ACIT5900

MASTER THESIS

In

Applied Computer and Information Technology

May 2023

Cloud-based Services and Operations

Investigate Performance Testing

Elias Khan

Department of Computer Science

Faculty of Technology , Art and Design

Oslo Metropolitan University

Acknowledgements

I was hired in the cloud department in a company called Atea, and based on what they seemed to find necessary for someone in this position, I chose the appropriate thesis for them , so I want to thank my employer and I look forward to work for them after summer.

I want to thank my supervisor more than anyone as he was always available whenever I needed it, to the point where I would not write any messages to him before work hours as I knew he would answer my calls. My supervisor gave me a lot of confidence and guidance during this project that I had heard others didn't have. He would refer to a previously done master thesis, structure to give ideas on structure, and that saved me from my panic of never having written a thesis before. So I want to thank my supervisor immensely for not just throwing me into the ocean before teaching me how to swim.

I also want to thank my family who supported me through this cheering me on the sidelines and were proud of me for even attempting this thesis.

Completing this thesis, with the medical circumstances I had, made this physically probably the hardest thing to complete I have ever done under the circumstances. And I also want to acknowledge myself for putting in the amount of effort I could muster as I did not believe I could finish this project.

Abstract

In the modern-day workforce today, there is one central thing that is as important as it has ever been. And that is the word productivity. Performance is very important in modern's society as the world is today far more corporately competitive than before as there are a lot of options, so it is important for the companies to make sure that they can test how they perform. In the modern day, a lot of software development is automated, and a popular automation tool is CI/CD pipelines , that have very popular tools for that job, like Gitlab or Jenkins, but how does it benefit anyone to have performance tests within these pipelines? Is it just a waste of space or are they useful? Could they be just tests of performance or is it more under the hood? This paper will tackle both performance tests and the dynamic performance test. Which will focus on history-based performance testing.

TABLE OF CONTENTS

Introduction……………………………………………………………………………………5

- Problem statement…………………………………………………7

Background…………………………………………………………………………………8

- CI/CD and Devops…………………………………………………8
- Briefly about Devops………………………………………………8
- Devops today…………………………………………………9
- Devops and the tools………………………………………………10
- Software Testing…………………………………………………12
- Two testing types……………………………………………………13
- Functional Testing…………………………………………………...13
- Non Functional testing……………………………………………14
- Performance testing……………………………………………….14
- Continuous integration…………………………………………….16
- Continuous Delivery………………………………………………18
- Literature Review…………………………………………………19

Approach ……………………………………………………………………………………28

Research Methods…………………………………………………28

- Project outline……………………………………………………..29
- Pitfalls……………………………………………………………..32

Result……………………………………………………………………………….33

- Framework for performance testing in CI/CD……………………………………………33
- How can we turn this into something dynamic?…………………………………………………….31

Implementation…………………………………………………………………………37

- The Tools……………………………………………………….37
- The Project Implementation…………………………………………………43
- Project Implementation…………………………………………………42

Analysis…………………………………………………………………………………….55

- Output…………………………………………………55
- What was completed …………………………………………………61
- What could have improved Output………………………………………..62

Discussion…………………………………………………………………………………….64

- Looking for background material……………………………………….64
- Finding the right technology……………………………………………65
- Implementation of the prototype…………………………………………66
- The thesis structure ……………………………………………,,,,…66
- Being educated…………………………………………,……….…67

Conclusion…………………………………………………………………………………68

Source…………………………………………………………………………………69

1. Introduction

The world has gone through a lot of different stages throughout it's history in terms of progression and improvement of different inventions. We can look at books[1] for instance; Actual books[1] weren't how we know them today, they were stone tablets, then after the stone tablets came the scrolls or animal skin, and lastly books made of paper were made, or books as we know them today. The next step from this, is not only a progression of books evolution, but physical media in general, blue-rays, video games, books are all becoming victim to digitalization. What this shows is that through trial and error and experimentation, they checked what performed the best over various criteria and went to the next invention. Through the process of digitalization there is a big need for software developers.

Software engineering is arguably one of the most important fields in such a digitalized age. What software engineering essentially is, is solving problems. With the trend of using digitalization of physical media as an example; People want to be able to carry their books everywhere without sacrificing space, thus the software engineers of the world created e-books. Whatever one can digitalize; it seems to be inevitable that it will be. With physical media and the reason why, it becomes digitalized whether it be portability, production costs or convenience puts a lot of stress on the software developers. There are a lot of options out there on a various degree of things that are digital. Using a fitness app for instance, there are several fitness apps available, and which one a consumer is going use is going to use is determined by different factors. But there needs to be a bare minimum that every app must do to even be in the conversation is to be functional. To know if an app is fully functional or not is usually done in the software testing stage of a software development process.

Software testing is the act of seeing if a piece of software works the way that it is supposed to without bugs. An example of how wrong it can go can be in the Heathrow Terminal 5 Opening. This was a baggage handling software in 2008[2], the testing did not factor in how it would be used in real life, so the system failed. Bags did not board the plane as the flights left. Hundreds of flights were cancelled due to it, which damages cost around 50 million dollars and there exists far more disastrous results due to faulty testing, so this should illustrate the importance of it.

What was just described is what is known as a functional test. What the definition is, is that this is about the business requirements. Non- functional testing is done with what the consumers expectations are. Non- functional testing is the umbrella in which performance testing is going under.

Performance tests are tests that involve finding out how a certain workload is performing based on different criteria. These tests often try to find out how fast something is, reliability and size of the application and so on. There are various ways this can be used practically. Response times is one, in relation to things like browsers and network, number of users at the same time, which is very common to see the effects of that performance when there is a Black Friday sale online or perhaps processing times for requests at servers. This is one of the testing methods that can be used in continuous integration/continuous delivery.

The way it can be used in dynamic performance testing is that if a Black Friday sale was done, you can look through what had happened before, what behavior was done before and act according to these behaviors. Setting the database to react to low or high traffic for instance, and make further decisions based on this.

Continuous Integration or Continuous Delivery (Which will now be referred to as CI/CD to make it simpler to understand). Continuous integration is about testing code and Continuous Delivery is about delivering software to production.

There are methods to automate this where the testing and the delivery is done through the process of a pipeline, referred to as a CICD pipeline. There are ways to combine the performance testing and CI/CD pipeline to achieve different results depending on the problem one wishes to solve or investigate.

Problem Statement

There are different aspects one can look closer at with the combination of CI/CD pipelines and performance testing as there are different types of performance testing. Static or dynamic testing[48] are two different types of software testing. The static testing is a testing method in which applications are tested without compiling code. Dynamic testing does execute the code and looks for behaviors[48]. Which is an aspect that could use more insight. The problem statement that will be examined is as follows; *Investigate the Implementation of dynamic performance testing in CI/CD pipeline.*

The word *investigate* in this context means that an experiment will be conducted, and the results must be looked over to see what conclusions can be drawn. *Implementation* covers what tools or concepts are needed in order to be able to be able to *investigate* anything at all. *Dynamic performance testing* is going to be in this context about looking for certain behaviors of scripts that will be executed in a *CI/CD pipeline* which is going to be in this context, an automated pipeline that will perform a continuous integration and continuous delivery process.

2. Background

The first section will primarily consist of concepts and terms used in the next sections. Also it will have information on the current state of these concepts in the industry today. The second will be about the research going on in this field and the journey towards that literature such as keywords and search engines used. Finally, we will be going over the relevant technologies for this specific thesis.

CI/CD and DevOps

Continuous integration and continuous delivery are part of DevOps, but before we can describe what they are ,we have to consider DevOps itself.

Briefly about DevOps

In June 2009, there was a conference called the Velocity 09 conference. At this conference John Allspaw and Paul Hammond where they held a segment called “10 Deploys A Day: Dev and Ops Cooperation at Flickr” where they discussed that developers and operations should cooperate and that not doing this shouldn’t be an option Patrick Debois was not able to attend this conference, but he was watching this remotely. What happened in October the same year is that Debois created his conference inspired by Allspaws and Hammond and titled it DevOpsDays. After the conference Debois hashtagged #DevOps on twitter which is how the term was coined[8]. Debois argued that there was a flaw in agile development[7]. His argument was that agile development did not lead to iterative, fast development, which is essentially what the goal of it was.

DevOps is made out of two words; development and operations. Now the development part should be self explanatory for most as it is a referens to the developers who create the software. Then there is operations which is not quite as clear in terms of what they do, but it is the system administrators, the people that operate business infrastructure. The reason for this collaboration between operations and developers is so the developers who create the software can best develop it so that it can meet the business requirements that the company has[7].

It has to be noted that DevOps is not only a technology, it is rather a methodology. A methodology that usually works like this; plan , code, build, test, release ,deploy , operate, monitor and feedback[15], then the loop resets and starts all over again. What this is supposed to do is to improve the development cycle.

DevOps Today

Today, DevOps is becoming more relevant as its methodology seems to have been employed by more and more companies. Since the *DevOpsDays* conference, it appears that about 74% [8]of all companies seem to have implemented this methodology in some way shape or form, however the 74% is from surveys, the self-reporting is more close to 81%[8].

Small, medium or big business have adopted this, and of the biggest companies that appear to have adopted this, there is; Amazon, Facebook, Netflix and many more have vowed it into their development cycle.

The reason why these companies are using the DevOps methodology is because of the set of benefits that are promised to come with it. For one, due to the nature of two parts of the company working together, there will be a transfer of skills from across the different teams. It improves the software's pipeline in a few different aspects like builds and deployment. Another benefit is that it has fast improvement since the DevOps team gets feedback faster as "DevOps: A Historical Review and Future Works" [9] Summarizes what the benefits are . They believe for it to be the four dimensions of DevOps; Collaboration, Automation, Measurement and Monitoring. So taking these benefits into account, it is not surprising that DevOps by the minimum is said to be used by 74%[8] of companies based on surveys and stretching as far as 84%[8] based on self-reporting for the maximum amount of companies who used it to a smaller or bigger extent.

However with the good must come the bad. DevOps comes with a set of disadvantages. Some of the disadvantages are; the cost of infrastructure, the increase of risk if it is not implemented correctly as well as the fact that it is can be hard to integrate. The latter is due to the fact that some companies have a culture that makes this difficult or that perhaps the organization is very big and it has a high level of complexity in it's software.

DevOps also comes with a variety of tools . Even though DevOps is not a tool in and off itself it does use technology to make it's methodology work. Some of the tools involve software

pipelines, utilizing a cloud and containerization. These tools are also ones that have been mentioned by Gokama and Singh in the form of different commonly used tools in the industry today such as Docker which is a container based tool and Jenkins with which you can create a CI/CD pipeline with such as with a bunch of tools.

DevOps: The Tools

Now the toys or tools of DevOps has a pretty big variety. When an organization is reorganized, the one factor that is incredibly important to keep in mind that determines the success or the failure from a companies previous structure to DevOps are the tools. The tools need to be properly chosen so that they are best fit to reach the goals. There are 9 different types of tools used for 9 different aspects of the organization: collaboration, planning, source control, issue tracking, configuration management, continuous integration, automated testing, deployment and database. What do each of these do[16]?

Collaboration tools are those tools that are made with the benefit to increase fast communication in order to save time and share information. Basically it allows a team to communicate regardless of time and distance. There are some examples of these like Slack for instance[17]. Slack is an application that businesses use to communicate. It is the standard app where one messages one another or change channels which is about organizing people and purpose in one place.

Planning tools are basically there for people involved to help each other plan towards goals and not hide anything from each other. This way capacity and priorities are more visible. Two of the tools that could be used in this scenario is Clarizen or Asana. Asana[18] is a planning tool, that allows you to manage your work as well as organize and work on what's important by making it possible to collaborate while doing so.

Source control tools are tools that basically an update tool specifically for databases. What it does is manage the changes to database, structure or content. An example of a tool like this is DBMaestro [19] which does what was just stated.

Issue tracking tools or troubleshooting give the benefit of issues being visible and responsiveness to these issues. If one is to collaborate , one should unify the internal troubleshooting with the customer generated ones. A tool that is used is Jira[20] which original word came from the Japanese word "Gojira" which means Godzilla as in the character from several Japanese monster

movies. Jira is a tool that companies use as to optimize costs, document flow and as an automation tool.

It is very important to maintain consistency in an IT environment. To make sure that problems don't arise between different environments. What these tools are, are the configuration Management tools. Puppet is the one that comes to mind, and this is a tool that configures servers through automating them.

The team must be able to merge code and get feedback from tests through automated tools throughout the day. These tools are what is referred to as the continuous integration tools. GitLab[22] is a very common one that is easy to learn for users and offers a modern experience for users.

Before you can push the code into the building process ,you need to be able to see if the quality of it is up to par. This is where automated tests come into play. Basically it is about getting feedback fast so that the quality improves and we reach the goal. Katalon[23] is a known tool that has about 1 million users and use used by 100.000 businesses. The selling point of this tool is that the user can focus solely on testing and doesn't need to meet the coding and building requirements for the tests.

To be able to maximize DevOps effectiveness, one needs to make deployments foreseeable, they need to be dependable, and they need to be frequent. Deployment tools or Continuous Delivery tools , main goals are to speed up the process to market, but also minimize the risk. One of the tools for this would be Ms Azure DevOps[24]. Azure is a tool that involves CD and it's philosophy is that of improving collaboration, planning and shipping. Fun fact is, if you type in "Ms Microsoft Azure DevOps" at Finn.no, you will be met with 100 job, meaning that this tool is quite in demand to know your way around.

At last we have the Database DevOps tools. Managing other things like code, tasks , configuration and such is not complete without the database. DB Maestro is again one of the tools that can be used here.

A lot of these tools that are necessary if you want to reorganize your organization if you want to move from the infrastructure you are currently using to DevOps. Do not be surprised if you will find some of the technologies to be used when solving the problem statement.

Software Testing

Checking ones functionality is one of the most important things to do in the software development cycle, this is referred to as the software testing process. Before one can get to the software testing itself, it would be beneficial to learn some facts about terms or inventions that are associated with software testing.

In 1822 Charles Babagge invented the first mechanical computer[31]. Due to system problems and the like , they already around this time had begun the software testing process. Thomas Edison was also part of the software testing story as even though he worked mostly with hardware, he wrote at one point a letter which he had coined the term "bug". The best way to summarize his letter is that he said that in all his inventions there was spotted faults , and after months depending on if they could fix it or not is what determined commercial success or failure. During the second world war once technology could break the secret of enemy communication and after it the terms "computer bug" and "debugging" were coined by Grace Murray Hopper.

In 1951 Joseph Juran wrote the book "Quality Control Handbook" where he emphasized how important it is to have quality software. His 3 main parts of Quality Assessment Management were; quality planning, control and improvement.

