Flower

Workflow management and heat-aware scheduling
for modern cloud infrastructures

TU Delft

Robert Carosi
Boris Mattijssen

Flower

Workflow management and heat-aware scheduling for modern cloud infrastructures

by

Robert Carosi
Boris Mattijssen

inpartial fulfilment 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 July 1, 2016 at 16:30 PM.

Student numbers:	4242130 - Carosi & 4233190 - Mattijssen	
Project duration:	April 18, 2016 – July 1, 2016	
Supervisors:	Dr. ir. A. Iosup,	TU Delft
	M. de Meijer,	Nerdalize

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

TU Delft

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 memorable programming workshops and omnipresent enthusiasm. 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.

Contents

List of Figures

List of Tables

1 Introduction

1.1. Context

Cloud computing has given users access to virtually infinite computing power. Cloud service providers operate data centers consisting of hundreds, sometimes thousands of interconnected machines. The machines dissipate heat that is undesirable because it may lead components to overheat. Air conditioners are used to cool the air and regulate the temperature in the data center. Doing so requires large amounts of electricity leading to a high carbon footprint. 98% of Dutch houses is heated by burning gas in a central heating system. Nerdalize is a company that aims to create a cloud by placing servers in CloudBoxes in people's homes. A CloudBox contains servers and a boiler. The heat dissipated by the servers is used to warm up the water in the boiler and complement the central heating system. This approach takes away the need for expensive cooling and other infrastructure overhead compared to a traditional data center. This results in a significantly lower footprint as illustrated in Figure 1.1.

Nerdalize aims to provide a platform to members of the scientific community and research institutions for running compute intensive *workflows*. A workflow is a sequence of steps that need to complete in a certain order. Each step runs on a machine performing a particular task, such as downloading data or performing a computation.

1.2. Problem Statement

The challenge set out by Nerdalize is to develop a *workflow management system* to run on their infrastructure while making optimal use of generated heat. Many workflow management systems exist but none support a *container native environment*. The boilers attached to the CloudBox have a *limited capacity* and a *maximum temperature*. Heat generated by the CloudBox when the boiler has reached maximum temperature is wasted. To minimize this waste it is useful to take boiler temperatures into account when making scheduling decisions.

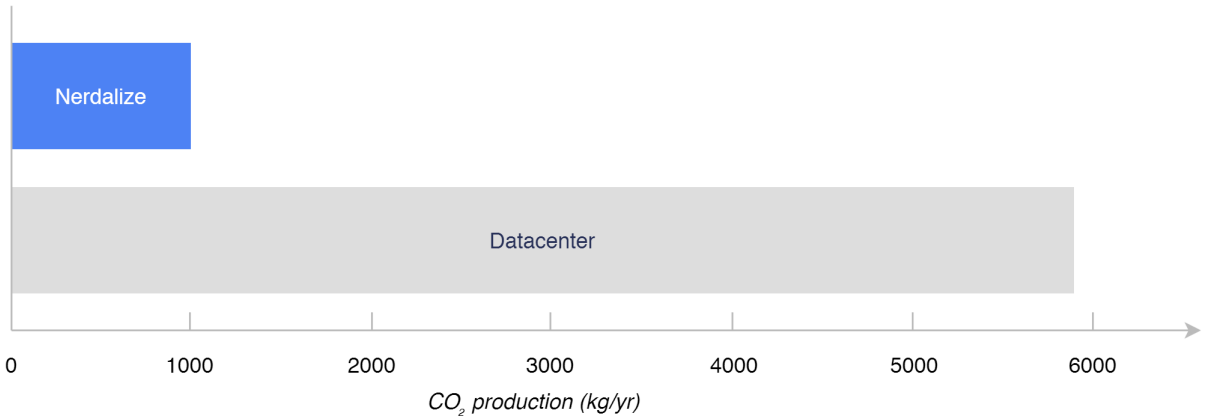


Figure 1.1: The carbon footprint per four servers, or one CloudBox, of the Nerdalize cloud (top) and a traditional data center (bottom).

1.3. Approach

The goal of this project is to manage workflows in this special context, of cloud servers deployed in private homes. Using the infrastructure deployed by Nerdalize as a use case, this project provides the conceptual contributions needed to manage workflows and the technical contributions of a realistic evaluation of the proposed workflow manager. The development of Flower started with two weeks of research. During this time we studied the strengths and weaknesses of several existing implementations. In cooperation with Nerdalize and other professionals from all over the world we specified system requirements. The goal was to develop a general solution to the problem such that it may be used by Nerdalize but also by others.

The 7 weeks after the research phase were used to design and implement Flower. We used modern software development methodologies to continuously deliver an incrementally improved product throughout the process. This allowed us to cope with changing requirements and other unexpected circumstances. The described approach led to a successful project in which we have made have made several contributions, scientific and otherwise. The main contributions are listed next in Section 1.4.

1.4. Main Contributions

In this thesis we present Flower, a workflow management system that has as main feature the first capabilities for *heat-aware scheduling*. Flower consists of several components. The first component is the workflow management system responsible for executing a workflow on a cluster of machines. The second part contains various temperature-based scheduling policies. To the best of our knowledge we are the first ones to introduce the notion of heat when making scheduling decisions. The final Flower component is a workflow visualizer to monitor workflow execution. We shared parts of knowledge gathered during the project with others through a technical blog post.

1. A workflow management system in a containerized environment. It is able to schedule arbitrary workflows defined by the user. Dependencies between steps can be specified enforcing one step to be executed before another.

2. A visualizer to monitor the current state of a workflow. The visualizer graphically displays the process of a workflow in real-time. This gives the user valuable insights when used as a monitoring tool. It also helps to detect problems which would otherwise be hard to find.

3. A survey of different scheduling policies to best make use of generated heat. Different policies are proposed and explained. An intuition is given as to what the expected outcome is and in which scenario it is useful. A policy is chosen depending on the user's objective.

4. An evaluation of one scheduling policy. Experimental work validates the Flower system and compares using one of the developed policies to a naive. The impact is illustrated through a calculation and a graphical comparison.

5. A blogpost about the internal workings of Kubernetes [13]. The blog post describes in great detail the internal workings of several Kubernetes components. This information is otherwise undocumented and extracted from reading the source code of the system. Because studying source code is difficult and time consuming we made our findings available online[1] such that others don't have to do the research again.

The contributions have been recognized by the open source community and others. Brendan Burns, the lead engineer of Kubernetes, has invited us to give a demo of the system to several members of the community. In this demo we will be able to demonstrate system functionality and receive feedback. Eventually we aim to contribute the system as a third party solution to the Kubernetes repository. Additionally we have been asked by Kubernetes product manager David Aronchick to write a blog post about working with Kubernetes. This will be done after the bachelor project is finished due to the limited time.

1.5. Structure

The structure of the thesis is as follows: Chapter 2 introduces relevant technologies and concepts as well as a survey of related work. Chapter 3 presents the main problem and outlines requirements for a solution.

[1] `http://borismattijssen.github.io/articles/kubernetes-informers-controllers-reflectors-stores`

Chapter 4 describes processes including research, development and collaboration. The design and implementation of Flower is covered in Chapter 5. Evaluation of Flower including the experimental work done to verify Flower can be found in Chapter 6. Chapter 7 discusses the implications of Flower and provides possibilities for future work. Finally Chapter 8 summarises the thesis and presents the final conclusion.

2 Background

This chapter is used to present background information related to the Flower project. In particular we describe any desired prior knowledge about technologies and concepts in Section 2.1. Section 2.2 describes various systems related to Flower.

2.1. Technologies and concepts

Throughout this report we attend to matter for which a prior knowledge of certain technologies and concepts is desired. In this section we describe those technologies and concepts in more detail. Section 2.1.1 gives an introduction to workflows and workflow management systems, Section 2.1.2 introduces the concept of containers, Section 2.1.3 explains the Kubernetes system, Section 2.1.4 describes the Kubernetes API server, and Section 2.1.5 goes into detail about Nerdalize's CloudBox.

2.1.1. Workflow Management

Many companies and scientists need to process increasingly larger amounts of data [27]. To support the increasing demand of computing power needed to processes this data, many computers are interconnected forming a grid or a cloud. Data processing is often done in discrete steps, where each step depends on the output of previous steps in the system. In the scientific world a common sequence of steps is analysis, simulation, data management, and visualisation [22]. The abstract representation of steps and dependencies is called a workflow. Workflows can be modelled as directed acyclic graphs (DAGs), in which steps are nodes and edges represent dependencies between steps. Using DAGs as a model allows for the use of extensive scientific research on graphs [34, 35] to optimize performance [33], reliability [26], and scalability [23] of the workflow. An example of a DAG is shown in Figure 2.1. The figure shows six steps and various dependencies. Step `B` can only start executing when step `A` finishes execution. Steps `C`, `D`, and `E` can each start in parallel when step `B` finishes.

Workflow management systems (WMS) have been around since the start of grid computing [24, 28, 30, 36]. According to Taverna [18] a WMS is "a piece of software that provides an infrastructure to setup, execute, and monitor scientific workflows". In other words a WMS schedules steps on appropriate machines and monitors their execution and dependencies, so users can focus on describing the workflow. An example of a workflow management tool is Pegasus [28] which lets the user describe their workflow in XML format. Pegasus maps steps to available machines and transfers necessary data required for execution. Other systems like Triana [24] provide users with a visual tool to create workflows and do monitoring. WMSs often contain functionality to optimize workflow performance, scalability and reliability. Section 2.2 will describe some workflow management systems in more detail.

2.1.2. Containers

A *container* is an operating-system level virtualization 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 do whatever is needed; in particular run applications with the necessary resources and environment configuration [29].

A container runs multiple processes in an isolated environment. Each process has a *process id* which is unique within the container itself. Processes outside the container are not visible inside the container.

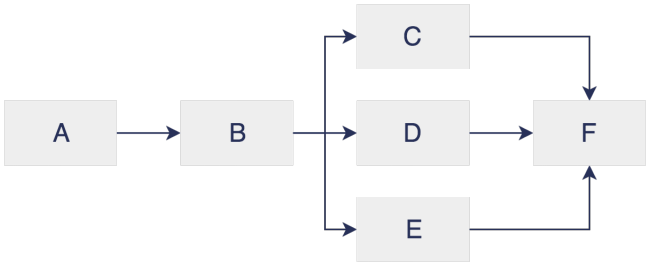


Figure 2.1: An example of a workflow presented as a DAG.

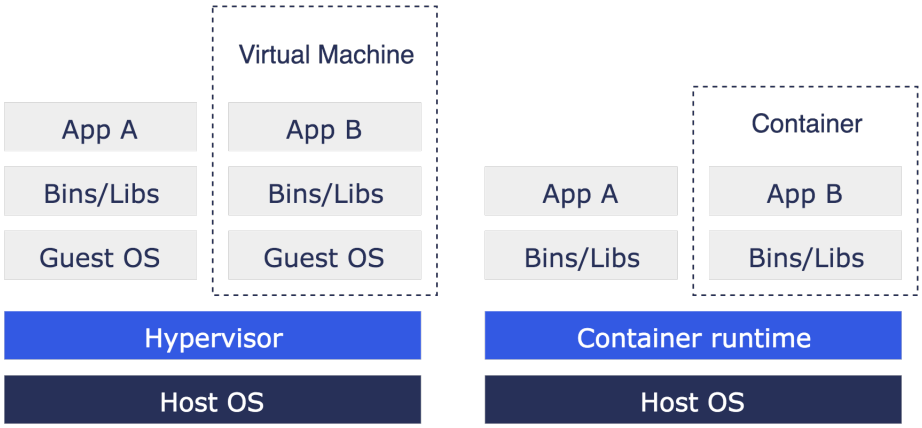


Figure 2.2: Comparison of virtual machines (left) and containers (right). Based on a figure from the Docker documentation (`https://www.docker.com/what-docker#/compare-block`).

This also applies to other resources such as networking, disk and users. Isolation is provided by *cgroups* and *namespaces* as Linux kernel modules.

When describing containers, *virtual machines* (VMs) are often used as an analogy. Containers and VMs are similar in that both are virtualized environments that run on a host machine. The differences are shown in Figure 2.2, showing the software stacks used by VMs and containers. The figure shows VMs have their own *operating system* (OS) called the *guest OS* which is separated from the host OS. Containers on the other hand share the kernel with the host OS. Using containers rather than VMs provides two major advantages: their size is smaller (MBs vs. GBs) and booting is faster (seconds vs. minutes).

In recent years containers have increased in popularity. This increase in popularity can be attributed to the release of the Docker software in 2013, which makes it easy to build, distribute and run containers [6].

2.1.3. Kubernetes

Kubernetes is an open-source project by Google for orchestrating containers on multiple hosts [13]. The main features of Kubernetes are scheduling and replication control. The scheduler assigns pods to an available host. A pod is the atomic unit in Kubernetes; it contains one or more tightly coupled containers. Replication control can be used to ensure that a specified amount of copies of a pod exist. The replication controller monitors pods and creates new instances in case of pod failure. Kubernetes consists of one master node and multiple worker nodes, as shown in Figure 2.3. The master node runs the scheduler and replication controller and exposes an API for communication with users and worker nodes. Each worker node has a *kubelet* daemon communicating with the master node through the API. Worker nodes are responsible for running pods on a host. Throughout this report we will use the term *job*. A job contains one or more pods. It terminates successfully when all of its containing pods have succeeded.

2.1.4. Kubernetes API server

The Kubernetes API server validates and configures data for API *resources*. These resources include nodes, pods, jobs and workflows and represent an entity relevant to the Kubernetes system. A resource is represented in YAML or JSON format. This format is well established and widely used, making client integrations easier.

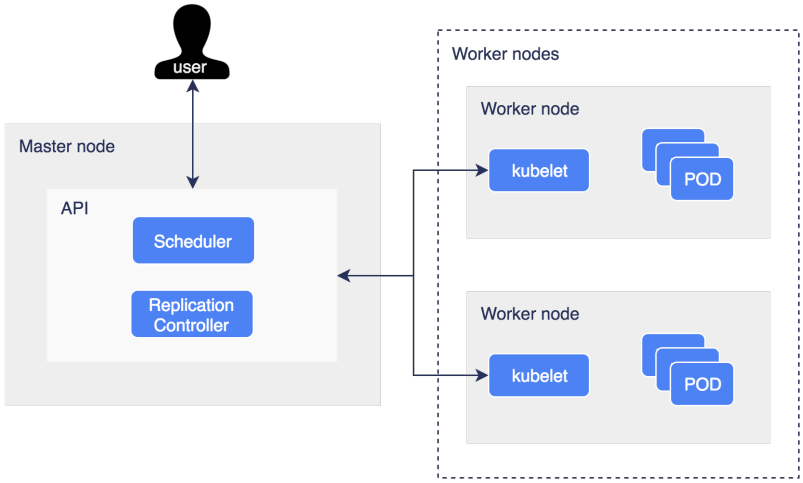


Figure 2.3: Kubernetes architecture showing master node (left) and worker (right) nodes. Based on a figure from the Kubernetes documentation (`https://github.com/kubernetes/kubernetes/blob/master/docs/design/architecture.md`).

All resources have a `spec` and a `status` field which play a central role in how Kubernetes does its orchestration. The spec is provided when a resource is created. It represents the desired state of that resource. Figure 2.4 illustrates an example spec which may be submitted to the Kubernetes API server. The spec defines a Pod with two containers, one running the `redis` image and one running a logger from the `google/glog image`. All resources have a metadata field containing meta information about the resource, not its contents. As can be seen in the figure, in this example a *label* has been set. Labels are key/value pairs used to specify identifying attributes of resources. They are used by Kubernetes to select groups of resources with similar properties. When the spec is submitted to the API server it will be stored and given a status. A status is the actual state of the resource at a given point in time, which may or may not be the same as the spec. Different Kubernetes components will constantly reconciliate to move the status towards the spec. This leads to a robust mechanism that copes well with unexpected failures and unforeseen circumstances.

```
apiVersion: v1
kind: Pod
metadata:
  name: cache
  labels:
    app: cache
spec:
  containers:
    - name: key-value-store
      image: redis
    - name: logger
      image: google/glog
```

Figure 2.4: An example Pod spec with two containers in YAML format.

The Kubernetes API server exposes a RESTful API serving as a gateway to the cluster's shared resources through which other components interact. Resources may be submitted to the API server which will store them in a distributed key-value store. The store ensures data is replicated across different nodes to be resilient to outages.

2.1.5. CloudBox

A CloudBox is the unit that will eventually make up the Nerdalize data center. CloudBoxes are placed in people's home. They consist of a water boiler and a number of servers. The water in the boiler is pumped through a heat exchanging component that transfers the heat produced by the servers to the water. From the home owners perspective the CloudBox functions much like a regular central heating system in that it

warms up water needed to warm up the house or take a shower. For Nerdalize the CloudBox is a node in a highly distributed cluster running compute intensive jobs for a wide variety of customers. The home owner receives free heating because Nerdalize pays for the electricity used to power the servers and heat up the water. Nerdalize builds a distributed data center with much lower overhead compared to traditional data centers. This is mainly because they do not need air conditioning to cool the servers, and because they do not have to build an actual data center saving on infrastructure costs.

2.2. Related work

In this section we will discuss some of the existing workflow schedulers (Section 3.1),

2.2.1. Pegasus

Pegasus is a workflow management system used to automate, recover and debug scientific computations. It is a sophisticated piece of software used by research organisations to run simulations and analysis on some of the largest computing grids in the world. These applications often involve the processing of large amounts of data. The computing grid consists of many heterogeneous machines provided by different, geographically spread out institutions. Pegasus introduces the notion of a workflow as "an abstract description of application components and their dependencies"[28]. The workflow may be represented as a directed acyclic graph (DAG). This approach leads to a highly flexible description where individual components may be replaced with alternative implementations. The abstract representation then undergoes several refinement steps geared towards making it executable and increasing performance. The result is an optimized, executable mapping from an abstract workflow onto physical machines.

2.2.2. Luigi

Luigi is a workflow manager written in Python at Spotify Inc. It helps users manage complex pipelines of batch jobs by handling dependency resolution, workflow management and visualization[2]. Luigi workflow specifications are written in Python leveraging the expressiveness of the programming language itself. This makes it easy to build up the dependency graph in which dependencies may include recursive references, date algebra etc. A visualizer is provided out of the box, providing valuable insights regarding task status, the dependency graph and workers running the tasks.

2.2.3. Airflow

The third existing solution is Airflow developed by AirBnB[1]. Airflow is similar to Luigi in that the system and the workflow definition are written in Python. Operators are a feature that distinguish Airflow. Types of operators include *sensor*, *remote execution* and *data transfers*. The sensor waits for an event to happen before proceeding to the next task in the workflow. Examples of events are a file appearing in a specified folder or the existence of a particular record in a SQL database. The second type of operator is remote execution, which triggers an external script. Finally the data transfers operator is responsible for moving data from one place to another. This data may be stored in a database of some kind or in a regular file. Airflow uses a plug-in architecture to ease the extension with new functionality.

2.2.4. Oozie

Apache Oozie is a workflow scheduler for Hadoop[3]. Oozie is tightly integrated with the Hadoop stack supporting Java map-reduce, Streaming map-reduce, Pig, Hive, Sqoop and Distc. A workflow is defined by an XML file which may contain two types of nodes. The first type is a *control flow node*, providing a mechanism to control the workflow execution path. The second type of node is the workflow *action node*, which describes the execution of a computation or processing task. XML properties are used to create connections between nodes. These connections may be conditional meaning that a corresponding predicate needs to be satisfied before the transition takes place.

3

Problem Analysis

3.1. Overview

In this chapter we will first define the problem as formulated by Nerdalize. From this problem definition we derive a problem statement which will be the central question we seek to answer during the project. In Section 3.3 we list several user stories which describe the system from the end-user perspective. These stories serve as a simplified description of the requirements defined in Section 3.4. Requirements are divided into categories and prioritized.

3.2. Problem Definition

To achieve optimal cluster utilization and ease of use for its customers, Nerdalize is building its main product the Nerdalize Cloud Engine (NCE) based on modern container orchestration technology. After extensive research Nerdalize has chosen to use the open source Kubernetes system as the basis for the NCE. In line with its business goals the first step is allowing clients to run high throughput workloads in a user friendly manner. Many workloads contain large parallelizable components. Running these on a cluster delivers a reduction in total workload throughput time. Modern cluster schedulers support complex workflows that allow non-trivial multi-step workloads to be executed automatically. Kubernetes has basic support for the concept of Job, but at the time of writing does not have workflow support.

Because Nerdalize uses cloud boxes installed in people's homes instead of traditional servers, the scheduling of a workflow poses new challenges and opportunities. The heat produced by a cloudbox is used to warm up water in a boiler attached to it. This water may then be used by the home owner however he or she likes. Because the amount of liters inside the boiler is fixed there is a maximum temperature of the water. When the water reaches maximum temperature, additional heat produced by the cloudbox is wasted. This waste may be avoided by using a scheduling strategy that takes the boiler's water temperature into account. Workflow steps should be scheduled in homes where the boiler has not yet reached the maximum temperature to avoid energy being wasted. Scheduling strategies should be easily replaceable such that new ones may be developed and used in the future.

The high level goals of this project are:

1. Research the state of the art in workflow management.
2. Research the current support in Kubernetes for workflow management.
3. Design and implement a workflow management system in or on top of Kubernetes, preferably in cooperation with the Kubernetes community.
4. Extend the workflow management system to support a replaceable scheduling policy that takes boiler temperature into account.
5. Carry out experimental work to measure the impact of the developed scheduling policy.

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

The derived problem description is split up into two questions:

1. *"How can workflows be executed in a distributed containerized environment?"*

2. *"How can workflows be scheduled to reduce the waste of generated heat?"*

3.3. User Stories

The interviews with Nerdalize helped clarify which use cases they encountered and what is needed to satisfy them. We then composed a list of user stories that encapsulate all functionality, such that we end up with a high level description of system functionality. The user stories are all in the format *As a <user>, I want <action>, so that I <reason>* to ensure consistency.

1. As a user, I want *the system to do scheduling in a distributed environment*, so that I *can make use of parallelism to improve turnaround time.*

2. As a user, I want *the system to take boiler temperature into account*, so that I *do not waste energy.*

3. As a user, I want *to add steps that depend on the completion of other steps*, so that I *can create a workflow.*

4. As a user, I want *to add or remove steps during workflow execution*, so that I *can make changes at run-time.*

5. As a user, I want *to ensure that certain steps do not run on the same machine*, so that I *can improve reliability.* network overhead.

6. As a user, I want *to customize the scheduler*, so that I *can tailor it to my needs.*

7. As a user, I want *to interact with the workflow through an API*, so that I *can integrate it with my existing system.*

8. As a user, I want *to run workflows on my local machine*, so that I *can test before deployment.*

3.4. Requirements

System requirements have been formulated in the research phase as well as during the implementation phase of the project. Most requirements emerged after interviewing Nerdalize employees. During the development of the system requirements have been dropped and changed. Changed requirements are described in more detail in Section 7.1.3. This section only contains the most up-to-date requirements. The original list of requirements can be found in the Research Report in (app).

