College of Engineering, Design and Physical Sciences
Department of Electronic and Computer Engineering

Distributed Computing Systems Engineering

Master's Thesis

Conception and Realization of a Distributed and Automated Computer Vision Pipeline

---

Michael Watzko
1841795

March 31st, 2020

Declaration: I have read and I understand the MSc dissertation guidelines on plagiarism and cheating, and I certify that this submission fully complies with these guidelines.

College of Engineering, Design and Physical Sciences
Department of Electronic and Computer Engineering

Distributed Computing Systems Engineering

Master's Thesis

Conception and Realization of a Distributed and Automated Computer Vision Pipeline

---

March 31st, 2020

A Dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science

Abstract

This thesis if about implementing a distributed system to process large work items with specialized hardware requirements Docker containers are used to deploy the system and for job isolation. It furthermore assists the user by organizing the work items into projects, tracking execution progress and monitoring hardware utilization. All this is reported to the user in an Angular Web-Application which also allows the user to customize the execution pipeline, inspect intermediate results and to actively manage stage executions. The decentralized system detects and recovers from node failures and uses elections to select the target node for job executions. It is focused on easy setup and extendability.

Keywords: Synchronization, Distributed, Coordination, Events, Messaging, Software, Architecture, Docker, Spring Boot, REST, Angular, Typescript

Contents

1 Introduction **1**

1.1 MEC-View 3

1.2 Focus of this thesis 3

2 Aims and Objectives **4**

2.1 Current Workflow 4

2.2 Desired Workflow 6

2.3 Deliverable Requirements 7

2.3.1 Non-Requirements 8

3 Literature Survey **9**

3.1 Similar solutions 9

3.1.1 Hadoop MapReduce 9

3.1.2 Build Pipelines 11

3.1.3 Camunda 12

3.1.4 Nomad 12

3.1.5 dCache 13

3.1.6 Further mentions 13

3.2 Docker 16

3.2.1 Technology 16

3.3 Data Storage 17

3.3.1 Remote File System 18

3.4 Angular Web-Application and REST API 19

4 Introducing Winslow **20**

4.1 Common Terminology 20

5 System Analysis **22**

5.1 System Context 23

5.2 Use Case Diagrams 24

5.2.1 Managing Pipelines and Projects 24

5.2.2 Managing Resources and Workspaces 25

5.2.3 Managing and Monitoring Executions 26

5.2.4 Monitoring Nodes 26

5.2.5 System Administration 27

5.3 Detailed Use Case analysis 27

5.4 System Architecture 28

5.4.1 Layer 1: Orchestration Service 28

5.4.2 Layer 2: Events and Resources 29

5.4.3 Layer 3: Backend Driver 29

5.4.4 Layer 4: Client Communication 30

6 System design **31**

6.1 Environment 31

6.2 Storage Technology 32

6.3 Execution Management 34

6.4 Event Synchronization and Communication 35

6.4.1 Messages 36

6.4.2 Synchronization 38

6.5 Docker interface 40

6.6 Directory Structure and Organization 41

6.6.1 Winslow working directory 42

6.6.2 Stage storage 42

6.7 Job Scheduling and Election System 44
6.7.1 Affinity and Aversion 45
6.8 Execution 47
6.9 CPU, RAM and IO Monitoring 48
6.10 User Interface 48
6.11 Continuous Deployment 49

7 Outcome and Measurements **50**
7.1 What did not work as expected 50
7.2 User Authentication and Security 50
7.3 High Level System Overview 51
7.4 Failure resilience 52
7.5 Evaluation 53

8 Summary and Conclusion **58**
8.1 Further work 59

Bibliography **60**

A Project Management **66**

B Winslow Instance Installation **68**

C Node Status Information **71**

D Screenshots **73**

E Interim Report **79**

Chapter 1

Introduction

Since the industrial revolution, humans strive for more automation in the industry as well as in the every day life. What was at first a cost saving measurement in factories, now also is a differentiation method for products. A new product must prove a higher level comfort to the customer than the previous generation as well as all the competitors. As such, the ambitions of the industry are focused on increasing the value of their products for the customer.

The automotive industry is one of the prime examples of this. Never was traveling from one place to another as comfortable as nowadays. Aspects like an elegant interior design, comfortable seats, air conditioning, entertainment systems and safety measurements need to be considered by car manufacturers to be competitive these days. The next luxury enhancement will be the autonomously driving vehicle. No longer shall the owner of a car steer it, but instead the car becomes his or hers personal chauffeur, driving the optimal route, the most comfortable way and being more reliable and safer than any human ever could.

The reason, autonomously driving cars are not common already, is their big complexity increase. Compared to already established technologies like parking assistants, entertainment systems or more efficient engine controllers, letting a computer reliably understand a certain traffic situation requires masses of input data and complex algorithms to process. As such, the problem itself becomes

massive and cannot be solved that easily. So the industry has no choice than to divide this into many small pieces and work out solutions step by step.

The MEC-View research project explorers one such step: whether and how to include external, steady mounted sensors in the decision finding process for partially autonomous vehicles in situations where onboard sensors are insufficient. To not disrupt traffic flow with non-human behavior, one needs to study and thereby watch human traffic. Automatically analyzing traffic from video footage requires a lot of computation power and can be further optimized by specialized hardware such as GPUs[1].

This thesis will conceptualize and realize a distributed and automated computer vision pipeline which is in this case used to analyzes traffic flow within video footage. Compared to an existing but highly manual workflow, the new system shall help to utilize the available hardware more efficiently by reducing idle times. Stage transitions and basic scheduling shall be automated to allow a user to plan and execute multiple projects ahead of time and in parallel.

[1]Graphics Processing Units

1.1 MEC-View

The MEC-View research project[1] - funded by the German Federal Ministry for Economic Affairs and Energy - aims to supplement the field of view of automated driving cars with road-side sensor data using 5G mobile communication. The sensor information is merged into an environment model on the so-called Mobile Edge Computing (MEC) server. This server is directly attached to the radio station to ensure low latency environment model updates.

The project is tested at an intersection in Ulm, Germany. Currently, there are 15 lidar and video sensors installed. Those sensors send their detections to the (MEC) server. A fusion-algorithm merges those detections into one environment model and sends it back to the (MEC) server and to the automated cars.

Additionally, general traffic flow is analyzed to learn about movement patterns. To do so, 4k video data is captured by an air drone from real world cross roads - not limited to the intersection in Ulm. On each frame of such a recording, cars are detected with a neuronal network. Detected cars are tracked throughout the video to compute the movement speed and position in time of each car. In an analysis of all vehicles, hot-spots of high and low traffic flow can be determined.

1.2 Focus of this thesis

This thesis conceptualizes and implements the distribution and automatization of a computer vision pipeline to increase the productivity in video analysis. It is not of concern for this thesis on how to retrieve the footage or what is further archived with the results of the processed video footage. The focus is on utilizing available hardware resources, to manage multiple projects simultaneously and to reduce the idle time of relevant hardware by slow human reaction and availability times - like working hour constraints. An easy setup and low maintenance is also desirable.

Chapter 2

Aims and Objectives

This chapter will discuss the program which shall be implemented. To do so, the problem to solve must be understood. To gather requirements and understand the technical hurdles to overcome, this chapter is split into two sections. First, a rough glance over the current workflow is given, which is followed by a more detailed description for the desired workflow.

2.1 Current Workflow

Currently, to analyze a video for the trajectories of recorded vehicles, the following steps are executed manually:

1. Upload the input video to a new directory on the GPU server
2. Execute a shell script with the video as input file and let it run (hours to days) until completed. The shell script invokes a Java Program - called TrackerApplication - with parameters on what to do with the input file and additional parameters.
3. The intermediate result with raw detection results is downloaded to the local machine and opened for inspection. If the detection error is too high,

the camera tracking has a drift or other disruptions are visible, the previous step is redone with adjusted parameters.

4. Upload the video and intermediate result to a generic computing server and run data cleanup and analysis. This is achieved with the same Java Program as in step 2, but with different stage environment parameters.

5. Download the results, recheck for consistency or obvious abnormalities. Depending on the result, redo step 2 or 4 with adjusted parameters again.

6. Depending on the assignment, steps 4 and 5 are repeated to incrementally accumulate all output data (such as statistics, diagrams and so on).

Because all those steps are done manually, the user needs to check for errors by oneself. Also, if a execution is finished or has failed early, there could be hours wasted until noticed, if the check intervals are too far apart, such as during nights or weekends. The current approach does not scale at all for the increasing amount of projects that need to be processed. Manually keeping track of all project states and intermediate results is tedious and has proven to be error prone.

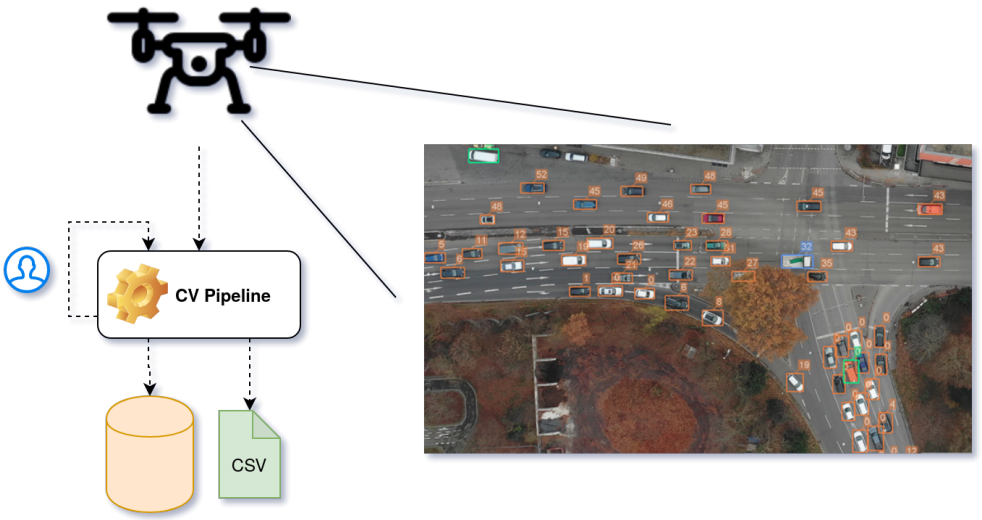

Figure 2.1: Overview of workflow

2.2 Desired Workflow

The desired workflow shall be supported through an user interface that provides an overview of all active projects and their current state, such as running computation, awaiting user input, failed or succeeded.

To create a new project, a predefined pipeline definition shall be selected as well as a name chosen. Because only a handful of different pipeline definitions are expected, the creation of such does not need to happen through the user interface. Instead, it is acceptable to have to manually edit a configuration file in such rare circumstances.

Once a project is created, the user wants to select the path to the input video. This file has to be been uploaded to a global resource pool at this point. The upload and download of files shall therefore also be possible through the user interface. Because a video is usually recorded in 4k (3840 x 2160 pixels), encoded with H.264 and up to 20 minutes long, the upload must be capable of handling files which are tens of gigabytes large.

Once a pipeline is started, it shall execute the stages on the most fitting server node until finished, failed or a user input is required. Throughout, the logs of the current and previous stage shall be accessible as well as uploading or downloading files from the current or previous stages workspace. In addition to the pipeline pausing itself for user input, the user shall be able to request the pipeline to pause after the current stage at any moment. When resuming the pipeline, the user might want to overwrite the starting point to, for example, redo the latest stage.

Mechanisms for fault tolerance shall detect unexpected program errors or failures of server nodes. Server nodes shall be easily installed and added to the existing network of server nodes. Each server node might provide additional hardware (such as GPUs), which shall be detected and provided.

For the ease of installation and binary distribution, Docker Images shall be used for running the Java Program for analyzing the videos as well the to be implemented management software.

2.3 Deliverable Requirements

From the desired workflow, the following requirements can be extracted:

- A user interface for interaction between the system and the user
- Storage management for global resource files as well as stage based workspaces
- Pipeline definition through configuration files
- Handling of multiple projects with independent progress and environment
- Reflecting the correct project state (running, failed, succeeded, paused)
- Log accumulation and archiving
- Accepting user input to update environment variables, resuming and pausing projects as well as uploading and downloading files into or from the global resource pool or a stages workspace.
- Assigning starting stages to the most fitting server node
- Detecting program errors (in a stage execution)
- Cope with node failures
- Using Docker Images as installation medium

Further non-functional requirements are, that the user interface is to be implemented as Angular Web-Application using a REST API (for more details see section 3.4) for data transfer. The system shall increase the productivity and hardware utilization, especially in times where the staff is absence. The need to constantly monitor and interfere with the system to progress projects shall be decreased by providing automated mechanisms where possible and appropriate.

2.3.1 Non-Requirements

To know the requirements and expectations of a system is essential, but knowing what is not expected by the system is at least as valuable. It prevents wasting resources, efforts and architectural specializations that will never be required or in the worst case, make further development harder by restricting available choices for the future.

The system to implement shall not strive to implement real time scheduling or low latency scheduling. The expected work items are big chunks that require hours to compute, whether the assignment of the work item takes sub-seconds or several seconds is nearly unnoticeable in the overall compute time.

Chapter 3

Literature Survey

This chapter is about accumulating information. Programs that are solving similar problems, as described in the desired workflow, or dealing with a subset of the problem are looked into. Fundamental knowledge for understanding this thesis is also acquired here.

3.1 Similar solutions

This sections focuses on programs trying to provide somewhat similar workflows. The reason for this is to use well established or suitable programs as middle-ware to reduce implementation overhead, or, where this is not possible, one might be able to gather ideas and learn about proven strategies to use or pitfalls to avoid while implementing custom solutions.

3.1.1 Hadoop MapReduce

For big data transformation, Hadoop MapReduce[2] is well known. With MapReduce, the input data is split into blocks of data and distributed onto Mappers. Mappers then apply the business logic and output intermediate results in form of key/value pairs. After shuffling, the Reduce stage will combine values from key/-value pairs with the same key. In the final output, each key is unique accompanied

with a value.

This strategy has proven to be very powerful to process large amount input data because Mappers and Reducers can work independently on their data-sets and therefore scale very well when adding further instances.

If the implementation were to be based on Hadoop MapReduce to achieve the desired workflow, it could be done like the following:

- Each video is split into many frames and each frame is applied to a Mapper
- A Mapper tries to detect all vehicles on a frame and outputs their position, orientation, size and so on
- The Reducer then tries to link the detections of a vehicle through multiple frames
- The final result would be a set of detections and therefore all positions for each vehicle in the video

But at the moment, this approach seems to be unfitting due to at least the following reasons:

1. It is not always trivial to reasonable link the detections of a vehicle. For example, a vehicle can be hidden behind a tree for a few frames until visible again. In addition, MapReduce requires the combination to be performed per common key. Until one is trying to link the detections of multiple frames, there is no common identifier that could easily be used as key. The position of a moving vehicle cannot be used as key, neither can the color or size, because of the noise of the camera, deviation in detection output and perspective distortions. The current implementation of the TrackerApplication is archiving this by finding similarities between detections, but for the Mapper it would be required to express this as a deterministic key.
2. MapReduce is great in combining many machines to solve a big computational problem. But at the moment, this is neither a desired nor given

condition. At the moment, there are a handful of very powerful workstations with specialized hardware. Therefore it is perfectly acceptable and sometimes required, when each workstation works through a complete video at a given time instead.

3.1.2 Build Pipelines

Build pipelines such as GitLab[3] and Jenkins[4] can also distribute the execution of stages onto other server nodes. In a common use-case, such build pipelines are used to build binaries out of source code, after a new commit into a SCM[1] repository was made. At IT-Designers GmbH GitLab as well as Jenkins are commonly used for scenarios exactly like this. A pipeline definition in GitLab CI/CD [5] or in a Jenkinsfile [6] describe stages and commands to execute. Each stage can be hosted on another node and be executed sequential or in parallel to each other.

Although this seems to be quite fitting for the desired workflow, there are two issues. First of all, such a pipeline does not involve any user input besides an optional manual start invocation. The result is then determined based on the state of the input repository. Second, such a pipeline is designed to determine the output (usually by compiling) whereas each run is independent from the previous and a repeated run shall provide the same result as the previous did. Usually, a new run is only caused by a change of the input data. However, the desired workflow differs in this aspects. A redo of a stage can depend on the result of the previous stage, for example, if the results are poor or the the stage failed. Instead of having multiple complete pipeline runs per project, the desired workflow uses a pipeline definition as base for which the order can be changed. Also, intermediate results need to influence further stages, even if repeated.

[1] Source Code Management

3.1.3 Camunda

Camunda[7] calls itself a "Rich Business Process Management tool" and allows the user to easily create new pipelines by combining existing tasks with many triggers and custom transitions. Camunda is focused upon visualizing the flow and tracking the data through a pipeline. The Camundas Process Engine[8] also allows user intervention between tasks.

One of the main supporting reason for it Camunda is the out of the box rich graphical user interface for process definition and interaction. Through its API[9], Camunda also allows custom external workers to execute a task. But it misses the capability to control which task shall be processed on which worker node which is required by the desired workflow. It does also not provide any concept on how to allocate and distribute resources. The user interface - while being rich overall - is quite rudimentary when it is about configuring tasks and would therefore require custom plugins to be developed for more advanced user interactions.

Camunda is also not designed to reorder stages or insert user interactions at seemingly random fashion. The user itself is considered more as a worker that gets some request, "executes" this externally and finally marks the request as accepted or declined. Mapping this to the desired workflow does not feel intuitive. Finally, there is also no overview of task executors, no centralized log accumulation and no file up- or download for global project resources.

3.1.4 Nomad

Nomad[10] by HashiCorp is a tool to deploy, manage and monitor containers, whereas each job is executed in its own container. It provides a rich REST API and can consider hardware constraints on job submissions. Compared to Kubernetes[11], which is similar but more focused on scaling containers to an externally applied load, it is very lightweight. It is also available in many Linux software repositories - such as for Debian - which makes the installation very easy.

Because there were no grave disadvantages found (depending on a third party

library can always be considered be a disadvantage for flexibility, error-pronous and limit functionality) Nomad is being considered as a middle-ware to manage and deploy stages. Others[12] seem to be using Nomad to manage and deploy containers for similar reasons. Nonetheless, further testing and prototyping will be required for a final decision.

3.1.5 dCache

"The goal of this project is to provide a system for storing and retrieving huge amounts of data, distributed among a large number of heterogenous server nodes, under a single virtual filesystem tree with a variety of standard access methods"[13]. dCache seems to be able to solve the storage access and distribution concern for the stages and sever nodes. When using dCache, one could store the global resources distributed between the server nodes. Built-in replication would prevent access loss on a node or network failure and an export through NFS[2] allows easy access for Linux based systems[14].

But the installation is complex and requires many services to be setup correctly, such as postgresql and many internal services such as zookeeper, admin, poolmanager, spacemanager, pnfsmanager, cleaner, gplazma, pinmanager, topo, info and nfs. The documentation is also rather outdated and incomplete which meant, early tests with a prototype setup took days to setup and behaved rather unstable (probably due to a wrong configuration). It is to be seen, whether such an complex and heavy system is actually required or if there are feasible alternatives.

3.1.6 Further mentions

The following list acknowledges programs that behave similar to the previously mentioned strategies or can be used as building blocks for custom implementations. Programs that are listed here, were looked into, but not all in great depth

[2]Network File System

because miss-fits or missing functionality were detected early on. The list is in no specific order:

- **Quartz**[15] is a Java based program to schedule jobs. Instead of doing so by using input, Quartz executes programs through a timetable and in certain intervals.

- **Luigi**[16] also executes pipelines with stages and is written in python. The advertised advantage is to define the pipeline directly in python code. But, this is at the same time the only way to define pipelines which contradictions with the existing Java TrackerApplication implementation.

