on comments in packages and function.

4.5. Development - Chosen Systems

4.5.1. Data Collection: Heapster

For data collection, we will further investigate Heapster, as it offers a good starting point. Heapster uses cAdvisor, a tool that measures the resource usage and performance of containers, and runs it within a Kubernetes cluster collecting measurements from all containers. Heapster also collects additional statistics from Kubernetes itself. It aggregates this data and makes it available via REST endpoints. Heapster, however, does not support all the necessary statistics as all network traffic is treated equal and monitoring of disk I/O also seems to be lacking.

4.5.2. Data Retrieval and Storage: InfluxDB (and SQL)

As we tend to steer away from reinventing the wheel, RDBMS will not be considered for the primary data due to the relatively convoluted set-up that would be required. Prometheus, InfluxDB and to a lesser extend Graphite seem to offer solutions to many of our problems and promise to do so right out-of-the-box, while also enabling a better representation of our data. Ganglia, and MonALISA where also looked at but they are outdated, and do not integrate with k8s out of the box.

InfluxDB was also chosen due to the experience already obtained by the employees and because integration would be easier with Nerdalize's existing systems, which also rely on InfluxDB. As at this moment it cannot be accurately predicted whether InfluxDB will suffice for Nerdalize's needs in the future, the system will be built as modular as possible, enabling a relatively easy switch to another database.

However, an RDBMS will still be used, mostly for the secondary data (E.g. client information, relating jobs to clients or storing specifications of all systems). Because most modern languages (including Go) allow communication with RDBMS's supporting SQL through a uniform interface, choosing a specific RDBMS is not necessary at this point in time, as they can be changed with very little effort.

4.5.3. GUI: To Be Determined

The GUI would ideally be integrated with the rest of Nerdalize's GUI. However, as Nerdalize recently attracted a new employee to enhance their GUI experience, the used frameworks and set-up might change in the near future. This means that, while not ideal, the exact set-up of our (possible) GUI will be determined at a later point in this project.

4.5.4. Data Visualization: Grafana or Chronograf

By choosing InfluxDB we are able to stay partly agnostic in our data visualization, giving us a fallback in case things go south. Having several options can be considered good practice, further emphasizing InfluxDB as a good choice. Apart from this, the choice also depends on the details of the GUI, which are not yet available at the time of writing.

5

Preliminary Design of Accumulus, a Monitoring and Cluster Analysis System

According to the requirements specified in chapter 3 a preliminary design was created. This chapter gives both an overview of the architecture of the system and shows how the components work together as a system.

5.1. Architecture Overview

The designed system will consist of different components, which are:

- An extended version of Heapster
- A mechanism to partially transfer SQL databases
- Accumulus
- Accumulus GUI

A complete overview of the architecture is shown in figure 5.1. This system is split in two parts, an instance of the client part will run in every client cluster, while the core part runs in a central location and is responsible for deploying the client part in each cluster. The central part connects to all user clusters to collect compute resource data, while it pushes data about the heat and power usage to the client cluster. This set-up is chosen in order to ensure that no sensitive data is stored in the client cluster.

Heapster

For this project, Heapster will be extended so additional tags can be added to metrics. An overview of extra tags is shown in table 5.1. Heapster will use an SQL Database to look up the right data, for example relating resources to the hardware they were used on, and adding the Cloudbox id. This database will be created when the cluster is started and only contain information of machines in the cluster, this is to prevent the customer from getting access to information such as the size of the company. Heapster collects metrics each interval and stores those measurements in InfluxDB.

InfluxDB

InfluxDB will be used for storing time series data. There are multiple instances of InfluxDB, one in each customer cluster and a central instance for long term storage and historic searching. The instances in the customer clusters will keep precise data. InfluxDB down-samples this precise data to a lower

