universität innsbruck

Leopold–Franzens–University
Innsbruck

Institute of Computer Science
Research Group DPS
(Distributed and Parallel Systems)

Multi-layer enactment engine for workflow applications

Bachelor Thesis

Florian Maier 01617122

Supervisor: Dr. Sashko Ristov

Innsbruck
4 March 2020

Abstract

In addition to HPC systems such as clusters or grids, scientists have also been using the cloud to run their scientific programs for some time. The cloud became a great way to run scientific programs that follow a workflow. Cloud computing providers such as Amazon Web Services (AWS) offer numerous methods that customers can use to run their programs optimally while saving both costs and time. Different resources are offered as a service, which differ in price and performance. This presents researchers with the task of determining the appropriate resource for their problem.
This bachelor thesis deals with how to combine different AWS services in order to save as much time and money as possible. To achieve this goal, an enactment engine is designed, with which a workflow-based program is shared between AWS EC2 instances and containers that are managed by AWS ECS. The aim is to combine the advantage of EC2, namely low prices, with the advantage of containers, namely fast execution.

Contents

List of Figures

1 Introduction

Workflows - often used in scientific applications - are a good way to solve computationally intensive tasks. Not surprisingly, over the past two decades, many developers have chosen to run their application in the cloud [12].

Cloud Computing is an integral part of many applications and services. Organizations use the cloud to store data, develop and test software, analyze data, perform calculations and much more [4].
With cloud computing, you no longer have to provide and maintain hardware. Instead, you can request the resources you need at any time. You only pay for what you use. As a result, the resources also seem to be infinite. No matter how much your program needs, you can simply request it [4][12].
The right use of cloud computing can save a lot of costs and increase efficiency. This is the reason why both large companies and start-ups use such services [5].

The aim of this bachelor thesis is to build an enactment engine which is used to optimize costs and performance of workflow applications. To achieve this goal, the scientific application Montage [15] will run on two different services from one of the most used cloud providers, Amazon Web Services (AWS) [3]. One of these services, Elastic Compute Cloud (EC2), is cheaper, but slower. The other service, Fargate, uses Container to be faster, but more expensive. Depending on the resource consumption of a job in the workflow, this job is executed on one of the two services. The combination of both services should lead to a general improvement in costs and performance.

The purpose of this work is further explained in chapter Motivation. Several services and concepts were used for this work. The main concepts are workflows, cloud computing and container. The chapter Background will explain and dis-

cuss them for those who are not familiar with them. After giving all necessary background information, chapter Implementation will explain how the enactment engine was implemented. Not only the procedure how the program was developed is discussed, but also problems that have arisen. Finally, in chapter Results we will show and discuss the results.

2 Background

2.1 Workflow Applications

Scientific Applications often make use of workflows. They are ideally suited for data analyses and orchestrating complex simulations [13]. Workflows are used to easily express multi-step computational tasks, for example retrieve data, reformat it and run an analysis on it [16]. Scientists used different options to run their application, usually academic HPC systems such as clusters and grids, or supercomputers and local workstations [12][13]. Cloud computing has also become an option in the past two decades.

A workflow is a directed asymmetric graph (DAG) which consists of a number of Tasks and dependencies between them [17][16]. An example of a DAG can be seen in figure 2.1.

Tasks and dependencies are defined by a workflow description written in a specific language such as XML and Json. For tasks, things like runtime and CPU and memory requirements are usually defined. Dependencies describe the file input/output, the network transfers and synchronization.
The tasks communicate through files which are either stored in a shared file system, or sent between the workflow nodes by the workflow management system. Each task requires a number of input files and generates some output files which are used as input for other tasks [13]. In workflow applications the data-transfer consists only of the input and output files. Executables are usually pre-installed.

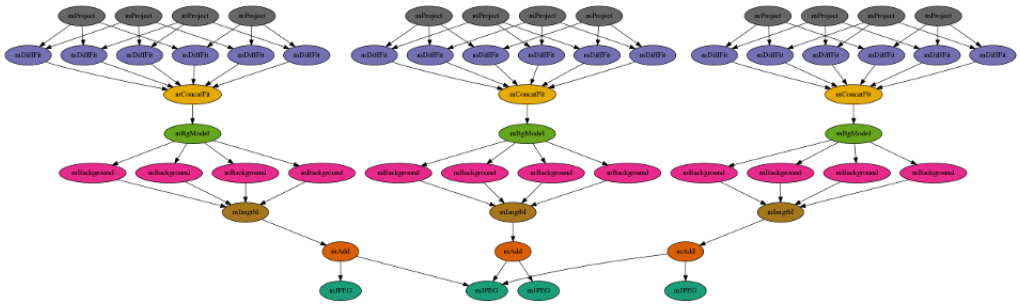

Figure 2.1: Montage Workflow as an example for a DAG [19]

2.2 Montage

Montage is used in this work as an example of a workflow application and is therefore the starting point for all tests and results.

Montage, an astronomical image mosaic engine, is a toolkit to create image mosaics using data collected from telescopes [15]. The montage workflow can have different sizes depending on how big the part of the sky is (measured in square degrees) which is covered by the output mosaic [12]. For this bachelor thesis I configured Montage to generate a 1.42-degree mosaic, resulting in a workflow containing 58 tasks.

Pegasus is a tool for helping developers working with workflows [16]. Since Pegasus is not used directly in this thesis, it will not be discussed in further detail. Rather, this work uses the Montage workflow generator, which uses Pegasus and creates a workflow description for Montage [19]. The generated workflow description ("DAX file") looks something like figure 2.2.

2.3 Container

To better understand the next chapter, the concept of containers is now explained. Even if we are talking about containers, we are always talking about Docker containers. Docker is an open source application that makes working with containers easier [14].