3.4.1. Functional Requirements

Based on the list of user stories we created the following functional requirements. The requirements are grouped into four categories which are: *workflow manager*, *step*, *workflow* and *scheduler*. The workflow manager is responsible for providing the scheduler with steps whose dependencies have been resolved. The scheduler ensures steps are assigned to a suitable machine according to a specified policy. A scheduling policy uses information about the workflow and the boiler temperatures to make a scheduling decision.

Requirements are prioritized using colored boxes representing categories as defined by the MoSCoW[14] method. The green box (🟩) stands for must have, the half full orange box (🟧) stands for could have and the almost empty red box (🟥) stands for won't have.

Workflow manager:

1. 🟩 The workflow manager executes a workflow based on a formal definition of steps and constraints.
2. 🟩 The workflow manager monitors the execution of the workflow
3. 🟥 The workflow manager is responsible for providing data required by a step.

step:

4. 🟩 steps' execution may depend on the status (failed, succeeded, running) of one or more other steps.
5. 🟥 steps may be other workflows.
6. 🟧 steps can be added and/or removed through the Workflow Manager API during the execution of a workflow.

Workflow:

7. 🟩 A workflow may run on a single machine or a cluster of machines.
8. 🟩 A workflow may be cancelled during execution.
9. 🟧 Resource usage of a workflow may be limited as specified by the user.

Scheduler:

10. 🟩 The scheduler's policy should be customizable so that users may tailor it to their needs.
11. 🟩 The default scheduling policy is able to schedule steps on the machine with the lowest boiler temperature.

Requirement 3 and 5 are beyond the scope of this project, due to time constraints. The system will be extensible in such a way that functionality may be added at a later point in time. Requirements 6 and 9 are the least crucial at this moment in time. We decided priorities in collaboration with Nerdalize to address the most pressing business needs first.

3.4.2. Non-functional Requirements

Apart from functional requirements, the interviews and user stories also require a set of non-functional requirements for the system to meet all the criteria. Non-functional requirements are used to judge the operation of a system. One cannot write code to implement them, rather they are emergent properties that arise from the system as a whole. The requirements are listed below in no particular order.

Usability

Usability refers to the degree to which users of a particular system are able to effectively and efficiently achieve objectives. It is very important for our system that it is designed such that the learning curve will be shallow rather than steep. This allows users to quickly get started and increase the likelihood the system will get adopted by the community.

Open source

The developed system should 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.

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 we will have less time 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.

4

The research and development process of Flower

During the project we have used processes and tools for assistance. These processes and tools can be divided into three categories. Section 4.1 describes the research processes, Section 4.2 goes into detail about the chosen development methodology, the chosen programming language, the testing strategy used, and the development tools used. Section 4.3 discusses collaboration with external parties and within the team.

4.1. Research

The research processes described in this section were used to gather necessary information for the design and implementation of Flower. 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

Five weeks before the official start of the project, we had a conversation with Nerdalize about the subject of the project. Eric from Nerdalize gave us an extensive list of technologies we had to become familiar with, which can be found in Appendix B. We had very little prior knowledge of the container ecosystem and distributed systems in general. To get familiar with *Docker* we watched two Youtube series from LearnCode.acadmey and Docker, and we watched several talks from DockerCon. Besides watching videos we read the recommended books about Docker. *Go* is the language used by Docker and Kubernetes, so we had to get familiar with that too. To learn Go we used several approaches before and during the project . Before the project we walked through the 65 tutorials on gobyexample.com. At the time Nerdalize had just decided to use Kubernetes for container orchestration. To get a basic understanding of Kubernetes we read Kubernetes Up & Running , watched videos on YouTube , and read the about the basic concepts on GitHub and the official docs . Based on recommendations by Nerdalize we got familiar with the following technologies related to *Distributed Systems*: Ansible , OpenStack , CoreOS , Hyper.sh , and etcd .

A week before the start of the project we had another conversation with Nerdalize about the subject of the project. We were told that the initial problem "Does a multi-tenant cloud require a 'classic' Cloud Management Platform, or can we build this more easily and flexibly with modern, container-based tooling such as Kubernetes and/or Mesos?" was already solved by Nerdalize engineers. Together with Nerdalize we decided on a new subject, which led to the problem description as described in Section 3.2. Although the subject changed, most research we had done was still very valuable.

4.1.2. During the project

Most of the research that is at the basis of Flower was conducted during the first two weeks of the project, also known as the *research phase*. At the end of these weeks we wrote a report describing the main results. The full report can be found in Appendix A. During the research phase and throughout the project we used three types of research: *interviews*, *literature*, and *reading source code*. Each of these is described in a separate section below.

Name	Company / Group
Ad van der Veer	Nerdalize
Eric Feliksik	Nerdalize
Mathijs de Meijer	Nerdalize
Remy van Rooijen	Nerdalize
Jesse Donkervliet	Nebu Bachelor Thesis
Tim Hegeman	Nebu Bachelor Thesis
Alexey Ilyushkin	Distributed Systems group TU Delft
Dario Minonne	Amadeus (FR)

Table 4.1: People we interviewed throughout the project.

Interviews
We interviewed people from different backgrounds for different reasons. These people, along with the company or group they belong to, are listed in Table 4.1. We used the interview with Eric Feliksik from Nerdalize to get a better understanding of the context of the problem. During this interview, Eric gave us a general overview of the technologies involved in the project. We interviewed Ad van der Veer and Mathijs de Meijer from Nerdalize, to go over different usecases relevant for Nerdalize. These use cases turned into user stories and served as the basis of our requirements. Remy van Rooijer works on the design of cloudboxes and boilers at Nerdalize. We interviewed him to get a better understanding of the hardware and the physical properties of the boiler. This understanding helped us design Heat Scheduler. Jesse Donkervliet and Tim Hegeman from Nebu did their bachelor thesis two years ago and got a very good grade. We interviewed them to ask about the way they approached the project and to see if we could learn something from their mistakes. Alexey Ilyushkin is a PhD student working in the Distributed Systems group at the Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS/EWI), TU Delft. He works on problems related to scheduling workflows. We used the interview with Alexey to get an insight of the different scheduling strategies and to get a better understanding of the difficulties that arise when scheduling workflows. Dario Minonne is an engineer at Amadeus in France who has also been working on a workflow management system for Kubernetes. The interviews with Dario were used to research the current state of his implementation and to get familiar with the Kubernetes community.

Literature study
We did a literature study for two main reasons. On the one hand we wanted to acquire background knowledge on workflow management systems in general. Sources for this information were scientific papers and books . On the other hand we wanted to survey related work, to find ways in which problems had been solved before in workflow management systems. Sources for the related work were scientific papers as well as project's homepages .

Researching source code
Throughout the project we have researched the Kubernetes source code. This research was necessary to get a deeper understanding of the working of certain Kubernetes components as well as to learn from their design patterns.

4.2. Development

This section is used to describe how we dealt with development of the system. In particular we discuss; the chosen development methodology in Section 4.2.1, the programming languages used in Section 4.2.2, testing strategies in Section 4.2.3, documentation in Section 4.2.4, and tools in Section 4.2.5

4.2.1. Development Methodology

During the project we made use of the SCRUM software development methodology to manage product development. Scrum is an iterative and incremental agile software development framework.

Scrum allows a small team of developers to cope with changing requirements and shifting priorities. Scrum also encourages team members to communicate often and openly about progress and encountered problems. The reason we chose an iterative approach is because we predicted changing requirements.

#	Weeks	Main activities	Focus
0	1-2	Acquire background information, gather requirements, create preliminary design	Research
1	3-4	Development of Flower	Development
2	5-6	Development of Flower	Development
3	7-8	Development of Heat Scheduler and writing final report	Development
4	9-10	Writing final report and prepare for presentation	Reporting

Table 4.2: An overview of all SCRUM sprints.

An alternative approach would have been the waterfall model. That model works well when all requirements are clear at the start of the project and are unlikely to change. This not being the case made it insuitable for us.

We decided to use two week sprints. Sprints always end with a demo to show the customer the latest stable version of the product. One week sprints seemed too short to both introduce new features and have a working demo at the end of the week. Sprints longer than two weeks would make the process less agile, since the project only consists of eight weeks of development. Two weeks seemed like a good duration for a sprint and this turned out to work very well. We were able to deliver a substantially improved product at the end of each sprint, while still coping with changes in requirements. An overview of the main progress made during each sprint is given in Table 4.2.

Sprint 0 was the research phase of the project. We used this sprint to do a background study, gather requirements and design a preliminary architecture. Sprint 1 was used to create the first functionality of Flower. Flower was able to execute workflow steps, without taking dependencies into account at the end of sprint 1. Sprint 2 was used to add all remaining functionality to Flower. At the end of the sprint Flower was able to execute workflows and take dependencies between steps into account. In sprint 3 we developed Heat Scheduler and started working on the final report. Sprint 4 was fully dedicated to the final report and to the presentation.

At the start of the project we created a project roadmap which can be found in Appendix C. This roadmap served as the initial contents of the product backlog; a list of tasks to be completed before the end of the project. During the project several requirements changed, which caused modifications to the product backlog. The changed requirements can be found in Section 7.1.1.

To keep track of the product backlog and the sprint backlogs we used Trello [20]. Figure 4.1 shows the Trello system. We created one Trello *list* for the product backlog, one list for the sprint backlog, one list with work in progress tasks and a separate list for every finished sprint. Lists contain cards representing finished or unfinished tasks. The product backlog contained high level tasks, such as 'steps can only run when their dependencies have been resolved'. When tasks moved from the product backlog to the sprint backlog they were broken up into smaller, more descriptive tasks.

4.2.2. Programming languages

Most code written during the project was written in the Go programming language. The available concurrency constructs such as channels make it well suited for distributed applications. Channels are a way to exchange information between threads significantly simplifying concurrent programming. We used this feature a lot in our project. The problems solved using channels are much more challenging in other languages. Go is very popular among developers working on container related technologies. This is largely due to the increasing popularity of the container platform Docker[6], which itself is written in Go. As a result there are many libraries available to solve problems common in distributed systems and interface with other related technologies. Lastly we plan to contribute to the Kubernetes community. Kubernetes itself is written in Go. The decision to choose a language the community is familiar with will increase the likelihood of the project gaining adoption. It will also make it easier for other developers to understand, encouraging modifications and engagement.

Apart from Go we used Javascript to create graphical user interfaces (GUI). The goal of these interfaces was to visualize what was going on in the system. We found writing a web application that made use of the Kubernetes API was the easiest way to do this. Any machine is able to access the GUI as long as it has a browser installed. Using Javascript gave us access to the many libraries available to create visualizations, speeding up development.

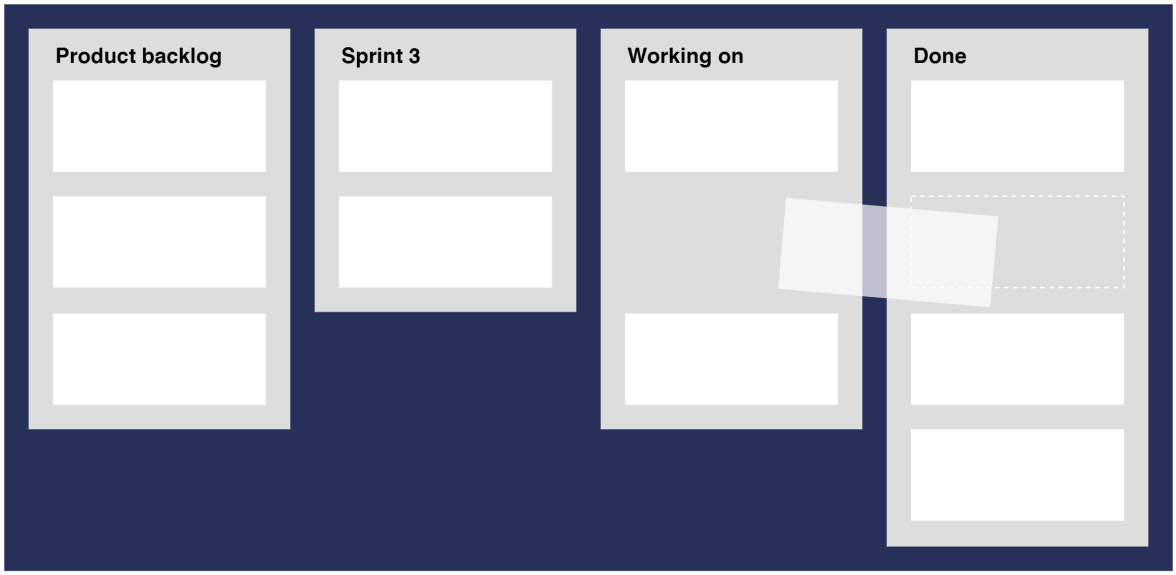


Figure 4.1: The Trello system with multiple lists holding different cards. The second card in *Working on* is being moved to *Done*.

4.2.3. Testing

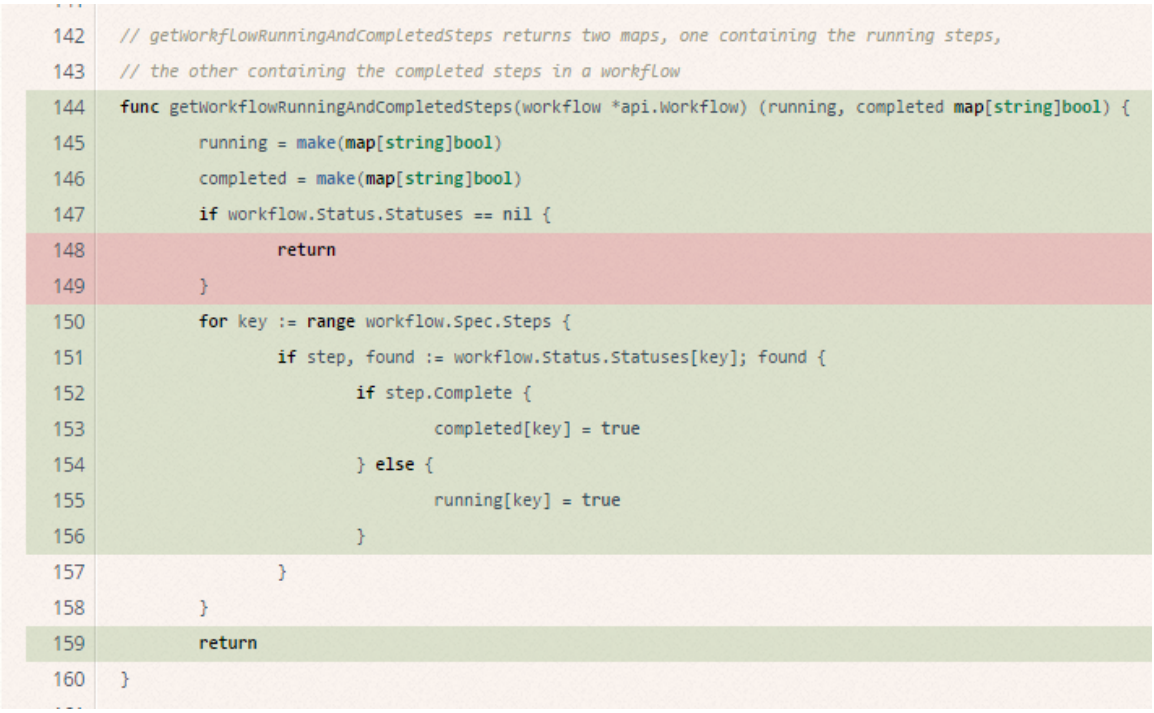
Every piece of code added to the system should ideally not contain any unexpected behavior. Tests are well suited to ensure this is indeed the case. Unit tests provide a way to verify that a piece of code, in isolation, produces a desired result. The added benefit of unit testing code is to prevent regressions. These occur when new functionality is added which breaks old code. By running a suite of tests on every build, new as well as old code is tested to ensure no regressions take place. End-to-end tests are used to test large parts of an application. We used this to test the system from a user perspective along with manual testing.

Unit tests

For unit tests we used Go's testing framework which is very minimal. The framework only provides a testing object which is passed to test functions. The testing object may perform assertions and report failures when they occur. Tests are written in a file starting with `_test.go`. This allows the test runner to find tests automatically. We chose the default testing framework because it is widely adopted in the Go community. It provides test coverage as well as benchmarking without any alterations to the testing code. Kubernetes uses the default testing framework which is convenient as we were able to copy and alter the tests for the components we reused.

Continuous integration

A convenient way to automate the testing process is through continuous integration (CI). For our project we used Travis CI [19]. We tried several alternatives which each did not do exactly what we wanted. In particular we tried Jenkins [10] and Wercker [21]. After realizing these solutions were unsuitable we made a shellscript to build the system, run all tests, and measure coverage. This is similar to what would happen on a CI server. To run our end-to-end tests we want to spin up Kubernetes on the CI server. As the instance on the server itself runs inside a Docker container this means support for recursive Docker support is required. The two solutions listed above both do not provide this whereas Travis does. The continuous integration server allows every code push to a remote repository to trigger the full set of tests. In case tests fail, a notification is sent describing what went wrong. This way of testing also makes sure that code runs on all machines rather than only on one developer's machine. Later we added code coverage measurements to the continuous integration process with coveralls [5]. By running Go's default testrunner with the `-coverage` flag it is able to generate a coverage file describing coverage line by line for all packages. This file is then submitted to coveralls through their API. On their website they provide a user interface to see how well each tested. Tested lines of code are colored green whereas non-tested lines get a red color, as can be seen in Figure 4.2. This helped us to ensure important parts of the code are covered by the tests.



```
142  // getWorkflowRunningAndCompletedSteps returns two maps, one containing the running steps,
143  // the other containing the completed steps in a workflow
144  func getWorkflowRunningAndCompletedSteps(workflow *api.Workflow) (running, completed map[string]bool) {
145          running = make(map[string]bool)
146          completed = make(map[string]bool)
147          if workflow.Status.Statuses == nil {
148                  return
149          }
150          for key := range workflow.Spec.Steps {
151                  if step, found := workflow.Status.Statuses[key]; found {
152                          if step.Complete {
153                                  completed[key] = true
154                          } else {
155                                  running[key] = true
156                          }
157                  }
158          }
159          return
160  }
```

Figure 4.2: A coverage visualization for the getWorkflowRunningAndCompletedSteps function.

Load test

Lastly we performed a load test on the system to see how it performs on a large cluster with a large workflow. The cluster we used consisted of 50 machines provided by Google Cloud Platform. The workflow we used had a thousand steps and almost a thousand dependencies. It was taken from the Pegasus website in DAX format which is incompatible with our YAML format. Because generating such a large workflow by hand would take a long time we wrote a DAX to JSON converter instead. It converts arbitrary DAX files to a corresponding YAML file. To simulate work being done in the steps we made a step sleep for thirty seconds before it transitioned to completion. Even under load the system performed well completing the entire workflow in a time very close to the critical path. The main overhead was caused by the time it takes for the Kubernetes API server to be notified when a step is completed.

4.2.4. Documentation

Documentation played a central role throughout the development process. We applied two different types of documentation with partially overlapping goals. The first one being the documentation of code in the source files. Every single function is preceded by a descriptive comment on what a function does and how it should and should not be used. The Go programming language contains a tool called fmt[9], which ensures every block of code contains a descriptive comment and is formatted according to the Go style conventions. This led to a well documented product which is beneficial in several ways. More than once we had to refactor parts of the codebase. It was then useful to read the comments before making a change to ensure no existing functionality would break. We do believe in self-documenting code and spent considerable time thinking of appropriate function names. After our project however, other developers will most likely be working on our codebase. What seemed very straightforward to us at the time of writing may not be so obvious to them when doing modifications. In this scenario comments help clarifying the behavior without having to make any assumptions or guesses. Documentation may be found online on the GoDoc [8] website.

The second type of documentation was writing a summary at the end of every day during the project. In this summary we would write what had happened that particular day and how the system changed. This helped us tracing back problems and making it easy to find the reason why a particular change had been made, even weeks after it took place. An added benefit of this chronological sequence of events was how it made writing the report much easier. Especially this section in which parts of the development process are described.

4.2.5. Tools

To support us with the creation of Flower and Heat Scheduler we used several tools. This section describes the tools we used, which are listed in Table 4.3.

Godep is a dependency managing tool for Go. Godep will look in the source code for imported packages. These packages are downloaded to the project in a folder called `vendor`. Using Godep ensures working code, even when the external packages update their code.
Gofmt formats Go programs. Using Gofmt ensures that code is always formatted in the same way. This is very convenient when working in teams, because normally everyone has their own formatting preferences. Using Gofmt made reviewing pull-request much easier for us. Only differences in real code are shown, such that you do not get distracted by differences in code format.
Gofmt formats Go programs. Using Gofmt ensures that code is always formatted in the same way. This is very convenient when working in teams, because normally everyone has their own formatting preferences. Using Gofmt made reviewing pull-request much easier for us. Only differences in real code are shown, such that you do not get distracted by differences in code format. **Go build** builds Go programs. Go build is run each time a `.go` file is saved. Using this tool helped us develop faster, since compile errors are shown in the IDE in real-time.
Go test tests Go programs or packages. We used go test to test our code as described in Section 4.2.3 and generate code coverage profiles.
Travis CI is a hosted continuous integration service used to build and test software projects hosted at GitHub. Travis CI is invoked by GitHub each code is pushed. We used Travis CI to make sure the program builds and no test fail, before merging code to the master branch.
Go vet examines Go code and reports suspicious constructs. We used go vet to warn us when our code contained style mistakes. Examples of style mistakes are uncommented public functions, comments that are syntactically incorrect, or struct names that stutter with the package name.
Go tool cover interprets code coverage profiles generated by `go test`. The tool comes with an HTML generator. We used the HTML generator to view coverage results in a web browser.
Coveralls is a hosted code coverage visualizer. We sent coverage reports to coveralls on each build, using Travis CI. Coveralls can be used to visualize coverage reports per git branch.
Grafana is a tool used to visualize metrics. Grafana makes is easy to create plots of real-time data. We used Grafana to visualize the boiler temperature for our simulation. Results and a screenshot of the simulation can be found in Section 6.2.
Git is a version control system. We used Git during the development of Flower and Heat Scheduler. Two major version control systems exist: Git and Subversion. We chose to use Git because of numerous reasons of which the most important are: we had prior experience with Git, GitHub supports only Git, Git is much faster than Subversion, and Git is easier to use than Subversion.
GitHub is an online hosted git repository service. There are many more git hosting services, we chose GitHub because of several reasons. We wanted a free and open repository such that everyone in the world can see our project. GitHub comes with *pull-requests* and *issues*, which are both very convenient collaboration tools. Github is also very easy to use and we had prior experience working with it.
Atom is a text editor with good Go support. Atom was mainly chosen for its Go support and because it was recommended by Nerdalize employees.
Vim is a terminal-based text editor. One of the team members did all his development in Vim. Vim also has good Go support and is mainly famous for its many keybindings.
Kubectl is the command line interface used to communicate with Kubernetes. We used kubectl as the main way of interacting with Kubernetes. Kubectl has many commands, of which we manily used: `kubectl get jobs` to list all jobs in the cluster, `kubectl create -f [file]` te create a new resource based on the specified file, and `kubectl delete job [job]` to delete a job.
Docker is a tool to build, run and ship containers. Kubernetes uses Docker as their default container run-time. Sometimes it was more convenient to use the `docker` command line interface, than to use `kubectl`. We also used Docker to build containers for Flower and the different Heat Scheduler components.
Make is a build automation tool. We used build to automatically build and ship docker containers to the container registry of Google Cloud.
Wireshark is a network protocol analyzer. Sometimes communication between Flower and Kubernetes was hard to track when debugging. Wireshark helped us in the debugging process, as it logs all HTTP traffic in chronological order.

