

Exploring the use of work-hungry agents to optimise automated testing

Master's thesis by Emil Granrud Gabrielli Spring 2021

Faculty of Technology, Art and Design (TKD)

Applied Computer and Information Technology Master's Programme Cloud-based Services and Operations

Exploring the use of work-hungry agents to optimise automated testing

Emil Granrud Gabrielli

14th of May 2021

Abstract

Current automated software testing is time consuming as the tests becomes more extensive. The requirement of a dedicated software infrastructure introduces waste of resources as the server stays dormant during times of no testing. Academic research does not consider the needs of practitioners in this area. This result in practitioners implementing their own solutions for software testing that does not follow research. This thesis challenges software test optimisation practises by going into a new direction; by creating a model that would divide a test into smaller tests which is then distributed among several computers for processing.

The resulting model is evaluated to explore how this model effects automated software testing and how it compares to a dedicated testing infrastructure. A simulation framework is created to expose the model and different testing infrastructures for scenarios that represents incoming tests and behaviour of workers and infrastructure. Analysis from the results of the simulation suggests that testing infrastructure would perform well during low frequency of testing and a short duration of test time. The resulting model have a higher percentage of utilisation compared to infrastructures, but is vulnerable for increasing the wait time for tests when the demand for available workers is not met. Data suggest that an implementation of the model as a cloud service would be more cost efficient than local infrastructure.

Contents

Li	st of	Figure	S	vi
Li	st of	Tables		vii
Li	st of	Listing	jS	ix
1	Intro 1.1 1.2		on m Statement	
2	Bac 2.1 2.2	2.1.1 2.1.2 2.1.3	ated Software Testing	6 8 9 10
	2.3	2.2.2 2.2.3 2.2.4 Cloud 2.3.1	Continuous Deployment	11 12 13 13 14 17
	2.4	2.4.1 2.4.2	and CI/CD Pipeline Technologies	19
	2.5	User so and De 2.5.1 2.5.2	cenarios for cloud computing, Automated Software Testing evOps	21

		2.5.3	Optimising the data center	22
		2.5.4	Implementing automated software testing to increase soft-	
			ware quality	23
		2.5.5	Demand for distributed testing	24
3	Apr	roach		27
•	3.1	Design	1	28
	0.2	3.1.1	Model characteristics	29
		3.1.2	Statistics on the tests and hosts	29
		3.1.3	Industry expectations and use cases	30
		3.1.4	Expected outcome	30
	3.2		nentation	31
	0.2	3.2.1	The simulation	31
		3.2.2	Data collecting and analysis	32
		3.2.3	Goal of our implementation	32
	_			
4		<u> </u>	the model	33
	4.1		ew of FEAST	33
	4.2			36
		4.2.1	Test Cycle	36
		4.2.2	Creator	37
		4.2.3	Raw Test	37
		4.2.4	Broker	38
		4.2.5	Partitioning and Encapsulation	38
		4.2.6	Bite	39
		4.2.7	Market	40
		4.2.8	Worker	41
		4.2.9	Test Results and Reports	41
	4.3		communication design patterns	41
		4.3.1	Anonymous testing	42
		4.3.2	Contract broker	
		4.3.3	Direct based communication	46
	4.4	Archite	ectural design	47
		4.4.1	Creator	48
		4.4.2	The Broker	50
		4.4.3	Worker	52
	4.5	Key M	lodel Processes	53
		4.5.1	Submit test	53
		4.5.2	Partitioning (and encapsulation)	54
		4.5.3	Availability	56
		4.5.4	Running a test	60

		4.5.5 4.5.6		61 63
	4.6		Report	03 64
	4.0	∈xamµ 4.6.1	In-house testing	64
		4.6.2	8	65
		4.6.3	Open-Source Cloud Native Application	66
	4.7		ary	68
_				
5	•		5	69
	5.1		1	69
		5.1.1	Main goal: Optimising completion times	69 70
	F 0	5.1.2	Subgoal: Reducing costs of testing	70
	5.2	-	nentation approaches	71
		5.2.1	5 1	71
	- 0	5.2.2	8	73
	5.3	Simula		74
		5.3.1	Creating the input files	74
		5.3.2	Worker and test classes	79
		5.3.3	Summary	85
	5.4		cussion on Comparing Costs	85
		5.4.1	Cost of a dedicated testing infrastructure	86
		5.4.2	Calculating costs for TTaaS	87
		5.4.3	Summary	88
6	Ana	lysis		89
	6.1	Simula	ation	89
		6.1.1	Scenario: Intense testing to a moderate amount of workers	89
		6.1.2	Scenario: High demand for workers	92
		6.1.3	Scenario: Overload of workers	94
		6.1.4	Scenario: Continuous submitted tests combined with local	
			infrastructure	97
		6.1.5	Scenario: Moderate testing to infrastructure and workers . 1	100
7	Disc	cussion	1	.03
	7.1	Analys	sis results	103
		7.1.1	Improving input files	104
		7.1.2	Reproducibility	
	7.2		nallenge of emulating multiprocessing	
	7.3		oning	
		7.3.1	Broker partitioning	
		7.3.2	Test creator partitioning	

	7.4	Model design	107
	7.5	Model aspect areas	108
	7.6	Evaluating the exploration	109
	7.7	Comparing FEAST to existing literature	109
	7.8	Future work	110
		7.8.1 Develop a prototype of FEAST	110
		7.8.2 Integrating FEAST in CI/CD pipeline technologies	110
	7.9	Revisiting the problem statement	111
8	Con	clusion	113
Re	References		
Α	Sim	ulation files	119
	A.1	Input generators	119
		A.1.1 Generator for infrastructure input files	119
		A.1.2 Generator for test input files	120
		A.1.3 Generator for worker input files	130
	A.2	Main simulation file	132

List of Figures

2.1	Continuous Delivery with Continuous Integration and Continuous	
	Deployment	12
2.2	Comparison of IaaS, PaaS and SaaS	15
2.3	Comparison on publications regarding IaaS, PaaS, SaaS, CaaS and FaaS. Data from <i>app.dimensions.ai</i> .	16
2.4	Simple illustration of how virtual machines and containerisation works on hardware	19
4.1	Class diagram showing the dependencies between our model (bro-	
	ker), one or more test entities and one test creator	34
4.2	Component diagram showing the different processes within each instance	35
4.3		35 37
-	Test cycle illustration	
4.4	Bite encapsulation illustration	40
4.5	Market illustration	40
4.6	Simplified illustration of communication between stakeholders for our broker solution	43
4.7	Alternative communication approach with a contract broker	44
4.8	Illustration of our model with a direct communication structure .	46
4.9	Illustration of the layered pattern	48
4.10	The CIA triad	49
4.11	Client-server pattern	50
4.12	Illustration of the traditional broker pattern and altered	51
4.13	Activity diagram of the optional pipe-filter pattern	52
4.14	Illustration of message queues	56
5.1	Illustration of the iteration of timestamps	
5.2	GitHub commit messages over time	75

6.1	Line graph of metrics for the scenario: Intense testing to a mod-	
	erate amount of workers	90
6.2	Percentages of utilisation in the scenario: Intense testing to a	
	moderate amount of workers	91
6.3	Average wait time for tests submitted per time in the scenario:	
	Intense testing to a moderate amount of workers	92
6.4	Line graph of metrics for the scenario: High demand for workers .	93
6.5	Percentages of utilisation in the scenario: High demand for workers	93
6.6	Average wait time for tests submitted per time in the scenario:	
	High demand for workers	94
6.7	Line diagram of metrics for the scenario: Overload of workers	95
6.8	Percentages of utilisation in the scenario: Overload of workers	96
6.9	Average wait time for tests submitted per time in the scenario:	
	Overload of workers	96
6.10	Line graph of metrics for the scenario: Continuous submitted	
	tests for infrastructure for a 64-core infrastructure	97
6.11	Line graph of metrics for the scenario: Continuous submitted	
	tests for infrastructure for a 32-core infrastructure	98
6.12	Line graph of metrics for the scenario: Continuous submitted	
6.10	tests for infrastructure for a 16-core infrastructure	98
6.13	Line graph of metrics for the scenario: Continuous submitted	~~
C 1 4	tests for infrastructure for a 8-core infrastructure	99
0.14	Percentages of utilisation in the scenario: Moderate testing to	101
	infrastructure and workers	101

List of Tables

4.1	Comparisons of benefits with both stakeholders in the broker model	43
4.2	Comparisons of benefits with both stakeholders in the contract broker model	46
4.3	Comparisons of benefits with both stakeholders in the API model	47
5.1	Overview of the costs associated with a dedicated testing infras-	
	tructure	86
5.2	Overview of costs for services from cloud vendor that focus on vCPU	87
6.1	Overview of the utilisation percentages of each core in the different infrastructures	99

Listings

4.1	Pseudocode of a unit test	53
4.2	Creating an entire new image	59
4.3	Creating a new image where dependency is cached	59
5.1	Code used to extract data from the GitHub commit messages dataset	75
5.2	Code snippet of <i>generate_workers.py</i>	76
5.3	Sample from a workers scenario	77
5.4	Code snippet of <i>generate_tests.py</i>	78
5.5	Sample for a test scenario file	79
5.6	Code block for the assigned_test() class method	81
5.7	Code block for the process() class method	81
5.8	Code snippet from the dunder method for the Worker class	82
5.9	Example output from the dunder method of a worker instance	83
5.10	Code block that is called whenever a test is assigned a worker	84
5.11	Dunder method for the test class	84
5.12	Example output of the dunder method for a test instance	84
A.1	generate_infrastructure.py: Creates infrastructure input files	119
A.2	generate_tests.py: Creates test input files	120
A.3	generate_workers.py: Creates the worker input	130
A.4	main.py: Runs the simulation based on two input files; worker sce-	
	nario and test scenario	132
B.1	graph_gen.py: Generates LaTeX Line Graphs	141

Acknowledgements

I would first and foremost like to express my deepest appreciation for my supervisor Kyrre Begnum. His expertise, enthusiasm and extensive knowledge has been a huge motivation for me throughout this thesis and the Master's programme. The help I have received from him has been extraordinary and is hugely appreciated. Thank you.

I would like to extend my sincere thanks to the faculty of the Applied Computer and Information Technology Master's Programme at OsloMet for introducing me to the world of cloud computing and the DevOps methodology which inspired me to pursue a career within virtualisation and cloud services.

Lastly, I would like to show my appreciation to my friends and family who has been supportive during the completion of this thesis.

Chapter 1

Introduction

Today's modern society is becoming increasingly technologically dependent to a point of no return. This results in interacting, both directly and indirectly, with technologies when performing day-to-day tasks such as paying for groceries with a debit card. Behind simple technologies such as these there is a concept which opens up a world of opportunities. This concept allows for complex processes such as transaction management, data retrieval and operating machinery through computer interfaces. In order for these technologies to operate, they run complicated computer programs and communicate with other computers across the globe. This connection of devices around the world creates a network which allows for international advancements within finance, education, health and entertainment. These advancements has been implemented within our lifestyles to the point in which new generations growing up today can not imagine a life without it.

The software installed on each computer or device is not static. In order to stay relevant, secure and competitive, software is constantly being updated. Software downloaded and installed last week may already be outdated, and a software update is required by the user in order to get the newest version.

It is becoming an increasing demand to have updates being delivered to the end user more frequently. The DevOps methodology encourages frequent releases and has been proven to increase the overall quality of software and to have a positive effect on the culture within a company, as covered in the DevOps survey by Forsgren (2018). A company that is too slow to release crucial updates may lose a client to a competing company which always keeps their software up to date.

A crucial aspect of software development is software testing. Before it is released out to production and is made available for the users, it is important to test the software so that it performs as intended. This is especially important when new functionalities are implemented. Frequent releases would therefore also foretell frequent testing. In order to test as frequently as new changes are released, developers tend to incorporate testing within the software development cycle where the software is automatically tested before release. This is called automated testing.

When the time comes for testing software, automated testing is used to check the software for errors before it is released to the consumer. This stage would typically be the process that is the most time consuming compared to other stages in the software delivery cycle such as integration, build and deployment. Optimizing this part of the software delivery process is an attractive avenue for companies that strive to release software faster while also ensuring a high quality.

Codoid (2019) points to the lack of technical knowledge of automated testing software and the oversight of development environment issues during testing to be some of the key problems of automated testing. With any method of optimising the automated test stage, updates to software could be released faster and at greater velocity to ensure that crucial updates reaches the end users faster. This would also provide the opportunity to expand the tests themselves, making them more extensive in order to cover a wider area of usages without sacrificing delivery time. However, a technical approach that addresses the former mentioned issues (technical knowledge of automation tools and environment problems) needs to be made. Current methods of testing software also relies on reserving a server for testing purposes, also known as a dedicated testing infrastructure. This infrastructure creates a limited capacity which is not reusable; mostly reserved for testing and stays dormant at any other time.

Due to the importance of automated testing, and the demand for shorter times to release updates, there is no lack of experimentation and attempts to decrease the delivery time of software by optimising the automated test stage. A common approach is to incorporate an algorithm which would affect how test data are generated or the order in which tests are scheduled to run. These approaches mainly focus on introducing changes to existing automated testing software. There is potential to develop a model that would be incorporated within existing automated testing software. This model could then be offered as either a service available on the internet, or a service developers can run on premise. This thesis will aim to go into a new direction in order to optimise automated testing by changing the way tests are run by utilising well-liked concepts within cloud technology.

1.1 Problem Statement

The problem statement for this thesis is to explore the effects of a system that would distribute workloads associated with automated testing to multiple computers for processing in an attempt to reduce delivery times and increase efficiency of automated software testing:

Explore the effect of automated testing by designing and developing a pull-based job-distribution scheme in order to achieve reduced completion times.

Automated Testing consists of tests written in such a way that they take parts of the software and compare the output of running these bits of code to an expected result. E.g. a function within a program can be imported and the results are compared to an already defined and expected result. This process is then automated by having specific events to trigger testing such as when code changes occur.

A model is to be created that emulates a *job-distribution scheme*. The resulting model will distribute jobs among certain entities, which would then complete the job they are assigned.

The time it takes from the start of the testing until it is finished is called the *completion time*. This defines when the test stage is completed regardless of the result of the test.

This thesis wants to *explore* how the designed model affects automated software testing and if completion times are reduced. In order to observe if optimisation has been achieved, the traditional way of software testing will run on the same types of measurements and testing as the resulting model. Using completion time is a good metric in order to measure optimisation but this thesis will consider usability, feasibility, costs and utilisation to evaluate the designed model.

1.2 Summary

To summarise, this thesis have as goal to optimise the automated test stage by going in a different direction than most approaches by challenging the way traditional testing is done. A model will be created that distribute small portions of a test between entities in the automated software testing process. This model would not alter the current way automated testing is performed, but build upon it to change how tests are run. In order to measure the effectiveness of the model, measuring completion times in different scenarios will give a good indication. However, this thesis will evaluate the benefits of the model by including metrics based on complexity, usability, utilisation and costs.

Chapter 2

Background

This chapter will first cover how automated software testing became integrated in the software development cycle followed by how software testing has evolved. How automated software testing is used today will also be covered.

Some specific areas and terminologies surrounding automated software testing will be described and covered. Afterwards, an introduction to some technologies used in this thesis and in the field of automated software testing are presented in order to give an understanding of their usage, which is important to know before proceeding further into this thesis.

At the end of this chapter some use cases will be introduced where the topics discussed in this chapter will be shown in practise and in which areas they are most commonly being used.

2.1 Automated Software Testing

In 1999, a book was published in order to give a good understanding on the fundamentals and importance of Automated Testing. *Automated Software Testing: Introduction, Management, and Performance*, written by Dustin, Rashka, and Paul (1999), talks about the shrinking schedules and resources that software managers and developers were experiencing. They bring up the fact that 90% of developers were missing ship dates, and that missing deadlines were common for 67% of developers (Dustin et al., 1999, p. 3). Furthermore, Dustin, Rashka,

and Paul writes that because of the importance of meeting deadlines, 91% of software developers were forced to remove key functionalities late in the development. The authors emphasises the pressure that developers was facing to reach deadlines. Reaching deadlines was more prioritised than having all of the necessary functionalities in place at release because missing the chance to reach the market as early as possible could result in the product being cancelled.

With the shrinking schedules and pressure to reduce costs associated with software development, the developers turned to automated testing; they wanted to test their software well enough but at the same time limit the test time. The definition of automated testing that stands in the book written in 1999 still holds today:

"The management and performance of test activities, to include the development and execution of test scripts so as to verify test requirements, using an automated test tool." (Dustin et al., 1999, p. 4)

Developers found great value in adopting automated software testing. Reusable test scripts and integration testing by using an automated test tool provided the greatest value.

Testing software before its release is still important today as it were before in order to ensure integrity and quality in software. Fundamentally, software testing has followed the same principles since its infancy. The only two areas of change have been the ordering of tests and the degree of integration into a larger pipeline.

2.1.1 Different types of testing

Though the word *testing* itself is very generic, there are different approaches that can be used for testing an application or software.

• Unit testing

In its simplest form, *unit testing* is an approach to test individual components in a isolated environment to see if it behaves as intended. If we would imagine the human body with all its internal organs, we could perform a unit test on the kidneys to see if they behave as intended. The unit test would then "import" the kidneys from the human body and isolate that organ only. We would then write a test that would feed the kidneys an input, in this case blood. Then we would define a desired output. The kidneys job is to purify blood from toxins and turn this into urine, so our output would be defined as blood without toxins and urine. The unit tests for our kidneys are now complete. When the test is run, the actual output from our kidneys with our input is compared to our desired output. Should the two outputs be equal then the test is a success and we can safely assume that our kidneys work.

• Integration testing

A more descriptive word for these types of testing could as well have been called "extended" unit testing. *Integration testing* is testing two or more components, that have a relation with each other, together to see that they work together as intended. In our human body analogy, we could perform an Integration test on our intestines to see that they perform well together based on the outputs and the inputs from each other.

• End-to-end testing (E2E)

End-to-end testing is testing to see if the system as a whole, with all of its dependencies, is working correctly or as expected. These types of testing is often based on a workflow such as ordering a product from a website. Continuing with the human body analogy, if we would perform an end-to-end test on the human body we would be testing all of the intricate functionalities that make up the human body. If we reduce the complexity a little, we could be performing an end-to-end test on the digestive system. We would then be testing the workings of the organs in our digestive system and its dependencies (like the brain for acknowledging when to eat and not).

The mentioned approaches for testing has different coverage and have their different advantages. Unit testing has the advantage of being fast, going into the specifics and test each component independently but does not consider how that component interacts with the other components in a system. Integration testing extends unit testing by testing two or more components and covers more of the relationship between components. End-to-end testing takes longer time but will give a good coverage on exactly how the user uses the system as a whole. An approach on testing that is favoured by Google is having 70% of testing consisting of unit tests, 20% integration tests and 10% end-to-end tests (Wacker, 2015). In this blog post, Wacker talks about how E2E Testing is good in theory but that it fails to impress in practise based on his experience working at Google. From a personal perspective, my impression of E2E testing is that it have to be thoroughly planned and run days before release and preferably late in production. Considering the fast-paced development within software I can understand why E2E testing is not something prioritised and that developers tend to favour unit testing.

2.1.2 Evolution of Automated Software Testing

Investigating the trends in software testing within the areas of academy and industry can help to see if there is a difference in focus between the areas or if they are closely related. If the problems covered by research do not meet the same concerns as those by the industry practitioners, then this would suggest that there is a distinction of the interests within these two areas.

An article published by Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Petersen, and Mäntylä (2012) acknowledges that software testing is a mature area within academic research but that there is a gap which does not recognise the benefits and problems that are actually encountered by practitioners. The authors refer to an article from 2006 by Glass, Collard, Bertolino, Bach, and Kaner (2006) which explores this gap in a little more detail. They claim that software testing is widely used in the industry but that it is not generalised and that most testers are self-learned. Furthermore, they argue that researchers would be familiar with "[...] white-box testing (a method that tests the code structure and the inner workings of an application) criteria based on control flow or data flow and elegantly ordered into a subsumption hierarchy" (Glass et al., 2006, p. 56), while practitioners would know about branch coverage (that every branch such as statements and loop to be executed at least once) at best. According to them, the issue is that practitioners do not have the time to implement the practices within software testing defined by academic research. They propose that the solution to incorporate proper software testing is to have it seamlessly incorporated into the development process. This is further supported in a conference paper by Bertolino (2007) which emphasises the benefits of incorporating a powerful integrated test environment which runs on automation.

2.1.3 Challenges within Automated Software Testing

When learning about the many benefits of automated software testing, one may be intrigued into knowing why this practise is not standardised and used everywhere. Testing of software has generally become increasingly complex, which impose challenges that has spawned research on this area. According to Jena, Das, and Mohapatra (2020, p. 111), an increase in complex projects and pressure to reduce test time stands for the imposed challenges.

If we address the challenges linked with software testing as a general, the main issue is a rise in complexity within the project. Every function, process and compatibility has to be tested in order to be fairly sure that the software will work as intended in production. This means that inclusive tests have to be written that covers the many aspects of the software; everything from login-forms to transaction processes. The wide area of different programming languages and technologies in use within software require a wide area of competence. One that maintains software tests might have to acquire knowledge about writing tests for many different programming languages. Furthermore, scripts that tests the software for security vulnerabilities are hugely beneficial but requires longer tests.

Automating is a key factor in the DevOps methodology, which will be presented in a later section, and testing is important in software delivery. The DevOps methodology is often adopted by companies that want to decrease delivery time of their software and included within this methodology is automated software testing. Though some companies might find the transition from a previous software delivery approach to adapting the DevOps methodology easy, other companies might find it a troublesome process. Morales, Yasar, and Volkman (2018) cover the challenges of implementing the DevOps methodology in Highly Regulated Environments where they address how to implement some of the principles within DevOps. The authors analyse obstacles in the *Software Development Lifecycle* and acknowledges that workflow that introduces manual processes that includes multiple teams will delay the software delivery process. They further conclude that Highly Regulated Environments impose restrictions that disallow for implementing the traditional practises within the DevOps methodology.

Although 88% of the participants in the practitioner survey done by Dudekula Mohammad Rafi et al. (2012) is positive to automated software testing, one of the respondents stated that test automation needs just as much maintenance as the software itself. 45% agrees with a statement done by the survey that states that the already existing automated software testing software is not compatible or does not fit well with the requirement of their existing tools. The survey concluded that most practitioners does not rely on automated software testing alone for testing and expect automated software testing tools to be used as a complement to manual testing.

2.2 DevOps

Also known as the DevOps movement, the combination of development and operations into a methodology emerged from a few organisations that wanted to build rapidly evolving distributed systems at scale which were also built securely and resiliently (Forsgren, 2018, p. 4). The methodology focus mainly on the process in which software is released and not specific tools. The methodology also has a strong focus on culture and how developers should be treated; as unique human beings instead of interchangeable components.

In this section the main focus is on the concepts of Continuous Integration, Continuous Deployment, Continuous Delivery and CI/CD pipelines. Of course, the DevOps methodology expands much further but it is within these topics that automated software testing is prominent.

2.2.1 Continuous Integration

In DevOps, changes should be applied quickly and frequently. Whether it is a small bug-fix or optimising the code, changes should be merged with the main source code right away. *Continuous Integration* (CI) works by firstly having a shared code repository in a version control system such as Git which may be available through GitHub or GitLab to the public or only with developers. Developers that clone the code repository may then work on it and push changes to the code repository.

When a change is pushed to the code repository, a build process is triggered that builds the app and tests different aspects of the code as a whole. These tests can include unit tests and tests based on security. This action that is triggered is known as continuous integration.

Traditionally in software development, developers depended on the feedback from customers or employees to make an application better. Developers would receive feedback once the application was released to production and was being used

by the users. If problems should arise such as missing compatibility on certain devices, developers would receive feedback and develop compatibility in the application. This is also known as the feedback loop where a customer will submit feedback that is evaluated by the company and that company will decide if to take action. Within traditional software development, the feedback loop occurs after the software has been released to production. Continuous integration shortens this feedback loop, which allows for more accurate validations and avoid repetitive tasks and human error (Anastasov, 2021). It is important to emphasise that because developers are encouraged to push changes to the main source code frequently, this process of integration has to be done quickly in order to know if changes cause conflicts, errors or break the entire application. Developers that push small changes rapidly will depend on the integration to provide feedback when a certain change causes issues in order to fix it quickly.

If the new integration successfully passes all tests, the application that is sent further down the line to be prepared for deployment.

2.2.2 Continuous Deployment

What exactly is "deployment"? The correct definition depends on what kind of application or software is being developed. If you develop an application for a iOS device then deployment would mean to push new releases to the App Store. A developer working on an internal web-based application for a company would push new releases to the web server that runs that service. We can generalise this term by saying that deployment is pushing code to production and making it available for end users.

Having updates pushed to a web-based application is, generally speaking, a straight forward technical process. New updates are most of the times not noticeable for the end user unless they cause new errors or a crash. But how often do you update your banking application on your phone? Update Windows? Or even that new exciting adventure-game? The answer is most likely "not that often", or "just when it auto-updates". Even though continuous integration encourages that developers push changes to the code frequently, one deployment usually consists of multiple software integrations and new changes (Forsgren, 2018, p. 16). An exception for this is if the company uses *Continuous Delivery*.

2.2.3 Continuous Delivery

The process of developers committing code to a code repository which triggers integration, which in turn may trigger deployment can all be wrapped around the term *Continuous Delivery* (CD). If *Continuous Deployment* is the act of collecting multiple new integrations to the software before releasing the updates to the end user, continuous delivery is the act of instantly releasing new updates to the end user when it passes all tests. An illustration of this can be seen in Figure 2.1.

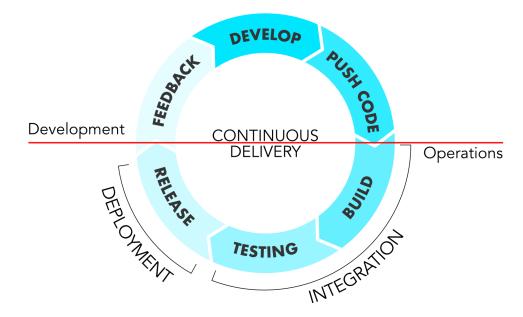


Figure 2.1: Continuous Delivery with Continuous Integration and Continuous Deployment

The DevOps methodology encourages that changes to software is to released into production frequently. According to Forsgren (2018), continuous delivery allows changes to be released to the end users safely, quickly and sustainably. Some of the principles of continuous delivery is to have quality built in. This is achieved by building a solid and open culture that consists of tools and people in which issues can be detected and resolved quickly (Forsgren, 2018, p. 42). Forsgren explains that it is also important to work in small batches, which may contradict the standards that some companies have in regards to software release. She elaborates that organisations tend to plan for bigger releases, but by dividing big work into smaller chunks we get valuable feedback on specific work which in turn can steer development to a better path.

Testing is crucial in continuous delivery, and as mentioned before is commonly implemented at the continuous integration stage. By rapidly releasing new changes to production it is important to make sure that it does not break any of the existing functionalities and that the new changes work as intended. In fact, due to the importance of testing testers should be encouraged to write exploratory tests against the latest builds (Forsgren, 2018, p. 44).

2.2.4 CI/CD Pipelines

A continuous integration / continuous deployment pipeline is the combination of those practises into an automated workflow, this is common to find among companies that practise the DevOps methodology in their software development. The CI/CD pipeline is commonly triggered either automatically when changes are pushed to a shared code repository or manually triggered by a developer at a scheduled time or build. A standard pipeline usually starts by fetching the latest version of the source code, build the application and perform certain tests to ensure that quality holds up. If the build passes all tests it will be sent further down the pipeline and be ready for release.

Several software and solutions exists in order to maintain an overview of this pipeline. These kinds of software may offer different functionalities and insight by keeping reports on the build or have the possibility to alert the developers when a new build is complete through their collaborative platform. Though they might have subtle differences from each other, these solutions tends to be quite similar in terms of where the tests are run. One characteristic they share is the assumption to use a exclusive infrastructure that is dedicated to testing, which is costly.

2.3 Cloud Computing

Cloud computing consists of technology that allows for computational resources such as hardware, software and infrastructure to be served and accessed through the internet. Because of the advantages with regard to cost and security to migrate to the cloud, there is a decreasing need for physical data centers and an increasing demand to use infrastructure from a cloud vendor. However, corporations can also use their existing physical infrastructure to create a private cloud which resides within the data center and migrate current workloads to it. Corporations tend to use infrastructure based on a hybrid (some private and some public vendors) or multicloud environment (multiple cloud vendors either public or private) due to costs, laws and regulations. These computational resources can be used and scaled on an on-demand basis, which would decrease costs compared to a unscalable infrastructure.

2.3.1 Cloud services

Cloud vendors tend to be versatile in what they offer in order to be best suited for the costumers needs instead of having the customer pay excessively for needless services. Some of the services can be categorised as *Platform-as-a-Service* (PaaS), *Infrastructure-as-a-Service* (IaaS), *Function-as-a-Service* (FaaS), *Containers-as-a-Service* (CaaS) and *Software-as-a-Service* (SaaS). All of these offers different kinds of services in order to best suite the customers needs, and some are more specific than others. There seems to be a trend amongst vendors to offer more simplified services that aims towards specific areas instead of being a broad service. E.g., Containers-as-a-Service offers a platform that is specifically dedicated to run containers while Infrastructure-as-a-Service offers a server where the customer may use one specific operating system, store a lot of different data and run containers. It is not unimaginable that as the demand for cloud services increase, more and more different types of services will be made available by current and emerging cloud vendors.