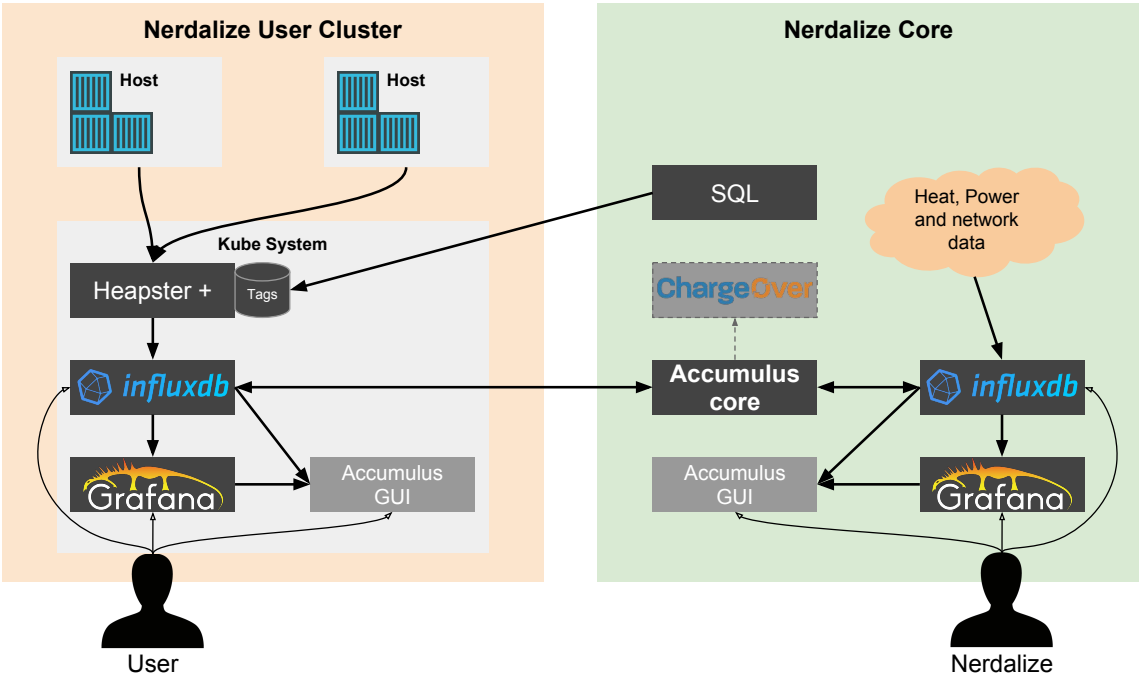


Figure 5.1: Overview of system architecture

Tag	Metrics	Reason
Cloudbox id	All metrics	Needed to estimate power use and heat production
Cluster id	All metrics	Needed to link metrics back to a customer
Cpu type	Pod and node	Gives the user insight into his performance on the type of cpu And gives Nerdalize data on cpu usage
Ram type	Pod and node	Gives Nerdalize data on ram usage
Disk type	Pod and node	Gives the user insight into his performance on the type of disk And gives Nerdalize data on ram usage

Table 5.1: Tags on metrics and reason for these tags

resolution, which will be collected by the central Accumulus instance. InfluxDB can also use retention policies to reduce disk usage by only keeping data for a certain time.

Grafana

Grafana is a dashboard application for time series, it visualises time series data. Heapster comes with its own version of Grafana, which has preconfigured dashboards for both cluster and pod performance. Grafana will be used for visual representation of the data, both the customer and Nerdalize will have access to this interface and the possible GUI that may be developed will most likely also use Grafana to render graphs.

Accumulus

An instance of Accumulus will run in the Nerdalize core. Accumulus is responsible for the connection between the customer cluster and the Nerdalize core cluster and fulfills three main functions, deployment, collection, and processing. Accumulus' deployment part will deploy the Accumulus components on new or existing customer clusters. The collection part will collect down-sampled data from all client clusters for billing and analytic purposes. Finally, the processing part collects the power usage and heat production data from each Cloudbox and splits them over the compute modules in a Cloudbox.

Accumulus GUI

Accumulus GUI is a graphical user interface aimed at the end user, in this interface the user can see his total resource usage and get an estimation of the cost and the ammount of CO^2 emission saved.

SQL

The SQL database running in the core cluster contains information on all clusters, Cloudboxes, jobs, and customers. As this data can be considered sensitive (the locations and total number of Cloudboxes, customer information etc.), it is distributed on a need-to-know basis to the customer clusters. This means that a cluster only has (partial) information on the Cloudboxes running inside the cluster.

6

Validation

The developed system is useful only if it satisfies two conditions. The system needs to be accurate, which will be discussed in chapter 6.1, and the system needs to have a small overhead, which will be experimentally verified by the experiments proposed in 6.3.

6.1. Accuracy

A measurement system is not useful if the measurements are inaccurate. Therefor the system needs te be tested for accuracy on all metrics it collects.

6.1.1. Compute Resources

Heapster receives its measurements directly from cAdvisor and as such, Accumulus' accuracy for resource usage depends entirely on that of cAdvisor. cAdvisor collects its measurements directly from the kernel and OS, because of this these measurements can be assumed accurate enough by doing a basic validation with a known impact on the system.

6.1.2. Power and Heat

The Cloudbox contains a class B electrical energy meter, which is legally required to have a measurement error of less than 2 per cent over the temperature range where Nerdalize is operating in[33].

All of the power consumed by the compute modules is transferred to the water, with the exception of the heat lost to the surroundings, meaning an upper bound can be given for the error with which the created (useful) heat is estimated. No useful heat is created if the water in the boiler already has the maximum temperature, something which is easily measured.

The actual heat used by the home owner is more complicated and can only be inferred by the measurements from two temperature sensors (10K NTC with a B constant of 3435 K), one in front of the heating area and one after. Whether this is enough for a precise estimate is still being discussed with Nerdalize.