- **Celery**[17] is focused on task execution through message passing and is written in Python. Intermediate results are expected to be transmitted through messages. Because there is no storage strategy and python adapter-code would have been required, Celery was dismissed.

- **IBM InfoSphere**[18] provides similar to Camunda a rich graphical user interface but for data transfer. Dismissed due to commercial nature.

- **qsub**[19][20] is a CLI[3] used in HPC to submit jobs onto a cluster or grid. Dismissed due to an expected high setup overhead, non-required multi-user nature and the fact, that it only provides a way to submit jobs.

- **CSCS**[21] High Throughput Scheduler (GREASY). Dismissed for similar reasons as qsub, although it is more light weight and hardware agnostic (it can consider CUDA/GPU requirements).

- **zsync**[22], similar to rsync, is a file transfer program. Zsync allows to only transfer new parts when a file that shall be copied already exists in an older version on the target. This tool might be useful when implementing a custom resource distribution strategy is required.

[3]Command Line Interface

- **OpenIO**[23] provides a distributed file system, is already provided as Docker image and provides a simple to use CLI. Because the NFS export is only available through a paid subscription plan, it was dismissed from further investigation.

- **SeaweedFS**[24] provides a scalable and distributed file system. The most interesting aspects are that it is rack-aware as well as natively supports external storage such as Amazon S3. When adding server nodes from the cloud this could allow all nodes to access the same file system while using rack-aware replication to reduce bandwidth usage and latency. A local test also proved that it is easy to setup, but because it cannot hot-swap nodes and was not able to recover when the seeweed master node became unreachable it was dismissed.

- **Alluxio**[25] provides a distributed file system but was dismissed because it itself requires a centralized file system for the master and its fallback instances

- **GlusterFS**[26] is another tool to provide a distributed file system with replication. It was bought by IBM but is nonetheless available through the software repository of many Linux distributions such as Debian. A local test showed that the setup is very easy and no adjustments of configuration files are required. However, the replication mechanism requires that an integer multiple of nodes of the replica value are assigned to the file system. This makes GlusterFS hard to use in a scenario, where adding and removing nodes are expected to happen frequently.

3.2 Docker

As describe in section 2.2, for easy deployment and setup Docker[27] images shall be used. This allows isolation of the executed work items and other host programs, as will be explain further on.

Docker is the name of a software that combines various isolation technologies to provide and execute third party software in virtual environments. Docker aims are to increase security and to simplify installation as well as maintenance of applications. Docker Swarm[29] and Kubernetes[30] use these fundamental features to scale applications, services and microservices in the cloud. Software is retrieved from a public registry, called Docker Hub[31] or from local or private registries.

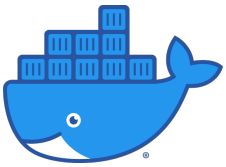

Figure 3.1: Official Docker "Moby" Logo[28]

3.2.1 Technology

Docker uses so called images to package and transport binaries with all their required libraries and configuration files as a read-only archive to the destination system. Instead of spawning a new process for a binary in the host environment, the image is used to create a new virtual environment - the so called container. Because the image includes all required libraries, it is possible to completely isolate the virtual environment which makes the host system invisible to any process inside the container. Changes to files within containers are stored separately as differentials[4], which allows an image to be used by multiple containers at once. Privileges, resource limitations and storage configurations can also be specified when creating a new container. Processes inside containers are unable to see

[4]for example by using OverlayFS[32]

other processes or files that are not part of or assigned to their container[5].

In contrary to a hypervisor, docker is archiving this without hardware and operating system virtualization. Instead of executing the container inside an additional virtualized operating system, they share the kernel of the host system. For that, the host needs to support additional isolation mechanisms At the time of writing this, only the Linux Kernel is capable to separate processes, network interfaces, interprocess communications, filesystem mounts and the time-sharing systems by namespaces. By configuring these namespaces, Docker is capable to isolate containers into virtual environments. Furthermore, control groups can be used to limit and constraint access to hardware resources. [33]

These approaches allow containers to run with very little additional overhead in comparison to running the application directly on the host. Containerization increases security by limiting what an application sees and is able to interact with, decreases maintenance overhead because of no additional operation systems to maintain and allows to run multiple instances of the same application besides each other with independent configurations and environments but with the same base image.

3.3 Data Storage

When distributing workload onto different machines, accessing input and output files becomes another concern which is not present when all computation and storage resides on the same physical machine. In theory, one could use portable mediums such as USB-Sticks, CD-ROMs or external HDDs, but in reality, this becomes very tedious really fast. More advanced users might be able to take advantage of a common network connection between the computers to copy files and directories to the required places. This strategy - still being tedious - surfaces another issue: dealing with multiple copies requires careful version management and

[5]This is the default behaviour. It is possible to manually lift or modify many boundaries Docker enforces for containers per default.

additional storage space. One certainly would not want to continue computation on outdated data set or have multiple copies of the same files.

The solution to that could be a network file system. To programs this solution seems to be just another local directory hierarchy, but in reality, the files might be located on another or multiple other machines.

3.3.1 Remote File System

Remote file systems provide access to data that does not need to be stored on the local machine and can often be shared with other clients as well. Some file systems are organized centralized, where at a single address and physical location all data is stored and retrieved from. This can have advantages in setup time, maintenance and communication complexity, because of their simplicity.

The Network File Systems (NFS) for example, which is primarily used in the Linux and Unix world, is centralized. The clients are connecting to a remote server to read and write files from and to. Since NFS 4.1 pNFS[6][34, p. 14, section 1.7.2.2] allows the server to spread read and write load across multiple servers and volumes, but because all metadata is still handled by one server, this approach still has a single point of failure. The Common Internet File System (SMB/CIFS) is similar to NFS in regards of providing access to a remote file system locally, and it is the approach primarily used by Windows computers. Similar to NFS, it has also a centralized approach.

More advanced file systems provide additional functionality like replication, which stores the data at more than one location to recover from hardware failures, take advantage of site awareness - which replicates data to geographically distant locations[7] - and decentralized communication endpoints that provide access to the data from more than a single point of failure.

File systems like GlusterFS (subsection 3.1.6), Alluxio (subsection 3.1.6), Sea-

[6]parallel NFS

[7]to not be affected by a burning data centre for example or to decrease access time for clients from the distant location

weedFS (subsection 3.1.6), OpenIO (subsection 3.1.6), dCache (subsection 3.1.5) and HadoopFS (subsection 3.1.1) try to provide these functionality.

3.4 Angular Web-Application and REST API

The user interface is supposed to be implemented by an Angular Web-Application. Angular[35] itself is a framework for mobile and desktop Web-Applications which uses templating to generate the website displayed to the user completely on the client side. It uses TypeScript[36], which is, in essence, an enriched subset of JavaScript with type annotations, that compiles back to JavaScript. This allows any recent web-browser to execute "TypeScript code". The additional type annotation in TypeScript supports developers by giving helpful error messages, instead of more obscure runtime errors.

This thesis uses REST (Representation State Transfer) to enable the Angular application to retrieve and push data to web-server. REST is by now commonly used to transfer state on top of HTTP in the internet. It utilizes the HTTP methods[37] communicate to request type and URLs to localize resources.

Chapter 4

Introducing Winslow

The program that shall implement the requested features is being called Winslow. This name refers to Frederick Winslow Taylor who was an American mechanical engineer and one of the first management consultants in the 19th century[38][39]. The term "taylorism" refers to his work[40]. Both strive to make people's work more efficient.

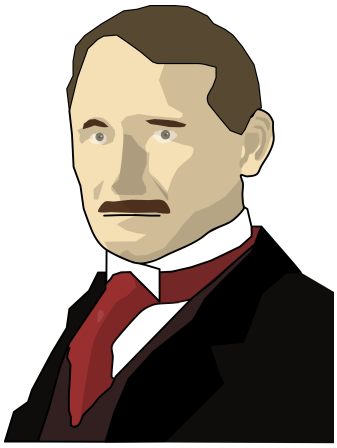

Figure 4.1: Winslow Logo

4.1 Common Terminology

This chapter explains the meaning of words and terminology used later on. It is crucial that every reader has the same understanding for the words used so that there are no wrong assumptions, expectations or surprises.

The root of a work item is called **project**. A project will usually refer to one video footage that shall be processed. Each project has its exclusive workspace for input, output and intermediate files. Furthermore a project has an author and possibly participants, that can help in steering and monitoring the processing. To do so, there is always a pipeline assigned to a project.

A **pipeline** consists of at least one but usually multiple stages that can at least be processed linearly - one after the other. Furthermore, a pipeline can define environment variables that are valid for all stages within the pipeline.

A **stage** is the smallest work unit that can be executed. A docker image (see section 3.2) is specified as execution environment as well as further stage specific environment variables, command line arguments and hardware requirements (such as CPU cores, GPUs and minimum available RAM). This allows stages to be based on common images but also to specify very precisely how to process the data in a certain scenario.

In addition, stages can be categorized in running, succeeded or failed while pipelines can be have active or paused as state. A **stage definition** specifies the above mentioned presets in a template to base multiple stages from, while a running or complete stage has all information of what is or was executed and allows no further alteration. For **pipeline definitions** this is similar, it can be used to specify a common order of execution. Once assigned to a project, it can freely adjust and change its copy of the pipeline.

The container process that is executing the binary of Winslow is called **Winslow instance**. A computer with a Winslow instance that is accepting or executing stages is called **execution node** with focus on the computer providing compute resources to Winslow.

Chapter 5

System Analysis

This chapter is about analysing the to be implemented system. It is important to find an architecture that is able to fulfil all requirements and that is able to be evolved for upcoming needs. But, as the YAGNI ("You Ain't Gonna Need It" [41, p. 36]) principle describes, this does not mean one should plan for speculative events. In some areas where changes or extensions are foreseeable, preparations can be considered, whereas if there is no such indication, one should only consider the actual and current need. In addition, keeping the SOLID[1][42, p. 86] principles in mind can help to improve correctness, simplicity, stability, changeability and testability[41, p. 162].

The analysis and design in this thesis follows the "Top-Down" approach[43, p. 171] in which the system is first generally outlined and then step by step partitioned into smaller and more detailed pieces.

[1] "**S**ingle-responsibility principle", "**O**open-close principle", "**L**iskov-substitution principle", "**I**nterface segregation principle" and "**D**ependency-inversion principle"

5.1 System Context

A System Context Diagram is suitable to find the boundaries of the system to implement and interactions with external components. The first step was therefore to create Figure 5.1. The notation is a slight deviation from the UML standard so that the control and data flow can also be displayed [43, p. 501].

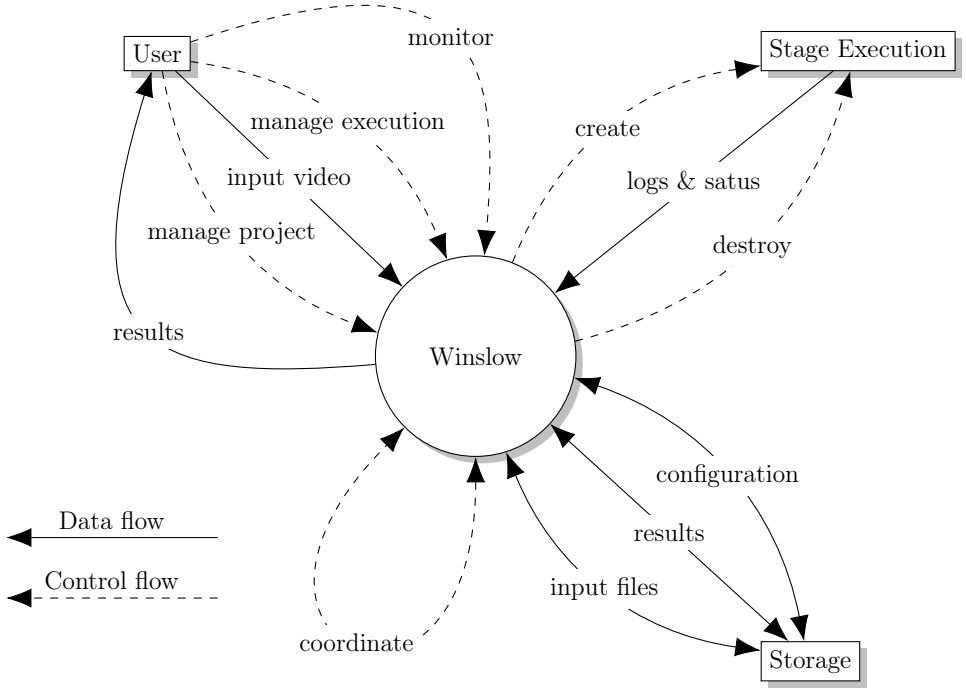

Figure 5.1: System Context Diagram

This thesis will not focus on all parts mentioned in Figure 5.1, but for the implementation it was essential to have an as complete overview as possible.

5.2 Use Case Diagrams

A System Context Diagram is not suitable to elaborate all interactions in great detail, but a Use Case Diagrams can be used for that. It helps to understand the customers needs [44] while the customer receives an impression on what will be reflected in the final product. Each but the very first of the following diagrams focuses on one actor that is going to interact with the system. Figure 5.2 shows a high-level use case overview of all categories that are relevant to users of the system:

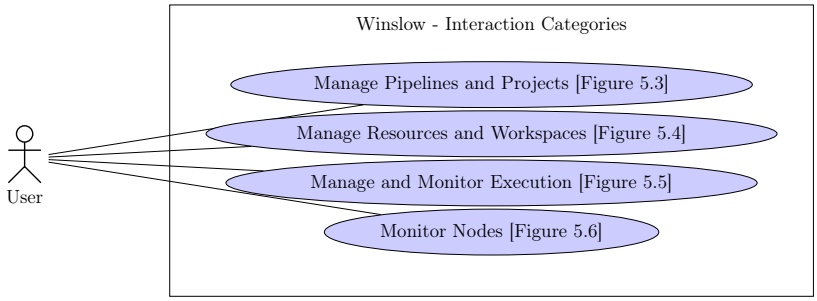

Figure 5.2: Use Case Diagram showing an high level overview of interactions

5.2.1 Managing Pipelines and Projects

One very important concern to the user of Winslow is to create an manage pipelines and to associate them to projects. The customer expressed the need to also attach tags to projects and to filter the overview by them. This does not affect any execution strategy of Winslow, but regardless, it is still important to the end user.

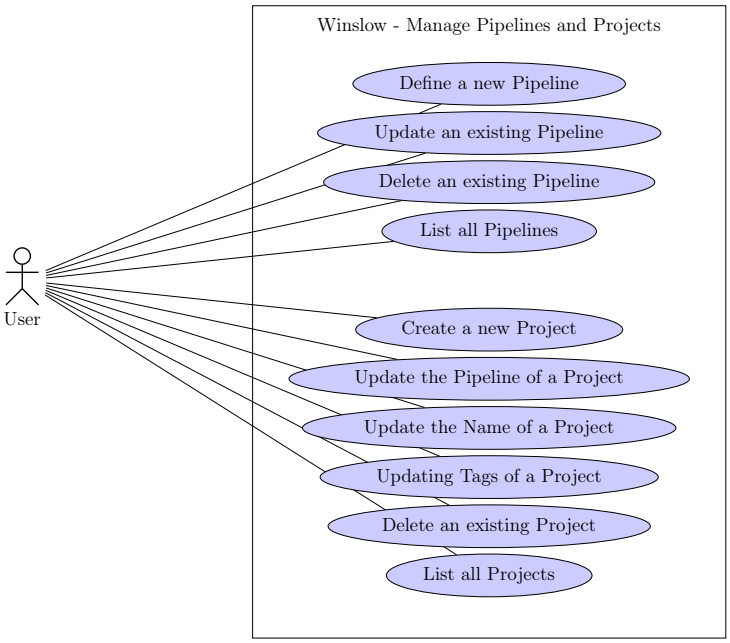

Figure 5.3: Use Case Diagram showing the pipeline and project management interactions

5.2.2 Managing Resources and Workspaces

Uploading, listing and downloading files is one of the main concerns of Winslow to enable stage executions, so it is to no surprise that this is important for the user as well.

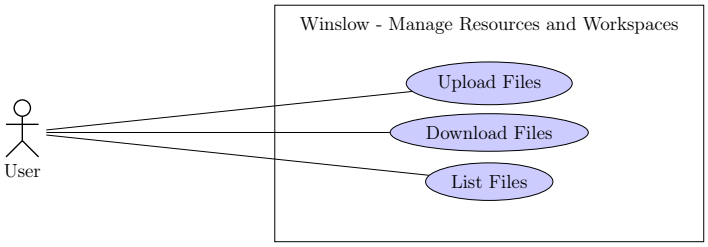

Figure 5.4: Use Case Diagram showing the resource management interactions

5.2.3 Managing and Monitoring Executions

The main goal of the system is stage execution. The user expressed interest in advanced control, such as aborting a running stage or pausing the stage execution of a project, as well as being able to view live logs and inspecting intermediate output files .

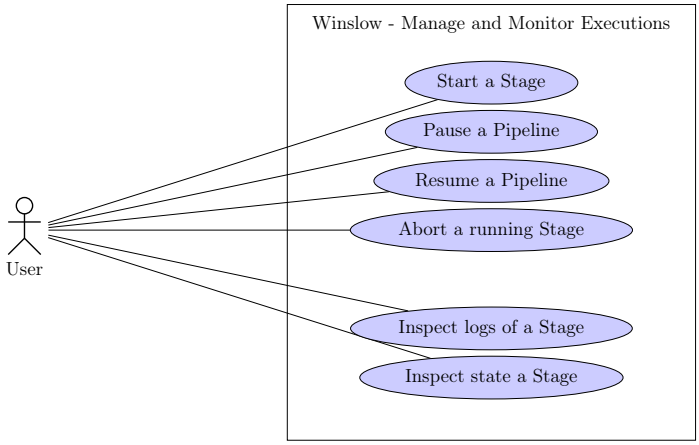

Figure 5.5: Use Case Diagram showing the control interactions

5.2.4 Monitoring Nodes

To monitor the usage of all nodes, an overview listing CPU usage, RAM usage and Network IO for each node was requested.

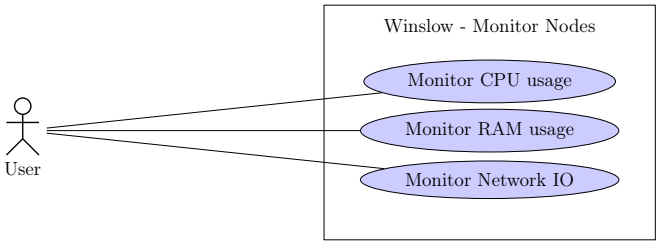

Figure 5.6: Use Case Diagram showing the monitor needs

5.2.5 System Administration

Furthermore, to the interactions with a general user, the system must provide further capabilities that are of concern to the administrator maintaining and installing Winslow. The wish for an easy setup and low maintenance overhead for Winslow was raised here. It should be easy to add a new installation to an existing group of instances.

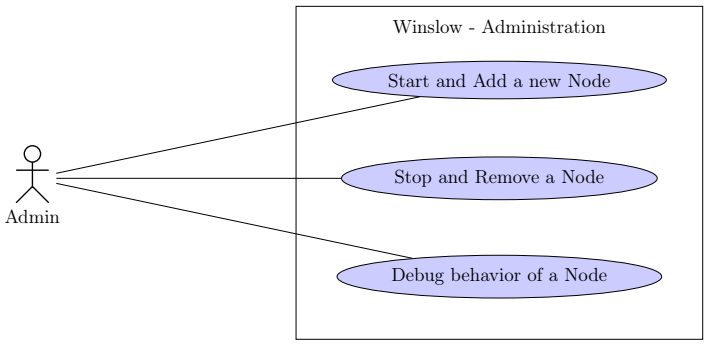

Figure 5.7: Use Case Diagram showing administrative interactions

5.3 Detailed Use Case analysis

The next step would be to write a detailed analysis for each use case, including information such as who is the initiator, which components are involved, what is the expected and what are alternative outcomes. But that would create further pages, with a lot of details for user interaction, which is not the focus of this thesis. For that reason, is omitted here.