With *Infrastructure-as-a-Service*, users are given access to their reserved server space at that cloud vendor. The customer can manage everything from the operating system and higher up the stack. *Platform-as-a-Service* limits things more. Vendors will give the customer a finished platform that are available for developing and running applications without having to worry about creating the underlying infrastructure. *Software-as-a-Service* is best explained as software that resides in the cloud and is accessible through the internet by either a browser or a graphical user interface. Imagine that your local, downloaded and installed version of all the Microsoft Office programs is accessible through a simple URL and everything that is written and created with those programs are stored in the cloud. An other example is Discord, a collaborative software that is perhaps mostly associated with gamers for voice chat and developer teams. One user can simply create their own private Discord server and manage it however they like through the cloud. A comparison on how these services differentiates in terms of what parts of the stack is managed by the provider can be seen in Figure 2.2.

STACK ON PREMISE

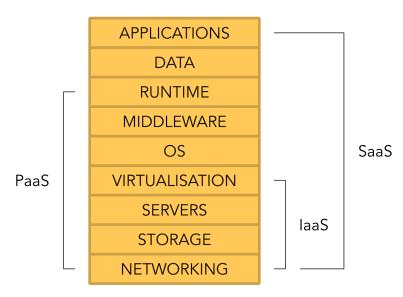


Figure 2.2: Comparison of IaaS, PaaS and SaaS

Function-as-a-Service is much more limited and simpler than SaaS in a way that it is a simple function that is triggered by certain events, such as webhooks. Due to its simplicity, Function-as-a-Service is often called a serverless architecture (Sroczkowski, 2021). Vendors that offer *Containers-as-a-Service* allows the customer to deploy containerised applications. Providers of CaaS will have a framework on premise that allows for easy orchestration of these containers by using, for example, Kubernetes for orchestration.

All of the different cloud services described until now are just some of the most popular services offered today by vendors. There are many more services available that try to best suite a group of customer's specific needs. One should assume that there will be more emerging services to come and the ones that are currently popular will perhaps be overthrown by an other service.

By looking at the number of publications within each of the different services, we can get a clue as to how the different services compare in terms of popularity and usage. As seen in Figure 2.3, Software-as-a-Service and Infrastructure-as-a-Service is the most popular services in terms of research papers with Platform-as-a-Service behind. All of the mentioned services are decreasing in popularity as we approach the present. Function-as-a-service is gaining traction and we can expect an increase in research done with this service. If we take a look at research papers done on Containers-as-a-Service we can see that little research

has been done in that specific service as of yet. As containers begin to gain more traction and containerisation may exceed the use of virtual machines, we may see an increase in academic research on the subject.

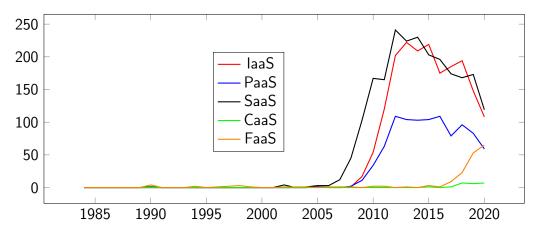


Figure 2.3: Comparison on publications regarding IaaS, PaaS, SaaS, CaaS and FaaS. Data from *app.dimensions.ai*.

The main takeaway message we can conduct from the presented results of research papers regarding clouds and services is that there are more of a purposeoriented focus on how a product should be tailored in stead of how to build it. There seems to be a higher focus on tailoring products based on IaaS or SaaS as compared to others. This trend might seem to be shifting as FaaS gains popularity.

Testing-as-a-Service

There is an additional service that has not been covered until now which is very relevant to this project: *Testing-as-a-Service*. Also known as TaaS, Testing-as-a-Service allows for companies to leverage the testing phases to a third-party that would do the necessary tests on behalf of the company.

This service stands out from the other "as-a-service" models by not being a directly, automated and distributed service easily available. TaaS is not an abstraction or a simplified service such as Software-as-a-service or Containers-asa-service. Instead, Testing-as-a-service seems to be the method of outsourcing the testing phase and its associated processes to a third-company that will be responsible for testing the software on behalf of the company. A company offering TaaS could be hired to perform a penetration test on a network or test a company's disaster recovery plan.

2.3.2 Virtualisation

Today, an ordinary home computer is capable of running multiple processes at once. In fact, by using the command ps aux on a MacOS or Linux based system terminal (or tasklist in cmd on Windows) will reveal the long list of active processes. Imagine if your entire computer got reserved, or "locked", to one process until it ended. E.g., let us say we want to use our browser and read the news. Then we want to open Microsoft Word in order to write a summary of what we found. We couldn't. Our computer is locked to the one process that is running our internet browser. We would have to close our browser process and open our Microsoft Word process, write something and then close it before repeating the cycle. As absurd as this might sound, this was the case for servers in data centers before virtualisation. One server that was capable of running multiple processes (such as two different web servers), was only restricted to only run one process. To clarify, it was possible to run several processes, but since they shared the same resources it was a possibility that one process might take more than intended or read data it shouldn't. To avoid these issues, one server was running one service.

The solution to this was virtualisation. Several processes could now be run on the same system but separated from each other in order to avoid any conflicts. This restricted one process, such as a web server, from interfering with resources that was used by an other process. Furthermore, with virtualisation one server could now run multiple operating systems.

Virtualisation works by using software to create an *abstraction layer* on top of computer hardware which allows the different hardware components to be divided into multiple virtual machines (IBM, 2019).

2.3.3 Virtual Machines and Containers

With virtual machines, each newly created instance can run different operating systems and behave independently of other instances even though they share the underlying hardware. Virtual machines allowed for new services to quickly be created or broken down and to most efficiently use the underlying hardware by utilising it as much as possible. Between the underlying hardware and each virtual machine lies the *hypervisor* which is the software that manages and coordinates the virtual machines and ensures that there is no interference between each instance. This allowed for services that were running on different independent

virtual machines to work together but at the same time enclosed to its own machine. Imagine that you can have virtual machines that acts as a web server (HTTP), file sharing server (FTP) and a SSH server all on the same underlying hardware.

Virtual machines are great for separating services and to test new software on different machines and operating systems. However, a downside is that virtual machines take time to start. With all the underlying processes it takes to create an instance of virtual machines, there was a need for a quicker way to virtualise applications for testing and to quickly take it down if it failed. A solution was to not virtualise on top of hardware but to virtualise on top of the underlying hosts operating system. Software that would allow for this type of virtualisation could be easily downloaded like any other software and suddenly you could virtualise applications from the terminal.

The technology that allowed for virtualisation on top of the hosts operating system was called *containerisation*. A new virtualisation process that used containerisation was simply called a container. A container could be run on every operating system, any server or in the cloud as long as it had the engine software to run it. These lightweight containers could be as easily created as they could be broken down in a fraction of the time it took to start virtual machines. Every container is built from a set of blueprints with instructions on how that container should be created. Containers are easily manageable and can be orchestrated further by using container orchestration software like Kubernetes.

From an architectural perspective, the difference between virtual machines and containers is that virtual machines depend on the hypervisor on top of the hardware to run the virtual machines while containers run on top of the hosts operating system. For an illustration on virtual machines and containerisation on hardware, see Figure 2.4. Containers are isolated from each other but can be part of a private internal network on the same machine while virtual machines are isolated from each other and acts as independent nodes on the network.

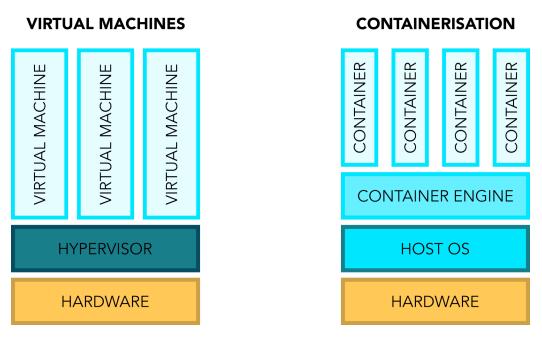


Figure 2.4: Simple illustration of how virtual machines and containerisation works on hardware

2.4 Cloud and CI/CD Pipeline Technologies

There are many variations of software within the areas of cloud, virtualisation, containerisation and CI/CD. In this brief section, some of the tools within these areas and technologies that are used later in this thesis is presented a long with their uses.

2.4.1 Docker

As told by one of the founders, Solomon Hykes, Docker came to be when he was 24 years old, just quit his job and returned to Paris to live with his mother and to start a company with his friends Kamel Founadi and Sebastien Pahl (Hykes, 2018). They wanted to take the already existing technology known as containers and build on this to create a tool that they themselves called "tools of mass innovation". They called their new company dotCloud and developed Docker. When Docker gained traction, and when dotCloud collaborated with Red Hat to their OpenShift PaaS (RedHat, 2013), dotCloud was renamed to Docker Inc.

Docker is today the most popular service for containerisation. It has its public repository of docker images, which is used to build containers, on Docker Hub and has the possibility to offer container orchestration with Docker Swarm.

2.4.2 Jenkins

Around the year 2010 the market got saturated with a lot of different tools for continuous integration. In the more recent years, many existing organisations that offer cloud services and version control have expanded their repertoire to include software for continuous integration. Software for continuous integration can be sorted into two categories; hosted and self-hosted. By these definitions, a *hosted service* would mean that the server resides with the company that offers the continuous integration software. A *self-hosted service* refers to the software being locally downloaded and being run on its own server that resides within a company or person that wants to use that software for continuous integration. Jenkins would be categorised as the latter.

Jenkins is a software that is used to automate processes related to software development. With Jenkins one can create a CI/CD pipeline for building, testing and deploying a application. As presented in section 2.2: *DevOps*, a CI/CD pipeline can be used within continuous delivery to release software efficiently and reliably. A software running Jenkins would be able to use the principles within continuous integration, continuous deployment and continuous delivery. Furthermore, Jenkins is very versatile when it comes to testing and offers support for many languages.

Despite the huge competition of different software that offers continuous integration, Jenkins is still widely used due to its wide support, ease of use and scalability.

2.5 User scenarios for cloud computing, Automated Software Testing and DevOps

This section will present three different scenarios with three different problem statements that emphasises how those problem statements can be solved using the previously mentioned technologies and methodologies.

2.5.1 Using the cloud for online gaming

Balder Entertainment has had quite a success as a small video game company. They mainly create single-player games for PC, which all have gotten big fan bases. By creating single-player games they have avoided using any networking in their games and decided to have any verification client-sided. Lately, they have experienced that players misuse the verification process and have managed to receive higher level items than intended. Furthermore, they experience a desire from the fan-base to incorporate a multiplayer mechanic in their current games and in later sequels. The board of directors and the developers at Balder Entertainment have their focus on meeting the desired functions from their fan base.

Being a small video-game company, Balder Entertainment lacks the budget for purchasing and maintaining servers in-house and look for other options. They can instead save money by taking advantage of cloud computing to handle their future needs. By using Function-as-a-Service (FaaS) they can do validation on the server-side, which would prevent players from cheating their way to get higher level items.

In order to meet the fan base's wish for a multiplayer game, Balder Entertainment decides to start development for their new game. "Hunting Jotunheim" is a multiplayer game where you and your friends team up as Norse Gods sent to Jotunheim to fight frost giants and collect clues to find and confront Ymir; the father of all the giants. While in Jotunheim, the players playing the Norse Gods will fight against other players playing as "frost giant generals" trying to protect Ymir. One match will last 30 minutes before a winner is decided; The Norse Gods or Frost Giants.

Knowing their options, Balder Entertainment could use Platform-as-a-Service (PaaS) where they supply the data on the players and the application. Players would then connect to a server running PaaS and play out the match before disconnecting. Alternatively, to further reduce costs and server maintenance, Balder Entertainment can use a vendor offering Function-as-a-Service (FaaS) and Containers-as-a-Service (CaaS). The FaaS would be used to find players searching for a match and when players are matched together, a container is created containing everything needed to run a match. The game would then connect them to that particular container created for them, which would be removed when the match is complete.

This approach is scalable and very responsive according to amount of players

online. Instead of searching for available servers and match players together, they will first match the players and then create a server they will play on. Since containers are lightweight and can easily be created, the waiting time would be minimal. Balder Entertainment would then save a lot of money by having such a scalable infrastructure.

2.5.2 Pushing new changes to production

Bob is 31 years old and working at a start-up called HealthTec. He is currently managing and developing their web services that are used by several hospitals in Norway. Due to high demand and to save time, management decided to push services to production early and have IT to integrate changes every Friday.

One of those working in IT that incorporates these changes to the production server is Bob. IT have the source code in a version control repository and Bob has been told to clone the repository locally to see that it is running as expected on his computer. When Bob has verified that everything is good and the new changes are working, he clones the repository to the server in production. Bob acknowledges that this task is tedious and repetitive and he believes that there are tools out there created to more efficiently integrate new code changes and push these changes to production.

Management has been experiencing pressure to keep their web services more updated and that requested changes and functionalities from the hospitals to be implemented faster. Being desperate, and optimistic, they decide to turn to the DevOps methodology to find practises that helps them deliver software quicker. Bob has been relived of the tedious job of manual integration and is now responsible for maintaining the CI/CD pipeline that allows changes from IT to be pushed to production quicker and more frequently.

2.5.3 Optimising the data center

Wyvern Solutions is a well established tech company that creates different services and solutions for their clients. Services within websites, dashboards, backup, transactions and CRM is created on behalf of the client and is hosted in their own data center. This data center consists of about 5,000 servers running virtual machines that the different services occupy. Whenever a developer works on a project for a client, they will create an instance of a virtual machine at the data center and wait for it to be initiated. Management has received complaints that the waiting time is constraining for the developers that often need to delete virtual machines and create new ones during development. The developers have expressed their interest in taking advantage of containerisation in order to optimise the development process.

The system administrators that are responsible for the servers at the data center report that many of the services that are in use by some clients are using a fraction of the hardware and storage compared to the virtual machine itself. The administrators sees this as a huge waste of resources. They know that by implementing a solution based on containerisation they would be able to free up a lot of space, which would reduce costs associated with running the data center.

Management is looking into the options of turning parts of the data center into a private cloud that runs a Containers-as-a-Service (CaaS) platform which would allow the developers to request containers faster and more efficiently than virtual machines. Administrators of the data center would see that a lot of resources would be cleared up and optimised. Management would want to evaluate if the current services running on virtual machines can be made into cloud native applications while they begin transforming their data center to a CaaS.

2.5.4 Implementing automated software testing to increase software quality

The 31 year old developer named Bob is very happy about his new position managing the CI/CD pipeline for HealthTec. Developers at the company notices a positive difference at the workplace and feel they are more acknowledged by the company's management. But not everything in the garden is as rosy.

Management has noticed a lot of reports of errors and crashes from the customers ever since they started with continuous integration and continuous delivery. Although they can fix the issues quite rapidly due to their new improved workflow, developers at HealthTec want to discover and fix bugs in the software before reaching the end users. They know, based on the feedback from customers, that many of the reported errors are easily fixable and that they could have avoided releasing these bugs to the customer if they only were aware of them sooner.

Developers and management at HealthTec embraces the need for frequent testing and wants to incorporate this in the software development cycle. Our trusty developer, Bob, is put on the task to start integrating automated testing to the CI/CD pipeline. With the benefit of HealthTec already being a well-established company, Bob is able to reserve their very own server to be the testing infrastructure at their data center. Developers are asked to write unit tests for the software they write, which will be run in the testing stage of the CI/CD pipeline. This stage will be automatically triggered as part of the development process whenever a developer commits a new build to the code repository. The developer that initiated the build will get feedback from the pipeline letting them know if it passed or failed the testing stage. If the test failed, then the build did not go to deployment.

2.5.5 Demand for distributed testing

RedFox Monitoring is a up-and-coming software company that specialises solely on creating software for monitoring systems and applications. They see requirements in the industry not fulfilled by other monitoring software and are positive that clients will value their unique functionalities. With a limited team of developers that work collaboratively on creating a complete monitoring suite, they hope that the software solutions they develop will become an attractive alternative.

There are many different functionalities that need to be included in the suite in order to fulfill the many requirements of the industry. From logs and graphs to alerting and integration in collaborative software, and all of these has to be tested. RedFox is yet to become a well-established company and has a limited budget. They do not have the funds or expertise to purchase and setup a good testing infrastructure and instead depend on the developers computers for testing. The developers are all experienced at writing unit tests. Therefore, the team decides that each developer should write unit tests for the functions that they are working on in a separate file. This file should import and test each function before the source code is committed to the code repository.

Many issues arise with the approach RedFox uses. They depend on each developer testing their own part of the software before the software is build as part of a CI/CD pipeline. Having the tests automatically run during the testing phase in their pipeline after code integration is a far better option because the code is tested each time a change to the code is submitted. Having the developers write unit tests for their functions is a good initiative. However, having the developers run the test before each commit takes up valuable work time. Each developer will have to change the source code, the test file and run the test. Therefore,

he or she is not able to continue working before the results from the unit test is in. If the unit tests were a success they can commit their work. If not they will have to go back to the code and find the bugs.

RedFox could take advantage of a service that would orchestrate testing by using the developers computers. This service could run in a container on their server or an employees computer, and have access to the other developers computer through software to run small bites of the tests. If each test was a success it would send a "OK"-signal to their pipeline to continue to deployment. This solution would support the idea RedFox has by utilising the employees computers and at the same time achieving the benefits from using a CI/CD pipeline.

Chapter 3

Approach

This chapter will present an overview of the approach for designing and implementing a model derived from the problem statement as described in the *Introduction* chapter:

Explore the effect of automated testing by designing and developing a pull-based job-distribution scheme in order to achieve reduced completion times.

To elaborate, this thesis aims to design and to implement a model that would leverage one workload onto several different entities and explore the effects it has on automated software testing in an attempt to achieve reduced completion times regardless of the actual test results.

When changes to software is desired to be rapidly released, as encouraged by the DevOps methodology, testing should become more efficient in order to reduce the overall time it takes for changes to be in production. When the test stage becomes more and more optimised, developers may include more and more complex testing based on other criteria that may otherwise not be included in every build.

In the problem statement as presented above, key elements are marked with bold text which can now be elaborated further. In order to create an accurate model based of the problem statement, two important aspects of the model have to be covered; how the tests are distributed and how the tests are run on each entity. These two aspects are the fundamentals that make up the *pull-based job*distribution scheme. A keyword is automated testing. Testing should be done automatically, and could be a part of a CI/CD pipeline, as described in section 2.2.4: CI/CD Pipelines. Therefore, we want to build upon current existing practises of automated software testing within the CI/CD pipeline. But, as described in section 2.4: Cloud and CI/CD Pipeline Technologies, there are many different solutions for implementing a CI/CD pipeline. Therefore, we cannot base our model of one specific software solution. When all of the testing is complete at all the different entities, we want to collect information on the different tests to see if it passes. However, this thesis aims to *reduce completion times*, but still want to collect data on the tests to see if the tests passed or failed. Lastly, one can not say for sure if this solution is more efficient than the current implemented solution, but this thesis will explore what effect such a model would have on automated software testing. In order to safely state if this model successfully reduced completion times, data will be presented from experimentation with the model and with traditional ways of testing.

In addition to completion time, other factors will be considered when evaluating the model. Technical feasibility, usability and cost reduction compared to current solutions will also be among the factors used to evaluate the model.

In order to efficiently distribute the different tests, and to support most tests on different operating systems and programming languages, each test would have a huge benefit of being encapsulated in its own docker container. Docker and containerisation is covered in section 2.4.1: Docker and section 2.3.3: Virtual Machines and Containers. Containers would give us more of a benefit due to a containers characteristics of being lightweight, easy to orchestrate and easy to transport. An alternative to containers is to use virtual machines for testing, but this would be a heavier process to run on the entity doing the testing. Furthermore, virtual machines rely on a hypervisor that sits on-top of the hardware. Containers have the benefit of running on top of the already existing operating system. Virtual Machines would also take minutes to be initiated, while containers only needs seconds.

3.1 Design

The design chapter will explain and present the theoretical components that need to be defined in order to develop a model based on the scheme explained in the problem statement. The main focus of this model would be to reduce the time it takes for testing to be completed. The resulting model will allow the workload of testing to be divided amongst different entities.

One can expect to find diagrams, pseudocode and detailed coverage of the model to accurately convey the theoretical attributes of the model in the design chapter so that this model might be recreated using different tools and approaches. Furthermore, different new terminologies for specifics in the model will be presented and described.

3.1.1 Model characteristics

There are several properties of the model that should be addressed and designed in order to create a model to help solve the problem statement. Based on the problem statement and the description of the "pull-based job-distributing scheme", the following characteristics should be drafted in the model:

• Handling submitted tests

In automated software testing, tests are written to test functions from software. The model will have to address how to handle all of the different files that are submitted.

• How to create the different jobs from one test

The model should take into consideration on how much work should be derived from one submitted test and what rules to use when creating one job.

• The decision to distribute these jobs

Which entities should be able to run the jobs is something that needs to be addressed and how these entities would receive each test.

3.1.2 Statistics on the tests and hosts

The data for the work that each entity would do should be measured and be given an appropriate scale that represents intensity and complexity. These types of data on the tests themselves will be used for measuring the appropriate benefit that specific entity will receive for running that test. Of course, data on the results of the tests should be gathered a long with feedback on that specific test in order to give the creators of the test best information in order to fix the issue that arose.

This model should take into consideration what effect the container has on the entity it is located on. For example, with Docker containers one can easily monitor metrics such as network I/O, CPU and Memory usage regarding the container with the command docker stats <container_name>. This will be hugely beneficial when gathering data on the container itself. This data could be analysed and used to see how big of a workload the test had on such an entity. Every bit of data that we want to take from the Docker container and collect can be stored in a data volume.

3.1.3 Industry expectations and use cases

There are many different CI/CD Pipeline software used by the industry in practise. In order for our model to be efficiently utilised by the industry, it needs to be easy to implement with little disruption to the whole pipeline or the test stage.

The many different industry practitioners may have an unique view on how our model would be implemented onto their software delivery pipeline. This may pose a challenge due to the vast variety on how automated software testing is done in a software delivery pipeline. Therefore, it may seem most valuable to have this model as a remote service which the pipeline sends a finished build to in order to successfully test the software. Different attributes such as test data and test results may reside with the remote service. When a test is finished at this remote service, the result of the test (successful or failed) may be sent back to the CI/CD Pipeline, which would then act accordingly.

Furthermore, because of the different implementations that exist in the industry, the model should be designed with versatility in mind. Different use cases and scenarios should be drafted and the model should be able to adapt to the different scenarios.

3.1.4 Expected outcome

The expected outcome from the design is a model that will work as guidance that can be used to develop a prototype or implementation of the model. Different

user scenarios should be drafted in order to cover the different use cases that will occur as the model is implemented in an industry setting. The model should be very inclusive towards the different scenarios in which it can be implemented.

3.2 Implementation

The implementation chapter will go through the approach of creating a prototype of the model, how it was created and how we can collect the data we want to measure and analyse. If we refer to the problem statement, the implementation chapter will address the effect our model from the design chapter have on automated testing. Detailed results and analysis will be presented in a later chapter.

3.2.1 The simulation

The model that will be presented in Chapter 4: Designing the model will be simulated in Chapter 5: Implementing the model. This simulation will use the processes of our model and run different scenarios in an attempt to explore how our model effects completion times in automated software testing.

It is crucial that the environment created for our model to be implemented in is also able to perform a simulation using the traditional way of Automated Software Testing. This will give a "common ground" in which our model and traditional ways can be matched against each other in order to correctly compare the different results from either approach to really observe the effect our model has on automated software testing.

From a critical point of view, the simulation should consider every aspect of the surrounding circumstances our model would encounter in a industrial implementation, such as container spawn time, image creation and internet performance. However, there are no guarantee that these characteristics can be accurately simulated as fluctuations would normally occur in an industrial setting that would affect the performance of the model. An attempt to implement these details in the simulation may be performed, but one has to consider if there are more value to limit the scope and focus on main attributes of our model that has less chance of fluctuations.

3.2.2 Data collecting and analysis

This thesis aims to explore the effect our model has on automated software testing. Therefore, we have to collect appropriate data to see which effect our model has on completion times. Data that should be relevant is the time it takes for the test stage in a CI/CD pipeline to be finished and how big and complex the tests are. These two factors are key, because it may turn out that our model might not be suited for small unittests but might excel in bigger and complex tests. In theory, we can expect that our model would be less efficient at smaller tests but will become increasingly more efficient when more tests are added on.

However, due to our decision to simulate our model instead of incorporating it accurately there are some factors that may affect our models performance.

3.2.3 Goal of our implementation

The goal of the implementation is to accurately portray the behaviour of the model and collect data that can be analysed in order to give an understanding of the effect that our model has on automated software testing.

With the simulation we will be able to measure how our model compare against traditional ways of automated software testing. By using different scenarios in the simulation we will be able to discover weaknesses and strengths in both approaches.

The simulation will be able to work from different inputs that will affect the scenario in which our model and the traditional approach would work with. Details about the inputs will be determined in the implementation chapter, but from this point we can determine that the inputs will affect the test and processing power of the testers.

Chapter 4

Designing the model

This chapter will present the design of the model named *FEAST* (Framework for Elastic Administration of Software Tests) and address its characteristics and attributes in more technical detail. First, an overview of the model is presented in order to gain a whole understanding of its workings accompanied by a section on terminologies that specifies certain parameters, attributes and the stakeholders. Each stakeholder will have an architectural pattern and different processes between them which will be presented. Lastly, this chapter will present scenarios in which the presented model may be implemented and discuss its benefits and downsides.

4.1 Overview of FEAST

When looking at the processes in a continuous integration / continuous deployment pipeline, the phase where testing is done on the software build is a phase that can be more optimised in order to reduce completion times and to make room for more testing.

This model is designed with no intentions of altering the entire CI/CD pipeline as a whole, but instead build upon the testing stage. The model is designed in a way that takes into consideration how it is to be integrated with already existing CI/CD pipeline practises. The intentions of the model is to break down the workload of incoming testing into smaller pieces, if possible, and have these smaller test pieces being tested by different entities, which would return the test results to our model. The model would then send a summarised result from the testing back to the CI/CD pipeline that initialised the testing. The entities that are running the smaller test pieces will benefit from this by receiving a reward in form of a small payment. How big the payment each worker that participate in testing should be is based on the intensity of the workload of that particular piece of testing. This way, instead of having hosts or servers acting as slaves, they will be classified as workers that are eager and yearning for more work to do. Our model will work as a "man-in-the-middle" or, more accurately, a *broker* that handles the communication between the test creators/contributors and the testing entities. Our broker will have an additional task; to divide the submitted test into smaller pieces and make these smaller pieces of tests encapsulated into containers and make them available for the testing entities.

Relations between stakeholders

Presented in Figure 4.1, the relations and dependencies between the functions in the broker, the test creator and one or more test entities is shown as a class diagram. This diagram is not extensive to include all aspects, but shows that each class have different attributes that is used to identify that specific instance and shows the communication structure between test entity, broker and test creator.

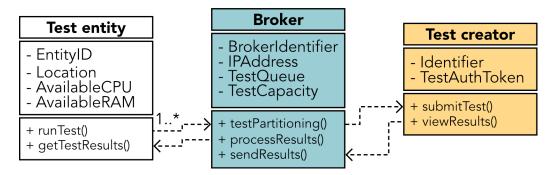


Figure 4.1: Class diagram showing the dependencies between our model (broker), one or more test entities and one test creator

As shown, the test creator is able to submit a complete test, which the broker will partition into smaller test pieces. When the complete test is partitioned into smaller test pieces these pieces are made available for, and requested by, one or more test entities. These test entities will run the smaller test and return the results to the broker, which will process the test results as a whole when all of the individual pieces are finished by the entities. When the broker is finished and created a result of the test, this summary is sent back to the test creator. In Figure 4.2, the relations between each instances component are shown more clearly to get a better overview of the flow of information and processes.

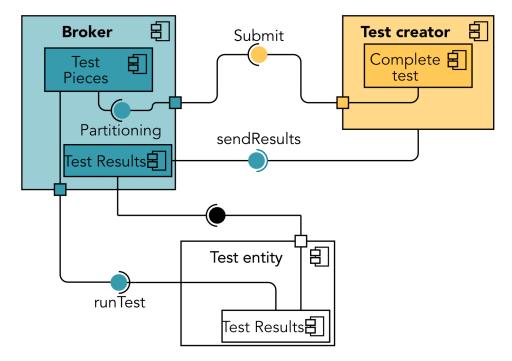


Figure 4.2: Component diagram showing the different processes within each instance

The component diagram emphasises the importance that our broker has. It alone processes the tests into smaller test pieces that are made available for our test entities to run. It then receives the test results from the entities (in our component diagram, just one), and sends a summary of the testing back to the creator.

The journey of a test

Our model will first receive a complete test that is submitted from a test creator. This complete test is broken down, or partitioned, into smaller test pieces that is encapsulated into their own independent container which is then put up and offered to the test entities that want to "work". Which entity that receives each independent piece for testing has to be evaluated in order to be most efficient at running that particular piece. This is addressed later in section 4.5.3 Availability. The entity that runs that particular test piece will gather datalogs from the testing, along with information if the test passed or not, and send these results to our broker before the container is taken down and deleted. Our model will

gather all of the datalogs from our test entities which will then be processed and summarised into a report that is eventually sent to our test creator. The entities that partook in that particular testing will at this point receive their reward based on their performance and how intense their part of the testing was.

4.2 Terminologies

This section will focus on clarifying, describing and explaining the new words and terminologies that will be used when explaining the concepts of the FEAST. The order in which the terminologies are presented is based on the overview of the journey a test goes through as described in section 4.1: Overview of FEAST.