6.2. Basic Validation

To validate the basic working of the system, Nerdalize has requested some smoke tests. In those experiment stress tests will be used to cause a predictable load on the system. Accumulus should then show the expected values for the load. Table 6.1 shows the resources that are tested and the expected loads that Accumulus should measure.

Resource	Test program	Expected result
CPU	Single core stress test	1000 millicores used
CPU	Dual core stress test	2000 millicores used
RAM	Generate 512Mb array	512Mb ram used
Network	Request 1Mb file from internet	1Mb network rx

Table 6.1: Basic Accuracy Validation.

6.3. Overhead

As stated, one of the non-functional requirements was an overhead of less than 10%. In section 6.3.1 and 6.3.2, two experiments for validating this are proposed. In section 6.3.3, an experiment for relating the checking interval to the total overhead is proposed.

6.3.1. Overhead of Accumulus in a Timed Benchmark

The goal of the experiment is to get insight in the overhead caused by Accumulus, compared to a bare Kubernetes cluster and a Kubernetes cluster running an unmodified version of Heapster. This is done by running one or several benchmarks, after which the total time required to finish the benchmark is noted. This is repeated a number of times for several cluster sizes, in order to get insight in the effect of the cluster size and average the influence of non-deterministic effects. A full table of tests is shown below

Benchmark	Cluster Size	Configuration	Expected result
Benchmark	Single node	Bare Kubernetes	baseline
Benchmark	Single node	Heapster	increased from baseline
Benchmark	Single node	Accumulus	increased from Heapster
Benchmark	5 nodes	Bare Kubernetes	baseline
Benchmark	5 nodes	Heapster	increased from single node Heapster
Benchmark	5 nodes	Accumulus	increased from 5 nodes Heapster
Benchmark	25 nodes	Bare Kubernetes	baseline
Benchmark	25 nodes	Heapster	increased from 5 node Heapster
Benchmark	25 nodes	Accumulus	increased from 25 node Heapster

Table 6.2: Overhead of Accumulus in a Timed Benchmark.

6.3.2. Overhead of a Monitored Accumulus Instance

The goal of the experiment is also to get insight in the overhead caused by Accumulus. This time, a cluster will perform a workload, and be monitored by an unmodified version of Heapster, and Accumulus. Both instances will monitor each other, and we will verify that they measure the same utilisation, then the overhead of Accumulus can be seen by comparing measurements between Heapster and Accumulus.

6.3.3. Difference in Overhead Based on Interval Length

In this experiment, the influence of the measuring interval on the overhead will be determined. For this, a benchmark is run in a Kubernetes cluster with Accumulus deployed, noting the time it takes to finish the benchmark. The benchmark is then run again with different values for the measuring interval, in order to see its effect. Again, each measuring interval is run several times to minimize the effect of non-deterministic behavior. A full overview of the diferent intervals that will be tested is shown in table 6.3

6.4. Scalability

Benchmark	Cluster Size	measurement interval	Expected result
Benchmark	5 nodes	every minute	baseline
Benchmark	5 nodes	every 30 seconds	2x baseline
Benchmark	5 nodes	every 15 seconds	4x baseline
Benchmark	5 nodes	every 5 seconds	12x baseline
Benchmark	5 nodes	every second	60x baseline

Table 6.3: Difference in Overhead Based on Interval Length.

A

Appendix A: Sprint Plans

A.1. Sprint 1

Sprint 1

161/160 hours | 140/160 hours | 11/13 completed

✓	User Story	Task	Assigned To	Estimated Effort	Priority	Actual Effort	Notes
x	As a developer i want a code repository so i can easily collaborate and have version control	Set up git repository	Diony	4	A	4	Two gitlab repositories have been set up
x	As a developer i want to do integration testing to ensure all software components integrate well	Set up continuous integration	Youp	6	A	24	Setting up continuous integration took longer than expected due to the complicated environement we need for testing
x	As a developer i want a kubernetes cluster to test and run my system on	Set up a kubernetes cluster	Both	2	A	6	Setting up a kubernetes cluster on google cloud took longer as we had to await approval from the company, clusters cost money
x	As a developer i want to test my code	Set-up unit testing framework and create a unit test	Youp	8	A	10	We had no experiance with gitlab's integrated CI, because of this setting up unit testing took longer
x	As a developer i want a test workload with a predictable load to validate the accuracy of my system	Create a test job for personal testing	Diony	4	B	4	test job is writen in go and has a predictable resource load.
x	As a developer i want to be able to quickly depoy my code	Set up deployment file for Kubernetes for automatic testing	Youp	14	B	16	
x	As a developer I want to have a clear model of what I'm going to create	Create model and document it	Both	16	A	20	a small issue with authentication took aditional time A lot of important decisions where made at nerdalize during this sprint, as such our architecture design changed multiple times
x	As a developer I want to be able to correlate data (e.g. specifications and client data)	Set up SQL database and populate it	Youp	8	B	2	Done, nerdalize already has a system that contains this information
x	As a user i want to see the resource use of jobs running on kubernetes	Monitor CPU, RAM, disk and network use	Diony	25	B	20	All metrics are already available in heapster with the exception of Disk IO, we found a solution for this but have problems implemented it.
x	As a user i want to be able to select metrics based on tags such as machine id and the job they belong to	Split resource usage correctly using tags	Diony	30	A	12	Most of the tags are in place. Some tags rely on our architecture design and have not been implemented due to the changes in this design
x	As a developer i want to know which network statistics are available	Investigate possibilities for network usage monitoring	Diony	12	C	8	In a meeting with nerdalize's network engineer we discovered that not all network statistics we want are available, a new set of metrics was selected.
	As a user i want to know how much power my cluster consumed and how much heat this provided	Store power and heat consumption	Both	16	B	4	A change in architecture design changed the way we want to do this, we did however research how this can be done and now have acces to the right database.
	As the billing department, I want my billing software to be able to receive usage data	Design API for querying data	Diony	16	B	10	Changes in our architecture design caused the API part of our project to be deprioritised as both influx and grafana have a good API and will contain all neccesary data in our new design.
	As a student I want to deliver a well writen research report	Write research report	Both	-	B	15	Due to late feedback on our first draft we still had to work on the research report