5.4 System Architecture

In Figure 5.8 a high level overview of the architecture is shown. It shows various services and regions with internal and external communication channels. The architecture follows a simplified[2] "Onion Architecture Pattern"[45], which is indicated by the layer numbers and colouring (inner layers are darker). For better readability, it is not arranged in circles.

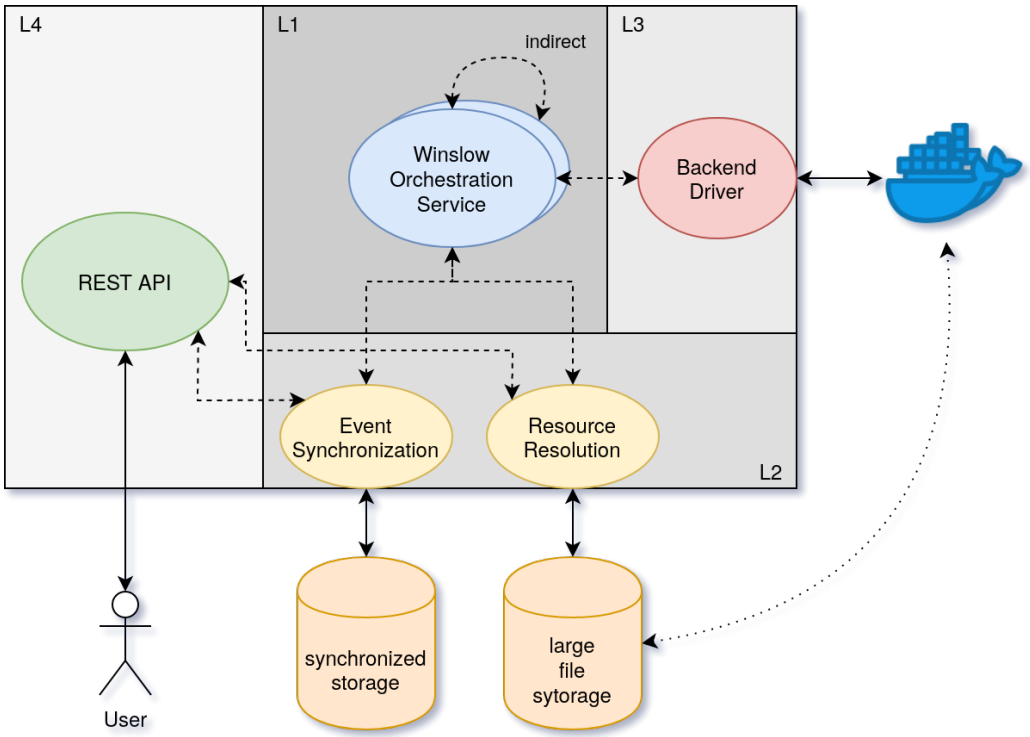

Figure 5.8: High level architecture overview of Winslow

The meaning of the layers and their services will be explained in the following sub-sections.

5.4.1 Layer 1: Orchestration Service

This layer is all about the fundamental business logic: when to schedule, start and abort which stage of what project a well as tracking the hardware utilization.

[2]fewer, collapsed layers

This level is also responsible to communicate these actions with all other Winslow instances, so that there is no duplicate or missing stage execution or undetected node failure.

5.4.2 Layer 2: Events and Resources

The second layer is essential for the first layer to interact with and understand its environment. It provides two important services: a synchronized communication channel and a large file storage.

The communication channel requires a storage to persist messages that are relevant for a duration of time. It also needs to support basic synchronization primitives to ensure consistency. Winslow instances that have just been started need to see events that were issued before their start and which are still relevant to be able to replicate the system state correctly.

For executing stages, the system must be able to store large files. The synchronized access can be ensured the event synchronization and thus the constraints to this storage is more relaxed than the one used to persist synchronization messages.

5.4.3 Layer 3: Backend Driver

This layer is responsible to interface with Docker. Once stage shall be executed, this driver is instructed to start a certain image with environment variables and the path to a prepared workspace. By separating this task to its own layer it will be easier to change the Docker API for another implementation or to move away from the Docker platform at all if necessary. A reason for this could be the change of needs, the appearance of a platform that is fulfilling the needs better or the disappearance of the currently used platform. The latter sounds not that common at first, but the recent partial acquisition of Docker Inc by Mirantis[46] proves that even very popular third party software is not going to be around for all eternity. By moving the driver implementation into its own layer, but leaving the interface definition in layer 1, the driver implementation can be

replaced easily without modifications to the inner layer. It then also complies to the "Dependency Inversion Principle"[41, p. 65].

While this driver is not supposed to access any storage itself, the stages need to read and write to the large file storage. Docker does support mounting local directories or remote NFS shares as volumes into the container natively, access from within the executed stage can then also be guarded to limit read and write access to only the required working directories.

5.4.4 Layer 4: Client Communication

The final layer in this architecture is the client communication layer. This service is not crucial for the actual execution and may be disabled on Winslow instances that shall not respond to a user requests. A reason for this could be, that SSL certificates are bound to sub-domain which might point to specific Winslow instances and any unsecure access is not desired. The REST API in this layer shall provide resources that can be accessed from the statically served Angular Web-Application[3] or from future tools to upload or download files, to control stage execution and to monitor usage and logs.

[3] see section 3.4

Chapter 6

System design

In this chapter the previously collected information is used to design the system in detail. The decision making process for that is documented here as well.

6.1 Environment

All Winslow instances are supposed to run inside a Docker container (as requested in section 2.2). The hosts will be Ubuntu or Debian Linux Servers of which some will have one or multiple GPUs. There will be no graphical user interface available on these servers.

The decision to implement Winslow in Java was made early on. This has three main reasons, first, the developer is experience in Java development, the server side web framework used is a Java library (see section 6.10) and the company this implementation is for has a focus on Java development. This means if the system is successful, maintenance and possible extensions might be authored by another developer who has probably knowledge in Java development as well.

6.2 Storage Technology

Organizing and accessing data is one of the main concerns for Winslow. In almost all use cases, multiple instances will need to access the storage. It must therefore be accessible from remote and by multiple Winslow instances simultaneously. For that, the following technologies are reviewed: NFS or SMB/CIFS (subsection 3.3.1), GlusterFS (subsection 3.1.6), SeaweedFS (subsection 3.1.6), OpenIO (subsection 3.1.6), dCache (subsection 3.1.5) and HadoopFS (subsection 3.1.1).

The following criteria were used for comparison. The scoring system provides points from 0 to 3, in which 0 means the best solution and 3 the worst. The points of the individual categories are added together, the lowest overall score shall then point to the best solution:

- **Installation Overhead**: How hard and extensive is the installation? Is there a good documentation? Is the package provided from within the main repository of Ubuntu, Debian, or alternatively, does it provide any other form of easy installation? Scoring 0 for installation through the main repository, 1 for an external repository, 2 for a distribution specific archive, 3 for any other installation solution.

- **Dependencies**: Are further services required for operation? How hard are they to setup? Do they need any additional configuration? Scoring 0 for no further dependencies, 1 for dependency requirements but without the need of manual configuration, 2 for dependencies that require manual configuration, 3 for dependencies that need to be installed manually and require manual configuration.

- **Single Point of Failure**: Is the solution decentralized and is it failure resilient? Is it site- or rack-aware or provide replication mechanisms to compensate? Scoring 0 for completely decentralized, 1 for no direct SPoF but with a dependency on a centralized backend, 2 for a mostly centralized

solution but with load balancing approaches, 3 for no single point of failure mitigations.

- **Docker integration**: How easy is it to integrate with Docker? Scoring 0 for native support, 1 for native but non-trivial support, 2 for support through additional exports facility, 3 for non-trivial solution.

- **Failure concerns**: Are there any noteworthy concerns? Was an early local test successful and reliable? Is it a known technology or "proven in use"? Who is developing it and what are the support guarantees? Scoring 0 for commonly used and no concerns, 1 for minor concerns, 2 for major uncertainty, 3 for abandoned technology.

In Table 6.1 the results are displayed:

	NFS S.	SMB/CIFS S.	GlusterFS	SeaweedFs	dCache	HDFS
Inst.	0	1	0	2	3	1
Dep.	1	0	0	0	3	1
SPoF	2	3	0	2	0	0
Docker	0	2	2	3	2	3
Fail.	0	0	1	2	2	0
Score	3	6	3	9	10	5

Table 6.1: Failure concern comparison of storage technologies

The worst scoring tool in this comparison is dCache. The installation overhead, missing documentation and uncertainty on reliable operation is too high for this project (see subsection 3.1.5).

For similar reasons SeaweedFs is placed second worst. Local tests could not access SeaweedFs reliable when a node failed and there is no trivial support for Docker nor does it provide an NFS export. Using it would require each started container to be manipulated so that the first operation is mounting the

storage with a custom binary and through a FUSE[1] mount. The tool also seems to be developed by a single person which introduces further uncertainty about reliability and support in the future.

HDFS and SMB/CIFS seem to be no terrible choice, but no especially good one either in this use case.

The two best scoring storage solutions are a simple NFS share and GlusterFS. While GlusterFS provides replication and decentralized access, a NFS share scores with its simplicity. The idea is to start with a plain NFS share for Winslow which can natively be utilized by Docker as volume mount and to revisit later whether the need for replication and decentralization persists. Because of the NFS export feature of GlusterFS, Winslow could then easily switch to GlusterFS.

6.3 Execution Management

As noted in subsection 5.4.1, the central business logic is to decide on when and to issue stage executions. Generally speaking, there are two approaches when executing jobs: local or remote. Both will be discussed next.

In the remote approach, the job is executed on a machine that is not the same that has the responsibility to manage the process. Continuous Integration (CI) platform Jenkins[4] does use this approach. A so called slave node (Jenkins) is contacted through a native interface (SSH for Linux servers) to transfer resources and to start the job process. In this scenario, the CI instance requires and stores login credentials for every remote machine to be able to login whenever needed. The system administrator has to create a new user accounts on the remote machines, install required programs and prepare the environments. One big risk for this approach is that in case of any security breach on the central CI instance, the attacker is also able to login on all remote machines.

GitLab[3] follows a more local approach, where the system administrator has

[1] Filesystem in Userspace

to manually install the GitLab Runner on machines that are then able to connect to GitLab and execute jobs. This runner is responsible in pulling jobs, executing them locally, monitoring and reporting back the outcome.

Winslow is going to follow the local approach. On each physical machine that is supposed to provide compute resources, Winslow needs to be installed. The installation is expected to be easy, as is based on starting a prepared Docker image. An election between the Winslow instances is selecting the most fitting hardware for a stage execution (see affinity and aversion in subsection 6.7.1). It is expected that each instance can judge this best on its own, as it knows which resources are available. Work is distributed to an executor as it is detected.

6.4 Event Synchronization and Communication

As discovered in subsection 5.4.2, there is the need for event synchronization across all Winslow instances. Without proper coordination, race conditions could cause stages to be started multiple times simultaneously, corrupt workspaces, configuration or project files. While starting too many stage executions are only wasting resources, data corruptions can lead to unrecoverable damage.

There are multiple ways to exchange data between systems that do not share the same process or machine. The most flexible implementation can be achieved by implementing a custom protocol on a raw TCP socket which allows to exchange blobs (Binary Large Objects) between exactly two nodes. For a blob or message to reach all nodes, a connection to every other node, a centralized broker, another topology such as a tree structure or broadcast messages would be required. Because this in itself seems to be a complex subject, third party alternatives were investigated.

Apache Kafka[47] is one such alternative. It is open source, distributed and is focused on providing a system wide ordered data and event stream that can be persisted over a certain time period. But as every third party service, it intro-

duces a dependency on the project. Not only in regards to interfacing and driver implementation, but also for maintenance and setup. Each Winslow Image would require to be shipped with a pre-configured Kafka service and the administrator must ensure that these services can reach each other - in addition to the storage solution.

Instead, it would be quite elegant, if an already planned connection between all Winslow instances could be reused for this, which could eliminate the need for third party dependencies here. For now, only a common storage is required. Re-using this as communication channel, a mechanism must be found to give each occurring event a sequence number, that is unique system wide. New events would not be allowed to be propagated by an instance before it processed all previous events. With this idea in mind, messages that can be used coordinated all required actions are defined next.

6.4.1 Messages

To coordinate the Winslow instances, the messages will be exchanged through the common event bus, which broadcasts every message to all instances.

Because this is a multi instance system which can suffer partial failure, all multi-part operations have to have a timeout attached, so that a failure is detectable. Without this timeout, one could not detect the absence of a finish signal for a multi-part operation, which could potentially block further operations for forever.

To account for transmission delay and slight clock offsets an additional time padding is granted. Because of the non-requirement for real-time scheduling this can generously be set to 5 seconds.

All exchanged messages must share a common structure, which is listed next:

- **issuer**: The id of the Winslow instance that published the message
- **command**: The state change request (described a bit further below)

- **subject**: What the command is about to change
- **timestamp**: The time of when the message was issued
- **duration**: Optionally, a non zero time period of the expected duration

With this message structure definition, the following commands can be used:

- **LOCK**: Locking a resource, commonly a project. A lock is exclusive to the issuer and can only one at a time can be granted for a subject.
- **EXTEND**: Updating an ongoing operation by extending the timeout. This can be used to signal that a lock is still required although its timeout is about to be reached. The duration of the original event is extended and therefore the timeout is pushed back. The issuer must be the same as the issuer of the operation that is referred to.
- **RELEASE**: Releasing a locked resource. The issuer must be the same node as the one that issued the **LOCK** previously.
- **KILL**: Non-gracefully stop an operation or destruction of a lock. This signals that an operation shall be stopped or that the lock shall be released immediately. This can be used to abort an running stage execution.
- **ELECTION_START**: Signals that an election for a stage execution has started. The signal must refer to an project that can make progress.
- **ELECTION_PARTICIPATE**: Signals that the issuer node is capable of executing the next stage of the mentioned project. It also includes a scoring for the affinity and aversion (more details in subsection 6.7.1). This message must always refer to a valid election.
- **ELECTION_STOP**: Signals that an election has finished. The participator with the best score is now allowed execute the next stage of the referred project. This is signal is only allowed to be issued by the same node that started the election process.

6.4.2 Synchronization

In a file system two files with the same name are not allowed to exist in the same directory, which is ensured by the filesystem. This can be used when publishing events in a common directory: a by all instances followed sequence number is used as file name. Publishing a new event will fail, if there already exists a file for that sequence number. Watching the directory will then also result in a stream of ordered events.

To publish an event, a naive implementation would first check the directory for the next number in the sequence and to create it, if it was not found. But this has a potential race condition: between checking for the file and creating it, another instance could have created the file. Luckily, the POSIX standard provides a flag when creating files, that will expose this conflict by returning an error, if "Both O_CREAT and O_EXCL are set, and the named file already exists"[48]. This mechanism is also exposed in Java by calling `Files.write(..)` [49] with `StandardOpenOptions.CREATE_NEW` [50]. The behaviour of publishing an event can therefore be summarized with Figure 6.1:

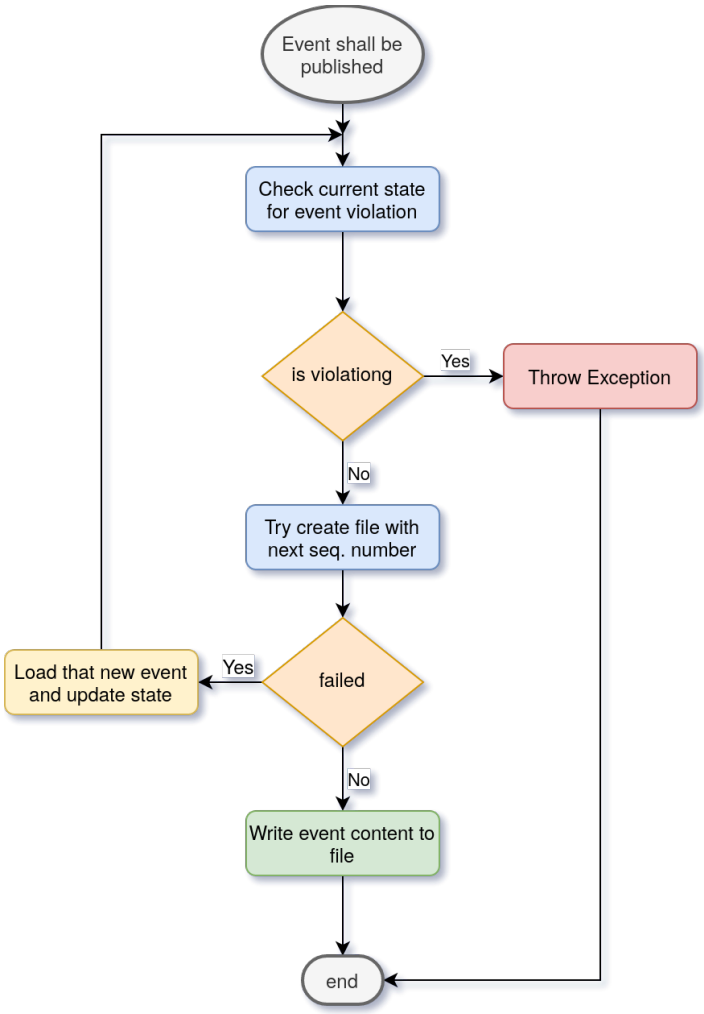

Figure 6.1: Event publishing process

Every Winslow instance is responsible for keeping track of all global states and check before publishing a new event whether the state would be violated by this event. If it is not, it will try to publish it by creating a new file with the next sequence number. If the file already exists, another instance did publish an event in the meantime. This externally published event must then be loaded and the global state updated before a new attempt in publishing the original event can be made. Eventually, this will either lead to successfully publishing the event or failing due to state violation - for example trying to lock a project that is already locked.

This synchronization mechanism ensures that each event is only published if it is not violating any constraints, such as locking resources that are already locked. It also allows for multiple locks and elections to happen concurrently, as long as there is no overlap.

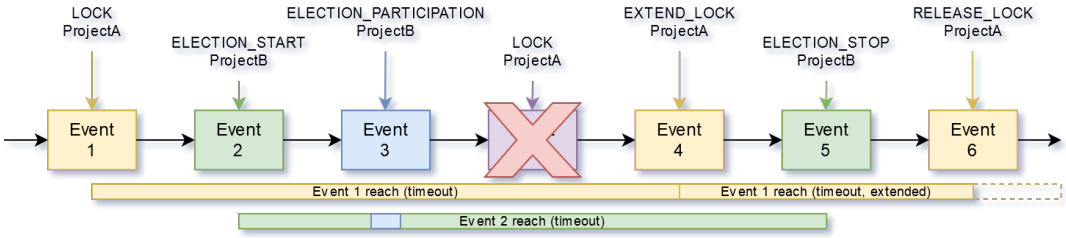

Figure 6.2: Sample event and lock series with lifetimes

In Figure 6.4.2 a series of events, their sequence number and commands are shown. The coloured rectangles below shows the duration of their lifetimes. The crossed-out event is violating the constraint that a resource can only be locked by one instance at a time: the yellow lifetime in this case. Because of this violation the even is never published to the event bus, which is demonstrate with the sequence number of the next event not skipping the number 4.

6.5 Docker interface

To interface with the Docker daemon a driver implementation is needed. It needs to on container creation commands with a set of environment variables and the path to a prepared workspace.

Docker provides a REST API[51] for third party libraries as well as implementations in Go and Python. There is no official Java API implementation. This means, either a third party Java library must be used, the REST calls that are required have to be implemented within this project or another alternative must be found. The Docker REST API is very expressive which is a reason to not implement the REST calls within this project.