With this much history and proof of how valuable this is, software testing today is a very big part of delivering quality solutions. Year by year the IT field is more in need of quality assurance engineers, testing tools and institutes.

"Software Testing" [32] has an explanation of software testing. Briefly it is explained as such; Software testing is about making the code do everything that it's supposed to do and doesn't do anything unintended. There shouldn't be any surprises.

The Singhs also give a pretty good breakdown on what the main objectives are in software testing product. Trying to find every fault in the software. Using a program with the intent of finding faults or defects of the software. And creating certain test cases for finding undiscovered errors.

The Two Software Testing Types

The word software testing makes sense by itself but it is an umbrella to two subcategories of it; Functional testing and Non-Functional testing[3]. These are the two main subcategories of software testing and they have different types of testing that defines them. For instance; Non-functional testing has a subtype of testing called Performance Testing, which is a type of testing that will be covered more in-depth than the ones that will be presented due to it being integral to this thesis.

Functional Testing

Functional testing is explained through the word; it is a test of functionality. Essentially you have a client that presents a set of requirements that needs to be met, then it is the developers job after have created the application, test that the application meets the functional requirements that is expected of from the client. With functional testing there are a lot of different types of tests that will be briefly explained; smoke testing, sanity testing, integration testing, regression testing, localization testing and user acceptance testing. These 6 testing types are not separate from each other as they are part of a process. The first three types of testing is about the initial build of a system and the latter three is more involved with the aftermath and user interface.

In order to make sure the critical functionalities are working, you have to perform Smoke testing, if you do not, then it could cause problems for further testing. Then you have to test functionality; individual components or bugs have to be examined to see if they work, this is Sanity testing. Once the individual components and functions have been tested, it is time to knit it together to see how they are able to work in conjunction with each other, this is where integration testing comes in.

Once the tests for the basic build of the system are done, one has to scour the build fix what the remaining bugs are, this is regression testing. Once the bugs are ironed out, one has to see if the application works in different languages and if the software still functions despite this. If the applications language is English and one switches to Spanish, it shouldn’t compromise the functionality; this type of testing is referred to as language testing. Finally you have the user acceptance testing which is essentially beta-testing. Testing if it is to the user’s satisfaction.

Non-Functional Testing

While Functional testing focuses most of it's attention on whether or not the business requirements are met, Non-functional testing's attention is centered around the performance of the requirements. There are three different types of tests as far as Non-Functional testing goes; the usability test, the security test and the performance test.

Accessibility is arguably one of the most important things that comes with an application can have after the app is functional. This type of testing is Usability testing. This type of testing is about testing the ease of which the user can interact with the interface. And then there is the Security Testing. This type of testing usually is about making sure you don't compromise different aspects of security. This includes authentication, authorization as well as accessibility to data. Finally we have gotten to performance testing.

Performance Testing

As mentioned before, the functional testing must be performed for the software to work properly. After this is done, we go into the non-functional testing method that is relevant for this thesis; performance testing[33].

If one is to use the keywords "Performance Testing" in google scholar one will come across something unrelated in terms of performance testing as things such as performance testing in sports or climate will appear ,so it is important to use some keywords to narrow down what you are looking for. Typing in the keywords "Performance Testing Software" will give you more what you will be looking for in relation to performance testing, and the reason why this is relevant is you will run into a very similar words in performance engineering, with papers like "the future of software performance engineering"[36.1]. Performance engineering [33] is about software attaining a performance goal.

The way performance testing works is that it determines parts of the software's performance. The common test metrics one will find to check the performance of an app is; stability, speed, scalability, responsiveness.

The goal of this testing method to look into performance of different aspects of an app. Some of these include; processing speed, the ability to perform a workload effectively , the most amount of users that can be handled concurrently and so on.

One could have several reasons to use performance testing in their organization. One could for instance compare different systems in order to find out which one performs the best. One could use performance testing to see if a systems performance are meeting the requirements the organization has set for it. Other things may include monitoring traffic or checking if a vendor's advertisement of their software holds true.

The process of performance testing is usually setup like so[34]; First you identify the environment, then can identify the acceptable criteria and at last you define the scenarios.

Before you can start the testing, you need to know all the details of the components you will be using. Familiarity with the environment will make identifying problems simpler. This is what the first step in the performance testing would be; Identifying the environment you will be testing in.

What is considered a success must be clearly defined before you test anything as the criteria will be dependent on each project. This is what identifying the criteria for the performance means.

Now is when you start the testing. You must be able to map out different aspects of the tests, like the different type of users who will be using the app. After the tests are complete you must analyze the results and repeat tests until you have eliminated the issues presented.

There are different types of performance testing[34] , but there are five that are the ones commonly associated with it; Load testing, Stress testing, Scalability testing, Volume testing, Soak testing, examples will be given so one can better understand how this works in practice.

One way you could be testing an applications stability, is load testing[35]. Essentially you want to have a realistic load you want the system to react to. An example could be, if you have an expected load of 1000 users, you can send traffic of 1000 users or less to a website and check how it behaves.

But if you are going above what the expected amount of users would be in order to test the limit of your system, that would be referred to as stress testing. Basically stress testing is applying a load to the website , to see how stable it is by sending traffic above the expected load, to see

what the response the system has to said load. It is not about applying such a load to the point where it negatively affect the system's performance.

This is where Scalability testing comes into play. Scalability testing is taking as big of a load as one can to see what happens to the system. An example would be if you have a website that has a certain amount of traffic with a certain response time, you just keep applying more traffic until the system crashes.

Volume testing is next, this type of testing is about sending large amounts of data to a database in order to test it's capacity. It could for instance sending a set amount of traffic to a webstore to see the behaviors. Unlike load testing that has a set amount, and stress testing that tests a threshold of it, in volume testing ,you will be testing the different loads of data to just to see the results of it, but the amount of is large.

Finally you have Soak testing is more similar to stress testing in a way. The difference is that with soak testing is that you are checking the behavior of the system by applying more traffic over a longer time frame in order check if the system is functioning properly. This is again to test the stability and response of said system. The real world application could be anything that one would leave on for a long time to test the behavior of the system ,for instance having consistent flow of traffic towards a website for longer periods of time.

Continous Integration

"Continuous Integration: Improving software quality and reducing risk[11] explains that if one person does a project and it is simple, it is not as much of an issue to do software integration , this changes immediately though once the project either becomes more complex, or if an additional person decided to join the project. What happens if either of these two components get added into the equation, is that the requirement for integrating things increases due to the fact that now the project is more complicated and now we have to ensure that software components work together, because two different people are working together as opposed to only one. Because of the complexity of the project and the necessity of more manpower, continuous integration is necessary.

There are differences between continuous integration and continuous delivery. Continuous integration is about developers changing and checking their code. Continuous delivery however,

is about delivering code to the necessary environments like the testing environment and the development environment.

Some of the things spoken here about continuous integration can be applied to continuous delivery, which is a later chapter, however now we are going to be sticking to the integration process itself.

Continuous integration is explained well in the paper “Continuous Integration”[12] in that it is a practice where there are members of a software development team that will take their work for the day and integrate it into the system. So for instance; Let’s say you have written your source code in JavaScript using Jenkins as the continuous integration tool, then you use a commit which is a way to push the code through the CI/CD pipeline which in this case would be Jenkins. After it is finished going through the pipeline, where it gets tested and built (more on this later down this thesis), it gets pushed into production. By going through the pipeline, it stops the software from causing integration problems along the way.

So what is continuous integration in relation to DevOps? Before continuous integration was used as the method for delivering apps, what they instead used was DevSecOps, which was what they used to implement security in all phases during development. What happened however was that continuous integration was discovered which meant that development became simpler, faster and with a smaller amount of risk by the software developers.

This brings up some benefits for both the company and the DevOps team itself. For one, what this does is, that it leads to more development frequency and the ability to push fixes to bugs into production faster. Another is probably the most important and the reason they switched from DevSecOps to continuous integration; the speed of how long it takes for code to be committed , all the way through to the production.

Continous Delivery

Continuous Integration, Continuous Delivery and Continuous Deployment[28] are all related. However while Continuous Delivery and Continuous Deployment sound the same and have some similarity. They are not the same, but it still is important to make a distinction among these as when one discusses one of these, the other two tend to come up, so it is important to learn the key differences.

Continuous integration[28] is about developers integrating changes into a source code when needed. It is validated through testing and as well as creating a build. Continuous Delivery is an extension of this. With Continuous Delivery[28], after the continuous integration has done it's job, it is continuous delivery's job to automatically deploy the changes made to testing and production after continuous integration creates the build. Continuous Deployment[28] is also what happens after continuous delivery. If a change is able to get through the stages of a pipeline that you made, it will be released to the customers unless it fails in the pipeline which prevents the deployment. The quickest way to summarize is that continuous integration is part of both continuous delivery and continuous deployment and that continuous deployment is basically identical to continuous delivery aside from the fact that continuous deployment is automatic. For this section we will be examining continuous delivery due to the fact that the problem statement regards CI/CD pipelines.

In 2010 there was a book called *Continuous Delivery* that was written by people that created continuous delivery itself; Jez Humble and David Farley. Their mentality was that while continuous integration was integrating code changes and it was done successfully, what wasn't being done was the software being delivered into production.

In the paper "Continuous Delivery: Huge Benefits, But challenges Too"[30] they state some of the benefits that come with continuous delivery as well as the challenges. The main benefits of continuous delivery are; The time to get to the market, improved customer satisfaction, better quality in products, developing the right products, reliable updates and more productivity and efficiency.

With the benefits presented companies will run into some challenges that they need to tackle. One that has been mentioned previously in the DevOps chapter regarding organizing applies here.

Literature Review

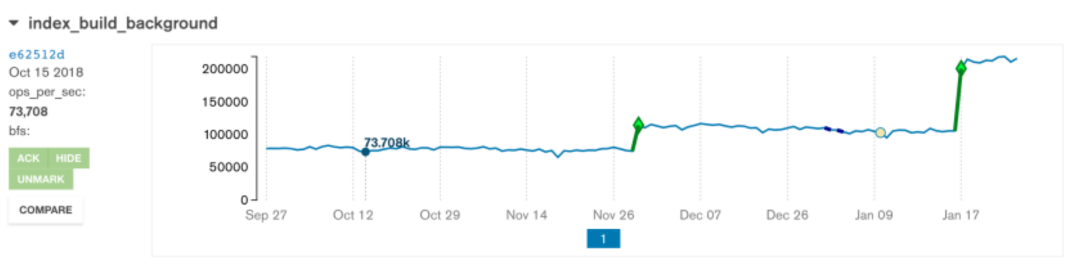
What will be covered here in this section is reviewing literature. Research that has been done in relation to what this thesis is going to cover. A lot of this may be in the same vain, and if something has been done at the exact same way, it is still good to get a second perspective on something.

The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System

This paper[39] was written by one representative of Columbia University and three from MongoDB [ref]. This paper researches into the problem of detecting performance changes within a continuous integration system. More specifically, it is using a software in presence of noise , and from there it will detect changes that are made through a prototype that is presented.

How this was accomplished was by making changes in a repository. After the changes was done, tests started to be performed and the commits that were responsible for the performance regression get identified. The result of their experiment trying to accomplish this was a success. The success entailed that they had a software that could process all the change points where before, a person who even had this as a full-time job couldn't handle it. Now , all this can be processed and presented through graphs, where a person called the "Build Baron", can just scan through the graphs and investigate the changes. There were problems however as they needed to be able to detect when these changes were made, and this will need to be worked on further.

The tools they did use was an algorithm to detect the change points, the algorithm being an R-package that was implemented with Python as well as a repository coded in the same. The results were stored in a MongoDB database, then the change point would be displayed in Jira, with Jira tickets during the change points, where a person would investigate it.



Jira graph displaying the result in the mongoDB.

Including Performance Benchmarks into Continuous Integration to Enable DevOps

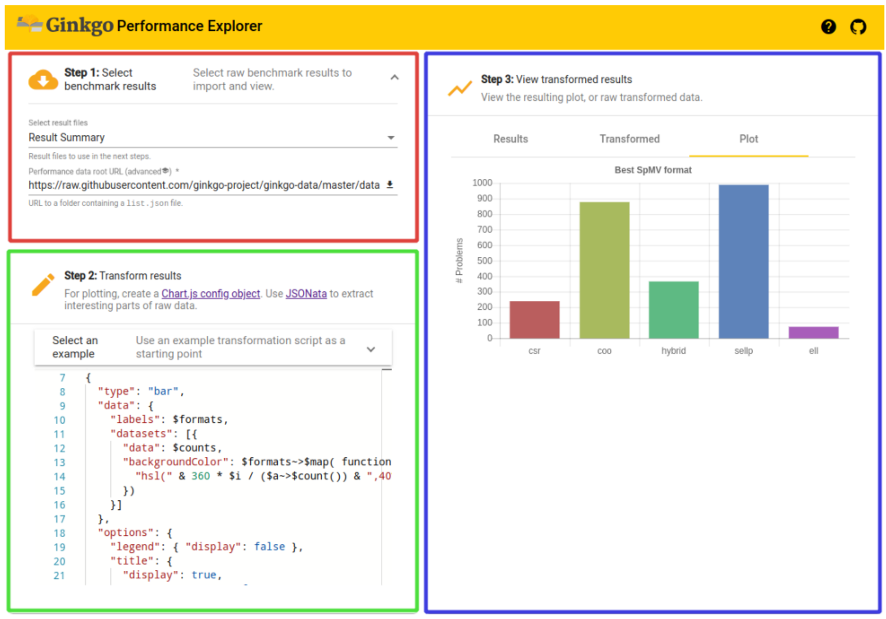
This paper was written by 3 different people from Kiel university; Jan Waller, Nils C.Ehmke, Willhelm Hasselbring[40]. A case study was performed where the three of them tackled a problem of automated benchmarks and how it may be included in continuous integration. They wanted to be able to catch problems earlier in the development cycle ,whether it was errors or performance issues. The result they ended up with was in fact something close to what they envisioned but not quite. Creating performance regression benchmarks to catch problems after software went into production was possible, however that isn't what they wanted. In their paper however, they state that they wanted to be able to implement the benchmarks in the early stages of the development cycle , which means the project was not fully completed.

What they did is that they used a setup with a tool called Kieker. Kieker is a monitoring system that monitors performance. This tool is based on Jenkins which was stated earlier to be the main tool for this case study. They used Kieker to visualize the performance regressions. Then they would use moobench, which is a tool that can immediately detect these performance regressions. This is done in increments nightly.

Towards Continuous Benchmarking: An Automated Performance evaluation Framework for High Performance software

The paper[47] "Towards Continuous Benchmarking: an automated performance evaluation framework for high performance software" had 9 different people that contributed to this all from different universities. These 9, create a prototype using a tool called Gingko. Gingko is a

library for algebra for manycore systems. Manycore systems are essentially processors that handle high performance software.