Main Problems Encountered

1	**Changes in architecture design** A lot of important design decisions where made by nerdalize during this sprint, those decisions had a lot of influence on how the environment our system needs to run in looks, as such we had to make changes to our architecture design.	25 We redesigned our architecture design twice and discused it with multiple members of nerdalize
2	**Inexperiance with Gitlab CI** Inexperiance with the Gitlab CI resulted in a long setup process for our CI, in the end we had to do a step back and do less thorough integration testing as setting up a cluster every time the code has to be tested was not deemed viable.	-

Figure A.1: Sprint 1

A.2. Sprint 2

✓	User Story	Task	Assigned To	Estimated Effort	Priority	Actual Effort	Notes
		Sprint 2		116/160 hours		96/160 hours	12/12 completed
x	As a developer i want to have a clear vision of what i need to do	Sprint review and planning	Both	2	A	2	
x	As Accumulus i want to deploy the necessary applpications on every new cluster	Make Cluster startup function	Youp	14	A		
x		Setup central SQL server	Youp	6	A		
x	As Heapster i want to know which nodes belong to which cloudbox and what hardware that cloudbox has	Make SQL Sync logic	Youp	10	B		
x	As Accumulus i want to be able to pull and push data between databases	Make pushing and pulling of influxdb data possible	Diony	8	A	12	Took longer than expected: The available API did not cantain all neccesary fucntions.
x	As Accumulus i want data to be downsampled before it is stored in the core	Add continuous query to downsample data	Diony	4	4	4	We found a way to do everything with a single downsample query
x	As Accumulus i want to have the heat and power metrics of cloudboxes available	Pull heat and power metrics in central Influx	Diony	2	B	4	Took longer, as for our test cluster this data is not available and thus has to be simulated
x	As a developer i want to know how cost and power usage are distributed over the components of a cloudbox	Research the cost and power usage of a cloudbox	Diony	4	B	2	
x	As a user i want to see how much power my jobs have cost	Create a power processor	Diony	8	B	12	Writing a function to structure the data in a good way took some time as our influx functions were not flexible enough yet
x	As Accumulus i want to have all metrics tagged with the cloudbox and hardware they are measured on	Tag all metrics with the cloudbox and specs	Diony	10	A	8	Done with the exception of the sql connection
x	As Accumulus i want to store metrics for billing and historical purposes on a central DB	Sync Influx databases	Diony	10	A	16	Took more time due to a lot of refactors to make the functions more versatile
x	As a Developer i want to show my system to the client so i can collect feedback	Prepare demo	Both	10	C	8	Demo was recieved really well, good feedback was given
x	As a student i want to deliver a well writen thesis report	Work on the thesis report	Both	30	B	30	

Figure A.2: Sprint 2

B

Project Proposal

Nerdalize Super Charge

Function-as-a-Service Cloud Resource Consumption Measurement System for a Distributed Cluster

a TU Delft Computer Science Bachelor Project Proposal

TL;DR

Nerdalize is a cost and energy efficient cloud provider that heats homes with the waste heat of its servers. It wants to sell cloud in a Function-as-a-Service model and needs a system that can measure actual resource consumption on a distributed container cluster.

What does Nerdalize do?

Nerdalize heats homes by placing servers into fiber connected homes. Nerdalize sells the server capacity as cloud services at significant lower cost than anyone else. The home owner can save ~€200 euro/year on his energy bill and the world saves ~3.5 tons of CO2 per household per year. Here is a man with a fantastic british accent explaining the basics.