So, what is a container? Every developer knows that dependencies can cause

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated: 2019-10-24 15:46:22.253809 -->
<!-- generated by: florian -->
<!-- generator: python -->
<adag xmlns="http://pegasus.isi.edu/schema/DAX" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="3.4" name="montage">
    <invoke when="on_success">/usr/share/pegasus/notification/email --report=pegasus-statistics</invoke>
    <invoke when="on_error">/usr/share/pegasus/notification/email</invoke>
    <invoke when="start">/usr/share/pegasus/notification/email</invoke>
    <transformation name="mDiffFit">
        <uses name="mDiffFit"/>
        <uses name="mFitplane"/>
        <uses name="mDiff"/>
    </transformation>
    <job id="ID0000001" name="mProject">
        <argument>-X <file name="poss2ukstu_blue_001_001.fits"/> <file name="region-oversized.hdr"/></argument>
        <uses name="pposs2ukstu_blue_001_001.fits" link="output" transfer="false"/>
        <uses name="pposs2ukstu_blue_001_001_area.fits" link="output" transfer="false"/>
        <uses name="poss2ukstu_blue_001_001.fits" link="input"/>
        <uses name="region-oversized.hdr" link="input"/>
    </job>
    ...
    <job id="ID0000058" name="mViewer">
        <argument>-red <file name="3-mosaic.fits"/> -1s 99.999% gaussian-log -green <file name="2-mosaic.fits"/></argument>
        <uses name="mosaic-color.jpg" link="output" transfer="true"/>
        <uses name="3-mosaic.fits" link="input"/>
        <uses name="2-mosaic.fits" link="input"/>
        <uses name="1-mosaic.fits" link="input"/>
    </job>
    <child ref="ID0000005">
        <parent ref="ID0000001"/>
        <parent ref="ID0000002"/>
    </child>
    ...
    <child ref="ID0000058">
        <parent ref="ID0000018"/>
        <parent ref="ID0000037"/>
        <parent ref="ID0000056"/>
    </child>
</adag>
```

Figure 2.2: shortened version of a DAX file

problems. And containers solve exactly these problems. A container is comparable to a box. Everything that has to do with each other is put in this box. Then you can use this box anywhere. You can be sure that it works the same everywhere. Container solve three problems:

- conflicting dependencies: if you need two different versions of a software, you simple put each version in a separate container
- missing dependencies: all necessary dependencies are inside the container. So, you can run it anywhere without the need to install missing packages
- platform differences: in this case you can compare container with java. You can run your container on every OS as long as Docker is installed

In contrast to hypervisors, another method of virtualization, which work directly on the hardware, containers virtualize at the operating system level. Two containers running on the same OS don't know anything about each other and that they are sharing resources. Using hypervisors you always need an OS. So, using hypervisors means running multiple operating systems at the same time. This

takes up a lot of resources. However, using containers, there is only one OS running (as their host OS). Containers act like processes: if they are not executing anything, they will not consume any resources. And since you don't have to start or stop an entire OS, it is really fast to start and stop containers.

2.4 Cloud Computing

In this section we will define the term "cloud computing" and talk about the cloud computing provider used in this thesis: Amazon Web Services (AWS). Moreover, the AWS services used in this work are discussed.

To define "Cloud Computing" let's refer to the definition of Boss et al. (2007) [18]: "a Cloud is a pool of virtualized computer resources" and a cloud allows "the dynamic scale-in and scale-out of applications by the provisioning and de-provisioning of resources, e.g. by means of virtualization".
So, cloud computing simply means run your application on virtualized computers, requesting the resources you need.

Clouds give developers new advantages over the HPC systems used otherwise [11]:

- root access: the user has root access to the operating system. This gives the developer full control over the system.
- VM images: the developer can create his own virtual machine images. This advantage is used in this thesis to run AWS instances with all necessary software already installed.
- on-demand provisioning: the developer can request an instance at any time with exactly the resources that he needs. It is a great advantage to save costs.

Amazon Web Services is the biggest cloud computing provider in the world [10]. It does not just offer virtual servers, scalable storage and databases, but much more services such as machine learning and artificial intelligence, data lakes and

analytics, and Internet of Things.
AWS offers its services to every kind of customers: startups, enterprises and public sector organizations. Millions of active customers and tens of thousands of partners worldwide benefit from the services of AWS.

This bachelor thesis made use of three AWS services, presented in the following subsections.

2.4.1 Elastic Compute Cloud

The AWS service Elastic Compute Cloud – short: EC2 – is a heavily used part of the cloud computing provider. With EC2, users can rent virtual servers on which they can run their applications. With EC2, Amazon promises secure and expandable virtual compute capacity.[6]. EC2 offers many advantages:

- You don't have to invest in hardware anymore
- So, you are faster developing and deploying your application
- You can request as many servers as you need whenever you need
- You have complete control over the resources you use
- You can create images of your running EC2 instances
- You can configure security and networking, and manage storage
- And much more... [7]

EC2 offers a huge amount of different instance types. These differ in the number of vCPUs (virtual CPU), memory, storage and network performance.

In this work EC2 is used as one of two services to run Montage in the cloud. Compared to the second service, Elastic Container Service + Fargate, EC2 is cheaper, but slower.

2.4.2 Elastic Container Service

The second AWS service used to run Montage is Amazon Elastic Container Services (ECS) [8]. In contrast to EC2, ECS manages container. What a container is has already been discussed in section Container.
For this task ECS offers two different options: EC2 or Fargate. Using the first option, the user has to provide a pool of EC2 instances. However, since only Fargate is used in this work, this first option will not be discussed further.
ECS Fargate is serverless compute for containers. That means you don't have to provide or manage any servers. The only things the user has to do are:

- package his application in containers