As it turns out, the Nomad Project, which was already investigated in subsection 3.1.4, provides a library with a thin abstraction layer and lots of reasonable default values. It also includes support to create container that use the Docker plugin from Nvidia[52]. This allows attaching GPUs to a container without the need of listing all libraries and device files manually[2]. The scheduling capabilities of Nomad are not used because the focus of nomad does only overlap partially with the needs of Winslow. A job is expected by Nomad to rerun, moved to different hosts and - in the service mode - scaled as it seems needed. The control of where the job is executed is then lost, which is not acceptable when different firewall rules apply to all Winslow instances and a judgement for the best hardware utilization was already made. Furthermore, when using GlusterFS or another cloud storage or compute capabilities, the network share is not always reachable through the same proxy for every node. The lack of control could then cause stage execution failures due to unreachable workspaces. Nomad will therefore only be used in the local Winslow container as thin layer to interface with an enriched Docker API.

6.6 Directory Structure and Organization

This section describes the organization of the working directory for the Winslow instances. Because a common network share is required by Winslow for event synchronization and the projects' workspaces, this was further extended to share the configuration and project files as well. This has the very nice side-effect, that there is no real setup to do when installing a new Winslow instance[3], it just needs access the to working directory.

For the configuration files, a principle found on Unix and Linux systems was applied: human readable text files. For complex structures YAML formatting is

[2]If you are interested in how tedious this could otherwise become, see `https://github.com/hashicorp/nomad/issues/3499#issuecomment-364214506`

[3]Appendix B lists the complete installation script for a new Winslow instance

used, while for simple key value files property formatting is used. This makes understanding the system state easier when debugging in error scenarios.

6.6.1 Winslow working directory

A brief summary of the by all instances shared working directory:

- `logs`: The location for stage log files. This includes system events and the console output with timestamps. The file name is formatted as `<project-id>-<stage-number>-<stage-name>`.
- `pipelines`: Pipeline definition file (YAML) are located here.
- `projects`: Project definition files (YAML) are located here.
- `resources`: The globally available input resources, see subsection 6.6.2.
- `run/events`: The directory used to synchronized events, see section 6.4.
- `run/nodes`: The directory used to publish node utilization, see section 6.9.
- `settings`: The directory used to share common configurations, currently this only contains a global environment variable configuration.
- `workspaces`: The per project specific workspaces are located here, see subsection 6.6.2.

Because the files in `run/events` and especially `run/nodes` are very temporary, the NFS server locates these in shared memory (`/dev/shm`) to reduce stress on IO and unnecessary wear on SSDs.

6.6.2 Stage storage

The storage organization for a project and its stages had a few unexpected concerns raised. First of all, to redo a stage, one needs to be able to access the

files that were the result of one, two or multiple stages before. Sometimes a stage wants to access intermediate data produces by multiple previous stages. Next, the input video footage needs to be accessed by multiple stages throughout the pipeline execution. Finally, some stage results are not intermediate but do already present some final results.

The first and second concern can be solved by providing a workspace directory for each stage, that is copied from the logically previous stage[4]. Once the computation of a stage is completed, the workspace is considered immutable and only used to source new workspaces from. This works fine for small intermediate results, but it does not work very well for large files - like the video footage. This delays the start of the stage execution, requires unnecessary storage due to multiple copies and provides no benefits in an archival and version control sense, because the video footage is not altered. So there needs to be another storage pool for input data, that is globally accessible and never changed: the global input storage pool. Providing one further storage pool for final results (global output pool), concern number three and four are also solved.

Because the very first stage has no workspace to source its files from, on creation of the project a workspace directory for the "zeroths" stage is created. The user can then provide the very first stage with a predefined and non-empty workspace if necessary.

This also solves te problem with delays due copy operations in NFS, which are performed client side. This means, the client reads the input file and writes to the output file[5]. This operation does not only take unnecessary long but also utilizes all available network bandwidth which then cannot be used by other applications.

To ensure that the global input and the previous intermediate results are not altered, the Docker daemon is instructed to mounted them as read-only filesystems. This also prevents bugs or errors from accidentality deleting unrelated

[4]the stage the new one is based on, this does not always need to be the numerical previous stage

[5]Server-side copy has just been standardized for NFS 4.2[53]

files[6]. Figure 6.3 summaries all this:

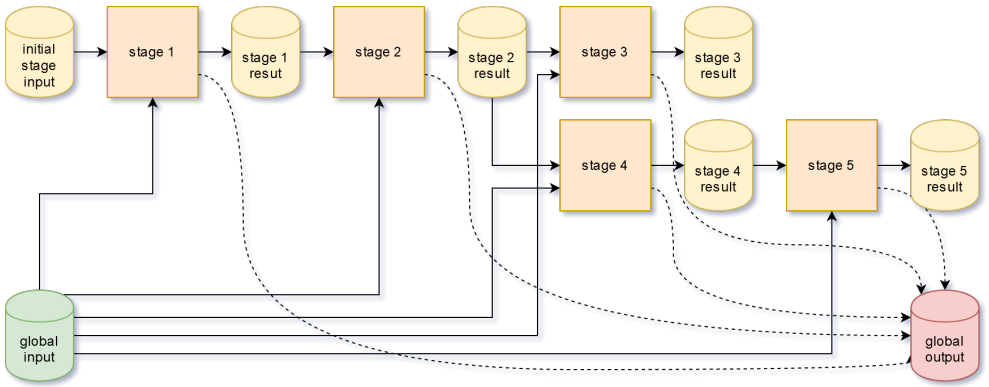

Figure 6.3: Stages with their intermediate and global resource pools

For a running stage the mounted directories looks like the following:

- `/input` is mounted read-only from
  `nfs-server:/winslow/workspaces/<project-id>/input`
- `/output` is mounted with write permissions from
  `nfs-server:/winslow/workspaces/<project-id>/output`
- `/workspace` is mounted with write permissions from
  `nfs-server:/winslow/workspaces/<project-id>/<stage-id>`
  and was created by the copying the workspace directory of the logically previous stage into it

6.7 Job Scheduling and Election System

Winslow has no ahead of time plan to schedule stage executions, the main reason for this that it is unknown how long a stage is about to take. One could consider to build statistics to eventually be able to guess the duration of repeatedly executed stages, but this seems error prone, does not cover seldom executed stages and

[6]"This so theoretical and will actually never happen in real life" proven otherwise: `https://github.com/valvesoftware/steam-for-linux/issues/3671`

is in addition unreasonable complex. Instead, the system is reacting to user submissions and on stages that have just completed. For each new execution, an election is then held in which the winner will execute the stage. Winslow instances that do not fulfil the resource requirements do not participate in the election.

6.7.1 Affinity and Aversion

To determine a winner, participants must be comparable. Winslow uses a two factor scoring mechanism for that, in which it judges how efficient a node is going to be utilized (affinity scoring) and how many unused resources will be wasted (aversion scoring). The instance with the highest affinity value out of the instances of the lowest aversion values wins.

This system was developed within this thesis but loosely inspired by Nomad[10], which also uses an affinity score[54] but no aversion equivalent. Instead of specifying this value in each job definition like in Nomad, Winslow derives it's values from the available resources and resource requirements. This requires less user interference, prevents blocking unused hardware and still delivers good results, as will be discussed next.

For determining the affinity scoring only requested resource categories c_i need to be considered. Through the resource monitoring (see section 6.9) the total resources $r_{c_{max}}$ as well as the reserved resources $r_{c_{in\_use}}$ are known. For every requested resource by the stage definition $c1..cx$, the ratio between requested r_{c_i} and available resources is calculated. The lowest ratio, and therefore the most pessimistic value, of any category is used as the affinity score:

$$\text{affinity} = \min_{c_1..c_x} \left(\frac{r_{c_i}}{r_{c_{max}} - r_{c_{in\_use}}} \cdot n_{c_x} \right) \tag{6.1}$$

As seen above, every node can additionally apply a resource category specific multiplier n_{c_x} for punishment or gratification. This allows to fine tune the score

on a per node basis.

For determining the aversion scoring, all by a node available resource categories $c1..c_a$ need to be considered. In contrast to the affinity score, the highest ratio of any unused resource is used as aversion score, which is again the most pessimistic value.

$$\text{aversion} = \max_{c_1..c_a} \left(\frac{r_{c_{max}} - r_{c_{in\_use}} - r_{c_a}}{r_{c_{max}} - r_{c_{in\_use}}} \cdot n_{c_a} \right) \tag{6.2}$$

The more resources are wasted - especially categories that are untouched at all - the worse the aversion score.

An example shall illustrate the scoring system. Lets consider the following three execution nodes exist:

	Free CPUs	Free memory	Free GPUs
Node 1	4	16 GiB	0
Node 2	12	58 GiB	0
Node 3	4	50 GiB	1

Table 6.2: Nodes to consider

The following three stages executions shall be assessed:

	Req. CPUs	Req. memory	Req. GPUs
Ex1	2	16 GiB	0
Ex2	4	48 GiB	0
Ex3	4	16 GiB	1

Table 6.3: Stage executions to consider

The following table shows the resulting affinity and aversion score. The winning node for is marked by the bold formatting.

	Node 1	Node 2	Node 3
Ex1	**0.5/0.5**	0.28/0.72	0.32/1.0
Ex2	-	**0.34/0.67**	0.96/1.0
Ex3	-	-	**0.32/0.68**

Table 6.4: Affinity/Aversion score for every stage definition and node constellation that provides all required resources

As shown in the example, the aversion score is important to prevent stages taking execution nodes that provide specialised hardware (such as GPUs) which is not required by the stage execution. In this example, it prevent Ex2 to be scheduled on Node 3, which provides a GPU.

In comparison to just forbidding executions on nodes with hardware that is not used, in scenarios where all general purpose nodes have failed, are unreachable or stage execution has very little requirements, this system would still schedule executions. But it tries as best as possible to prevent blocking unused resources, like Ex1 and Ex2 running on GPU nodes.

6.8 Execution

The execution of a stage is started on the Winslow instance that won the corresponding election. It first locks the project, to re-read the configuration and to prevent any changes to in the meantime. The workspace is being prepared (see subsection 6.6.2), environment variables collected and the backend driver instructed to start the new container. Any data logged to the containers' stdout or stderr is collected and written to a log file exclusive to this stage execution (see subsection 6.6.1). System related events, such as errors pulling an image, exhausted resources or sudden aborts are also logged to this file.

A lock on the project is not held during all the time the stage is being executed, because it would prevent stages being enqueue and or prepared by a user in the

meantime. Instead, after successfully starting the new container and the log file has been created, a new lock pointing to the log file is created. The currently executed stage is noted in the project file and then the lock on it is released. As long as the lock for a log file is alive, no election for the project is allowed.

Once the container has stopped - and the stage has therefore either failed or succeeded - the lock on the project is acquired again to update the stage state. After this, both locks are released, potentially triggering the next election.

6.9 CPU, RAM and IO Monitoring

Monitoring the hardware utilization can be accomplished by parsing the contents of `/proc/cpuinfo`[7], `/proc/stat`, `/proc/meminfo`[8], `/proc/net/dev` and `/proc/diskstats`[9]. These special files in the `/proc` directory are text files generated by the Linux kernel to summarize the current utilization[55].

Each Winslow instance is repetitively reading and parsing these files and writing a summarized version to the shared workspace directory `/run/node/<node-name>`. An summary example can be seen in Appendix C.

6.10 User Interface

To display data on the user interface[10], REST requests are sent to the web-backend of Winslow. To handle these requests and to serialize the responses Spring Boot[56] is used by Winslow.

To not stress the event bus with lock requests caused by the user, all data retrieval operation open and read configuration files without locking them. This bears the risk of reading incompletely written files, but as the user interface is not

[7] `https://www.kernel.org/doc/Documentation/cputopology.txt`

[8] `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`

[9] `https://www.kernel.org/doc/Documentation/ABI/testing/procfs-diskstats` and `https://www.kernel.org/doc/Documentation/iostats.txt`

[10] Screenshots in Appendix D

crucial to the system and as it must account for failed requests due to network issues anyway, this is considered acceptable. The user interface would simple resubmit its request.

When modifying settings or triggering stage executions, the request handling must obey the locking rule to not introduce inconsistent state to the system (due to a dirty read[57, p. 2] that could occur without locking). This means, on a HTTP POST request, the affected resource is being locked, updated and released. The release will then trigger a Winslow instance to check the file for potentially starting an election.

6.11 Continuous Deployment

Winslow uses GitLab CI to continuously deliver runnable Docker Images after each git push. Every code change triggers a new build pipeline, that builds and tests Winslow, packages all dependencies[11] into a Docker image and pushes it to our in house private Docker Registry. Broken code or failing unit tests will result in a notification for the developer and prevents the image from being published.

To update an execution node, the Docker container can simple be stopped and discarded. The installation script[12] then pulls and starts the most recent image.

[11] Nomad and the Angular Web-Application
[12] Which is displayed in Appendix B

Chapter 7

Outcome and Measurements

In this chapter the outcome is measured and evaluated.

7.1 What did not work as expected

It is impossible to plan a system ahead of time without making some mistakes. The biggest issue that surfaced on an early implementation of Winslow was caused by too short-termed EXTEND signals (see subsection 6.4.1). In combination with short but high network bandwidth usages, the event was published too late, which led the other instances consider the node to have failed. This was solved by increasing lock timeouts and issuing the EXTEND signal halfway through the timeout instead of very close to the end.

7.2 User Authentication and Security

One aspect that was not yet mentioned at all is user authentication and security. Winslow is prepared in this regards because every project has an owner field and a member list and every web access checks these. An internal user repository also associates users with groups and provides defaults, such as the user and group "anonymous" and "root" for no and full access privileges, respectively. But at the

moment, there is no user login. Instead, to the system all requests seem to be issued by "root". In the future it is planned to be replaced this by an Single Sign-On implementation that uses the user accounts of our company. Winslow is then given a user name which it will use to identify the corresponding internal user. Unknown users are not allowed to see or alter anything, known users will be able to create and view their own projects and projects they are listed as members in.

Winslow also supports secure HTTP (HTTPS) whenever a SSL certificate is provided[1].

7.3 High Level System Overview

Figure 7.1 shows the high level system overview of Winslow as it is implemented and in use by the end of this thesis in our company.

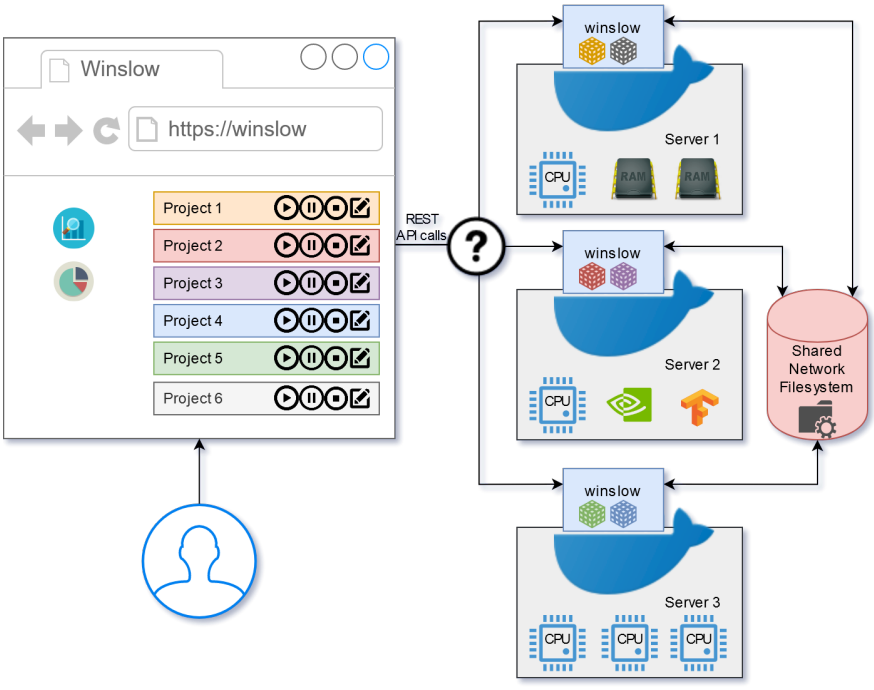

Figure 7.1: High level view on the system structure

The user can access the system through the Web-Application, which has no

[1]This can be seen in Appendix B

indicating to tell from which Winslow instance it is served. The three compute nodes supply different hardware, one with 128 GiB RAM, one with 4 NVIDIA RTX 2080 Ti, and one with 32 Intel CPU cores. All instances access the same NFS share for synchronization and work distribution.

7.4 Failure resilience

One of the requirements for Winslow is to be resilient against node failures. Node failures can be simulated easily by killing the Docker container of a Winslow instance. The behaviour that is then displayed depends on whether that killed instance was actually executing a stage. If it did not, the only change is that in the Web-Application the node with its utilization reports will vanish.

But if it did execute a stage, it held locks for it. These locks will expire, and after the additional grace period (see subsection 6.4.1), the other instances will notice this, lock the affected project themselves and mark the execution as failed. Currently, the project is then paused and a user confirmation awaited. An alternative to this could be to re-schedule the stage automatically.

7.5 Evaluation

The main focus of Winslow is to improve the efficiency in utilizing available hardware. Without this automated approach, new stages were seldom started outside of work hours. To estimate whether Winslow is providing any improvements, the actions that are scheduled by Winslow outside of these hours are to be assessed.

The following graphs are based on activity metrics, that were logged in the time period from the first usage in mid January 2020 until counter actions for COVID-19 in early March 2020 prevented active use due to the lock-down in Germany. For each day of the week and hour of the day an associated cell shows the activity encoded in colours. Furthermore, red lines show the begin and end of a typical work day. Interestingly, the actual position of these red lines could also be inferred by the graphs themselves, due to activity spikes.

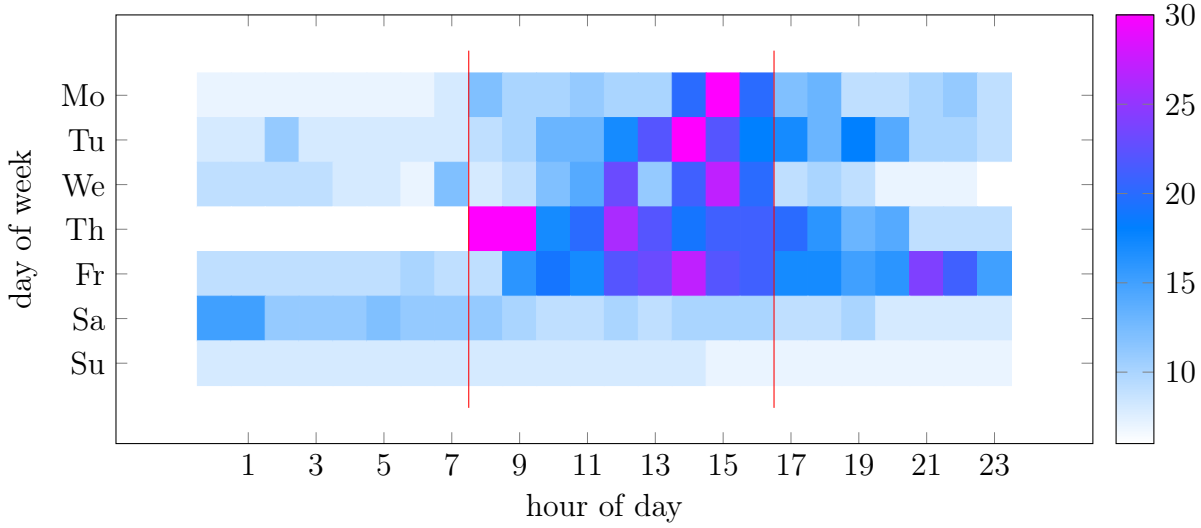

Figure 7.2: Actively running stages at a given day of the week and time of day

Figure 7.2 and Figure 7.3 show the overall activity. In Figure 7.2 the number of running stages per hour is shown and in Figure 7.3 the number of started stages per hour is shown. An increased activity between the read line - within the working hours - is visible. But especially in Figure 7.3 an activity outside

these times can be seen as well.