Nerdalize earns its moneys by being a cloud provider and is currently focusing on the High Throughput Compute market for SMB's, scientists and engineers. We can already beat the major cloud providers like Google, Amazon and Azure at their own game on a small scale. Our goal is deploying the Nerdalize system over 10^5 homes in the next 3 years and turn our continent into a distributed, energy efficient datacenter.

To make this possible we have some really smart people working on building the best tech we can imagine. Some of the things we've built so far include:

- **Computing-as-a-Service** a new user friendly way to run High Throughput Computing workloads for scientists and engineers. The backend is writting using goa and the frontend uses electron.
- **Cloud Benchmarker** the world's first (and so far only) system that benchmarks actual costs for actual (docker packaged) workloads across all major cloud providers. Enabling us to predict project costs and turnaround time. It was written mostly in Python, interfacing with influxdb and uses terraform.
- **KaaS (Kubernetes-as-a-Service)** a serverless AWS lambda based system for spinning up Kubernetes clusters across different cloud providers. It uses DynamoDB and a lot of Go code.

- **gitbits** a git based, content addressable, deduplicated, encrypted and efficient large file transfer method that punches through corporate firewalls. Written in Go.
- **Nerdalize Compute Module** an extremely energy efficient server cooling system based on passive full immersion 2-phase cooling which makes us the most energy efficient cloud provider in existence.
- **Nerdalize CloudHeater and CloudBox** server heating systems that heat your room, your house or your domestic hot water. Including our custom designed measurement and control systems PCB's and monitoring system using influxdb and grafana
- **Nerdalight** our own 10Gbps Fiber-to-the-Home network infrastructure that connects our heating systems into a high performance cluster

Problem Description

Currently most cloud providers charge their customers in a *pay-per-claimtime* cost model. The customer claims a particular instance type (e.g. a GCE n1-standard-16 instance) and pays for the time this machine was reserved for use the customer.

The instance type usually defines a particular Virtual Machine (VM) configuration. For example the n1-standard-16 GCE instance type is a VM with 16 vCPU's and 60GB of RAM Memory. A network attached persistent disk can be allocated to such an instance. Network throughput to and from this machine is capped and usually there is a charge per GB egress.

Thus an instance type can be viewed as a collections of Big4 (Processing, Memory, Storage and Network) resources that are reserved for use by a customer.

When the customer orders his VM the cloud provider locates a physical machine that can supply these resources and boots the Virtual Machine on it, which claims the resources for use by the customer.

If the customer underutilizes the resources his VM provides these resources cannot be easily distributed to other VMs:

- the GBs of RAM a customer's VM is not using cannot be allocated to another VM
- the VM is allocated vCPU cores (hyperthreads) that cannot be used by other VM's or processes
- the VM's disk space is reserved and cannot be shared with other VM's or customers

This leads to less than optimal resource utilization on the infrastructure of the cloud provider. Nonetheless these resources do have capital and operation expenses and these costs must be recuperated or the Cloud Provider will soon find itself out of business. This means customers pay for unused resources, the

cloud provider has a lower return on investment on its infrastructure and a whole lot of energy is wasted keeping all those underutilized machines running.

We should do this much, much better! Luckily we have a plan!

Nerdalize runs Kubernetes clusters on its distributed server fleet. Our customers execute computationally intensive bag-of-task type *workloads* consisting of many *tasks* on these clusters. Each task is a Docker container with certain resource requests and resource limits that is placed by the Kubernetes scheduler on a physical machine. A container behaves very much like an OS-process in the sense that it can share CPU cores, RAM Memory, persistent storage and network resources dynamically on a very granular level.

Nerdalize is building a Function-as-a-Service computational system and needs a system that can accurately measure resource consumption.

Challenges in this project *will* include:

- eliciting the information requirements of cluster users, Nerdalize Operators, Billing and Sales people.
- understanding modern process container resource models
- understanding modern container orchestration schedulers (like kubernetes)
- defining the to-measure resources, their units and measurement methods
- validating that measurement methods are accurate and reliable
- implementing a system that measures resource consumption and deploying it on a live cluster

Challenges this project *can* include:

- learning a whole swath of modern cloud technologies including: Go, InfluxDB, grafana, Kubernetes, Docker, Terraform, Ansible, etcd, YAML
- working with a group of highly motivated, experienced people who's skillsets range from discounted cashflow modelling through thermodynamic simulations to
- open sourcing part or whole of the implementation
- publishing a blog post about the project through Google's
- beating your supervisor on the foosball table

How can Nerdalize help you?

- we pay you a monthly internship stipend
- we have an ad libitum stroopwafel policy
- psychological warfare at the foosball
- watch a growing startup from the inside
- flexible working times

What we expect from you

- a commitment to do something awesome at very high quality. We'll push you to go the extra mile.
- English proficiency (spoken and written)
- come work in our office, it's fun!