- specifying the amount of resources (CPU and memory) needed for every container
- defining networking and Identity and Access Management (IAM) policies

After that, the application is ready to launch. Figure 2.4 illustrates how ECS works.
If you use ECS, there are four different levels: Cluster, Service, task definition and container definition. See Figure 2.3 for illustration. Chapter Implementation explains what each level does and how it was integrated into the engine.

Compared to EC2, ECS Fargate is faster, but more expensive.

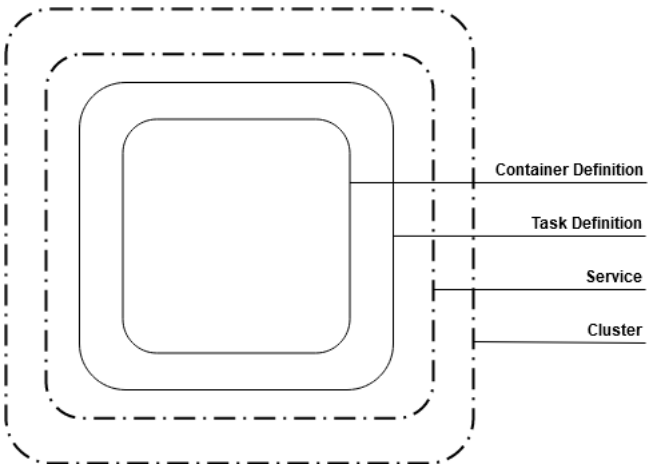

Figure 2.3: Different levels of ECS

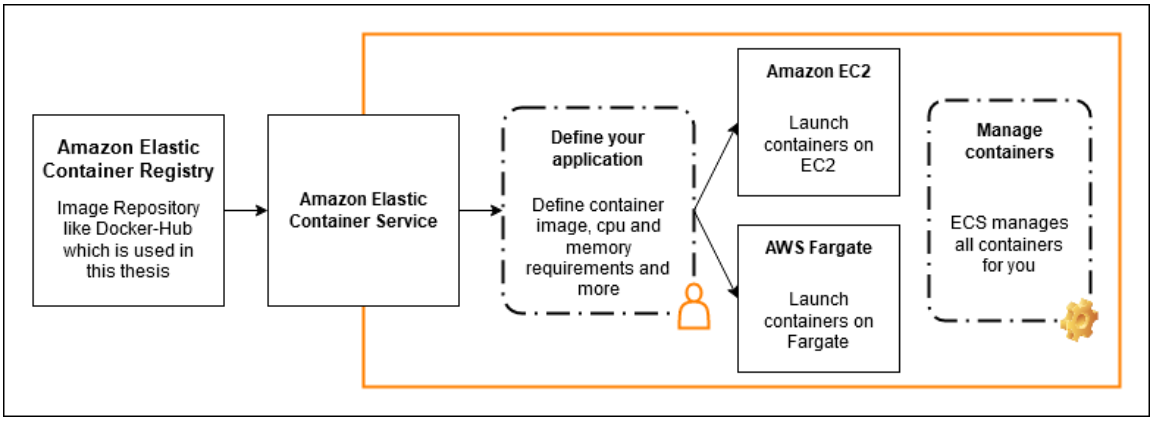

Figure 2.4: Illustration how ECS works

2.4.3 Simple Storage Service

As mentioned in section Workflow, workflows typically communicate through files. This is where the third service comes into play: Amazon Simple Storage Service (S3) [9]. Amazon S3 is an object storage service that offers high scalability, data availability, security and performance.
This is how it works: The user uploads a file to an S3 bucket. Thereupon this file is accessible via an url.

In this work every Montage-workflow-job downloads all input files from S3, processes the data and uploads all output files again on S3. Since the cost of S3 is very low, a run through the workflow costs less than a cent.

3 Motivation

The goal of this bachelor thesis is to lower the costs and to increase the performance of workflow applications when executing in the cloud. This chapter will explain why this is important.

3.1 Performance

Scientific applications are often very performance-intensive. Some tasks might need hours, days or months to finish. It is therefore necessary to carry out the tasks as efficiently as possible. The best example was recently provided by Google with the first successes in building a quantum computer [2]. A research team of Google, led by John Martinis, an experimental physicist at the University of California, created a quantum computer which could solve a 10.000 years task in 200 seconds. Even if my program is not as efficient as a quantum computer, it reduces the makespan by something.

3.2 Cost Savings

However, as performance improves, so does the cost. And rightly, not every company is willing to pay more for some time savings. Cloud computing does not solve this problem, but improves the situation. The keyword is "pay-as-you-go". If you have your own hardware, you always pay for everything, even if you don't use everything. Using cloud computing, you only pay for what you use.
As mentioned in the Introduction I will use two different services of AWS. They have different advantages and disadvantages. Combining them leads to an improvement of the performance and sometimes it is even cheaper than the slower execution.

3.3 Flexibility through cloud computing

By using a cloud computing provider like AWS it is very easy to change system components. Would you like test your program on another operating system, you do not have to set up a new operating system, but can simply select another instance.

3.4 Simplification of multilayered Execution

Imagine having a workflow application. The workflow contains a few hundred jobs and the input data is very small. For reasons explained in the course of the thesis, you decide to run the program on EC2. So you align your program according to the API of EC2. Suddenly the input files change so that the execution on EC2 is too slow. In order to switch to ECS, which would now be the better choice, the entire API must now be changed. There are tools that optimize the portability of applications for clouds. An example of this would be Apache jclouds. However, you then have to decide on which cloud service you want to run your entire workflow. Imagine now that the input files would change again. Half of them are very small, but others are very large. It would be best to execute the small jobs on EC2 and the large jobs on ECS. This is where this enactment engine comes in. It allows you to choose which service to run for each job. You don't have to worry about the different APIs.

4 Implementation

This chapter deals with the implementation of the enactment engine.

The goal is to develop an enactment engine with which workflow applications can be executed in the cloud as cheaply and quickly as possible. Two AWS services are combined for this: EC2 and ECS Fargate. By combining the two services, their advantages should also be combined: velocity and affordability. ECS and EC2 are used for executing tasks of Montage, while AWS S3 is used to store intermediate results.

The user should be able to indicate how he would like to combine the services to go either the fastest, cheapest or a quick and cheap way. Depending on the type of application and what it is used for, each of these three ways can be beneficial. Therefore, the aim of this thesis is to find out how the two services can best be combined.

The engine was completely written in Java, using XML as language for two input files (DAX and settings) and bash when working on instances.

4.1 Overview

In this section we will talk about how the enactment engine works. This section will only give you an overview. In the next section, the main parts of the engine will be discussed.

Figure 4.1 shows the system architecture, which is described in this chapter.

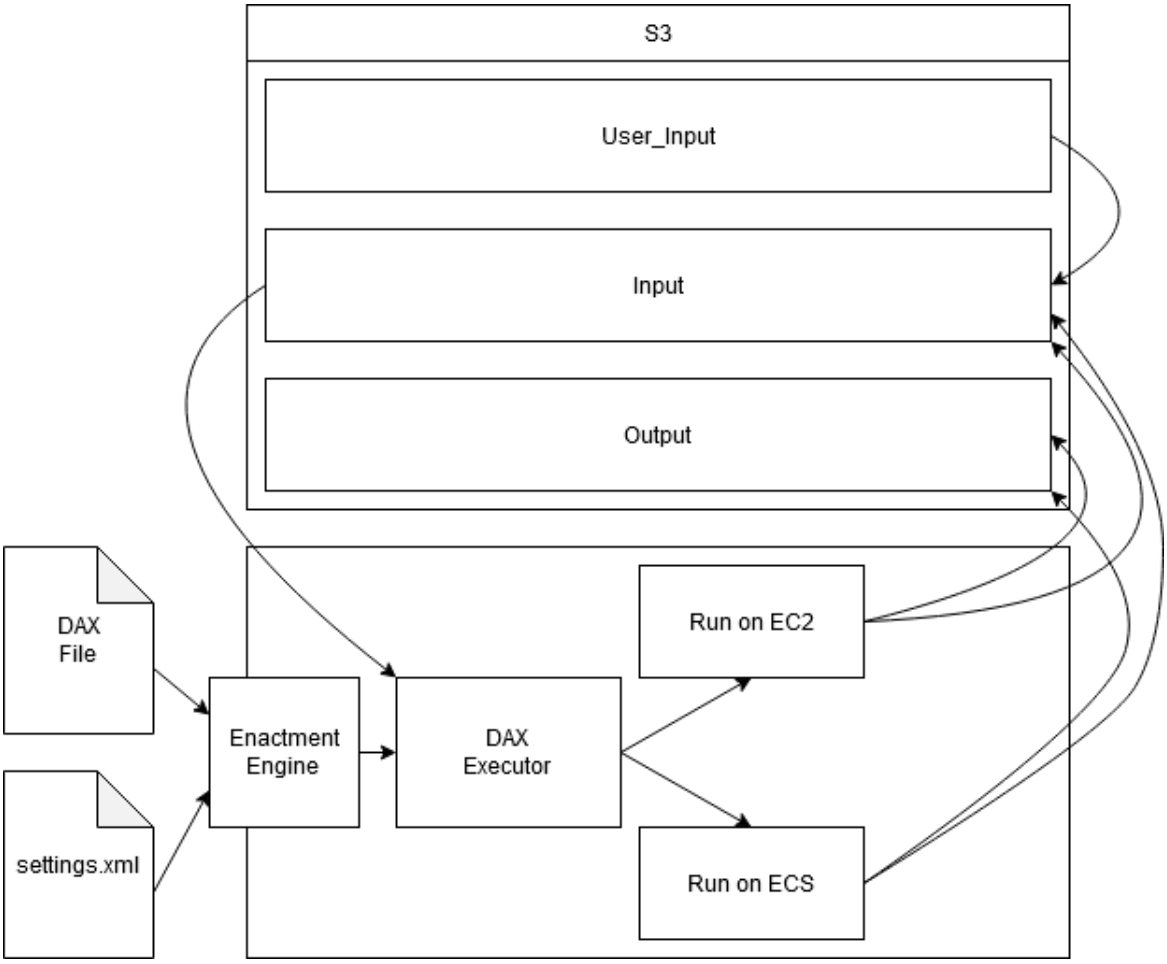

Figure 4.1: Overall system architecture of the enactment Engine

4.1.1 The input

Before starting the enactment engine, the user has to do some preparation and has to provide some input.

Preparation:
The user must have his own AWS account. He has to create a S3 bucket including a folder named `"User_Input"`. Moreover, the user must provide a Docker-Hub account to save the container images. **Note:** the enactment engine works for every workflow application. However, if a different program is used as Montage, a container image must be provided on which all programs and dependencies are installed.

Input regarding Montage: The user has to provide two things regarding Montage: input-files and the workflow description (DAX file). All input files has to be located inside the user-input S3-folder. Furthermore, if the user wants to execute his application on EC2, he has to provide an EC2 snapshot with all necessary software installed, including the AWS-cli.

Input regarding the enactment engine: To configure the enactment engine, a XML-file "settings.xml" is needed. Figure 4.1.1 shows an example of how this file should look like. The file contains the AWS credentials of the user, the default or per-job requirements of CPU and memory, the execution-type, an EC2 instance ami, a maximum amount of instances running at the same time and the current prices per hour.