This figure shows what the prototype looks like

They created an automatic performance evaluator. This evaluator was meant to collect results on high performance computing platforms. What they by their own admission, envisioned a global database that can compare libraries and software components.

Their problem that needed to be solved was an automated framework that enabled automated workflow for performance evaluation of different libraries. The way they went about this, was ,they created a framework for evaluating gingko libraries by it's performance. This process was what they wanted to do automatically. What the Gingko explorer does is analyze data from a repository containing performance results collected to performance platforms.

Audition: a DevOps-oriented service optimization and testing framework for cloud environments

“Audition: a DevOps-oriented service optimization and testing framework for cloud environments” is a paper[49] that executes a prototype with the intention of being a type of Audition, which is what the name of it is. Below is a sample of an output of how the audition system works.

```
<omitted output>
Role: wordpress_web
  2nd place: m1.large with ami-def89fb7
       $0.240 per instance-hour
  1st place: c1.medium with ami-def89fb7
       $0.145 per instance-hour

Creating MLN code for role:
wordpress_web: c1.medium/ami-def89fb7
```

The figure of the output of the audition system.

What the three writers of this paper wanted to do; Gaute Borgenhault, Kyrre Begnum and Paal E. Engelstad, is they wanted to create a system that would work like an audition for a role in a movie. They succeeded as what you see above is the two virtual machine images; m1.large and c1.medium are the representation of the actors in the audition. Basically, it is comparing the two prices of the two images and determined which one was better based on the criteria they decided to set for it, which in this case would be which one’s cheaper.

What they wanted to accomplish is where they felt they could improve their solution. What they wanted as well is the ability to be able to have the actors(images) be paid by the minute, not by the hour. This way the audition wouldn’t be as expensive.

Including Performance Benchmarks into Continuous Integration to Enable DevOps

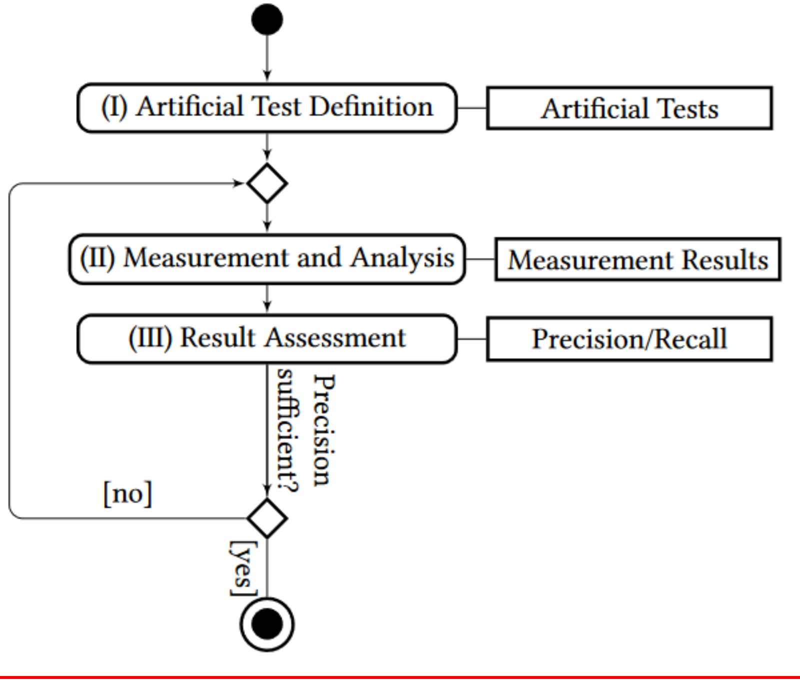
This paper was written by 3 different people from Kiel university; Jan Waller, Nils C.Ehmke, Willhelm Hasselbring[50]. A case study was performed where the three of them tackled a problem of automated benchmarks and how it may be included in continuous integration. They wanted to be able to catch problems earlier in the development cycle ,whether it was errors or performance issues. The result they ended up with was in fact something close to what they envisioned but not quite. Creating performance regression benchmarks to catch problems after software went into production was possible, however that isn’t what they wanted. In their paper

however, they state that they wanted to be able to implement the benchmarks in the early stages of the development cycle , which means the project was not fully completed.

What they did is that they used a setup with a tool called Kieker. Kieker is a monitoring system that monitors performance. This tool is based on Jenkins which was stated earlier to be the main tool for this case study. They used Kieker to visualize the performance regressions. Then they would use moobench, which is a tool that can immediately detect these performance regressions. This is done in increments nightly.

How to detect performance changes in software history: Performance analysis of software system versions

Written in 2018, David Reichelt and Stefan Kuhne wrote a paper regarding the problem of source could changes[51]. By improving performance through positive changes and avoiding negative ones through the knowledge of structure in source code changes. Their primary tools to do this were Apache Common IO which develops input and output functionality, which was what was used for version history, and Junit that was used for. The conclusion of the project was that they managed to detect the performance changes through the version history, however, they were not able to find the root cause of these changes , which would be the next step of the process.



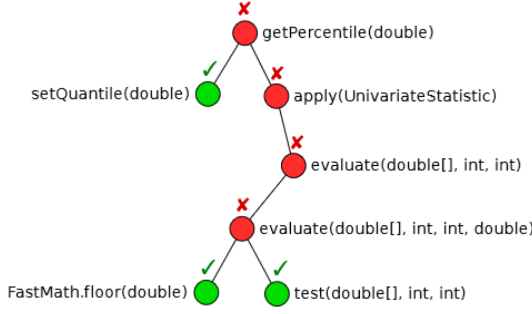
Above is a figure of how they distinguish performance

The way the conclusion and the end of their project was reached was through the presentation of a performance analysis. They built up a knowledge base of changes that had impact on the performance of a software by analyzing unit test version history of a repository. They took two unit tests and measured the performance changes.

Automated root cause isolation of performance regressions during software development

“Automated root cause isolation of performance regressions during software development”[52] approaches the problem of integrating root cause analysis of performance into development using unit tests and revision history with a case study through SAP as development infrastructure.

What Christopher Heger, Jens Happe and Rozbeh Frahabod did was use a unit test and revision history graphs for automatic detection and analysis of root cause problems through a development process. Built on hybrid regression detection strategy, they managed to build their approach. Bisection over revision change graphs and analysis of performance annotated call trees to perform root cause analysis. They ,by their own admission, were not able to stand alone tools that offered performance metrics.



Here is what a typical call tree looks like.

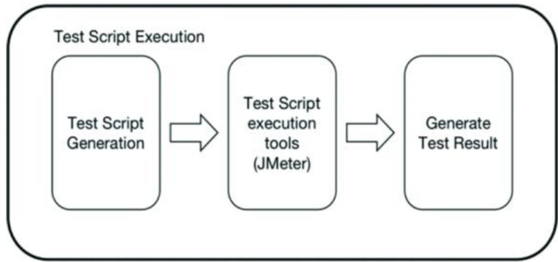
Unit Testing performance with stochastic performance logic

Stochastic performance logic[53] is a mathematical tool for expressing and evaluating performance requirements. SPL attaches these evaluations to individual methods to define common performance tests. The 7 writers of "Unit testing performance with stochastic performance logic" to use unit testing for performance testing with SPL.

They used the SPL to let developers assumptions be implement performance tests based on the JDOM project. The JDOM Project is about turning XML data into java code. To do this they used the Junit for java, Git and Subversion for support and eclipse and Hudson for integration. For the prototype they used, they used the coding language C#. In the end they used SPL in an environment that attaches performance requirements to performance tests.

Implementation of Continuous Integration and Continuous Delivery (CI/CD) on Automatic performance testing

This paper is like the first half of this thesis. It focuses on implementing CI/CD into an automatic performance test through a prototype. It does not mention a pipeline however. What they did in order to prepare for what they were going to implement, was create workflows to map out processes.



This is what a process workflow looks like.

For repository they used Gitlab, and for Jenkins they build the history of the CI. They managed to finish their project, however they mention it could only be done through JMeter GUI , which is a java-based software and is a testing interface, and JMeter Script with CLI Triggers which is the version based on command line. This trigger was of necessity as without the trigger, they could not do this automatically or periodically.

Machine Learning-assisted Performance Testing

Finding papers that are very close to this thesis in terms of subject matter is was a difficult task in terms of the dynamic testing portion. However , that reaches leaves us at this paper and the one following this.

The subject matter[55] that is being tackled here is in regards to stress testing. Performance testing and how it is done under stress, as in extreme conditions to find performance breaking points, particularly for complex software systems.

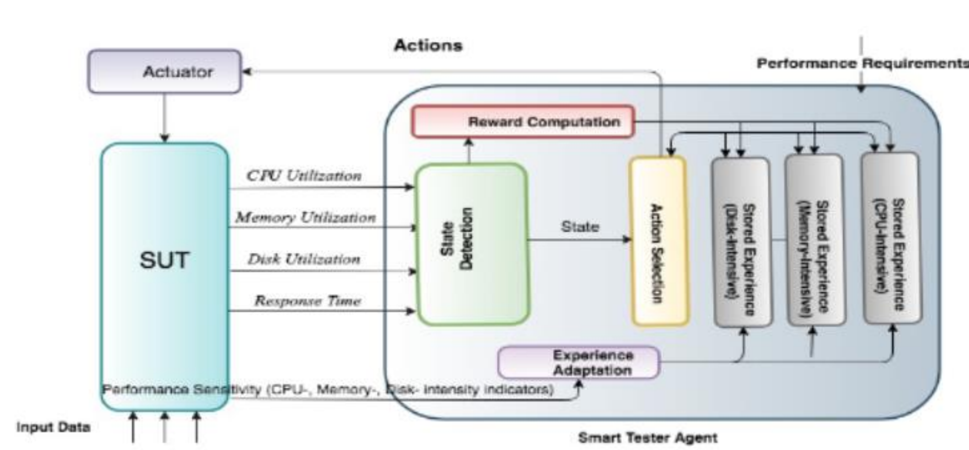
During the creating of this prototype, the writer developed a model-free build that reinforced learning. What this build did,was , it was self-adaptive to stress testing framework. It was able to learn what was optimal for a stress test without having a model of the system under the test. For the benchmarking programs, they used Build-Apache, n-queens,dcraw.

Continuous Software Performance Assessment: Detecting performance problem of Software Libraries on Every Build

This paper[56] involved using the tools Java and Go as open source software and cloud instances were used such as google compute engine to develop methods that allow software microbenchmarks into CI by using the cloud and techniques to reduce execution time.

The reason why this author wanted to do this, is because they wanted to attempt at a prototype to solve the problem of software performance degradation. Software performance can become costly and it isn't widely used continuously for developers or companies. The goal of this mixture of a prototype and case study wa reached. The goal was to bring software benchmarking to CI. They were able to enable performance problem detection for every software build.

M. H. Moghadam



A figure of the stress test framework.

What was shown to be the result was improved efficiency and reduced effort for creating conditions while dependency was reduced on source code and models. What needed to be worked on further was the lack of support in terms of workload. And workload was a point that was wanted to further the research in regards to stress test conditions.

3 Approach

With knowledge of concepts and research of similar nature to this thesis have been presented, a plan must be made to be able to cover the problem statement at hand. The problem statement at hand is; *Investigate the Implementation of dynamic performance testing in CI/CD pipeline.* The primary goal of this problem statement is to take the results of the performance test and compare it to the previously done performance tests. It is also something that is to be attempted to be analyzed graphically as well.

Research Methods

The purpose of research is to contribute to a subject by advancing knowledge. There are several methods to do research, but there are three main ones. explanatory research, descriptive research and exploratory research[37].

When something happens, a researcher generally wants to know the cause and effect of it; or more specifically why something is happening. This is what is known as the explanatory research method. One of the ways one can perform research explanatory is through a case study, an example of what this looks like is to look at the literature review of this thesis.

The descriptive research method is another choice. A descriptive research characteristic is that it tend to involve a lot of data. By having all this data, one should be able to find correlation between different things. This does not mean correlation in the manner of cause and effect, that is an important distinctive factor that separates the descriptive method from the explanatory method. While explanatory research is the question of “why”, descriptive research is the question of “how”.

The final method is the exploratory one. The exploratory research method is possibly the most “scientific” in that you must discover things. The descriptive method is the “how”, the explanatory method was the “why”, leaving the exploratory with the “what” as in “what is happening here”. The goal is to “explore”. You want to investigate an idea, whether it is theoretically defined or defined some other way.

Project’s Outline

The basis of this thesis is around dynamic performance testing within a CI/CD pipeline and what could happen. Out of the three different research method, this problem statement seem to fit the description of explanatory research. The primary reason for this is because the exploratory and descriptive methods are about the “what” and the “how” questions, while this paper is about why investigate the implementation of dynamic performance testing in CI/CD Pipelines.

Seeing as CI/CD pipeline and dynamic performance testing are the core aspects of this thesis, the first course of action is to find a list of the most popular CI/CD pipeline tools and the most popular performance testing tools as well as how to make a graphical interface for the results. Finding the most popular, may not imply they are the best, however, one core element usually in something being popular is, it’s accessibility which means less time will be spent understanding how it works. What it also could mean, is that there is more community support for it; the more popular it is, the easier of a process troubleshooting is. Some of the first tools that will be experimented with are the ones used during semesters before this thesis as the element of familiarity plays a part. After the tools have been figured out, the next step is to figure out how the functionalities come into play.

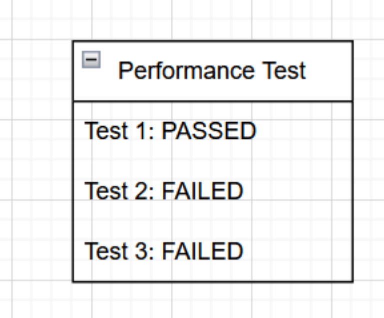
The gameplan is to first find a tool for; performance testing, CI/CD pipelines and a way to graphically create diagrams that work. Try to complete the project by implementing everything to solve the problem statement. Then try to find other ways of doing so, and present these in a less technical and more theoretical way as the time restrictions do not permit further implementation.

When final decisions have been made, is when we start writing the result chapter. Chosen tools, what ended up becoming part of the project, answers to the problem statement all goes in the result chapter.

Goal on a technical level

What needs to be done to see if we have reached the goal is simple. First we need to get an Openstack container to connect to Gitlab with a runner. This is taken into consideration that the openstack virtual machine has installed docker and the container itself. Not too much weight will

be put on the docker side in this research paper as it will just be a few commands in a command line . After that is done, we need to write a script that performs a regular performance test.