4.2.1 Test Cycle

A *test cycle* is used to describe the process from the moment our creator creates and submits a test to the creator receives the report containing the results.

If we break down the test cycle, these are several processes included that makes up the whole cycle, which is illustrated in Figure 4.3. The processes in the test cycle starts with the creator submitting a test to our broker. Our broker will then portion the test into smaller parts that are made available to our workers. The workers will run the tests once they request work and are assigned a test portion. The information on how the test went once completed is sent by our workers and are then collected by our broker, which will combine these tests into a report which is sent back to our worker.

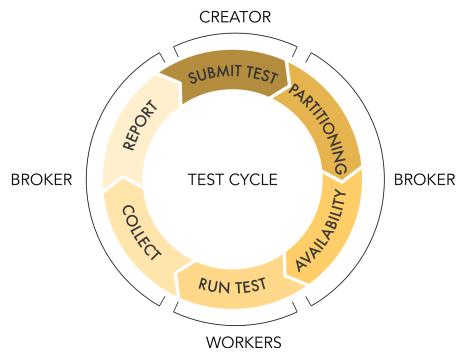


Figure 4.3: Test cycle illustration

4.2.2 Creator

A person or company that submits a *raw test* to our model is called a *creator*. This party could also be considered a client in work terms.

The information that our creator holds in terms of the inner workings of our model and which worker is doing the testing is determined by the structures defined and explained in section 4.3: Model communication design patterns. Using the anonymous testing model would provide no insight to our creator about who workers that did the testing. Both the contract broker and API based communication models would provide the client with information regarding the workers that completes the testing.

4.2.3 Raw Test

When a creator submits a test that is yet to be partitioned by a broker we call this a *raw test*. This data will consist information such as the programming language

used for testing, the software that needs to be tested and of course the tests themselves.

4.2.4 Broker

By definition, a *broker* is described as a third party that manages transactions between two parties, mostly a buyer and a seller. It is fitting to use this description for the center stakeholder in the FEAST model as it will act as a "processing point" between our test creators and those that will actually do the testing. Our test creators will only communicate with our broker and rely on it to receive a report about the test. Those *workers* that are doing the testing will only reach out to our broker in order to request work. In other words, every bit of communication between the creators of the tests and the testers will happen through the broker.

The broker will also be the entity that is responsible for keeping records on the workers that participate in testing. This information will then be used to distribute the benefits to our workers. How much each different worker should receive is decided by the platform in which our model has been implemented on, and can be based on factors such as work intensity, amount and performance.

4.2.5 Partitioning and Encapsulation

When there are mentions of partition in this project, we are mainly referring to the act of splitting a test as a whole into smaller test bites based on decisions such as the amount of unit tests. These individual small code blocks, and its dependencies, are packaged together into a container. This is encapsulation. Often used to explain container technology, encapsulation is packaging code into a container to create a shippable application or service.

Broker splitting vs. Test creator encapsulation

There are two possible approaches to encapsulate each test into test bites. The first approach rely on the broker to have a sufficient amount of knowledge about unit tests in different programming languages in order to accurately and most efficiently divide the whole test into test bites. The second approach rely on the test creator to be the one partition the test into one or more containers, or bites, before submitting those to the broker. These two approaches are the

best practices of software testing where the former is more code focused, which means that the developers are required to program in a way that makes unit tests possible. An example of a code focused approach for testing is unit tests. The latter approach to testing tests the service as a whole and does not rely on a code structure that supports testing. A file to create the container image (called Dockerfile in Docker) would be included in the code base with instructions on how to create the image and run a container.

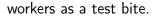
If the task of encapsulation lays with the broker, the test contributed by the creator will be in one solid code. Additionally, the test creator would be required to provide information on which programming language the tests are written in to let the broker know what language it is looking at. Based on this information, the broker will then have adequate information to be able to know how to look for unit tests in the source code and also to retrieve their dependencies. The broker should also have access to a repository of container images that would act as templates in which each unit test would be inserted into before made available to the workers.

Should we relieve the broker from the job of partitioning and encapsulation, then we may allow the test creator to be in charge. This would give the advantage of letting those that actually wrote the tests to evaluate and decide the best way to partition and encapsulate. They could then partition the test based on functionality, dependency or unit tests. The test creators would be given more freedom, but more job would be required by them.

One advantage of letting the broker split the test into smaller test bites is a possible reduction in test time. The broker would be allowed to look at the source code, split the unit tests into smaller bites, encapsulate these into their own container and distribute these to the workers. This allows for the tests to be run in parallel, which could reduce completion times. If the test creator submits one container for testing, then this would result in one intense test bite that would be distributed to one worker. This would result in longer test time.

4.2.6 Bite

When a *raw test* has been *partitioned* by our *broker*, it will end up as multiple bites that will be made available for our *workers* to complete after each bite has been *encapsulated*. An illustration of this is shown in Figure 4.4 where (1) a raw test has been submitted by a creator, which is then (2) broken down, or partitioned, by our broker and lastly (3) encapsulated and made available for our



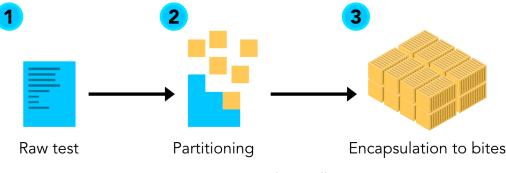
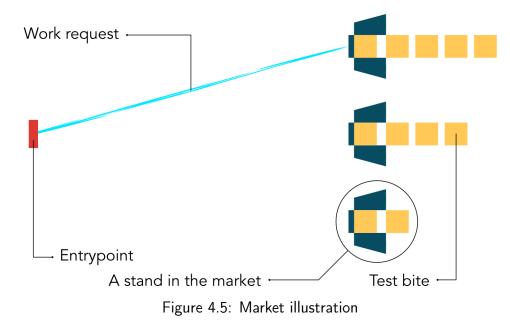


Figure 4.4: Bite encapsulation illustration

4.2.7 Market

Each partition of the test is made available for the workers at what we would call the *market*. This component is where workers request work and where our broker is placing the encapsulated bites of the *raw test*.

If you imagine a market with many different stands that would sell everything from fresh garlic and clothes to statues and cloves, our broker will place test bites in different stands which will be picked up by a workers that request work. A illustration of this can be seen in Figure 4.5.



4.2.8 Worker

A *worker* is a single host/entity that is part of a collective of other workers that our broker has access to. The workers as a collective share a common desire to do testing in order to receive benefits. Though they are part of a collective, they will work independently and request work from our *broker* independently. When a worker completes a test they will receive benefits from our broker based on factors about the test.

Each worker will have different attributes that specifies how effective they are at testing. These attributes may be based on the computers specifications, location and accessible memory and processing power. Further attributes such as when the worker is available to work could also be used to describe the effectiveness of a worker.

4.2.9 Test Results and Reports

Test Results refers to the outcome of one particular test done by one individual worker based on the results from running one test bite that the worker retrieved. A *report* is the final feedback generated by our broker based on the different test results. The resulting report will be sent to the creator, which completes a test cycle.

4.3 Model communication design patterns

There are different ways of incorporating the communication between the stakeholders in our model. This section will present three different approaches in which our model may be applied, each of which have their own pros and cons in terms of control and flexibility.

The FEAST will have two main stakeholders to consider; testers and test creators. Seeing that the model will always have a centralised role, the decisions on how transparent the communication between the testers and test creators will lie entirely on those applying our model.

The following three proposed approaches to manage communication between the stakeholders will give an overview and show the diverse ways our model may be

implemented.

4.3.1 Anonymous testing

One approach would be to implement our model with neither of the stakeholders knowing anything of each other. The platform, in which the model would be implemented, would handle the relationship and connectivity between our test creators and testers and be responsible for anonymity.

Using this implementation, our model will have to reside within a third-party that is responsible for the connectivity between both stakeholders. Neither of the stakeholders would have any control but rely entirely on the third-party to maintain anonymity and data integrity. A simplified overview of the communication between stakeholders in our model can be seen in Figure 4.6. Our model can be described as an indirect way for companies to communicate with testers, which will ensure anonymity between the two parties.

This type of communication can also be classified as "anonymous communication" because neither of the two stakeholders (testers and those submitting the tests) know anything about each other. Both parties trust that the broker will keep track of the different tests and know the true relations between the test submitted by test creators, test pieces and the testers.

From the perspective of those running the test, they will be oblivious to who they are testing for. However, they will have the benefit of always testing as long as there are testing to be done. The tests that the companies submit will be made available for all testers. This approach will be unbiased towards which company testers test for and which testers are chosen by the company.

The companies will have to sacrifice the control of who runs their tests, but will gain the luxury to always have testers available to run their tests. However, companies might desire some sort of priority in testing, which this implementation does not consider. No company will be favored as opposed to others.

An overview of the different characteristics with the described anonymous testing implementation can be seen in Table 4.1. This implementation will highly benefit the testers. They are not obligated to be available for testing within certain hours, but may be available whenever they want and perform tests how long they desire. This implementation shines when there is a high amount of testers available for testing and where the demand for testing exceeds the amounts of tests. In this

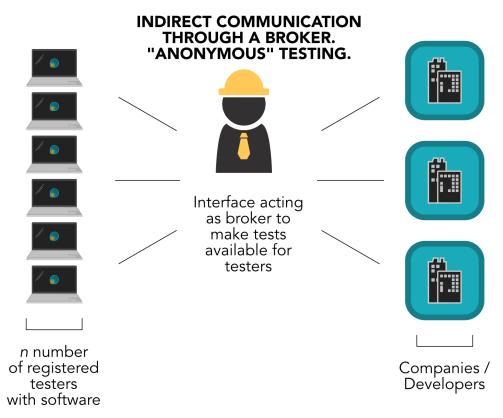


Figure 4.6: Simplified illustration of communication between stakeholders for our broker solution

scenario, the companies will have the benefit of always having testers testing their software and can expect a quick response to the test. The more available workers, the more optimal workers may be chosen to perform the tests.

Stakeholder	Control	Flexibility	Anonymity
Testers	High	High	High
Companies	None	None	High

Table 4.1: Comparisons of benefits with both stakeholders in the broker model

4.3.2 Contract broker

Alternatively, the communication could be more transparent. Instead of a anonymous communication between the stakeholders, we could have our model rely on the contracts created between the stakeholders on a platform. The use of this communication structure for our model can then be called a "contract broker" that manages and holds contract between the parties.

The third-party that manages these contracts would then serve as a place where companies can show interest of outsourcing their testing to. This third-party would provide every company with their own broker that would manage the communication and distribution of tests between the company and its testers. People that are interested in signing up as testers could then view the list of interested companies and sign up as a tester for that particular company. Companies might have certain requirements for their testers such as location, computer performance and availability and might want to evaluate the different testers before going into a contract with them. This third-party that manages the contract would have to evaluate whether they would allow one tester to sign contracts with multiple companies, or arrange exclusive contracts, e.g. that limits one computer to one company.

DIRECT COMMUNICATION THROUGH A BROKER THAT MANAGES CONTRACTS

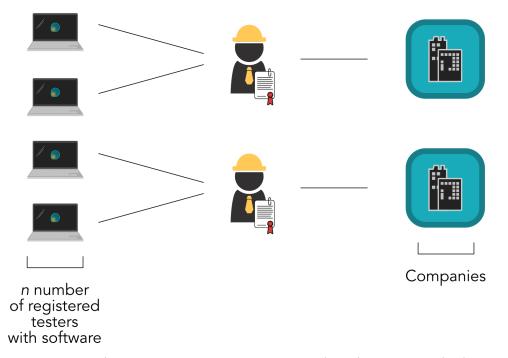


Figure 4.7: Alternative communication approach with a contract broker

Our model will still be responsible for dividing the work amongst several testers, but the platform would be responsible for creating individual environments where one company has its assigned testers, which is achieved by giving each company their own broker. The platform would then have isolated models that works with one company and the testers that have signed contracts with that particular company. A simplified illustration of the contract broker can be seen in Figure 4.7.

The direct communication model would sacrifice efficiency in favor of control. With a control over how many testers one company has, it limits the efficiency for not allowing the numbers of testers to expand further than the set limit. This might be a desired way for some companies that want to limit the pricing and instead sacrifice test time. Constraints might apply in the contracts where one company might require that the testers are available for running tests in certain hours and have a certain amount of hardware specifications.

But using this type of communication introduces more aspects that are of concern from one testers perspective. The testers are only testing on behalf of the company they have a contract for. Even though the tester is available to test for a large period of time, exactly how much work that worker is assigned to do is entirely up to the company. In order to become more efficient at testing once a tester becomes available, they will have to sign contracts with multiple parties in order to fulfill the potential testing that they want done. Seeing that the testers would have control of who they are testing for might be beneficial in terms of transparency, but will limit the time they are actively testing. As shown in the overview in Table 4.2, both stakeholders will benefit from this communication structure. However, testers will have the responsibility to be available for testing as per their agreement with the company. If they want to become more active in testing they may, as previously mentioned, sign up as testers for more than one party. This would introduce a risk. Should company A, B and C send out tests to the same tester, they might experience getting a processing power from that particular tester that differs from the one specified in their agreement.

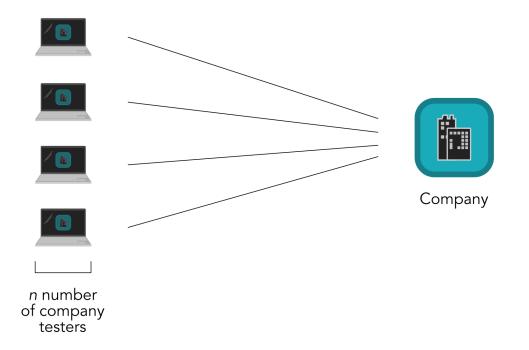
This design pattern and its characteristics would create a business relationship between the company and its testers. The testers would be obliged to meet the requirements set by the company in the contract signed by the two parties, which will require the testers to act as professional entities. These testers will have to maintain their machines in order to ensure that they meet their obligations. This approach will not appeal to the participants that wants to easily and casually participate in testing whenever they please.

Stakeholder	Control	Flexibility	Anonymity
Testers	High	Medium	None
Companies	High	Low	None

Table 4.2: Comparisons of benefits with both stakeholders in the contract broker model

4.3.3 Direct based communication

The last communication model to be covered puts the responsibility of testing entirely to a company. Instead of outsourcing the testing to a third-party, the company will run their own service that could be strictly available only for their own employees. This model can be called a "direct communication model". An illustration of this can be shown in Figure 4.8.



DIRECT COMMUNICATION

Figure 4.8: Illustration of our model with a direct communication structure

With this structure, the model can be integrated as a service (such as a SaaS or FaaS) or a self-hosted service that resides within the data center of a company. One company will then have control over their own broker. Testers can join a company's broker and become active testers for that company. This could potentially give more control to the company in terms of who gets to join and what benefits they get by running tests. The company is given no restrictions and can operate their testing as they would like. They can limit the testers to only consist of employees or reach out to people that live locally.

As shown in table 4.3, this structure would not be beneficial for testers at all due to a missing platform where testers may sign up for to become an active tester. Such a structure would create closed environments where companies independently structure the amount of testers and who that may become a tester.

Stakeholder	Control	Flexibility	Anonymity
Testers	Low	Low	None
Companies	High	High	None

Table 4.3: Comparisons of benefits with both stakeholders in the API model

4.4 Architectural design

Up until this point there has been no mention of the architectural design behind the model. Important aspects of our model has been covered, but defining the software architecture of our model will give great benefit when explaining the different relations between the different components in the model. Explaining the patterns will also yield a fundamental understanding of the complexity within each different aspect, which will be beneficial when implementing this model. This section is divided into sections that corresponds with the total amount of stakeholders for our model; the *creator, the broker* and *the testers*. Each stakeholder, and its processes, will use certain architectures when implemented which will be covered in this section.

4.4.1 Creator

The architectural pattern for the creator will consist of a *Layered pattern*. Typically, within a layered pattern you will first find the *presentation layer* followed by *business logic, persistence layer* and the *database layer* at last (Richards, 2015). Our model does not require a database to store data to, but our model will depend on the other layers, as displayed in Figure 4.9.

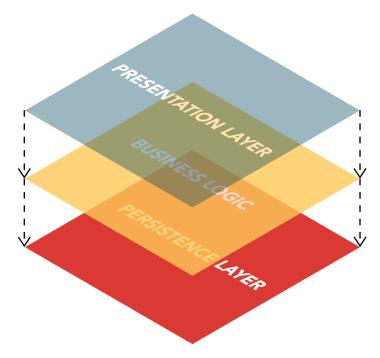


Figure 4.9: Illustration of the layered pattern

The presentation layer will be the user interface in which the creator interacts with to create tests. This layer can either consist of the front end and back end in a web application, or the GUI (*graphical user interface*) of a software application. The design of the interface, how the reports are displayed and the forms in which our creator can use to create the tests are important factors that lies within the presentation layer.

When retrieving or creating data to the database, *business logic* will be responsible for encoding the real business rules into logic within the application or software. The business logic will decide exactly how data is to be created, stored and changed. This are in most cases heavily influenced by laws and regulations such as the GDPR (*General Data Protection Regulation*), a law set by the European Commission to protect the European citizens' online data from misuse. The business logic for our model will generally focus on how the data is kept secure through the test cycle and that it is made anonymously. Keeping the CIA triad (see Figure 4.10) in mind will come in handy at the business logic layer to ensure security and integrity for the data.



AVAILABILITY Figure 4.10: The CIA triad

It is important to keep in mind that the testers should not be able to retrieve any information regarding the software that is being tested. A company will treasure its software. The model should be able to test software that might originate from a company's R&D (Research and Development) department without having to be concerned about the software being leaked.

If we keep up the database analogy, the persistence layer is where operations within CRUD (*create, retrieve, update,* and *delete*) would happen. We can, for a minute, pretend that our broker would be the database in this analogy. The persistence layer will be responsible for accurately transmitting the test data from the business logic layer to the broker, and the results from the broker to our creator through the business logic and presentation layer. So, instead of the persistence layer using operations within CRUD, it will be responsible for communicating with the broker.

The persistence layer is especially important because of its key role in our model by communicating with the broker. By having this layer as an intermediate entity between our broker and the creator, we can address issues related to, for example, which broker our creator should rely on. We can assume that an implementation of our model on a wide enough scale will have different brokers located on different locations with different workers associated with each different broker. The persistence layer will make sure to establish a connection (or a relationship) to the most suitable broker. By isolating this issue to the persistence layer, this layer can change as much as it would prefer without affecting the other layers.

One more software architectural pattern that is present is the relationship between the creator and the broker. We can consider this pattern to be a *client-server pattern* in which a client is connected to a server. Our broker would then act as a *server* that will process a request of an input made by the creator, which acts as the *client*. Our server will then process the input by having workers test it and report back to the server, which will generate a report and send back to the client.

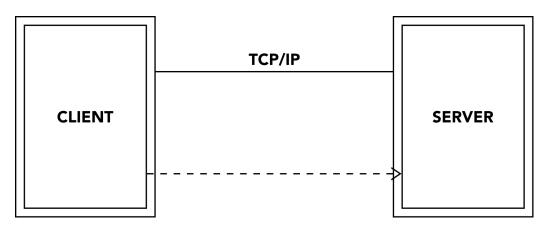


Figure 4.11: Client-server pattern

4.4.2 The Broker

Our broker will incorporate one main software architectural pattern; the *broker pattern*. This type of pattern is responsible for coordinating communication between components. This pattern suits our model very well because our model distributes test bites to different testers. However, the broker architectural pattern depends on the different components to report to the broker about their specification which allows the broker to make a decision on where to send data based on those specifications. In Figure 4.12, the traditional broker pattern is illustrated as well as the altered pattern that will be better suited for this model.

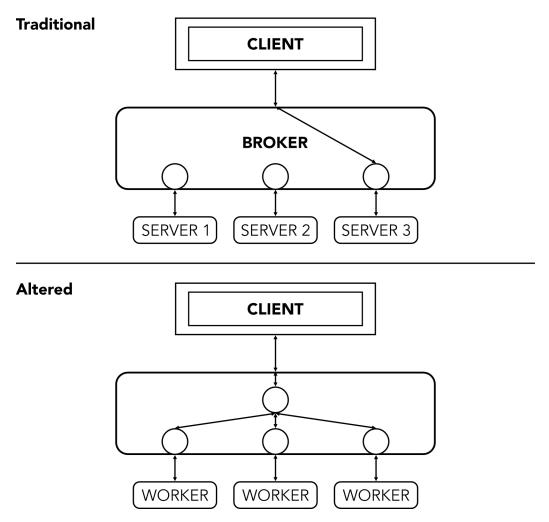


Figure 4.12: Illustration of the traditional broker pattern and altered

Optionally, one can incorporate the *pipe-filter pattern* in order to mainstream the process of encapsulation. Filters can be created to capture different tests from different programming languages and encapsulated these in their own container with the appropriate environment. If tests are written that is not recognised as tests, or written in an unsupported programming language, these would not be captured by any filter and could be returned to the creator as error messages. This pattern would have to be incorporated on a step between the broker pattern and the creator. Our model could potentially become more efficient by introducing the pipe-filter pattern before the broker pattern. This is further illustrated in Figure 4.13, which shows a activity diagram on the workflow the filter would go through.

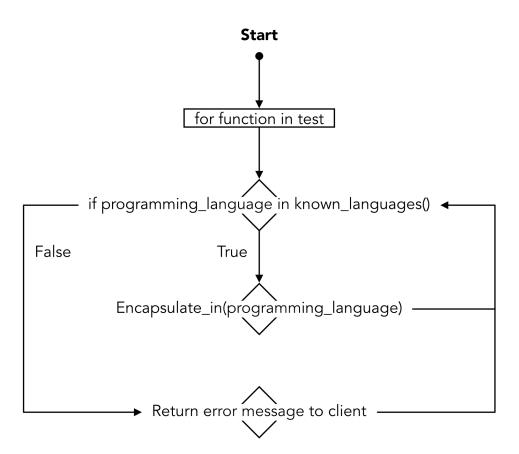


Figure 4.13: Activity diagram of the optional pipe-filter pattern

4.4.3 Worker

A worker will sign up to be part of the broker architectural pattern. In the relation to the traditional broker pattern, a worker would act as a *server* that does certain tasks that are assigned by the broker. Specifications about the worker will be sent to the broker that the broker uses to give appropriate work. The relationship between the broker and any given worker would share similarities with the client-server pattern where the broker would act as *client* and the worker as a *server*.

The worker will also, like the creator, use a layered pattern in order to present data to the owner of the host. The presentation layer will consist of an interface in which the owner may use to see the overview of tests completed and the earned benefits. The business layer will ensure integrity of data that best represent the business logic in the implementation of the model. If anonymity is desired then the business layer has to ensure this. The persistence layer will take the results of the completed workload and send the most crucial data back to the broker, which includes pass/fail and logs.

4.5 Key Model Processes

This section will cover the different processes associated with the model. Processes and the communication between the different stakeholders will be thoroughly described and illustrated. The processes that will are covered are the same ones that are described as part of a test life cycle as described in *4.2: Terminologies*.

4.5.1 Submit test

If the broker is responsible for partitioning and encapsulating, then the creator will have to make ready a set of tests that are to be submitted to the broker. Probably the simplest, and most preferred, type of tests would be unit tests. Our creator will prepare a set of tests based on the unit testing framework in the programming language of their choice. A pseudocode of a unit test can be seen in *Listing 4.1* where a function from the software is imported and used within a unit test function. At its simplest form, this test will just return either True or False.

```
1 START
2 IMPORT function1
3 Function test-function1
      Pass In: nothing
4
      Runs a function and compares a desired output to actual output
5
      Pass Out: BOOLEAN
6
      SET test_input: INT
7
      SET desired_output: INT
8
      SET test_results: call: function1(test_input)
9
      IF test_results IS desired_output THEN
10
          RETURN True
11
      ELSE
12
          RETURN False
13
      ENDIF
14
15 Endfunction
```

Listing 4.1: Pseudocode of a unit test

This function is very simplified in order to show the basics of unit tests. Frameworks such as Pythons unittest offers more complicated ways of performing unit tests and structures each unit test as functions within classes. The resulting implemented model will have to know how to handle unit tests within each programming language due to the fact that different languages have different structures on their unit tests. For example, Python3 uses functions within a class that inherit a test object while Golang uses specific test functions.

Should the test creator handle partitioning and encapsulation, then the broker does not need to know the programming language used to create the tests. Since the decision to where to divide the tests lays with the test creator, the broker will only pass the finished encapsulated tests to the workers.

When a test creator submits a test to be sent to the broker, they will receive an identifier which will be used to routinely check with the broker how their test are doing. This is know as performing a *busy-wait*, in which a repeated execution of code is performed while waiting for an event to occur. The event in this case will be for the broker to be finished gathering all the test results. This would result in the test creator to be performing routine connections to the broker in order to check for the test results instead of the broker reaching out to the test creator. One advantage of this would be that the test creator does not have to worry about loosing the possibility of receiving a test report in the event of changing IP addresses.

4.5.2 Partitioning (and encapsulation)

Once the test as a whole has been successfully sent to the broker a long with the software that is to be tested, the time for partitioning and encapsulation starts. The programming language should be defined prior to this process by the creator so that there will be no mistakes as to what language the tests are written in.

There are several ways to decide where to partition the whole test. A broker may be implemented so that it splits the test into bites based on each function. This would create a lot of small bites made available for our workers but might introduce unnecessary duplicates of functions that are to be tested. One function may have many unit tests associated with it. By encapsulating each unit test associated with this function, the function has to be copied within each encapsulation. This causes a lot of bites that may include duplicate dependencies. As shown in the equation below, we can calculate the number of bites one test would result to. If we assume test to be t which contains a n amount of functions f:

$$t = \{f_1, f_2, f_3, \dots, f_n\}$$

Each function f will have a set U_f of possible unit tests u associated with it:

$$U_f = \{u_1, u_2, u_3, \dots, u_n\}$$

If we let u(f) denote the possible independent unit tests from the set U_f which can be executed independently of each other, then we can use the following expression to find out the number of possible bites B_t of test t:

$$B_t = \sum_{i=1}^n u(f_i)$$

The approach of having multiple containers contain the same function and dependencies but different unit tests might seem redundant, but might be applicable and most efficient in some cases.

To avoid duplicates of the function that is to be tested, we can partition the unit tests based on each unique function. One bite would then consist of one function that is to be tested and the associated unit tests. The number of bites would therefore not be based depending on each unit test, but rather the amount of functions.

The types of test bites in term of size and requiring more processing power would be the integration tests. Because of their dependencies to other functions, each encapsulation of an integration test will have to include the required functions.

In the case of a broker being responsible for encapsulation and partitioning, the broker will have to go through the source code and retrieve every function that is present and match these with the provided unit tests. When each function is retrieved and matched with all unit tests, these will be encapsulated with their own environment in a container. This container will be based on a programming language, e.g. Python, and contain the piece of source code that is to be tested (the function) and its associated unit tests. The containers can be optimised by not including every library that it is dependent on but rather analyse what libraries are imported in the file to that particular function. This is very useful especially if a large library is used throughout the source code, e.g. MatPlotLib

for Python. Should a vast library be used in a function then that dependency will be included in every encapsulation which will result in more intense testing processes. The creators of the tests will therefore be responsible for this instead of the broker. The creators should therefore be more specific as to what functions they are importing when using big libraries.

4.5.3 Availability

The resulting container image that is created from the partitioning process will be made available for the workers that can retrieve the image and run it on their host.

Technical approaches to communication structure

There are several approaches to implement and manage the communication between the broker and workers; TCP/IP Sockets, using message queues or a asynchronous API.

A TCP/IP Socket approach would require an open connection to be established between the broker and a worker. Communication between the broker and a worker will consecutively go through this socket until the connection is closed. A disadvantage of this approach is that it requires the connection to be open for the entire duration of the test, which makes this vulnerable to network disruptions.

Message queues is essentially a queue of messages that are waiting to be sent between applications. These messages consists of a sequence of work that await processing by the receiver. As illustrated in Figure 4.14, the broker will act as a producer of messages (or test bites) which will be put in a message queue. The workers will then be authorised (e.g. by giving them an access token) to subscribe to this queue and start to consume the processes that are put into this message queue. The broker will put the messages in the queue and know that the processes will be taken if there are available workers, or when a worker becomes available.



The last communication approach, using asynchronous API, would resemble that between the test creator and the broker. Each worker would communicate with the broker frequently to see if there are any work to do. The broker, which now know about the subscribed workers, can make ready a particular test for the worker that asks for work. Alternatively, the broker could control a queue of test bites and distribute these to the workers that requests work.

In principle, the first workers to respond should also be the ones that are first in line to receive work. Should one worker be first in line but not meet the requirements for one particular test, the next person in line that is fitting to do the job should be assigned that particular test case. One can say that this would follow a FIFO-type (First In, First Out) queuing system.

When the broker finds a suited worker, a message is sent to the host to ask if they are still available to do the work, if so they will receive the container image. If no reply is received from the worker when reaching out to them, they will be put out of the queue. Once the worker has received the container image, a reply is sent to the broker to confirm this. The broker will then wait for a second reply from that worker, which will contain the results of the tests including logs.

When a worker is working on a test, there are two approaches that can be implemented to make sure that the workers are working on a test and when to assume that the worker has abandoned testing. The first approach relies on the similar processes of "busy-waiting" as mentioned previously. This would involve the worker to frequently send a message back to the broker letting them now that they are working and not to assume that they have left testing.