Tool	Category	Website
Godep	Development	https://github.com/tools/godep
Gofmt	Development	https://golang.org/cmd/gofmt/
Go build	Development	https://golang.org/pkg/go/build/
Go test	Development	https://golang.org/pkg/testing/
Travis CI	Development	https://travis-ci.org/
Go vet	Analysis	https://golang.org/cmd/vet/
Go tool cover	Analysis	https://godoc.org/golang.org/x/tools/cmd/cover
Coveralls	Analysis	https://coveralls.io/
Grafana	Analysis	http://grafana.org/
Git	Code management	https://git-scm.com/
GitHub	Code management	https://github.com/
Atom	Editor	https://atom.io/
Vim	Editor	http://www.vim.org/
Kubectl	Deployment	http://kubernetes.io/docs/user-guide/kubectl-overview/
Docker	Deployment	https://www.docker.com/
Make	Deployment	https://www.gnu.org/software/make/
Wireshark	Network analysis	https://www.wireshark.org/
DAX to JSON	Utility	http://bit.ly/1UjKRFr
JSON to YAML	Utility	https://www.browserling.com/tools/json-to-yaml

Table 4.3: Tools used to create Flower and Heat Scheduler.

DAX to JSON is a utility we created to allow for faster testing of large workflows. The Pegasus [28] project comes with a workflow generation tool called Pegasus Generator. The output of Pegasus Generator is in DAX file format, which in fact is XML. Pegasus Generator has some default workflows, which can be up to 1000 nodes. By using the *DAX to JSON* converter we were able to quickly generate large input files for Flower.

4.3. Collaboration

During the project we collaborated with various parties. Most notable is collaboration with Nerdalize team members, the open source community and collaboration within the team. Collaboration with Nerdalize and within our own team is described in Section 4.3.1, community engagement and communication is described in Section 4.3.2, collaboration tools used during the project are further detailed in Section 4.3.3.

4.3.1. Nerdalize

During the project we worked at the Nerdalize office. Nerdalize provided us with our own desk, which was in the same office as the other Nerdalize employees. Working in the same office as the Nerdalize employees proved to be very useful to us. It allowed us to quickly tap into their rich pool of knowledge about Go, Docker, and Kubernetes.

At the start of the project we had no experience writing Go programs. During the first weeks of the project we participated in a Go workshop organized by Nerdalize employee Ad. Ad is an experienced Go programmer, with much knowledge about Docker and Kubernetes as well. The workshops were held every Monday morning, for which we had to prepare homework. Participating in these workshops helped us understand the basics of Go and taught us the best practices when writing Go code.

During the project we had a weekly meeting on Friday afternoons with Nerdalize CTO Mathijs. We used these meetings to report the current status of the project and to discuss any encountered problems. The meetings were often also used as the *sprint review*. During the project we have encountered some challenges with working with the open source community. It has been very helpful to discuss these challenges with Mathijs.

Nerdalize has a *beer and demo* moment on Friday afternoon, during which a team of Nerdalize employees demos their project. We have demoed the latest stable version of our code, during these *beer and demo* moments three times. The demos were always at the end of a SCRUM sprint, to conform to the SCRUM methodology. Doing these demos kept us motivated to deliver a working product throughout the project.

Tool	Category	Website
Hangouts	Communication	`https://hangouts.google.com/`
Slack	Communication	`https://slack.com/`
GitHub	Storage	`https://github.com/`
Google drive	Storage	`https://drive.google.com/`
Kami	Utility	`https://www.kamihq.com/`

Table 4.4: Tools used to create Flower and Heat Scheduler.

4.3.2. Open source community

Kubernetes is an open source project with an active community. Channels for communicating with the Kubernetes community include: Slack [15], Stack Overflow [16], GitHub pull-requests [7], and GitHub issues [7]. These channels are actively used by members of the community to share ideas, propose extensions and report issues.

At the start of the project the idea was to include Flower in Kubernetes source code, such that we could contribute back to the community. We had noticed the existence of a proposal for a workflow management system for Kubernetes on GitHub. We contacted the contributors of this proposal via email to see if we could cooperate. It turned out that a man named Dario was in charge of the proposal. We had several video calls with him to discuss a possible collaboration. We ended up not working together but his input has been very useful to us.

We have encountered several bugs in Kubernetes, since we were working with alpha features. We used the GitHub issues system to report these bugs. We have used this same system to participate in conversations related to the bugs we found.

4.3.3. Tools

We used several tools to support collaboration and communication within the team and with external parties. Table 4.4 shows the collaboration tools that we used. The way each tool works and how we used it, is described below.

Hangouts is a video call application by Google. We used Hangouts to communicate with a professional from Amadeus because it is located in France.
Slack is a communication tool much like a chat application. Kubernetes has its own Slack group with multiple channels. Channels are used to split conversations up in to different topics. We used the `#kubernetes-users` channel to ask other users about working with Kubernetes. We also used the `#kubernetes-dev` channel to talk to Kubernetes developers, about software implementation details.
GitHub an online hosted git repository service. We used GitHub to host our code but also to keep track of issues and to create pull-requests. Communicating through pull-requests turned out to be very effective for us. Using this type of communication groups the conversation to a specific piece of code, which makes it easy to read back up on a conversation.
Sharelatex is a hosted LaTeX editor. Sharelatex allows for collaborative editing in the same file and project. We used Sharelatex for both the research report and the final report.
Google Drive is a hosted storage solution by Google. Google drive comes with its own office suite. We used Google Drive to collaboratively work on the same files and keep all project files in a central place.
Kami is an online PDF annotation tool. Kami allows users to collaboratively annotatate PDF files with comments and highlights. We used Kami extensively during the research phase of the project to highlight interesting parts in scientific papers and books.

5

Design and Implementation of Flower

This chapter describes each part of Flower in detail. Section 5.1 gives an overview of Flower as a whole. In Section 5.2 the Workflow Manager subsystem and how it relates to other subsystems is described in more detail. The section is split up in two parts. Section 5.2.1 describes the architecture, and Section 5.2.2 which describes how we implemented it. The Heat Scheduler and the different scheduling policies are covered in Section 5.3.

5.1. The Flower system

In this section we describe the architecture and the implementation of Flower in Section 5.1.1 and Section 5.1.2 respectively.

5.1.1. Architecture

Flower consists of three subsystems that together provide workflow management and scheduling. An overview of the these subsystems in the cluster is shown in Figure 5.1. The cluster - or cloud - is a set of connected machines which can be used to run containers.

Workflow Manager is responsible for executing workflows created by the user. To create a workflow, the user submits a workflow specification to Kubernetes. Kubernetes is used to store the specification and the current state of the workflow. A state diagram of a workflow is shown in Figure 5.2a. The diagram shows how a workflow transitions to the *executing* state when all preconditions are satisfied. When a workflow is executing, its steps transition from a *pending* state to a *completed* state. Figure 5.2b shows the state diagram of a step. A step can start executing when all its dependencies have been resolved. Workflows reach the *completed* state, when all steps are completed.

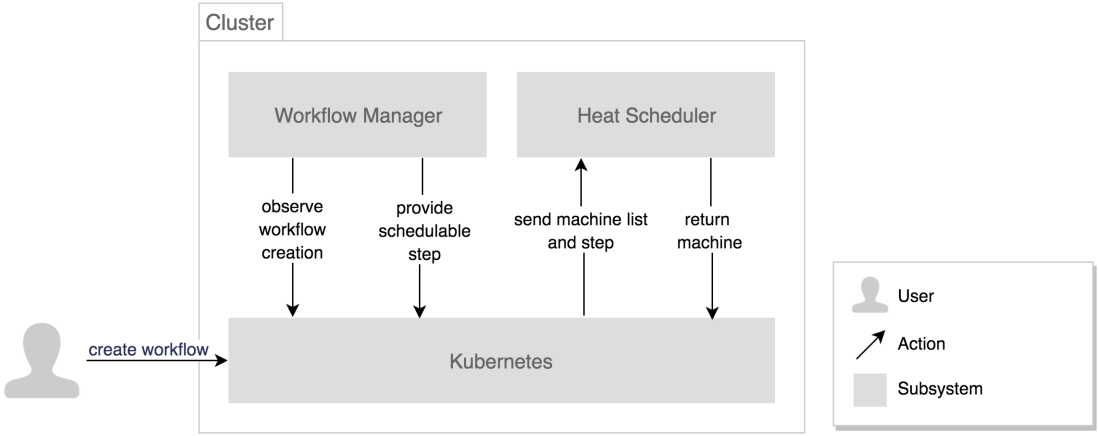
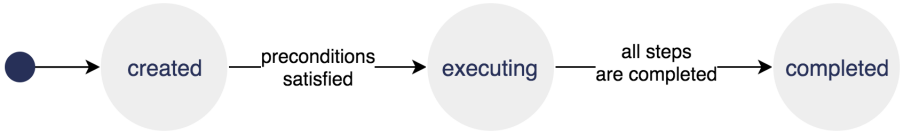
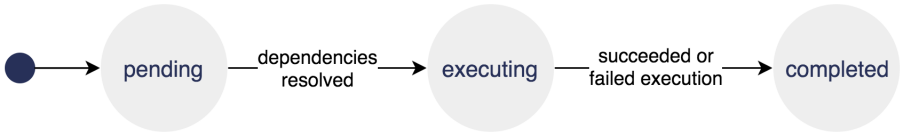


Figure 5.1: Overview of all subsystems in the cluster.

Workflow Manager observes Kubernetes for creations and modifications of a workflow specification. When



(a) The various stages of a workflow.



(b) The various stages of a step.

Figure 5.2: State diagrams of a workflow (top) and a step (bottom).

a new workflow is added to Kubernetes or when an existing one is altered, Workflow Manager will perform validation and verify which workflow steps are ready for execution. When the dependencies of a step have been resolved, it is handed over to Kubernetes to schedule it on a suitable machine.

The Kubernetes scheduler will determine which machines are suitable for the step to execute on. It will do this by checking a list of predicates for each machine. Kubernetes sends the list of suitable machines to Heat Scheduler. Heat Scheduler returns a single machine based on a custom heat-scheduling policy described in more detail in Section 5.3. The schedulable step will be executed on the returned machine.

5.1.2. Implementation

We built both Workflow Manager and Heat Scheduler on top of Kubernetes. Kubernetes is "an open source system for managing containerized applications across multiple machines" [12], which is described in more detail in Section 2.1.3. We decided to build our proof of concept implementation on top of Kubernetes for a number of reasons: Kubernetes (i) orchestrates containers, (ii) has a default scheduler which can be extended, (iii) is open source, (iv) is used by Nerdalize, making this integration useful to them.

Kubernetes is used as the central store for workflows because of its fault tolerant storage. It uses a distributed key-value store, such that data doesn't get lost when one of the machines in the cluster goes down. YAML or JSON text formats may be used to describe a workflow as these are formats Kubernetes uses to store resource representations.

Flower uses a microservices architecture [32] to run components in their own environment. Microservices are small services running one process or a few tightly coupled processes. They can be independently deployed, often in an automatic way. We use Kubernetes to deploy microservices that run inside a container. Kubernetes has advanced features for running microservices such as the *replication controller*. The replication controller is used to monitor a running microservice and restart it when it goes down. We run both Workflow Manager and Heat Scheduler as microservices. A more in-depth description of the architecture and implementation of these microservices can be found in Section 5.2 and Section 5.3 respectively.

5.2. The Workflow Manager component

This section describes the Workflow Manager component. Workflow Manager is responsible for executing workflows, which are defined as a collection of steps with optional dependencies. First the architecture is described in Section 5.2.1, after which we go over the implementation details in Section 5.2.2

5.2.1. Architecture

Workflow Manager is a workflow execution system that is built to run workflows on modern cloud services. As a proof of we chose to build Workflow Manager for Kubernetes. Workflow Manager interfaces with Kubernetes through the Kubernetes API. The API endpoints Workflow Manager uses are described in more detail in Section 5.2.2. The individual components of Workflow Manager, are shown in Figure 5.3. Each of these components and their responsibilities is described below.

Kubernetes API Server

The Kubernetes API server stores a representation of all resources in the cluster. It functions as the main gateway to communicate with Kubernetes. In the context of the Workflow Manager component, the API

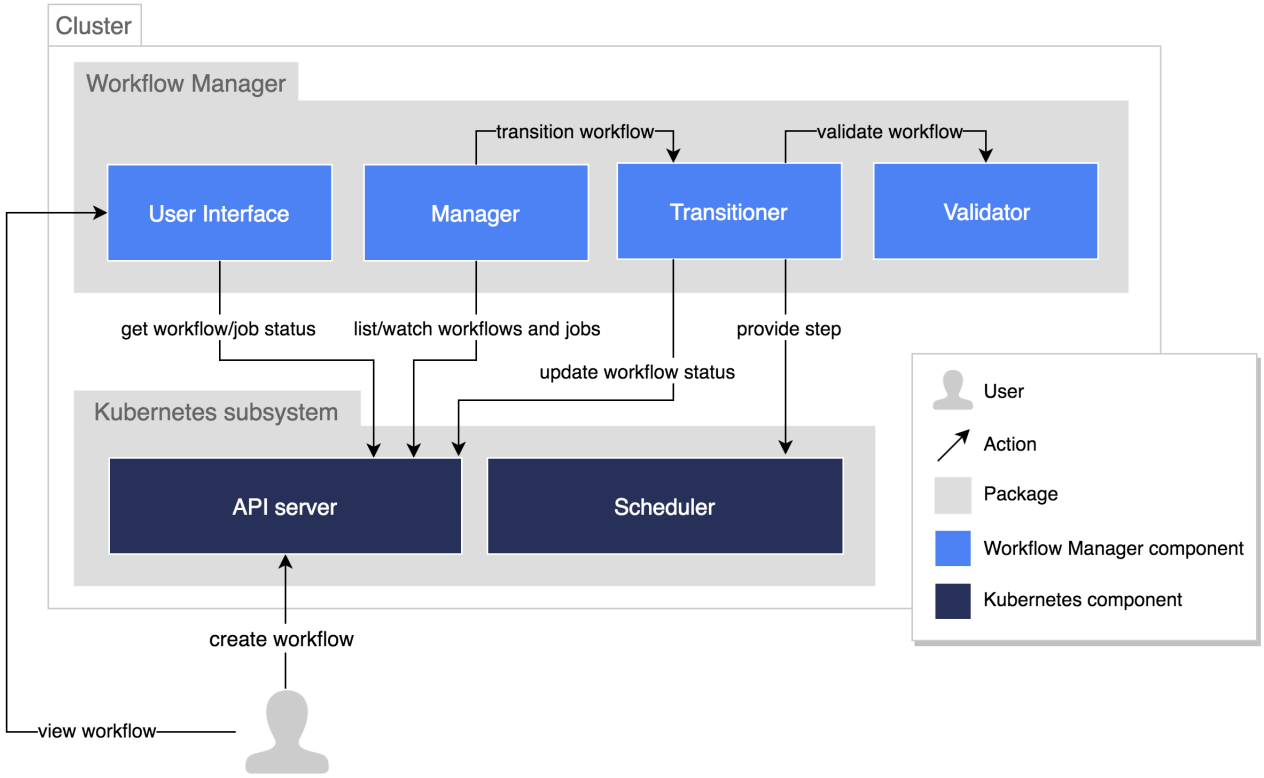


Figure 5.3: Overview of Workflow Manager package and the interactions with the Kubernetes package.

server can be seen as an upstream store that holds all workflows and jobs. To create a workflow, the user sends the workflow specification to the API server. Workflow Manager is able to observe the workflows stored by the API server and process them. Steps in a workflow correspond to jobs in Kubernetes.

Manager

The manager is responsible for providing the *transitioner* with workflows that need processing. It lists and watches workflows and jobs that are stored in the API server. Listing is used to query all workflows/jobs from the API server. Watching is a mechanism to receive updates on changes of an individual workflow or job when they occur. These changes can be anything from a change in a workflow's specification to the change of a job's status (running, completed, failed).

When the manager receives a new workflow or notices an update to an existing workflow or job that belongs to a workflow, the transitioner is called to process the workflow.

Transitioner

The transitioner is responsible for transitioning a workflow from its current state to its desired state. A workflow will only be transitioned if it is validated by the validator. The current state of a workflow is provided by the manager. The desired state of a workflow is for it to be complete. This is the case when all workflow steps have finished execution. Steps with resolved dependencies are provided to the scheduler. When all workflow steps have finished, the transitioner will set the workflow status to *complete* and update to the API server.

Validator

When a workflow is submitted it is first validated before processing starts. The validation process consists of different parts. The most important property of a workflow is that its dependency graph contains no cycles. To verify this we sort the DAG topologically. A topological order exists if and only if the graph does not contain a cycle. In Figure 5.4 the pseudocode for the topological sorting algorithm based on *Depth First Search* (DFS) [25] is shown. Other parts of the validation include, making sure the specification is valid JSON or YAML, checking if step names are unique and ensuring that steps contain all the fields for processing.

```
for all v in V do visited(v) = 0
for all v in V do mark(v) = 0
for all v in V do if (visited(v)==0) DFS(v)

DFS(v) {
  mark(v) = 1
  visited(v) = 1
  for all w in adj(v) do {
      if (visited(w)==0) {
          DFS(w)
      } else {
          if (mark(w)==1) print "found a cycle"
      }
  }
  mark(v) = 0
}
```

Figure 5.4: Topological sort based on DFS. Based on a code snippet made available for a lecture at Carnegie Mellon University (http://www.cs.cmu.edu/afs/cs/academic/class/15451-s06/www/lectures/DFS-background.txt).

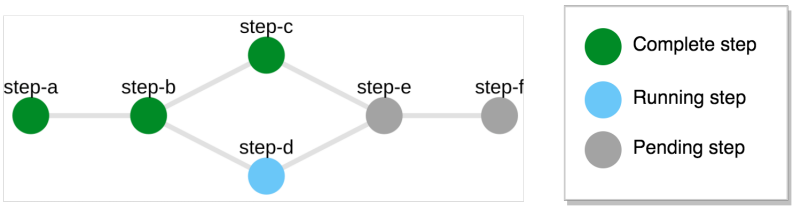


Figure 5.5: A screenshot of the user interface of Flower. Green nodes represent completed steps, blue nodes represent running steps, and gray nodes represent pending steps.

Scheduler

A step - and its corresponding job - are provided to the scheduler when it is ready for execution. The scheduler places the job on a machine in the cluster, such that it can be executed. For this project we extended the scheduler, to schedule based on a custom heat-scheduling policy. The architecture of our scheduler is explained in more detail in Section 5.3.

User interface

Flower comes with a graphical user interface (GUI) to visualize the current state of a running workflow. A screenshot of the GUI is shown in Figure 5.5. The GUI shows the workflow as a DAG, with a node for each step. Green nodes represent completed steps, blue nodes represent running steps, and gray nodes represent pending steps. The GUI polls the API server for the current state of the workflow and its jobs and updates its representation accordingly.

5.2.2. Implementation

This section describes the implementation of Workflow Manager in more detail. Workflow Manager is designed to run on top of Kubernetes, which is heavily reflected in the implementation. A sequence diagram of Workflow Manager is shown in Figure 5.7. The different components from the diagram are explained below.

Throughout this section we often refer to the workflow resource. Figure 5.6 contains an object diagram showing the composition of the workflow resource. From the diagram it can be seen that a `Workflow` is composed of a status and a specification called `WorkflowStatus` and `WorkflowSpec` respectively. The workflow specification is specified by the user, while the status gets updates by Workflow Manager as the execution progresses.

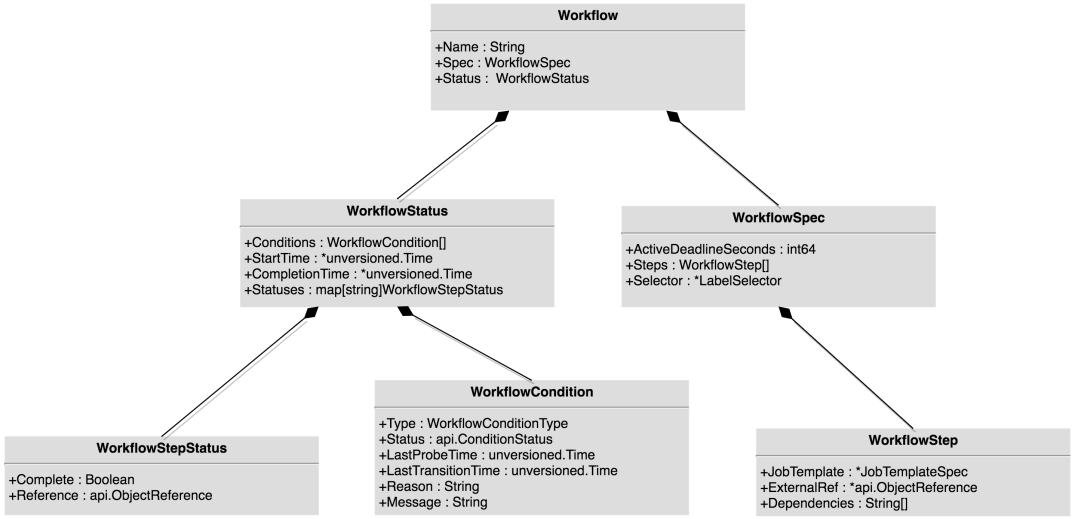


Figure 5.6: Object diagram showing the composition of the Workflow object.

Method	Path	Description
GET	/apis/extensions/v1beta1/jobs	list objects of kind Job
GET	/apis/extensions/v1beta1/watch/jobs	watch individual changes to a list of Jobs
POST	/apis/extensions/v1beta1/namespaces/{namespace}/jobs	create a Job
GET	/apis/nerdalize.com/v1alpha1/workflows	list objects of kind Workflow
GET	/apis/nerdalize.com/v1alpha1/watch/workflows	watch individual changes to a list of Workflows
GET	/apis/nerdalize.com/v1alpha1/namespaces/{namespace}/workflows/{name}	read the specified Workflow
UPDATE	/apis/nerdalize.com/v1alpha1/namespaces/{namespace}/workflows/{name}	replace the specified Workflow

Table 5.1: Kubernetes API endpoints used by Workflow Manager. Source: `http://kubernetes.io/kubernetes/third_party/swagger-ui/`.

Client

The client is used to communicate with the Kubernetes API server. Messages to and from the API server are encoded in JSON text format. Using JSON is very convenient, because Go makes it easy to convert JSON objects to structs. Table 5.1 shows the Kubernetes API endpoints that are used by Workflow Manager. These endpoints can be grouped in endpoints regarding jobs and workflows. Three observations can be made from this table that are worth describing.