```
<Settings>
  <AWS accessKeyId="ABCDEFG" secretKey="ABC123DEF456/>
  <Resources default_cpu="2048" default_memory="8192" defaultExecution="EC2">
     <Job name="mProject" cpu="2048" memory="8192" execution="EC2"/>
  </Resources>
  <EC2 maxInstances="12" ami="ami-09766d5ad3ea67eb8" type="t2medium" currentPrice="0.0536"/>
  <ECS currentPriceCPU="0.04656" currentPriceRAM="0.00511"/>
</Settings>
```

Figure 4.2: Example of settings.xml

4.1.2 The algorithm

This subsection talks about the process after starting the enactment engine.

The first step is reading and parsing the input. The engine reads the DAX file and creates a job-object for every workflow task, including informations about parent and child tasks, used commands and more. After that, the engine reads the settings file. Moreover, it creates two more S3 folders, one for the input and one for the output. While the `"User_Input"` folder remains untouched, files in the input folder are changed frequently.

The second step is only for the ECS execution. The engine starts an EC2 instance with Docker installed, which downloads the application-container-image. Then, it creates a shell script for every job and saves it inside the container. Every

script looks something like this: 1) Download input files from S3 2) execute the commands of the job 3) Upload output files to S3. At the end, save it as a new container image and upload it to Docker-Hub.

The third step is the execution of the workflow. The class "JobSorter" manages all Jobs, knowing which jobs are already done and which ones are the next. Before starting the execution, the engine checks which AWS service to use (EC2, ECS or both), because the execution is a little different. After that, the execution starts.

EC2 execution:

- Until all jobs are done, repeat:
  - wait for ready jobs (ready = there are no parent jobs or all parent jobs are done)
  - get all ready jobs
  - using First-come-First-serve order, each job requests an EC2-instance. If there is a running instance executing no job, take this instance. Else, create a new instance, if there are less running instances than the maximum amount of instances specified in settings.xml. Otherwise, wait until an instance gets free.
  - when a job is done, notify the loop: other jobs could be ready.
  - to avoid costs, every instance should be busy all the time. If there are less remaining jobs than active instances, free instances are terminated. Example: 12 instances are busy and there are 4 remaining jobs. Now, 5 jobs are done resulting in 7 busy and 5 free instances. 4 of the 5 instances are used for the remaining jobs and the last instance gets terminated.

ECS execution:

- Create a ECS cluster
- Until all jobs are done, repeat:

- wait for ready jobs (ready = there are no parent jobs or all parent jobs are done)
- get all ready jobs
- register a task definition for each job. The task definition specifies the execution type (Fargate), container image, CPU and memory requirements, security settings and more. For CPU and memory requirements check, if the job has his own requirements set inside settings.xml. Else, take the default ones.
- run each task. Fargate will download the container image and start it. The entrypoint is the shell script of the job.
- when a job is done, notify the loop: other jobs could be ready.

4.1.3 Remarks

In order to make the execution with containers more efficient, the containers used in this thesis are based on Alpine Linux [1]. Alpine Linux is build around musl libc and busybox, which makes it more resource efficient and way smaller than other GNU/Linux distributions. The containers used only include Alpine Linux, AWS-cli and Montage. In order to reduce the container size even further, I removed all Montage components that are not required. The final container has a compressed size of 113.35 MB. For comparison: previously containers based on Ubuntu were used. These had a size of approximately 285 MB.

4.2 Realization of the Enactment Engine

In this section the main parts of the engine will be discussed.

Main

The main class describes the course of the application. It is divided into 3 parts: init, execution, end. No new program logic is used, but finished methods of other classes are strung together.
The init part is responsible for the preparation. There the S3 database is set up correctly, the settings.xml file is read in and other preparations are made here.

When working with containers, this part can be used to create the shell scripts and to edit the container image.
The execution part is the main part of the engine. It executes the algorithm that was explained in the previous section.
The end part ends the program. Its main task is to terminate EC2 instances, if they have not yet been terminated.

S3

The S3-Handler class includes all methods regarding the communication with S3 buckets. It is an important class because S3 is accessed very often: in the init part of the main class, the input files are copied from the `"User_Input"` folder to the Input folder. The input and output folders are also created automatically here. In execution, both the containers and the EC2 instances use S3 commands to download input from S3 and upload the output.

Parser

The engine contains two different Parser: one for parsing the settings file, one for the DAX file. The DAX File Parser can parse any xml file that is structured according to the same pattern as the generated workflow description from Pegasus. A section of a DAX file can be seen in Figure 2.2. The settings parser follows its own pattern, as demonstrated in Figure 4.1.1.

EC2

There are several classes that are responsible for working with EC2. The AWS API is used to create and edit EC2 instances and to create security groups. However, SSH and the AWS cli are used to interact with EC2 instances.