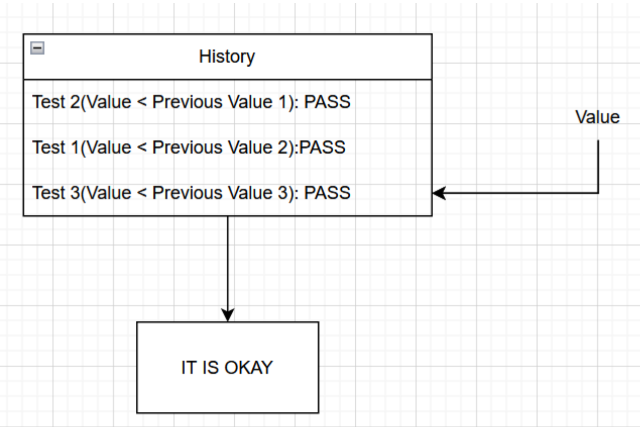


This is a performance test figure

This is in order to have a basis for the dynamic performance testing and so that one can have a foundation for what we want to build going forward. The way we know this test is functional is that we will be inputting a value, and the value will be compared to a threshold, if this value is above or below this threshold, there will be an output. An example could be if it is above a certain value it could say it is okay, if it is below a certain value, it could say it is not okay.

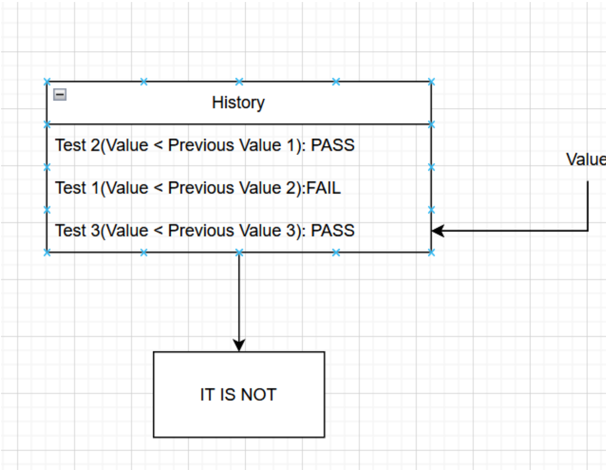
From there what will happen is we will write a second script. This script will contain both the dynamic performance test and the way to create the graph. Creating the graph itself is just going to be an attempt to put in the values into something like excel, then try to create a graph from that.

As far as the dynamic performance testing goes, it can look like something like this.



The figure of a dynamic performance test that meets the condition

This is what it looks like when you compare a value to the previous values and based on the conditions you get a pass or fail. If they all pass, you get the okay. If it fails however and it doesn't meet the conditions.



A performance test that doesn't meet the conditions.

This is in essence what the project is going to be. How we know if we succeed is whether we found out if we got an okay or a not okay based on the conditions laid out in the history.

Once the technical part is finished and the result chapter is finished as it will be written alongside the implementation, all that is left is to write the discussion chapter, which functions like an

autobiography that explains thoughts and the like, and the conclusion chapter, which will be the answer to the problem statement.

The Pitfalls

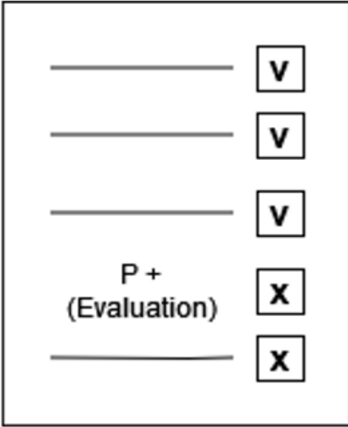
Now during this project, the biggest problem that could occur is time management. The consensus about short thesis, is that there usually is a lack of time, the lack of time could be caused by different factors, underestimating the scope of the project, lack of advancement technically causing it to be a time sink or just a lack of direction. There is also the lack of communication between the student and the supervisor. Or perhaps even not knowing the structure of how a master thesis is meant to be written. There are several pitfalls, and they are unpredictable even if you know them.

4. Result

With the former research presented, this is the chapter in which we will present answers to this thesis problem statement. In this chapter, the first part will include how this problem statement will be solved theoretically. Next the second part will be the implementation of the theory. Finally, the analysis will be done based on results we have collected.

A Framework for performance testing in CI/CD

In order to approach the problem, we first have to consider how performance tests can be integrated into CD/CI pipelines consider the following figure.

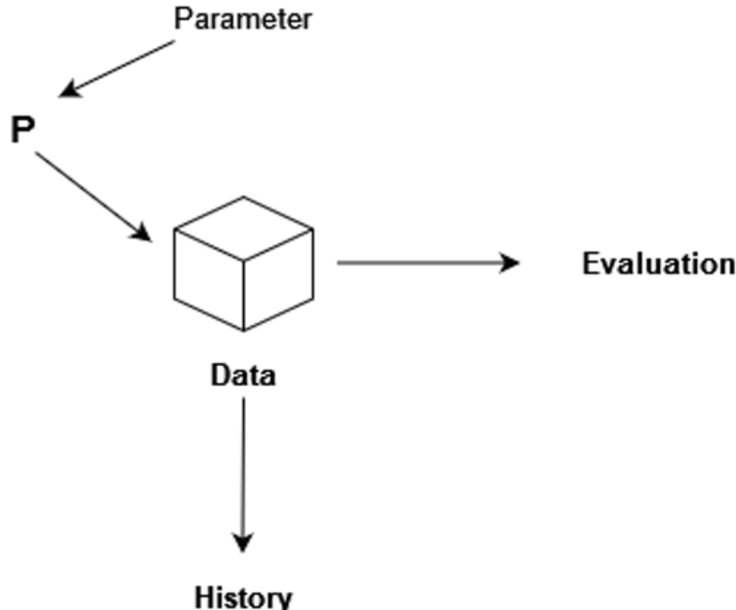


This is the list of test that is represented as a paper.

The figure above illustrates a representation of a list of tests that are being conducted for a software project. Each test can be a performance test that is being conducted. The v's and the x's represent tests that have passed or failed. In practice, these v's and x's are Boolean variables which will be assigned a "true" or a "false" value. "true" will represent that a test has passed, "false" is going to mean it's failed.

When it comes to a performance test however, a parameter needs to be set in order to provide a threshold to help translate a numerical value to a Boolean. For example, in context of previous core measurements of a performance test, it can be represented as the; response time, memory or

other values one wishes to measure in relation to performance. In this simple form, a performance test can be viewed as a basic yes/no test by adding a user-supplied threshold and performing an evaluation.



A user-supplied parameter which acts as a threshold

The figure above is a basic representation of this functionality the parameter is tested against the data. As it is tested on the data,which represents the output of a given performance test. there will be an evaluation done to see if it meets the requirements that are supplied. The figure also contains a history, which means one could collect results over time. If one has a set of data that has been evaluated and shown results it does help to see what effects the parameters had previously, and one could compare previous tests with what one currently has.

R = {pt1,pt2,pt3}

Pt1 = [r1,r2,r3]
Pt2 = [r1,r2,r3]

↓

Change - CI/CD - Boolean
Stop Test

The cycle when it comes to the performance test.

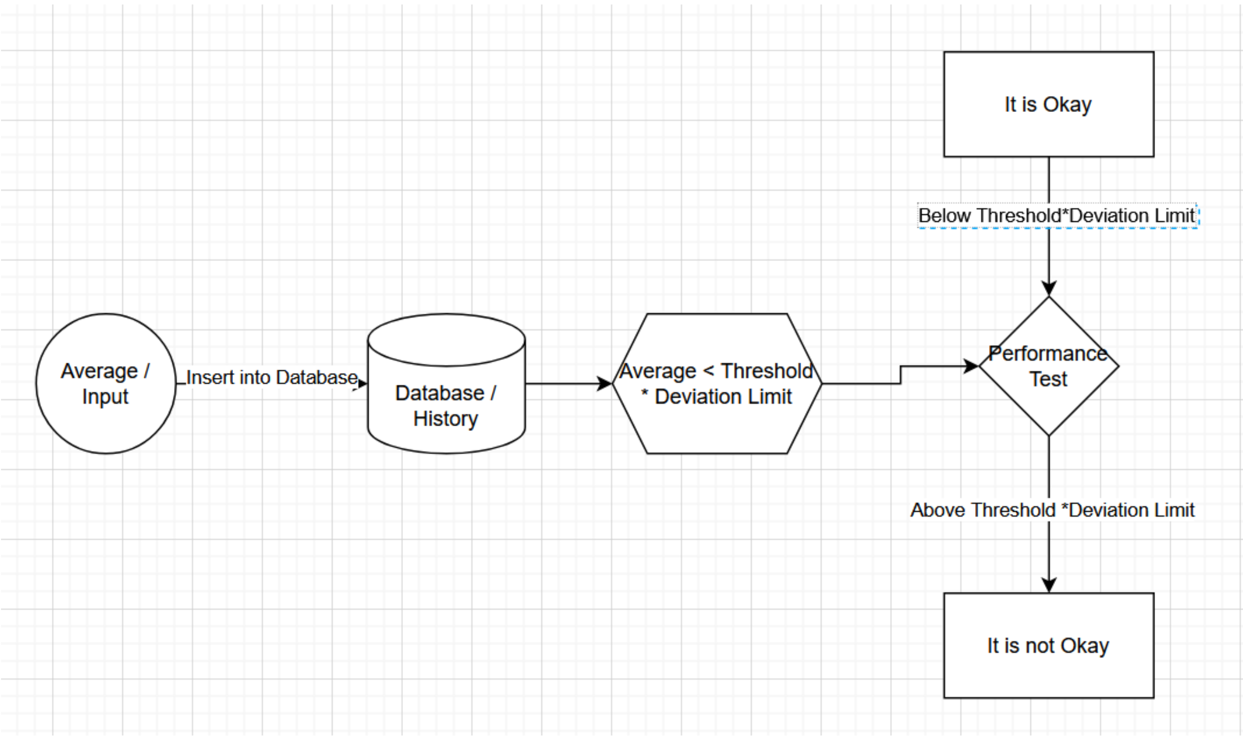
Time, one will get a history of results for each of the individual tests. For example, performance test pt1 will be executed once per test, but will also provide a history for itself which can be used for more advanced equations with or without a user supplied parameter.

This equation shows the performance test R and it's results. The {pt1,pt2} parts of the result that show that there is, in this case more than just a yes or a no answer for each test.

The part below with Pt1= [r1,r2], signified that individual performance test has its own history. For instance, the first Pt on this list,Pt1 has its own history where the results are saved each time it is ran. This type of functionality is typically not part of a CD/CI software package. Transitions from a basic evaluative function as described above to a history-based approach is the core interest of this project. This involves exploring how such a functionality can be added as well as it's uses. There are numerous ways to use a history of results, however, and this project does not aim for deep statistical analysis and anomaly detection. Though our ambition is to pave the way for these kinds of approaches in the future.

How can we turn this into something dynamic?

First, we need to make sure that our terms are defined. So, to reiterate from the problem statement. Dynamic means that the script is going to react in a certain way determined by what the conditions are. What these conditions can vary, however, what is going to be done in this paper is going to be taking the average of something, then multiplying the threshold of something with a deviation limit.



The figure that shows how this will function dynamically.

This is what it will look like in terms of how it will function. The average will be gathered by httperf which will be a webserver performance tool that will be elaborated on in the section in regards to the implementation, but it essensially is testing the response time with the openstack instance. Then the average of that will be inserted into the database. Threshold will be calculated based on the previously results and multiplied with the division limit and if the average is smaller then it will be okay, if it is bigger, it will be not okay. This is in essence how this will work dynamically.

5. Implementation

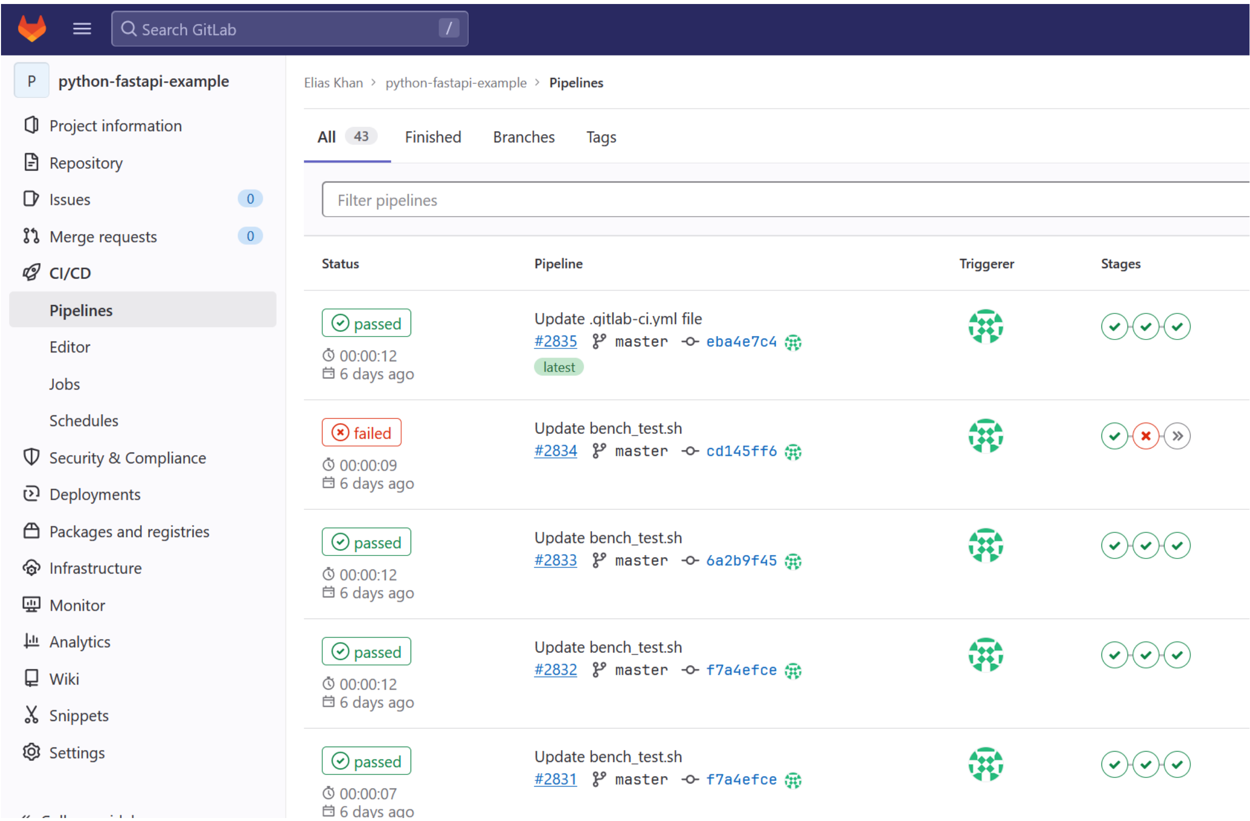
For the implementation of the project, there are several components that will be overviewed. First the basic tools will be discussed to make sure what is used. Then after that, scripts and such will be presented as well to show how it runs. Note that until the project implementation section, all of these screenshots in the tool section are just for educational purposes, they do not represent the actual project scripts.