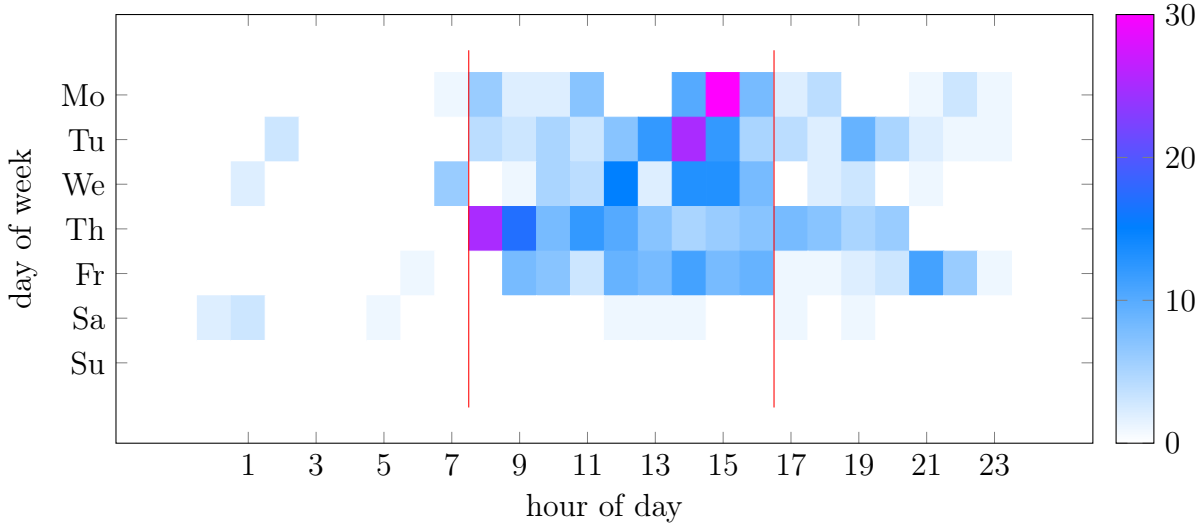

Figure 7.3: Stages starting at a given day of the week and time of day

Figure 7.4 and Figure 7.5 show only stages that were automatically triggered by Winslow.

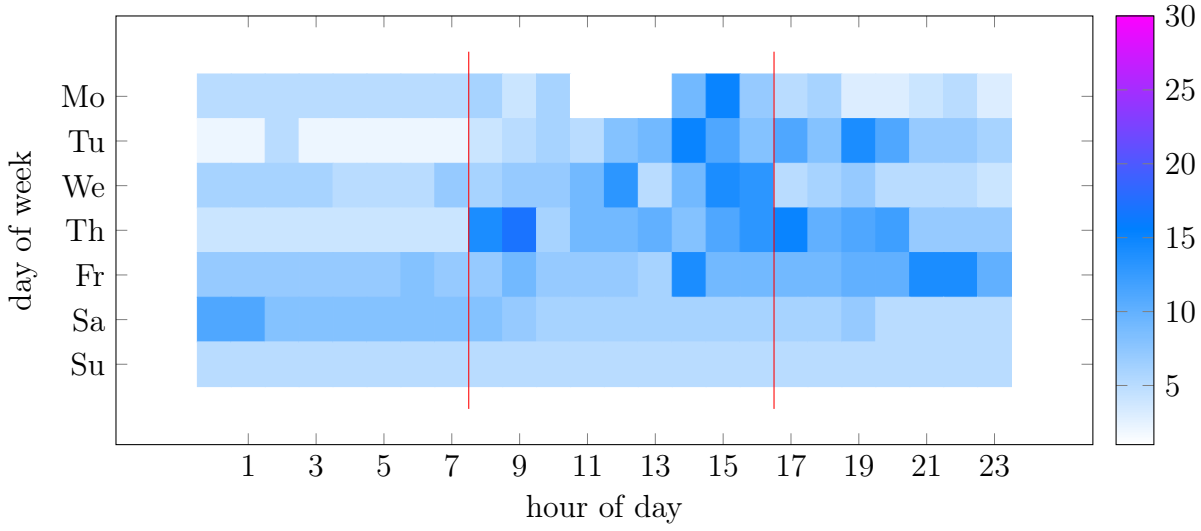

Figure 7.4: Actively running stages at a given day of the week and time of day which were started automatically

Figure 7.2 clearly shows that there is quite a bit of activity even without direct user input. This is especially obvious by looking at Figure 7.5: there are many stages started in-between the late evening and midnight, a second wave at about one o'clock in the morning and activity on Saturdays. Without Winslow these stages would not have been triggered.

The graphs imply that the system was actively used to schedule work to be executed on the weekends. This can be seen in Figure 7.5 by the highlighted areas at late hours on Fridays, early and late hours on Saturdays as well as the continuous but decreasing overall activity from Saturday to Sunday in Figure 7.4. The gap of activity on Mondays at around noon in Figure 7.4 could indicate that the results were evaluated here.

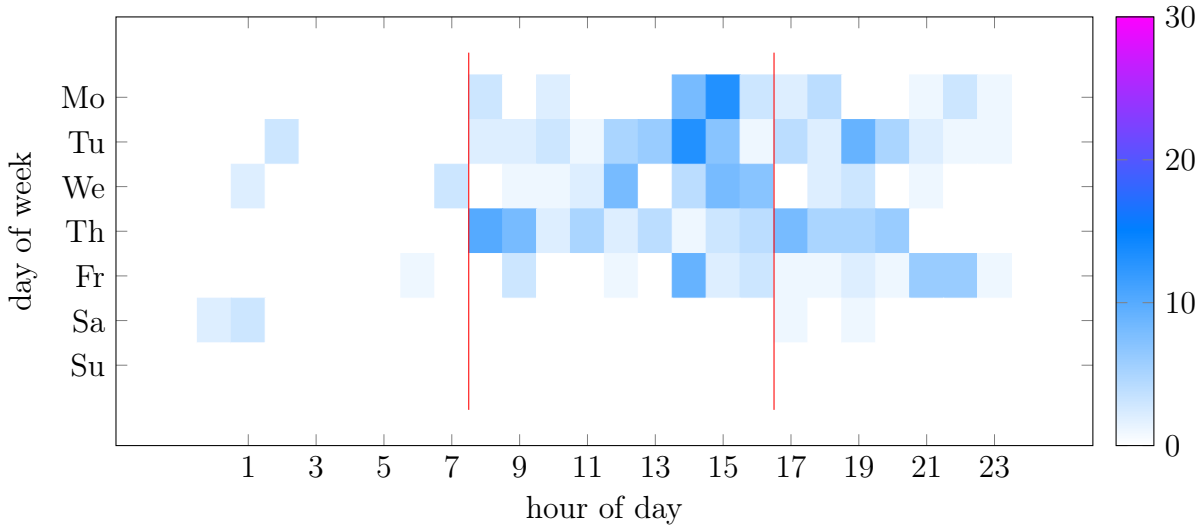

Figure 7.5: Stages automatically starting at a given day of the week and time of day

For the graphs above, the following table summarizes the results:

	absolute	relative	gain
All	2003 h	100 %	
Within work time	1575 h	78.6 %	
Outside work time	428 h	21.4 %	
Triggered by Winslow overall	1131 h	56.5 %	+129 %
Triggered by Winslow outside work time	286 h	14.3 %	+16 %

Table 7.1: Categorization of stage execution hours

Overall, in the test period 2003 hours were spent computing - this includes overlapping stage executions - of which 78.6% were executed within regular working hours. To see the benefits of Winslow, the hours spent in stage executions that were automatically issued are of interest. Here, Winslow triggered 1575 hours or 78.6% of all stage executions which did not require staff assistance. 286 hours were overall gained by automatically triggering and performing stage executions outside of the working hours. This is an increase of +16% compared to no automated triggers.

To put this into perspective, typically week consists of 40 hours of work and 128 hours with the staff being absent. So in theory 128 hours (+220%) per week could additionally be spent executing stages, which raises the question why the numbers of Winslow are so small in comparison.

To answer this, one needs to look again at the heat-maps of Figure 7.2 and Figure 7.4. Winslow is being used, and as for Figure 7.5 discussed, it is being used for long running stage executions with one or two automatic triggers at nights. But it is used even more during working hours, as can be seen in Figure 7.2, where all pink cells fall within working hours. Furthermore, not all executions seem to take an entire night until staff assistance is required again, which can be seen in Figure 7.2 where the colours start to fade in the late evening and during the night. There is also no guarantee that the user is trying to completely exhaust all compute resources in his/hers absence.

In conclusion, Winslow is being used - and that actively, with over 2003 hours worth of compute time[2]. It is supporting the users by being the triggering for over half of all compute time spent and additionally enabling the staff to let large multi-stage tasks be executed over night or the weekend. If it would not be beneficial to the staff, it would not be used in such an extend so shortly after implementation.

[2]in the previously mentioned time period

Chapter 8

Summary and Conclusion

Winslow helps to improve the usage of available hardware resources by additionally triggering stage executions in times where the staff is absent or not actively watching. It decreases the effort required from the staff to progress projects. It helps to keep order by separating workspaces and organizing them into projects. The user interface provides an interaction method that is easy to understand provides upload and download mechanism without resorting back to console commands.

In the implementation phase an interesting and minimalistic way to synchronize events based on a commonly shared directory was discovered and implemented. The coordination is decentralized to cope with node failures. It uses an election system to find fitting execution nodes without blocking unused resources. Winslow instances are easy to setup and because of its few dependencies easy to maintain.

The evaluation shows that Winslow is successfully increasing productivity, hardware utilization and that it is beneficial to the user.

8.1 Further work

In the scope of this thesis a system was implemented that distributes the workload of a computer vision pipeline onto several compute nodes. Automatically scheduling stages improves the utilization of available hardware - especially at times where the staff is absent, like at nights or weekends. A system like this provides an endless stream of ideas to further improve the user experience or for extensions and capabilities to add.

Adding the support for WebSockets could help to make the Web-Application more responsive to serve side changes. Pushing desktop notifications to the client or sending e-mails could reduce the response time of the staff for when a stage failed or a project needs attention otherwise. GlusterFs could be investigated for whether the (site) replication feature could reduce the usage of network bandwidth and availability.

Bibliography

[1] MEC-View Dr. Rüdiger W. Henn. *Mobile Edge Computing basierte Objekterkennung für hoch- und vollautomatisiertes Fahren.* URL: `http://mec-view.de/` (visited on 09/29/2019).

[2] The Apache Software Foundation. *Apache Hadoop.* URL: `https://hadoop.apache.org/` (visited on 09/19/2019).

[3] Inc. GitLab. *The first single application for the entire DevOps lifecycle.* URL: `https://about.gitlab.com/` (visited on 09/21/2019).

[4] jenkins.io. *Jenkins. Build great things at any scale.* URL: `https://jenkins.io/` (visited on 09/19/2019).

[5] Inc. GitLab. *GitLab CI/CD. Pipeline Configuration Reference.* URL: `https://docs.gitlab.com/ee/ci/yaml/` (visited on 09/19/2019).

[6] jenkins.io. *Using a Jenkinsfile.* URL: `https://jenkins.io/doc/book/pipeline/jenkinsfile/` (visited on 09/19/2019).

[7] Camunda Services GmbH. *Workflow and Decision Automation Platform.* URL: `https://camunda.com/` (visited on 09/19/2019).

[8] Camunda Services GmbH. *Process Engine API.* URL: `https://docs.camunda.org/manual/7.6/user-guide/process-engine/process-engine-api` (visited on 09/19/2019).

[9] Camunda Services GmbH. *Rest Api Reference.* URL: `https://docs.camunda.org/manual/7.8/reference/rest` (visited on 09/19/2019).

[10] HashiCorp. *Nomad.* URL: https://www.nomadproject.io/ (visited on 09/19/2019).

[11] HashiCorp. *Nomad. Nomad vs. Kubernetes.* URL: https://www.nomadproject.io/intro/vs/kubernetes.html (visited on 09/19/2019).

[12] Bc. Pavel Peroutka. *Web interface for the deployment andmonitoring of Nomad jobs. Master's thesis.* URL: https://dspace.cvut.cz/bitstream/handle/10467/80106/F8-DP-2019-Peroutka-Pavel-thesis.pdf (visited on 09/22/2019).

[13] Tigran Mkrtchyan Patrick Fuhrmann. *dCache. Scope of the project.* URL: https://www.dcache.org (visited on 09/19/2019).

[14] Patrick Fuhrmann. *dCache, the Overview.* URL: https://www.dcache.org/manuals/dcache-whitepaper-light.pdf (visited on 09/19/2019).

[15] Inc. Terracotta. *QuartzJob Scheduler.* URL: http://www.quartz-scheduler.org/ (visited on 09/19/2019).

[16] Data Revenue. *Distributed Python Machine Learning Pipelines.* URL: https://www.datarevenue.com/en/blog/how-to-scale-your-machine-learning-pipeline (visited on 09/19/2019).

[17] Ask Solem. *Celery: Distributed Task Queue.* URL: http://www.celeryproject.org (visited on 09/19/2019).

[18] IBM Knowledge Center. *IBM InfoSphere. DataStage.* URL: https://www.ibm.com/support/knowledgecenter/en/SSZJPZ_9.1.0/com.ibm.swg.im.iis.ds.design.doc/topics/c_ddesref_Server_Job_Stages_.html (visited on 09/19/2019).

[19] Wikipedia contributors. *Qsub. Wikipedia, The Free Encyclopedia.* URL: https://en.wikipedia.org/w/index.php?title=Qsub&oldid=745279355 (visited on 09/22/2019).

[20] The University Of Iowa. *Basic Job Submission. HPC Documentation Home / Cluster Systems Documentation.* URL: https://wiki.uiowa.edu/display/hpcdocs/Basic+Job+Submission (visited on 09/22/2019).

[21] Swiss National Supercomputing Centre. *High Throughput Scheduler.* URL: https://user.cscs.ch/tools/high_throughput/ (visited on 09/19/2019).

[22] Colin Phipps. *zsync. Overview.* URL: http://zsync.moria.org.uk/ (visited on 09/19/2019).

[23] OpenIO. *High Performance Object Storage for Big Data and AI.* URL: https://www.openio.io/ (visited on 09/22/2019).

[24] Chris Lu. *Simple and highly scalable distributed file system.* URL: https://github.com/chrislusf/seaweedfs (visited on 09/22/2019).

[25] Colin Phipps. *Alluxio. Data Orchestration for the Cloud.* URL: https://www.alluxio.io (visited on 09/19/2019).

[26] Inc Red Hat. *Gluster. Free and open source scalable network filesystem.* URL: https://www.gluster.org (visited on 09/19/2019).

[27] Inc. Docker. *Enterprise Container Platform.* URL: https://www.docker.com/ (visited on 09/22/2019).

[28] Inc. Docker. *Docker. Docker Logos and Photos.* URL: https://www.docker.com/company/newsroom/media-resources (visited on 01/16/2020).

[29] Inc. Docker. *Docker Documentation. Swarm mode key concepts.* URL: https://docs.docker.com/engine/swarm/key-concepts/ (visited on 03/17/2020).

[30] The Kubernetes Authors. *Kubernetes. What is Kubernetes.* URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 03/17/2020).

[31] Inc. Docker. *Docker Hub. Build and Ship any Application Anywhere.* URL: https://hub.docker.com/ (visited on 09/22/2019).

[32] Neil Brown. *Linux Kernel Documentation. Overlay Filesystem.* URL: https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt (visited on 03/29/2020).

[33] Inc. Docker. *Docker Documentation. Docker overview.* URL: https://docs.docker.com/engine/docker-overview/ (visited on 01/16/2020).

[34] IETF | Internet Engineering Task Force. *RFC 5661. Network File System (NFS) Version 4 Minor Version 1 Protocol.* URL: https://tools.ietf.org/html/rfc5661%5C#section-1.7.2.2 (visited on 03/26/2020).

[35] Google. *Angular.* URL: https://angular.io/ (visited on 03/29/2020).

[36] Microsoft. *TypeScript. JavaScript that scales.* URL: https://www.typescriptlang.org/ (visited on 03/29/2020).

[37] IETF | Internet Engineering Task Force. *RFC 7231. Hypertext Transfer Protocol (HTTP/1.1).* URL: https://tools.ietf.org/html/rfc7231#section-4 (visited on 03/29/2020).

[38] John F. Mee. *Frederick W. Taylor. American inventor and engineer.* URL: https://www.britannica.com/biography/Frederick-W-Taylor (visited on 03/31/2020).

[39] Wikipedia contributors. *Frederick Winslow Taylor — Wikipedia, The Free Encyclopedia.* [Online; accessed 17-March-2020]. 2020.

[40] The Editors of Encyclopaedia Britannica. *Taylorism. scientific management system.* URL: https://www.britannica.com/science/Taylorism (visited on 03/31/2020).

[41] J. Goll. *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik: Strategien für schwach gekoppelte, korrekte und stabile Software.* Springer Fachmedien Wiesbaden, 2018.

[42] R.C. Martin, J.M. Rabaey, A.P. Chandrakasan, et al. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003.

[43] J. Goll. *Methoden des Software Engineering: Funktions-, daten-, objekt- und aspektorientiert entwickeln*. Springer Fachmedien Wiesbaden, 2012.

[44] "Use Case Modeling". In: *Use Case Driven Object Modeling with UML: Theory and Practice*. Berkeley, CA: Apress, 2007, pp. 49–82.

[45] Jeffrey Palermo. *The Onion Architecture*. URL: `https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/` (visited on 03/26/2020).

[46] Inc. Docker. *Press Release. Docker Restructures and Secures $35 Million to Advance Developer Workflows for Modern Applications*. URL: `https://www.docker.com/press-release/docker-new-direction` (visited on 03/23/2020).

[47] The Apache Software Foundation. *Apache Kafka*. URL: `https://kafka.apache.org/` (visited on 03/29/2020).

[48] GNU.org / The Free Software Foundation. *Opening and Closing Files*. URL: `https://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html` (visited on 03/29/2020).

[49] Oracle. *Java Platform SE 7. Files*. URL: `https://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html` (visited on 03/29/2020).

[50] Oracle. *Java Platform SE 7. StandardOpenOptions*. URL: `https://docs.oracle.com/javase/7/docs/api/java/nio/file/StandardOpenOption.html#CREATE_NEW` (visited on 03/29/2020).

[51] Inc. Docker. *Develop with Docker Engine API*. URL: `https://docs.docker.com/engine/api/` (visited on 03/29/2020).

[52] NVIDIA. *NVIDIA Container Toolkit*. URL: `https://github.com/NVIDIA/nvidia-docker` (visited on 03/29/2020).

[53] IETF | Internet Engineering Task Force. *RFC 7862. Network File System (NFS) Version 4 Minor Version 2 Protocol*. URL: https://tools.ietf.org/html/rfc7862 (visited on 03/29/2020).

[54] HashiCorp. *Nomad. Affinity Stanza*. URL: https://nomadproject.io/docs/job-specification/affinity/ (visited on 03/31/2020).

[55] *The /proc filesystem*. URL: https://www.kernel.org/doc/Documentation/filesystems/proc.txt (visited on 03/29/2020).

[56] Inc. or its affiliates VMware. *Spring*. URL: https://spring.io/ (visited on 03/29/2020).

[57] Hal Berenson, Phil Bernstein, Jim Gray, et al. *A Critique of ANSI SQL Isolation Levels*. June 1995. URL: https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/ (visited on 03/31/2020).