The fist is the *API group*, which is the part after `/apis/` in the table. By using the third party resource API from Kubernetes, we were able to register our own Kubernetes API endpoints. We registered these endpoints under the *nerdalize.com* API group.

The second notable observation is the `namespaces` part in the table. Namespaces are used by Kubernetes to support multiple virtual clusters in the same physical cluster.

The last observation are items one and five from the table, which describe the endpoints used for watching resources. Watching an endpoint sets up an HTTP long polling connection to the server. Each time a resource changes, a new event is pushed over the HTTP connection. An example of such an event in JSON format can be found in Figure 5.8. The watching mechanism is very convenient as it allows changes to be pushed by the server. This reduces network delay compared to periodically polling the server. Unfortunately watching functionality for third party resources such as workflows, was still in *alpha* production state, hence very unstable. This forced us to write our own watching mechanism by periodically listing the server and manually keeping track of the differences.

Informer

The informer lists and watches the client to stay informed about changes that happen to workflows and jobs. The informer will reflect these changes to an in-memory store. We used the informer as a fast cache compared to requests to the API server. The informer provides convenient event handlers for *created*, *updated*, and *deleted* events. The use of these event handlers is explained in the *manager* section.

The informer is a Kubernetes object that uses different components such as `Controller`, `DeltaFIFO`, and `Reflector` behind the scenes. Explaining these component is out of the scope of this report. We

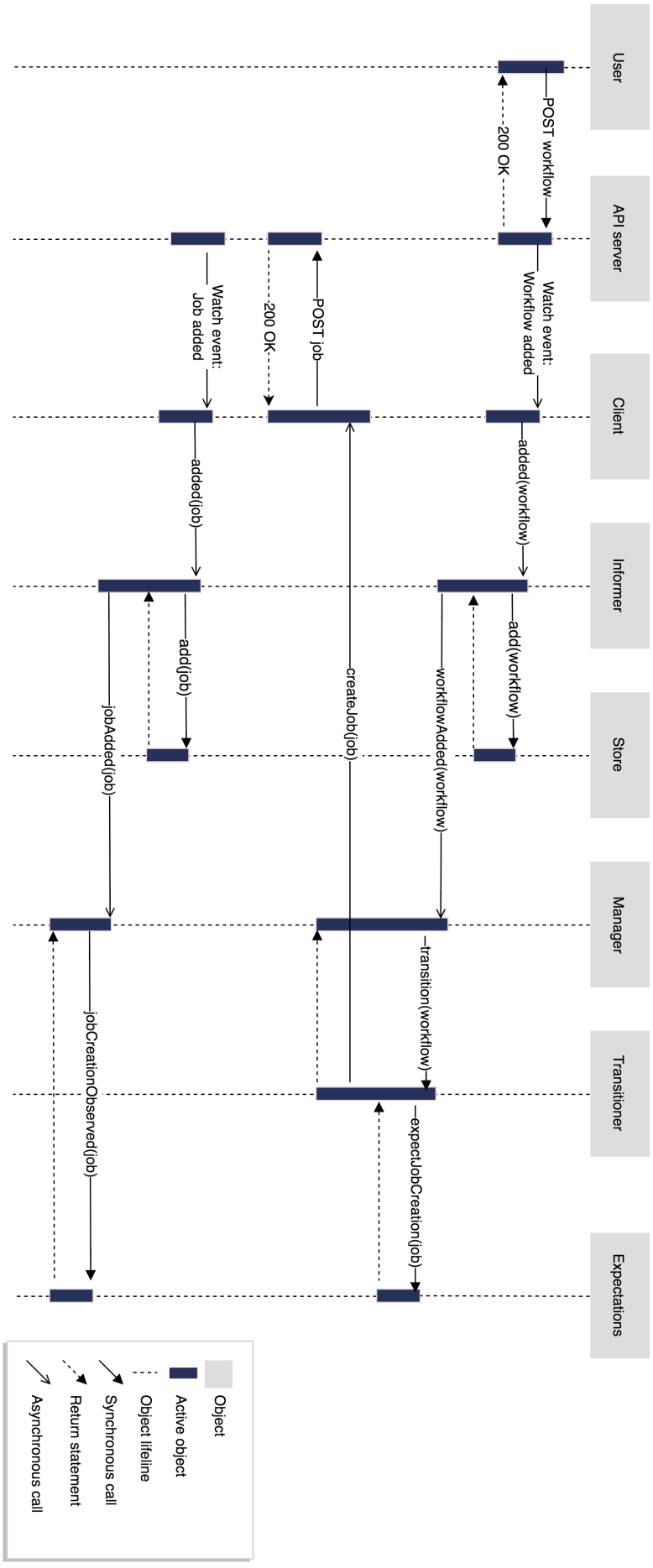


Figure 5.7: A sequence diagram of Workflow Manager. The diagram shows interaction between the user, the API server, and Workflow Manager.

```
{
  "type": "ADDED",
  "object": {
    "kind": "Job",
    "metadata": {
      "name": "test-job",
      ...
    },
    "spec": {
      ...
    },
    "status": {
      "conditions": [
        {
          "type": "Complete",
          "status": "True",
        }
      ]
    }
  }
}
```

Figure 5.8: An example of a watch event in JSON format.

did write a blog post about it, which can be found at `http://borismattijssen.github.io/articles/kubernetes-informers-controllers-reflectors-stores`.

Store

The in-memory store can be used to quickly list all workflows and jobs in the cluster. The only object that is allowed to write to the store is the informer. Other components have read-only access. Using this strategy, the store always reflects the state of the API server.

Manager

The manager is responsible for delegating the processing of workflows. It contains a list of workflows that need processing in a thread safe queue called the *workerqueue*. On creation the manager starts multiple threads, called *workers*. Each worker constantly checks the workerqueue to see if there are workflows that need processing. When a worker receives a workflow from the workerqueue, it is handed over to the *transitioner* for processing.

The manager uses the informer's event handlers to stay updated when a workflow or job is created, updated, or deleted. The manager has a *workflow informer* and a *job informer*, because it needs to stay informed about both workflows and jobs. Whenever one of the event handlers of the workflow informer is called, the corresponding workflow will be added to the workerqueue for processing.

The sequence diagram in Figure 5.9 shows the flow of actions when a *workflow* is created. When created the manager creates the workerqueue and multiple workers. For the sake of simplicity we only show one worker in the diagram. The workers will try to pop a workflow from the queue, by calling its blocking `pop()` function. When a *workflow* is created, the informer calls the `addWorkflow` event handler in the manager. The `addWorkflow` function will push the *workflow* onto the workerqueue. The workerqueue can now respond to the pop call from the worker, by returning the created *workflow*. The worker will then delegate the processing of the workflow to the transitioner.

When one of the event handlers in the *job informer* are called, a similar flow of actions take place. Since the workerqueue only contains workflows, the workflow corresponding to the job should be identified first. When a workflow is created, it is assigned a unique identifier (UID). When a job belonging to a workflow is created, it is assigned a `nerdalize.com/workflow-uid` label containing the corresponding workflow UID. When an event handler in the job informer is called, the `nerdalize.com/workflow-uid` label can be used

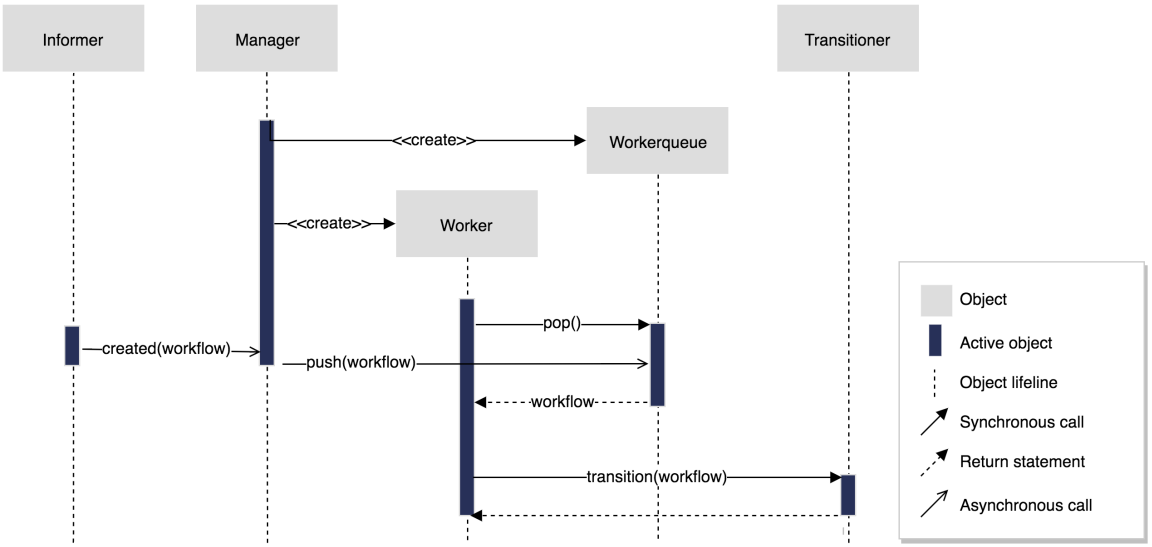


Figure 5.9: Sequence diagram of the manager and the workers.

to find out which workflow the job belongs to. That workflow is then pushed onto the workerqueue for processing.

Transitioner

The transitioner is responsible for transitioning a workflow from its current state to its desired state. When provided with a workflow, the transitioner will check if the preconditions of the workflow are met. The preconditions ensure that the workflow is (i) a valid workflow, (ii) not yet finished, and (iii) has not passed a deadline specified by the user in the specification.

If the preconditions are met, the transitioner will continue processing. It processes each step whose dependencies have been resolved. Two distinguishable actions can be taken for such a step. If no job exists for a the step, the job will be created using the client. When a job is created, its existence will be noticed by the Kubernetes scheduler. The scheduler will make sure that the job executes on a suitable machine. If a job does exist for the step, the transitioner will obtain the status of the job. The status of a job can be one of *active*, *succeeded*, or *failed*. The obtained status will be included in the status of the workflow.

When all the workflow steps have finished execution, the workflow is assigned a `WorkflowCompleted` status condition. The client is used to update the status of the workflow on the API server. The locally updated workflow may contain outdated information compared to the information present at the API server. In this case when the client tries to update the API server will reject it. The transitioner then indicates to the manager that the workflow should be requeued, such that the whole transition can be retried.

Expectations

Communication with the API server can take a long time in case the API server runs on a different machine and the network is congested. As a result ofthis delay, the in-memory store always lags the current state of resources in the API server. This could have problematic consequences, such as the transitioner creating a job twice. This can happen because creation of the first job is not directly reflected in the store. To prevent problems like these from happening we used *expectations*.

Explaining how expectations work can best be done using the sequence diagram in Figure 5.10. The diagram shows three periods during which the manager is active, indicated by the black bullets. In the first period the transitioner checks if the expectations for *job* are satisfied. The result is true, so *job* gets created. The transitioner tells the *expectations* object to expect *job* to be created. When the second period starts, the creation of *job* is not yet reflected in the store. This is due to the network delay indicated on the *Kubernetes API server* object. Since *job* is not yet present in the store, the transitioner decides to create it. Before creation the transitioner first checks the *expectations* and finds out that the expectations for *job* are not satisfied. For this reason *job* is not created again. In the thrid period the response from the API server finally arrives. This causes the manager to get notified. The manager notifies *expectations* that it observed the creation of *job*. The notification from the manager in reality goes through the informer, which is omitted for the sake of clarity.

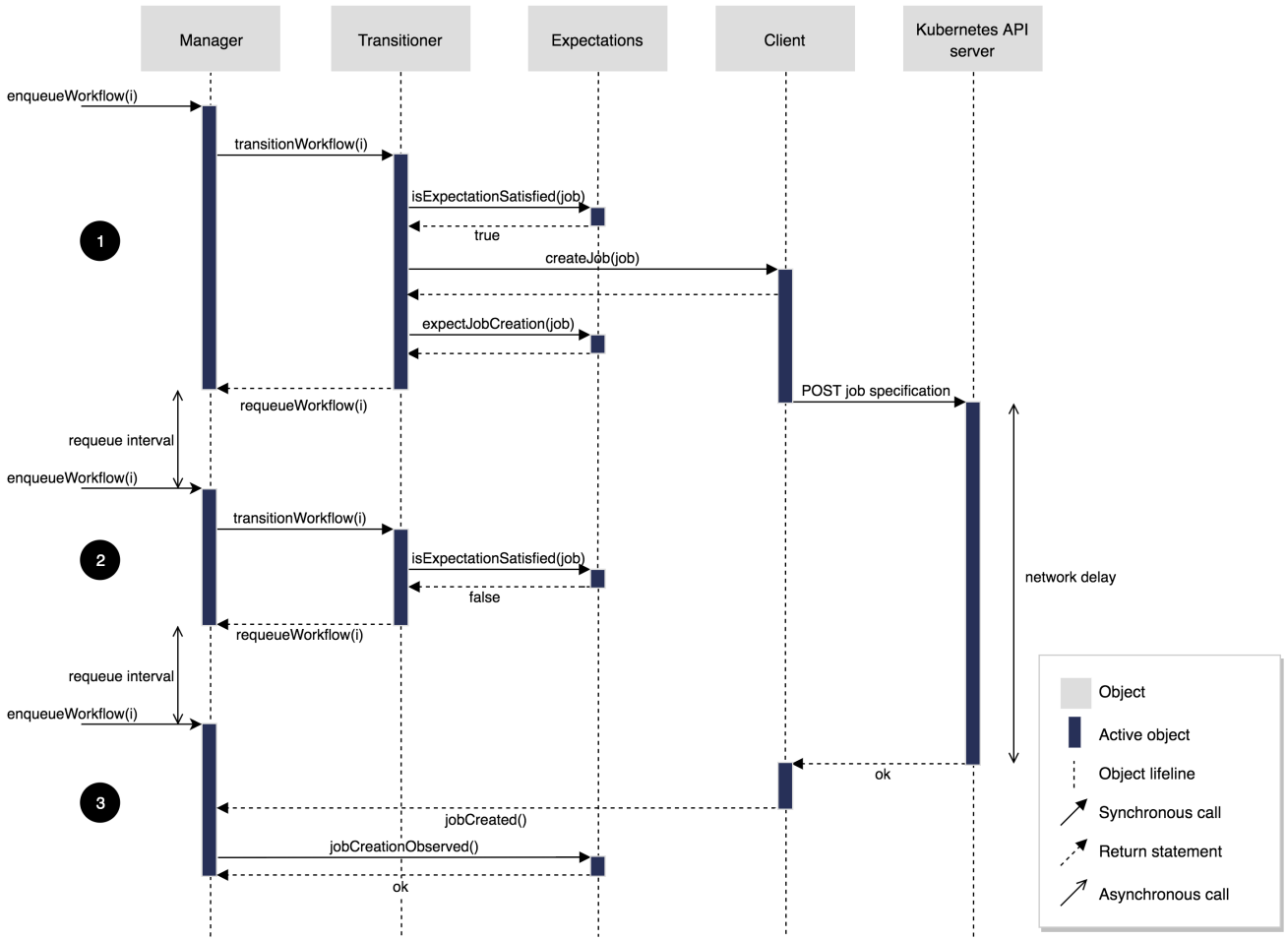


Figure 5.10: Sequence diagram, showing the added value of using expectations. The black bullets are used to indicate three distinct events.

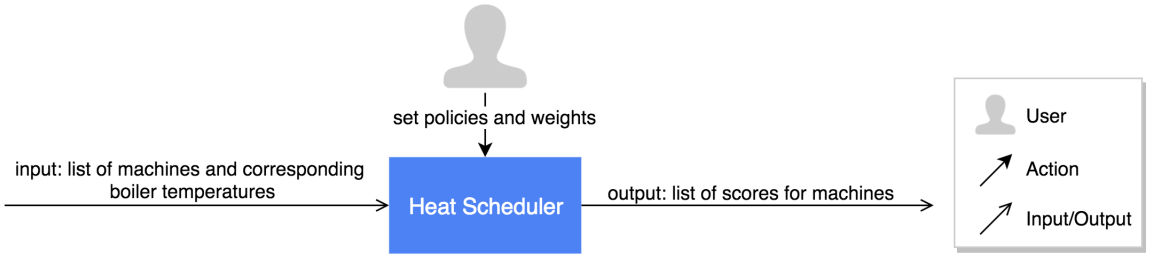


Figure 5.11: Overview of the heat scheduler. The user specifies which policies and the weight of these policies. The scheduler calculates a score for each machine based on the policies.

Eventbroadcaster

We used an *event broadcaster* to log creation of jobs. When a job is created the event broadcaster logs the event to the API server. The user is able to access these logs using the Kubernetes command line interface to get a chronological history of events.

5.3. The Heat Scheduler component

Heat Scheduler is a subsystem of Flower that can schedule steps based on selected scheduling policies. It is responsible for scheduling workflow steps on a CloudBox whose boiler has not yet reached the maximum temperature to minimize wasted heat. To make this scheduling decision Heat Scheduler must know the temperatures of all CloudBoxes in the cluster. The temperature is provided by a temperature sensor.

5.3.1. Architecture

Heat Scheduler is used to select the most favourable machine for a step to run on. Heat Scheduler assigns a score to each machine based on a list of policies and a given weight for each policy. Each policy can be represented by a priority function, which assigns a score between 0 and 10 to each machine. The score of each priority function is multiplied by a user specified weight. All products add up to the final score of a machine. A machine with the highest score is the most favourable machine. For example, suppose there are two priority functions, `priorityFunc1` and `priorityFunc2` with weighting factors `weight1` and `weight2` respectively, the final score of some machine *m* is: `(weight1 × priorityFunc1(m)) + (weight2 × priorityFunc2(m))`.

In the following sections we present four scheduling policies. Each policy expects a list of machines at its input and returns a list of scores corresponding to these machines. Each machine contains information about the current, requested, and average temperature of its corresponding boiler and information about the outside temperature of its physical location. An input-output diagram is shown in Figure 5.11. The user can specify which scheduling policies should be used and what their corresponding weight factors are, as can be seen from the diagram.

5.3.2. Scheduling policy: Minimum temperature

Intuition

This policy favors machines connected to a boiler whose water temperature is the lowest of all boilers.

Technical details

This scheduling policy assigns a score to each machine corresponding to its boiler temperature. The score for each machine is calculated by taking the ratio of the minimum boiler temperature of all boilers to the boiler temperature corresponding to the machine. In other words, the lower the temperature of a boiler, the higher the score of its corresponding machines. The algorithm for this policy is presented in Algorithm 1 in pseudo-code.

5.3.3. Scheduling policy: User request

Intuition

Users that consume heated water often desire to control the temperature of the water. An example is when the user wants to control the temperature of the central heater which is connected to a boiler. This policy favors machines connected to a boiler of which the difference between the current and the desired temperature is the largest.

Algorithm 1: Lowest temperature

Data: $M \leftarrow$ list of machines
$n \leftarrow$ number of machines
Result: $S \leftarrow$ scores for machines
begin
  $t_{min} \leftarrow minTemp(M)$ // get minimum temperature of all boilers
  for $i = 1...n$ **do**
    $t_{machine} \leftarrow temp(M_i)$ // get temperature of the boiler corresponding to M_i
    $S_i \leftarrow (t_{min}/t_{machine}) \times 10$ // calculate score for M_i

Technical details
The *user request* scheduling policy assigns a score to each machine corresponding to its boiler and its requested temperature. The height of the score for each machine is relative to the difference between the temperature of its boiler and its requested temperature. In other words, the greater the difference between the temperature of a boiler and its requested temperature, the higher the score of its corresponding machines. The algorithm for this policy is presented in Algorithm 2 in pseudo-code.

Algorithm 2: User request

Data: $M \leftarrow$ list of machines
$n \leftarrow$ number of machines
Result: $S \leftarrow$ scores for machines
begin
  // get maximum difference between current and requested temperature of all boilers
  $d_{max} \leftarrow maxDiff(M)$
  for $i = 1...n$ **do**
    $t_{requested} \leftarrow req(M_i)$ // get requested temperature of the boiler corresponding to M_i
    $t_{current} \leftarrow temp(M_i)$ // get current temperature of the boiler corresponding to M_i
    $d_{machine} \leftarrow t_{requested} - t_{current}$
    if $d_{machine} > 0$ **then**
      $S_i \leftarrow (d_{machine}/d_{max}) \times 10$ // calculate score for M_i
    else
      $S_i \leftarrow 0$ // do not give negative score to M_i

5.3.4. Scheduling policy: Usage profile

Intuition
This scheduling policy uses historical data to make predictions about the future. The policy favors machines connected to a boiler with the lowest water temperature, averaged over the past week.

Technical details
This scheduling policy assigns a score to each machine corresponding to its boilers average temperature. A running average of the past week is provided as an input. The score for each machine is calculated by taking the ratio of the minimum average boiler temperature of all boilers to the average boiler temperature corresponding to the machine. In other words, the lower the average temperature of a boiler, the higher the score of its corresponding machines. The algorithm for this policy is presented in Algorithm 3 in pseudo-code.

5.3.5. Scheduling policy: Outside temperature

Intuition
This scheduling policy works by taking the weather forecast into account. The policy favors machines located at places where the outside temperature is lowest. By places we mean the geographical location of the machine. This policy can be used when the network of computers spans multiple climate zones. Colder areas are more likely to require heat, therefore scheduling should prefer these areas.

Algorithm 3: Usage profile

Data: $M \leftarrow$ list of machines
$n \leftarrow$ number of machines
Result: $S \leftarrow$ scores for machines
begin
  $t_{min} \leftarrow minAvgTemp(M)$ // get minimum average temperature of all boilers
  for $i = 1...n$ **do**
    $t_{machine} \leftarrow avgTemp(M_i)$ // get average temperature of the boiler corresponding to M_i
    $S_i \leftarrow (t_{min}/t_{machine}) \times 10$ // calculate score for M_i

Technical details
This scheduling policy assigns a score to each machine corresponding to the outside temperature. The score for each machine is calculated by taking the ratio of the minimum outside temperature of all machines to the outside temperature corresponding to the machine. In other words, the lower the outside temperature, the higher the score of the machines corresponding to that location. The algorithm for this policy is presented in Algorithm 4 in pseudo-code.

Algorithm 4: Outside temperature

Data: $M \leftarrow$ list of machines
$n \leftarrow$ number of machines
Result: $S \leftarrow$ scores for machines
begin
  // get minimum outside temperature corresponding to the machines
  $t_{min} \leftarrow minOutsideTemp(M)$
  for $i = 1...n$ **do**
    $t_{machine} \leftarrow outsideTemp(M_i)$ // get outside temperature corresponding to M_i
    $S_i \leftarrow (t_{min}/t_{machine}) \times 10$ // calculate score for M_i

5.3.6. Implementation

Heat Scheduler is exclusively used to create a list of scores based on a list of machines and can therefore not be used as a scheduler on its own. We decided to implement Heat Scheduler as an extension to the default scheduler of Kubernetes. The advantage of extending the Kubernetes scheduler are the default policies provide by Kubernetes. The Kubernetes scheduler checks a list of predicates [11] for each machine in the cluster, when provided a step. The predicate returns whether a particular machine satisfies a requirement to run a step on that machine. An examples of a predicate used by Kubernetes is `FitsResources`, which checks if the machine has enough resources (memory and CPU) to run the requested step.