The Tools

GitLab's role

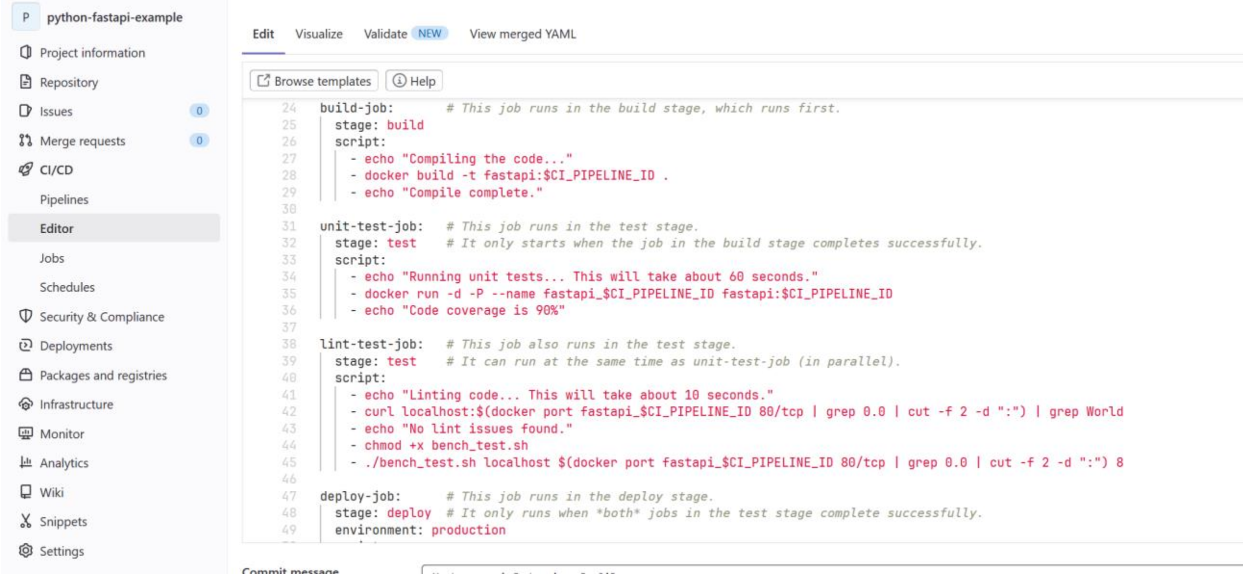
There are parts that are related to Gitlab that may be shown in another section of this thesis further down as it is believed to be more fitting there, parts such as Gitlab runners.

With the chosen software, there are a few important aspects of it that needs to be covered. Once you select a project, you can select an option titled CI/CD, which is where the work for this project and the feasibility for our implementation will be attempted. Within this menu, you have four other crucial parts in making the pipeline function; pipelines themselves, the editor, the jobs and the schedules.



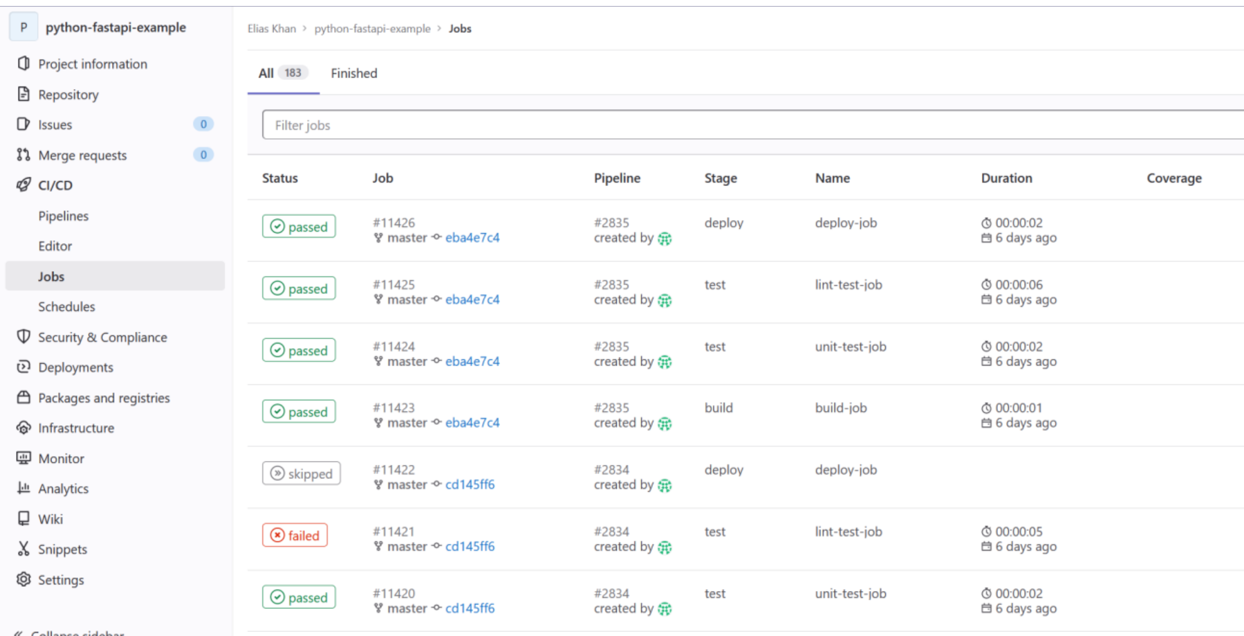
This is what the Gitlab dashboard of the project will look like.

The pipeline part itself shows the different pipelines you have. As you click on one of them you will be able to see the different processes that it goes through. The building, testing and the deployment. If something were to go wrong under the run, it will stop and show during what part of the run it crashed. If something crashed , one should be going to the editor in order to check what has happened during the run on the specific part that is highlighted on the pipeline that crashed.



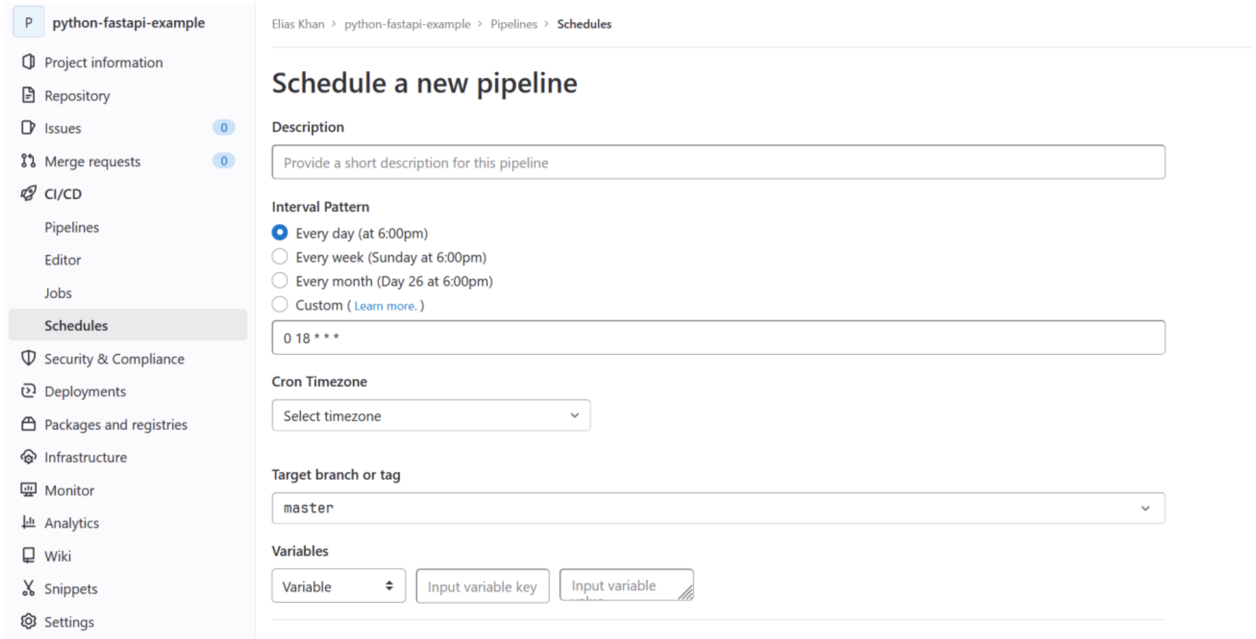
The view of the Editor and the gitlab-ci.yml

Once you have opened the editor you will be greeted with a file known as the gitlab-ci.yml. This file is the one that runs the different parts of the pipeline and you are able to insert what code you wish to run within each of the stages. If something were to happen and it were to crash in different parts of it, this is whereas mentioned before, it will show itself within the pipeline window in the previous section.



This is what the jobs look like

The jobs will show you similarly to what the pipeline shows. However, unlike the pipeline stage where it will show the different parts of the pipeline, it will instead show the steps themselves in text format. It will also show where it went wrong so to speak, what it skipped, what failed and other types of messages to explain the state of the pipeline step.



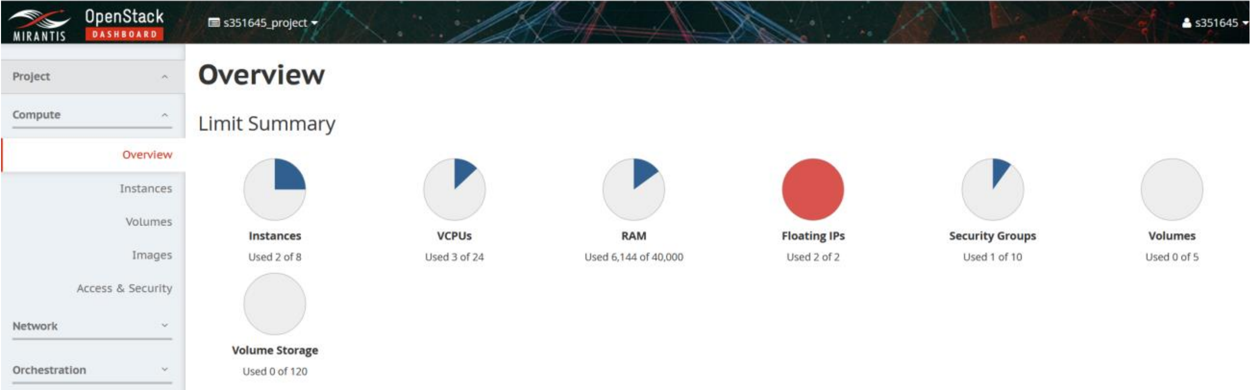
This is what the schedule part of GitLab looks like.

At last we have the final part of the GitLab pipeline which is the schedule. The schedule of the CI/CD pipeline's job is to run it as times that are specified. It could be every day, week, month or you can custom build it as well as pick a time zone. This is arguably one of the more important parts of the pipeline outside of crucial functions to make it work, because it builds on one of the cornerstones of continuous integration and continuous delivery, automation. However, for this project it was not a part which was emphasized as it was not a crucial part of it, and it would have been a novelty instead of a basic need.

Openstack's Role

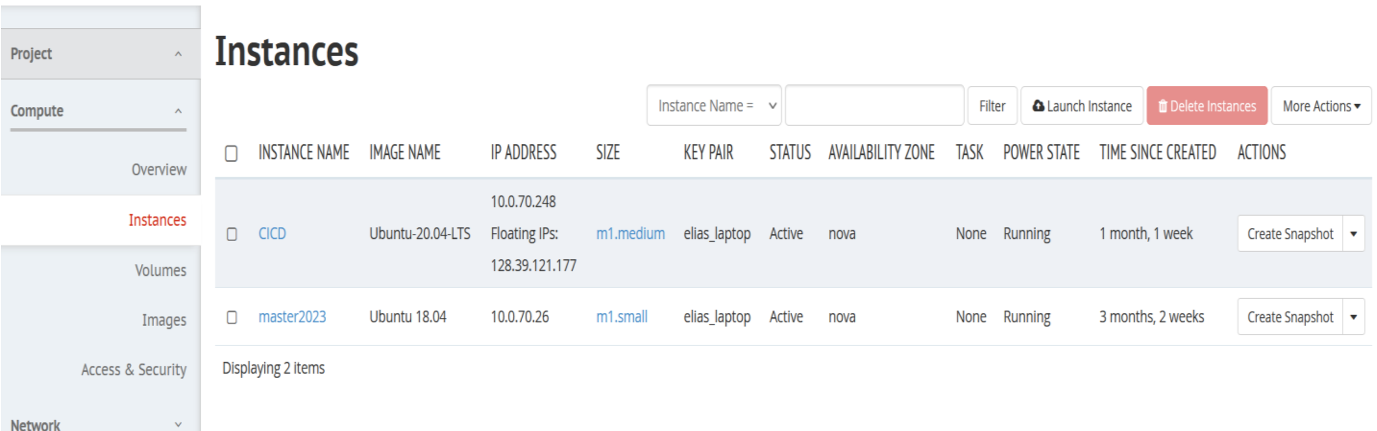
In order to make the project possible, it was necessary to be able to find a way to run it which is why Openstack was chosen. Openstack is an open source tool that allows the users to operate with clouds. The simpler term would be it allows the user to create virtual machines which are software made to emulate a computer.

Openstack has several functions, however not all of them will be relevant to this thesis, or they are aspects of it that usually will be automated into every newly run instance, so it is only of importance to select and explain the ones that are important. When you enter Openstack , this is what the opening dashboard will look like.



The dashboard of OpenStack.

From there you will press on the Instance option on the menu where you will be lead to the server that will be running the project itself.

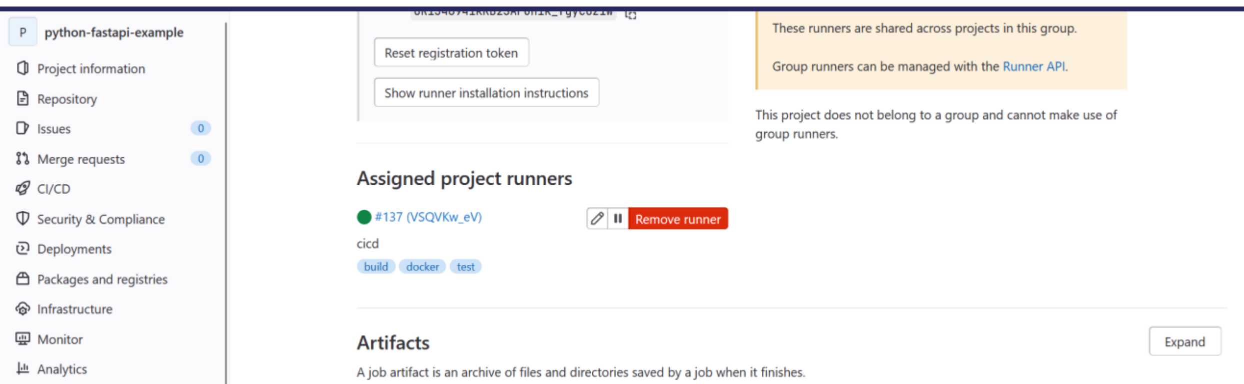


The instance CICD that is running the project.