C

Requirements

SuperCharge Functional Requirements

discussion draft v 20161221

Actors

Nerdalize Compute Billing
Financial dept. that needs billing information to send invoices to customers.

Nerdalize Compute Sales
Sales dept. that charges customers for compute resources with a particular pricing model.

Nerdalize Compute Engine API
User facing API that shows resource utilization and cost to user:

User Stories

The **Nerdalize Compute Engine API** wants to:

- read accurate historical [resource/cost] consumption per [cluster/per user/per project/per workload/per job] to inform the user of his pas resource consumption
- read accurate current [resource/cost] consumption **rate** per [cluster/per user/per project/per workload/per job] to inform the user of his burn rate
- predict accurate future [resource/cost] consumption **rate** per [cluster/per user/per project/per workload/per job] to allow the user to predict his resource consumption, project turnaround time and total project cost.

The **Nerdalize Compute Billing Dept.** wants to:

- have sufficient information for billing purposes

The **Nerdalize Compute Sales Dept.** wants to:

- have flexibility for different billing models
- have the data support their current billing model

Bibliography

[1] E. A. Brewer, *Lessons from giant-scale services,* IEEE Internet Computing **5**, 46 (2001).

[2] A. Vaughan, *How viral cat videos are warming the planet,* (2016).

[3] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, *The cost of a cloud: research problems in data center networks,* ACM SIGCOMM computer communication review **39**, 68 (2008).

[4] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier, *A federated multi-cloud paas infrastructure,* in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on* (IEEE, 2012) pp. 392–399.

[5] M. Cilissen, M.H.J. Van Elsas, *Design of a performance benchmarker for fully distributed iaas clouds,* (2015).

[6] M. Al-Roomi, S. Al-Ebrahim, S. Buqrais, and I. Ahmad, *Cloud computing pricing models: a survey,* International Journal of Grid & Distributed Computing **6**, 93 (2013).

[7] M. L. Massie, B. N. Chun, and D. E. Culler, *The ganglia distributed monitoring system: design, implementation, and experience,* Parallel Computing **30**, 817 (2004).

[8] W. Barth, *Nagios: System and network monitoring* (No Starch Press, 2008).

[9] X. Zhang, J. L. Freschl, and J. M. Schopf, *A performance study of monitoring and information services for distributed systems,* in *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on* (IEEE, 2003) pp. 270–281.

[10] M. Mansouri-Samani and M. Sloman, *Monitoring distributed systems,* IEEE network **7**, 20 (1993).

[11] Nerdalize, *Example benchmark report,* (2016).

[12] S. U. Mendel Rosenblum and VMWare, *The reincarnation of virtual machines,* (2004).

[13] A. Hassain, *Performance of docker vs vm's,* (2014).

[14] D. Inc., *What is docker?* (2016).

[15] Q. F. Hassan, *Demystifying cloud computing,* (2011).

[16] Interoute, *What is iaas?* (2017).

[17] T. Bronchain, *How hypernetes brings multi-tenancy to microservice architectures,* (2015).

[18] The Kubernetes Authors, *Kubernetes: Production-grade container orchestration,* (2016).

[19] R. Carosi and B. Mattijssen, *Flower,* (2016).

[20] The Kubernetes Authors, *Reference documentation: Services,* (2016).

[21] The Kubernetes Authors, *Managing compute resources,* (2017).

[22] The Kubernetes Authors, *Volumes,* (2017).

[23] J. Kincaid, *Google's go: A new programming language that's python meets c++,* (2009).

[24] Minikube, *Minikube,* (2016).

[25] Heapster, *Heapster,* (2017).

[26] InfluxData, *Key concepts,* (2016).

[27] Prometheus, *Prometheus - overview,* (2016).

[28] Ganglia, *Ganglia monitoring system,* (2005).

[29] H. B. Newman, I. C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu, *Monalisa: A distributed monitoring service architecture,* arXiv preprint cs/0306096 (2003).

[30] InfluxData, *Introducing the tick stack,* (2016).

[31] The Go Programming Language, *Package testing,* (2012).

[32] The Go Programming Language, *Go fmt your code,* (2013).

[33] The European Union, *Directive 2004/22/ec of the european parliament and of the council of 31 march 2004 on measuring instruments,* (2004).

C

Project Proposal

Nerdalize Super Charge

Function-as-a-Service Cloud Resource Consumption Measurement System for a Distributed Cluster

a TU Delft Computer Science Bachelor Project Proposal

TL;DR

Nerdalize is a cost and energy efficient cloud provider that heats homes with the waste heat of its servers. It wants to sell cloud in a Function-as-a-Service model and needs a system that can measure actual resource consumption on a distributed container cluster.

What does Nerdalize do?