Alternatively, the broker could have a time in mind as to when to expect the test results to be received. Expected time is based on the specifications provided by the worker. A good margin is added to the time to compensate for any unexpected fluctuations such as temporary slow internet connection. Should the time run out before a test result has been received, a message can be sent to the worker to find out if that worker is still active. Should the worker be non-responsive, an other worker will be assigned the work and the connection to the "slow" worker will be terminated. It would then be too late for the slow worker to send test results. The expected time set by the broker might also be shared with the worker a long with the work so that they know how much time they have and may cease the testing if the time is running out. This worker can then announce that they are available for work much sooner than if they were to complete work that were too late anyway. The time shared with the worker have to altered to consider just exactly how much time the worker has on running the

container image and not include details such as extra margins or fluctuations in the connection speed.

Whichever approach is chosen, the broker will have to know how to act if a worker becomes non-responsive, has left testing or uses to much time when they are assigned a test to complete. When no test results have been sent from a particular worker, then the broker may send a message to ask if that worker is still active and discontinue the relationship with that worker if no response is received. Should the worker be non-responsive or take too much time to test, then the test bite sent to that worker will be distributed to an other worker.

Addressing the Payload

An important aspect of the model is handling the payloads sent by the broker to each tester. A payload can be described as the main cargo in transportation, and within the computational world it refers to the actual data that is transferred between two devices in a connection.

In an implementation of the model, including the time it takes for distributing the payload and the dependencies is something that should be considered due to its impact on transmission time and test time.

Transmission time of a payload can be calculated with the approach mentioned by Kurose and Ross (2013, p. 30) that calculates the transmission time for a network packet. The max size of a packet sent with TCP/IP is 524 280 bits (or 64K bytes). We can therefore calculate approximately how many packets P_d our container image would be divided into if we divide the size of the image in bits b with the max packet size:

$$P_d = \frac{b}{524280}$$

From there we can calculate the time it takes to transmit one packet of the container image to one worker by assuming that the overhead time O consists of communication factors such as the packet size in bits pb, internet speed in bit/s i, distance from the broker to the worker d and signal speed s we can use the following expression to calculate how long it will take to transmit one packet:

$$O = \frac{2pb}{i} + \frac{d}{s}$$

In this model, the container image distributed to the testers would be considered as an important payload. However, this payload is unique and has to be addressed.

Container images created with Docker saves storage space by layering changes instead of creating new images from the bottom whenever new changes occur. This will drastically reduce time to create a container and save storage space.

A short demonstration of building a image from scratch can be seen in listing 4.2 where a new image for a website is created for prototyping purposes. It is created with Flask and Python and is not incorporated in a webserver yet, it instead rely on the development server that is included in the Flask framework. The build took 76.8 seconds where 54.8 of these seconds were used to download the underlying docker image that this image is based on.

1	Emi	ls-MBP: emilgabrielli\$ docker build -t "website:v12" .	
2	[+]	Building 76.8s (8/8) FINISHED	
3	=>	[internal] load build definition from Dockerfile	0.1s
4	=>	=> transferring dockerfile: 212B	0.1s
5	=>	[internal] load .dockerignore	0.0s
6	=>	=> transferring context: 2B	0.0s
7	=>	[internal] load metadata for docker.io/library/python:3.7	2.8s
8	=>	[internal] load build context	2.1s
9	=>	=> transferring context: 710.28kB	2.0s
10	=>	<pre>[1/4] FROM docker.io/library/python:3.7</pre>	54.8s
11	=>	[2/4] COPY	4.5s
12	=>	[3/4] RUN pip install -r requirements.txt	11.8s
13	=>	exporting to image	2.4s
14	=>	=> exporting layers	2.4s
15	=>	=> writing image	0.0s
16	=>	=> naming to docker.io/library/website:v12	0.0s

Listing 4.2: Creating an entire new image

For comparison, in listing 4.3 the underlying image is already downloaded. Changes are detected in step 2/4, so only the new changes from this step and up are applied to the newly created image while the underlying dependencies are taken from cache.

```
1 Emils-MBP: emilgabrielli$ docker build -t "website:v13" .
2 [+] Building 17.3s (8/8) FINISHED
3 => [internal] load build definition from Dockerfile 0.1s
4 => => transferring dockerfile: 212B 0.1s
5 => [internal] load .dockerignore 0.0s
```

6	=> => transferring context: 2B	0.0s
7	=> [internal] load metadata for docker.io/library/python:3.7	1.5s
8	=> [internal] load build context	1.1s
9	=> => transferring context: 915.00kB	1.0s
10	=> CACHED [1/4] FROM docker.io/library/python:3.7	0.0s
11	=> [2/4] COPY	1.2s
12	=> [3/4] RUN pip install -r requirements.txt	11.5s
13	=> exporting to image	1.8s
14	=> => exporting layers	1.8s
15	=> => writing image	0.0s
16	=> => naming to docker.io/library/website:v13	0.0s

Listing 4.3: Creating a new image where dependency is cached

In terms of how Docker operates with layering, optimisations can be done to our model to prioritise those workers that have completed similar work before. A worker that will have to download the underlying image before building the image for testing will have a disadvantage compared to one that already have that underlying image cached. A worker that has completed similar work, and therefore have underlying image cached, will run the tests much faster. It would therefore be beneficial for the broker to distribute images based on, e.g. Python, exclusively to workers that have already processed a Python image. However, we can assume that the most beneficial method for completing a set of tests as fast as possible is to distribute the test bites to as many as possible. The maximum time to wait in this case would then be the sum of the overhead time for every packet plus the time it takes to complete the longest test with the biggest dependencies to download. This aspect of how to handle the payload is worth considering, and if Docker is involved it is worth addressing layering for optimisation.

4.5.4 Running a test

Once a worker has been assigned a test bite, the worker will automatically create and run a container from the container image. The output logs created from running the container will have to be saved and sent back to the broker. In the case of using Docker containers, the logs of a container can be retrieved by running the command docker logs <container_name>. The broker will expect to receive these logs from the workers.

Neither the worker or the broker will have any benefit of been given insight to the file that contains the logs from the container. On the contrary, it will sacrifice

confidentiality. It is of high importance that only the test creator receives and views these logs in order to fix any errors or bugs in the code. Referring back to the CIA triad mentioned in section 4.4.1: *Creator*, keeping unknown parties from getting insight in the source code is important to uphold confidentiality. Therefore, it would be best to have these logs be written by using encryption as a security measure.

4.5.5 Collecting output and rewarding workers

Whenever a worker is finished with its test bite, and the test results have been successfully received by the broker, the worker will receive benefits. These benefits should be stored on the server-side as not to be vulnerable to client-side manipulation.

Which factors should be included in order to determine the appropriate benefits for the worker may vary. Initially, the calculations should include a currency rate. This rate may be a fixed pay rate (e.g. 10¢/seconds testing) or based on how much the creator paid for having that test be run (e.g. 2\$) which will then be divided among the model implementers and its workers. The former may be somewhat difficult to justify due to the varying computer specifications the workers may work with. One worker with a small CPU should not necessarily be paid more because it took longer to complete a test due to poor specifications. Instead, we can have the broker to expect how long time it will take to complete a certain test and pay by this rate (e.g. 10¢/expected seconds of testing). Alternatively, the fixed pay rate might be altered based on the workers specifications. This way, a more powerful computer will receive more per second, and a less powerful computer will receive less per second. The latter payment approach is much simpler to rely on where the amount is shared between the workers and model implementers, but this may cause a huge variation of fluctuations from the workers perspective.

We can calculate the percentage that one worker will receive in benefits, based on its work time, by first assuming that B_t is a set of all the bites b belonging to one submitted test:

$$B_t = \{b_1, b_2, b_3, \dots, b_n\}$$

Each bite b will have its own execution time e which will be based on the time of previous, similar, completed tests:

e(b) = Execution time (from history)

The collective execution time E of test t can therefore be expressed as follows:

$$E_t = \sum_{i=1}^n e(b_i)$$

This calculation will not be able to tell us how long the test will take, but how much time the workers will be paid for.

The price p received by the worker for each bite b is based on the payment P from the test creator for the test t and can be expressed as follows:

$$p_b = \frac{e(b)}{E_t} * P_t$$

One point to consider is the relationship between the transmission of payload and the payment. In a scenario where the completion time of a test is much shorter than transferring the test and downloading the subsequent dependencies, the expression for e(b) may be extended to include the overhead time for transferring the entire image and downloading the dependencies.

$$e(b) = Execution time + total overhead$$

There is a risk, when splitting the tests into bites, that the time of a tests total overhead will exceed the time of running the test. If overhead would be included in the calculation for payment, then the time it takes to download dependencies would exceed the time for testing by a comparably large amount. In that case, then the expression could be simplified because the time it takes for testing would be insignificant. If we assume payment per bite p_b to be equal to 1 divided by the amount of bites n multiplied by the payment for the test.

$$p_b = \frac{1}{n} * P_t$$

However, if the test creator were in charge of encapsulating the tests into their own containers and the broker would just pass these containers along to a worker, then the test time will be dominant. Overhead time would most likely no longer exceed the test time. This would also eliminate running tests in parallel, unless the creator would set this up as the test is submitted. In this case where overhead would no longer be dominant in terms of time, the former expression would work by calculating the percentage each worker would receive.

4.5.6 Report

The broker will wait for every worker to transmit the test results before generating a final test report. This test report will be similar to what is normally expected to be the output log from a test suite, such as Jenkins, which will contain useful information to the creator about how well the software performed with the unit tests.

There are several decisions needed to take regarding how detailed the report should be. Initially, due to the fact that our model will be part of a CI/CD pipeline, the report should mainly consist of a message that tells the pipeline whether the software passed all tests or not. If all tests passed, the CI/CD pipeline may continue with delivery. If not then actions from the developers should be made and further information regarding the test should be provided.

The report can contain a lot of useful information regarding each unit test. Though this may be useful, all of this information might cause some delay in transport especially if there was a huge test. The most important factors to include is information regarding those unit tests that did not pass. Logs from those unit tests are most important to send back to the test creator, which will better help the developers behind the software and the unit tests to fix bugs and errors.

The test results from those unit tests that did not pass, which could be encrypted from the worker, could be combined in one report while still encrypted. This would then just pass this along back to the test creator, which would then be able to decrypt and read the logs. The only "open" information that the broker will have to know is if all the tests was successful or if one or more failed in order to send this information back to the pipeline. Keeping these data encrypted will not allow the broker to get an insight in the logs, further increasing confidentiality.

4.6 Examples on model implementations and their implementations

So far, FEAST has been presented and described with different approaches in mind. Which approach that is best suited will depend entirely on the situation and context of the implementation.

This section will present some scenarios in which the model may be implemented, which approach that should be utilised and how the model would work in practise.

4.6.1 In-house testing

A tech company with a good amount of employees might take advantage of the model by integrating it in-house. This would allow the company to have a controlled test environment with a max amount of testers to be equal the amount of computers that are active during business hours. Each computer would then run the tests in the background and not interfere with the employee while working.

Each employee will then have a copy of the "worker software" installed on their work computer. This software will then register as a worker to that company's broker.

Communication model

The most attractive communication model would be the "API based communication" model. This would give the company control of who joins that company's broker. As long as it is installed on the employees worker computer, there might not be a need for the worker/employee to have any control or receive benefits for testing. Every computer that is owned by that company could therefore be registered to the broker and participate in testing. This would be the most secure option because every computer is owned by the company and the risk for data leakage is reduced.

Workers benefits

This implementation scenario assumes that only company-owned computers are used for testing. The company will most likely decide to not give any payment

to the employees for running tests since the computers are property of the company. An important factor is that the testing should not be so noticeable for the employees when performing other work.

Discussion

Implementing the model within a company will yield many benefits. The company will be able to perform in-house testing instead of relying on third-parties and save money that would otherwise go a testing infrastructure. Furthermore in most practises, employees rely on computers to do work. This means that the company will have to purchase computers for their employees anyway. By implementing this model the company will gain further benefits from their already purchased devices.

A disadvantage of this implementation is that computer processing power is drawn from the employees computers. This might affect the work for the employees when testing is in progress. The effect of this can be minimised by purchasing more powerful computers for the employees but this would in return result in a increased budget. More powerful computers might also tempt the company to utilise more processing power. This will increase the efficiency of testing but again might affect work performance by the employee. Furthermore, most employees use laptops which might be "unstable" in terms of performance if the employees were to come and go.

4.6.2 Offering a "True" Testing-as-a-Service

By redefining the *Testing-as-a-Service* definition as described in section 2.3.1: *Cloud services*, a vendor can offer a *True Testing-as-a-Service* which will more closely resemble the other cloud services.

The cloud service vendor that offer this service can implement our broker on different locations across the globe to offer a wide coverage. This vendor will also be responsible for hiring workers, which can be done by offering benefits to those that will download the worker software and participate as a worker for that vendor. In order to attract more processing power and coverage, these workers may be anyone with a working computer.

Communication model

The best option for this implementation would either be the *Anonymous testing* model or the *Contract broker* model. All depending on what business model that vendor follows.

An anonymous model would be more efficient where the workers are decided based on location from the closest broker. This would be the unbiased approach. The contract broker would be more transparent where the vendor would run a platform where other companies could post their interest in testing and independent workers would sign up as a worker for that specific company. This option would be the most balanced in terms of control.

Workers benefits

Vendors that would offer "True" Testing-as-a-Service would either rely on their own testing infrastructure to become workers, or recruit members from the public.

If the vendors are confident that their own testing infrastructure might suit the need for companies around the globe, they do not have to worry about benefits to the workers because they already own all the necessary infrastructure. Vendors that wish to recruit workers will have to tempt with a good benefit plan. The latter will ensure a wider area of coverage.

Discussion

This implementation would potentially open a new world of testing and introduce a service focused on simplification and automation like the other services offered by vendors. Companies will have an advantage of outsourcing testing to a vendor that charges based on usage, and the public will see an opportunity to gain some income by becoming an active worker.

4.6.3 Open-Source Cloud Native Application

The model described in this chapter may be developed into an Open-Source Cloud Native Application. What does this mean? How does this differ from the other described scenarios?

By breaking down the words that make up the title for this section, everything becomes clearer. First, *Open-Source* refers to the Open-Source Initiative and

Open Source Software, which is publicly available services published under the OSD-compliant license. This license grants all the rights to use, change and share the software in modified and unmodified form. In other terms, free. A *Cloud Native Application* refers to the fact that the application or service is packaged in a container and deployed as a *microservice*. The service itself would therefore be designed and developed with the intention of working in, or across, public clouds, private cloud and multicloud environments.

Having this model as a Open-Source Cloud Native Application would open many possibilities in terms of *Testing-as-a-Service*. The two previously mentioned scenarios may come to fruition and third-party vendors might build upon the model to create new exciting platforms that opens up a new world of possibilities for TaaS. We could see private companies run in-house testing using the model and third-party vendors gather a huge network of workers which companies can outsource testing to.

Communication model

This type of implementation is unique. As an Open-Source Cloud Native Application, this model should cover a wide area of uses. Therefore, for this implementation it should be the users decision as to which communication model that should be used. As an Open-Source project, it should therefore be possible to offer the three different communication models so that the decision lies with the user.

Workers benefits

As is the case of the communication model, so should the benefits for the workers be depending on the users needs. A company downloading the model may not desire a benefit program, while a vendor might want to implement a benefit program to the workers.

Discussion

The pros and cons is difficult to evaluate on this type of implementation seeing that an Open-Source Project is very versatile. But as a Open-Source project, the model would receive a lot of attention from the Open-Source community and gain a lot of traction. Open-Source Software seems to be favored due to their versatility. Just like Prometheus, Docker, Kubernetes and other Open-Source Cloud Native Applications, FEAST could also be among them to offer a better way to make automated testing.

4.7 Summary

This chapter has thoroughly presented the model and different scenarios in which the model can be used. Different attributes of the model and how they might be altered in accordance with a given situation has also been covered.

The model will, based on the implementation, be confined within a communication model. Three communication models that seem to be the most appropriate for the model has been covered and how these differentiate in terms of control, flexibility and anonymity from the perspective of the testers and the companies has been discussed.

Different terminologies associated with the model has been presented and explained a long with each stakeholder and the associated processes within the model in accordance with the *Test Cycle*. Each stakeholder has their own software architectural design. Which architectural pattern that goes into the design of each stakeholder has been drafted.

Lastly, this chapter covers different scenarios in which the model may be implemented. How the scenarios differ in terms of preferred communication model, the workers benefits and its pros and cons has been covered.

Chapter 5

Implementing the model

This chapter will focus on implementing and simulating the main premise that makes FEAST unique; distributing partitioned tests among testing entities. Throughout this chapter, decisions as to what to implement and what attributes to focus on is covered as well as justifying decisions made towards the implementation.

5.1 Goals of the implementation

Before going into details surrounding the approach for our simulation and its related decisions, it is important to reiterate what our goals are and what we want to measure. This is important to know in order to justify the decisions being made later and to be able to make appropriate decisions that lead us closer to our goal.

5.1.1 Main goal: Optimising completion times

The central goal of the implementation, and as previously mentioned, is to see if the scheme of having distributed workers to complete smaller tests is a more optimised approach as compared to traditional methods of testing software. For this to be accurately evaluated, simulations should be run focusing on this aspect. The simulation should be able to take in a full test that is passed through the two simulated ways of testing; using a dedicated testing infrastructure and our distributed model.

This goal will be very biased towards the metrics. With the goal of optimising completion times, the only metric that matters is, specifically, the time it takes for a test to be complete. As mentioned before, we consider completion time to start from the moment testing begins and stop as soon at testing is finished. However, data will be gathered on other constraints to evaluate the model based on utilisation of hardware. This data will later be presented and analysed.

Because of the focus on simulation, and comparing, completion times in the traditional way versus our distribution model, the simulation will not consider some of the aspects covered in the model. Aspects such as containerisation, container load time, transmitting container image and other communication aspects between the broker and workers will not be considered in the simulation. Instead, a focus on simulating two environments and compare the completion times and hardware utilisation in both environments will be prioritised.

5.1.2 Subgoal: Reducing costs of testing

The model will depend on workers to complete tests. These workers will then be paid for their time working. It is therefore central to investigate if our model is more cost-effective than having a dedicated testing infrastructure.

In order to explore and compare the two methods for testing in terms of costs, some data has to be gathered in terms of how much money is used in testing infrastructure and associated costs such as electricity usage. When it comes to our distributed model, we will not be able to gather any data on how much our model would cost a company because a model such as this are yet to exist. Therefore, approximations on this premise would be based on pricing from different cloud vendors that offer services that uses similar resources as FEAST.

The costs drafted will be most suited for an implementation based on the "*True*" *Testing-as-a-Service* implementation as described in subsection *4.6.2: Offering a "True" Testing-as-a-Service.* This implementation is the most suited due to the fact that it would act as a "pay-per-use" service in which the users have, figuratively, access to a infinite amount of resources but pay for the amount they use.

5.2 Implementation approaches

This section will go through the planned approaches for implementing the model with respect to the previously mentioned goals.

5.2.1 Simulating completion times

A simulation will be created that can simulate different scenarios that differentiate in the number of workers, testing infrastructure specifications and test intensity. The simulation should be able to output metadata regarding the simulation and how different the completion times were.

Python

The simulation will be created by using Python3, a general-purpose objectoriented programming language, to create a script that will be able to run different scenarios. The scenarios for collecting data on the performance of the job-distribution model will consist of two factors; amount of tests and workers. These different scenarios, for both the worker scenario and testing scenario, will be defined in input files which defines activity every minute for 24 hours. These Input files will be read by the Python3 script, which will turn each test and worker into their respective Python objects. This will allow us to create attributes for each object that keep track of data for that particular test of worker. We can then iterate over these instances and collect data that will allow us to collectively analyse how well our model performed.

Workers performance on test time

In order to accurately simulate the behaviour of the workers, the simulation will check the performance of each worker. Every object that represent a worker will have attributes that represent their computer performance, which influence how well they run tests. These attributes will be used by the simulation to simulate the expected time it takes for one testing entity to complete a test. For example, if a test is expected to take 1 minute, then a computer running on 80% performance would use 1 minutes and 12 seconds to complete the test.

The challenge of simulating parallel processing

In order to accurately simulate the behaviour of multiple entities working simultaneously, we are depending on processing in parallel. This can be achieved by using the Python library *multiprocessing*. This library will enable us to run processes simultaneously in the available processing cores in the computer in which the simulation is being run. Though this will accurately portray how the workers would work on each independent test, it is only limited to running simultaneous tasks based on the amount of processors in the computer where the simulation is run. This is commonly limited to 4 or 8 processors.

The library works by spawning new processes based on a function in the script, which can then be fed inputs and processed. Building on the multiprocessing library in order to accurately create a simulation that would use each computer core in order to emulate workers would create a more accurate model but would create limits that would severely hurt our analysis. Using a different approach that would allow us to take advantage of hundreds of workers significantly increases our measurements to analyse the effectiveness of the model.

Therefore, the decision was made to imitate parallel processing by setting the time each test would take and have a iteration of all the minutes of a day. Whenever the iteration hits a time in which a test is submitted, that test is then distributed to the first worker in line followed by the next scheduled test. If a test is distributed at 21:00, and is expected to be finished in 2 minutes and we iterate by 1 minute at a time, then that test would be finished two iterations later, or 21:02. This approach would not limit the number of workers and though it is simplified, it would serve as a huge benefit in order to simulate scenarios in a huge scale.

The resulting iteration will therefore contain phases that the simulation goes through during one timestamp. These steps can be seen illustrated in Figure 5.1. First, (1) one iteration, or a passing minute, is processed by every worker. Should one worker have a test assigned to it, they will at this point have worked one minute of the test time. Our simulation will then (2) check for tests and workers scheduled at that point in the iteration and add these to an active list. Should some workers at this point be finished with a test, these are (3) checked for availability and added to a list of available workers. The list of tests are lastly (4) distributed among the active workers, starting with the first worker in line.

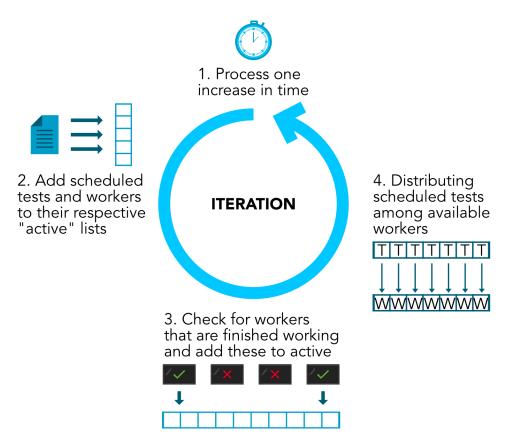


Figure 5.1: Illustration of the iteration of timestamps

5.2.2 Measuring costs of testing

The approximations should be able to justify the price that a company would pay in order to use an implementation of our model based on TTaaS. By using the approximation, we should be able to assume if a an implementation of FEAST as TTaaS would be more cost efficient than a local testing infrastructure.

Cost of Testing Infrastructure

In order to create an picture on how much a testing infrastructure would cost, data have to be gathered that accurately portray the costs associated with purchasing and maintaining a testing infrastructure. We should have data estimates that includes factors such as the one-time purchase of the infrastructure, power usage and maintenance.

Cost of the TTaaS

Data regarding pricing of already existing cloud services can be gathered and evaluated in order to gain a good starting point towards the fundamental pricing. As the other cloud services, billing should be based on the used resources used by the company performing the testing. We are to assume that the TTaaS would always be available as a *on-demand* service that company's can use whenever they please.

5.3 Simulation

This section will go through the creation of a simulation that would help to measure the performance of the model. The simulation would act as an environment that would take in two inputs and run a simulated situation based on the data from the two input files. Throughout this section, one can expect to find detailed deconstruction of the components and the exploration that led to the final simulation. Code snippets used throughout this section is taken from the source code for the simulation, which can be viewed in its entirety in the Appendix.

5.3.1 Creating the input files

Before creating the content of each file, we will have to assume that these files will contain a lot of data in a huge number of rows if they are to represent tests submitted and worker activity for each hour and minute of one day. For the simplest and most efficient readability by our simulation, this information will have to be stored as structured data. A comma-separated values file (*.csv*) will easily allow us to create a header to contain the information for each column followed by rows of data separated by a comma. These files will be easily read by Python.

Each file should contain relevant information regarding each *instance* of either a worker or a test. First step would therefore be to decide the different attributes each instance should possess.

Instead of relying on a random generator that would randomly decide whenever a test is supposed to be scheduled, we can base this decision on real world data. A huge dataset containing 2.3GB of GitHub commit messages by Dave (2021) will create a good understanding that we can use to base the decision as to how often, during a day, tests should be submitted and scheduled for testing.

```
1 # Dictionary for counting number of tests per timestamp
2 time_count = {}
_3 # I/O and For-loop used to extract data from dataset containing data on
      GitHub commits
4 with open("full.csv", "r") as file:
      for row in file.readlines():
5
          # ReGex used to find the timestamp
6
          match = re.search("[0-9][0-9]+:[0-9][0-9]:[0-9][0-9]",row)
7
          if match:
8
              print(match.group(0))
9
              entry = match.group(0)[:5]
10
              # if-statement used to count how often a certain timestamp is
11
      used
              if entry in time_count:
12
                  print("Time found in dictionary, incrementing")
13
                  time_count[entry] += 1
14
              else:
15
                  print("Time not found, adding entry")
16
                  time_count[entry] = 1
17
```

Listing 5.1: Code used to extract data from the GitHub commit messages dataset

Using the code in listing 5.1, we will get a dictionary containing the timestamps of our dataset and the count where each one of these timestamps occur. In Figure 5.2 the frequency of commit messages over time is illustrated.

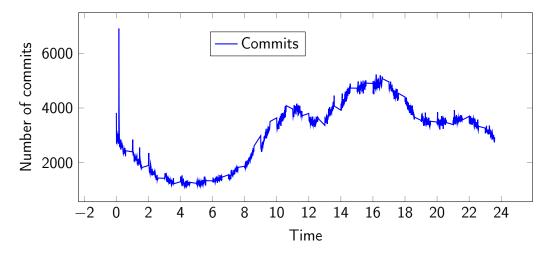


Figure 5.2: GitHub commit messages over time

By using the values in the dictionary as values and weights, our test randomisation is able to randomly select timestamps in which tests will be submitted but based on the frequency from the dataset. This way, we move away from "full random" and move closer to the territory of probability.

It should be noted that the extreme values of 6911 at timestamp 00.16 and 4115 at 00.17 was taken out of the dictionary because of its peculiar presence. It was uncertain if this was an anomaly generated by the RegEx code or if it were accurate data. The data was taken out as a precaution in case this would dramatically effect measurements from the scenarios.

Worker scenarios

The worker instances will vary in how long they are available to work, how much processing power they will have to work with and when they joined the testing. The following attributes are used to characterise a certain worker instance:

- WorkerID (Range: *W0001-W9999*, e.g. *W0026*): This attribute is used to identify each instance of the worker object. The identifier is given to an instance of the test object to refer to a certain worker and can be used further to retrieve specifications for a worker.
- StartTimeAvailable (Range: 00.00-23.59, e.g. 13.38): Contains the timestamp for when a particular worker will become available to work.
- **PerformanceRating** (Range: 0.60-2.00, e.g. 0.79): Percentage of performance expressed in decimals. This attribute will determine how efficient that worker is at performing tests. This percentage is used to affect the test time within the test instance that is given to the worker.
- AvailabilityTime (Range: 20.00-180.00, e.g. 105.35): Explains how long a worker is available to work as expressed in minutes. Whenever this number hits zero, the worker will no longer be available to perform any work.

By creating a script to generate the input files for the workers, some parameters can easily be tweaked in order to create completely different scenarios.

^{1 #} Worker counter used for creating worker ID's
2 worker_increment = 1

```
4 # Variable that consists of the number of workers that the taskfile should
       start of with
5 initial_workers = 4
6
7 # iterates over the range of the integer set in the initial_workers and
     uses enumerate
s for _ in range(0, initial_workers):
     # Creates an entry in the dictionary that consists of the WorkID as
     key: "W0001" and values for StartTimeAvailable, PerformanceRating and
     AvailabilityTime
      worker_initial_dictionary["W"+str(worker_increment).zfill(4)] = {
10
                       "StartTimeAvailable": '0.00',
11
                      "PerformanceRating":round(random.uniform(0.6,2),2),
12
                      "AvailabilityTime":round(random.uniform(20,180),2),
13
                       }
14
      # incrementing with 1
15
     worker_increment += 1
16
```

Listing 5.2: Code snippet of generate_workers.py

The code presented in listing 5.2 creates the *initial* workers, which represents those that will be available at the first minute in our simulation. These will be scheduled to start work at the timestamp 00.00 and are those available right when the simulation start.

By modifying the iteration to include different values in *StartTimeAvailable*, we can use Python to generate a *.csv* file that contains our workers scenario. A sample from one of the worker scenarios can be seen in listing 5.3.

```
1 WorkerID,StartTimeAvailable,PerformanceRating,AvailabilityTime
2 W0001,0.00,0.79,45.47
3 W0002,0.02,1.99,159.75
4 W0003,0.02,1.26,36.69
5 W0004,0.05,0.9,26.19
6 W0005,0.07,0.63,132.17
7 W0006,0.11,1.2,28.09
8 W0007,0.33,1.26,32.08
9 W0008,0.43,1.36,79.93
10 W0009,0.44,1.86,53.57
11 W0010,0.45,1.68,138.4
12 W0011,0.49,0.84,45.91
13 W0012,0.50,0.8,42.22
14 W0013,0.52,1.94,74.02
15 W0014,0.55,1.14,140.44
```

Listing 5.3: Sample from a workers scenario

Testing infrastructure scenarios

The testing infrastructure scenarios will simulate a dedicated server that is reserved solely for testing. We will base the number of active workers to the number of cores in a typical server, which is 8, 16, 32 and 64. The number of cores will therefore act as *workers*, since we will assume that the server will run multiprocessing for completing the tests. The main difference between the input file for infrastructure scenarios versus worker scenarios is that it will always be available and have a static performance rating of 1.