The virtual machine that is running this project is the CICD instance. When you are launching the instance from scratch, you will be asked to give certain required specifications. The requirements that are significant is shown on the right side of the virtual machines name. The image name is Ubuntu-20.04-LTS, this being the version of ubuntu that is being ran on this virtual machine. There was no real reason for choosing this specific one other than that it was the

latest version. There have been problems before on previous projects when you needed a more specific version, but that was not the case here. The size refers to the memory in which can be used. Medium was chosen as anything more than anticipated to be needed would be a waste of space, however other complications could have arisen from having too little space, like certain scripts not being able to execute due to a lack of space on the virtual machine, but medium was anticipated to be enough due to the very basic nature of the scripts. You have two IP addresses here, however the floating IP, which is 128.39.121.177 is the one that is relevant as it will allow us to log into this server from our local computer.

How this is connected to the Gitlab is where the Gitlab runner comes in. By going into the project and pressing on the Settings menu, you can scroll down to Runners, expand the menu, scroll down and find the runners of the project.



The runner that is commected to the Openstack server.

Runner #137 project

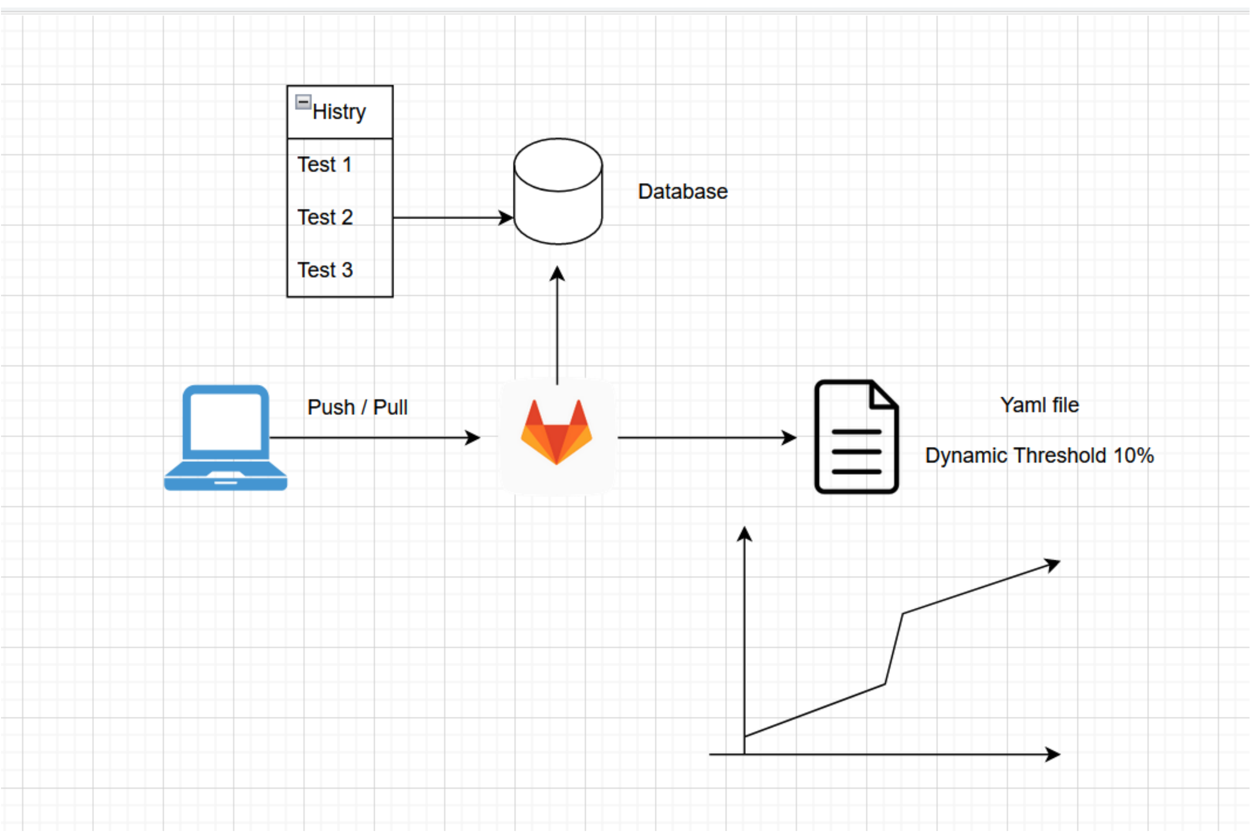
Property Name	Value
Active	Yes
Protected	No
Can run untagged jobs	Yes
Locked to this project	No
Tags	build docker test
Name	gitlab-runner
Version	15.11.0
IP Address	128.39.121.177
Revision	436955cb
Platform	linux
Architecture	amd64
Description	cicd
Maximum job timeout	

The details of the runner.

The figure above shows the specifications of the runner. What is the most important is the IP address. This IP address, 128.39.121.177 is the same s the floating IP previously mentioned which means they are connected.

The project implementation

From this section, what is represented here is what is a representation of the project itself. Now that the individual pieces of importance in relation to the project itself have been explained, it is time to present the results of the implementation. The result of the problem statement which was; Investigate the implementation of dynamic performance testing in a CI/CD Pipeline.



This figure shows the history that is gathered in gitlab as well as the CI yml file that deals with the history databse and the ci yml file's dynamic threshold.

We need to make clear how this will work. First, a user will push a piece of code into gitlab. Then what will happen is the code will be put through a pipeline in the case of a yml file. Here you have a dynamic threshold of 10% which will be tested against already done results in history will be tested against. Then it will be put into a graph.

The first thing that must happen for the implementation to work is that the user must have an interaction with Gitlab. This interaction will involve having some sort of code sent into GitLab as it is being compiled. This is going to be in our python-fastapi-example repository.

Name	Last commit	Last update
app	Added main files	1 year ago
.gitlab-ci.yml	Update 2 files	1 month ago
Dockerfile	Added main files	1 year ago
README.md	Add README.md	1 year ago
bench_test.sh	Update bench_test.sh	3 months ago
dyn_bench_test.sh	Update dyn_bench_test.sh	3 weeks ago

The python-fastapi-example repository.

By looking at the repository, then files, you will find the gitlab-ci.yml. This file is the one that will be the scripted version of the CI/CD pipeline.

```
19  stages:          # List of stages for jobs, and their order of execution
20    - build
21    - test
22    - deploy
23
24  build-job:       # This job runs in the build stage, which runs first.
25    stage: build
26    script:
27      - echo "Compiling the code..."
28      - docker build -t fastapi:$CI_PIPELINE_ID .
29      - echo "Compile complete."
```

the first half of the gitlab-ci.yml file.

```
29      - echo "Compile complete."
30
31  unit-test-job:   # This job runs in the test stage.
32    stage: test    # It only starts when the job in the build stage completes successfully.
33    script:
34      - echo "Running unit tests... This will take about 60 seconds."
35      - docker run -d -P --name fastapi_$CI_PIPELINE_ID fastapi:$CI_PIPELINE_ID
36      - echo "Code coverage is 90%"
37
38  lint-test-job:   # This job also runs in the test stage.
39    stage: test    # It can run at the same time as unit-test-job (in parallel).
40    script:
41      - echo "Linting code... This will take about 10 seconds."
42      - curl localhost:$(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") | grep World
43      - echo "No lint issues found."
44      - chmod +x bench_test.sh
45      - ./bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") 8
46      - chmod +x dyn_bench_test.sh
47      - ./dyn_bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") 10 15
48
49  deploy-job:      # This job runs in the deploy stage.
50    stage: deploy  # It only runs when *both* jobs in the test stage complete successfully.
51    environment: production
52    script:
53      - echo "Deploying application..."
54      - docker stop fastapi_$CI_PIPELINE_ID
55      - docker rm fastapi_$CI_PIPELINE_ID
56      - echo "Application successfully deployed."
```

The second half of the gitlab-ci.yml file

The two figures of scripts above are going to go through the entire process of the pipeline and the performance test itself. As we go through the different parts of the script, we will go to those

scripts that are relevant for that part of the pipeline. This pipeline consists of, as described below the stages part of the script; a build-job, two test-jobs and a deploy-job. All this does is to explain what the stages of the pipelines are. Every job has a line with the name stage, and all this does is just assign the name to the job. The script part is also where one writes the code for the jobs.

Below that you have the first stage which is the build-job. In this stage you have a basic line that will print out the message of the code being compiled. After you see the line “docker build -t fastapi:$CI_PIPELINE_ID. This line is going to build a docker container with the name of fastapi and it is giving it an ID as well. After this the compilation is completed.

```
24  build-job:       # This job runs in the build stage, which runs first.
25    stage: build
26    script:
27      - echo "Compiling the code..."
28      - docker build -t fastapi:$CI_PIPELINE_ID .
29      - echo "Compile complete."
```

The picture of the build-job.

This is a good place to bring up the code that it is compiling. This is found in the repository tab as well, in the app folder, there a main file that is written in python is found. The main.py file will look something like this.

main.py 236 bytes

```python
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}
```

the main.py file.

This is a very basic script that all it does is to print out Hello World, when it does will be explained further down the gitlab-ci.yml file when this will come into play. One thing that may be a bit confusing may be the fastapi[42] importing as FastAPI. This is a high performance web framework.

This is also where one could look at the Dockerfile as the “docker build” command is being used.

Dockerfile 69 bytes

```dockerfile
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7

COPY ./app /app

```

TheDockerfile script.

The script above is a very simple basic Dockerfile script. All this script is supposed to do, is to build the docker container, and what this specifically does is install python 3.7 into the container.

This is so we can make the main.py file work in the container. Then we are just copying the ./app folder into the container, which is where the main.py file resides.

Further down the gitlab-ci.yml file. We will find the unit-test-job.

```
31  unit-test-job:   # This job runs in the test stage.
32    stage: test    # It only starts when the job in the build stage completes successfully.
33    script:
34      - echo "Running unit tests... This will take about 60 seconds."
35      - docker run -d -P --name fastapi_$CI_PIPELINE_ID fastapi:$CI_PIPELINE_ID
36      - echo "Code coverage is 90%"
```

The unit-test job of the gitlab-ci.yml file.

This is where the testing begins. Here what is happening is that the docker container that was built in the build-job stage is now being ran. This test cannot start without a successful build job.

After this we have a second test, the linit-test.

```
38  lint-test-job:   # This job also runs in the test stage.
39    stage: test    # It can run at the same time as unit-test-job (in parallel).
40    script:
41      - echo "Linting code... This will take about 10 seconds."
42      - curl localhost:$(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") | grep World
43      - echo "No lint issues found."
44      - chmod +x bench_test.sh
45      - ./bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") 8
46      - chmod +x dyn_bench_test.sh
47      - ./dyn_bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") 10 15
48
```

the linit-test-job of the gitlab-ci.yml file.

Here we have the linit-test-job. This is where most of the important things happen in the pipeline. On line 42 we have a curl and a line which involves localhost and the command docker port with the name of the container. What is happening here is that we are accessing this docker container. If everything went well, you are looking at the command on the line after, where it will tell you that “no linit issues found”.. Afterwards, you see a “chmod +x bench_test.sh” command. This command is about allowing the bench_test.sh script to be run. Very often you will see that a script will not run, and you will get an error method, mostly, if this happens, it is due to a lack of permission. The final line, line 45 you have a line that runs the ./bench_test.sh script on the localhost on the specific docker container that was just connected to before on line 42.

This is one of the most crucial part of the thesis and where a lot of questions will be answered. The ./bench_test.sh script itself.

```
1   #!/bin/bash
2
3
4
5   IP=$1
6
7   PORT=$2
8
9   THRESHOLD=$3
10
11  echo "Starting benchmark on $1 port $2, Threshold is $THRESHOLD"
12
13  which httperf
14
15  #httperf --server $IP --port $PORT --uri / --rate 150 --num-conn 700 --num-call 1 --timeout 5
16
17  avg=$(httperf --server $IP --port $PORT --uri / --rate 150 --num-conn 700 --num-call 1 --timeout 5 | grep "Connection time" | grep avg | awk
18
19  #avg=2
20
21  echo -e "avg: $avg, threshold: $THRESHOLD" # >> bench_log
22
23  echo -e "Making decision based on performance"
24
25  st=$(echo "$avg < $THRESHOLD" | bc)
26
27  if [ "$st" -eq 1 ]; then
28  echo -e "it is ok";
29  exit 0
30  else
31  echo -e "it is not ok (avg = $avg, threshold = $THRESHOLD)";
32  exit 1
33  fi
```

This is the ./bench_test.sh script.

The ./bench_test.sh script is the one of the most important parts of the entire implementation part. What we are seeing here in essence is the performance test.

The first three variables are very crucial to this, but they do not come into play until line 11. There will be a closer look into the results of this in the analysis section further down, but for now it will be explained in a basic sense. The IP=$1 variable is essentially taking in the ip address. In this case the ip address would be localhost as it is where the docker container is ran. PORT=$2 stands for the port that it is going to be using for this script. When the docker container was built, a port was created alongside it.THRESHOLD=$3 is the number that was seen in the linit-test-job part of the gitlab-ci.yml file.

On line 13 what we are seeing which httperf. Httperf[43] is a tool that test web performance and is being called by which. Below on line 17 we see avg which just returns the average value, but the average value of what? Inside avg what you have is httperf with a bunch of variables. What is found here is that httperf is looking for a variable in server, which is the IP=$1 variable, which is going to be localhost, the port variable which is going to be the PORT=$2 which is the port that was generated by the docker build command, and the THRESHOLD=$3 variable which was the variable we set to see the performance measured up to how we wanted it to. The rate[44]

variable, is about the rate at which the connections are made. Num-conn is about the number of connections made. Num-call is about the number of connections that will be create and timeout is just the amount of seconds httperf is willing to wait for a response from the server. The remains is just printing out the average and the connection time string.

The next line of importance is line 25, here, where the average is checked to be smaller than the threshold and it's put into the st variable. The if-test at the bottom is there to confirm whether or not it is smaller or bigger than the threshold and if isn't then you will be told, and it will show the average and the threshold.