After Kubernetes has gone over the predicates, the list of remaining machines is sent to the Heat Scheduler. Heat Scheduler returns the score for each machine per priority function. Kubernetes combines these scores with its own priority functions and schedules the step on the machine with the highest total score. An example of a Kubernetes priority function is `ImageLocalityPriority`, which assigns a higher score to a machine that already has the image of the container on its local disk. Kubernetes communicates with Heat Scheduler through its HTTP API using JSON encoding for data.

6

Evaluation of Flower

6.1. Quality assurance

To ensure the code we wrote during the project is of high quality we employed two different measures. During the project we had an independent party rate our code based on aspects such as maintainability and compliance to best practices. This evaluation is further described in Section 6.1.1. The second measure we took involved using automated tools to keep different metrics within a desirable range. Code metrics are further described in Section 6.1.2.

6.1.1. Software Improvement Group

The Software Improvement Group (SIG) is a company that evaluates code. SIG uses static analysis to evaluate the code, resulting in a score between 1 and 5 *stars*. Feedback from SIG can be taken into account to improve maintainability and scalability of the code. During the project our code will be evaluated by SIG twice. The first time we sent our code to SIG was on May 27th. We received feedback by June 3th, which gave us enough time to improve the code based on their findings. The second SIG evaluation is on June 17th. This is on the same day as the deadline of this report, which means we cannot reflect on the second feedback here.

The system scored 3.5/5 stars, which means the code scores above average on SIG's maintainability model. Areas of improvement were *Duplication* and *Unit Size*. The entire analysis of SIG can be found in Appendix D. Since SIG did not tell us to which components the feedback was related, we asked them for clarification. In their reply they suggested that we should focus only on *Unit Size* for the following functions: `transitionWorkflow()`, `process()`, and `processJobStep()` in `transitioner.go` and `onReady()` in `code.js`.

We addressed the feedback by breaking up the aforementioned functions into smaller functions with less responsibility. Table 6.1 shows the *lines of code* (LOC) and *effective lines of code* (eLOC) in each function before and after SIG feedback. Effective lines of code are the lines of code in a function excluding comments, whitespace and log statements. As can be seen from the table we have drastically reduced the sizes of the functions. We expect good results from the second SIG feedback, since we took the feedback into account for existing and newly added code.

6.1.2. Code metrics

To assure good code quality we took several measurements into account. We used the code coverage system Coveralls as described in Section 4.2.3. At the end of the project we had 69% lines of relevant code covered

	LOC			eLOC		
Function	before	after	% decrease	before	after	% decrease
`transitionWorkflow()`	49	34	144%	26	19	137%
`process()`	76	29	262%	53	17	312%
`processJobStep()`	60	22	273%	44	19	232%
`onReady()`	94	8	1175%	88	4	2200%

Table 6.1: Lines of code (LOC) and effective lines of code (eLOC) for functions, before and after SIG feedback.

	Language	Files	Code	Comment	Blank	Total
Workflow Manager	All	29	3499	655	381	4535
	Go	19	3029	620	346	3995
	XML	1	272	0	0	272
	Javascript	1	92	3	9	104
	Markdown	3	37	0	7	44
	Shell	2	36	15	13	64
	HTML	1	17	0	1	18
	CSS	1	9	0	1	10
	Make	1	7	17	4	28
Heat Scheduler	Total	15	738	145	133	1016
	Go	13	724	111	125	960
	Make	2	14	34	8	56

Table 6.2: Lines of code (LOC) in Flower components, grouped by programming language.

with tests. This is still 31% away from full line coverage. This can be explained by the fact that Go code is very verbose in error checking. Much of the error checking consists of checking circumstances that are very unlikely to happen, such as the kernel being out of file descriptors. We did not test every single one of the possible errors, since they are very unlikely to happen and hard to simulate. Line coverage could be further improved by adding more end-to-end tests. End-to-end tests can be used to test much of the untested code responsible for communicating with Kubernetes. We compared our line coverage to Kubernetes. Kubernetes is a good comparison, as there are a lot of similarities between Kubernetes code and our code. Kubernetes has 59% lines of relevant code covered. From this we may conclude that our code is well tested.

Table 6.2 shows the lines of code for Flower and Heat Scheduler respectively. From Table 6.2 we can extract Heat Scheduler has a *comment-to-code ratio* of 20.4% for the Go code. According to Caltech [4] this ratio means that we have *good* documented code. Heat Scheduler has a *comment-to-code ratio* of 15.3%, which is *average* documented code accroding to Caltech. We used less comments in Heat Scheduler than in Flower, because Heat Scheduler is simpler and the code is more self-explaining.

6.2. Experimental work

To validate our system we designed an experiment. The experiment itself is described in detail in Section 6.2.1. The results of the experiment and the implications for stakeholders are covered in Section 6.2.2. Graphs are provided to visually summarize important aspects of the experiment.

6.2.1. Experiment

The experiment was carried out using machines provided by *Google Cloud Platform* (GCP). We chose GCP because Nerdalize uses it for development and we had been given free credit at the GCP conference. With this credit we were able to use about 50 machines for 10 hours. The specifications of the machines we used are shown in Table 6.3. The CPUs used by the GCP machines are very similar to the ones Nerdalize will eventually use in the CloudBox. This made them a good choice for the experiment.

Setup

GCP makes it very easy to create a Kubernetes cluster. Through a graphical user interface the type of machine as well as the desired amount are specified. With a click of a button the cluster is configured by registering every machine with the Kubernetes master. While this approach is significantly easier than configuring the cluster by hand it was not suitable for our experiment. Our implementation relies on a particular client that is only available in Kubernetes 1.3. At the time the latest Kubernetes version available on GCP was 1.2. We ended up modifying the shell script used by GCP to spin up the cluster. The script is available on the Kubernetes GitHub page[1]. After running the modified script we ended up with a Kubernetes cluster running version 1.3. The cluster was not visible on the online GCP dashboard, probably because we circumvented the setup page. All other functionality was untouched which made for an appropriate experiment environment.

With the cluster ready we created a pod containing Flower and a pod containing our custom scheduler.

[1] `https://github.com/kubernetes/kubernetes/tree/master/cluster/gce`

Property	Value
Type	n1-standard-2
Operating System	CentOS 6
vCPUs	2
Processor	2.6GHz Intel Xeon E5
Memory	7.5Gb

Table 6.3: GCP instance specifications

After registering the custom scheduler with the default Kubernetes scheduler the setup was complete. Arbitrary workflows could be submitted to the Kubernetes API server, Flower would then decide which steps are ready for scheduling and the custom scheduler finds a suitable machine to run the step's job on.

For the experiment we simulated a household consisting of four people. There is a CloudBox installed along with a boiler with 150 liter capacity. This setup simulates an average family with two children. Several other assumptions were made before the experiment took place:

Assumptions

1. The temperature of water entering the boiler is 15 degrees Celsius.
2. The maximum temperature of the boiler is 55 degrees Celsius.
3. The volumic capacity of the boiler is 150 liters.
4. The energy capacity of the boiler is 25.000.000 joules.
5. If boiler temperature is 15 degrees it is 'empty' and contains 0 joules worth of energy.
6. If boiler temperature is 55 degrees it is 'full' and contains 25.000.000 joules worth of energy.
7. Additional heat produced when the boiler is full is 'wasted'.

Assumptions were made in consultation with Nerdalize employees responsible for thermodynamics and hardware development. Assumption 3. was calculated as the energy needed to heat 150 liters of water from 15 to 55 degrees Celsius. The experiment consisted of running a test workflow twice, once using the default Kubernetes scheduler and once using the developed custom Heat Scheduler using the minimum temperature policy. We chose this policy because of its simplicity and ability to illustrate the point. Boiler temperatures are monitored as the workflow progresses. At the start of each run we reset the initial boiler temperatures to the same values. This made comparison possible. We used the Epigenomics workflow, which is one of the examples available on the Pegasus [31] website. The workflow is in a format used by Pegasus and is incompatible with Flower. Because creating a similar workflow by hand would take a long time we wrote a DAX to JSON converter instead[2]. It is able to convert an arbitrary workflow in DAX format and turn it into the JSON format recognized by Flower. The workflow consists of 997 steps and 1,234 dependencies. Each step contained a job which would run a stress [17] program. We configured the program to use a random percentage of available CPU for thirty seconds on a given machine. This simulates a CPU intensive computation being performed.

Simulation

The scenario which the experiment simulates is a cluster of CloudBoxes installed in a typical home consisting of four people, two adults and two children. It expects no warm water is used during the execution. At the start of the simulation it is 7PM, the children have just been washed and gone to bed. At the end of the simulation both parents will take a shower consuming most of the boiler's warm water. To simulate this situation initial boiler temperatures were chosen randomly from a normal distribution with a mean of 60 and a standard deviation of 20. These values were derived from water consumption data made available by the engineers at Nerdalize.

Because the GCP instances do not contain an actual boiler, we wrote several components to simulate a boiler. Initial boiler temperatures are assigned by the *Initializer*. During the experiment machine cpu

[2] `https://github.com/nov1n/kubernetes-workflow/tree/master/dax-to-json`

Endpoint	Method
`/setup?std={standard_deviation}&mean={mean}`	`GET`
`/reset`	`GET`

Table 6.4: A description of the Initializer's client facing API.

utilization was converted to corresponding generated heat by a *boiler simulator*. These two components are now described in more detail.

The Initializer runs a web service with a client facing API which is described in Table 6.4. When a request is made to the `/setup` endpoint the Initializer generates a *label* with the initial temperature value for every machine in the cluster. Labels in Kubernetes specify identifying attributes of objects that are meaningful and relevant to users. The label is randomly generated from a normal distribution of which the parameters may be configured by the user through `std` and `mean` in the query string. The label is assigned to a machine through the Kubernetes API server. The API server updates its internal representation of the particular machine to reflect the label assigned. Labels are well suitable in this case because Kubernetes components such as the scheduler may easily access them. This is important as our custom scheduler needs the temperature labels to make scheduling decisions.

The `/reset` endpoint resets the temperature label for every machine to the value previously calculated by `/setup`. Calling `/reset` before `/setup` is not allowed as previous values are not available. Reset functionality is useful to test how different scheduling policies affect the same initial boiler temperatures. Test results of running the same workflow with different scheduling policies can be found in Section 6.2.2.

Boiler Simulator The boiler simulator simulates a boiler attached to a machine. The boiler is not yet present because CloudBoxes are still under development. After the initial boiler temperature is set by the Initializer the boiler simulator keeps the temperature up-to-date corresponding to the heat generated by the machine. The simulator periodically collects cumulative cpu utilization from *Heapster*. Heapster is a kubernetes component that exposes various metrics through a RESTful API. CPU utilization is measured in *millicore seconds* which is equivalent to a thousandth of a core used for one second. For every machine the cpu utilization is used to compute the corresponding temperature increase. The conversion from cpu utilization to temperature increase is done by taking into account the heat generated per millicore second, which depends on the type of CPU present in the machine. For testing purposes it is convenient to increase the rate at which the boiler heats up. As a result we introduced a scaling factor which is multiplied by the result of the conversion before it is assigned as a label to a machine. Every time a temperature label is updated, the timestamp along with the temperature is also stored in a database to be used by the user interface.

Scaling factor
The boiler simulator used a scaling factor to 'speed up time'. Instead of using the actual joules/second generated by the CPU it used a value that made the short simulation representative for a much longer time in reality. The 7 minutes it took to complete the workflow represent approximately 1,36 hours in reality. The calculation can be found in Appendix E.

Hypothesis
The goal of the experiment is to measure how much heat is wasted. This waste occurs in two ways. The first way is when steps are run on CloudBoxes whose boilers are full. Additional heat in this case is wasted because the boiler has reached maximum temperature and the CloudBox vents warm air to the outside. The second type of waste occurs when more warm water is used than is available in the boiler. In this case the central heating system needs to produce the warm water not provided by the boiler. The first run uses the default Kubernetes scheduler which is unaware of boiler temperatures. The default scheduler uses a *round robin* scheduling strategy which means that a step, on average, has an equal probability of landing on a given node. Because not every step produces the same amount of heat, we expect some CloudBoxes will quickly fill up while others are still empty. The Heat Scheduler however is aware of boiler temperatures. Its scheduling strategy is to schedule steps on the coldest boiler. In this case all CloudBoxes supposedly will warm up evenly, minimizing wasted heat. We expect less heat to be wasted when the custom Heat Scheduler is used compared to the default scheduler, due to a more even distribution of heat.

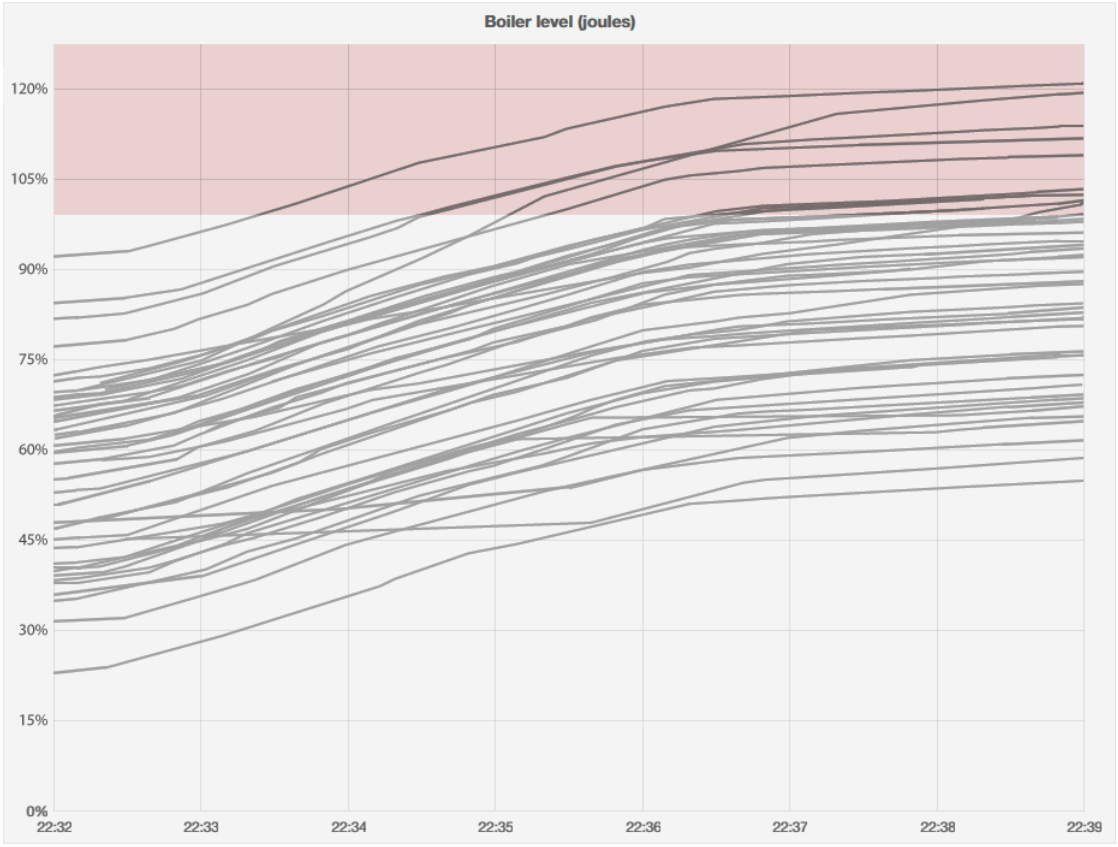


Figure 6.1: The boiler temperature during the execution of the test workflow using the default scheduler. Energy produced in the gray area (under 100%) is useful; energy produced otherwise is wasted.

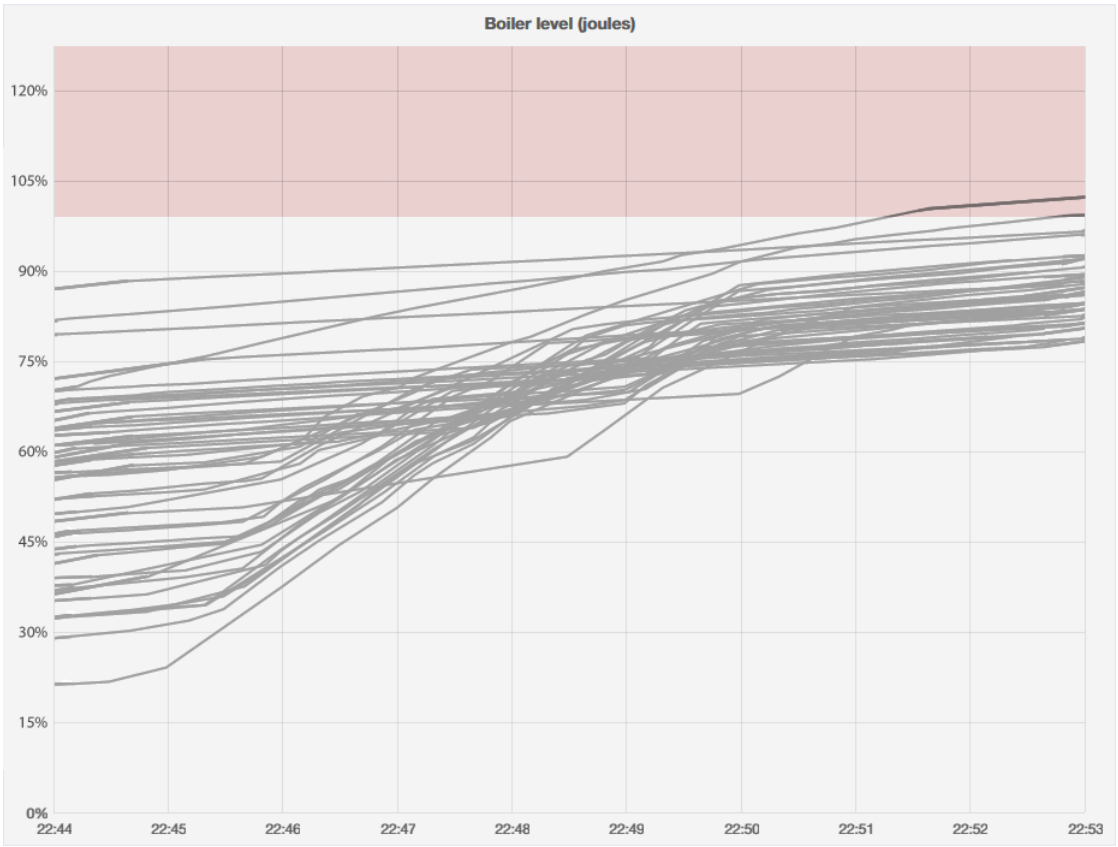


Figure 6.2: The boiler temperature during the execution of the test workflow using the custom Heat Scheduler. Energy produced in the gray area (under 100%) is useful; energy produced otherwise is wasted.

6.2.2. Results

In this section we will describe the results of running the experiment. We explain how the results were gathered, what they mean and how they relate to our expectations. Figure 6.1 and Figure 6.2 illustrate how the boiler temperature changes over time with the two different schedulers. Every series represents a CloudBox. On the vertical axis the boiler temperature is depicted as a percentage of the maximum temperature. The horizontal axis shows time. Background color represents whether energy is being wasted. While lines fall within the light area between 0% and 100%, energy is used to warm up the water in the boiler. The red (dark) area from 100% up signifies heat is wasted because the boiler has reached maximum temperature. Figure 6.1 corresponds to a workflow executed on the system using the default Kubernetes scheduler. As we expected lines are far apart meaning steps are assigned to CloudBoxes which are full while not yet full ones are available. Furthermore several lines enter the red (dark) area signifying heat waste. The Heat Scheduler was used in the second run corresponding to Figure 6.2. In this graph we see that the lines are much closer together and almost all stay within the light area. From this we may conclude that that given our assumptions the Heat Scheduler makes more efficient use of generated heat.

Bottom line

We will now compare the wasted heat during the execution of the experiment using the default scheduler versus using the custom Heat Scheduler. As previously described, at the end of our simulation each of the two parents will take a shower. An average shower consumes 9 liters per minute and has a duration of 12 minutes. Leading to 108 liters of water. Water used for showering will be 38 degrees Celsius. Water starts at 15 degrees Celsius and must be heated to 36 degrees Celsius. To warm 1 liter of water 1 degree Celsius 4180 joules worth of energy are needed. The total amount of joules needed for the two showers is calculated as follows:

$$9L/m \times 12m \times (36C - 15C) \times 4180J/L/C \times 2people = 18.960.480\,joules$$

Now we convert the amount of joules to the percentage of the total boiler capacity as follows:

$$\frac{18.458.880J}{25.000.000J} = 0,76 = 76\%$$

This means that all the CloudBoxes whose level was under 76% at the moment the showers took place, need to resort to the central heating system to provide for the additional warm water needed. In case the default scheduler is used, the central heating costs to provide the additional energy amount to at least €32,00 per family per year. In case the Heat Scheduler is used, all heat is provided by the boiler saving the full €32,00 per year. This is on top of the yearly savings of €300/year Nerdalize promises to homeowners using a CloudBox installed, leading to a significant 10% increase. For more details about the calculation see Appendix F. This saving only takes into account the scenario described above. Similar scenario's may occur during the day leading to more savings.

To verify the impact, other experiments are required using real CloudBoxes, real users and longer time spans. The simulation possibly differs from reality in some scenarios which may lead to different results. Using real users verifies if the assumptions made during the experiment regarding water consumption are realistic. Lastly longer experiments will illustrate whether the savings are consistent over time.

7

Discussion & Future Work

7.1. Discussion

Data centers are becoming increasingly important for our daily lives, supporting the ICT needs of our society. However, they raise important operational costs, and an energy footprint that raises ecological concerns. Instead of current data centers, an alternative for the future could be hosting servers in private homes, using the heating generated by servers for the benefit of the home owner, and the revenue from the server for the benefit of this distributed data center's operator. Much research and engineering needs to happen before this approach is proven viable at the scale of our society's data center needs. In this work, we take an important step in the right direction, by designing and implementing Flower, a workflow manager for such distributed data centers that can also take heating into its scheduling decisions. In this section we will discuss the main challenges encountered in Section 7.1.1, comment on the experiences using the Go programming language in Section 7.1.2, and describe the changed requirements in Section 7.1.3.

7.1.1. Main challenges

During the project we have encountered several challenges. This section is used to describe the challenges. We will describe the challenges working with the open source community, explain the pitfalls when relying on alpha functionality, and discuss the difficulties of writing scalable and concurrent applications.

Open-source community

At the start of the project, the plan was to to include Flower in Kubernetes source code, such that we could contribute to the community. We had noticed that a Professional from France had already written a proposal for a workflow management system for Kubernetes. We got in contact to see if we could collaborate on the implementation of the proposal because it covered most of our requirements.

Some code had been written as a work-in-progress (WIP). Just before the end of the research phase, the WIP was rejected by Kubernetes. Kubernetes team members argued that adding workflows would make Kubernetes overly complicated. They suggested implementing workflows as a third party project. Luckily we had not yet started implementing, but we had to redesign our initial architecture because the solution now had to run outside of Kubernetes.