Test scenarios

Each instance of a test will not include the same amount of attributes as one worker instance. The only data that will be used to create variation among the different test instances will be when the test is submitted and how long it is estimated to take. The following list of attributes are used to characterise the different test instances:

- **TestID** (Range: *T0001-T9999+*, e.g. *T12125*): Used to uniquely identify an instance of a test. This attribute will be given to a worker to keep track of the completed tests and what test a worker is currently working on.
- ScheduledTime (Range: 00.00-23.59, e.g. 04.54): This attribute will tell the simulation the time this test was submitted and when this test should be put in line to be tested.
- **TestTime** (Range: 1.00-2.59, e.g. 2.39): As expressed in minutes, this attribute contains the time that particular test will take. When this test is assigned to a worker, the worker's performance will be used to change how long the test takes.

Creating the test input files will be similar to generating the worker scenarios. The only difference will be that instead of fully randomising when the tests are scheduled to run, we will base this decision on the values and weights dictionary created in Listing 5.1. In Listing 5.4, two lists are created that contains the values (or population) and weights respectively. This will then be used to decide the timestamp which a test will be generated for.

^{1 #} Creates a list based on the times from the dictionary of sorted items
2 population = [x for x,y in sorted_items]

Listing 5.4: Code snippet of *generate* tests.py

By creating the different attributes as mentioned previously and writing these to a *.csv* file, the resulting file will contain a huge number of rows that contain different test. These tests have their own timestamps to when they will be scheduled to run, based on the dataset containing GitHub commit messages, and attributes defining their characteristics. A sample of the resulting file can be seen in listing 5.5.

1 TestID,ScheduledTime,TestTime
² T0001,00.00,1.42
3 T0002,00.00,1.12
4 T0003,00.00,2.18
5 T0004,00.00,1.49
6 T0005,00.00,1.50
7 T0006,00.00,1.36
8
9
10
11 T12136,23.59,2.21
12 T12137,23.59,1.27
13 T12138,23.59,2.37
14 T12139,23.59,2.32
15 T12140,23.59,1.39
16 T12141,23.59,2.45

Listing 5.5: Sample for a test scenario file

5.3.2 Worker and test classes

Our simulation will benefit from converting each worker and every test into objects within the Python script in order to gather analysis of the performance. Each instance will be created from a class that acts as a blueprint for which attributes, methods and calculations that belong to that particular object will be drafted. When the simulation is over, we can iterate over the worker or test instances and collect data by referring to a particular attribute we are interested in using in our analysis.

Worker class

The class that each worker, including the infrastructure cores, object will be created from will consist of attributes and calculations that are specific for the workers only. The only exception is the attribute containing processing power, which will affect the test time of the test distributed to the worker. From the worker scenario input file, each specified attribute in that file is given to each worker object. The following attributes are instance variables that holds data we are interested to collect and analyse:

- *self*.**CompletedTests** (Type: *array*, e.g. ["T0030", "T0060", "T0140"]): This array will keep track on the *TestID* for every test completed by that particular worker. This is used later to measure performance by analysing how many tests every worker completed during their working period.
- *self*.**WorkingIntervals** (Type: *array*, e.g. [2,3,2,1]): Used to collect the times used by that worker working. One integer represents the number of iterations in the for-loop that went by when that worker was busy working on one particular test.
- *self*.**WaitingIntervals** (Type: *array*, e.g. [0,1,1,0]): Contrary to the *WorkingIntervals* array, this array will store the intervals from when the worker was finished with a test (or more precisely, became available to work) to when that worker was assigned a new test to run.

All of the worker objects are added to a class attribute array at the time of creation called *instances*. This class attribute can be referred to by calling the class name and the attribute name (Worker.instances). This array directly refers to the instances, and we can therefore iterate over this array to call class methods, gather data or check specific values.

The worker class have one method called *self.assigned_test(self,test)*. This method is called whenever a worker is assigned a test to run. A part of this method is calculating the time for the assigned worker to complete the given test. This is calculated in this method and then changes the time stored in the test object by overwriting the original test time with the calculation. This method

will also store the waiting interval for this worker by using the a checkpoint flag to keep track of time.

```
1 def assigned_test(self, test):
      self.AssignedWork = True
2
      self.AssignedTest = test_overview_dictionary[test].TestID
3
4
      original_testtime = test_overview_dictionary[test].TestTime
      number_after_performance = round((float(original_testtime)*60) / float
\mathbf{5}
      (self.PerformanceRating))
      test_overview_dictionary[test].TestTime = float(f"{
6
      number_after_performance//60}. {number_after_performance%60}")
      # append the time differenece since we finished the last test to "
7
      WaitingIntervals"
      self.WaitingIntervals.append(self.time_diff(time,self.checkpoint))
8
      # set new checkpoint
9
      self.checkpoint = time
10
```

Listing 5.6: Code block for the assigned test() class method

An other method for this class is the *self*.**process**(*self*) method which is called for every iteration in the for-loop. This method will subtract 1 minute from the availability time of the active workers. Depending on the status of the worker, if a worker is assigned a test to work on, one minute is subtracted from the test time as well. Should the tester be available for work and will not receive a test during that iteration, then the waiting time interval for that worker will increase by 1. Should a test have more than 0 minutes left but less than 1 minute (e.g. 0.55), then this value is replaced with 1 whole minute to keep the worker busy for an additional iteration. The method can be seen in listing 5.7.

```
1 def process(self):
      self.AvailabilityTime -= 1
2
3
      if self.AvailabilityTime > 0:
          if self.AssignedTest:
4
              test_overview_dictionary[self.AssignedTest].TestTime -= 1
\mathbf{5}
              if test_overview_dictionary[self.AssignedTest].TestTime < 1</pre>
6
      and test_overview_dictionary[self.AssignedTest].TestTime != 0:
                   test_overview_dictionary[self.AssignedTest].TestTime = 1
7
              elif test_overview_dictionary[self.AssignedTest].TestTime ==
8
      0:
                   self.CompletedTests.append(test_overview_dictionary[self.
9
      AssignedTest].TestID)
                   self.AssignedTest = None
10
11
                   self.WorkingIntervals.append(self.time_diff(time,self.
      checkpoint))
```

12	<pre>self.checkpoint = time</pre>
13	else:
14	<pre>self.AssignedWork = False</pre>
15	<pre>self.TotalWaitTime += 1</pre>
16	else:
17	finished_workers.append(self)
18	<pre>self.JoinedTesting = False</pre>

Listing 5.7: Code block for the process() class method

A "dunder" (double underscore) method is defined within the worker class called __str__(self). This method will be called whenever the print() statement is used to refer directly to a object that derives from the worker class. Within this method, a long string is created and returned which will be printed as output in the terminal. A section of the dunder method can be seen in listing 5.8.

```
1 def __str__(self):
     return f"{self.WorkerID}: \n\tStarted working: {self.
2
     StartTimeAvailable}\
      \n\tPerformance: {round(self.PerformanceRating*100)}%\
3
      \n\tRemaining availability: {self.AvailabilityTime}\
^{4}
\mathbf{5}
      . . .
      \n\tTest statistics:\
6
      \n\tTest distribution min: {str(np.min(self.WorkingIntervals)) if bool
7
      (self.WorkingIntervals) else 'N/A' } \
8
      \n\tWait interval statistics:\
9
      \n\tWait distribution min: {str(np.min(self.WaitingIntervals)) if bool
10
      (self.WaitingIntervals) else 'N/A'}\
11
      \n\tWorker utilization: {self.Utilization}%"
12
```

Listing 5.8: Code snippet from the dunder method for the Worker class

This method will print out extensive data about each worker in order to measure efficiency of the implementation, and we want to take a look into the following:

- Minimum values: We want to be able to store the shortest times a worker had to work and what the minimum time a worker had to wait before being given a test.
- Median values: The mean value of waiting and working times in the workers will give us some indication on the longevity of the majority of the tests distributed among the workers.

- Mean values: Gives us the average time workers had to wait and work.
- **75** percentile: Collecting data on the 75 percentile will give us the estimates on at what point three quarters of the data lies and ultimately determine how many high values affect the analysis.
- Maximum values: Maximum values will tell us how intense the longest tests any worker were given.

An example of the output from the dunder method that shows the different calculations can be seen in listing 5.9.

```
1 W0526:
2 Started working: 23.51
3
   Performance: 60%
4 Remaining availability: 48.89
5 JoinedTesting: Yes
6 Available to work at current time (23.59): No
7 Assigned Test: T11205
   Completed Tests: T11155 T11194
8
9
   Completed Tests count: 2
   Waiting Time List: [0, 1, 1]
10
11 Test statistics:
12 Test distribution min: 2
13 Test distribution median: 3.0
14 Test distribution mean: 3.0
<sup>15</sup> Test distribution 75%: 3.5
   Test distribution max: 4
16
   Wait interval statistics:
17
   Wait distribution min: 0
18
   Wait distribution median: 1.0
19
   20
   Wait distribution 75%: 1.0
21
22 Wait distribution max: 1
23 Total wait time: 2
   Worker utilization: 75%
24
```

Listing 5.9: Example output from the dunder method of a worker instance

Test class

The test class will have very few attributes and methods. An object deriving from this class will just act as an entity that occupies a certain time for the worker it is given to. The attributes defined in the input files for test scenarios will be given to the test instance along with the following attributes:

- *self*.**AssignedWorker** (Type: *string*, e.g. "W0035"): This variable will contain the "WorkerID" for the worker that got assigned that test.
- self.WaitTime (Type: integer, e.g. 3): This integer will describe how many iterations, or minutes, that passed before the test got assigned a worker.

Each instance of a test created from the test class will be added to a class attribute array, the same method as described regarding the workers. This will allow us to iterate through all of the test instances.

The test class have one main method called *self*.assigned worker(*self*, *worker*), which works similarly as the worker class method with almost the same name. This method is called whenever the test is given to a worker and will store the amount of iterations that passed before it got assigned to a worker. This code block can be seen in listing 5.10.

```
1 def assigned_worker(self, worker):
2   self.AssignedWorker = worker
3   if float(time) == float(self.ScheduledTime):
4       self.WaitTime = 0
5   else:
6       self.WaitTime = self.time_diff(time,self.ScheduledTime)
```

Listing 5.10: Code block that is called whenever a test is assigned a worker

The dunder method defined in the test class is very limited but comes in handy to see data regarding the wait time for a test. This dunder method can be seen in listing 5.11, while an example of the output in terminal are shown in listing 5.12.

```
1 def __str__(self):
2    return f"{self.TestID}: \n\tScheduled for testing at: {self.
2    ScheduledTime}\
3    \n\tTime it takes for testing: {self.TestTime}\
4    \n\tAssigned to worker: {self.AssignedWorker}\
5    \n\tThis test waited in queue for: {self.WaitTime} iterations"
```

Listing 5.11: Dunder method for the test class

```
1 T0014:
```

² Scheduled for testing at: 00.01

```
3 Time it takes for testing: 0
4 Assigned to worker: W0004
5 This test waited in queue for: 7 iterations
6 T0015:
7 Scheduled for testing at: 00.01
8 Time it takes for testing: 0
9 Assigned to worker: W0001
10 This test waited in queue for: 10 iterations
```

Listing 5.12: Example output of the dunder method for a test instance

Creating LaTeX graphs

In order to visualise the results and performance of the simulation, a script called graph_gen.py was created that contain two line graphs that can be imported to the simulation script. This function takes a list of plots as input (e.g. [[(01:00,61),(02:00,86)],[(01:00,31),(02:00,13)],[(01:00,24]),(02:00,51)]] and output the resulting plot that can be used in LaTeX. The resulting graphs can be seen used in the analysis chapter.

5.3.3 Summary

This section has presented the implementation of the simulation of our jobdistribution model and given an accurate description of its different components that work together to simulate a scenario based on two input files. These input files were created using two generator scripts which were also covered. The focus of this subsection has been to address how data is created, stored and made accessible for analysis. The visualisation and analysis of these data can be viewed in the Analysis chapter.

5.4 A Discussion on Comparing Costs

This section will discuss the costs associated with purchasing a dedicated infrastructure compared to the approximate costs of an implementation of the FEAST model as a TTaaS. The aim of this section is not to provide a price estimate for the costs of such an implementation, but rather explore the costs of infrastructure and how much currently available cloud services costs.

5.4.1 Cost of a dedicated testing infrastructure

A dedicated testing infrastructure would mean that the company that wants to do testing in-house will have to reserve a server that they posses and assign this server for testing purposes. This server will have to be available all the time and be ready to receive tests to run at any time.

If we are to assume that a new start-up company wants to do testing in-house, but does not have any servers at the moment, the first step would be to purchase and set up a server at their facility. According to RDI (2021), the cost of a server lies in the range of \$3,000-\$5000 and \$660-\$825 (4-5 hours of work) for setting up the server. These figures a bit higher than the estimates by Lahn (2021), which puts the cost of purchasing a server to be between \$1,000-\$3000 and \$400-\$750 (for the same 4-5 hours of work) for setting up the server. They further mention that monthly costs of power usages would come at a price of \$30,24 for server in a lower price range and \$60,24 for those in the higher range (not including cooling and internet expenses). RDI further estimates that 1 to 10 hours are required monthly for maintaining the server with a average fee of \$150 per hour. The following table presents the discoveries found associated with costs of a testing infrastructure:

Cost	Price	Frequency
Server purchase	\$1,000-\$5,000	One-time
Power usage	\$30,24/\$60,24	Monthly
Setup	\$400-\$825	One-time
Maintenance	\$150-\$1500	Monthly

Table 5.1: Overview of the costs associated with a dedicated testing infrastructure

Based on the table above we can create three classes of infrastructure where we can also assume that maintenance costs of all three servers would be the same (\$150-\$1500 monthly):

- High-tier infrastructure: Best performance but also the most costly. Server purchase: *\$5000.* Power usage: *\$60,24.* Setup: *\$800.*
- Middle-tier infrastructure: Medium performance, well-balanced. Server purchase: \$3000. Power usage: \$30,24. Setup: \$550.

• Low-tier infrastructure: Lowest performance but the cheapest. Server purchase: \$1000. Power usage: \$30,24. Setup: \$400.

5.4.2 Calculating costs for TTaaS

By taking a look at the service offered by cloud vendors, we can estimate approximately how much an implementation of FEAST as a TTaaS would cost. Specifically, explore the costs of services that share similar traits to how TTaaS would behave.

We can look for similarities between our TTaaS implementation and other cloud services by looking at which resources are being used. Computational power would be the main computer resource that our TTaaS would use. Therefore, we can look at how much cloud vendors charge for renting server space with a high vCPU (Virtual Central Processing Unit). In table 5.2, the prices from some of the most popular cloud vendors are presented. All prices are in US Dollars based on server locations in London, Europe West or UK South and are on-demand.

Vendor	Instance (Cores)	Price \$/hr	Source
Google	c2-standard-16	1.076048	Google (2021)
	(16)		
Google	c2-standard-30	2.01759	Google (2021)
	(30)		
Google	c2-standard-60	4.03518	Google (2021)
	(60)		
AWS	c6g.4xlarge (16)	0.6464	Amazon (2021)
AWS	c6g.8xlarge (32)	1.2928	Amazon (2021)
AWS	c6g.metal (64)	2.5856	Amazon (2021)
Microsoft Azure	F16s v2 (16)	0.808	Microsoft (2021)
Microsoft Azure	F32s v2 (32)	1.616	Microsoft (2021)
Microsoft Azure	F64s v2 (64)	3.232	Microsoft (2021)

Table 5.2: Overview of costs for services from cloud vendor that focus on vCPU

The discovered prices ranges from 0.6464/hr for a 16-core vCPU to 4.03518/hr for a 60-core vCPU. Estimated costs for an implementation of TTaaS would therefore have a starting price in this range, without considering the costs associated with benefits given to workers. The discoveries can be summarised:

- For a 16-core vCPU, the price ranges from \$0.6464 to \$1.076048 per hour
- For a 32-core vCPU, the price ranges from \$1.2928 to \$2.01759 per hour
- For a 64-core vCPU, the price ranges from \$2.5856 to \$4.03518 per hour

5.4.3 Summary

Discoveries in this section suggests that the expenses of reserving a dedicated server for testing exceeds the costs associated with using a TTaaS implementation. A trend among cloud services is offering a very low price per hour of usage. This, combined with paying for only the resources that are used, makes this option very reasonable compared to a dedicated testing infrastructure.

Chapter 6

Analysis

This chapter will present and analyse the gathered data from the simulation as presented in the implementation chapter.

6.1 Simulation

In this section, data from the simulation is presented which will focus on the relationship between amount of workers, tests submitted and the tests remaining in different scenarios.

6.1.1 Scenario: Intense testing to a moderate amount of workers

This scenario will provide tests in a intense continuous flow where the amount of tests submitted per minute is based on the statistics gathered from the GitHub commit message dataset. There will always be at least one test submitted per minute.

The worker scenario will initially start with 20 available workers and will over time spawn workers in a moderate amount varying from 0 workers to 2 workers per minute. Within working hours (10-18), there is a chance that the frequency in which workers spawns is increased with a chance of spawning up to 4 workers

per minute.

The following files were used for this scenario:

- Test scenario: continuous_intense.csv. This input file contains a total of 12,141 tests which has a default test time between 1.00 and 2.59 minutes. The most intense period of this scenario is between 16.00 and 16.30 where up to 20 tests are submitted per minute.
- Worker scenario: 20initial_increase_in_worktimes.csv. This scenario will simulate a steady flow of workers joining testing with an increase around the working hours where up 4 workers might join per minute.

Figure 6.1 displays the metrics plotted over time. At around 10.00, we clearly see the relation between remaining tests and the available workers. When the increase of remaining tests happens at around 09.00, the workers that joins testing at 10.00 will quickly be assigned work and reduce this queue dramatically even though this time has the most tests submitted.

At the end of the day, when the frequency of workers are reduced, the remaining tests will skyrocket. The remaining workers will then always have work to do, and the wait time for each test is increased.

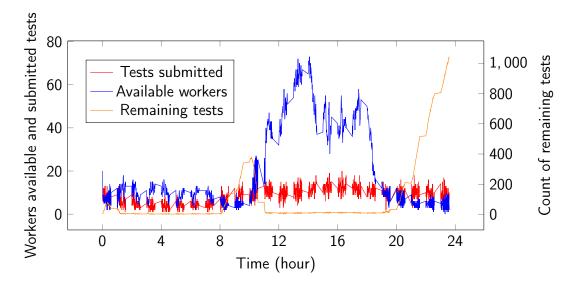


Figure 6.1: Line graph of metrics for the scenario: Intense testing to a moderate amount of workers

Workers in this scenario will have a very good utilisation percentage where the majority of workers lay within 30% to 40% and 38 workers had a utilisation percentage of 67%. This is as expected due to the huge amounts of testing. If we put Figure 6.1 and Figure 6.2 next to each other, we can assume that during the busiest hours, utilisation of each worker was drastically decreased. It was at this moment that there was not enough tests to supply the demand for work by the amount of workers. Interestingly, this shifts at around the 20.00 mark when the demand for workers skyrocket but there is not enough workers to meet this demand.

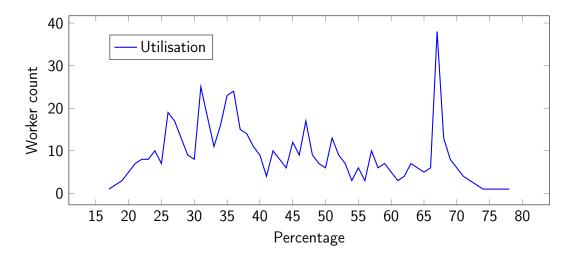


Figure 6.2: Percentages of utilisation in the scenario: Intense testing to a moderate amount of workers

In this scenario, there is a huge span of varying utilisation percentages for the workers. However, there are no cases of "starvation" from any worker. The FEAST is able to prevent starvation and distribute test pieces to the available workers. The higher percentages would correspond well with the balanced data at the start of the day and the lower worker availability at the end, and the lower utilisation percentages corresponds with the overwhelmingly amount of workers. Although there are some with fewer utilisation percentages, FEAST seems to be able to distribute tests among workers and prevent starvation.

The line graph in Figure 6.3 further confirms the suspicion from Figure 6.1, and we see clearly that the wait time starts to increase at 08.00 and continue to rise until 10.00. At this point, more workers join the testing and the wait time is dramatically reduced before increasing again at 20.00.

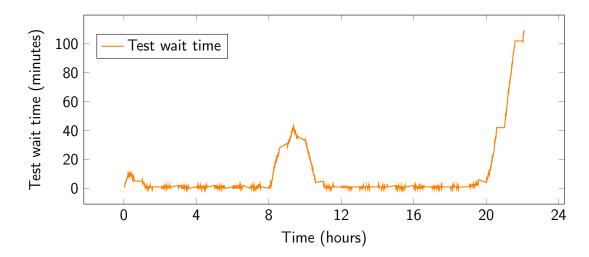


Figure 6.3: Average wait time for tests submitted per time in the scenario: Intense testing to a moderate amount of workers

6.1.2 Scenario: High demand for workers

The test input of this scenario will provide a continuous and moderate amount of tests throughout the simulation.

Workers will be sparse throughout the duration of the simulation, which results in a high demand for workers to complete tests but not enough workers to complete them. This scenario is used to visualise the extreme situation where there will be a huge demand for workers and that the tests are continuing being submitted in a fast pace and how this will affect test times. The following files were used for this scenario:

- Test scenario: continuous_moderate.csv. This input file contains a total of 2002 tests which has a default test time between 1.00 and 2.59 minutes. As with the previously used test scenario, the most intense period of this scenario is between 16.00 and 16.30 where up to 20 tests are submitted per minute.
- Worker scenario: *Oinitial_reduced_in_activity.csv*. This scenario will simulate a reduced presence of workers scattered through the day.

In Figure 6.4, the metrics collected from this scenario are shown which shows the enormous increase of remaining tests. If the simulation would continue further

than the 24 hours we would see the number getting higher and higher suggesting that it would not be possible to recover from the disastrously long waiting times each test have to wait before being picked by a worker. In this situation, a huge amount of workers would have to join the testing in order to bring the number of remaining tests down.

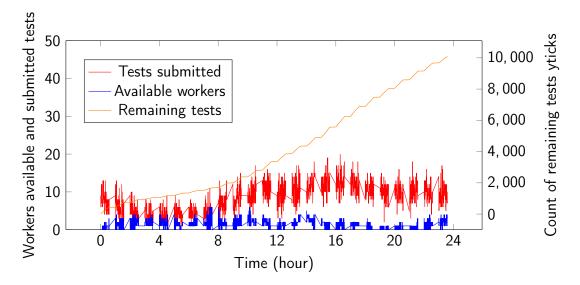


Figure 6.4: Line graph of metrics for the scenario: High demand for workers

The few workers that are working in this scenario will always be able to work, and a result of this will have a higher percentage in utilisation, as shown in Figure 6.5.

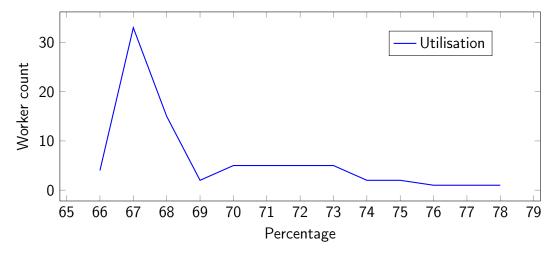


Figure 6.5: Percentages of utilisation in the scenario: High demand for workers

The effect this scenario has on the waiting time for each test is shown in Figure 6.6. After the first 6 hours, a newly submitted test would be expected to wait over 1000 minutes, or 16 hours, before being distributed to a worker.

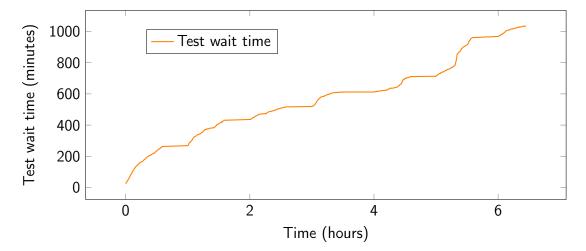


Figure 6.6: Average wait time for tests submitted per time in the scenario: High demand for workers

6.1.3 Scenario: Overload of workers

As a contrast to the previous scenario, this scenario emulates a case where workers are plenty and tests are rare.

Test input for this scenario will consist of tests being submitted at different times during the simulation instead of a continuous flow. This will simulate an implementation that rarely receives tests to distribute.

The worker scenario will simulate a steady activity of workers joining the testing and being able to receive tests to complete.

The following files were used for this scenario:

- Test scenario: varying_low.csv. A input file that contains tests being distributed sparsely during the simulation. The file contains 718 tests with a test time varying from 1.00 to 2.59.
- Worker scenario: *Oinitial_steady_activity.csv*. This worker scenario will create 327 workers during the simulation. These workers will steadily be

joining the testing throughout the duration of the simulation.

The data from the simulation running this scenario can be seen in Figure 6.7. We can observe that the amount of available workers are holding steady above 20 for most of the simulation. An observation worth mentioning is the sudden spike in remaining tests at the start of the simulation. This would suggest that, because of no active workers at 00.00, some tests were submitted at a time where no workers were available. Even so, this huge spike translates into 5 tests that are waiting, while in the rest of the simulation this number is steadily at a maximum value of 2.

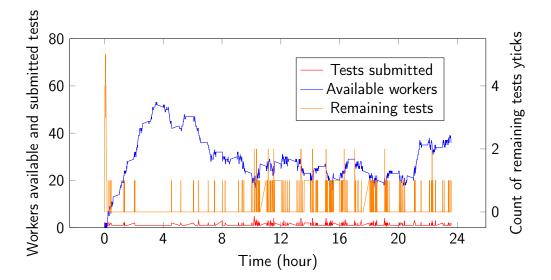


Figure 6.7: Line diagram of metrics for the scenario: Overload of workers

This scenario in which workers are plenty while the tests themselves are a rarity translates to low utilisation for the workers. At most, a worker will be able to utilise 13% of their resources.

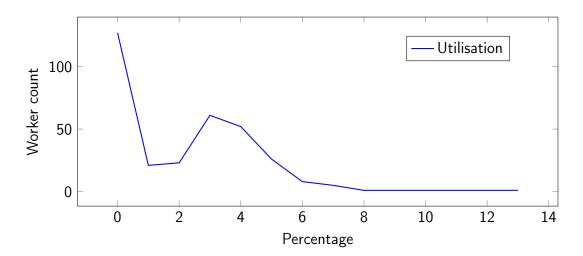


Figure 6.8: Percentages of utilisation in the scenario: Overload of workers

The spike in the beginning of the simulation as mentioned before would translate into 9 minutes of waiting time. This is a long time to wait, but reflects how important it is for workers to be available and plenty. At the time of the spike (about 00.02 to 00.16) there was a total of 9 tests submitted to only two available workers. This would explain the spike and further emphasises the importance a bigger active worker pool. In this scenario, starvation occurs in which no workers will be able to get any work done and therefore no payments.

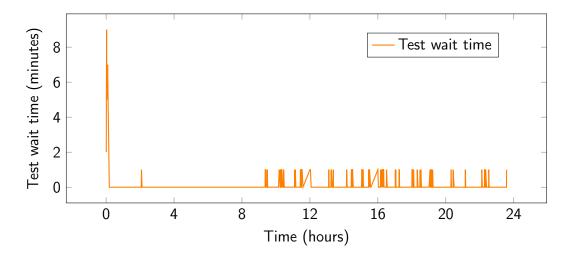


Figure 6.9: Average wait time for tests submitted per time in the scenario: Overload of workers

6.1.4 Scenario: Continuous submitted tests combined with local infrastructure

The infrastructure inputs are based on the number of cores in a typical server, which ranges from 8 cores to 64. What is interesting to see is how these different infrastructures would perform on a test file that would more resemble testing within a company, varying_low.csv.

The following four line graphs (Figure 6.10, Figure 6.11, Figure 6.12 and Figure 6.13) presents the results from the simulation with the use of infrastructure with the different cores (8, 16, 32 and 64).