The bench-test.sh script is in essence our script that is testing the response time of the server by testing the response time of the server putting up against a threshold, where you at the end are left with whether it is, and if it is, we can make a decision based on whether it met the criteria or not.

```
38  lint-test-job:   # This job also runs in the test stage.
39    stage: test    # It can run at the same time as unit-test-job (in parallel).
40    script:
41      - echo "Linting code... This will take about 10 seconds."
42      - curl localhost:$(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") | grep World
43      - echo "No lint issues found."
44      - chmod +x bench_test.sh
45      - ./bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") 8
46      - chmod +x dyn_bench_test.sh
47      - ./dyn_bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") 10 15
48
```

the linit-test-job of the gitlab-ci.yml file.

The second half of this script that is important is what is happening on line 46 and 47. The same commands are executed as for the bench_test.sh script. On line 46 is given permission to run, and on 47 it is run on the same docker container as the bench_test.sh script. This is the dyn_bench_test.sh script. This is where the dynamic performance test will take place.

The scripts below is the first and second half of the dyn_bench_test script.

```
1   #!/bin/bash
2
3   IP=$1
4
5   PORT=$2
6
7   MEMORY_LENGTH=$3
8
9   DEVIATION_LIMIT=$4
10
11  echo "Deviation limit is set to $DEVIATION_LIMIT"
12
13  # fetch previous values and calculate threshold
14
15  TEST_NAME="dyn_httperf"
16
17  echo History of previous $MEMORY_LENGTH values is:
18
19  echo "select * from bench_log where project_id = '$CI_PROJECT_NAME' and test_id = '$TEST_NAME' limit $MEMORY_LENGTH" |  mariadb -u cdci --pas
20
21  HISTORY_COUNT=$(echo "select * from bench_log where project_id = '$CI_PROJECT_NAME' and test_id = '$TEST_NAME' limit $MEMORY_LENGTH" |  maria
22
23  if [ "$HISTORY_COUNT" -gt 0 ]; then
24  THRESHOLD=$(echo "select value from bench_log where project_id = '$CI_PROJECT_NAME' and test_id = '$TEST_NAME' limit $MEMORY_LENGTH" |  maria
25
26  else
27  # failsafe in case history is empty
28  THRESHOLD=100
29  fi
```

This is the first half of the dyn_bench_test script.

From line 2 to 9 you have a set of variables that will reoccur through the script. Here you have IP address,the Port adresse and the memory length. The first two are of the same variety as the previous script. Memory length, however, is the length of which how many previous results in the history is shown. And deviation limit will be explained further down the script. On line 15 you have the test name variable which is used to fetch the ID of where the stored tests are. On line 19 you have a statement made to the database shown in the figure before which will be printed out. The next line shows history count, which basically shows how many history statements there are as in countable previous tests. Then you have an if test where there is any count then it will try to create a threshold.

```
30
31  echo "Starting benchmark on $1 port $2, Threshold is $THRESHOLD"
32
33  which httperf
34
35  #httperf --server $IP --port $PORT --uri / --rate 150 --num-conn 700 --num-call 1 --timeout 5
36
37  avg=$(httperf --server $IP --port $PORT --uri / --rate 150 --num-conn 700 --num-call 1 --timeout 5 | grep "Connection time" | grep avg | awk
38
39  #avg=2
40
41  # insert value into history
42  echo "insert into bench_log ( project_id,test_id,value) values ( '$CI_PROJECT_NAME','dyn_httperf','$avg')" | mariadb -u cdci --password=cdcip
43
44  echo -e "avg: $avg, threshold: $THRESHOLD" # >> bench_log
45
46  echo -e "Making decision based on performance"
47
48  st=$(echo "$avg < ( $THRESHOLD * 1.$DEVIATION_LIMIT )" | bc)
49
50  if [ "$st" -eq 1 ]; then
51  echo -e "it is ok";
52  exit 0
53  else
54  echo -e "it is not ok (avg = $avg, threshold = $THRESHOLD)";
55  exit 1
56  fi
57
58
```

This is the second half of the dyn_bench_test script.

Then on line 37 an average is being calculated by the httperf command. After that the value is being put into history. On line 48 ,the st variable which is going to be the determining factor of whether or not there was a pass or fail will be calculated. It is going to take the avg and see if it is bigger than the threshold multiplied by 1. The deviation limit.

If it passed the test, it will be giving the message “it is ok”. The criterion for passing is if the statement on line 48 is true. This project already has been fed values prior to this. If fails, it would present the statement on line 54.

An example of a failed test is if the average is higher than the 1 multiplied with the deviation limit multiplied with the threshold as seen on line 48. Which is why line 42 is very important. By feeding the database negative values, on purpose we can make the equation on line 48 low enough to where the average is higher, meaning that it will be a failed test.

```
32 # insert value into history
33 echo "insert into bench_log ( project_id,test_id,value) values ( 'CI_PROJECT_NAME','dyn_httperf',0.155)" | mariadb -u cdci --password=cdcipas
34
35 echo "insert into bench_log ( project_id,test_id,value) values ( 'CI_PROJECT_NAME','dyn_httperf',0.114)" | mariadb -u cdci --password=cdcipas
36
37 echo "insert into bench_log ( project_id,test_id,value) values ( 'CI_PROJECT_NAME','dyn_httperf',0.3)" | mariadb -u cdci --password=cdcipass
38
39 echo "insert into bench_log ( project_id,test_id,value) values ( 'CI_PROJECT_NAME','dyn_httperf',0.1)" | mariadb -u cdci --password=cdcipass
40
41 echo "insert into bench_log ( project_id,test_id,value) values ( 'CI_PROJECT_NAME','dyn_httperf',0.1)" | mariadb -u cdci --password=cdcipass
42
43 echo "insert into bench_log ( project_id,test_id,value) values ( 'CI_PROJECT_NAME','dyn_httperf',0.1)" | mariadb -u cdci --password=cdcipass
44
45
46 echo -e "avg: $avg, threshold: $THRESHOLD" # >> bench_log
47
48 echo -e "Making decision based on performance"
49
50 st=$(echo "$avg < ( $THRESHOLD * 1.$DEVIATION_LIMIT )" | bc)
51
52 if [ "$st" -eq 1 ]; then
53 echo -e "it is ok";
54 exit 0
55 else
56 echo -e "it is not ok (avg = $avg, threshold = $THRESHOLD)";
57 exit 1
58 fi
```

Feeding low values to the database

By doing this we have effectively made our performance test dynamic as it now will respond depending on the behavior that was observed by the script. The threshold will be low enough for the average to be higher than and thus we will get a “it is not ok” message.

And then we have the final part of the script gitlab-ci.yml file.

```
56
57 deploy-job:      # This job runs in the deploy stage.
58   stage: deploy  # It only runs when *both* jobs in the test stage complete successfully.
59   environment: production
60   script:
61     - echo "Deploying application..."
62     - docker stop fastapi_$CI_PIPELINE_ID
63     - docker rm fastapi_$CI_PIPELINE_ID
64     - echo "Application successfully deployed."
```

This is the deploy-job stage of the gitlab -ci.yml file.

At the end we have the deploy-job stage. What happens here is in essence only figuratively deploying the application. In the previous section discussing the theory, we showed a section showing the pipeline and there was deployment, but that was to show in essence what it would be. What happens after the “Deploying application….” message appears is that the docker container is first stopped, and then it is removed. After you will get the final message below the “docker rm” command.

This is how the final product looked and it appears that it did in fact happen to work the way it was anticipated using the different parts of Gitlab and Openstack to achieve what was necessary. What the results are , are going to be analyzed in the next section, and it will also be showing what the results of these scripts as well through the whole pipeline.

6. Analysis

With the scripts completed and the implementation finished, there needs to be some way to see the end results of the scripts and the tools. In this section, what is going to be looked at is the results presented from those scripts. What will be looked at is if we achieved what we set out to, or if we even got closer to an answer. After this, regardless of what the end results are, there will be an analysis of what could be done from where this concept can be explored further.

Output

In the previous section, what was presented was a lot of scripts in Gitlab, however, it will be in this section, where we will be presented with the results of those scripts as they will be run like jobs in the CI/CD Pipeline. The scripts will be presented first, these will not be explained again as they already have been, and then the output of those scripts will be presented. It must be noted that scripts such as bench_test.sh and the main.py scripts will not be explained as they are not directly presented as output.

```
24  build-job:       # This job runs in the build stage, which runs first.
25    stage: build
26    script:
27      - echo "Compiling the code..."
28      - docker build -t fastapi:$CI_PIPELINE_ID .
29      - echo "Compile complete."
```

The picture of the build-job.

```
1  Running with gitlab-runner 16.1.0 (b72e108d)
2    on prosjekt07 eAUQ1Y3C, system ID: s_9cee7ee5c474
3  Preparing the "shell" executor                                              00:00
4  Using Shell (bash) executor...
6  Preparing environment                                                       00:00
7  Running on prosjekt07...
9  Getting source from Git repository                                          00:00
10 Fetching changes with git depth set to 50...
11 Reinitialized existing Git repository in /home/gitlab-runner/builds/eAUQ1Y3C/0/s351645/python-fastapi-exa
   mple/.git/
12 Checking out b1deeb8e as detached HEAD (ref is master)...
13 Skipping Git submodules setup
15 Executing "step_script" stage of the job script                             00:02
16 $ echo "Compiling the code..."
17 Compiling the code...
18 $ docker build -t fastapi:$CI_PIPELINE_ID .
19 #0 building with "default" instance using docker driver
20 #1 [internal] load build definition from Dockerfile
21 #1 transferring dockerfile: 106B done
22 #1 DONE 0.1s
23 #2 [internal] load .dockerignore
24 #2 transferring context: 2B done
25 #2 DONE 0.2s
```

```
26 #3 [internal] load metadata for docker.io/tiangolo/uvicorn-gunicorn-fastapi:python3.7
27 #3 DONE 0.9s
28 #4 [1/2] FROM docker.io/tiangolo/uvicorn-gunicorn-fastapi:python3.7@sha256:1acb5c4deb1704b498e3821344ca1b
   5cc01c364d0ac7c0fd52dec05c715f5a53
29 #4 DONE 0.0s
30 #5 [internal] load build context
31 #5 transferring context: 56B done
32 #5 DONE 0.1s
33 #6 [2/2] COPY ./app /app
34 #6 CACHED
35 #7 exporting to image
36 #7 exporting layers done
37 #7 writing image sha256:9903a9cf3fa707ad91b55ae5454bf08a0be8b49a7f30b38f856a0cb45c36b834 0.0s done
38 #7 naming to docker.io/library/fastapi:2882
39 #7 naming to docker.io/library/fastapi:2882 0.1s done
40 #7 DONE 0.1s
41 $ echo "Compile complete."
42 Compile complete.
44 Cleaning up project directory and file based variables                      00:00
46 Job succeeded
```

The output of the build-job output

The figure above presents the output we get from the build-job script. The main purpose of the build script was to build our container and compile our code. The build-job was completed and this was a success.

```
31  unit-test-job:   # This job runs in the test stage.
32    stage: test    # It only starts when the job in the build stage completes successfully.
33    script:
34      - echo "Running unit tests... This will take about 60 seconds."
35      - docker run -d -P --name fastapi_$CI_PIPELINE_ID fastapi:$CI_PIPELINE_ID
36      - echo "Code coverage is 90%"
```

The unit-test job of the gitlab-ci.yml file.

```
1  Running with gitlab-runner 16.1.0 (b72e108d)
2    on prosjekt07 eAUQ1Y3C, system ID: s_9cee7ee5c474
3  Preparing the "shell" executor                                   00:00
4  Using Shell (bash) executor...
6  Preparing environment                                            00:00
7  Running on prosjekt07...
9  Getting source from Git repository                               00:00
10 Fetching changes with git depth set to 50...
11 Reinitialized existing Git repository in /home/gitlab-runner/builds/eAUQ1Y3C/0/s351645/python-fastapi-exa
   mple/.git/
12 Checking out b1deeb8e as detached HEAD (ref is master)...
13 Skipping Git submodules setup
15 Executing "step_script" stage of the job script                  00:01
16 $ echo "Running unit tests... This will take about 60 seconds."
17 Running unit tests... This will take about 60 seconds.
18 $ docker run -d -P --name fastapi_$CI_PIPELINE_ID fastapi:$CI_PIPELINE_ID
19 22ce38e220b23065fdf773098ab232d7a328747b3c68e9fcee7f874542feb348
20 $ echo "Code coverage is 90%"
21 Code coverage is 90%
23 Cleaning up project directory and file based variables           00:00
25 Job succeeded
```

This is the unit-test job output.

This unit-test job proved to be successful as its job was to run the docker container. There is not much else to take note of here.

```
38  lint-test-job:   # This job also runs in the test stage.
39    stage: test    # It can run at the same time as unit-test-job (in parallel).
40    script:
41      - echo "Linting code... This will take about 10 seconds."
42      - curl localhost:$(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") | grep World
43      - echo "No lint issues found."
44      - chmod +x bench_test.sh
45      - ./bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") 8
46      - chmod +x dyn_bench_test.sh
47      - ./dyn_bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") 10 15
```

The linit-test-job of the gitlab-ci.yml file.

```
1  Running with gitlab-runner 16.1.0 (b72e108d)
2    on prosjekt07 eAUQ1Y3C, system ID: s_9cee7ee5c474
3  Preparing the "shell" executor                                      00:00
4  Using Shell (bash) executor...
6  Preparing environment                                               00:00
7  Running on prosjekt07...
9  Getting source from Git repository                                  00:00
10 Fetching changes with git depth set to 50...
11 Reinitialized existing Git repository in /home/gitlab-runner/builds/eAUQ1Y3C/0/s351645/python-fastapi-exa
   mple/.git/
12 Checking out b1deeb8e as detached HEAD (ref is master)...
13 Skipping Git submodules setup
15 Executing "step_script" stage of the job script                     00:10
16 $ echo "Linting code... This will take about 10 seconds."
17 Linting code... This will take about 10 seconds.
18 $ curl localhost:$(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") | grep World
19   % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
20                                  Dload  Upload   Total   Spent    Left  Speed
21 100    17  100    17    0     0   1000      0 --:--:-- --:--:-- --:--:--  1062
22 {"Hello":"World"}
23 $ echo "No lint issues found."
24 No lint issues found.
```

This is the first half of the linit-test.job output

```
24  No lint issues found.
25  $ chmod +x bench_test.sh
26  $ ./bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d ":") 8
27  Starting benchmark on localhost port 32782, Threshold is 8
28  /usr/bin/httperf
29  httperf: warning: open file limit > FD_SETSIZE; limiting max. # of open files to FD_SETSIZE
30  avg: 0.8, threshold: 8
31  Making decision based on performance
32  it is ok
33  $ chmod +x dyn_bench_test.sh
34  $ ./dyn_bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d
   ":") 10 15
35  Deviation limit is set to 15
36  History of previous 10 values is:
37  python-fastapi-example dyn_httperf     0.8
38  python-fastapi-example dyn_httperf     0.7
39  python-fastapi-example dyn_httperf     0.8
40  python-fastapi-example dyn_httperf     0.8
41  Starting benchmark on localhost port 32782, Threshold is 0.775
42  /usr/bin/httperf
43  httperf: warning: open file limit > FD_SETSIZE; limiting max. # of open files to FD_SETSIZE
44  avg: 0.8, threshold: 0.775
45  Making decision based on performance
46  it is ok
48  Cleaning up project directory and file based variables                                    00:00
50  Job succeeded
```

This is the second half linit-test job output.