Creation of an EC2 instance

To create an EC2 instance, some things are needed. The most important things are the user credentials. They are needed so the engine can interact with the users AWS-account. Then, a security group is required to limit the permissions of the instances. The engine automatically creates one giving the instance all necessary

permissions. The next thing needed to create an instance is the region. Amazon offers a huge amount of different regions across the world. In this thesis only `"EU_Central"` is used. Moreover, an AMI (Amazon Machine Image) is required. The AMI should be based on Ubuntu so that the engine is guaranteed to work. In addition, all programs required for execution with EC2 must already be included in the AMI. Finally, the insatance type is required.

Interacting with an EC2 instance

A separate thread is created for all communication with an EC2 instance, which claims the instance for itself. All the information that the thread needs is included in the creation of the thread. EC2 instances are used not only for the execution of Montage, but also for other tasks. For example, instances are used to create new container images. Therefore, what is to be done is already said when the instance is created. When the instance runs a job of the workflow, the thread receives all the information about it. That would be: the input files and the commands to be executed. If the thread executes a command on an instance, it connects to it via SSH.

Container

A container image which already contains all the necessary programs and dependencies is required for execution with ECS.
There are two ways to create container images: either you create a Dockerfile in which you write everything you need. Or you download a Docker image, start a container with it, "enter" this container and change it. You can then save it as a new container image. The second variant is used in this thesis, since a lot is changed in the container image and many scripts are generated dynamically. If the user uses a program other than Montage, he must provide a container image himself. This is because installing the right dependencies can be very hard.

To create the Montage container image, a container image based on Alpine with aws-cli preinstalled was used. This image was chosen because Alpine Linux does not offer a package for aws-cli and the manual installation is complex. Montage was installed on this image. In order to reduce the size of the image, all unnec-

essary files were deleted manually.

In the next step, the shell scripts are created on the container image provided. The DAX file is read out and a script is created for each job in the workflow. This contains commands for downloading the input files, for uploading the output files and all Montage commands. If a container is later created based on the container image, the container automatically starts with the script that is responsible for the respective job.

An EC2 instance with Docker preinstalled is used to create the container image as described. The final image is uploaded to Docker-Hub.

ECS Fargate

The ECS-Handler class is responsible for working with ECS. We remember the different levels that were presented in Chapter Background: Cluster, Service, task definition and container definition.
A cluster is simply the holder for all ECS tasks. This is created at the start of execution. It does not incur any costs and therefore does not have to be deleted. With a service you can run multiple instances of the same task definition. If a task aborts - be it due to an error or intentional - a new instance is started immediately. Services were used at the start of the implementation, but no longer since they do not offer the desired functionality.
A task definition is required to run containers. There you define CPU and memory requirements, network mode and so on.
The container definition is used to define the container image, the name of the container, the entrypoint and more.
Task definition and container definition are created together in this implementation. When the Executor starts a workflow job on ECS, it creates a new task definition inside the cluster. CPU and memory requirements are either the default values or the values specified specifically for this job. The name of the container is the name of the job and the entrypoint is the script created for this job.

Dax-Executor

The `DAX_Executor` class handles the complete execution of the workflow. It follows the algorithm explained in the previous section. Therefore, it is no longer discussed in detail here.
The class is divided into two parts: one for ECS and one for EC2. A start method analyses the settings file at the beginning and executes either the ECS, the EC2 or both versions accordingly.
The procedure for execution with ECS has already been explained enough in section Algorithm and ECS Fargate and will therefore not be explained further.The execution with EC2 is discussed in more detail.

As already mentioned, the user can specify a maximum number of instances running at the same time. When the execution loop explained in section Algorithm wants to run a new job, an instance is requested on which the job can be executed. A new class comes into play: the `EC2_Allocator`. A new thread is created for each request. First, this thread checks whether another thread has already completed its task and therefore an instance has become free. If so, the thread uses this free instance for its task. If no instance is free, a new EC2 instance is created, provided the specified maximum has not yet been reached. If the maximum has been reached, the thread waits until another thread releases its instance.

Statistics