Appendix A

Project Management

The project management went mostly as planned. All deadlines regarding the implementation and deployment were met as planned. A few of the milestones for documentation were pushed back slightly, because some parts took a bit longer to write about than anticipated at first. But, because a time puffer at the end was considered from the beginning, the project has been finished in time. The time schedule can be seen in Figure A.1:

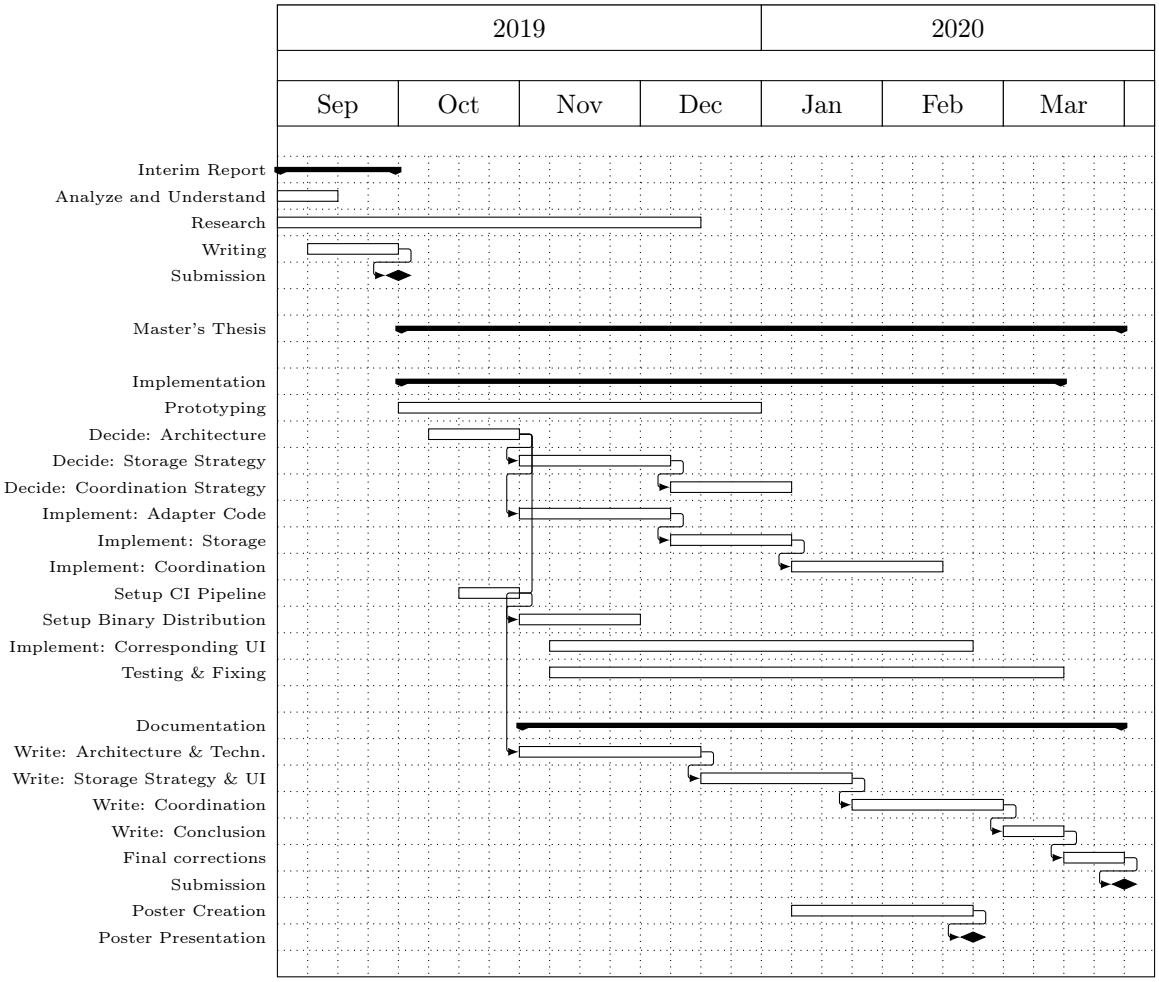

Figure A.1: Time schedule

Appendix B

Winslow Instance Installation

At the time of writing this, the script in Listing B.1 is all that is required to install a new Winslow instances. It spans a total of 72 lines of bash code, of which many lines print debug information or help to organize the script.

```
1  #!/bin/bash
2
3  NODE_TYPE="executor"
4  STORAGE_TYPE="nfs"
5  STORAGE_PATH="johnny5.itd-intern.de:/data/streets/winslow"
6  ADDITIONAL="-p 446:4646 -e WINSLOW_DEV_ENV=true -e
       WINSLOW_DEV_REMOTE_USER=root"
7
8  PARAMS="$@"
9  IMAGE="repo.itd-intern.de/winslow/node"
10 NODE_NAME="$(hostname)"
11 CONTAINER_NAME="winslow"
12 GPUS="$(ls /dev/ | grep -i nvidia | wc -l)"
13 WORKDIR="/winslow/"
14
15 SUDO=""
16
17 if [ "$(id -u)" -ne 0 ] && [ "$(id --name -G | grep -i docker | wc
       -l)" -eq 0 ]; then
```

```
18      SUDO="sudo"
19  fi
20
21  $SUDO docker pull $IMAGE
22
23  if [ "$KEYSTORE_PATH_PKCS12" != "" ]; then
24      ADDITIONAL="$ADDITIONAL -p $HTTPS:8080 -v
        $KEYSTORE_PATH_PKCS12:/keystore.p12:ro -e
        SERVER_SSL_KEY_STORE_TYPE=PKCS12 -e
        SERVER_SSL_KEY_STORE=file:/keystore.p12 -e
        SECURITY_REQUIRE_SSL=true -e SERVER_SSL_KEY_STORE_PASSWORD="
25  fi
26
27
28  echo ""
29  echo ""
30  echo ""
31  echo " :::::  Going to create Winslow Container with the following
        settings"
32  echo ""
33  echo "   HTTP Port     '$HTTP'"
34  echo "   HTTPS Port    '$HTTPS'"
35  echo "   Docker Image '$IMAGE'"
36  echo "   Storage Type '$STORAGE_TYPE' @ '$STORAGE_PATH'"
37  echo ""
38  echo "   Work Directory '$WORKDIR'"
39  echo "   Node Name       '$NODE_NAME'"
40  echo ""
41  echo "   Detected GPUs: $GPUS"
42  echo ""
43  echo "   Additional Docker Parameters:  '$ADDITIONAL'"
44  echo "   Additional Winslow Parameters: '$PARAMS'"
45  echo ""
46
47  sleep 1
```

```
48
49 if [ $($SUDO docker ps --filter "name=$CONTAINER_NAME" | wc -l) -gt
       1 ]; then
50         echo " ::::: Stopping already running Winslow instance"
51             $SUDO docker stop "$CONTAINER_NAME" > /dev/null && echo
       "  :::: Done" || (echo " :::: Failed"; exit 1)
52     fi
53
54 $SUDO docker rm "$CONTAINER_NAME" > /dev/null
55
56 echo " ::::: Starting Winslow Container now"
57 $SUDO docker run -itd --privileged \
58     --restart=unless-stopped \
59     --name "$CONTAINER_NAME" \
60     $(if [ "$GPUS" -gt 0 ]; then echo "--gpus all"; fi) \
61     -p $HTTP:8080 \
62     $ADDITIONAL \
63     -e WINSLOW_STORAGE_TYPE=$STORAGE_TYPE \
64     -e WINSLOW_STORAGE_PATH=$STORAGE_PATH \
65     -e WINSLOW_WORK_DIRECTORY=$WORKDIR \
66     -e "WINSLOW_NODE_NAME=$NODE_NAME" \
67     $(if [ "$NODE_TYPE" == "observer" ]; then echo "-e
       WINSLOW_NO_STAGE_EXECUTION=1"; fi) \
68     -v /var/run/docker.sock:/var/run/docker.sock \
69     $IMAGE \
70     $PARAMS
71
72 $SUDO docker attach "$CONTAINER_NAME"
```

Listing B.1: The whole installation script for Winslow

Appendix C

Node Status Information

Listing C.1 shows what a status file of an execution node in `/run/nodes/` looks like:

```
name = "srv515"

[cpuInfo]
modelName = "Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz"
utilization = [0.029411765, 0.009803922, 0.0, 0.009803922, 0.0,
    0.0, 0.0, 0.0, 0.0, 0.029411765, 0.0, 0.0]

[memInfo]
memoryTotal = 67447889920
memoryFree = 55159222272
systemCache = 39576674304
swapTotal = 1074786304
swapFree = 815677440

[netInfo]
transmitting = 2888
receiving = 2042

[diskInfo]
reading = 0
```

```
writing = 188416
free = 440000196608
used = 48637153280

[[gpuInfo]]
vendor = "nvidia"
name = "GeForce RTX 2080 Ti"

[[gpuInfo]]
vendor = "nvidia"
name = "GeForce RTX 2080 Ti"

[[gpuInfo]]
vendor = "nvidia"
name = "GeForce RTX 2080 Ti"

[[gpuInfo]]
vendor = "nvidia"
name = "GeForce RTX 2080 Ti"

[buildInfo]
date = "2020-02-12 14:24:39"
commitHashShort = "cb259423"
commitHashLong = "cb259423a47a38530abae211cf0108fbaf01d852"
```

Listing C.1: Sample for a node status information

Appendix D

Screenshots

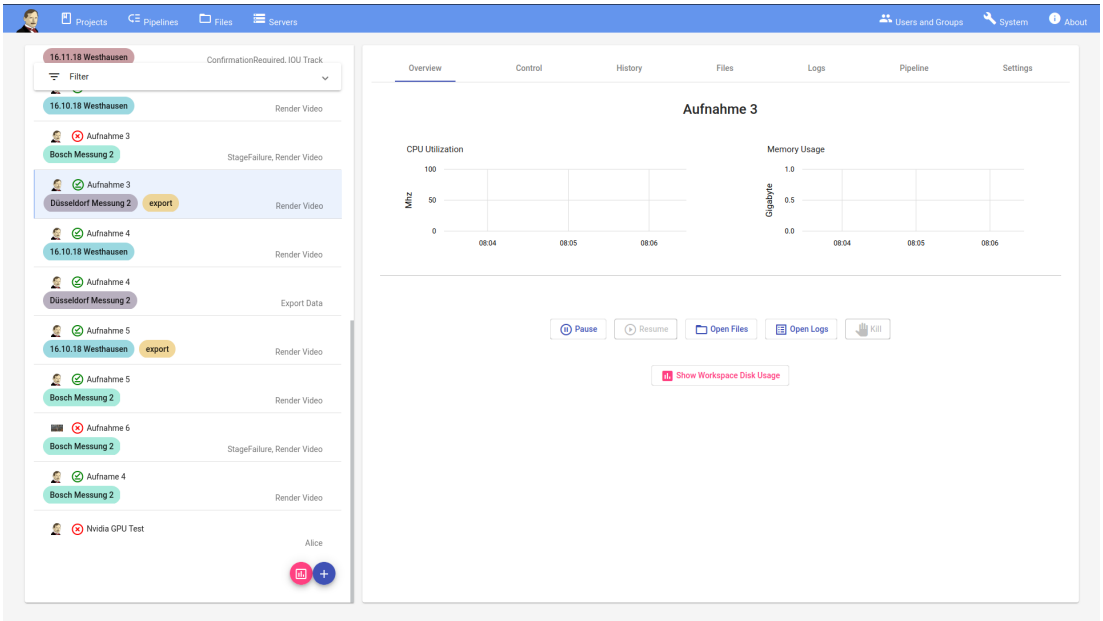

Figure D.1: Left: Known projects, Right: Overview for the selected project

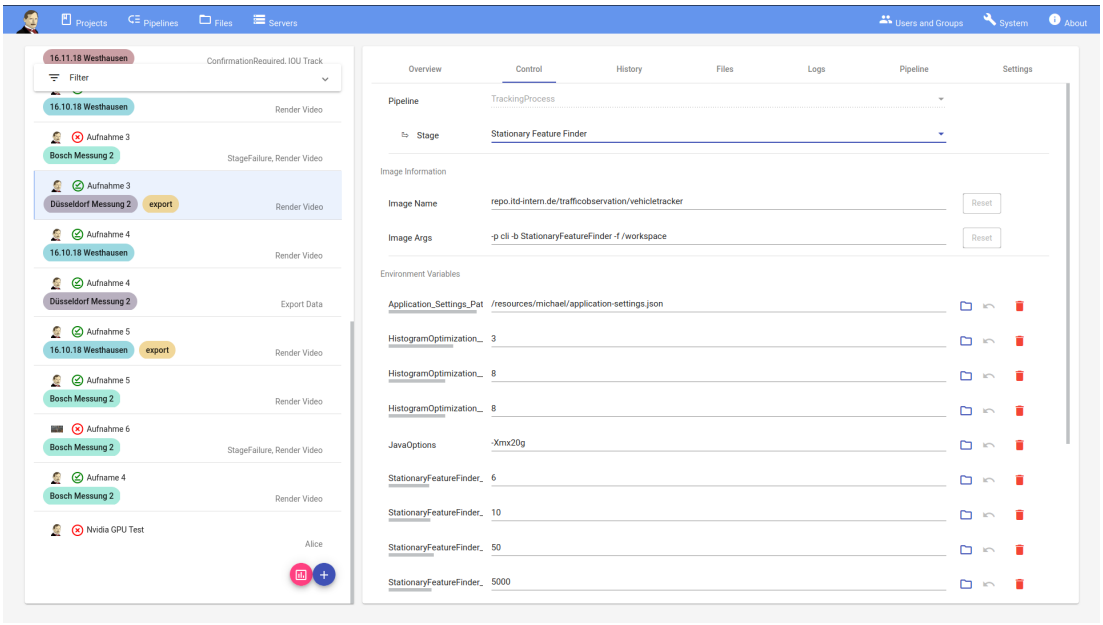

Figure D.2: Left: Known projects, Right: Control for the selected project

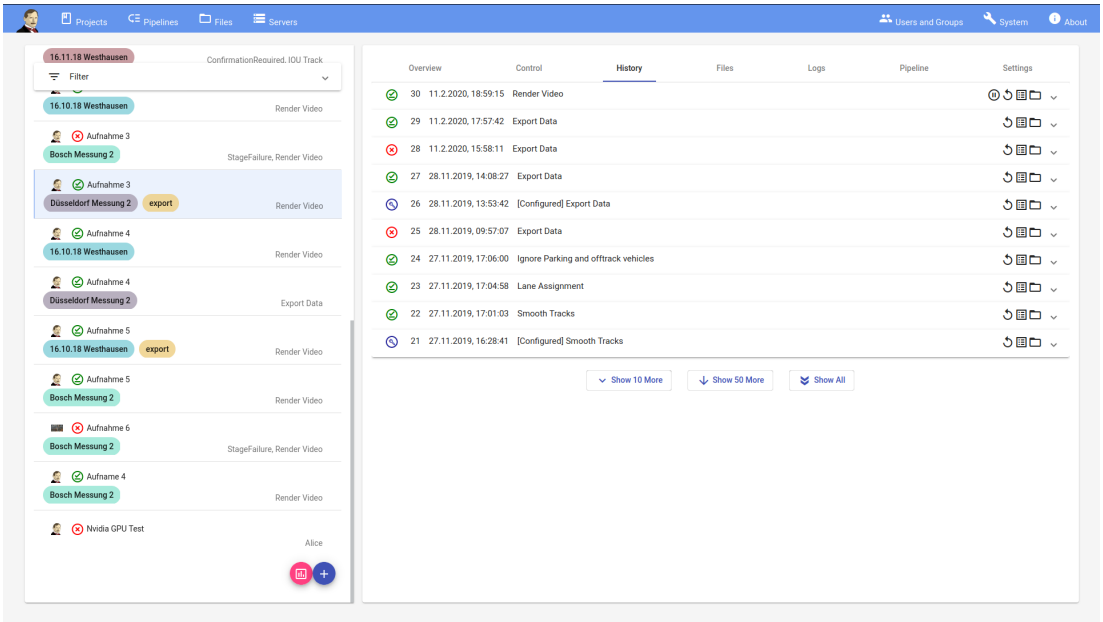

Figure D.3: Left: Known projects, Right: History of the selected project

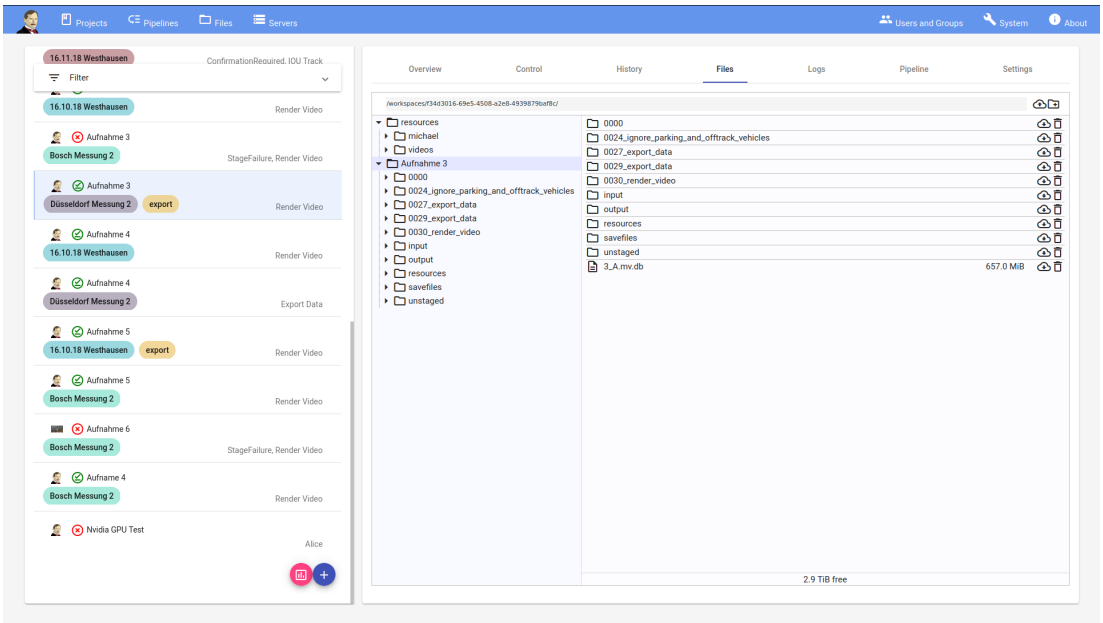

Figure D.4: Left: Known projects, Right: Files associated with the selected project

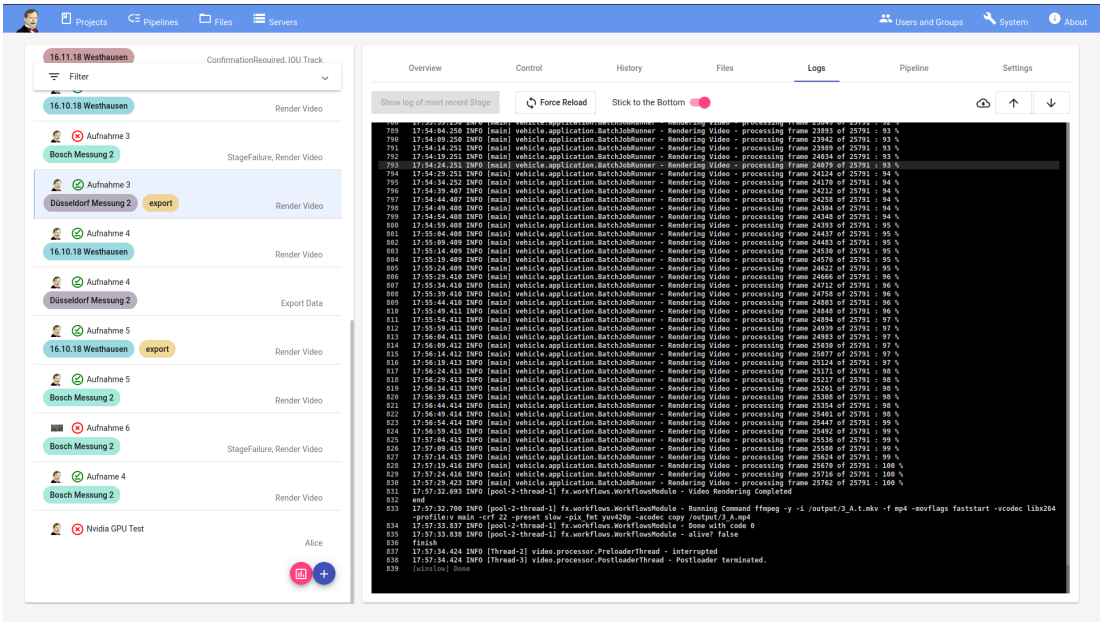

Figure D.5: Left: Known projects, Right: Most recent logs for the selected project