Nerdalize heats homes by placing servers into fiber connected homes. Nerdalize sells the server capacity as cloud services at significant lower cost than anyone else. The home owner can save ~€200 euro/year on his energy bill and the world saves ~3.5 tons of CO2 per household per year. Here is a man with a fantastic british accent explaining the basics.

Nerdalize earns its moneys by being a cloud provider and is currently focusing on the High Throughput Compute market for SMB's, scientists and engineers. We can already beat the major cloud providers like Google, Amazon and Azure at their own game on a small scale. Our goal is deploying the Nerdalize system over 10^5 homes in the next 3 years and turn our continent into a distributed, energy efficient datacenter.

To make this possible we have some really smart people working on building the best tech we can imagine. Some of the things we've built so far include:

- **Computing-as-a-Service** a new user friendly way to run High Throughput Computing workloads for scientists and engineers. The backend is writting using goa and the frontend uses electron.
- **Cloud Benchmarker** the world's first (and so far only) system that benchmarks actual costs for actual (docker packaged) workloads across all major cloud providers. Enabling us to predict project costs and turnaround time. It was written mostly in Python, interfacing with influxdb and uses terraform.
- **KaaS (Kubernetes-as-a-Service)** a serverless AWS lambda based system for spinning up Kubernetes clusters across different cloud providers. It uses DynamoDB and a lot of Go code.

- **gitbits** a git based, content addressable, deduplicated, encrypted and efficient large file transfer method that punches through corporate firewalls. Written in Go.
- **Nerdalize Compute Module** an extremely energy efficient server cooling system based on passive full immersion 2-phase cooling which makes us the most energy efficient cloud provider in existence.
- **Nerdalize CloudHeater and CloudBox** server heating systems that heat your room, your house or your domestic hot water. Including our custom designed measurement and control systems PCB's and monitoring system using influxdb and grafana
- **Nerdalight** our own 10Gbps Fiber-to-the-Home network infrastructure that connects our heating systems into a high performance cluster

Problem Description

Currently most cloud providers charge their customers in a *pay-per-claimtime* cost model. The customer claims a particular instance type (e.g. a GCE n1-standard-16 instance) and pays for the time this machine was reserved for use the customer.

The instance type usually defines a particular Virtual Machine (VM) configuration. For example the n1-standard-16 GCE instance type is a VM with 16 vCPU's and 60GB of RAM Memory. A network attached persistent disk can be allocated to such an instance. Network throughput to and from this machine is capped and usually there is a charge per GB egress.

Thus an instance type can be viewed as a collections of Big4 (Processing, Memory, Storage and Network) resources that are reserved for use by a customer.

When the customer orders his VM the cloud provider locates a physical machine that can supply these resources and boots the Virtual Machine on it, which claims the resources for use by the customer.

If the customer underutilizes the resources his VM provides these resources cannot be easily distributed to other VMs:

- the GBs of RAM a customer's VM is not using cannot be allocated to another VM
- the VM is allocated vCPU cores (hyperthreads) that cannot be used by other VM's or processes
- the VM's disk space is reserved and cannot be shared with other VM's or customers

This leads to less than optimal resource utilization on the infrastructure of the cloud provider. Nonetheless these resources do have capital and operation expenses and these costs must be recuperated or the Cloud Provider will soon find itself out of business. This means customers pay for unused resources, the

cloud provider has a lower return on investment on its infrastructure and a whole lot of energy is wasted keeping all those underutilized machines running.

We should do this much, much better! Luckily we have a plan!

Nerdalize runs Kubernetes clusters on its distributed server fleet. Our customers execute computationally intensive bag-of-task type *workloads* consisting of many *tasks* on these clusters. Each task is a Docker container with certain resource requests and resource limits that is placed by the Kubernetes scheduler on a physical machine. A container behaves very much like an OS-process in the sense that it can share CPU cores, RAM Memory, persistent storage and network resources dynamically on a very granular level.

Nerdalize is building a Function-as-a-Service computational system and needs a system that can accurately measure resource consumption.

Challenges in this project *will* include:

- eliciting the information requirements of cluster users, Nerdalize Operators, Billing and Sales people.
- understanding modern process container resource models
- understanding modern container orchestration schedulers (like kubernetes)
- defining the to-measure resources, their units and measurement methods
- validating that measurement methods are accurate and reliable
- implementing a system that measures resource consumption and deploying it on a live cluster

Challenges this project *can* include:

- learning a whole swath of modern cloud technologies including: Go, InfluxDB, grafana, Kubernetes, Docker, Terraform, Ansible, etcd, YAML
- working with a group of highly motivated, experienced people who's skillsets range from discounted cashflow modelling through thermodynamic simulations to
- open sourcing part or whole of the implementation
- publishing a blog post about the project through Google's
- beating your supervisor on the foosball table

How can Nerdalize help you?