Another part of the work is statistics. This has no effect on the execution and can be omitted. There are two types of statistics: time measurement and cost measurement. With everything that is done, the time is measured and stored in a local database. To measure the costs, the current AWS prices must be entered in the settings.xml file. These change every now and then and need to be updated. Based on the execution time, the CPU and memory requirements and the prices, the costs are then calculated in US dollars. Both the prices for EC2 and for ECS are always calculated per second, whereby at least one minute must be paid. So it would be optimal not to run a container or EC2 instance for less than a minute to give away money.

With EC2, the price depends only on the instance used. At ECS you pay depending on what you have specified as CPU and memory requirement.

Here are the exact formulas for the cost calculation of ECS Fargate:

$$vCPU\ charges = Number\ of\ Tasks\ *\ Number\ of\ vCPUs\ *\ Price\ per\ CPUSecond$$

$$memory\ charges = Number\ of\ Tasks\ *\ Memory\ in\ GB\ *\ Price\ per\ GB$$

$$total\ Fargate\ charges = vCPU\ charges\ +\ memory\ charges$$

5 Evaluation

This chapter delves deeper into the results of the tests discussed at the beginning of Chapter Implementation. First, it is summarized again what the goal of this thesis is. Then the testbed is presented. After that, the tests and the results are discussed. In the end, it is checked whether the previously set goal has been achieved and whether expectations have been met.

5.1 The goal

The main goal of the tests is to speed up the execution of Montage at the lowest possible price. While EC2 is cheap, ECS is fast. By dividing the individual Montage workflow jobs well between these two services, the best possible combination should be created. The optimal combination should look like that the entire Montage process is faster than just using EC2 and cheaper than just ECS. Another good combination could be that the execution is significantly faster than EC2 and ECS, but still more expensive.
So the two important factors measured in the tests are runtime and cost.

5.2 Testbed

The AWS region eu_central_1 was selected for all tests because this region is the closest and supports AWS Fargate.
The EC2 instances run on the Linux distribution Ubuntu. The container image, however, is based on Alpine Linux, which is optimized for container applications. Since ECS Fargate has to download the container image, it is also necessary to mention its size, namely 113.35MB (compressed). The Montage input files must also be downloaded, which can have a major impact on execution time. In the tests, there were 36 input files with a size of up to 12.3MB.

The AWS prices change constantly and therefore have to be adjusted. The following prices (USD per Hour) were used for these tests:

- t2-small: 0.023
- t2-medium: 0.0464
- c5-large: 0.085
- c5-xlarge: 0.17
- ECS per vCPU: 0.04048
- ECS per GB: 0.004445

The next chapter lists which tests have been carried out. Each test was run three times and the results reflect the average. The only exceptions are the tests that were carried out on c5-large and c5-xlarge. These tests were only run twice because the costs were significantly higher.

5.3 The results

To achieve this goal, after the engine was developed, many tests with different configurations were carried out. It was measured how long the program takes, how much everything costs and how long the individual jobs in the workflow take. The data collected was then summarized in several diagrams and the Pareto optimality was drawn in.
The Pareto optimality is a principle that states that it is not possible to improve one value without worsening the other. This means that the results, which follow the principle of Pareto optimality, are the best possible in one respect. The user then only needs to know which property he values most(e.g. price or performance) and then select a pareto optimal configuration accordingly.

To collect results, Montage is run three times with different configurations. For every configuration, run Montage

- on EC2-instances
- on ECS Fargate

- mixing the above methods

The different configurations include changes of:

- EC2 instance type
- maximum amount of EC2 instances running at the same time
- different CPU and memory values for ECS (per container) and EC2 (per instance)

The following tests were carried out:

- EC2 t2-small with 12 maxInstances (100% – in Montage maximum 12 jobs are running at the same time)
- EC2 t2-small with 8 maxInstances (66.6%)
- EC2 t2-small with 4 maxInstances (33.3%)
- EC2 t2-medium with 12 maxInstances (100%)
- EC2 t2-medium with 8 maxInstances (66.6%)
- EC2 t2-medium with 4 maxInstances (33.3%)
- EC2 c5-large with 12 maxInstances (100%)
- EC2 c5-xlarge with 12 maxInstances (100%)
- ECS 1024 cpu / 2048 memory
- ECS 2048 cpu / 4096 memory
- ECS 4096 cpu / 8192 memory
- MIX t2-small, 12 maxInstances, 1024 cpu / 2048 memory
- MIX t2-small, 8 maxInstances, 1024 cpu / 2048 memory
- MIX t2-small, 4 maxInstances, 1024 cpu / 2048 memory
- MIX t2-medium, 12 maxInstances, 1024 cpu / 2048 memory

- MIX t2-medium, 8 maxInstances, 1024 cpu / 2048 memory
- MIX t2-medium, 4 maxInstances, 1024 cpu / 2048 memory
- MIX c5-large, 12 maxInstances, 2048 cpu / 4096 memory
- MIX c5-xlarge, 12 maxInstances, 4096 cpu / 8192 memory

NOTE: In the tests in which both services were used, the EC2 instances always have the same resources available as the containers. In this way you can compare both services well. Here you can see the available resources per EC2 instance type:

- T2-small: 1 vCPU (=1024), 2GB memory (=2048)
- T2-medium: 2 vCPU (=2048), 4GB memory (=4096)
- C5-large: 2 vCPU (=2048), 4GB memory (=4096)
- C5-xlarge: 4 vCPU (=4096), 8GB memory (=8192)

C5 instances are optimized for data processing. T2 instances can maintain high CPU performance for as long as a workload requires.