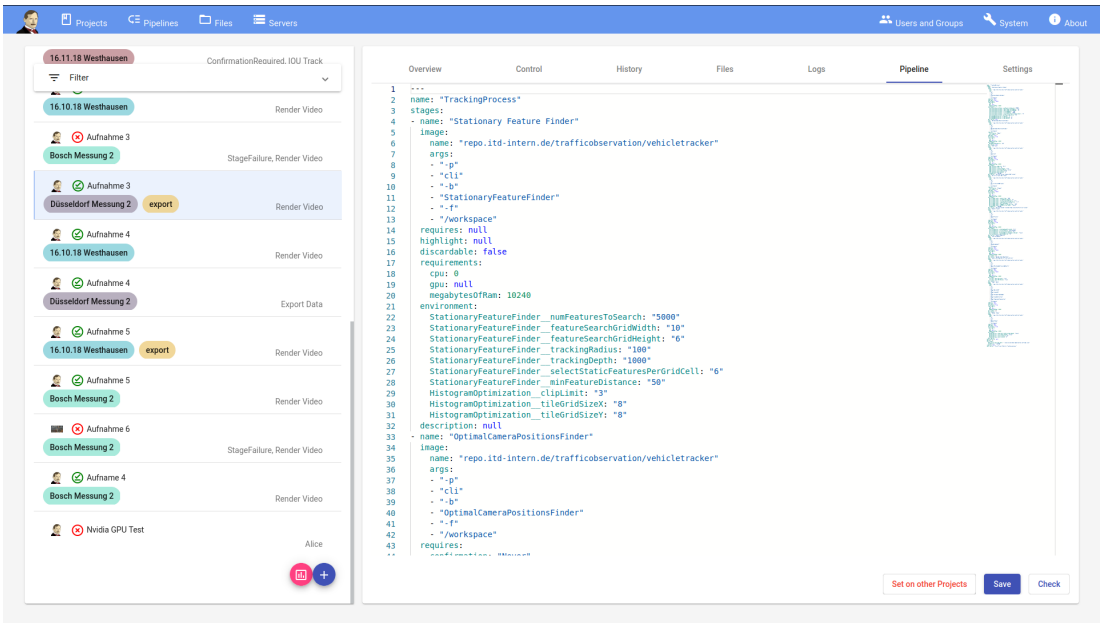

Figure D.6: Left: Known projects, Right: Pipeline of the selected project

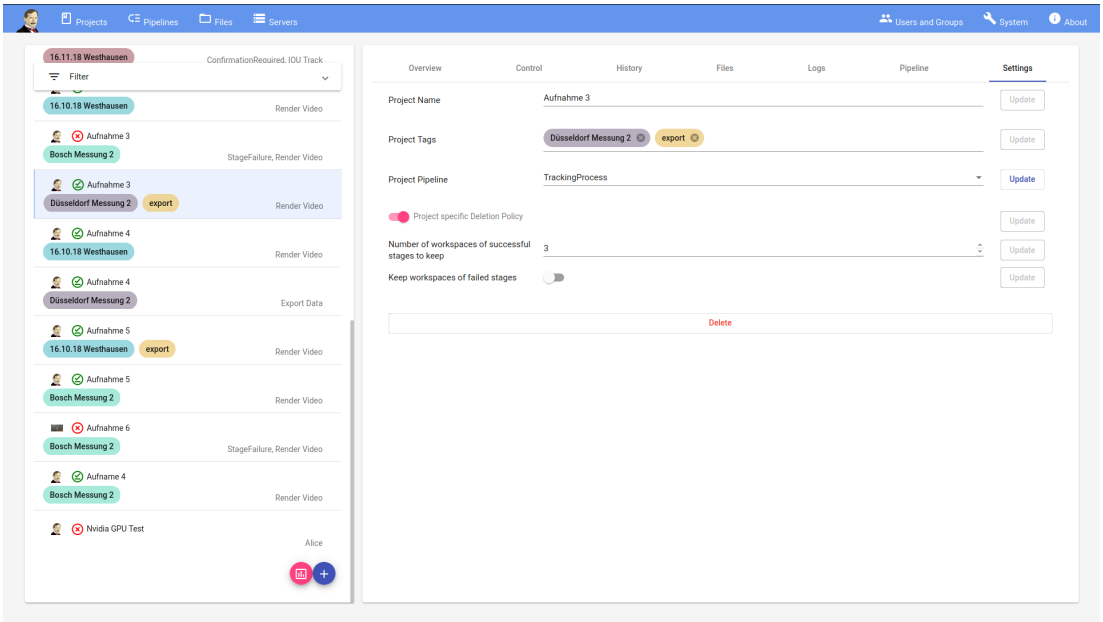

Figure D.7: Left: Known projects, Right: Settings of the selected project

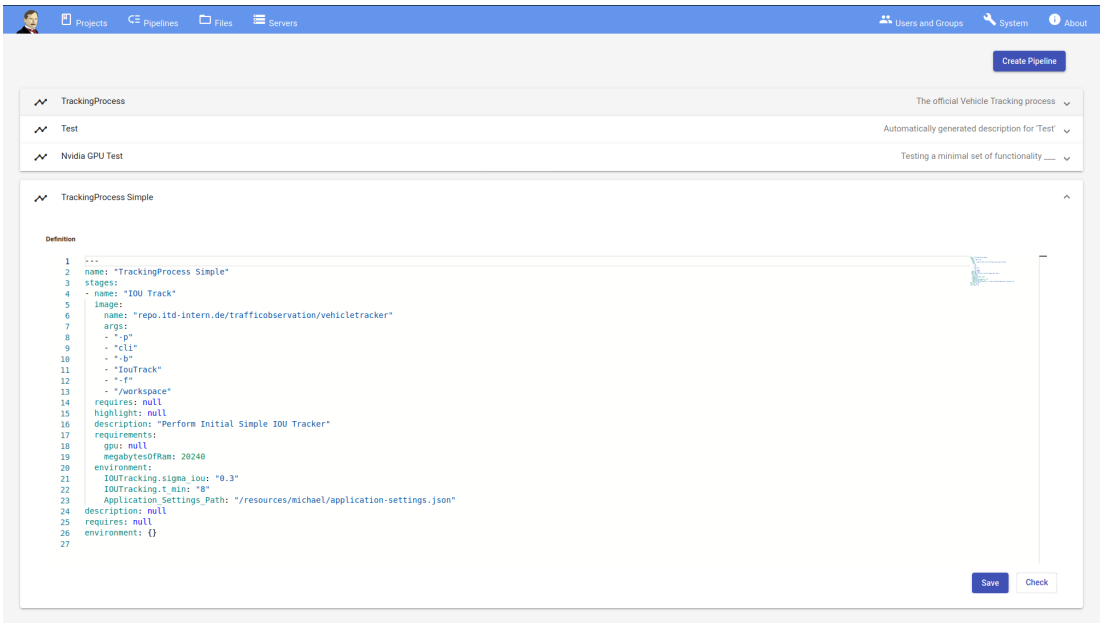

Figure D.8: Edit view for pipeline definitions

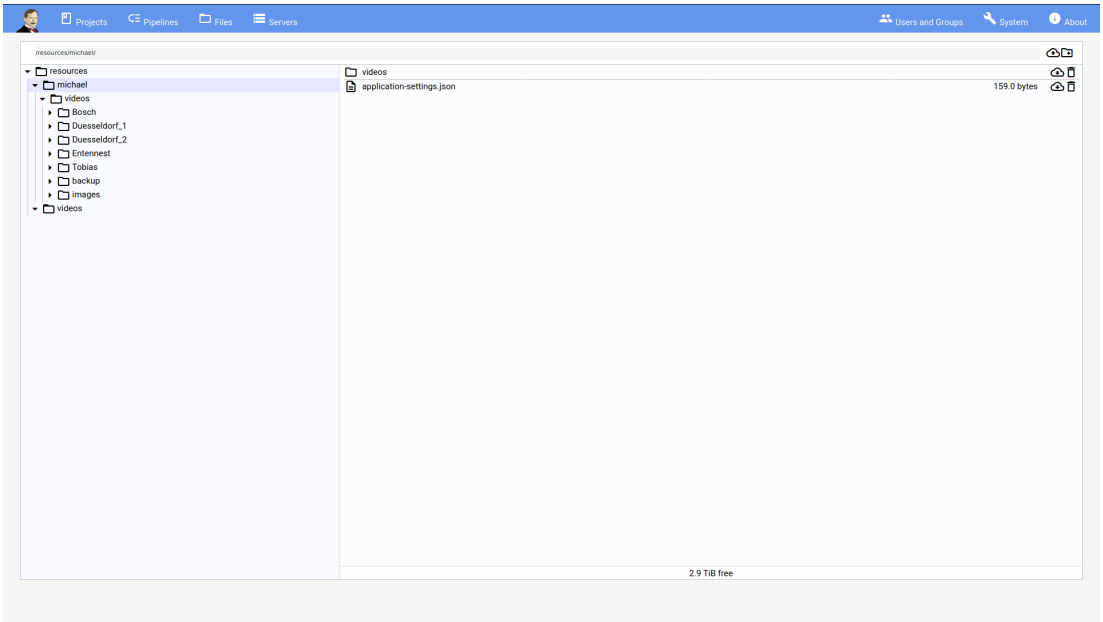

Figure D.9: File explorer which is not restricted to a project

Figure D.10: Utilization overview for every attached Winslow instance

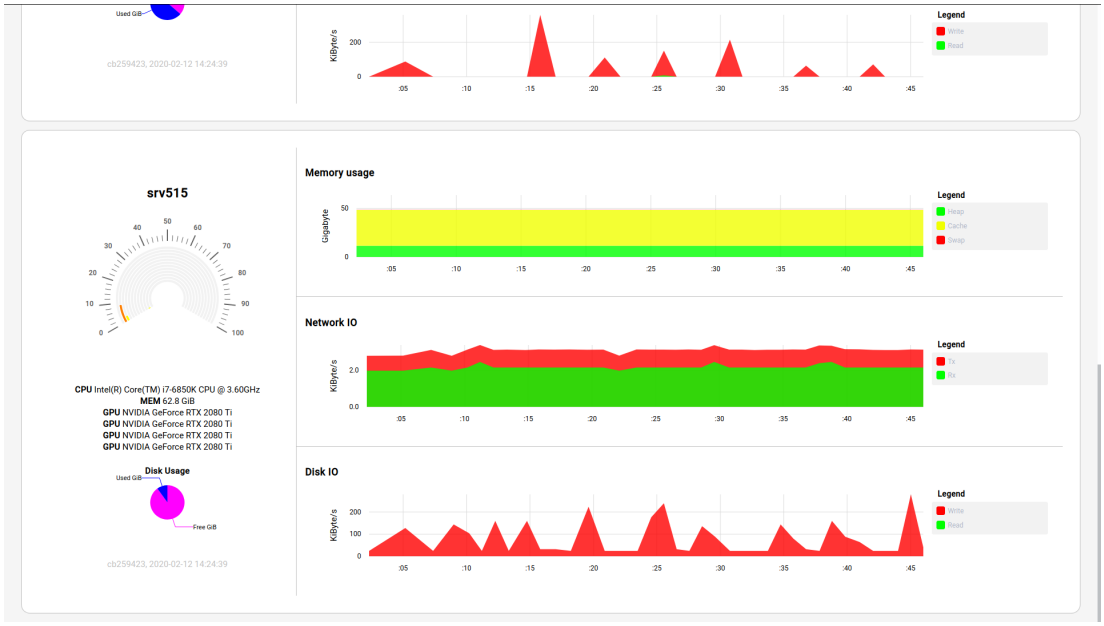

Figure D.11: Utilization overview for every attached Winslow instance

Appendix E

Interim Report

Contents

1 Introduction **1**

1.1 MEC-View 2

2 The Program **3**

2.1 Current Workflow 3

2.2 Desired Workflow 4

2.3 Deliverable Requirements 5

3 State of the art **6**

3.1 Similar solutions 6

3.1.1 Hadoop MapReduce 6

3.1.2 Build Pipelines 8

3.1.3 Camunda 8

3.1.4 Nomad 9

3.1.5 dCache 9

3.1.6 Further mentions 10

3.2 Docker Integration 12

4 Outcome and further work **13**

4.1 Storage 13

4.2 Coordination 13

4.3 Binary distribution 14

4.4 User Interface 14

5 Schedule **15**

Bibliography **18**

Chapter 1

Introduction

Since the industrial revolution, humans strive for more automation in the industry as well as in the every day life. What was at first a cost saving measurement in factories, now also is a differentiation method for products. A new product must prove a higher level comfort to the customer than the previous generation as well as all the competitors. As such, the ambitions of the industry are focused on increasing the value of their products for the customer.

The automotive industry is one of the prime examples of this. Never was traveling from one place to another as comfortable as nowadays. Aspects like an elegant interior design, comfortable seats, air conditioning, entertainment systems and safety measurements need to be considered by car manufacturers to be competitive these days. The next luxury enhancement will be the autonomously driving vehicle. No longer shall the owner of a car steer it, but instead the car becomes his or hers personal chauffeur, driving the optimal route, the most comfortable way and being more reliable and safer than any human ever could.

The reason, autonomously driving cars are not common already, is their big complexity increase. Compared to already established technologies like parking assistants, entertainment systems or more efficient engine controllers, letting a computer reliably understand a certain traffic situation requires masses of input data and complex algorithms to process. As such, the problem itself becomes massive and cannot be solved that easily. To solve this, the industry has no choice than to divide this into many small pieces and work out solutions to it step by step.

The MEC-View research project explorers one such step: whether and how to include external, steady mounted sensors in the decision finding process for partially autonomous vehicles in situations where onboard sensors are insufficient. To not disrupt traffic flow with non-human behavior, one needs to study and thereby watch human traffic. Automatically analyzing traffic from video footage requires a lot of computation power and can be further optimized by specialized hardware such as GPUs.

This thesis will conceptualize and realize a distributed and automated computer vision pipeline which can be used to analyzes traffic flow within video footage. Compared to an existing but highly manual workflow, the new system shall help to utilize the available hardware more efficiently by reducing idle times. Stage transitions and basic scheduling shall be automated to allow a user to plan and execute multiple projects ahead of time and in parallel.

1.1 MEC-View

The MEC-View research project[1] - funded by the German Federal Ministry for Economic Affairs and Energy - aims to supplement the field of view of automated driving cars with road-side sensor data using 5G mobile communication. The sensor information is merged into an environment model on the so-called Mobile Edge Computing (MEC) server. This server is directly attached to the radio station to ensure low latency environment model updates.

The project is tested at an intersection in Ulm, Germany. Currently, there are 15 lidar and video sensors installed. Those sensors send their detections to the (MEC) server. A fusion-algorithm merges those detections into one environment model and sends it back to the (MEC) server and to the automated cars.

Additionally, general traffic flow is analyzed to learn about movement patterns. To do so, 4k video data is captured by an air drone from real world cross roads. On each frame of such a recording, cars are detected with a neuronal network. Detected cars are tracked throughout the video to compute the movement speed and position in time of each car. In an analysis of all vehicles, hot-spots of high and low traffic flow can be determined.

Chapter 2

The Program

This chapter will discuss the program which shall be implemented. To do so, the problem to solve must be understood. To gather requirements and understand the technical hurdles to overcome, this chapter is split into two sections. First, a rough glance over the current workflow is given, which is followed by a more detailed description for the desired workflow.

2.1 Current Workflow

Currently, to analyze a video for the trajectories of the recorded vehicles, the following steps are executed manually:

1. Upload the input video to a new directory on the GPU server
2. Execute a shell script with the video as input file and let it run (hours to days) until completed. The shell script invokes a Java Program - called TrackerApplication - with parameters on what to do with the input file and additional parameters.
3. The intermediate result with raw detection results is downloaded to the local machine and opened for inspection. If the detection error is too high, the camera tracking has a drift or other disruptions are visible, the previous step is redone with adjusted parameters.
4. Upload the video and intermediate result to a generic computing server and run data cleanup and analysis. This is achieved with the same Java

Program as in step 2, but with different stage environment parameters.

5. Download the results, recheck for consistency or obvious abnormalities. Depending on the result, redo step 2 or 4 with adjusted parameters again.

6. Depending on the assignment, steps 4 and 5 are repeated to incrementally accumulate all output data (such as statistics, diagrams and so on).

Because all those steps are done manually, the user needs to check for errors by oneself. Also, if a execution is finished or failed early, there could be hours wasted if the regular check intervals are too far apart, such as during nights.

2.2 Desired Workflow

The desired workflow shall be supported through a rich user interface. This user interface shall provide an overview of all active projects and their current state, such as running computation, awaiting user input, failed or succeeded.

To create a new project, a predefined pipeline definition shall be selected as well as a name chosen. Because only a handful of different pipeline definitions are expected, the creation of such does not need to happen through the user interface. Instead, it is acceptable to have to manually edit a configuration file in such rare circumstances.

Once a project is created, the user wants to select the path to the input video. This file has to be been uploaded to a global resource pool at this point. The upload and download of files shall therefore also be possible through the user interface. Because a video is usually recorded in 4k (3840 x 2160 pixels), encoded with H.264 and up to 20 minutes long, the upload must be capable of handling files which are tens of gigabytes large.

Once a pipeline is started, it shall execute the stages on the most fitting server node until finished, failed or a user input is required. Throughout, the logs of the current and previous stage shall be accessible as well as uploading or downloading files from the current or previous stages workspace. In addition to the pipeline pausing itself for user input, the user shall be able to request the pipeline to pause after the current stage at any moment. When resuming the pipeline, the user

might want to overwrite the starting point to, for example, redo the latest stage.

Mechanisms for fault tolerance shall detect unexpected program errors or failures of server nodes. Server nodes shall be easily installed and added to the existing network of server nodes. Each server node might provide additional hardware (such as GPUs), which shall be detected and provided.

For the ease of installation and binary distribution, Docker Images shall be used for running the Java Program for analyzing the videos as well the to be implemented management software.

2.3 Deliverable Requirements

From the desired workflow, the following requirements can be extracted (shortened and incomplete due to early project stage):

- Rich user interface
- Storage management for global resource files as well as stage based workspaces
- Pipeline definition through configuration files
- Handling of multiple projects with independent progress and environment
- Reflecting the correct project state (running, failed, succeeded, paused)
- Log accumulation and archiving
- Accepting user input to update environment variables, resuming and pausing projects as well as uploading and downloading files into or from the global resource pool or a stages workspace.
- Assigning starting stages to the most fitting server node
- Detecting program errors (in a stage execution)
- Cope with server node failures
- Docker Image creation for the Java Binary as well as the program implementation, preferred in an automated fashion.