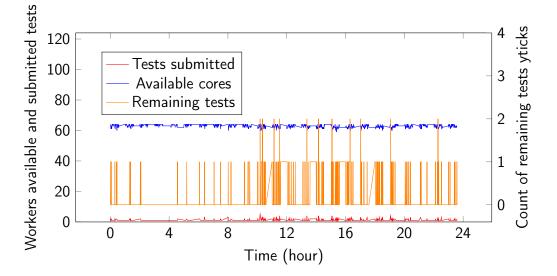


Figure 6.10: Line graph of metrics for the scenario: Continuous submitted tests for infrastructure for a 64-core infrastructure

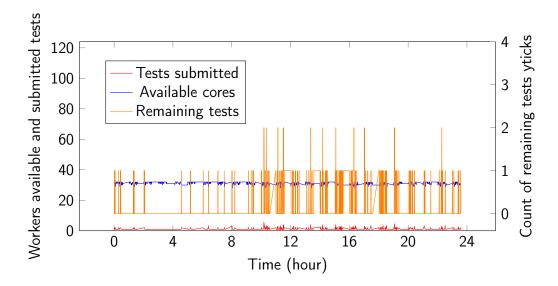


Figure 6.11: Line graph of metrics for the scenario: Continuous submitted tests for infrastructure for a 32-core infrastructure

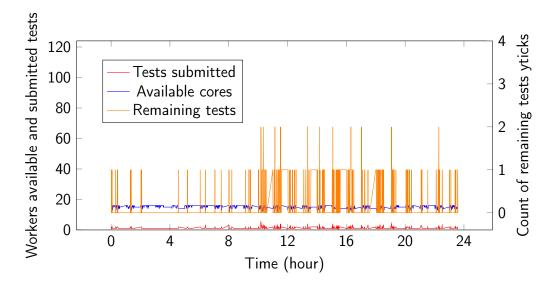


Figure 6.12: Line graph of metrics for the scenario: Continuous submitted tests for infrastructure for a 16-core infrastructure

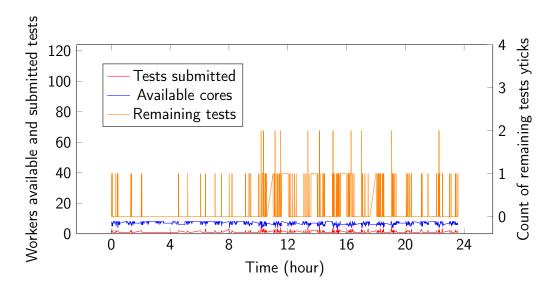


Figure 6.13: Line graph of metrics for the scenario: Continuous submitted tests for infrastructure for a 8-core infrastructure

If we take a look at the utilisation percentages as shown in table 6.1, the utilisation in a 8-core infrastructure is very low per core and the more cores the more wasteful becomes the infrastructure. When a 64-core infrastructure has a optimisation percentage of 2% for each core, then these cores will be dormant 98% of the time. Although a 64-core infrastructure will perform quite well in times for demanding testing, it will be wasteful in the long run.

Cores / Utilisa- tion	14%	8%	7%	4%	3%	2%
8	8	0	0	0	0	0
16	0	1	15	0	0	0
32	0	0	0	18	14	0
64	0	0	0	0	0	64

Table 6.1: Overview of the utilisation percentages of each core in the different infrastructures

6.1.5 Scenario: Moderate testing to infrastructure and workers

This section will run the same test inputs on a worker scenario and on each infrastructure and compare how they differ in terms of utilisation.

The test inputs that will be run for both solutions simulates a continuous moderate flow of tests where there will always be one test submitted each minute.

The worker scenario will simulate 5 initially active workers and keep a steady activity with an increase during working hours.

The following files were used for this scenario:

- **Test scenario**: *continuous_moderate.csv*. This test scenario simulates a steady flow of submitted tests. A total of 2002 tests will be submitted through the simulation.
- Infrastructure scenario: infra_8.csv, infra_16.csv, infra_32.csv and infra_64.csv. These are the same infrastructure scenarios that are used in section 6.1.4: Scenario: Continuous submitted tests combined with local infrastructure.
- Worker scenario: *5initial_increase_in_worktimes.csv*. This worker scenario will simulate 5 initially active workers and simulate a steady activity. During working hours up to 4 workers may join testing one minute.

As seen in Figure 6.14, this scenario shows the distributed utilisation percentages with FEAST while local infrastructures is very steady. This corresponds well with the behaviour of FEAST where there are varying amount of workers available at any given time, resulting in fluctuations. The infrastructures, which is available all the time with the same amount of cores, will have a steady utilisation percentage.

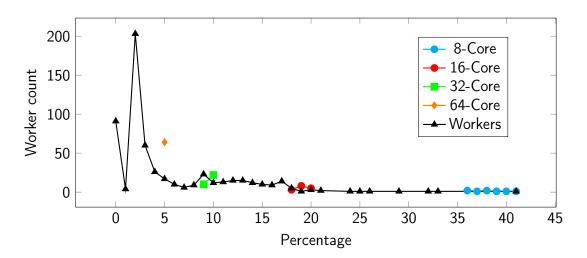


Figure 6.14: Percentages of utilisation in the scenario: Moderate testing to infrastructure and workers

Chapter 7

Discussion

The aim of this thesis has been to create a model that would optimise automated testing by stepping away from traditional methods, and the associated approaches to optimise the traditional ways of software testing, by including agents in a jobdistribution scheme. This chapter will discuss and evaluate various aspects and decisions done in the thesis associated with the implementation and the design of the model. The discussion will lead up to the conclusion, which will act as a final evaluation for the complete work performed in this project.

Throughout the chapter, suggestions for future work within this field will be presented and addressed.

7.1 Analysis results

The analysis gave a good insight in the effectiveness of the simulation of the model. With the comparison on how the model would compare to a dedicated testing infrastructure, we were given further insight in the benefits and the weak-nesses of the model compared to having a dedicated testing infrastructure.

According to the analysis, the model has a benefit in terms of utilisation. With an infrastructure of 8 cores and up, the highest percentage of utilisation will be 14% based on the testing scenario of varying tests, which means that 86% of the potential server resource is not being fulfilled. This discovery would suggest that a company would benefit from using a implementation of our model for testing instead of relying on a dedicated testing infrastructure.

The analysis did not produce any results that suggests that our model is more efficient at processing tests than that of a dedicated testing infrastructure. This is most likely due to the low variation of the test time in the input scenarios. Under the right conditions, where the incoming tests are kept at a low and the testing times are short, a local testing infrastructure would perform well, not considering the economic disadvantages of having a dormant infrastructure. An implementation of the model that does not have enough workers to process tests will create massive latency that gets progressively worse when the demand for processing tests continue to not be met. This was expected to happen, but that the massive increase in average wait time per test skyrocketed to 16 hours in the first 6 hours was unexpected. An implementation of the model would therefore have to be able to maintain a steady status of available workers to process the incoming tests. However, should the amount of active workers become too high then FEAST is vulnerable for starving some workers. Having a rich amount of available workers for the submitted tests will result in reduced test wait time and a high utilisation percentage from the workers.

7.1.1 Improving input files

The data that the analysis is based on can be further improved to reflect more on realistic scenarios. The GitHub commit dataset that the testing input files were based on is a great source to find a pattern that tells us the time of day a developer is most likely to push changes to a code repository, which triggers a build which would again trigger testing. Input files could be improved by having more factors being based on datasets that reflect real-world situations such as how many tests are being run each day by a company and how varied a test would last.

The input files which reflect worker availability is heavily based on unpredictability, which could accurately portray worker behaviour in a real-world implementation of the model. An implementation that is based on the indirect communication through a broker model as explained in section 4.3: *Model communication design patterns* would have to consider unpredictability in terms of how many workers are active, when they join testing and how long they plan to stay. Carefully created input files could be created to cover further scenarios that more accurately reflects the characteristics of the other communication design pattern. These carefully crafted worker scenarios could be based on datasets that contain data on similar

behaviour such as when people sit down to work at their computer.

7.1.2 Reproducibility

The implementation presented in this thesis simulates the behaviour of FEAST in scripts made with Python. These scripts are accessible in the Appendix for this document and on GitHub (Gabrielli, 2021). It is encouraged to retrieve these scripts and run simulations based on the included input files but also generate new scenarios that can be used with the simulation to simulate different behaviour.

7.2 The challenge of emulating multiprocessing

One key element in the model, regardless of the implementation, is the ability to run simultaneous tests in parallel. Significant time was used in the implementation phase to try to emulate this by utilising a computers CPU cores to run tests in parallel through a process called *multiprocessing*. This decision led down the rabbit hole which was filled with problems that needed to be addressed, new libraries to learn and a new way of thinking on how computers run programs. The pursuit of creating a prototype may have been blinding and lead somewhat away from the actual goal of the implementation; to portray the behaviour of the model and collect data that can be analysed. Instead of fumbling about in order to try to get a prototype to work, a step backwards was made in order to look at this from a new perspective. From there, a decision was made to emulate tests being run in parallel by creating a simulation. Though continuing down the rabbit hole could eventually lead to a prototype that was more true to the model, this would increase the risk of committing more significant time to develop a prototype that would be less useful than a simulation in terms of analysis. The decision to simulate tests being run in parallel yielded more gain in terms of accurate measurement to evaluate the models performance than a prototype developed in the same amount of time would be able to tell.

Developing the simulation went above and beyond the expectations. Emulating parallel testing made the process of collecting extensive data about each worker, test and iteration much simpler. Using already known methods within Python3 made the end results more surprising due to the sheer simplicity of the inner

workings of the model but yet portrayed the complex communication between the components.

For future work, a prototype that is more true to the model could be created to more accurately emulate and measure the key characteristics of the model. This prototype could take advantage of multiprocessing by utilising a workers CPU cores to see how that factor would affect the results and efficiency of the model.

7.3 Partitioning

An other key element that the final simulation does not consider is the stage of partitioning. In the simulation, finished partitioned tests are being fed into the simulation and then distributed among the workers. In a true implementation, partitioning is something that have to be addressed.

7.3.1 Broker partitioning

The model has been designed with the intention for the broker to be the one to partition the incoming tests into smaller test bites. This decision was based on the intention to not alter the way current developers would test software; through unit tests. For simplicity sake, having the broker analyse the incoming test file and decide where to partition would mean that the test creator does not have to alter the way they do testing. All they would be required to do is to learn about the new testing platform, but they would keep writing unit tests the way they are used to.

7.3.2 Test creator partitioning

Alternatively to having the broker being the one to partition the test, this task could be given to the test creator. The test creator would therefore be the one partitioning their own tests in containers which would be sent to the broker for distribution among the workers. The benefit of this approach is that the creator could freely partition the tests into containers based on the dependencies. E.g. functionalities that depend on extensive libraries could be gathered in one container image which would result in the dependency to only be downloaded

once, then avoiding the issue of multiple workers downloading the same extensive dependency. This would also lock the broker out from viewing the source code, which would strengthen confidentiality.

7.4 Model design

The model that has been designed and presented in this thesis is very centered around how the model would be implemented in a real-world scenario. One would say that the approach taken to design the model has been from the perspective of a software developer, or an engineering point of view where the goal is to create an abstract model that could act as a blueprint. If we refer back to subsection 2.1.2: *Evolution of Automated Software Testing*, where the gap between the industry and academica is described, the approach to designing this model would be leaning towards addressing the concerns of industry practitioners. This applied method of creating such a model covers realistic concerns and approaches that can be found in a industry setting.

If this model was to be designed from a researchers point of view, the model would become more formal and would follow the conventional methods within software testing in order to produce logical evidence that would support the probability of the traits of the model. Such is the case for a model that attempts to estimate test effort by Srivastava, Varshney, Nama, and Yang (2012), or the theoretical model developed by Smith and Keil (2003) that tries to model the reluctance in software developers to transmit negative software project status up the hierarchy to managers. For this thesis, the focus rested on creating a model that would act as a concept which would allow itself to be easily transformed into an actual implementation. In retrospect, it was not as much of a decision to create a concept as much as it was already settled that this was the right way to design the model. Like an unconscious habit, the thought of creating the model in a different way never crossed the mind. As a personal reflection, this would be noteworthy when facing a similar problem in the future as it would make me somewhat biased towards the approach of designing the model before reflecting on the model's purpose. This bias matches quite well with how the Master's programme is structured; focusing on applied computer and information technology.

The exploration aspect in this thesis has been focused on covering the necessary areas, and how thorough each area has been explained has been depending on their complexity in terms of an industry implementation. These areas extend

further than what this thesis has been able to cover, but creates a foundation that could inspire future work within these areas.

7.5 Model aspect areas

Through the design of the many aspects of the model, various different areas have been briefly visited and presented, some of which would be eligible for their own projects. Even though this thesis briefly covers the different aspects of the model to portray how they all interconnect, each one of these aspects are worth diving deeper into. The main aspect that this thesis has shown is the journey a test goes through from the test creator, through the broker and testers, and finally back to the test creator. Though this may sound simple enough, this aspect touch a lot of different areas that are worth exploring.

One aspect that has been addressed, but not covered in its entirety is how the model would be implemented into a CI/CD pipeline. It has been previously addressed that the model should not alter the entire pipeline but instead integrate the model within it. Creating a "remote service", or a platform, that the build would be sent to along with the tests themselves is one approach that has been mentioned. However, investigating further how the integration would occur and if it is possible it could be done seamlessly without dependencies such as plug-ins is an interesting area to explore.

A factor which is very prominent in the design of the model but very difficult to address is how the broker should split the test submitted from the creator to smaller pieces. This aspect touches a lot of issues that need to be explored in order to find a solution. One of these issues is how the splitting of unit tests should occur when involving dependencies. What would be the effect of splitting the unit tests based entirely on their dependencies, and would this affect efficiency? Would a pipe-filter pattern be a valid solution to optimise the splitting of unit tests?

When designing the architectural design for the components in the model, there is an emphasis on the importance of data integrity and security in the communication portion of the model. There are several problems that need to be addressed within this area. How and when should encryption occur? What insight of the test itself should be given to the testers, and how would this affect data integrity.

Lastly, there is a desired focus on costs when FEAST is implemented as a TTaaS.

How much such a service would cost a company to use and how much each worker should be paid is a vast area that requires more work.

7.6 Evaluating the exploration

The model created in this thesis is designed from scratch; no previously created model has been the foundations for the FEAST. This required a lot of exploration in different areas ranging from software architecture for each stakeholder to the payment each worker should receive. Although the areas has been covered to a certain degree which is feasible for creating an implementation of the model, more coverage is needed within the different areas.

7.7 Comparing FEAST to existing literature

Glass et al. (2006) discusses the difficulties of putting the prototypes developed by researchers into practise; they require tools that are perceived by practitioners as a luxury. They further explain that industry practitioners tends to take care of their own testing practises and that this is a right and responsibility that have to be preserved. This is well reflected throughout Chapter 4: *Designing the model* which addresses the different approaches that can be used to implement the model.

Throughout this thesis, there has been a focus on the applied implementations of the model and the challenges and concerns that needs to be addressed with respect to industry applications. This would suggest that this thesis would stand amongst literature that share these views and address these issue as opposed to academic papers.

An interesting observation occurred when rereading sections from the DevOps Survey by Forsgren (2018). She emphasise that there are benefits in working with smaller batches (Forsgren, 2018, 42). Although she talks about this in context with organisational structures and how developer team work, it is interesting to see the similarities between that statement and the aim of this thesis; to divide tests into smaller batches for quicker testing.

7.8 Future work

This chapter has presented different aspects that needs to be addressed with the model and mentioned future work possibilities for each aspect. This section will more thoroughly explore the future work within some of the aspects.

7.8.1 Develop a prototype of FEAST

There are more work that needs to be done with the implementation of the model. This thesis has simulated the behaviour of FEAST, the next logical step would be to create a prototype. Chapter 4: *Designing the model* defines a good blueprint in which a prototype can be based on. Several aspects has to be addressed and developed such as partitioning in the broker or partitioning from the creator. Furthermore, the prototype would be able to experiment with further functionalities not yet covered in this thesis, such as utilising multiprocessing at the worker in order to process more test bites at once.

Such a project would be fitting for a Bachelor's thesis where the goal is to create a working prototype of FEAST and evaluate how true the prototype ended up being in regards to the model designed in this thesis.

7.8.2 Integrating FEAST in CI/CD pipeline technologies

Further drafting is needed to discover exactly how the model may be universally adapted into CI/CD pipeline technologies. If an implementation of the model offer TTaaS, how does the automated testing phase become outsourced to the third-party that offers TTaaS? If the model is implemented as a in-house solution, how will the tests be sent to the broker for distribution?

There is a lot of room for exploration and experimentation to address these specific implementation scenarios.

• Would it be optimal to create a third-party platform that would integrate the outsourcing of testing with a plug-in?

- Should the model be extended to offer all stages of the Continuous Integration and Continuous Deployment pipeline into its own solution that can be offered as a in-house solution?
- Will creating a third-party platform that manages brokers and provide the IP addresses of one broker to the pipeline support most CI/CD pipeline technologies?

These are all exciting areas worth diving deeper into.

7.9 Revisiting the problem statement

The problem statement as defined early in this thesis was the following:

"Explore the effect of automated testing by designing and developing a pull-based job-distribution scheme in order to achieve reduced completion times."

Throughout this thesis, a thorough model, called FEAST, has been designed and presented which is based on the concept of a *pull-based job-distribution scheme*. The behaviour of this model has been simulated in order to collect data that would help to see how FEAST compares to the traditional ways of testing; using a dedicated testing infrastructure.

"Effect" turned out to be too broad of a term and has not been covered in its entirety through this thesis. Although metrics has been explored such as test time and utilisation, many more metrics of the model may be evaluated. However, seeing that this thesis is exploratory, an open term such as "effect" may be justified. One does not know for sure what they are going in to. By keeping the open term "effect", one will keep an open mind to allow themselves to explore and discover the unique metrics that constitutes as "effects". If the problem statement would substitute "effect" for "utilisation", then a design and implementation would be aimed towards this metric instead of being open to discover new measurements.

In retrospect, focus was shifted during the simulation from focusing on completion times to focusing on utilisation of each worker and infrastructure. Data was collected from the simulation on test wait times, utilisation and the relationship between available workers and remaining tests. There is no confidence in concluding whether FEAST would reduce completion times at this point, only the assumption that when enough workers are available, then our model would perform equally well or better than a testing infrastructure. However, there are more factors to include in order to more confidently support this fact such as the time it takes for a broker to partition, distribute and collect results.

Based on what has been discovered by the simulation and analysis, FEAST would provide better utilisation than a dedicated infrastructure as long as there is enough available workers for the submitted tests. We can assume that an implementation of FEAST as a "True Testing-as-a-Service" would be considerably cheaper to use than purchasing and maintaining a dedicated server based on the discoveries on costs from different cloud vendors. If a TTaaS implementation of the FEAST is able to provide enough workers to keep test wait time at a minimal, then this service would become a viable alternative to a dedicated testing infrastructure.

Chapter 8

Conclusion

The goal of this thesis has been to design and implement a model in an attempt to reduce completion times associated with automated software testing.

This thesis provides a thoroughly designed model named *FEAST*, or *Framework for Elastic Administration of Software Tests*, which is based on a "pull-based job-distribution scheme".

The behaviour of FEAST has been simulated in order to measure its effects on software testing compared to traditional testing infrastructure. Data from the simulations were inconclusive towards the reduction of completion times, but provided data that suggests a higher utilisation percentage with FEAST than local testing infrastructure.

A publicly available implementation of FEAST could create a new cloud service which is referred to in this thesis as "True Testing-as-a-Service". This thesis speculate that such an implementation would be more cost efficient for businesses compared to purchasing and maintaining a local testing infrastructure.

Future work is encouraged to develop a prototype of FEAST and address how FEAST would be implemented in a CI/CD pipeline.

References

- Amazon. (2021, May). Amazon EC2 On-Demand Pricing. Retrieved 2021-05-05, from https://aws.amazon.com/ec2/pricing/on-demand/
- Anastasov, M. (2021, February). Continuous Integration (CI) Explained. Retrieved 2021-02-10, from https://semaphoreci.com/continuous -integration
- Bertolino, A. (2007, May). Software Testing Research: Achievements, Challenges, Dreams. In Future of Software Engineering (FOSE '07) (pp. 85–103). Minneapolis, MN, USA: IEEE. Retrieved 2021-04-10, from http://ieeexplore.ieee.org/document/4221614/ doi: 10.1109/FOSE.2007.25
- Codoid. (2019, May). What Are The Most Common Issues Affecting Automated Testing? Retrieved 2021-04-19, from https://codoid.com/what-are -the-most-common-issues-affecting-automated-testing/
- Dave, D. (2021, April). GitHub Commit Messages Dataset. Retrieved 2021-04-26, from https://www.kaggle.com/dhruvildave/github-commit -messages-dataset
- Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Petersen, K., & Mäntylä, M. V. (2012). Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In 2012 7th International Workshop on Automation of Software Test (AST) (pp. 36–42). doi: 10.1109/IWAST.2012.6228988
- Dustin, E., Rashka, J., & Paul, J. (1999). Automated Software Testing: Introduction, Management, and Performance. Addison-Wesley Professional. Retrieved 2021-02-05, from https://books.google.no/ books?id=-Jxobzm20NkC&dq=What+is+automated+software+ testing&lr=&source=gbs_navlinks_s
- Forsgren, N. (2018). Accelerate: The science of lean software and devops: building and scaling high performing technology organizations. IT Revolution Press.

- Gabrielli, E. G. (2021, May). *ast-model-simulation*. Retrieved from https://github.com/emilgab/ast-model-simulation
- Glass, R. L., Collard, R., Bertolino, A., Bach, J., & Kaner, C. (2006, August). Software Testing and Industry Needs. *IEEE Software*, 23(4), 55-57. Retrieved 2021-02-26, from https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1657939
- Google. (2021, May). All pricing. Retrieved from https://cloud.google.com/ compute/all-pricing
- Hykes, S. (2018, March). Au revoir. Retrieved 2021-02-17, from https://
 www.docker.com/blog/au-revoir/
- IBM. (2019, June). Virtualization. Retrieved 2021-02-12, from https://www .ibm.com/cloud/learn/virtualization-a-complete-guide
- Jena, A. K., Das, H., & Mohapatra, D. P. (2020). Automated software testing: foundations, applications and challenges. (ISBN: 9789811524554 OCLC: 1237817622)
- Kurose, J. F., & Ross, K. W. (2013). *Computer networking: a top-down approach* (6th ed ed.). Boston: Pearson. (OCLC: ocn769141382)
- Lahn, M. (2021, January). How much does a server cost for a small business in 2021. Retrieved 2021-05-05, from https://www.servermania.com/ kb/articles/how-much-does-a-server-cost-for-a-small -business/
- Microsoft. (2021, May). Linux Virtual Machines Pricing. Retrieved 2021-05-05, from https://azure.microsoft.com/en-us/pricing/details/ virtual-machines/linux/
- Morales, J. A., Yasar, H., & Volkman, A. (2018, May). Implementing DevOps practices in highly regulated environments. In *Proceedings of the* 19th International Conference on Agile Software Development: Companion (pp. 1–9). Porto Portugal: ACM. Retrieved 2021-03-29, from https://dl.acm.org/doi/10.1145/3234152.3234188 doi: 10.1145/ 3234152.3234188
- RDI. (2021, May). How Much Does a Server Cost for a Small Business. Retrieved 2021-05-05, from https://rdi-it.com/how-much-does -a-server-cost-for-a-small-business/
- RedHat. (2013, September). Press Release: Red Hat and dotCloud Collaborate on Docker to Bring Next Generation Linux Container Enhancements to OpenShift Platform-as-a-Service [Press Release]. Retrieved 2021-02-17, from https://www.redhat.com/en/about/press-releases/ red-hat-and-dotcloud-collaborate-on-docker-to-bring-next -generation-linux-container-enhancements-to-openshift
- Richards, M. (2015). Software architecture patterns: understanding common architecture patterns and when to use them. (OCLC: 1047880356)

- Smith, H. J., & Keil, M. (2003, January). The reluctance to report bad news on troubled software projects: a theoretical model. *Information Systems Journal*, 13(1), 69–95. Retrieved 2021-05-10, from http://doi.wiley .com/10.1046/j.1365-2575.2003.00139.x doi: 10.1046/j.1365-2575 .2003.00139.x
- Srivastava, P. R., Varshney, A., Nama, P., & Yang, X. S. (2012). Software test effort estimation: a model based on cuckoo search. *International Journal of Bio-Inspired Computation*, 4(5), 278. Retrieved 2021-05-10, from http://www.inderscience.com/link.php?id=49888 doi: 10 .1504/IJBIC.2012.049888
- Sroczkowski, P. (2021, February). Cloud: IaaS vs PaaS vs SaaS vs DaaS vs
 FaaS vs DBaaS. Retrieved 2021-02-16, from https://brainhub.eu/
 blog/cloud-architecture-saas-faas-xaas/
- Wacker, M. (2015, April). Just Say No to More End-to-End Tests. Retrieved 2021-04-11, from https://testing.googleblog.com/2015/04/just -say-no-to-more-end-to-end-tests.html

Appendix A

Simulation files

The simulation contains a lot of different files; from the generators that generates input scenario files to the main simulation file. This part of the appendix will contain all of the different Python files that were used. All files, including input files, are available on GitHub: https://github.com/emilgab/ast-model-simulation.

A.1 Input generators

A.1.1 Generator for infrastructure input files

```
1 import csv
2
3 # Contains number of individual cores/workers for the infrastructure
     scenario
4 \text{ cores} = 8
5
6 # Name of the resulting .csv file
7 csv_filename = "INFRASTRUCTUREFILE.csv"
8
9 with open(csv_filename,"w") as csv_file:
      infra_writer = csv.writer(csv_file, quotechar='"', quoting=csv.
10
      QUOTE_MINIMAL)
     infra_writer.writerow(["CoreID","StartTimeAvailable","
11
     PerformanceRating","AvailabilityTime"])
12
     for x in range(1, cores+1):
          infra_writer.writerow(["I"+str(x).zfill(4),"00.00","1","1440"])
13
```

A.1.2 Generator for test input files

```
1 import re
2 import random
3 import csv
4 import string
5 # Dictionary for counting number of tests per timestamp
6 # time_count = {}
7 #
s # # I/O and For-loop used to extract data from dataset containing data on
      GitHub commits
9 # with open("full.csv", "r") as file:
       for row in file.readlines():
10 #
            # ReGex used to find the timestamp
11 #
12 #
            match = re.search("[0-9][0-9]+:[0-9][0-9]:[0-9][0-9]",row)
            if match:
13 #
                print(match.group(0))
14 #
15 #
                entry = match.group(0)[:5]
                # if-statement used to count how often a certain timestamp
16 #
      is used
17 #
                if entry in time_count:
                    print("Time found in dictionary, incrementing")
18 #
                     time_count[entry] += 1
19 #
20 #
                else:
21 #
                    print("Time not found, adding entry")
22 #
                    time_count[entry] = 1
23 #
24 # print(time_count)
25 # Results from the GitHub dataset extraction
26 result = {'12:27': 3444, '01:23': 2141, '01:21': 2134, '01:01': 2688,
27 '23:58': 2785, '15:56': 4797, '15:55': 5011, '15:54': 4892,
28 '00:51': 2450, '00:50': 2445, '00:49': 2479, '15:48': 4735,
29 '00:47': 2405, '06:12': 1345, '08:52': 2331, '09:56': 3237,
30 '16:21': 5019, '20:38': 3239, '11:54': 3882, '22:52': 3367,
31 '09:23': 2811, '07:10': 1450, '16:02': 4529, '07:01': 1401,
32 '15:00': 4726, '06:53': 1362, '06:51': 1490, '06:41': 1413,
33 '06:38': 1343, '17:55': 4476, '12:08': 3571, '12:02': 3529,
34 '20:42': 3530, '18:43': 3809, '17:54': 4266, '18:24': 4043,
35 '18:14': 4078, '11:57': 3886, '18:11': 4228, '18:09': 4383,
36 '18:07': 4140, '02:06': 1664, '02:03': 1947, '02:46': 1495,
37 '17:21': 4651, '18:49': 3940, '19:49': 3551, '18:27': 3953,
38 '20:27': 3463, '10:31': 3684, '10:56': 3857, '09:50': 3155,
```