After a week we found out that a large part of the code was reusable. We suggested working together with multiple times, but ended up not. This was mainly due to company policies on his side and time constraints on our side. We did however use this code as a starting point for Flower.

The code looked very extensive. We however soon found out that the code contained a lot of bugs. The code had probably never been run, not even for testing purposes. We invested a lot of time refactoring the code and making it operational. We later discovered that the code was very similar to existing Kubernetes code. Looking back at the process we should have used the Kubernetes code as a starting point, because it was better documented.

Kubernetes alpha

Kubernetes team members advised us to implement Flower as a third party project, using their *third party resource API*. There are two main advantages of using the third party resource API: (i) Kubernetes takes care

of storing data in a distributed manner and (ii) it looks like Flower is integrated into Kubernetes to end-users. It should however be noted that the third party resource API is still in alpha, which means it is very unstable. We have encountered the situation in which certain bugs were fixed, but other bugs were introduced at the same time. Working with unstable code like this was time consuming but at the same time very educational. We had to engage with the open source community to address issues, request fixes and test new revisions.

Scalability & Concurrency
Flower should run in a distributed environment and must be able to handle large workloads. These are two requirements that introduced challenges. Because Flower had to run on large scale, we used parallel processing in multiple places. Parallel processing makes the throughput of the program higher, but also makes it harder to debug. We have encountered bugs which took us entire days to solve, because of the difficult nature of concurrent problems.

Designing for distributed systems turned out to be hard. It requires other design patterns than those used in non-distributed applications. This is because (i) communication can last unexpectedly long and (ii) the program can more easily reach unexpected states. We ended up with a robust system, using design patterns such as *expectations* described in Section 7.1.1.

7.1.2. Programming language experiences
Most of the Flower codebase is written in Go. Go is a very young programming language developed by Google, dating back to only 2012. In this section we would like to reflect on Go to give some insight in this new language.

Writing Go is similar to writing C. Although its syntax is similar syntax to C, it has more advanced features such as garbage collection, dependency management, and built-in concurrency control. Go has no classes, which might seems odd for a modern language focused on scalability. To cope with the need for structured code, Go uses *structs*, *interfaces* and *methods*.

We liked working with Go. It is very easy to create scalable concurrent programs in Go because of the ease of creating threads, called *goroutines*. Inter-thread communication is convenient using *channels*. We had to get used to working with structs and interfaces instead of classes at first, but soon these concepts made perfect sense. A feature we did not like as much is the verbose error checking of Go. Go does not have exceptions, but uses multiple returns values of which the last one is often an *error*. Multiple return values are an interesting feature of Go, but it means that error checking has to be done in the middle of pieces of code.

7.1.3. Changing requirements
The difference between the list of requirements in Section 3.4 and those in the research report, is the *scheduler* section. Initially Nerdalize wanted the system to schedule workflow steps onto either the same machine or on a separate machine. This requirement was introduced by Nerdalize, because steps can share the same disk when scheduled on the same machine. During the project Nerdalize changed their approach to this problem. The idea is to place a network disk close to the machines in the cluster. This way steps do not have to run on the same machine to share data. The requirement to run steps on separate machines was only introduced because of its similarities with the aforementioned requirement. Because of the new approach, both requirements were dropped.

Nerdalize introduced a new requirement to replace the old ones. Nerdalize requested a scheduling policy that is able to schedule steps on the machine with the lowest boiler temperature. This scheduling policy is convenient for them and for the environment because it wastes less heat. The main problem with this requirement was that no boilers were yet deployed. We therefore had to mimic the boiler by measuring CPU usage and translating this to heat produced. The Heat Scheduler is described in more detail in Section 5.3

7.2. Future Work
In this section we will discuss next steps for the Flower project. Valuable additional functionality can be found in Section 7.2.1. Future testing strategies that should be used to make Flower production ready are described in Section 7.2.2. The idea of contributing Flower to the open source community is explained in Section 7.2.3

7.2.1. Additional functionality
Flower could be extended to provide more functionality to the user. In this section we discuss the possible extensions of Flower. Most of these extensions result from unimplemented *must-not-have* and *might-have*

requirements from Section 3.4 and brainstrom sessions with Nerdalize employees and other professionals.

Resource usage limiting of workflows
Flower is able to execute a given workflow, but does not yet have a general solution to resource limiting. With resource limiting we mean the restriction of CPU and memory usage for a given workflow. Kubernetes supports CPU and memory to be limited for a given pod. In Flower workflow steps run as pods. Therefore we can currently limit the resource usage of the individual steps, but not for the workflow as a whole. Limiting the workflow as a whole is beneficial in certain cases where it does not matter how much resources the individual steps consume. This could for example be used by Nerdalize to bill customers per workflow.

Scheduled workflows
Scheduled workflows are workflows that are assigned a (periodic) starting time. Specifying a periodic starting time makes sense in cases where workflows need to run on a daily/hourly basis. Examples of such workflows are periodically generated business reports. To implement this functionality the *CRON expression* syntax could be used, because many people are familiar with its usage in *crontab* files.

Nested workflows
Flower could be extended to support nested workflows. These are workflows nested into other workflows, much like modules or packages to a programming language. The usage of nested workflows allows for encapsulation of a series of processes that can be easily shared and integrated into other workflows. Flower is designed to deal with nested workflows, however the code to execute a nested workflow does not yet exist.

Additional scheduling policies
The scheduling policies presented in Section 5.3 are the first of its kind. To the best of our knowledge workflow scheduling policies that take heat dissipation into account do no yet exist. In this report three scheduling policies are presented. We foresee that more scheduling policies will be added in the future. These can be both policies regarding heat dissipation and other policies. Examples of other policies are scheduling based on economically favorable machines. Such a policy would for example take the price of electricity into account. Other policies could take network delay or data transfer cost into account.

7.2.2. Testing in production
The design and implementation described in Chapter 5 and the experimental work discussed in Section 6.2 describe the current state of Flower well. Flower has proven to be a working system, being able to handle large workloads and deliver scheduling capabilities. It should however be noted that Flower is a proof of concept system and has not yet run in production environments. In this section we describe two ways of testing Flower to make it more suitable to run in production.

Testing real workflows
The experiment described in Section 6.2 was conducted by running a test workflow. For this test workflow we took the dependency graph of the well-known workflow, Epigenomics. The steps in the workflow were simple *stress* programs, that utilized the CPU for a given percentage. As future work we recommend Flower to be tested with real-world workflows. An example of such a workflow could be the real Epigenomics workflow.

Testing with real boilers
Flower currently relies on simulated temperatures of the water inside the boilers, as explained in Section 6.2. Flower should be tested with real CloudBoxes and real boilers once they are being deployed. Testing could be done as a simulation first, with data that is gathered from the real CloudBoxes. If the simulation shows expected results, the scheduling policies can be deployed to run in production.

7.2.3. Contributing Flower to the open source community
Flower has been designed to work with Kubernetes from the start. The initial idea was to get Flower in Kubernetes source code, as described in Section 7.1.1. After we decided to move Flower to a third party application, we still kept the possibility in mind to contribute back to the community. Kubernetes has a separate repository for contribution projects called *contrib*. The idea is to place Flower in this repository. Besides showing a sign of esteem it also has benefits for the Flower project. There will be an entire community maintaining and extending the Flower source code.

8

Summary & Conclusion

8.1. Summary

In this thesis we have presented Flower, a workflow management system in a containerized environment with heat-aware scheduling capabilities. In particular we have covered the processes used to create Flower, the design and implementation of Flower, the various heat-aware scheduling policies, and the experimental work to show the impact of such policies.

The development of Flower started with two weeks of research. During these weeks we interviewed both Nerdalize employees and professionals from various fields. Besides interviews a literature study was conducted to gather knowledge about related work and workflow management systems in general. The 7 weeks after the research phase were used to design and implement Flower. To manage product development, we made use of the agile development methodology Scrum. Further assistance was provided by multiple tools, such as Travis CI for continuous integration and GitHub for team communication. During these weeks we have had extensive conversations with employees from Nerdalize and professionals from companies including Google, RedHat and CoreOS.

The design of Flower has led to a fully functional workflow management system with heat-aware scheduling capabilities. To the best of our knowledge the notion of heat-aware scheduling has never been introduced to the scientific world before. Flower executes workflows, represented by a directed acyclic graph, on a set of interconnected machines, or cloud. As a proof of concept we implemented Flower on top of Kubernetes. Kubernetes manages deployment of containerized applications on a cluster of machines. Each workflow step is scheduled and executed on the most suitable machine. The most suitable machine is selected according to scheduling policies that we created, taking heat dissipation into account. Heat-aware scheduling is an important feature for Nerdalize, because its business revolves around making optimal use of heat dissipation by computers.

To verify the behaviour of Flower and the added value of its scheduling policies, we conducted an experiment. The experiment shows a reduction of wasted heat when a workflow of 997 steps is run on 50 machines. The results of this experiment are show in Figure 8.1. The boiler temperature exceeds 100% for certain machines in the bottom graph. From the figure we can conclude that less energy is wasted when a heat-aware scheduling policy is used. From the experiment we conclude a minimal saving of €32 per family per year. This is an additional 10% to the approximately €300 saving Nerdalize brings a family annually.

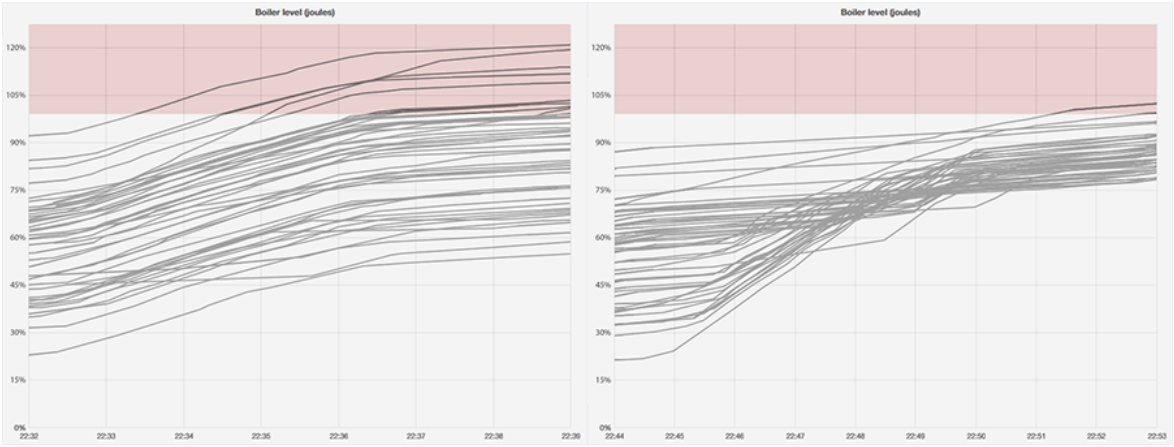


Figure 8.1: The boiler temperature during the execution of the test workflow using the default scheduler (left) and the custom Heat Scheduler (right).

8.2. Conclusion

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

1. *"How can workflows be executed in a distributed containerized environment?"*
   To execute workflows in a distributed containerized environment, we designed and implemented Flower, consisting of a Workflow Manager. Workflow Manager ensures that workflow steps, whose dependencies have been resolved, are placed on a machine in a cluster. Actual placement of these steps is outsourced to Kubernetes, a system used to orchestrate containerized applications in a cluster.

2. *"How can workflows be scheduled to make optimal use of generated heat?"*
   Flower has several scheduling policies to make optimal use of generated heat. Each policy expects a list of machines at its input and returns a list of scores corresponding to these machines. Each machine contains information about the current, requested, and average temperature of its corresponding boiler and information about the outside temperature of its physical location. The machine with the highest score is used to execute a part of the workflow.

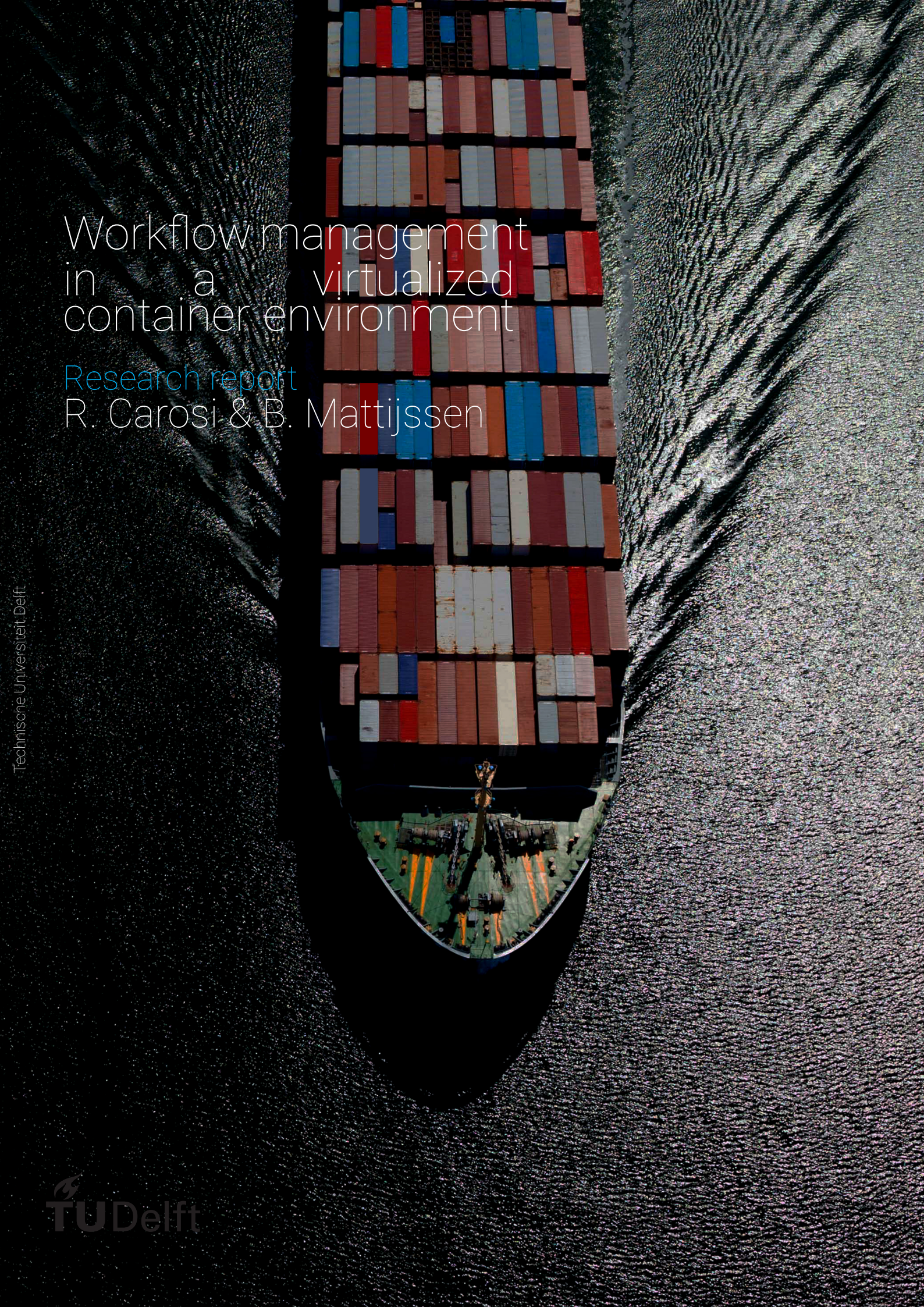
In Section 3.4 we presented a list of requirements for Flower. Table 8.1 shows the requirements and indicates which requirements are met. Requirements are prioritized using colored boxes representing categories as defined by the MoSCoW[14] method. The green box (🟩) stands for must have, the half full orange box (🟧) stands for could have and the almost empty red box (🟥) stands for won't have.

#	Priority	Requirement met	Description
1	🟩	yes	Flower executes a workflow according to a formal definition (Section 5.2.1).
2	🟩	yes	Flower monitors workflows to ensure execution and provides the user with a monitoring tool (Section 5.2.1).
3	🟥	no	Flower is not responsible for providing data, because this is the responsibility of the user.
4	🟩	partially	Step execution may depend on the completed status of other steps (Section 5.2.2).
5	🟥	no	Flower does not feature nested workflows (Section 7.2.1).
6	🟧	yes	Steps can be added or removed during workflow execution (Section 5.2.2).
7	🟩	yes	Workflow may run on a single machine or a cluster of machines (Section 5.1).
8	🟩	yes	Workflow may be cancelled during execution (Section 5.2.2).
9	🟧	no	Flower does not feature resource limiting for workflows (Section 7.2.1).
10	🟩	yes	Flower offers users the ability to customize the scheduler by adding policies and specifying weights (Section 5.3).
11	🟩	yes	Flower features a scheduling policy that takes minimum boiler temperature into account (Section 5.3.2).

Table 8.1: Flower requirements and indications which requirements are met.

A

Research Report



Workflow management in a virtualized container environment

Research report

R. Carosi & B. Mattijssen

Technische Universiteit Delft

TU Delft

Contents

1 Introduction

Nerdalize heats houses with computing power by packaging modern servers into a heating system and installing it into houses. Nerdalize makes money by selling server time as a cloud provider for CPU intensive applications. Their vision is to become a container-as-a-service (CaaS) [26] provider running on a distributed network of computer powered heaters. Their current focus is on CPU intensive applications such as scientific computations.

Scientific computations are often done in discrete steps such as analysis, simulation, management, and visualisation of data [18]. These steps may have dependencies between them to ensure steps only execute when others are finished. The abstract representation of steps and dependencies is called a workflow. Much research has been done in the field of workflow management systems [21, 24, 27, 32] in the past but no suitable solutions exist for managing workflows in a containerized environment.

A solution for managing workflows in a containerized environment is proposed in this report. The report is structured as follows: chapter 2 provides the necessary background regarding workflows and container related technologies, chapter 3 lists the requirements of the system, chapter 4 describes our approach for developing the system, and chapter 5 shows the preliminary planning of the project.

2 Background

This section provides background information required to understand the system. Section section 2.1 describes the technologies used. In particular workflow management, containers and Kubernetes. Section 2.2 contains the problem description as provided by Nerdalize. Finally related work can be found in section 2.3.

2.1. Technologies

This section provides information about the different technologies used throughout the report. Three different technologies are described as follows: subsection 2.1.1 gives an introduction to workflows and workflow management systems, subsection 2.1.2 introduces the concept of containers, and subsection 2.1.3 explains the Kubernetes system.

2.1.1. Workflow Management

Many companies and scientists need to process a lot of data [23]. To support the increasing demand of computing power needed to processes this data, many computers are interconnected forming a grid or a cloud. Data processing is often done in discrete steps, where each step depends on the output of previous steps in the system. In the scientific world a common sequence of steps is analysis, simulation, data management, and visualisation [18]. The abstract representation of steps and dependencies is called a workflow. Workflows can be modelled as directed acyclic graphs (DAGs), in which tasks are nodes and edges represent dependencies between tasks. Using DAGs as a model allows for the use of extensive scientific research on graphs [30, 31] to optimize performance [29], reliability [22], and scalability [20] of the workflow. An example of a DAG is shown in Figure 2.1. The figure shows six steps and various dependencies. Step `B` can only start executing when step `A` finishes execution. Steps `C`, `D`, and `E` can each start in parallel when step `B` finishes.

Workflow management systems (WMS) have been around since the start of grid computing [21, 24, 27, 32]. According to Taverna [15] a WMS is "a piece of software that provides an infrastructure to setup, execute, and monitor scientific workflows". In other words a WMS schedules tasks on appropriate machines and monitors their execution and dependencies, so users can focus on describing the workflow. An example of a workflow management tool is Pegasus [24] which lets the user describe their workflow in XML format. Pegasus maps tasks to available machines and transfers necessary data required for execution. Other systems like Triana [21] provide users with a visual tool to create workflows and do monitoring. WMSs often contain functionality to optimize workflow performance, scalability and reliability. Section 2.3 will describe some workflow management systems in more detail.

2.1.2. Containers

A *container* is an operating-system level virtualization method that provides a completely isolated environment, simulating a closed system running on a single host. It gives the user the ability to have an environment to do whatever is needed; in particular run applications with the necessary resources and environment configuration [25].

A container runs multiple processes in an isolated environment. Each process has a *process id* which is unique within the container itself. Processes outside the container are not visible inside the container.

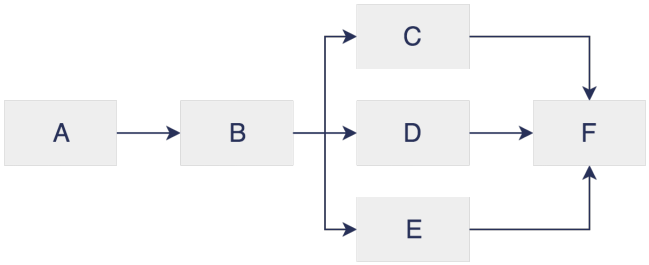


Figure 2.1: An example of a workflow presented as a DAG.

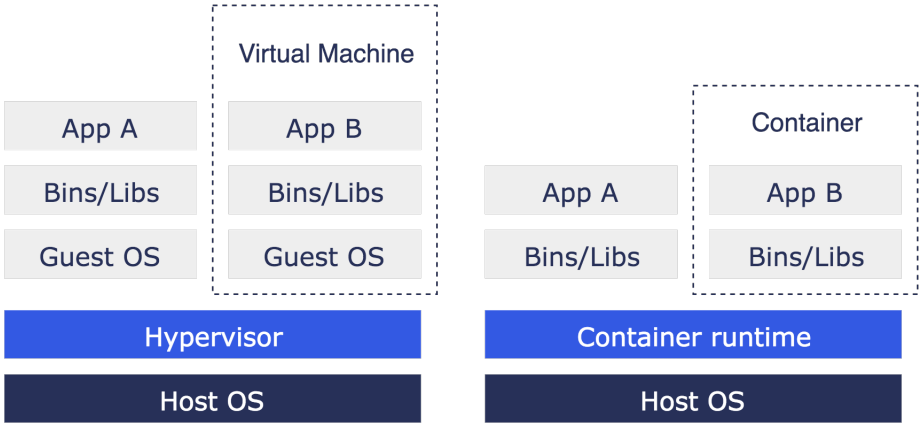


Figure 2.2: Comparison of virtual machines (left) and containers (right). Based on a figure from the Docker documentation (`https://www.docker.com/what-docker#/compare-block`).

This also applies to other resources such as networking, disk and users. Isolation is provided by *cgroups* and *namespaces* as Linux kernel modules.

When describing containers, virtual machines (VMs) are often used as an analogy. Containers and VMs are similar in that both are virtualized environments that run on a host machine. The differences are shown in Figure 2.2, showing the software stacks used by VMs and containers. The figure shows VMs have their own *operating system* (OS) called the *guest OS* which is separated from the host OS. Containers on the other hand share the kernel with the host OS. Using containers rather than VMs provides two major advantages: their size is smaller (MBs vs. GBs) and booting is faster (seconds vs. minutes).

In recent years containers have increased in popularity. This increase in popularity can be attributed to the release of the Docker software in 2013, which makes it easy to build, distribute and run containers [6].