- we pay you a monthly internship stipend
- we have an ad libitum stroopwafel policy
- psychological warfare at the foosball
- watch a growing startup from the inside
- flexible working times

What we expect from you

- a commitment to do something awesome at very high quality. We'll push you to go the extra mile.
- English proficiency (spoken and written)
- come work in our office, it's fun!

D

Requirements

SuperCharge Functional Requirements

discussion draft v 20161221

Actors

Nerdalize Compute Billing
Financial dept. that needs billing information to send invoices to customers.

Nerdalize Compute Sales
Sales dept. that charges customers for compute resources with a particular pricing model.

Nerdalize Compute Engine API
User facing API that shows resource utilization and cost to user:

User Stories

The **Nerdalize Compute Engine API** wants to:

- read accurate historical [resource/cost] consumption per [cluster/per user/per project/per workload/per job] to inform the user of his pas resource consumption
- read accurate current [resource/cost] consumption **rate** per [cluster/per user/per project/per workload/per job] to inform the user of his burn rate
- predict accurate future [resource/cost] consumption **rate** per [cluster/per user/per project/per workload/per job] to allow the user to predict his resource consumption, project turnaround time and total project cost.

The **Nerdalize Compute Billing Dept.** wants to:

- have sufficient information for billing purposes

The **Nerdalize Compute Sales Dept.** wants to:

- have flexibility for different billing models
- have the data support their current billing model

Bibliography

[1] E. A. Brewer, *Lessons from giant-scale services,* IEEE Internet Computing **5**, 46 (2001).

[2] A. Vaughan, *How viral cat videos are warming the planet,* (2016).

[3] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, *The cost of a cloud: research problems in data center networks,* ACM SIGCOMM computer communication review **39**, 68 (2008).

[4] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier, *A federated multi-cloud paas infrastructure,* in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on* (IEEE, 2012) pp. 392–399.

[5] M. Cilissen, M.H.J. Van Elsas, *Design of a performance benchmarker for fully distributed iaas clouds,* (2015).

[6] M. Al-Roomi, S. Al-Ebrahim, S. Buqrais, and I. Ahmad, *Cloud computing pricing models: a survey,* International Journal of Grid & Distributed Computing **6**, 93 (2013).

[7] M. L. Massie, B. N. Chun, and D. E. Culler, *The ganglia distributed monitoring system: design, implementation, and experience,* Parallel Computing **30**, 817 (2004).

[8] W. Barth, *Nagios: System and network monitoring* (No Starch Press, 2008).

[9] X. Zhang, J. L. Freschl, and J. M. Schopf, *A performance study of monitoring and information services for distributed systems,* in *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on* (IEEE, 2003) pp. 270–281.

[10] M. Mansouri-Samani and M. Sloman, *Monitoring distributed systems,* IEEE network **7**, 20 (1993).

[11] Nerdalize, *Example benchmark report,* (2016).

[12] S. U. Mendel Rosenblum and VMWare, *The reincarnation of virtual machines,* (2004).

[13] A. Hassain, *Performance of docker vs vm's,* (2014).

[14] D. Inc., *What is docker?* (2016).

[15] Q. F. Hassan, *Demystifying cloud computing,* (2011).

[16] Interoute, *What is iaas?* (2017).

[17] T. Bronchain, *How hypernetes brings multi-tenancy to microservice architectures,* (2015).

[18] The Kubernetes Authors, *Kubernetes: Production-grade container orchestration,* (2016).

[19] R. Carosi and B. Mattijssen, *Flower,* (2016).

[20] The Kubernetes Authors, *Reference documentation: Services,* (2016).

[21] The Kubernetes Authors, *Managing compute resources,* (2017).

[22] The Kubernetes Authors, *Volumes,* (2017).

[23] J. Kincaid, *Google's go: A new programming language that's python meets c++,* (2009).

[24] Heapster, *Heapster,* (2017).

[25] Minikube, *Minikube,* (2016).

[26] InfluxData, *Key concepts,* (2016).

[27] Prometheus, *Prometheus - overview,* (2016).

[28] Ganglia, *Ganglia monitoring system,* (2005).

[29] H. B. Newman, I. C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu, *Monalisa: A distributed monitoring service architecture,* arXiv preprint cs/0306096 (2003).

[30] InfluxData, *Introducing the tick stack,* (2016).

[31] The Go Programming Language, *Package testing,* (2012).

[32] The Go Programming Language, *Go fmt your code,* (2013).

[33] Kubernetes, *Helm,* (2017).

[34] *Sig,* (2017).

[35] X. Fan, W.-D. Weber, and L. A. Barroso, *Power provisioning for a warehouse-sized computer,* in *ACM SIGARCH Computer Architecture News*, Vol. 35 (ACM, 2007) pp. 13–23.

[36] The European Union, *Directive 2004/22/ec of the european parliament and of the council of 31 march 2004 on measuring instruments,* (2004).

[37] A. Waterland, *stress,* (2014).