39 '06:16':	1380,	' 07:25':	1520,	'16:12':	4795,	'06:05':	1285,
40 '06:04':	1342,	'05:58':	1369,	' 05:56':	1236,	'08:55' :	2503,
41 '05:53':	1239,	'01:22':	2123,	'01:19':	2138,	'13:55':	4083,
42 '10:16':	3438,	'13:12':	3602,	'11:50':	3822,	'17:26':	4638,
43 '08:25':	2144,	'08:24':	2081,	' 07:17':	1711,	'16:16':	4723,
44 '16:15':	4936,	'16:09':	4698,	' 06:21 ' :	1361,	'13:31':	3959,
45 '14:24':	4405,	'13:13':	3747,	' 13:53':	3950,	' 04:36':	1215,
46 '20:29':	3708,	' 16:49':	5040,	'16:41':	4733,	' 10:27 ' :	3342,
47 '15:31':	4620,	'14:35':	4397,	^{13:17}	3564,	'12:56':	3661,
48 '09:25':	2982,	' 09:11':	2736,	' 18:32 ' :	3920,	' 15:01':	4719,
49 '08:39':	2325,	' 02:02 ' :	2352,	'18:25':	3945,	' 17:20':	4692,
50 '20:46':	3458,	'12:42':	3455,	' 18:29 ' :	4104,	'18:28':	3820,
51 '10:51':	3784,	' 08:45':	2226,	' 08:53 ' :	2539,	' 20:33':	3581,
52 '02:41':	1516,	' 04:40 ' :	1248,	' 01:52 ' :	1856,	' 19:04':	3747,
53 '00:16':	6911,	'00:15':	2811,	'13:16':	3615,	' 15:43':	4699,
54 '21:36':	3656,	' 06:40':	1323,	' 05:55 ' :	1229,	' 05:47 ' :	1363,
55 '05:46':	1302,	' 05:26':	1261,	' 17:53':	4322,	'13:47':	3886,
56 '13:02':	3460,	' 13:00':	3365,	'15:45':	4827,	'12:44':	3400,
57 '17:39':	4394,	'15:44':	4817,	'21:18':	3550,	'23:38':	3044,
58 '21:21':	3743,	? 21:04 ? :	3661,	' 21:03 ' :	3408,	' 20:55':	3445,
59 '13:18':	3683,	' 09:22':	2955,	' 09:19':	2757,	'16:24':	4735,
60 '07:56':	1762,	' 06:09':	1345,	'14:08':	4301,	'22:57':	3319,
61 '19:07':	3643,	' 11:49':	3900,	'08:35':	2117,	'18:41':	4066,
62 '23:37':	2976,	' 13:20':	3621,	?21:09? :	3764,	'11:18':	3947,
63 '19:15':	3517,	'08:18':	1963,	'16:11':	4936,	' 17:09':	4784,
64 '15:50':	4663,	'18:52':	3782,	' 18:51':	3869,	' 18:47':	3739,
65 '18:37':	3935,	' 17:50':	4585,	' 15:36':	4808,	' 15:35':	4686,
66 '00:28':	2478,	'23:24':	3081,	' 20:17':	3213,	'11:55':	3727,
67 '09:15':	2663,	' 09:12':	2662,	' 05:57':	1219,	' 05:52':	1255,
68 '20:36':	3584,	'18:05':	4143,	' 18:03 ' :	4384,	'18:02':	4270,
69 '18:01':	4226,	'14:21':	4330,	'23:18':	3169,	' 15:57':	4904,
70 '06:17':	1291,	' 09:17':	2674,	' 06:13':	1339,	' 05:51':	1378,
⁷¹ '19:27':	3831,	'11:41':	3919,	' 11:28 ' :	3911,	' 17:24':	4581,
72 '07:55':	1814,	' 07:42':	1703,	' 10:21 ' :	3397,	' 14:52 ' :	4622,
73 '14:50':	4655,	'06:27':	1343,	'22:46':	3382,	' 17:46':	4485,
74 '23:03':	3212,	'10:46':	3841,	' 09:36':	2919,	' 06:10 ' :	1369,
75 '09:28':	2910,	'18:53':	3836,	' 21:51':	3703,	? 20:09 ? :	3609,
76 '19:37':	3482,	'00:13':	2766,	' 16:10 ' :	4906,	' 15:23':	4791,
77 '14:06':	4227,	'15:08':	4972,	'08:28':	2111,	' 16:31':	4956,
78 '16:30':	4924,	' 16:29':	4951,	'13:28':	3926,	' 11:02 ' :	3679,
79 '11:01':	3747,	'17:47':	4449,	' 09:47 ' :	3124,	' 09:35 ' :	2957,
80 '16:23':	4794,	' 17:14':	4666,	^{'17:13'} :	4659,	^{'19:12'} :	3586,
⁸¹ '12:11':	3423,	' 12:10':		' 17:08':		'11:17':	
82 '08:16':	2056,	' 14:46':		' 01:33':	1983,	? 21:24 ? :	
83 '18:06':	4278,	' 07:32':	1642,	' 06:11':	1341,	' 13:08':	
84 '06:07':	1346,	' 06:03':	1455,	^{13:59}	4069,	' 05:43':	
85 '10:49':	3922,	' 17:12':		'22:08':	3509,	; 17:01 ;	
86 '17:00':	4955,	' 08:41':		' 19:34':	3388,	'18:18':	
87 '02:17':				' 02:14':	1928,	' 18:10':	

88 '18:15':	-		-		-	'14:47':	
89 '11:09':		' 02:01 ' :	-	'08:40':	-	'17:17':	
90 '22:10':	3540,	'21:08':	-	' 03:39':		'03:05':	
91 '03:00':		'08:22':	-	'17:02':	4684,	'14:31':	
92 '13:30':		'12:52':		' 09:59':	3509,	' 09:55 ' :	3215,
93 '08:36':	2236,	'08:32':	2168,	'15:34':	-	'06:49':	1522,
94 '06:48':	1426,	'06:18':	1321,	'12:14':	3678,	'11:59':	3719,
95 '11:32':	3954,	'16:14':	-	'15:05':	4788,	'22:00':	3695,
96 '04:35':	-	'00:44':	2417,	'17:52':	-	' 01:30 ' :	2067,
97 '01:04':		'17:03':	4718,	'19:55' :		'16:45':	
98 '00:43':	2437,	'16:43':	4943,	'16:42':	4767,	'16:33':	
99 ' 15:47':	-	'01:58':		'10:55':	4073,	'15:42':	
100 '05:34':			4345,	^{22:07}	3643,	'23:06':	3262,
101 '23:05':			3303,	'17:10':	4966,	'14:10':	4004,
102 '01:03':	2557,	' 02:33':	1680,	'16:25':	4721,	'02:04':	1871,
103 '00:33':	2631,	^{19:17}	3349,	'11:11':		' 01:38 ' :	2050,
104 '01:27':	2235,	'13:44':	4050,	'08:54':	2374,	' 05:13 ' :	1255,
105 '11:26':	4160,	' 10:50 ' :	3686,	'14:28':	4599,	' 02:50 ' :	-
106 '14:14':			-	'16:06':		'12:17':	
107 '09:05':	2396,			^{17:36}	4630,	'10:11':	
108 '17:11':	4785,	' 07:59':	1827,	'15:32':	4629,	'09:31':	2886,
109 '14:29':	4518,	'14:23':	4336,	' 05:27 ' :		'14:13':	4049,
110 '14:04':	3919,	' 05:00':	1243,	'20:12':	3469,	' 00:29 ' :	2696,
111 '16:37':	4739,	'22:39':	3455,	^{20:40}	3608,	'12:50':	3555,
112 '17:48':	4483,	'10:38':	3660,	^{19:01}	3557,	'23:07':	3258,
113 '22:34':	3597,	' 19:00':	3548,	'17:35':	4371,	'06:22':	1466,
114 '08:02':	1862,	' 08:17':	2047,	'14:16':	4135,	' 09:29 ' :	3020,
115 '20:28':	3564,	' 02:36':	1621,	'17:34':	4493,	'02:32':	1599,
116 '02:31':	1583,	'17:29':	-	'01:28':	2259,	' 01:09 ' :	2297,
117 '01:55':	1863,	' 00:39':		^{19:09}	3613,	'17:45':	4510,
118 '21:53':	3723,	'16:28':	4993,	'20:02':	3554,	'12:57':	3650,
119 '19:26':	3472,	'19:24':	3596,	'21:22':		'03:01':	1414,
120 '20:45':	3407,	'10:41':	3671,	' 05:09':	1307,	' 05:08 ' :	1337,
121 '05:07':		'05:06':	1224,	' 09:38':		'20:04':	-
122 '22:06':		' 19:05 ' :	3777,	'15:25':		' 00:21 ' :	2579,
123 '13:14':	3496,	' 09:27':	2902,	'18:26':	4031,	'18:23':	-
124 '18:22':	-	'16:51':	-	'12:26':	3559,	'00:10':	
125 '07:30':				'13:41':		^{'19:32'} :	
126 '11:27':				'14:37':		'22:54':	3391,
127 '14:51':						20:44':	3575,
128 '11:15':			3969,	'03:10':		'11:08':	3809,
129 '10:59':			3710,	' 10:33':	3797,	'00:01':	2698,
130 '10:24':				'12:15':	3560,	' 10:13 ' :	
131 '19:58':			4898,	'23:53':	2972,	'23:11':	3105,
132 '15:09':	4695,	'14:26':	4436,	'14:25':		' 19:16 ' :	3453,
133 '06:46':				'06:44':	-	'15:22':	
134 '14:54':			-			'16:54':	
135 '16:46':							
136 '16:38':	4952,	'16:36':	4820,	'21:45':	3571,	' 00:53 ' :	2444,

137 '00:25':	2643,	; 10:09:	3431,	' 10:08 ' :	3353,	' 13:05 ' :	3454,
138 '18:55':	3676,	' 09:52 ' :	3276,	' 11:03 ' :	3720,	' 11:39':	3825,
139 '13:36':	3944,	'06:26':	1371,	^{'19:39'} :	3424,	' 01:37':	2119,
140 '19:19':	3462,	'05:25':	1245,	'11:13':	3887,	' 03:25 ' :	1306,
141 '18:54':	3689,	' 05:21':	1216,	'16:20':	5228,	'00:00':	3823,
142 '23:59':	2927,	'14:58':	4888,	' 14:57':	4605,	' 23:30':	3050,
143 '13:25':	3742,	'05:22':	1257,	' 10:14':	3282,	' 19:03':	3610,
144 '12:33':	3379,	'12:32':	3486,	' 09:26 ' :	2911,	' 09:21':	2774,
145 '09:20':	2835,	' 09:18':	2755,	' 09:16 ' :	2784,	'23:46':	3041,
146 '10:32':	3492,	'01:11':	2157,	' 02:10 ' :	1684,	' 17:04':	4855,
147 '20:50':	3407,	'14:56':	4570,	' 12:09 ' :	3583,	^{10:37}	3535,
148 '17:33':	4366,	' 20:30':	3557,	' 01:16 ' :	2260,	' 00:58 ' :	2364,
149 '18:35':	3966,	'15:26':	4929,	'18:08':	4194,	'23:55':	2803,
150 '14:55':	4558,	'18:34':	3892,	' 08:59':	2612,	' 15:41':	4846,
151 '13:52':	4218,	' 03:51':	1242,	' 16:19':	4684,	' 15:53':	4765,
152 '15:52':	4995,	' 18:50':	3809,	' 15:49':	4946,	' 16:32':	4733,
153 '18:20':	4073,	'14:43':	4477,	'23:42':	2843,	'17:37':	4544,
154 '22:31':	3344,	'14:17':	4320,	'14:15':	4196,	'20:06':	3436,
155 '18:17':	3946,	'23:43':	2991,	' 02:16':	1872,	'21:33':	3503,
156 '12:13':	3470,	'12:12':	3576,	' 18:45 ' :	4091,	' 09:40 ' :	3008,
157 '11:37':	3958,	'04:04':	1344,	¹⁷ :27 ¹	4628,	' 17:22':	4656,
158 '08:43':	2318,	'08:42':	2291,	' 13:40 ' :	3866,	' 02:43 ' :	1552,
159 '00:48':	2325,	'15:13':	4602,	' 12:35':	3388,	' 10:43':	3618,
160 '18:58':	3580,	'15:40':	4532,	' 15:29':	4930,	' 01:57 ' :	1813,
161 '12:41':	3487,	; 00:20:	2670,	'14:45':	4724,	'14:44':	4436,
162 '16:47':	4880,	' 11:36':	4052,	' 05:50 ' :	1261,	' 05:49':	1486,
163 '14:42':	4657,	'13:39':	3954,	' 10:02 ' :	3244,	'15:16':	4825,
164 '12:28':	3374,	'18:12':	4231,	'08:58':	2569,	'16:44':	5074,
165 '17:42':	4448,	' 16:35':	4753,	'11:24':	4145,	'12:24':	3690,
166 '17:41':	4439,	' 09:53':	3202,	' 22:15 ' :	3530,	' 19:11':	3708,
167 '14:40':	4687,	'16:55':	4925,	' 07:12 ' :	1458,	' 17:59':	4537,
168 '02:52':	1364,	' 01:51':	1881,	' 16:50 ' :	4949,	' 16:04':	4655,
169 '13:48':	3965,	' 05:37':	1366,	^{17:31}	4469,	' 17:30':	4542,
170 '01:36':	2115,	^{'19:29'} :	3551,	' 15:33':	4777,	'23:14':	3204,
171 '05:36':	1293,	' 05:35':	1235,	' 05:33':	1278,	' 13:19':	3601,
172 '18:42':	3811,	' 02:59':	1438,	' 02:42 ' :	1520,	' 20:39':	3626,
173 '07:27':		'15:46':		' 14:32':		'13:06':	
174 '21:15':							1282,
175 '23:15':							
176 '06:50':					3679,	'03:04':	1614,
177 '21:02':	3445,	'21:11':	3594,	' 16:07 ' :	4779,	' 07:05 ' :	1481,
178 '15:04':	4626,	' 21:59':	3568,	' 03:37 ' :	1341,	' 03:34':	1365,
179 '01:59':				' 00:32 ' :		' 18:31':	
180 '23:47':			3604,	' 15:15':		' 11:40':	3851,
181 '00:40':				' 06:24 ' :		' 23:01':	3072,
182 '15:28':						' 09:00 ' :	2985,
183 '15:59':							3665,
184 '12:46':							
185 '07:16':	1705,	' 07:15':	1519,	' 07:13 ' :	1532,	'11:21':	3827,

186 '11:23':	-	^{10:57}	-	'21:52':	-	'13:23':	-
187 '09:44':	-	'01:40':	-	'14:36':	-	'21:44':	-
188 '03:23':	-	'15:20':		'10:10':	-	'08:21':	-
189 '17:19':		'07:53':	-	'09:33':		'10:22':	
190 '15:19':		'19:08':	-	'12:59':		'15:17':	
191 '15:21':	-	'17:18':	-	'01:12':	-	'01:10':	-
192 '18:30':	-	'15:14':	-	'23:27':	-	'12:48':	-
193 '19:18':	-	'23:34':	-	'16:56':	-	'06:23':	-
194 '21:20':	-	'13:03':		'09:30':		'20:52':	-
195 '14:41':	-	'05:18':		'20:15':		'19:59':	
196 '02:58':		'17:07':		'09:06':		'06:01':	
197 '05:59':	-	'20:21':		'20:00':		'12:34':	
198 '03:38':	-	'04:59':		'02:39':	-	'21:39':	-
199 '01:41':	-	'21:34':		'16:22':		'21:17':	
200 '21:16':		'00:17':		,00:09,:		'16:58':	-
201 '19:47':		'09:14':		'08:08':		'23:56':	
202 '00:06':		'10:35':	-	'20:32':	-	'14:27':	
203 '03:52':	-	' 01:35 ' :		^{20:57}	3487,	' 05:23 ' :	
204 '09:41':		' 03:27 ' :	1406,	'03:02':	-	'19:45' :	
205 '14:34':		'11:20':	3771,	'22:55':		'22:35':	3492,
206 '20:26':	-	' 09:13 ' :		'02:11':	-	^{19:50}	
207 '00:22':	-	'23:22':	-	'21:05':		'19:44':	
208 '22:47':	3247,	' 09:03 ' :	-	'12:01':	3670,	'17:56':	4409,
209 '11:56':	3683,	'08:48':	2323,	'18:00':	4401,	'15:02':	4618,
210 '03:16':	1421,	'21:38':	-	'21:43':	3554,	'12:38':	3575,
211 '03:11':	1451,	'19:48' :	3645,	'10:44':		'03:41':	1256,
212 '17:40':	-	'17:25':	-	' 07:51':	1850,	'10:28':	3653,
213 '04:08':	1440,	^{17:49}	4374,	'18:38':	3918,	'22:24':	3457,
214 '20:58':		'11:58':	3686,	'13:43':	3877,	'11:42':	
215 '19:31':	-	^{20:37} [:]	3460,	'13:49':	-	'14:30':	
²¹⁶ '04:27':	-	'12:29':	-	'18:56':		'06:30':	1264,
217 '14:05':	-	'22:18':	-	'17:23':		'19:51':	-
218 '00:08':	-	'11:52':	-	^{10:19}		'21:55':	
219 '09:48':	-	'23:02':	-	'23:57':		'06:54':	
220 '05:19':	1173,	'11:25':	-	'23:16':	3071,	'13:57':	4143,
221 '03:40':	1254,	'12:30':	3486,	^{20:03}	3848,	' 09:49 ' :	3321,
222 '08:50':		' 02:08 ' :	,	'15:30':		'12:36':	
223 '12:05':						'16:48':	
224 '06:36':				' 06:29 ' :		'06:08':	
225 '21:06':						'21:32':	
226 '20:22':		'18:39' :	3745,	'10:36':	3819,	' 09:10 ' :	
227 '09:07':		'22:02':		'20:16':		'16:08':	4963,
228 '21:14':		'14:02':		'13:15':		' 09:09 ' :	
229 '09:08':		' 08:38 ' :		' 08:03 ' :		^{20:19}	
230 '06:47':		' 09:43 ' :		'18:21':		'03:14':	
231 '18:13':		; 00:07 ;		; 00:05 ;		'10:29':	
232 '07:40':	-	'07:06':		'03:18':		'20:23':	
233 '21:48':		'18:36':		'18:46':		'06:58':	
234 '04:02':	1318,	'00:55 ' :	2439,	'12:37 ' :	3549,	'12:58 ' :	3647,

							0 - 1 0
235 '06:28':		'09:24':	-	20:07:		'21:01':	
236 '20:59':		'12:22':	-	'00:31':		'00:03':	
237 '23:00':		'23:54':	-	'00:36':		'04:34':	
238 '23:49':		'05:38':	-	'03:15':	-	'14:59':	
239 '11:43':		'11:07':	-	'10:58':		'13:29':	
240 '12:31':		'02:22':		'05:11':		'00:41':	
241 '11:30':		'07:38':		'05:29':		'13:46':	-
242 '21:57':		'07:45':		'00:30':		'13:58':	-
243 '06:25':		'15:24':		'11:22':	-	'06:20':	
244 '06:19':	-	'10:00':		'06:37':		'19:38':	
245 '23:40':		'04:43':		'04:11':		'04:10':	
246 '13:09':		^{'04:09'} :		^{'04:03'} :		'22:14':	
247 '11:12':		^{'02:48'} :	-	^{'02:47'} :	'	^{'00:27'} :	
248 '17:05':		'21:37':	-	'10:53':	- · ·	'10:12':	-
249 '01:14':		^{'02:53'} :	-	² 08:30 ² :		'14:20':	
250 '10:15':		'01:45':	-	'17:06':		'16:59':	-
251 '09:42':		'15:27':	-	'21:12':		'20:13':	
252 '09:57':		'08:20':	-	'16:18':		'08:05':	-
253 '01:31':		'13:01':		^{'04} :18 ['] :		^{22:59} ²	
254 '19:14':		^{20:49} ²⁰		^{'07:34'} :		^{'05:28'} :	-
255 '06:00':		'23:23':	-	'23:21':		'13:45':	-
256 '22:09':		'13:37':	-	^{'19:46'} :		'01:15':	-
257 '12:47':		^{20:01} ²⁰		'18:40':		^{12:07} ¹²	
258 '00:34':		'19:35':	-	'13:21':		'14:53':	-
259 '14:01':		'04:57':		'04:56':		^{'06:55'} :	
260 '07:02':		'17:28':	-	^{'02:26'} :		'02:21':	-
261 '17:16':		'02:05':		'09:02':		'04:48':	
262 '00:12':		'23:50':		^{21:13²}		'11:38':	
263 '08:56':		'17:15':		'02:20':		'05:20':	1160,
264 '14:09':		'12:45':		^{20:47} ²⁰		'07:29':	1672, 3668
265 '20:18':		^{'00:23'} :	-	^{23:20} ²³		'12:49':	-
266 '01:26': 267 '19:21':		^{'07:41'} :	-	'21:49':	-	'19:36':	-
267 '19.21'. 268 '21:27':		'04:20': '01:56':	-	'22:20': '04:52':		'13:10': '12:51':	
268 21.27 . 269 '19:30':		'04:07':		'04:01':	-	^{12.51} .	
270 '18:04':		'16:34':		'16:26':		·19:40':	
270 10:04 : 271 '11:53':		'10:52':		'14:22':		·23:09':	
271 11.00 . 272 '04:00':						'12:55':	
272 °04.00 . 273 °05:30':						'13:26':	
274 '05:39':						^{10.20} ¹	
275 '07:00':				'22:11':		^{'19:57'} :	
276 '19:52' :				'08:34':		'08:12':	
277 10:02 .				^{'00:11'} :		^{'09:04'} :	
277 11:00 · · · · · · · · · · · · · · · · · ·				^{'08:51'} :		'01:17':	
278 11.51 . 279 '17:58':				'13:04':		'13:38':	
280 '19:43':				'13:27':		^{10:00} ·	
280 10:10 · · · · · · · · · · · · · · · · · · ·				'00:18':		'20:12':	
282 '03:06':				'14:38':			
283 '22:41':							
	,	• • • •	,		·,	•••••	,

284 '07:18':	1582,	' 07:08':	1476,	' 15:11':	4639,	' 15:37':	4747,
285 '19:54':	3314,	' 19:02 ' :	3497,	' 09:51 ' :	3177,	'23:51':	2989,
286 '22:37':		' 03:30':	1371,	'08:01':		' 07:58 ' :	
287 '10:54':	3818,	'12:04':		'06:43':	1349,	' 02:07 ' :	1653,
288 '10:17':		' 02:30 ' :		' 02:23 ' :		' 08:27 ' :	
289 '02:28':		'11:19':		'00:54':		'05:14':	1269,
290 '23:35':	-	'08:31':	-	' 08:19 ' :		'14:00':	-
291 '16:01':	-	'09:32':	-	'04:21':		'19:25':	
292 '07:07':		'18:33':		'23:32':	•	'23:29':	
293 '04:41':		'13:24':	-	'10:48':		'10:42':	
294 '10:40':				'21:30':		'12:06':	
295 '10:47':				'04:14':		'01:24':	
296 '08:10':	-	'08:00':		'07:33':		'20:24':	
297 '05:15':	-	'04:42':		'21:40':		'03:44':	
298 '15:06':	-	'20:08':		'19:06':		'19:41':	
299 '16:05':			-	'20:05':		'04:38':	-
300 '00:02':				'22:13':		'13:35':	
301 '20:31':	-			'23:19':		'13:22':	
302 '07:37':	-	'07:22':		'21:23':		'19:13':	
303 '03:32':				'14:11':		'03:22':	
304 '03:19':				'01:47':		'00:42':	
305 '09:58':		'08:44':		'06:14':		'07:20':	
306 '12:16':		'00:24':		'18:16':		'07:54':	
307 '14:49':		'15:07':		'05:05':		'22:29':	
308 '01:25':	-	'20:54':		'04:13':		'20:56':	
309 '15:18':	-	'04:39':		'21:31':		'10:06':	
310 '23:28':	-	'19:10':		'22:28':		'07:44':	'
311 '05:31':	-	'03:54':	-	'23:33':		'22:17':	
312 '03:55':		^{'08:29'} :	2178,	'09:45':		'02:51':	
313 '20:41':		'01:49':	1989,	'13:11':		'10:05':	
314 '16:03':		'22:33':		^{20:11} ²⁰		'23:10':	
315 '03:56':	-	^{22:03} ²		^{19:56}	'	'12:19':	
316 '09:39':		'11:06':		'13:42':		'17:32':	
317 '04:47':		'02:40':		^{'05:01'} :		^{'00:57'} :	
318 '10:26':		² 08:57 ² :		'22:12':		^{'07} :24':	
319 '01:50': 320 '22:22':	-	^{22:26} ²		'16:52':		'02:24':	
320 ² 22.22 ³ . 321 ³ 12:21 ³ :		'13:54': '16:57':	-	'08:11': '23:52':		'13:32':	-
321 12.21 . 322 '14:03':		·00:46:		^{23.32} . ^{23.32} .		'14:18': '19:20':	
323 '10:45':		^{'00:10'}		^{'03:43'} :		^{10.20} .	
323 10:40 : 324 '02:12':		'08:14':		'03:45':		'12:20':	
324 02:12 : 325 '18:48':		'23:44':		'17:43':		'08:04':	
326 '22:51':		'22:36':		'04:51':			1463,
327 '05:16' :		'05:24':		^{'01:34'} :		,06:06:	-
328 '22:38':		'11:47':		'14:19':		·17:44':	-
329 10:07 :		^{11.17} , ¹⁰⁷ :57 [,]		'20:25':		'20:43':	-
330 '09:01':	-	'16:13':		'13:34':		·02:37:	1660,
331 '00:35':				'23:36':		'21:28':	
332 '19:33':							
	,		,		. ,		

333 '22:19':	3478,	' 07:21':	1599,	'21:58':	3570,	' 02:27 ' :	1586,
334 '23:31':	3209,	'02:45':	1511,	'21:07':	3620,	' 07:28 ' :	1631,
335 '11:04':	3770,	'00:04':	2840,	'20:48':	3405,	'03:42':	1228,
336 '19:28':	3705,	; 02:00 ;	1898,	'13:56':	4246,	'01:08':	2293,
337 '00:52':	2420,	'15:12':	4601,	'10:20':	3459,	' 09:54':	3220,
338 '04:58':	1232,	'08:46':	2258,	' 07:52 ' :	1640,	'11:10':	3836,
339 '22:43':	3315,	' 08:13':	1915,	'22:44':	3478,	' 08:07 ' :	1963,
340 '10:04':	3395,	'00:26':	2691,	'22:01':	3469,	'12:54':	3618,
341 '21:50':	3513,	' 03:57':	1299,	'23:04':	3224,	'23:48':	2872,
342 '07:36':	1674,	' 07:35':	1703,	' 10:25 ' :	3432,	'11:34':	3775,
343 '04:05':	1284,	'05:54':	1366,	' 08:09 ' :	1958,	' 04:19 ' :	1359,
344 '07:09':	1549,	'14:48':	4673,	'22:25':	3556,	'12:23':	-
345 '17:38':	4466,	'22:30':	3394,	'04:31':	1160,	' 07:47':	
346 '15:38':	5002,	' 02:38':	1605,	' 04:53 ' :	1265,	'12:43':	3526,
347 '03:24':	1322,	'03:07':	1494,	'03:17':	1293,	'21:41':	3643,
348 '12:53':	3623,	^{19:53}	3338,	'18:44':	3966,	'20:20':	3573,
349 '06:39':	1419,	' 20:53':	3505,	' 06:59':	1470,	' 11:29':	4090,
350 '04:23':	1179,	'21:42':	3486,	' 07:23 ' :	1549,	' 01:13 ' :	2133,
351 '07:50':	1665,	' 10:01':	3313,	' 02:19 ' :	1554,	' 03:09 ' :	1569,
352 '01:44':	1977,	'20:35':	3542,	'22:40':	3385,	'21:19':	3512,
353 '04:33':	1238,	'02:25':	1701,	' 02:09 ' :	1774,	'12:25':	3334,
354 '20:10':	3451,	; 01:07:	2376,	' 03:33 ' :	1377,	' 19:23':	3602,
355 '19:22':	3374,	' 08:15 ' :	2051,	'01:54':	1815,	' 01:48 ' :	1926,
356 '04:30':	1178,	'06:31':	1394,	'04:24':	1196,	'03:26':	1284,
357 '03:48':	1314,	' 03:36':	1373,	?21:00? :	3394,	; 00:56:	2428,
358 '03:49':	1297,	' 03:47 ' :	1273,	' 02:55 ' :	1371,	' 03:53 ' :	1254,
359 '03:35':	1286,	'06:32':	1406,	'13:07':	3601,	'08:06':	1910,
360 '07:43':	1723,	' 09:37':	2915,	' 16:39':	4784,	' 04:44':	1289,
361 '01:05':	2361,	' 01:46 ' :	2034,	' 00:45 ' :	2379,	' 04:50 ' :	1281,
362 '01:32':	2008,	'22:27':	3583,	' 11:45':	3917,	' 22:53':	3576,
363 '22:49':	3474,	'23:41':	3059,	' 07:49':	1747,	' 01:02 ' :	2842,
364 '08:23':	2074,	'11:46':	3884,	' 04:06 ' :	1252,	' 02:29 ' :	1659,
365 '03:31':	1224,	; 01:00:	2395,	' 02:49 ' :	1527,	' 06:52 ' :	1397,
366 '01:29':	2147,	'12:00':	3802,	' 20:34':	3540,	' 04:12 ' :	1234,
367 '01:53':	1899,	' 05:17':	1219,	' 02:18 ' :	1741,	' 06:02 ' :	1307,
368 '04:55':	1280,	'04:54':	1271,	' 04:49 ' :	1211,	' 23:12 ' :	3180,
369 '11:44':	4083,	' 07:04':	1444,	' 05:02 ' :	1195,	' 06:15 ' :	1319,
370 '03:58':	1257,	' 16:17':	4697,	'22:23':	3608,	' 02:56 ' :	1399,
371 '03:28':	1389,	'11:31':	3921,	' 21:35':	3537,	' 05:48 ' :	1419,
372 '05:42':	1265,	' 02:54':	1552,	' 01:18 ' :	2170,	' 05:03 ' :	1267,
373 '07:39':	1702,	' 21:54':	3550,	' 04:26 ' :	1266,	' 00:37':	2649,
374 '22:04':	3640,	'06:56':	1402,	' 03:13 ' :	1420,	' 06:34':	1484,
375 '04:17':	1299,	' 02:44':	1450,	' 10:03 ' :	3290,	' 04:46 ' :	1244,
376 '05:41':	1210,	' 02:34':	1557,	' 20:51':	3645,	'22:32':	3403,
377 '04:32':	1248,	'03:08':	1553,	' 21:29 ' :	3504,	' 02:13 ' :	1960,
378 '11:48':	3818,	'22:56':	3193,	' 06:42 ' :	1364,	' 03:03 ' :	1553,
379 '08:26':	2060,	'04:22':	1246,	'21:25':		' 03:29 ' :	1303,
380 '22:45':	3415,	' 12:03':	3524,	'23:45':	3078,	' 07:48':	1785,
381 '06:35':	1421,	'23:17':	3212,	'03:46':	1315,	'04:25':	1165,

```
382 '03:50': 1249, '02:35': 1614, '10:30': 3538, '07:26': 1602,
383 '00:59': 2436, '04:16': 1236, '04:28': 1176, '07:11': 1472,
384 '01:43': 2073, '07:03': 1386, '03:20': 1355, '01:20': 2165,
385 '04:37': 1342, '02:57': 1380, '00:14': 2695, '07:46': 1822,}
386
387 # Uses list comprehension to create the 24 hours (0-23).
_{388} hrs = [str(x) for x in range(0,24)]
389 # Uses list comprehension to create the minutes in an hour (0-59)
390 # Uses .zfill() to create (00, 01, 02, 03 instead of 0,1,2,3)
391 min = [str(x).zfill(2) for x in range (0,60)]
392 # list called "times" that will consist of all the times through a day:
      "00.00-23.59".
393 times = []
394 # for-loop that iterates through the hrs and min lists and creates the
      times for a day.
395 for x in hrs:
      for y in min:
396
           times.append(f''{x}.{y}'')
397
398
399 # Creates a new dictionary for processing times that will replace ":" with
       400 new_dictionary = {}
401
402 for x,y in result.items():
      x = x.replace(":",".")
403
      new_dictionary[x] = y
404
405
406
407 # Sorting the dictionary based on the time
408 dictionary_items = new_dictionary.items()
409 sorted_items = sorted(dictionary_items)
410 sorted_items = list(sorted_items)
411
412 # A new dictionary that will contain the tests throughout a day based on
      the statistics from the GitHub dataset
413 chance_dictionary = {}
414
415 # Creates a list based on the times from the dictionary of sorted items
416 population = [x for x, y in sorted_items]
417 # Creates a list of the different "weights" that are used to pick a time
      later
418 weights = [y for x, y in sorted_items]
419 # Iteration used to create the tests. The bigger number to iterate more
      often tests will come.
420 for _ in range(round(len(population))//2):
      # Picks a time of day to send a test, based on the weights (can also
421
      be known as "odds")
      # "population" refer to the list containing each timestamp
422
       # "weights" refer to the list containing the count for each timestamp
423
```

```
pick = random.choices(population,weights)[0]
424
       # Checks if the time is already in the dictionary, if so then add a
425
      random character to have it be unique
       if pick in chance_dictionary:
426
           # write "pass" for ignoring duplicates
427
           while pick in chance_dictionary:
428
               # While the test is in the dictionary, a new character will be
429
       added to make it unique.
               new_pick = pick[:5] + string.ascii_uppercase[random.randint(0,
430
      len(string.ascii_uppercase)-1)]
431
               break
           # Adds the characteristics on how long the test will take.
432
           chance_dictionary[new_pick] = random.choices([1,2],[8,2])[0] +
433
      round(random.uniform(0.00,0.59),2)
       else:
434
           # If the time is not already in the dictionary, it will be added.
435
           chance_dictionary[pick] = random.choices([1,2],[8,2])[0] + round(
436
      random.uniform(0.00,0.59),2)
437
438 dictionary_keys = chance_dictionary.keys()
439 dictionary_keys = [x[0:5] for x in dictionary_keys]
440
441 # Uncomment this for-loop to ensure that there will always be at least 1
      test for every minute
442 # for time in times:
       if time.zfill(5) not in dictionary_keys:
443 \ #
444~\#
                chance_dictionary[time.zfill(5)] = random.choices
      ([1,2],[8,2])[0] + round(random.uniform(0.00,0.59),2)
445 \ #
       else:
446~\#
            pass
447
448 # Sorts the test based on time.
449 chance_dictionary_items = chance_dictionary.items()
450 new_sort_items = sorted(chance_dictionary_items)
451
452 # Creating the CSV FILE
453 with open("TESTFILENAME.csv", "w") as csv_file:
       test_writer = csv.writer(csv_file, quotechar='"', quoting=csv.
454
      QUOTE_MINIMAL)
       counter = 1
455
       test_writer.writerow(["TestID", "ScheduledTime", "TestTime"])
456
       for x,y in new_sort_items:
457
           test_writer.writerow(["T"+str(counter).zfill(4),x[0:5],str(round(y))
458
       ,2)).ljust(4,"0")])
           counter += 1
459
```