2.1.3. Kubernetes

Kubernetes is an open-source project by Google for orchestrating containers on multiple hosts [11]. The main features of Kubernetes are scheduling and replication control. The scheduler assigns pods to an available host. A pod is the atomic unit in Kubernetes; it contains one or more tightly coupled containers. Replication control can be used to ensure that a specified amount of copies exist. The replication controller monitors pods and creates new instances in case of pod failure. Kubernetes consists of one master node and multiple worker nodes, shown in 2.3. The master node runs the scheduler and replication controller and exposes an API for communication with users and worker nodes. Each worker node has a *kubelet* daemon communicating with the master node through the API. Worker nodes are responsible for running pods on a host.

Throughout this report we will use the terms *job* and *label*. A job contains one or more pods. It terminates with a `success` status when all of its containing pods succeeded. Labels allow selecting an entity in the system based using specific selectors. Among these entities are jobs and pods.

2.2. Problem description

This section describes the problem we are trying to solve. In section 2.2.1 the reader finds the original problem description as provided by Nerdalize. section 2.2.2 we provide a modified version of the problem statement

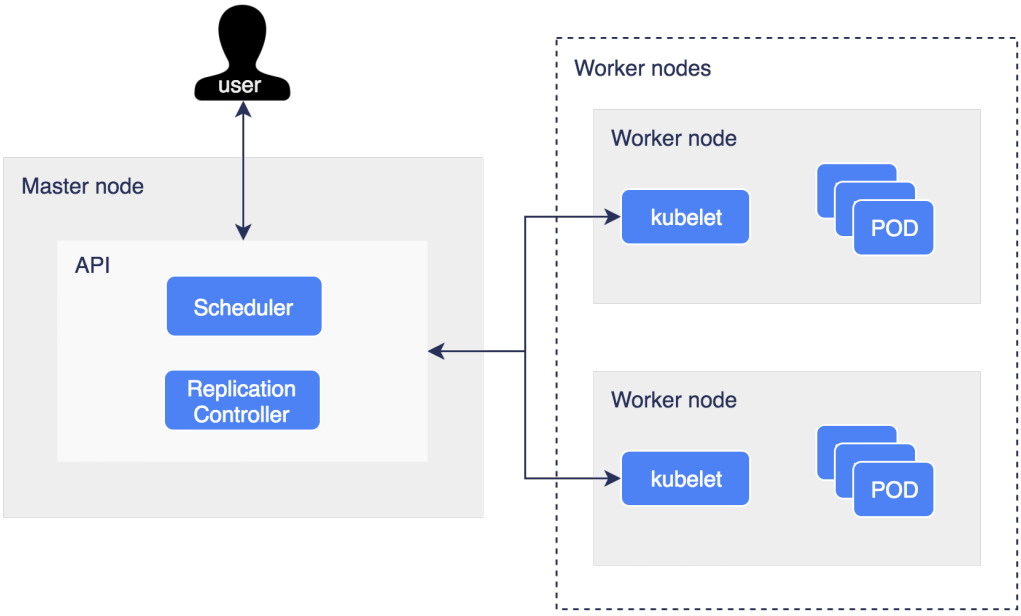


Figure 2.3: Kubernetes architecture showing master node (left) and worker (right) nodes. Based on a figure from the Kubernetes documentation (`https://github.com/kubernetes/kubernetes/blob/master/docs/design/architecture.md`).

to suit both the academic and the business requirements.

2.2.1. Original problem description

To achieve optimal cluster utilization and ease of use for its customers Nerdalize is building its main product the Nerdalize Cloud Engine (NCE), based on modern container orchestration technology. After extensive research Nerdalize has chosen to use the open source Kubernetes system as the basis for the NCE. In line with its business goals the first step is allowing clients to run high throughput workloads in a user friendly manner. Many workloads contain large parallelizable components. Running these on a cluster delivers a reduction in total workload throughput time. Modern cluster schedulers support complex workflows that allow non-trivial multi-step workloads to be executed automatically. Kubernetes has basic support for the concept of Job, but at the time of writing does have workflow support.

The high level goals of this project are:

1. Research the state of the art in workflow management.
2. Research the current support in Kubernetes for workflow management.
3. Design and implement workflow management in or on top of Kubernetes, preferably in cooperation with the Kubernetes community.

2.2.2. Derived problem description

The derived problem description will be the basis for the main research topics addressed in this project. It is based on the problem description from Nerdalize and the academic requirements specified by TU Delft. The derived description differs from the original description by being more abstract rather than focus on implementation details. A proof of concept implementation will be developed to show how the abstract concepts may be used to solve the real world problem posed by Nerdalize.

The derived problem description is:
"How can workflows be managed in a distributed containerized environment?"

2.3. Related work

In this section we will discuss some of the existing workflow schedulers (Section 3.1),

2.3.1. Pegasus

Pegasus is a workflow management system used to automate, recover and debug scientific computations. It is a sophisticated piece of software used by research organisations to run simulations and analysis on some of the largest computing grids in the world. These applications often involve the processing of large amounts of data. The computing grid consists of many heterogeneous machines provided by different, geographically spread out institutions. Pegasus introduces the notion of a workflow as "an abstract description of application components and their dependencies"[24]. The workflow may be represented as a directed acyclic graph (DAG). This approach leads to a highly flexible description where individual components may be replaced with alternative implementations. The abstract representation then undergoes several refinement steps geared towards making it executable and increasing performance. The result is an optimized, executable mapping from an abstract workflow onto physical machines.

2.3.2. Luigi

Luigi is a workflow manager written in Python at Spotify Inc. It helps users manage complex pipelines of batch jobs by handling dependency resolution, workflow management and visualization[2]. Luigi workflow specifications are written in Python leveraging the expressiveness of the programming language itself. This makes it easy to build up the dependency graph in which dependencies may include recursive references, date algebra etc. A visualizer is provided out of the box, providing valuable insights regarding task status, the dependency graph and workers running the tasks.

2.3.3. Airflow

The third existing solution is Airflow developed by AirBnB[1]. Airflow is similar to Luigi in that the system and the workflow definition are written in Python. Operators are a feature that distinguish Airflow. Types of operators include *sensor*, *remote execution* and *data transfers*. The sensor waits for an event to happen before proceeding to the next task in the workflow. Examples of events are a file appearing in a specified folder or the existence of a particular record in a SQL database. The second type of operator is remote execution, which triggers an external script. Finally the data transfers operator is responsible for moving data from one place to another. This data may be stored in a database of some kind or in a regular file. Airflow uses a plug-in architecture to ease the extension with new functionality.

2.3.4. Oozie

Apache Oozie is a workflow scheduler for Hadoop[3]. Oozie is tightly integrated with the Hadoop stack supporting Java map-reduce, Streaming map-reduce, Pig, Hive, Sqoop and Distc. A workflow is defined by an XML file which may contain two types of nodes. The first type is a *control flow node*, providing a mechanism to control the workflow execution path. The second type of node is the workflow *action node*, which describes the execution of a computation or processing task. XML properties are used to create connections between nodes. These connections may be conditional meaning that a corresponding predicate needs to be satisfied before the transition takes place.

3

Requirements

This chapter describes the requirements of the system. The requirements are gathered mainly by interviewing Nerdalize employees as an initial starting point, since they can best describe Nerdalize's needs. In the interviews employees provided us with several use cases, which we translated into users stories and requirements. We plan on involving the Kubernetes open-source community to refine or extend the current requirements, as they probably have other use cases then Nerdalize.

The rest of this chapter is organized as follows: Section 3.1 describes the interviews with Nerdalize employees, in section 3.2 we list the user stories and section 3.3 lists the resulting requirements.

3.1. Interviews

The first interview we had was with Eric Feliksik, an engineer at Nerdalize. This was shortly after we had chosen the bachelor project, three weeks before its start. At this time we were very new to the field of distributed so we discussed many of the important concepts. He helped us get an overview of the technologies involved and how they work together. We asked many questions to address knowledge gaps we had. His answers consisted mostly of architecture diagrams drawn on a whiteboard. After the meeting we took home a reading list with recommended literature to get up to speed as quickly as possible.

The second interview was with Mathijs de Meijer and Ad van der Veer, both employees at Nerdalize. The goal of this interview was to get their input to construct use cases and eventually requirements. These use cases can be found in Appendix A. Together we went over the pros and cons of several different approaches, finally settling for a preliminary design. The resulting user stories can be found below.

3.2. User Stories

The interviews with Nerdalize helped clarify which use cases they encountered and what is needed to satisfy them. We then composed a list of user stories that encapsulate all functionality, such that we end up with a high level description of system functionality. The user stories are all in the format *As a <user>, I want <action>, so that I <reason>* to ensure consistency.

1. As a user, I want *the system to do scheduling in a distributed environment*, so that I *can make use of parallelism to improve performance*.
2. As a user, I want *to add tasks that depend on the completion of other tasks*, so that I *can create a workflow*.
3. As a user, I want *to add or remove tasks during workflow execution*, so that I *can make changes at runtime*.
4. As a user, I want *to ensure that certain tasks do not run on the same machine*, so that I *can improve reliability*.
5. As a user, I want *to co-locate tasks that work on the same data*, so that I *can reduce network overhead*.
6. As a user, I want *to customize the scheduler*, so that I *can tailor it to my needs*.

7. As a user, I want *to interact with the workflow through an API*, so that I *can integrate it with my existing system*.

8. As a user, I want *to run workflows on my local machine*, so that I *can test before deployment*.

3.3. Requirements

Based on the list of user stories we created the following requirements. The requirements are grouped into four categories which are: *workflow manager*, *task*, *workflow* and *scheduler*. The workflow manager is the component responsible for handing tasks for which all constraints are satisfied to the scheduler. The scheduler ensures tasks are assigned to a suitable host according to a specified policy. A scheduling policy uses information about the workflow and the state of the hosts to make a scheduling decision.

Requirements are prioritized using colored boxes representing categories as defined by the MoSCoW[12] method. The green box (🟩) stands for must have, the orange box (🟧) stands for could have and the red box (🟥) stands for won't have.

Workflow manager:

1. 🟩 The workflow manager executes a workflow based on a formal definition of tasks and constraints.
2. 🟩 The workflow manager monitors the execution of the workflow
3. 🟥 The workflow manager is responsible for providing data required by a task.

Task:

4. 🟩 Tasks' execution may depend on the status (failed, succeeded, running) of one or more other tasks.
5. 🟥 Tasks may be other workflows.
6. 🟧 Tasks can be added and/or removed through the Workflow Manager API during the execution of a workflow.

Workflow:

7. 🟩 A workflow may run on a single machine or a cluster of machines.
8. 🟩 A workflow may be cancelled during execution.
9. 🟧 Resource usage of a workflow may be limited as specified by the user.

Scheduler:

10. 🟩 The scheduler's policy should be customizable so that users may tailor it to their needs.
11. 🟩 The default scheduling policy can handle requests from the user to run two or more tasks on separate hosts.
12. 🟩 The default scheduling policy can handle requests from the user to run two or more tasks on the same host.

Requirement 3 and 5 are beyond the scope of this project, due to time constraints. The system will be extensible in such a way that functionality may be added at a later point in time. Requirements 6 and 9 are the least crucial at this moment in time. We decided priorities in collaboration with Nerdalize to address the most pressing business needs first.

4

Our approach

In this chapter we will discuss our approach to make the project successful. In particular section 4.1 will cover the development methodology we used and why it was chosen over several alternatives. Section 4.2 introduces the programming language used for the implementation. How tools are used througout the project to support project related processes is described in section 4.3. The preliminary design is introduced in section 4.4 which contains an architecture overview, an explanation of how and why we use Kubernetes, and an API design. Section 4.5 will describe measures we take to ensure the product will be of high quality. The last section 4.6 will go over the risks associated with the project and how we seek to minimize them.

4.1. Development methodologies

During the project the team will make use of agile development methodologies. Scrum is the framework of choice. In this section we will explain this choice as well as go over some of the available alternatives.

Scrum is an iterative and incremental agile software development framework for managing product development [13]. It allows a small team of developers to cope with changing requirements and shifting priorities. Scrum also encourages team members to communicate often and openly about progress as well as encountered problems. This leads to a highly flexible development process. The reason we chose an iterative approach is because we foresee many changes as the project progresses. We will work closely with the engineers at Nerdalize as well as the Kubernetes community. Both may have different requirements or change requirements as new wishes become more apparent.

An alternative approach is the waterfall model. It works well when all requirements are clear at the start of the project and are unlikely to change. As this is not the case it is unsuitable.

Now that it is clear that agile practices are best suited for our project, it still is worth noting alternatives to scrum exist. The main reason we chose scrum over for example XP[17] is the fact that the software team at Nerdalize uses scrum. By using the same methodology we are able to learn from them how it is used in practice.

4.2. Programming language

The implementation of our design will use the Go programming language. The available concurrency constructs make it well suited for distributed applications such as our project. Go is very popular among developers working on container related technologies. This is largely due to the increasing popularity of the container platform Docker[6], which itself is written in Go. As discussed in section 4.4, our solution will build on top of the open source project Kubernetes, which is also written in Go.

4.3. Tools usage

Tools are used to support project related processes. Two types of tools will be discussed in this section. subsection 4.3.1 will focus on development tools whereas subsection 4.3.2 will go over process tools. Whenever alternatives are available, the decision to choose a particular solution is substantiated.

4.3.1. Development tools

Development tools are used to manage, test and manipulate code. Vim[16] and Atom[5] are popular text editors which will be used to add and remove code as well as navigate the codebase. Other choices such as Microsoft Visual Studio Code or Sublime Text were considered. As many of these editors provide similar functionality however, our choice was based mostly on personal preference. Editor plugins provide additional functionality to make them better suited for editing code in the Go programming language. Features include running tests from within the editor, detecting and fixing formatting issues and looking up online documentation.

Our team consists of two members who will simultaneously work on the same codebase. Git[7] is a version control tool that helps with this collaboration process. Version control also provides an easy way to revert to an older version when changes do not have the desired effect. Whenever changes are ready to be shared, they are pushed to Gitlab[8]. This is a hosted service providing a central repository for all developers to download the latest changes. Alternatives to Git such as SVN or Mercurial have been considered. The team however has used Git in the past, making it an attractive choice. Git has the added benefit that it is used by many in the industry and academia. As our project will be open source this allows for a smooth interaction between contributors.

The Go toolchain provides many useful tools. The test runner can be used to run all tests in a package and obtain a formatted report. Optionally the test runner may also report code coverage if so desired. The race detector will take a piece of code and test for situations in which a race condition occurs. If such a condition is found it is reported along with the executed instructions that led up to it. Finally gofmt is a tool that ensures code is compliant with the style guide. If this is not the case, the tool has the ability to fix any formatting issues.

4.3.2. Process tools

Process tools help to facilitate many aspects of the process. These aspects include planning, prioritizing, communication etc. Nerdalize uses Google apps for work. In particular it uses Gmail for all communication between team members. Gmail provides access to an address book with contact information about all employees. Google calendar is used to schedule (group) meetings. It is possible to see other's calendars to check their availability.

To complement the scrum methodology we will use Trello. This is an online tool to create *cards* and add information to them. In scrum, a product backlog is used to list features to be implemented in order of importance. Using Trello one would create a card for each of the items and drag them in the correct order. Cards have a title and a description similar to a traditional post-it note. Unlike post-its cards may also have labels associated with them, a due date and members assigned to it. This meta information makes the solution very flexible as cards may be changed without the limited space of a post-it. Trello is used by Nerdalize to facilitate scrum. This is the main reason we chose to use it as we are able to learn from their experience and get support as the project progresses.

4.4. Preliminary design

According to the requirements specified in chapter 3 we created a preliminary design. In subsection 4.4.1 we will give an overview of the architecture of the system and in subsection 4.4.3 we will describe the preliminary API.

4.4.1. Architecture overview

One of the requirements is for the system to consist of different components, communicating with one another. The main components are the *workflow manager*, the *scheduler* and the *controller*. A complete overview of the architecture is shown in Figure 4.1.

The user communicates to the workflow manager through the API. It is responsible for handing tasks for which all constraints are satisfied over to the scheduler. Each task may have multiple constraints, which specify a dependency on another task's status. An example of a constraint is `taskB.status == 'succeeded'`. The generic term *constraint* was chosen in favor of *dependency* to make it possible to use other types of constraints in the future. The status of a task can be one of the following: *running*, *succeeded*, or *failed*.

The scheduler is responsible for assigning tasks to one of the available hosts, according to a scheduling policy. It is aware of the hosts in the cluster and their available resources. Conforming to requirements 11 and 12 from section 3.3 the default scheduling policy is able to handle constraints to schedule two or more tasks on separate hosts or schedule two or more tasks on the same host. The default scheduling policy can be

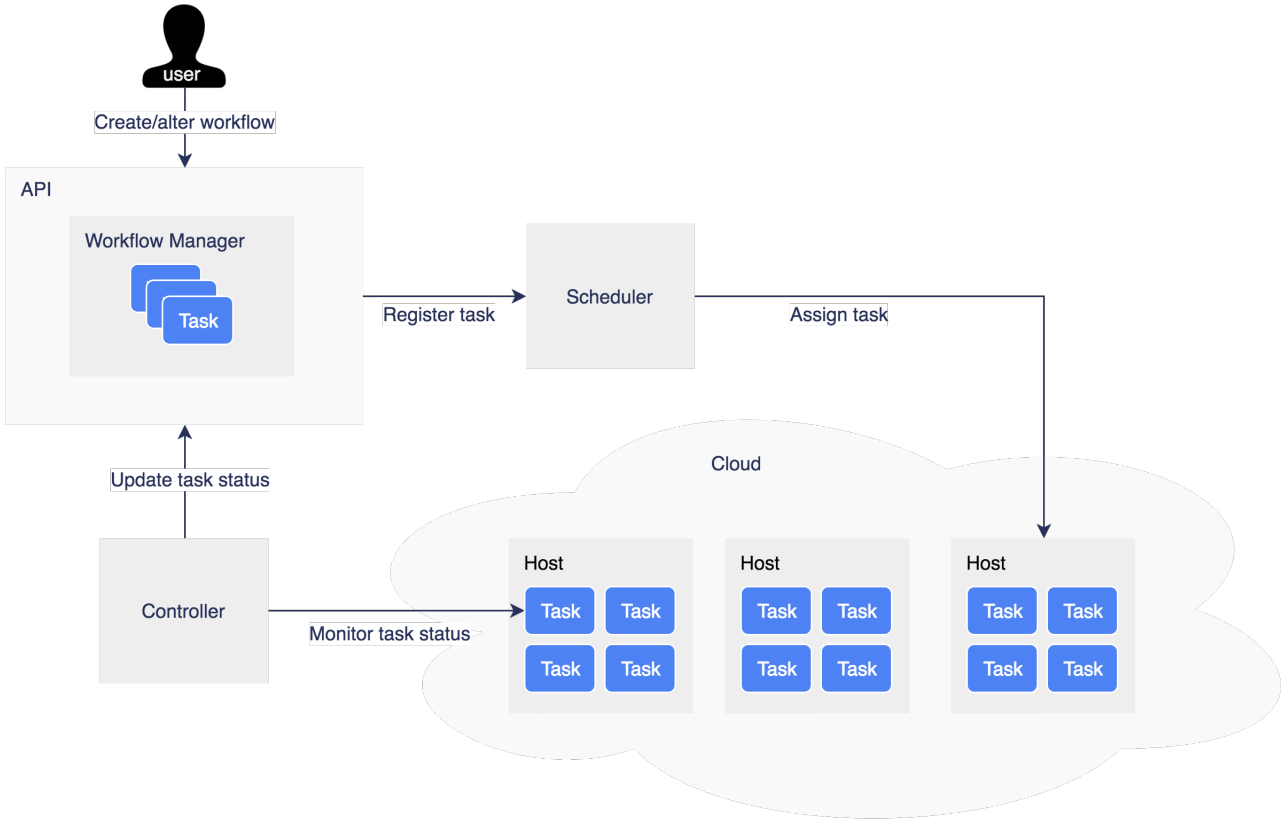


Figure 4.1: Overview of system architecture

replaced by custom policies to account for user-specific needs.

The controller monitors running tasks and reports their status to the workflow manager through the API. We chose to create a separate component for the workflow manager, the workflow scheduler and the workflow controller to ensure each has a single responsibility.

4.4.2. Kubernetes

The design created in subsection 4.4.1 will be integrated with Kubernetes. We decided to build our proof of concept implementation on top of Kubernetes for a number of reasons: Kubernetes (i) orchestrates containers, (ii) has default scheduling and controller components which can be extended, (iii) is open source, (iv) is used by Nerdalize making this integration useful to them.

Kubernetes provides a default scheduler and controller which both are replaceable by a custom implementation. Default components will be implemented one at the time to ensure a working system throughout the process. The first step is to add monitoring functionality to the controller. The second step is to implement the workflow manager with the ability to execute DAG workflows. The last step is to replace the Kubernetes default scheduler with a custom one with support for workflow specific constraints as defined in section 3.3. This order of implementation was chosen (i) because the workflow manager uses information provided by the controller and (ii) because the workflow manager is not dependent on a specific implementation of the scheduler.

Tasks are implemented as *jobs* in Kubernetes. Monitoring by the controller is done using Kubernetes' label system. An example of a label is `jobName`, which can be used to select a job by its name. Scheduling of jobs on appropriate hosts is done by using labels as well. By using labels the scheduler can identify two or more jobs that should or should not run on the same machine.

4.4.3. API design

The API is used by system components and by the user to interact with the workflow manager. We chose to use the REST architectural style because it is widely used in the industry. As a result it provides a familiar

Type	Path	Parameters	Response	Description
POST	/workflow	name (string)	wid (int)	Create a new workflow
GET	/workflow	wid (int)	name (string)	Get a workflow name
DELETE	/workflow	wid (int)		Remove a workflow
GET	/workflow/tasks	wid (int)	tasks (array)	Get the tasks of a workflow
POST	/task	wid (int) name (string) dependency (string, string)	tid (int)	Add a new task to the workflow
GET	/task	tid (int)	status (pending, running, succeeded, failed)	Get the status of a task
DELETE	/task	tid (int)		Remove a task from a workflow
UPDATE	/task	tid (int) status (pending, running, succeeded, failed)		Update the status of a task

Figure 4.2: Initial API design.

framework to other developers.

The initial design of the API is shown in Figure 4.2. It should be noted that the API is very likely to change during the project when requirements or design elements change.

Two services are provided by the workflow manager. The first one is related to workflows. Through the API a user may view, create, edit or delete workflows. The second service is related to tasks. Tasks may be added or removed from a workflow and may have their status updated.

4.5. Quality Guarantees

We will take different measures to guarantee a high quality product. These measures relate to both the code and the process as both are a crucial part of the project. In the first section we will go over how documentation is done and in which ways it contributes to the overall quality. After that our testing strategy is described which ensures written code behaves the way it is supposed to. The final section describes evaluation methods used to measure the quality of the system.

4.5.1. Documentation

The working process will be documented in the final report. This report will be updated weekly to reflect the advancements made and the hurdles encountered. Keeping this log will help us and interested readers understand decisions and changes made throughout the project. The report is written in LaTeX using the ShareLatex[4] online collaborative editor. This allows us to work on the report together and see the work as it progresses.