Chapter 3

State of the art

In this chapter, programs solving similar problems, as described in the desired workflow, or dealing with a subset of the problem are looked into. The reason for this is to use well established or suitable programs as middle-ware to reduce implementation overhead. Where this is not possible, one might be able to gather ideas and learn about proven strategies to use or pitfalls to avoid while implementing custom solutions.

3.1 Similar solutions

This sections focuses on programs trying to provide somewhat similar workflows.

3.1.1 Hadoop MapReduce

For big data transformation, Hadoop MapReduce[2] is well known. With MapReduce, the input data is split into blocks of data and distributed onto Mappers. Mappers then apply the business logic and output intermediate results in form of key/value pairs. After shuffling, the Reduce stage will combine values from key/-value pairs with the same key. In the final output, each key is unique accompanied with a value.

This strategy has proven to be very powerful to process large amount input data because Mappers and Reducers can work independently on their data-sets and therefore scale very well when adding further instances.

If the implementation were to be based on Hadoop MapReduce to achieve the desired workflow, it could be done like the following:

- Each video is split into many frames and each frame is applied to a Mapper
- A Mapper tries to detect all vehicles on a frame and outputs their position, orientation, size and so on
- The Reducer then tries to link the detections of a vehicle through multiple frames
- The final result would be a set of detections and therefore all positions for each vehicle in the video

But at the moment, this approach seems to be unfitting due to at least the following reasons:

1. It is not always trivial to reasonable link the detections of a vehicle. For example, a vehicle can be hidden behind a tree for a few frames until visible again. In addition, MapReduce requires the combination to be performed per common key. Until one is trying to link the detections of multiple frames, there is no common identifier that could easily be used as key. The position of a moving vehicle cannot be used as key, neither can the color or size, because of the noise of the camera, deviation in detection output and perspective distortions. The current implementation of the TrackerApplication is archiving this by finding similarities between detections, but for the Mapper it would be required to express this as a deterministic key.
2. MapReduce is great in combining many machines to solve a big computational problem. But at the moment, this is neither a desired nor given condition. At the moment, there are a handful of very powerful workstations with specialized hardware. Therefore it is perfectly acceptable and sometimes required, when each workstation works through a complete video at a given time instead.

3.1.2 Build Pipelines

Build pipelines such as GitLab[3] and Jenkins[4] can also distribute the execution of stages onto other server nodes. In a common use-case, such build pipelines are used to build binaries out of source code, after a new commit into a SCM[1] repository was made. At IT-Designers GmbH GitLab as well as Jenkins are commonly used for scenarios exactly like this. A pipeline definition in GitLab CI/CD [5] or in a Jenkinsfile [6] describe stages and commands to execute. Each stage can be hosted on another node and be executed sequential or in parallel to each other.

Although this seems to be quite fitting for the desired workflow, there are two issues. First of all, such a pipeline does not involve any user input besides an optional manual start invocation. The result is then determined based on the state of the input repository. Second, such a pipeline is designed to determine the output (usually by compiling) whereas each run is independent from the previous and a repeated run shall provide the same result as the previous did. Usually, a new run is only caused by a change of the input data. However, the desired workflow differs in this aspects. A redo of a stage can depend on the result of the previous stage, for example, if the results are poor or the the stage failed. Instead of having multiple complete pipeline runs per project, the desired workflow uses a pipeline definition as base for which the order can be changed. Also, intermediate results need to influence further stages, even if repeated.

3.1.3 Camunda

Camunda[7] calls itself a "Rich Business Process Management tool" and allows the user to easily create new pipelines by combining existing tasks with many triggers and custom transitions. Camunda is focused upon visualizing the flow and tracking the data through a pipeline. The Camundas Process Engine[8] also allows user intervention between tasks.

One of the main supporting reason for it Camunda is the out of the box rich graphical user interface for process definition and interaction. Through its API[9], Camunda also allows custom external workers to execute a task. But it misses the

[1]Source Code Management

capability to control which task shall be processed on which worker node which is required by the desired workflow. It does also not provide any concept on how to allocate and distribute resources. The user interface - while being rich overall - is quite rudimentary when it is about configuring tasks and would therefore require custom plugins to be developed for more advanced user interactions.

Camunda is also not designed to reorder stages or insert user interactions at seemingly random fashion. The user itself is considered more as a worker that gets some request, "executes" this externally and finally marks the request as accepted or declined. Mapping this to the desired workflow does not feel intuitive. Finally, there is also no overview of task executors, no centralized log accumulation and no file up- or download for global project resources.

3.1.4 Nomad

Nomad[10] by HashiCorp is a tool to deploy, manage and monitor containers, whereas each job is executed in its own container. It provides a rich REST API and can consider hardware constraints on job submissions. Compared to Kubernetes[11], which is similar but more focused on scaling containers to an externally applied load, it is very lightweight. It is also available in many Linux software repositories - such as for Debian - which makes the installation very easy.

Because there were no grave disadvantages found (depending on a third party library can always be considered be a disadvantage for flexibility, error-pronous and limit functionality) Nomad is being considered as a middle-ware to manage and deploy stages. Others[12] seem to be using Nomad to manage and deploy containers for similar reasons. Nonetheless, further testing and prototyping will be required for a final decision.

3.1.5 dCache

"The goal of this project is to provide a system for storing and retrieving huge amounts of data, distributed among a large number of heterogenous server nodes, under a single virtual filesystem tree with a variety of standard access methods"[13]. dCache seems to be able to solve the storage access and distribution concern for the stages and sever nodes. When using dCache, one could store the

global resources distributed between the server nodes. Built-in replication would prevent access loss on a node or network failure and an export through NFS[2] allows easy access for Linux based systems[14].

But the installation is complex and requires many services to be setup correctly, such as postgresql and many internal services such as zookeeper, admin, poolmanager, spacemanager, pnfsmanager, cleaner, gplazma, pinmanager, topo, info and nfs. The documentation is also rather outdated and incomplete which meant, early tests with a prototype setup took days to setup and behaved rather unstable (probably due to a wrong configuration). It is to be seen, whether such an complex and heavy system is actually required or if there are feasible alternatives.

3.1.6 Further mentions

The following list shall acknowledge programs that behave similar to the previously mentioned strategies. Programs that are listed here, were looked into, but not in-depth because miss-fits were detected early on (listed in no specific order):

- **Quartz**[15] is a Java based program to schedule jobs. Instead of doing so by using input, Quartz executes programs through a timetable and in certain intervals.
- **Luigi**[16] also executes pipelines with stages and is written in python. The advertised advantage is to define the pipeline directly in python code. But, this is at the same time the only way to define pipelines which contradictions with the existing Java TrackerApplication implementation.
- **Calery**[17] is focused on task execution through message passing and is written in Python. Intermediate results are expected to be transmitted through messages. Because there is no storage strategy and python adapter-code would have been required, Calery was dismissed.
- **IBM InfoSphere**[18] provides similar to Camunda a rich graphical user interface but for data transfer. Dismissed due to commercial nature.

[2]Network File System

- **qsub**[19][20] is a CLI[3] used in HPC to submit jobs onto a cluster or grid. Dismissed due to an expected high setup overhead, non-required multi-user nature and the fact, that it only provides a way to submit jobs.

- **CSCS**[21] High Throughput Scheduler (GREASY). Dismissed for similar reasons as qsub, although it is more light weight and hardware agnostic (it can consider CUDA/GPU requirements).

- **zsync**[22], similar to rsync, is a file transfer program. Zsync allows to only transfer new parts when a file that shall be copied already exists in an older version on the target. This tool might be useful when implementing a custom resource distribution strategy is required.

- **OpenIO**[23] provides a distributed file system, is already provided as Docker image and provides a simple to use CLI. Because the NFS export is only available through a paid subscription plan, it was dismissed from further investigation.

- **SeaweedFS**[24] provides a scalable and distributed file system. The most interesting aspects are that it is rack-aware as well as natively supports external storage such as Amazon S3. When adding server nodes from the cloud this could allow all nodes to access the same file system while using rack-aware replication to reduce bandwidth usage and latency. A local test also proved that it is easy to setup, but because it cannot hot-swap nodes and was not able to recover when the seeweed master node became unreachable it was dismissed.

- **Alluxio**[25] provides a distributed file system but was dismissed because it itself requires a centralized file system for the master and its fallback instances

- **GlusterFS**[26] is another tool to provide a distributed file system with replication. It was bought by IBM but is nonetheless available through the software repository of many Linux distributions such as Debian. A local test showed that the setup is very easy and no adjustments of configuration files

[3]Command Line Interface

are required. However, the replication mechanism requires that an integer multiple of nodes of the replica value are assigned to the file system. This makes GlusterFS hard to use in a scenario, where adding and removing nodes are expected to happen frequently. It was therefore dismissed.

3.2 Docker Integration

As describe before (see section 2.2), for easy deployment, the implementation as well as the stages shall be executed inside Docker[27] containers. This allows easier isolation of the stages and workspaces from each other and other host programs. Because one needs to communicate with the Docker daemon, this increases the complexity for the implementation. But by using third party libraries, the increase in complexity can be limited.

Chapter 4

Outcome and further work

In this chapter the main concerns are listed. For each concern the current progress is described as well as further work that needs to be done.

4.1 Storage

One of the central concerns is the storage management. The program needs to make input files available on each execution node and collect the results once the computation is complete. There are a few main architectural strategies to approach this. Simplified, either at a centralized location which is accessed by all execution nodes, a copy of the input files to the execution nodes or decentralized and distributed between all execution nodes and replication. The advantages and disadvantages can depend on the specific implementation and is therefore discussed in combination of such (see chapter 3).

Further testing is required to decide whether a more complex storage system is required, or the simplicity of a centralized solution outweighs the setup and maintenance overhead.

4.2 Coordination

Another important concern is the coordination of the nodes. A central coordinator with external server nodes, such as GitLab and Jenkins have, might not be sufficient for more complex and longer lasting pipelines. The probability that the

master would need to be offline while there is a stage executed, is in the scenario of the desired workflow higher than for GitLab or Jenkins, because the stage is being execution for hours or days. Coupling stage execution plans on node availability ahead of time, as well as recovering from a sudden master failure implies additional implementation complexity. A decentralized coordination needs to be able to do this as well, but also allows the usage of the system while a node failed or is unreachable due to maintenance. With further prototyping and research a reasonable solution shall be found.

4.3 Binary distribution

In a time where containers are common and have proven to be usable, the installation of the binaries directly on the operating system they are executed on shall be avoided. There shall be no manual, nor automatic but custom file copies of the binaries or images from one server to the other. Experience shows, that without a proper management, this can easily become a mess, in which it is no longer clear, which files or images belong to which version. At the same time, making all binaries publicly available through the Docker Hub[28] is no option either. Whether a self hosted Docker Registry[29] could be the solution to this will be determined in further testing.

4.4 User Interface

Providing a useful user interface might not be important to the functionality of the system itself but for the user experience. A bad user experience will cause a system not to be used. It became common practice for a rich user experience to be web based and interactive with JavaScript. For a potentially decentralized system, it is also advantageous to be able to access a disconnected node in the same manner as the remaining system, which further encourages a web based solution. Web based solutions such as React and Angular shall therefore be investigated for being used as user interface.

Chapter 5

Schedule

The following figure shows schedule for further and past work:

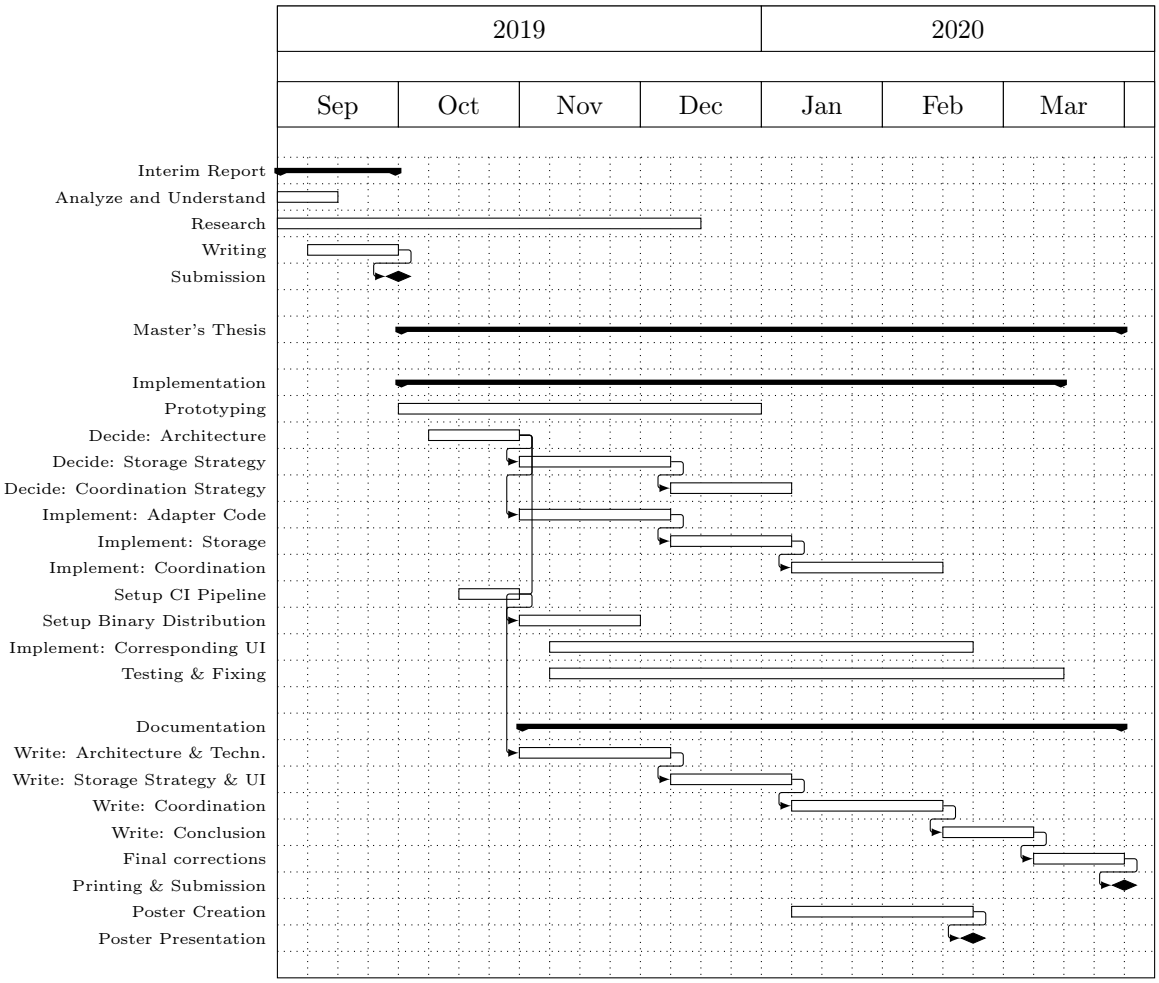

Figure 5.1: Time schedule

Bibliography

[1] MEC-View Dr. Rüdiger W. Henn. *Mobile Edge Computing basierte Objekterkennung für hoch- und vollautomatisiertes Fahren.* URL: `http://mec-view.de/` (visited on 09/29/2019).

[2] The Apache Software Foundation. *Apache Hadoop.* URL: `https://hadoop.apache.org/` (visited on 09/19/2019).

[3] Inc. GitLab. *The first single application for the entire DevOps lifecycle.* URL: `https://about.gitlab.com/` (visited on 09/21/2019).

[4] jenkins.io. *Jenkins. Build great things at any scale.* URL: `https://jenkins.io/` (visited on 09/19/2019).

[5] Inc. GitLab. *GitLab CI/CD. Pipeline Configuration Reference.* URL: `https://docs.gitlab.com/ee/ci/yaml/` (visited on 09/19/2019).

[6] jenkins.io. *Using a Jenkinsfile.* URL: `https://jenkins.io/doc/book/pipeline/jenkinsfile/` (visited on 09/19/2019).

[7] Camunda Services GmbH. *Workflow and Decision Automation Platform.* URL: `https://camunda.com/` (visited on 09/19/2019).

[8] Camunda Services GmbH. *Process Engine API.* URL: `https://docs.camunda.org/manual/7.6/user-guide/process-engine/process-engine-api` (visited on 09/19/2019).

[9] Camunda Services GmbH. *Rest Api Reference.* URL: `https://docs.camunda.org/manual/7.8/reference/rest` (visited on 09/19/2019).

[10] HashiCorp. *Nomad.* URL: `https://www.nomadproject.io/` (visited on 09/19/2019).

[11] HashiCorp. *Nomad. Nomad vs. Kubernetes.* URL: https://www.nomadproject.io/intro/vs/kubernetes.html (visited on 09/19/2019).

[12] Bc. Pavel Peroutka. *Web interface for the deployment andmonitoring of Nomad jobs. Master's thesis.* URL: https://dspace.cvut.cz/bitstream/handle/10467/80106/F8-DP-2019-Peroutka-Pavel-thesis.pdf (visited on 09/22/2019).

[13] Tigran Mkrtchyan Patrick Fuhrmann. *dCache. Scope of the project.* URL: https://www.dcache.org (visited on 09/19/2019).

[14] Patrick Fuhrmann. *dCache, the Overview.* URL: https://www.dcache.org/manuals/dcache-whitepaper-light.pdf (visited on 09/19/2019).

[15] Inc. Terracotta. *QuartzJob Scheduler.* URL: http://www.quartz-scheduler.org/ (visited on 09/19/2019).

[16] Data Revenue. *Distributed Python Machine Learning Pipelines.* URL: https://www.datarevenue.com/en/blog/how-to-scale-your-machine-learning-pipeline (visited on 09/19/2019).

[17] Ask Solem. *Celery: Distributed Task Queue.* URL: http://www.celeryproject.org (visited on 09/19/2019).

[18] IBM Knowledge Center. *IBM InfoSphere. DataStage.* URL: https://www.ibm.com/support/knowledgecenter/en/SSZJPZ_9.1.0/com.ibm.swg.im.iis.ds.design.doc/topics/c_ddesref_Server_Job_Stages_.html (visited on 09/19/2019).

[19] Wikipedia contributors. *Qsub. Wikipedia, The Free Encyclopedia.* URL: https://en.wikipedia.org/w/index.php?title=Qsub&oldid=745279355 (visited on 09/22/2019).

[20] The University Of Iowa. *Basic Job Submission. HPC Documentation Home / Cluster Systems Documentation.* URL: https://wiki.uiowa.edu/display/hpcdocs/Basic+Job+Submission (visited on 09/22/2019).

[21] Swiss National Supercomputing Centre. *High Throughput Scheduler.* URL: https://user.cscs.ch/tools/high_throughput/ (visited on 09/19/2019).

[22] Colin Phipps. *zsync. Overview.* URL: http://zsync.moria.org.uk/ (visited on 09/19/2019).

[23] OpenIO. *High Performance Object Storage for Big Data and AI.* URL: https://www.openio.io/ (visited on 09/22/2019).

[24] Chris Lu. *Simple and highly scalable distributed file system.* URL: https://github.com/chrislusf/seaweedfs (visited on 09/22/2019).

[25] Colin Phipps. *Alluxio. Data Orchestration for the Cloud.* URL: https://www.alluxio.io (visited on 09/19/2019).

[26] Inc Red Hat. *Gluster. Free and open source scalable network filesystem.* URL: https://www.gluster.org (visited on 09/19/2019).

[27] Inc. Docker. *Enterprise Container Platform.* URL: https://www.docker.com/ (visited on 09/22/2019).

[28] Inc. Docker. *Docker Hub. Build and Ship any Application Anywhere.* URL: https://hub.docker.com/ (visited on 09/22/2019).

[29] Inc. Docker. *Docker Documentation. Docker Registry.* URL: https://docs.docker.com/registry/ (visited on 09/22/2019).