Listing A.2: generate tests.py: Creates test input files

A.1.3 Generator for worker input files

```
1 import random
2 import csv
3 import string
4
5 # Creating the dictionaries that will consist of our workers
6 worker_initial_dictionary = {}
7 worker_additional_dictionary = {}
9 # Worker counter used for creating worker ID's
10 worker_increment = 1
11
12 # Variable that consists of the number of workers that the taskfile should
      start of with
13 initial_workers = 0
14
15 # iterates over the range of the integer set in the initial_workers and
     uses enumerate
16 for _ in range(0, initial_workers):
      # Creates an entry in the dictionary that consists of the WorkID as
17
      key: "W0001" and values for StartTimeAvailable, PerformanceRating and
      AvailabilityTime
      worker_initial_dictionary["W"+str(worker_increment).zfill(4)] = {
18
                                                        "StartTimeAvailable":'
19
      0.00',
                                                        "PerformanceRating":
20
      round(random.uniform(0.6,2),2),
                                                        "AvailabilityTime":
21
      round(random.uniform(20,180),2),
                                                        }
22
      # incrementing with 1
^{23}
      worker_increment += 1
24
25
26 # Uses list comprehension to create the 24 hours (0-23).
_{27} hrs = [str(x) for x in range(0,24)]
28 # Uses list comprehension to create the minutes in an hour (0-59)
29 # Uses .zfill() to create (00, 01, 02, 03 instead of 0,1,2,3)
_{30} min = [str(x).zfill(2) for x in range (0,60)]
31 # list called "times" that will consist of all the times through a day:
      "00.00-23.59".
32 times = []
33 # for-loop that iterates through the hrs and min lists and creates the
      times for a day.
34 for x in hrs:
35
     for y in min:
36
         times.append(f"{x}.{y}")
37
```

```
38 # Iterates through each time of the day
39 for time in times:
      # This try/except block will introduce an increase in workers at the
40
      hours 10:00-18:59
      # Uncomment for creating files based on this.
41
      # try:
42
            new_time = int(time[0:2])
      #
43
            if new_time in range(10,18):
      #
44
      #
                for additionals in range(1,6):
45
                     chance = random.randint(0,8)
      #
46
                     if chance == 1:
47
      #
      #
                         worker_additional_dictionary["W"+str(
48
      worker_increment).zfill(4)] = {
                             "StartTimeAvailable": time,
      #
49
                             "PerformanceRating":round(random.uniform(0.6,2)
      #
50
      ,2),
      #
                             "AvailabilityTime":round(random.uniform(20,180)
51
      ,2),
                         }
52
      #
      #
                         worker_increment += 1
53
      # except:
54
55
      #
            pass
      # This for-loop add additional workers (additional as in addition to
56
      the initial workers) to the dictionary called "
      worker_additional_dictionary"
      for additionals in range(1,5):
57
          chance = random.randint(0,34)
58
          if chance == 1 or chance == 6:
59
              worker_additional_dictionary["W"+str(worker_increment).zfill
60
      (4)] = \{
                   "StartTimeAvailable": time,
61
                   "PerformanceRating":round(random.uniform(0.6,2),2),
62
                   "AvailabilityTime":round(random.uniform(20,180),2),
63
              }
64
              worker_increment += 1
65
66
67 # Combines the two dictoinaries
68 total_workers_dictionary = {**worker_initial_dictionary, **
      worker_additional_dictionary}
69
70 # Defines the filename for the CSV file
71 csv_filename = "WORKERFILENAME.csv"
72 # Generating the CSV FILE
73 with open(csv_filename,"w") as csv_file:
      worker_writer = csv.writer(csv_file, quotechar='"', quoting=csv.
74
      QUOTE_MINIMAL)
      worker_writer.writerow(["WorkerID","StartTimeAvailable","
75
      PerformanceRating", "AvailabilityTime"])
      for x,y in total_workers_dictionary.items():
76
```

worker_writer.writerow([x,y["StartTimeAvailable"],y[" PerformanceRating"],y["AvailabilityTime"]])

Listing A.3: generate workers.py: Creates the worker input

A.2 Main simulation file

```
1 import numpy as np
2 from latex_files.graph_gen import linear_graph, linear_graph_double_y
3 import ast
4 # Classes
5 class Worker:
      instances = []
6
      def __init__(self, WorkerID, StartTimeAvailable, PerformanceRating,
7
      AvailabilityTime):
          self.WorkerID = str(WorkerID)
8
9
          self.StartTimeAvailable = float(StartTimeAvailable)
          self.InitialStartTime = float(StartTimeAvailable)
10
          self.PerformanceRating = float(PerformanceRating)
11
          self.AvailabilityTime = float(AvailabilityTime)
12
          self.JoinedTesting = False
13
          self.AssignedWork = False
14
          self.AssignedTest = None
15
          self.CompletedTests = []
16
          self.NumberOfCompletedTests = len(self.CompletedTests)
17
          self.WorkingIntervals = []
18
          self.WaitingIntervals = []
19
          self.checkpoint = None
20
          self.TotalWaitTime = sum(self.WaitingIntervals)
21
          self.instances.append(self)
22
23
      def __str__(self):
24
25
          return f"{self.WorkerID}: \n\tStarted working: {self.
      StartTimeAvailable}\
          \n\tPerformance: {round(self.PerformanceRating*100)}%\
26
          \n\tRemaining availability: {self.AvailabilityTime}\
27
          \n\tJoinedTesting: {'Yes' if self.JoinedTesting else 'No'}\
28
          \n\tAvailable to work at current time ({time}): {'Yes' if self.
29
      AssignedWork == False and self.JoinedTesting else 'No'}\
          \n\tAssigned Test: {self.AssignedTest if self.AssignedTest else '
30
      None'}\
          \n\tCompleted Tests: {' '.join(self.CompletedTests)}\
31
          \n\tCompleted Tests count: {str(len(self.CompletedTests))}\
32
33
          \n\tWaiting Time List: {self.WaitingIntervals}\
          \n\tTest statistics:\
34
```

77

```
\n\tTest distribution min: {str(np.min(self.WorkingIntervals)) if
35
     bool(self.WorkingIntervals) else 'N/A' } \
          \n\tTest distribution median: {str(np.median(self.WorkingIntervals
36
     )) if bool(self.WorkingIntervals) else 'N/A'}\
          \n\tTest distribution mean: {str(np.mean(self.WorkingIntervals))
37
     if bool(self.WorkingIntervals) else 'N/A'}\
          \n\tTest distribution 75%: {str(np.percentile(self.
38
     WorkingIntervals,75)) if bool(self.WorkingIntervals) else 'N/A'}
          \n\tTest distribution max: {str(np.max(self.WorkingIntervals)) if
39
     bool(self.WorkingIntervals) else 'N/A'}\
          \n\tWait interval statistics:\
40
          \n\tWait distribution min: {str(np.min(self.WaitingIntervals)) if
41
     bool(self.WaitingIntervals) else 'N/A'}\
          \n\tWait distribution median: {str(np.median(self.WaitingIntervals
42
     )) if bool(self.WaitingIntervals) else 'N/A'}\
          \n\tWait distribution mean: {str(np.mean(self.WaitingIntervals))
43
     if bool(self.WaitingIntervals) else 'N/A'}\
          \n\tWait distribution 75%: {str(np.percentile(self.
44
     WaitingIntervals,75)) if bool(self.WaitingIntervals) else 'N/A'}
          \n\tWait distribution max: {str(np.max(self.WaitingIntervals)) if
45
     bool(self.WaitingIntervals) else 'N/A'}\
          \n\tTotal wait time: {self.TotalWaitTime}\
46
          \n\tWorker utilization: {self.Utilization}%"
47
      def assigned_test(self, test):
48
          self.AssignedWork = True
49
          self.AssignedTest = test_overview_dictionary[test].TestID
50
          original_testtime = test_overview_dictionary[test].TestTime
51
          number_after_performance = round((float(original_testtime)*60) /
52
     float(self.PerformanceRating))
          test_overview_dictionary[test].TestTime = float(f"{
53
     number_after_performance//60}.{number_after_performance%60}")
          # append the time differenece since we finished the last test to "
54
     WaitingIntervals"
          self.WaitingIntervals.append(self.time_diff(time,self.checkpoint))
55
          # set new checkpoint
56
          self.checkpoint = time
57
58
      def process(self):
59
          self.AvailabilityTime -= 1
60
          if self.AvailabilityTime > 0:
61
              if self.AssignedTest:
62
                  test_overview_dictionary[self.AssignedTest].TestTime -= 1
63
                  if test_overview_dictionary[self.AssignedTest].TestTime <</pre>
64
     1 and test_overview_dictionary[self.AssignedTest].TestTime != 0:
                      test_overview_dictionary[self.AssignedTest].TestTime =
65
      1
                  elif test_overview_dictionary[self.AssignedTest].TestTime
66
     == 0:
```

```
self.CompletedTests.append(test_overview_dictionary[
67
      self.AssignedTest].TestID)
                        self.AssignedTest = None
68
                        self.WorkingIntervals.append(self.time_diff(time,self.
69
       checkpoint))
                        self.checkpoint = time
70
               else:
71
                    self.AssignedWork = False
72
                    self.TotalWaitTime += 1
73
74
           else:
               finished_workers.append(self)
75
               self.JoinedTesting = False
76
77
       def time_diff(self, original, subtraction):
78
           orig = original.split(".")
79
           subs = subtraction.split(".")
80
           origminutes = int(orig[0]) * 60 + int(orig[1])
81
           subminutes = int(subs[0]) * 60 + int(subs[1])
82
83
           return int(origminutes - subminutes)
84
85 class Test:
       instances = []
86
       def __init__(self, TestID, ScheduledTime, TestTime):
87
           self.TestID = str(TestID)
88
           self.ScheduledTime = ScheduledTime
89
           self.InitialScheduledTime = ScheduledTime
90
           self.TestTime = float(TestTime)
91
           self.AssignedWorker = "N/A"
92
           self.WaitTime = None
93
           self.instances.append(self)
^{94}
       def __str__(self):
95
           return f"{self.TestID}: \n\tScheduled for testing at: {self.
96
      ScheduledTime}
           \n\tTime it takes for testing: {self.TestTime}\
97
           \n\tAssigned to worker: {self.AssignedWorker}\
98
           \n\tThis test waited in queue for: {self.WaitTime} iterations"
99
100
       def assigned_worker(self, worker):
101
           self.AssignedWorker = worker
102
           if float(time) == float(self.ScheduledTime):
103
               self.WaitTime = 0
104
           else:
105
               self.WaitTime = self.time_diff(time,self.ScheduledTime)
106
107
       def time_diff(self, original, subtraction):
108
           orig = original.split(".")
109
           subs = subtraction.split(".")
110
           origminutes = int(orig[0]) * 60 + int(orig[1])
111
           subminutes = int(subs[0]) * 60 + int(subs[1])
112
```

```
return int(origminutes - subminutes)
113
114
115 ### PRE-PHASE ###
116
117 # Uses list comprehension to create the 24 hours (0-23).
118 hrs = [str(x).zfill(2) \text{ for } x \text{ in } range(0,24)]
119
120 # Uses list comprehension to create the minutes in an hour (0-59)
121 # Uses .zfill() to create (00, 01, 02, 03 instead of 0,1,2,3)
122 \min = [str(x).zfill(2) \text{ for } x \text{ in range } (0,60)]
123
124 # list called "times" that will consist of all the times through a day:
      "00.00-23.59".
125 times = []
126
127 # for-loop that iterates through the hrs and min lists and creates the
      times for a day.
128 for x in hrs:
129
      for y in min:
           times.append(f"{x}.{y}")
130
131
132 # Reference the workerlist file to simulate worker scenario
133 worker_scenario = "workerlists/5initial_increase_in_worktimes.csv"
134
135 # Reference the tasklist file to simulate test scenario
136 test_scenario = "tasklists/continuous_moderate.csv"
137
138 # NOTE:
139 # The following dictionaries are marked as "queues".
140 # These dictionaries will contain ALL tests and ALL workers in the
      scenario.
141 # The dictionaries will be checked by our simulation to know when a worker
        is available or when a test is submitted.
142 # Creates a dictionary that will later be used to store the worker objects
        as values where the key is the workerID
143 worker_dictionary_queue = {}
144 with open(worker_scenario,"r") as csv_file:
       for row in csv_file.readlines()[1:]:
145
           split_row = row.split(",")
146
           worker_dictionary_queue[split_row[0]] = Worker(split_row[0],
147
      split_row[1], split_row[2], split_row[3])
148
149 test_dictionary_queue = {}
150 with open(test_scenario,"r") as csv_file:
       for row in csv_file.readlines()[1:]:
151
           split_row = row.split(",")
152
           test_dictionary_queue[split_row[0]] = Test(split_row[0],split_row
153
       [1],split_row[2])
154
```

```
155 # Creates duplicate dictoinaries that will be used later for referencing
156 worker_overview_dictionary = dict(worker_dictionary_queue)
157 test_overview_dictionary = dict(test_dictionary_queue)
158
159 # Following lists will contain the active tests and workers at any given
      time based on the times in the tasklists and workerlists
160 active_worker_list = []
161 active_test_list = []
162
163 finished_workers = []
164 finished_tests = []
165
166 # This dictionary will contain information regarding the completed tests.
167 completed_test_dictionary = {}
168
169 # Dictionary to keep track on how many available workers there are per
      time (Not working)
170 available_workers_per_round = {}
171
172 # Dictionary to keep track on the amount of tests submitted each round
173 total_tests_per_time = {}
174
175 for x,y in test_overview_dictionary.items():
      test_time = y.ScheduledTime
176
      if test_time in total_tests_per_time:
177
           total_tests_per_time[test_time] += 1
178
179
      else:
          total_tests_per_time[test_time] = 1
180
181
182 # Dictionary to keep track of the remaining tests per time
183 remaining_tests_per_time = {}
184
185 ### END OF PRE-PHASE ###
186
187 ### SIMULATION ###
188 ### |
189 # Iterates through the times of the day
190 for time in times:
       ### --> SIMULATION "WORK ON TEST"-PHASE: Every worker will process the
191
       tests they are assigned by "1" (representing 1 minute)
       for key,value in worker_overview_dictionary.items():
192
           if value.JoinedTesting == True:
193
               value.process()
194
       ### --> SIMULATION "CHECK"-PHASE: Checking the queue dictionaries and
195
      adding them in the active lists <-- ###
       # Adds the tests scheduled for current time to the "active_tests_queue
196
      " from the "test_dictionary_queue"
      tests_to_remove_from_queue = []
197
       for key, value in test_dictionary_queue.items():
198
```

```
if len(str(value.ScheduledTime))==3:
199
               value.ScheduledTime = "0"+(str(value.ScheduledTime)+"0")
200
           if str(value.ScheduledTime).zfill(5) == time:
201
               active_test_list.append(value)
202
               tests_to_remove_from_queue.append(key)
203
       for item in tests_to_remove_from_queue:
204
           del test_dictionary_queue[item]
205
       # Adds the available workers to active workers list from the
206
      worker_dictionary_queue.
      workers_to_remove_from_queue = []
207
208
       for key, value in worker_dictionary_queue.items():
           if len(str(value.StartTimeAvailable)) < 5:</pre>
209
               if len(str(value.StartTimeAvailable))==3:
210
                   value.StartTimeAvailable = "0"+(str(value.
211
      StartTimeAvailable)+"0")
               elif len(str(value.StartTimeAvailable).split(".")[-1]) == 2:
212
                   value.StartTimeAvailable = str(value.StartTimeAvailable).
213
      zfill(5)
214
               elif len(str(value.StartTimeAvailable).split(".")[-1]) == 1:
                   value.StartTimeAvailable = str(value.StartTimeAvailable).
215
      ljust(5,"0")
           if str(value.StartTimeAvailable).zfill(5) == time:
216
               active_worker_list.append(value)
217
               worker_overview_dictionary[key].JoinedTesting = True
218
               worker_overview_dictionary[key].checkpoint = time
219
220
               workers_to_remove_from_queue.append(key)
       for item in workers_to_remove_from_queue:
221
           del worker_dictionary_queue[item]
222
       ### END "CHECK"-PHASE ###
223
       ### |
224
      ### --> SIMULATION "CHECK FOR AVAILABLE WORKERS PHASE": Checks for
225
      workers that are finished working and are now available <-- ###
       for key, value in worker_overview_dictionary.items():
226
           if value.AssignedWork == False and value not in active_worker_list
227
       and value.JoinedTesting == True:
               active_worker_list.append(value)
228
       ### END "CHECK FOR AVAILABLE WORKERS PHASE" ###
229
       ### |
230
       ### --> Check for available workers and appends this value, with the
231
      time, in the dictoinary "available_workers_per_round"
       available_workers_per_round[time] = len(active_worker_list)
232
       ### --> End <--
233
       ### |
234
       ### --> SIMULATION "DISTRIBUTE TESTS PHASE": Goes over the tests in
235
      the active_test_list and distributes these tests among the available
      workers <-- ###
       for test in active_test_list:
236
           try:
237
               assigned_worker = active_worker_list[0]
238
```

```
assigned_worker.assigned_test(test.TestID)
239
               test.assigned_worker(assigned_worker.WorkerID)
240
               active_worker_list.remove(assigned_worker)
241
               active_test_list.remove(test)
242
           except:
243
244
               pass
       ### END SIMULATION "DISTRIBUTE TESTS PHASE"
245
       ### |
246
       ### --> Check the remaining tests and adds this information to the
247
      dictionary "remaining_tests_per_time"
       remaining_tests_per_time[time] = len(active_test_list)
248
249 #### END SIMULATION ####
250
251 #### START POST-PHASE ####
252
253 # Creates a dictionary to contain the different test wait times for each
      test
254 # KEY: The scheduled time a test was made availble
255 # VALUE: The time that test had to wait before it was given a worker
256 test_wait_time_stats = {}
257 for x in Test.instances:
       if x.WaitTime == None:
258
259
           pass
      else:
260
           # If the scheduled test time already exists, then we will add
261
      these values together and divide with 2
           if x.InitialScheduledTime in test_wait_time_stats:
262
               test_wait_time_stats[x.InitialScheduledTime] = (
263
      test_wait_time_stats[x.InitialScheduledTime] + x.WaitTime)/2
264
           else:
               test_wait_time_stats[x.InitialScheduledTime] = x.WaitTime
265
266
267 # Uses list comprehension to round the float number of test wait time to
      an integer.
268 test_wait_time_stats = {x: round(y) for x,y in test_wait_time_stats.items
       ()}
269
270 # Creates a list and appends values that will represent the test time for
      each timestamp
271 test_wait_stats = []
272 test_wait_stats.append([(x,y) for x,y in test_wait_time_stats.items()])
273
274 # String that will be used together with the different with() functions to
       create similar filenames
275 filename = "worker"
276
277 # Calculates the utilisation for each worker and adds this value to the
      workers
278 worker_utilization = []
```

```
279 for x in Worker.instances:
      util = round(sum(x.WorkingIntervals)/(sum(x.WorkingIntervals)+sum(x.
280
      WaitingIntervals))*100) if sum(x.WorkingIntervals)+sum(x.
      WaitingIntervals) > 0 else 0
      worker_utilization.append((x.WorkerID,round(util)))
281
      worker_overview_dictionary[x.WorkerID].Utilization = util
282
283
284 # Creates a list that will contain the utilisation stats
285 utilization_stats = []
286 for x in Worker.instances:
      utilization_stats.append(x.Utilization)
287
288
289 # Dictionary that will count the appearence of each percentage
290 utilization_count = {}
291 for x in utilization_stats:
      if x in utilization_count:
292
           utilization_count[x] += 1
293
294
      else:
295
           utilization_count[x] = 1
296
297 # Sorts the dictionary based on the percentage
298 dictionary_items = utilization_count.items()
299 sorted_items = sorted(dictionary_items)
300 sorted_items = list(sorted_items)
301
302 # Creates a list that will contain the different values of percentage and
      count in a toople
303 utilization_stats = []
304 utilization_stats.append([(x,y) for x,y in sorted_items])
305
306 # Creates lists that will contain tooples that each contain data on the
      count based on the dictionaries defined in the pre-phase
307 plot_graphs_available_tests = []
308 plot_graphs_available_tests.append([(x,y) for x,y in total_tests_per_time.
      items()])
309 plot_graphs_available_tests.append([(x,y) for x,y in
      available_workers_per_round.items()])
310 plot_graphs_available_tests.append([(x,y) for x,y in
      remaining_tests_per_time.items()])
311
312 # Prints the data for each worker and each test
313 for worker in Worker.instances:
314
      print(worker)
315
316 for test in Test.instances:
      print(test)
317
318
319 ### STOP POST-PHASE ###
320
```

```
321 ### File outputs ###
322 #
323 # Uncomment the I/O operations for printing data on test wait time,
      availability data and utilization
324 #
325 #with open("output_"+filename+"_testwait","w") as file:
       file.write(str(test_wait_stats))
326 #
327
328 #with open("output_"+filename+"_availability","w") as file:
       file.write(str(plot_graphs_available_tests))
329 #
330
331 #with open("output_"+filename+"_utilization","w") as file:
332 #
       file.write(str(utilization_stats))
333
334 ### Line graph for LaTeX generator ###
335 #
_{336} # Uncomment this I/O operation to pass data from a file generated above
      into the a graph function.
337 # The graph function "linear_graph_double_y" only works with _availability
       files.
338 #
339 #with open("output_worker_availability","r") as file:
340 #
       linear_graph_double_y(ast.literal_eval(file.readlines()[0]))
```

Listing A.4: *main.py*: Runs the simulation based on two input files; worker scenario and test scenario

Appendix B

LaTeX Graph Generator file

This file contains two functions that were used to create the line graph in this thesis. One graph would create a standard line graph with plots along one x-and one y-axis. The second function would create plots with two y-axis and one x-axis.

```
1 def linear_graph(plots):
\mathbf{2}
      print("-----\n")
      print(r"\begin{figure}[H]")
3
      print(r"
                  \centering")
\mathbf{4}
     print(r"
                  \begin{tikzpicture}")
\mathbf{5}
     print(r" \begin{axis}[")
6
     print(r" scale only axis,")
7
     print(f" height=5cm,")
8
      print(r"
                  width=\textwidth-1cm,")
9
     print(r"
                  legend style={at={(0.3, 0.9), anchor=west}},")
10
     print(r"
                  /pgf/number format/.cd,")
11
      print(r"
                 use comma,")
12
      print(r"
                  1000 sep={}]")
13
      print(r"
                  ]")
14
15
      plot_names = ["One", "Two", "Three", "Four", "Five", "Six"]
16
      colors = ["red", "blue", "orange", "green", "black"]
17
18
      for iteration,plot in enumerate(plots):
19
          print(r"
                       \addplot[mark=none,thick,"+colors[iteration]+"]
20
      coordinates {")
          for x,y in plot:
21
               print(f"
                                ({x},{y})")
^{22}
                      };")
          print(r"
23
                       \addlegendentry{"+plot_names[iteration]+"}")
          print(r"
24
```

```
25
      print(r"
                   \end{axis}")
26
      print(r"
                   \end{tikzpicture}")
27
                   \caption{Placeholder text}")
      print(r"
28
                   \label{Placeholder text}")
      print(r"
29
      print(r"\end{figure}")
30
31
32
33 def linear_graph_double_y(plots):
      plots_one = plots[0:2]
34
      plots_two = plots[2:]
35
      plot_names = ["One", "Two", "Three", "Four", "Five"]
36
      colors = ["red", "blue", "orange", "green", "black"]
37
      used_plot_names = []
38
      print("-----\n")
39
      print(r"\begin{figure}[H]")
40
      print(r"
                   \centering")
41
      print(r"
                   \begin{tikzpicture}")
42
43
      print(r"
                   \pgfplotsset{")
      print(r"
                   scale only axis,")
44
      print(f"
                   height=5cm,")
45
      print(r"
                   width=\textwidth-2.7cm,")
46
      print(r"
                   legend style={at={(0.3,0.9),anchor=west}},")
47
      print(r"
                   /pgf/number format/.cd,")
48
      print(r"
                   use comma,")
49
      print(r"
                   1000 sep={}]")
50
      print(r"
                   scaled x ticks=base 10:3,")
51
      print(r"
                   }")
52
      print(r"
                   \begin{axis}[")
53
                   axis y line*=left,")
      print(r"
54
      # print(r"
                     ymin=0, ymax=100,")
55
                   xlabel=x-axis,")
      print(r"
56
      print(r"
                   ylabel=y-axis 1,")
57
      print(r"
                   1")
58
59
      counter = 0
60
61
      for plot in plots_one:
62
          print(r"
                       \addplot[",colors[counter],']')
63
          print(r"
                       coordinates{")
64
          for x,y in plot:
65
               print(f"
                            ({x},{y})")
66
          print(r"
                       }; \label"+"{"+plot_names[counter]+'}')
67
          used_plot_names.append(plot_names[counter])
68
          counter += 1
69
70
      print(r"
                   \end{axis}")
71
      print(r"
                   \begin{axis}[")
72
      print(r"
                   axis y line*=right,")
73
```

```
print(r"
                  axis x line=none,")
74
      # print(r"
                    ymin=0, ymax=100,")
75
      print(r"
                  ylabel=y-axis 2")
76
      print(r"
                  ]")
77
      for iteration,name in enumerate(used_plot_names):
78
                     \addlegendimage{/pgfplots/refstyle=",name,r"}\
          print(r"
79
      addlegendentry{","plot ",iteration,"}")
80
      for plot in plots_two:
81
          print(r"
                     \addplot[",colors[counter],']')
82
          print(r"
83
                      coordinates{")
          for x,y in plot:
84
              print(f"
                           ({x},{y})")
85
                     }; \addlegendentry"+"{"+plot_names[counter]+'}')
          print(r"
86
          used_plot_names.append(plot_names[counter])
87
          counter += 1
88
      print(r"
                  \end{axis}")
89
90
      print(r"
                   \end{tikzpicture}")
^{91}
      print(r"
                   \caption{Placeholder text}")
92
                  \label{Placeholder text}")
      print(r"
93
      print("\end{figure}\n")
^{94}
```

Listing B.1: graph_gen.py: Generates LaTeX Line Graphs