Many lines of code will be written over the span of two months. To make it easy for us and for other people to understand and change the code, it will be thoroughly documented. The Go programming language contains a tool called fmt[9], which ensures every block of code contains a descriptive comment and is formatted according to the Go style conventions. We will use this tool to ensure all the code is properly documented before it is pushed to a remote repository. Documentation in HTML format may be generated from the code, making it easily accessible for everyone.

4.5.2. Testing

Every piece of code added to the system should not contain any unexpected behavior. Tests are well suited to ensure this is indeed the case. Unit tests provide a way to verify that a piece of code, in isolation, produces the desired result. The added benefit of unit testing code is to prevent regressions. These occur when new functionality is added which breaks old code. By running a suite of tests on every build new as well as old code is tested to ensure no regression takes place. End-to-end tests are used to test large parts of an application. We will include these tests to make sure the entire application works as expected.

A convenient way to automate this process is through continuous integration. For our project we will use Jenkins[10]. We chose this solution because of our previous experience with it. It provides Go support

through a plug-in making it quick and easy to set up. The continuous integration server will allow every code push to a remote repository to trigger the full set of tests. In case tests fail, a notification is sent describing what went wrong. This way of testing also makes sure that code runs on all machines rather than only on one developer's machine.

4.5.3. Evaluation Methods

It is important to think about how to measure the quality once the system is finished. In this section we will discuss different methods that may be used for this purpose.

The first methods we will use is a code quality report generated by the Software Improvement Group (SIG)[14]. This report summarizes their analysis by assigning scores to various aspects of the code. These aspects include coupling, maintainability and code duplication.

The second method evaluates the correctness of the system. Evaluation is done by running well-known workloads such as Montage, CyberShake, LIGO, and the Pegasus workflow generator [19, 28]. The metric to measure the performance of the system is *execution time*, which can be compared to the theoretical optimal run time.

4.6. Risk Analysis

The project has several associated risks. In this section we will highlight some of the most important ones and how we will deal with them.

The first major risk is our decision to use Kubernetes. This project led by Google is currently in active development by a large and lively community. Many companies use it in production to orchestrate and monitor their containers. As the container ecosystem is rapidly evolving there is a possibility Kubernetes will lose momentum in favor of another technology. The workflow manager implemented in our project however, takes a more abstract approach such that the concepts described may be implemented in whatever technology is relevant at the time.

The second risk is our unfamiliarity with the Go programming language. Go is very popular in the container community. Kubernetes is also written in Go, and hence our implementation will be as well. Learning a new language may be difficult, especially when it contains many unique features such as with Go. To get up to speed we practiced before the start of the project. During the project we will have weekly workshops with an experienced Nerdalize engineer who will teach us important concepts from a practical point of view.

Our limited experience on programming distributed systems should be taken into account. Errors in distributed systems are significantly harder to detect and to fix due to their non-deterministic nature. Some concepts include mutexes, locks, threads and channels. The Go toolchain recently added support for a race detector which may be run on a piece of code to ensure no race conditions take place. This will greatly aid the debugging process and may preemptively give warnings when subtle errors are introduced.

Lastly we hope to use the supportive community surrounding Kubernetes as well as the people at Nerdalize to provide us with guidance where necessary. As the code will ultimately be open source the whole world will have the ability to look at the implementation and suggest changes where appropriate. We will take this help with both hands as most of the aforementioned people are seasoned engineers.

5

Planning

In this chapter an initial planning is described. It includes planning with respect to engineering, the report, the presentation, and community engagement. An overview of the planning is presented in Figure 5.1.

In the engineering section, planning regarding code implementation is shown. The section includes the main components which need to be implemented and deadlines for the software improvement group (SIG). The reporting section includes deadlines and drafts of the research report and the final report. Workshops, presenting at Nerdalize and presenting at the distributed systems (DS) group are shown in the presenting section. The community engagement section contains a planning for submitting a proposal and contributing to Kubernetes.

We are aware of the risk that is involved with contributing to the open source community. It is possible that the review of the proposal will take more time than originally planned. If this is the case, we will start implementing the proposed solution before the community agrees. This is to make sure that we have a working system by the end of the project.

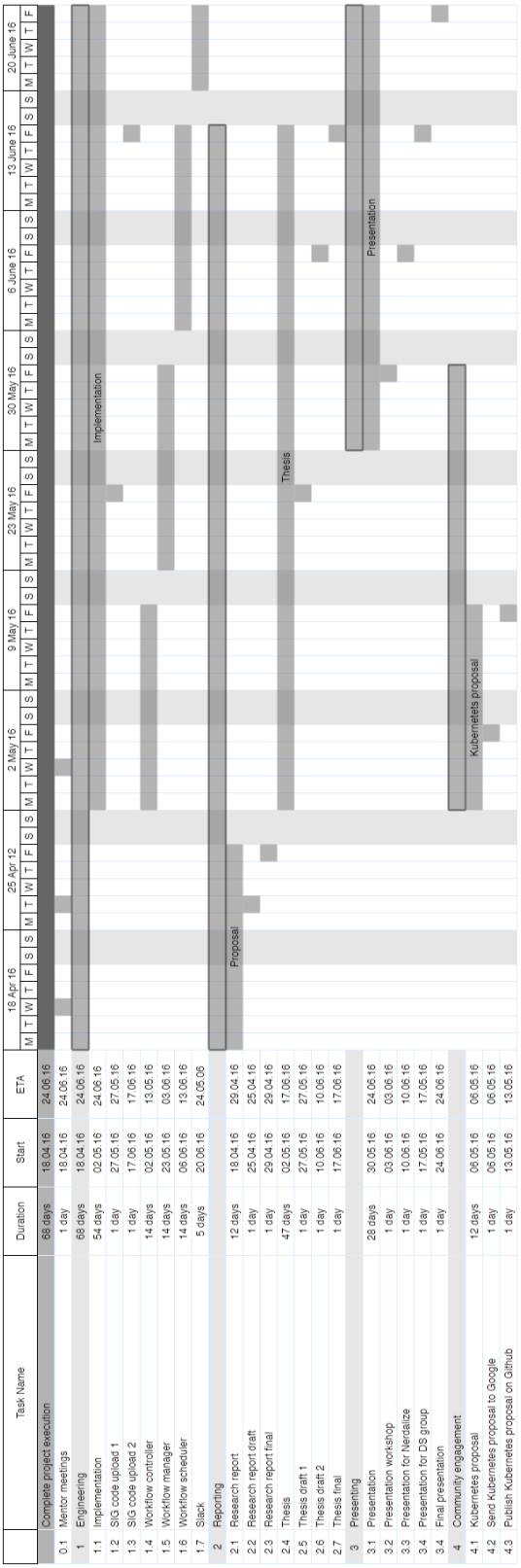


Figure 5.1: Preliminary planning of the project.

A

Use cases

The following information was provided by Nerdalize:

In principle workflows can be designed to be of arbitrary complexity, it is hypothesized that most practical (Nerdalize) use cases can be covered with a relatively small feature set. We sketch some possible workloads encountered in the wild below.

Workloads can consist of multiple steps executed in parallel.
Example: Assuming a embarrassingly parallelizable workload, on >=1 machine do:

1. download data partition from networked storage
2. functional transformation
3. upload results to networked storage

Workloads can consist of multiple steps of which all must be executed in parallel on the same machine.
Assuming a single threaded, parallelizable workload:

1. Download input data from networked storage
2. Run multiple instances of the workload software in parallel on the same machine (e.g. 1 process per physical core), all processes need low latency access to the same dataset
3. Upload results to networked storage

Workloads can consist of multiple steps of which all must be executed in parallel on the different machines.
Example: Assuming a parallelizable workload, where the software is multicore optimized.
On multiple machines:

1. Download an input data item from networked storage
2. Run one instance of the workload software
3. Upload results to networked storage

Workloads can consist of multiple parallel steps of which >1 step can be scheduled to run on one machine.
Example: Assuming parallelizable workload, where the software is not multicore optimized.
On multiple machines, start multiple processes that do:

1. Download an input data item from networked storage
2. Run one instance of the workload software
3. Upload results to networked storage

Workloads can consist some serial and some parallel steps.
Example:

1. On 1 machine:
   (a) Download all input data
   (b) Do some transformation (e.g. partition an image into tiles)
2. Then on multiple machines:
   (a) Transform all partitions input into output, one partition per machine
3. Then on one machine:
   (a) Stitch all partition output together

Some steps can be conditional on the output of a previous step.
Example:

1. Download input data from networked storage
2. Functional transformation
3. Choose from 3 types of Post processing based on aspects of the output of the functional transformationg
4. Upload post processed output data to networked storage

Bibliography

[1] Airflow. https://pythonhosted.org/airflow/start.html. Accessed: 2016-04-25.

[2] Luigi. https://github.com/spotify/luigi. Accessed: 2016-04-25.

[3] Oozie. https://oozie.apache.org/docs/4.2.0/index.html. Accessed: 2016-04-25.

[4] Sharelatex. https://www.sharelatex.com. Accessed: 2016-04-25.

[5] Atom editor. https://atom.io/. Accessed: 2016-04-28.

[6] Docker. https://www.docker.com/. Accessed: 2016-04-28.

[7] Git. https://git-scm.com/,. Accessed: 2016-04-28.

[8] Gitlab. https://about.gitlab.com/,. Accessed: 2016-04-28.

[9] gofmt. https://golang.org/cmd/gofmt. Accessed: 2016-04-25.

[10] Jenkins. https://jenkins.io/. Accessed: 2016-04-29.

[11] Kubernetes. http://kubernetes.io/. (Accessed on 04/29/2016).

[12] Moscow model. http://www.ietf.org/rfc/rfc2119.txt. Accessed: 2016-04-29.

[13] Scrum alliance. https://www.scrumalliance.org/. Accessed: 2016-04-27.

[14] Sig. https://www.sig.eu/. Accessed: 2016-04-25.

[15] What is a workflow management system? | taverna. http://www.taverna.org.uk/introduction/what-is-a-workflow-management-system/. (Accessed on 04/28/2016).

[16] Vim editor. http://www.vim.org/. Accessed: 2016-04-27.

[17] Xtreme programming. http://www.extremeprogramming.org/. Accessed: 2016-04-27.

[18] Adam Barker and Jano Hemert. *Parallel Processing and Applied Mathematics: 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007 Revised Selected Papers*, chapter Scientific Workflow: A Survey and Research Directions, pages 746–753. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-68111-3. doi: 10.1007/978-3-540-68111-3_78. URL http://dx.doi.org/10.1007/978-3-540-68111-3_78.

[19] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. In *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, pages 1–10. IEEE, 2008.

[20] Scott Callaghan, Ewa Deelman, Dan Gunter, Gideon Juve, Philip Maechling, Christopher Brooks, Karan Vahi, Kevin Milner, Robert Graves, Edward Field, David Okaya, and Thomas Jordan. Scaling up workflow-based applications. *Journal of Computer and System Sciences*, 76(6):428 – 446, 2010. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/j.jcss.2009.11.005. URL http://www.sciencedirect.com/science/article/pii/S0022000009001214. Special Issue: Scientific Workflow 2009The 2nd International Workshop on Workflow Management and Application in Grid Environments amp; The 3rd International Workshop on Workflow Management and Applications in Grid Environments.

[21] David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian Taylor, and Ian Wang. Programming scientific and distributed workflow with triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[22] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda. Griphyn and ligo, building a virtual data grid for gravitational wave scientists. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 225–234, 2002. doi: 10.1109/HPDC.2002.1029922.

[23] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003. ISSN 1572-9184. doi: 10.1023/A:1024000426962. URL `http://dx.doi.org/10.1023/A:1024000426962`.

[24] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3): 219–237, July 2005. ISSN 1058-9244. doi: 10.1155/2005/128026. URL `http://dx.doi.org/10.1155/2005/128026`.

[25] Patrick Galbraith. Docker: Containers for the masses. `http://patg.net/containers,virtualization,docker/2014/06/05/docker-intro/`, jun 2014. (Accessed on 04/29/2016).

[26] Betty Junod. Containers as a service (caas) as your new platform for application development and operations | docker blog. `https://blog.docker.com/2016/02/containers-as-a-service-caas/`, feb 2016. (Accessed on 04/29/2016).

[27] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[28] Gaurang Mehta. Workflowgenerator - pegasus - pegasus workflow management system. `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`, oct 2009. (Accessed on 04/29/2016).

[29] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi. Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 401–409, May 2007. doi: 10.1109/CCGRID.2007.101.

[30] Robert Endre Tarjan. *Data structures and network algorithms*, volume 44. Siam, 1983.

[31] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[32] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, page gkt328, 2013.

B

Reading list

At the start of the project Nerdalize gave us this list of technologies we had to become familiar with:

- Ubuntu 15.10
- Docker
- Kubernetes
- Golang
- Hyper.sh and Hypernetes
- Virtual Machines (the concept)
- OpenStack

Nerdalize provided us with the following reading list:

- Ansible for DevOps (book)
- The Docker & Container Ecosystem (book)
- Docker Networking and Service Discovery (book)
- Kubernetes Up & Running (book)
- Gobyexample.com (website with tutorials on Golang)
- Golang.org (official Golang website)
- Hyper.sh (website)
- Kubernetes.io (website with official Kubernetes docs)

C

Initial roadmap

This appendix shows the initial roadmap created for the project. Milestones are described using numbers. They are divided into smaller tasks which are labeled using letters.

Prerequisites

1. Research libraries for:
   (a) CLI functionality
   (b) DAG data structure
   (c) Logging
2. Develop a testing strategy

The WF Manager can

3. perform basic job operations
   (a) start/stop a job
4. handle workflows specified in a formal format
   (a) define initial workflow specification format: list of independent jobs
   (b) start workflows from a specification file
5. monitor the status of a workflow
   (a) get job status (running, failed, success)
6. start a workflow with two jobs and a dependency constraint
   (a) extend workflow specification format to support dependency constraints
   (b) validate workflow specification (dag)
   (c) resolve dependencies and supply scheduler with jobs
   (d) stop a workflow prematurely
7. make scheduling decisions using workflow information
   (a) support placement constraint for 1+ jobs on different machines
   (b) support placement constraint for 1+ jobs on same machine

D

SIG Feedback

We received the following feedback from SIG in Dutch. This appendix shows two emails since we asked for a clarification about the analysis in the first email.

D.1. Initial analysis

De code van het systeem scoort bijna 3.5 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Duplication en Unit Size.

Voor Duplicatie wordt er gekeken naar het percentage van de code welke redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.

De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

D.2. Clarification

In het geval van Unit Size zijn transitionWorkflow(), process(), en processJobStep() in het bestand transitioner.go goede voorbeelden. De aanbeveling om eens naar de lengte van methodes te kijken geldt niet alleen voor de Go-code, ook in JavaScript kan bij code.js het één en ander aangepast worden. Jullie hebben daar nu een erg lange onReady() callback. Die is nu al vrij moeilijk leesbaar, maar deze aanpak wordt echt een probleem op het moment dat jullie code verder gaat groeien. Het is beter om de functionaliteit in die callback uit de splitsen naar aparte functies op basis van functionaliteit. Jullie doen dat al een klein beetje met highlightNextElement(), maar die aanpak zou je eigenlijk verder willen doorvoeren.

Aangezien de Unit Size een stuk problematischer dan de (kleine) hoeveel duplicatie is, stel ik voor dat jullie gezien de tijd je focussen op het verder verbeteren van bovenstaande voorbeelden. Als jullie naast die voorbeelden nog meer aanpassingen doen zijn we helemaal blij.

E

Scale factor calculation

The heat generated by the CPU we used for different numbers of cores is displayed in Appendix E. As we can see the heat does not scale linearly with the amount of cores. For the experiment we used GCP machines with 2 cores corresponding to 100 joules/second. In the calculation below we calculate how long the experiment would take if we would use the real joules/second generated rather than the scaling factor.

cores	joules/second
0	83
1	90
2	100
4	112
8	133
12	153
16	176

Table E.1: Heat generation in joule/s for different numbers of cores with 100% utilization. Source: Measurements of real Nerdalize servers, provided by Nerdalize employee Remy van Rooijen

$t_s = 420s$	simulation time
$s = 500$	scaling factor used by the boiler simulator
$e = \frac{1}{95\%} = 1,053$	heat exchange efficiency factor
$c_s = 2$	cores per machine in the simulation
$c_r = 16$	cores per machine in the CloudBox
$b_s = 1$	motherboards per machine in the simulation
$b_r = 6$	motherboards per machine in the CloudBox
$h_s = 50\,j/s$	generated heat per CloudBox on a GCP machine
$h_r = 11\,j/s$	generated heat per coresecond in the CloudBox
$l = 23\% = 0,23$	average machine load factor
$h = \frac{h_s \times h_s}{h_r \times h_r} = \frac{50 \times 2}{11 \times 16} = 0,57$	calculate cores scale factor
$b = \frac{b_s}{b_r} = \frac{1}{6} = 0,17$	calculate motherboard scaling factor
$t_r = t_s \times s \times h \times b \times e \times l =$	
$420 \times 500 \times 0,57 \times 0,17 \times 1,053 \times 0,23 = 1,36h$	calculate time it would take in reality

F

Heat Scheduler savings

$w = 32750000$	joules wasted per simulation per machine
$m = 50$	amount of machines used in simulation
$d = 365$	days in a year
$n = 6$	amount of machines per CloudBox
$g = 28300000$	joules generated burning 1 m3 of gas in euros
$p = 0,63$	price of gas per m3
$a = \frac{w}{g} = \frac{32750000}{28300000} = 1,16$	m3 gas wasted
$s = \frac{a \times p \times n}{m} = \frac{1,16 \times 0,63 \times 6}{50} = 0,09$	daily amount of euros saved per CloudBox
$y = s \times d \times n = 0,015 \times 365 = 32,00$	yearly amount of euros saved per CloudBox

G

Infosheet

Flower infosheet

Title:	**Workflow management and heat-aware scheduling for modern cloud infrastructures**
Date of presentation:	**July 1, 2016**
Client:	**Mathijs de Meijer**, Nerdalize
Coach:	**Dr. ir. Alexandru Iosup**, Parallel and Distributed Systems Group, EWI, TU Delft
Team members:	**Robert Carosi**, r.g.b.carosi@student.tudelft.nl
	Boris Mattijssen, b.j.h.mattijssen@student.tudelft.nl

Cloud computing has given users access to virtually infinite computing power. Cloud service providers operate data centers consisting of hundreds, sometimes thousands of interconnected machines. The machines dissipate heat that is undesirable because it may lead components to overheat. Air conditioners are used to cool the air and regulate the temperature in the data center. Nerdalize is a company that aims to create a cloud by replacing servers in a data center with CloudBoxes in people's homes. A CloudBox contains servers and a boiler. The heat dissipated by the servers is used to warm up the water in the boiler and complement the central heating system. The challenge set out by Nerdalize is to develop a workflow management system to run on their infrastructure while making optimal use of generated heat. Many workflow management systems exist but none supporting a containers based environment.

Flower is a workflow management system in a containerized environment with heat-aware scheduling. Flower consists of several components. The first component is the workflow management system responsible for executing a workflow on a cluster of machines. The second part contains various temperature-based scheduling policies. To the best of our knowledge we are the first ones to introduce the notion of heat when making scheduling decisions. The final Flower component is a workflow visualizer to monitor workflow execution.

To manage product development, we made use of the agile development methodology Scrum. Further assistance was provided by multiple tools, such as Travis CI for continuous integration and GitHub for team communication. During the project we have had extensive conversations with employees from Nerdalize and professionals from companies like Google, RedHat, CoreOS and Amadeus.

Bibliography

[1] Airflow. https://pythonhosted.org/airflow/start.html. Accessed: 2016-04-25.

[2] Luigi. https://github.com/spotify/luigi. Accessed: 2016-04-25.

[3] Oozie. https://oozie.apache.org/docs/4.2.0/index.html. Accessed: 2016-04-25.

[4] Caltech comment-to-code ratio. http://everything2.com/title/comment-to-code+ratio. Accessed: 2016-06-23.

[5] Coveralls. https://coveralls.io/. Accessed: 2016-04-27.

[6] Docker. https://www.docker.com/. Accessed: 2016-04-28.

[7] Github. https://github.com/. Accessed: 2016-06-23.

[8] Godoc. https://godoc.org/. Accessed: 2016-06-23.

[9] Gofmt. https://golang.org/cmd/gofmt/. Accessed: 2016-06-23.

[10] Jenkins. https://jenkins.io/. Accessed: 2016-04-29.

[11] Kubernetes scheduler policies. https://github.com/kubernetes/kubernetes/blob/master/docs/devel/scheduler_algorithm.md, . Accessed: 2016-06-23.

[12] Kubernetes readme. https://github.com/kubernetes/kubernetes/, . Accessed: 2016-06-23.

[13] Kubernetes. http://kubernetes.io/. Accessed: 2016-04-29.

[14] Moscow model. http://www.ietf.org/rfc/rfc2119.txt. Accessed: 2016-04-29.

[15] Slack. https://slack.com/. Accessed: 2016-06-23.

[16] Stack overflow. http://stackoverflow.com/. Accessed: 2016-06-23.

[17] stress-ng. http://kernel.ubuntu.com/~cking/stress-ng/. Accessed: 2016-06-23.

[18] What is a workflow management system? | taverna. http://www.taverna.org.uk/introduction/what-is-a-workflow-management-system/. Accessed: 2016-04-28.

[19] Travis ci. https://travis-ci.org/. Accessed: 2016-04-27.

[20] Trello. https://trello.com/. Accessed: 2016-04-25.

[21] Wercker. http://wercker.com/. Accessed: 2016-04-27.

[22] Adam Barker and Jano Hemert. *Parallel Processing and Applied Mathematics: 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007 Revised Selected Papers*, chapter Scientific Workflow: A Survey and Research Directions, pages 746–753. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[23] Scott Callaghan, Ewa Deelman, Dan Gunter, Gideon Juve, Philip Maechling, Christopher Brooks, Karan Vahi, Kevin Milner, Robert Graves, Edward Field, David Okaya, and Thomas Jordan. Scaling up workflow-based applications. *Journal of Computer and System Sciences*, 76(6):428 – 446, 2010.

[24] David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian Taylor, and Ian Wang. Programming scientific and distributed workflow with triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[25] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[26] E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams, and S. Koranda. Griphyn and ligo, building a virtual data grid for gravitational wave scientists. In *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pages 225–234, 2002.

[27] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1(1):25–39, 2003.

[28] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3): 219–237, July 2005.

[29] Patrick Galbraith. Docker: Containers for the masses. `http://patg.net/containers,virtualization,docker/2014/06/05/docker-intro/`, jun 2014. Accessed: 2016-04-29.

[30] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[31] Gaurang Mehta. Workflowgenerator - pegasus - pegasus workflow management system. `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`, oct 2009. Accessed: 2016-04-29.

[32] Sam Newman. *Building Microservices*. "O'Reilly Media, Inc.", 2015.

[33] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi. Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 401–409, May 2007.

[34] Robert Endre Tarjan. *Data structures and network algorithms*, volume 44. Siam, 1983.

[35] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[36] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, page gkt328, 2013.