There are three things to consider when executing:

- Due to the runtime, the various Montage jobs can be divided into two groups: mProject jobs and all other jobs. Jobs from type mProject have a long runtime, while all other jobs are done very quickly. Depending on the configuration, mProject jobs take up to 30 times longer than the others.
- AWS Fargate takes some time to deploy the serverless infrastructure. In the tests, this time averaged 30 seconds. So this is a fixed time that is needed every time a container is started.
- At AWS you always pay for at least one minute.

Based on these three points, a conclusion can now be made:
Depending on the configuration, some jobs only take around 10 seconds. If you add Fargate's 30-second preparation time, we are at 40 seconds. So less than a minute. Money is lost here. So in order to save costs, only jobs with a runtime

of more than 60 seconds should be executed on containers – in case of Montage, only mProject jobs. EC2 also calculates for at least one minute, but the engine uses the same EC2 instances again and again. So even if a job only takes 10 seconds, the runtime of an instance is still over a minute.

In the tests in which both EC2 and ECS Fargate were used, the mProject jobs were executed on ECS and the remaining jobs on EC2. **NOTE:** How the jobs are distributed across the two AWS services is still up to the user. It is therefore also easily possible to distribute the jobs differently than in these tests.

Results of EC2 only

Based on the test data resulting from the execution of Montage on EC2 instances, four statements could be made:

- The more resources are available, the faster the program will run. There is a significantly larger distance between t2-small and t2-medium than between t2-medium and c5-xlarge, although the number of resources is doubled for both. This suggests that further increasing resources will save even less time.
- C5-xlarge costs almost 4 times as much as t2-medium. The time-difference is too small to be worthwhile.
- In terms of time, C5-large is about the same as t2-medium, but significantly more expensive.
- Decreasing maxInstances reduces costs. However, the time required increases significantly.

The results can be seen in Figure 5.1. Results with a red symbol are pareto optimal.

Results of ECS only

If we execute each job from the Montage workflow in a container, the following statement can be made: The more resources are available, the faster the program will run. However, the time savings are not very large if you take into account

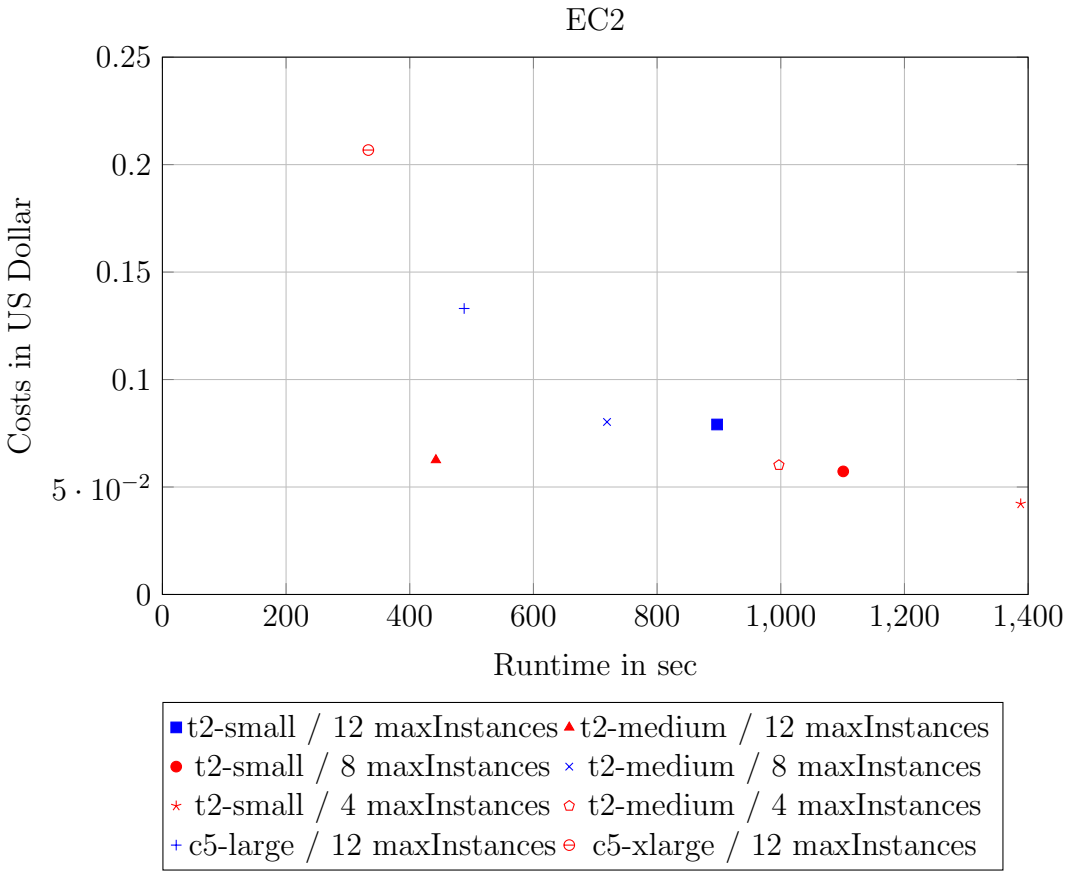

Figure 5.1: Results of EC2 only

the price differences.
Figure 5.2 shows the results of the ECS only execution.

Results of using EC2 and ECS

When EC2 and ECS are combined, good improvements in runtime can be observed. The results can be seen in Figure 5.3, where red signs mark pareto optimal configurations.
As already mentioned, only the mProject jobs were executed on containers. Precisely because these jobs with long runtimes were outsourced, executions with low-resource EC2 instances became significantly faster:
The execution of Montage with 4 t2-small instances took on average 1388 seconds. The execution with ECS with the same resources (1024 CPU, 2048 memory) took about 741 seconds. If we now mix both AWS services, the entire execution takes

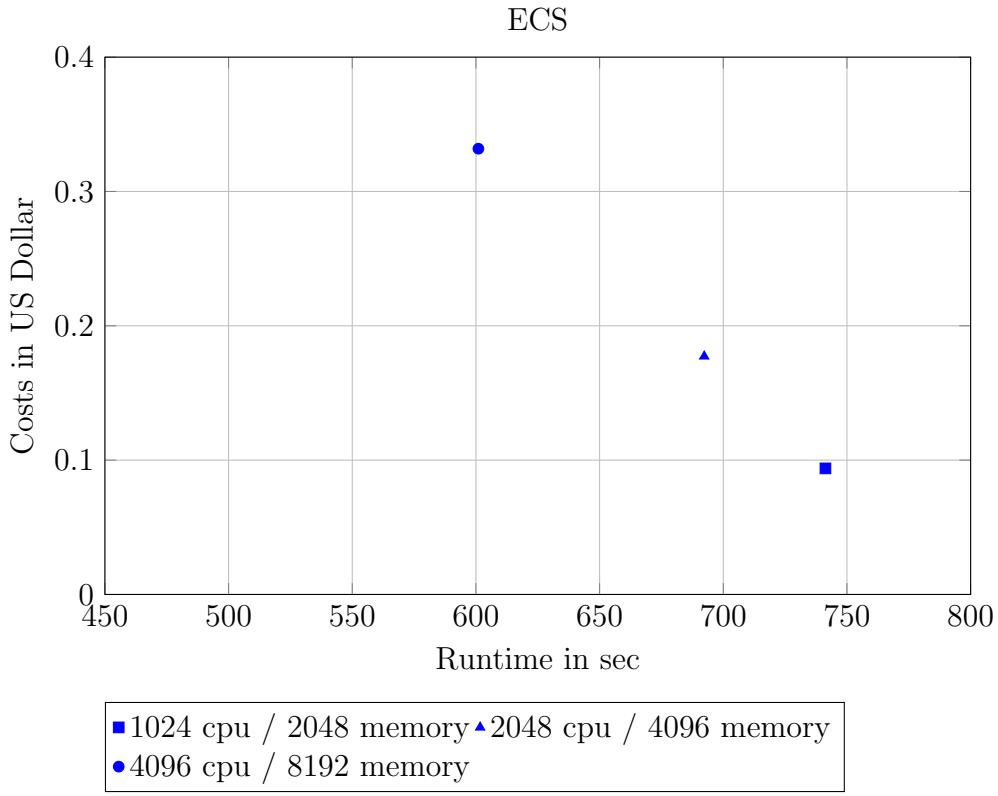

Figure 5.2: Results of ECS only

only 466 seconds. The costs are higher than for the EC2-only execution and lower than for the ECS-only execution.
All other configurations also improved in the same way.

Results regarding to mProject Jobs

The average results of each configuration for an mProject job are discussed here. Figure 5.4 shows the results, with the red symbols marking pareto optimal configurations. The following can be seen from the results:

- T2-small is very slow, but it's the cheapest version. However, the difference is not so big that it would be worth it.

- c5-large has worse results than t2-medium, although both instance types have the same resources available. This indicates that Type T2 instances are better suited for the job than Type C5 instances.

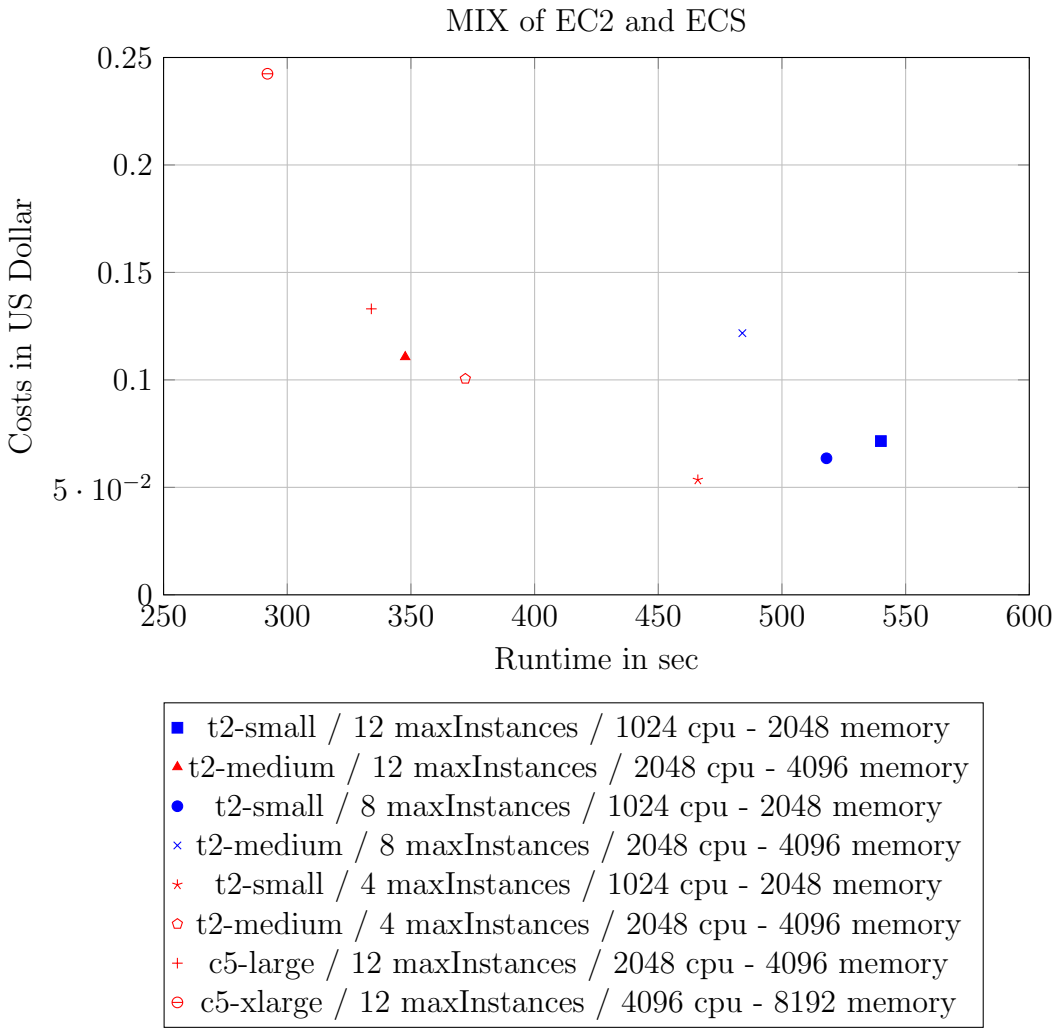

Figure 5.3: Results of mixed strategy

- The execution times of the other configurations are very similar.
- Providing more resources here does not mean that the result will be significantly better. This is probably due to the implementation of the jobs. If a job does not use the resources well (e.g. if it cannot work on several CPU cores), then more resources are of no use.
- Since it depends on the implementation of the job, an automatic distribution of the jobs to EC2 or ECS makes no sense. Rather, the user needs to know his program and how it uses resources.

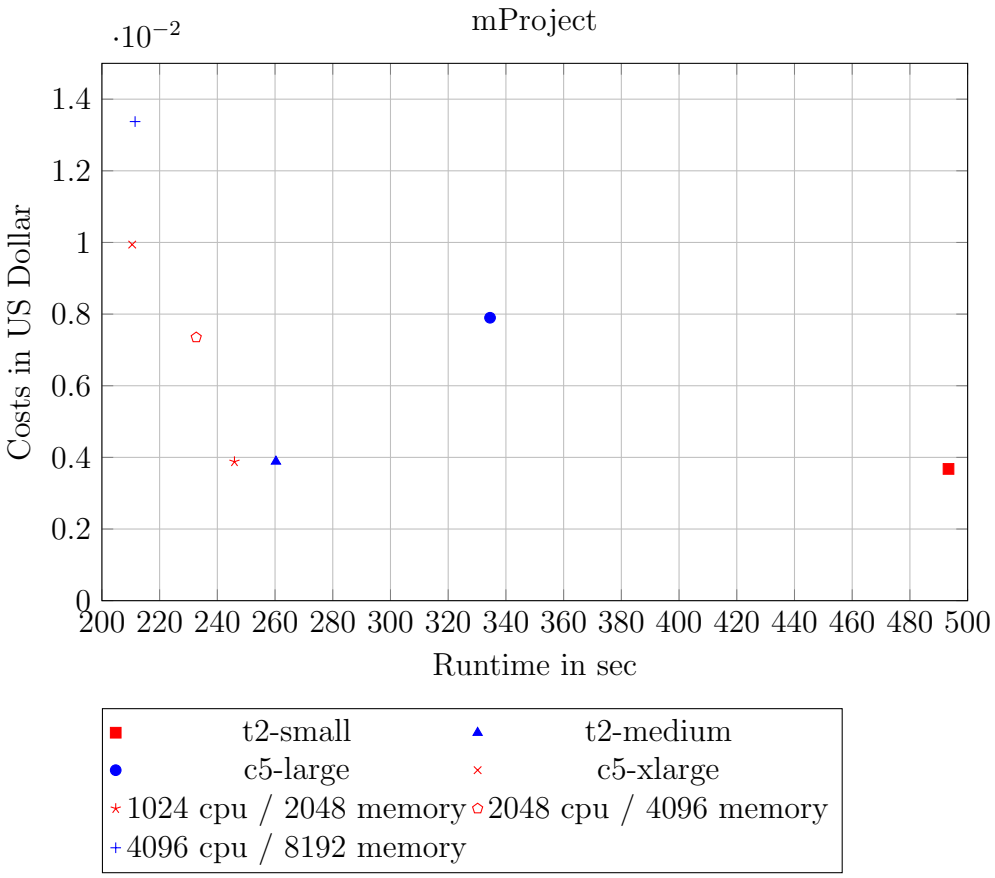

Figure 5.4: Results of mProject jobs

Discussion of all Results

If you compare all the collected results (shown in Figure 5.5 – red signs indicate pareto optimal solutions), you can determine which version of the execution is the best:

- The Mixed versions with c5-xlarge and the normal version with c5-xlarge are the fastest, but also the most expensive.
- The Mixed versions with t2-medium (12 and 4 maxInstances) are fast too and less expensive.
- The normal t2-medium (12 maxInstances) is cheaper and a bit slower.
- The mixed version with t2-small (4 maxInstances) is close to the normal t2-medium.

- The normal t2-small (4 maxInstances) is the cheapest one, but very slow (1388 seconds).
- ECS only is not a good idea, since short jobs are much longer than normal due to 30sec starting-time of Fargate.
- But combined with EC2 it is very effective and can boost the execution time.

5.4 Discussion

If you look at the results, the engine has had a visible success. Out of 19 different test configurations, 8 configurations are pareto-optimal. 5 of these 8 are mixed configurations, including the fastest version. This shows that a mixture of EC2 and ECS can offer an advantage. Although ECS only experiments are dominated by EC2 experiments, the enactment engine still dominates many EC2 only configurations when it uses mixed experiments.
The success of the engine also depends on the program to be executed. For a program whose jobs use resources poorly, other configurations may be more appropriate. Therefore, the distribution of jobs between the two AWS services and the distribution of resources is left to the user. In order to find the best possible configuration for any program, tests are necessary. The results obtained here can serve as a template for other programs.

During implementation I figured out that containers are indeed faster, but Fargate is not always. The reason for this is that Fargate takes some time (approx. 30 seconds) to provide the underlying structure. This makes short tasks on EC2 instances faster than on Fargate. This is an important point to consider as an user of the engine.

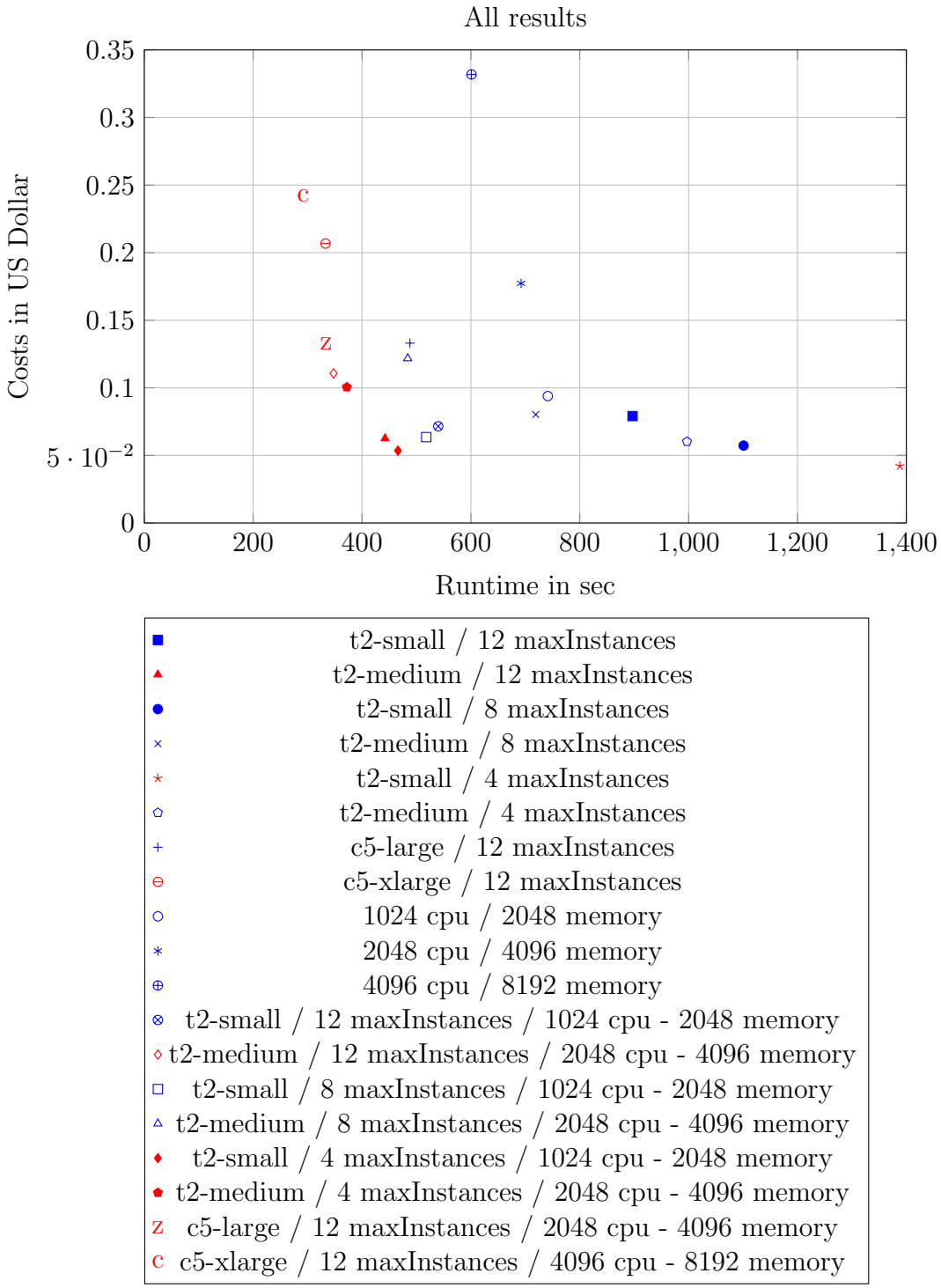

Figure 5.5: Overview of all results

6 Conclusion and future work

It has been shown that the enactment engine has advantages over a simpler implementation. Unlike many existing tools, this engine does not ensure the portability of an entire workflow to different cloud providers, but divides the workflow and takes care of the execution of each individual workflow job separately. In this way, the resource requirements of each individual job can be addressed. The user has the freedom to choose how much resources each job receives and on which service it should run.
Based on the test results of this bachelor thesis and possibly based on his own test results, the user can choose between different configurations. It is up to him to choose the cheapest, the fastest or a quick and cheap configuration.

You could also extend this thesis by developing an algorithm that learns from existing results and automatically calculates the best configurations. In this way, the user no longer has to test which is the most sensible option for his application. Furthermore, in addition to EC2 and ECS Fargate, you could install other AWS services or switch to other cloud computing providers.

7 Additional Informations

"This research made use of Montage. It is funded by the National Science Foundation under Grant Number ACI-1440620, and was previously funded by the National Aeronautics and Space Administration's Earth Science Technology Office, Computation Technologies Project, under Cooperative Agreement Number NCC5-626 between NASA and the California Institute of Technology" [15].

Bibliography

[1] Alpine linux website. Website, 2019. Online available https://alpinelinux.org/about/; accessed 28.12.2019.

[2] F. Arute, K. Arya, R. Babbush, et al. Quantum supremacy using a programmable superconducting processor. Paper, 2019. Online available https://www.nature.com/articles/s41586-019-1666-5; accessed 20.12.2019.

[3] Aws website - main page. Website, 2019. Online available https://aws.amazon.com/?nc2=h_lg; accessed 20.12.2019.

[4] Aws website - what is cloud computing? Website, 2019. Online available https://aws.amazon.com/what-is-cloud-computing/?nc1=h_ls; accessed 20.12.2019.

[5] Aws website - customer success. Website, 2019. Online available https://aws.amazon.com/solutions/case-studies/?hp=tile&tile=customerstories&customer-references-cards.sort-by=item.additionalFields.publishedDate&customer-references-cards.sort-order=desc; accessed 18.12.2019.

[6] Aws website - amazon ec2. Website, 2019. Online available https://aws.amazon.com/ec2/; accessed 23.12.2019.

[7] Aws docs - what is amazon ec2? Website, 2019. Online available https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html; accessed 23.12.2019.

[8] Aws website - amazon elastic container service. Website, 2019. Online available https://aws.amazon.com/ecs/?nc1=h_ls; accessed 23.12.2019.

[9] Aws website - amazon s3. Website, 2019. Online available https://aws.amazon.com/s3/?nc2=h_ql_prod_fs_s3; accessed 23.12.2019.

[10] Aws website - cloud computing with aws. Website, 2019. Online available https://aws.amazon.com/what-is-aws/?nc1=f_cc; accessed 22.12.2019.

[11] G. Juve, E. Deelman, B. Berriman, et al. An evaluation of the cost and performance of scientific workflows on amazon ec2. Paper, 2012. Online available https://link.springer.com/article/10.1007/s10723-012-9207-6; accessed 22.12.2019.

[12] G. Juve, E. Deelman, K. Vahi, et al. Scientific workflow applications on amazon ec2. Paper, 2009. Online available https://ieeexplore.ieee.org/abstract/document/5408002; accessed 20.12.2019.

[13] G. Juve, E. Deelman, K. Vahi, et al. Data sharing options for scientific workflows on amazon ec2. Paper, 2010. Online available https://dl.acm.org/citation.cfm?id=1884693; accessed 21.12.2019.

[14] Dirk Merkel. Docker: Lightweight linux containers for consistent development anddeployment. Paper, 2014. Online available https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf; accessed 24.12.2019.

[15] Montage caltech website. Website, 2019. Online available http://montage.ipac.caltech.edu/; accessed 18.12.2019.

[16] Pegasus website. Website, 2019. Online available https://pegasus.isi.edu/; accessed 21.12.2019.

[17] Sasko Ristov. Modeling scalable applications. Slides, 2019. presented in summer semester 2019.

[18] C. Weinhardt, A. Anandasivam, B. Blau, et al. Cloud computing - a classification, business models, and research directions. Paper, 2009. Online available https://link.springer.com/article/10.1007/s11576-009-0192-8; accessed 22.12.2019.

[19] montage-workflow-v2. Website, 2019. Source code online available on git https://github.com/pegasus-isi/montage-workflow-v2; accessed 21.12.2019.