This is the most crucial part of the implementation. It is first connecting to the docker container which is proven to be successful. Then it runs the bench test script which then will point to the average of which was 0.8 compared to the threshold that was put. After that performance decision is decided, it runs the dyn_bench script. It sets the deviation limit which is set manually by the developer for this project. Then it runs through the previous 10 results, which are only 4 in this case, then it is successful in doing so, it will open the port of the container so it can run it's dynamic test which is proven to be below the threshold which means it passed the test. This section of the script went well.

```
56
57 deploy-job:      # This job runs in the deploy stage.
58   stage: deploy  # It only runs when *both* jobs in the test stage complete successfully.
59   environment: production
60   script:
61     - echo "Deploying application..."
62     - docker stop fastapi_$CI_PIPELINE_ID
63     - docker rm fastapi_$CI_PIPELINE_ID
64     - echo "Application successfully deployed."
```

This is the deploy-job stage of the gitlab-ci.yml file.

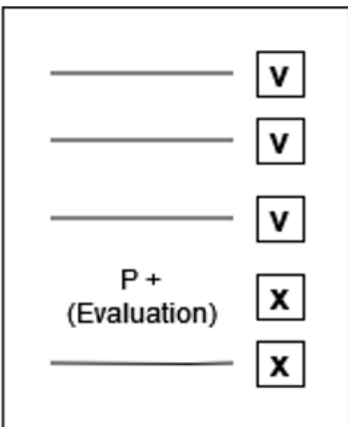
```
1  Running with gitlab-runner 15.11.0 (436955cb)
2    on cicd VSQVKw_e, system ID: s_22fd7788926f
3  Preparing the "shell" executor                                    00:00
4  Using Shell (bash) executor...
6  Preparing environment                                             00:00
7  Running on cicd...
9  Getting source from Git repository                                00:01
10 Fetching changes with git depth set to 50...
11 Reinitialized existing Git repository in /var/lib/gitlab-runner/builds/VSQVKw_e/0/s351645/python-fastapi-
   example/.git/
12 Checking out eba4e7c4 as detached HEAD (ref is master)...
13 Skipping Git submodules setup
15 Executing "step_script" stage of the job script                   00:01
16 $ echo "Deploying application..."
17 Deploying application...
18 $ docker stop fastapi_$CI_PIPELINE_ID
19 fastapi_2863
20 $ docker rm fastapi_$CI_PIPELINE_ID
21 fastapi_2863
22 $ echo "Application successfully deployed."
23 Application successfully deployed.
25 Cleaning up project directory and file based variables            00:00
27 Job succeeded
```

Output of the deploy-job script

This is the final stage as there is not much of significance to the project that happens here in a literal sense. In a figurative sense, this is where we would deploy our project. But what is happening here from 16 through 23 is the container is being stopped and removed. At the end we are presented with job succeeded.

What was completed.

Now with the output broken down, it is time to display what was finished by revisiting the previous sections of this paper. When we look at the scripts that are done. The only script that does what is being set out to be done, is the linit-test script as this contains both the performance tests and the dynamic tests comparing it to previously done results.



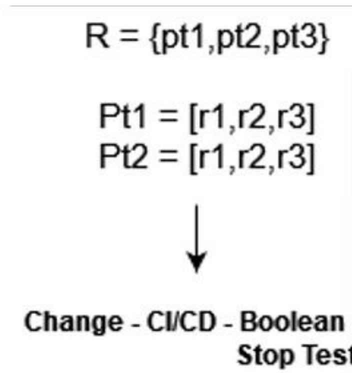
This is the list of test that is represented as a paper.

This figure above can be compared to this part of the section of the script below.

```
25  $ chmod +x bench_test.sh
26  $ ./bench_test.sh localhost $(docker port fastapi_$CI_PIPELINE_ID 80/tcp | grep 0.0 | cut -f 2 -d "
27  Starting benchmark on localhost port 32782, Threshold is 8
28  /usr/bin/httperf
29  httperf: warning: open file limit > FD_SETSIZE; limiting max. # of open files to FD_SETSIZE
30  avg: 0.8, threshold: 8
31  Making decision based on performance
32  it is ok
```

This is the part of the script that runs bench_test.

This is one of the performance tests of this script. You can see the average is being tested up towards the threshold and that it is being told that the decision that is being made is okay. This is a part of the performance test portion of the research question.



The figure above is the history of performance tests.

The figure above shows the history of performance tests being compared to what was done previously.

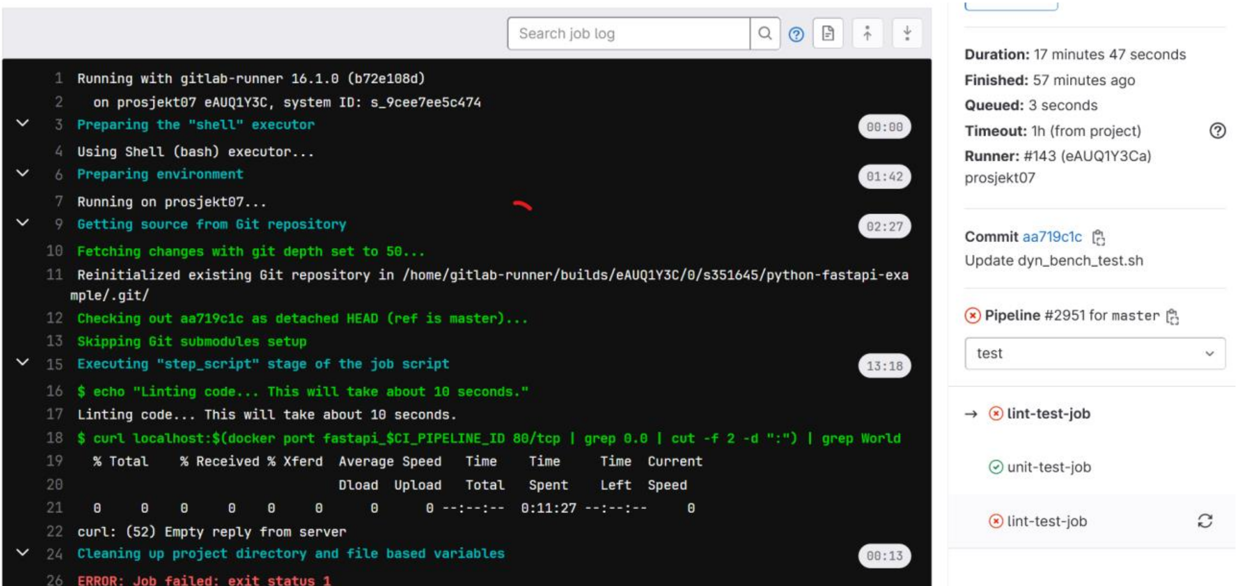
```
35  Deviation limit is set to 15
36  History of previous 10 values is:
37  python-fastapi-example dyn_httperf     0.8
38  python-fastapi-example dyn_httperf     0.7
39  python-fastapi-example dyn_httperf     0.8
40  python-fastapi-example dyn_httperf     0.8
41  Starting benchmark on localhost port 32782, Threshold is 0.775
42  /usr/bin/httperf
43  httperf: warning: open file limit > FD_SETSIZE; limiting max. # of open files to FD_SETSIZE
44  avg: 0.8, threshold: 0.775
45  Making decision based on performance
46  it is ok
```

This shows the history of performance tests and the threshold and average.

This test then compares what was done previously as shown in the lines 36 to 40 then running the average against the threshold and made it’s decision that the current average meets the threshold requirements. This was successful.

What was incomplete.

Now with what was not completed. Unfortunately feeding the negative values into the Openstack docker container did not work as intended due to maintenance on servers or other connection problems in general taking 30 minutes to connect to the server before losing the connection to it



The failed server connection.

7. Discussion

Now we are at the discussion section, this is where the journey will be discussed and how this thesis reached the end goal. The approach chapter will also be laid out as a reference for if the destination was reached the intended way, or if we got sidetracked along the way to where the thesis ended up.

Looking for background material

Looking for background material was at first viewed as a bit easy as there didn’t seem at first glance that anyone could have done something exactly like this thesis. There was an idea, that maybe there had been research done when it came to CI/CD pipelines because of the fact that it is not a new concept and at this point, since CI/CD Pipelines[46] has been a concept since 2009. There has been quite a lot of extensive research on this. Most of the words that had been used to find this information involved “CI/CD” or each word individually to an extent, or “Performance test” together.

When it comes to performance testing, research was also very extensively done, so there wasn’t a lot of articles that were too much of a deviation from the points made about performance tests in this thesis.

However, the surprise was big when it was revealed to me that there was quite a lot of research done on both CI/CD Pipeline, so finding articles was not that hard. A lot of articles are used, not explicitly CI/CD pipeline, but continuous integration or continuous delivery. Articles would also use words like progression regression and use benchmarks to find these which in essence is what a performance test is.

It was a surprise though how little information was revealed using the keywords “dynamic performance testing” in google scholar. There probably are other names in which this is done, however those names weren’t thought of. Some articles were found in regards to something of the sort however, even if it wasn’t satisfactory.

There was a fear of just reiterating points that had already been made. This fear disappeared as it became apparent that, even though some of these articles were in the same breath as this thesis, the ones found were not very specifically in a CI/CD Pipeline itself or dynamic performance testing, but more in the continuous integration or delivery process. Even though these two are similar, the process being in a pipeline that is automatically processed is different as these articles did not seem to be going in this direction.

Even if there was some initial fear in regard to just reiterating points other people had made, I believe adding another opinion, even if it is the same opinion to a topic, is beneficial to the field as, if more people came to the same conclusion, even if it may be disproven later due to more information being presented, it would confirm at the present moment further what already has been confirmed.

Finding the right technology

Before finding there had been a settlement on what technology that was supposed to be used, the one that ended up being used, ended up being Gitlab. However, Jenkins was an option that was considered before Gitlab was thought of. The search for technology to use for this thesis was through google searches with keywords “top 10 CI/CD pipelines” and go through a list, however there was only one option tested first, and this was Jenkins before going to Gitlab.

Gitlab has far more familiarity as it was used in previous courses before writing this thesis while Jenkins was new. Also, Jenkins required plugins to be used while Gitlab did not. Another determining factor for choosing Gitlab was the easiest time to connect with the Openstack virtual machine that was necessary for the project to come together.

Looking for a technology to host this project was a bit of a challenge because there was not a lot of familiarity with virtual machines in general. However, the decision to pick an Openstack server was based on familiarity. This tool can allow the user to create virtual machines with an IP address. For safety reasons as well as familiarity, Openstack was also used because the Oslo metropolitan university, has their own set of accounts that they pass to students, which makes it so, if there were any technical errors that weren’t solvable alone, it would be easier to consult the administrative personnel in charge of these for advice.

Openstack did prove to create a lot of problems however as there would be problems with the server which the project is hosted on. Which means if the connection between Gitlab and the server is somehow compromised, it could be disastrous for the project depending on whether crucial changes are needed.

Implementation of the prototype.

Creating the prototype was a bit more difficult than originally imagined. There were a lot of parts that had an idea of how it was going to be put together. A CI/CD Pipeline was needed and a performance test tool or something of the sort was needed. I was not aware that I would need to use an Openstack server, but I needed something to run it on with the docker container.

Now putting these three together was very challenging as, even though I did have docker container experience previously, I had not actually run a docker container with anything in it before. Building the pipeline itself was the easiest part as was creating the virtual machine on the Openstack server.

However, the difficulty came from writing the shell script needed to execute the code and the specific part of the Gitlab jobs as the part where the bread and butter of the project happens, required a lot of trial and error due to the sensitivity of the gitlab-ci.yml file.

There still is disappointment in that not all the wanted features were fulfilled, but due to a lack of resources, it wasn’t put together optimally. The reason why is due to maintenance needed on the Openstack server, changes that were crucial for this project were compromised. The dynamic test itself that is. All that needed to be done is feed the low values to the database, and you would have a dynamic test that created a threshold of previous values that would give a failed test and the dynamic part of this would be complete.

The graph however was not something that was feasible with the short time span.

The Thesis Structure

One of the least satisfactory parts of this thesis is the sources section. Writing down sources simply and then going back to change it later, was primarily what the plan was. However, seeing how time-consuming this project was, it was hard to find the time to make it look a bit more professional. A fear of mine is I might have forgotten crucial sources, but I do not believe so

after going through the thesis. However, something might still have been missing, and I just am not aware of it.

Also not giving the proper names and numberings to subsections of under categories was a disappointment too, and a lack of time.

Being Educated

Throughout this thesis, a lot was learnt during the writing process. Not having written a thesis before was being taught as it was being written, even though a bachelors is similar, a bachelor student feels more like a construction worker helping on his part, while the master's feels more like one is an architect.

A bachelor is more interested in the answer to a question that's been asked, while a master is more about getting closer to an answer to question that doesn't have an answer.

There were parts of this thesis that were educational in the sense that for me, who did not know too much about dynamic performance testing and CI/CD pipelines, a lot of this was new material I did not know too much about. Seeing all the different ways that performance tests could be used was interesting as it just struck me as something that primarily only had a single use, which was to just look at the performance of something, to make that performance improve. It did not occur to me that one could use it for more than just making something better, as one could use performance tests for security reasons, or even for bug searching.

Making one's own pipeline was very interesting. I did not know that Gitlab had their own pipeline that they could use, as I was primarily using it for code repository purposes. Making the yml file was very hard to do as it required a lot of trial and error unfortunately, however the yml file's very basic structure of the different jobs was easy to understand, and I can see how this could be beneficial for me outside of this project.

8. Conclusion

The goal of this project was to *Investigate the implementation of dynamic performance testing in CI/CD Pipeline.* This was proven to have some useful advantages , but it was also proven that it could have been further improved as the core concept wasn't successful, the tools necessary and the time constraints made it problematic to get to the ideal end.

There was some research that was done that was very familiar to what was done in this thesis in the performance testing department, but not in the dynamic performance testing area as it was hard to find something for such.

The CI/CD Pipeline that was used was Gitlab as it was easily accessible after trying another technology, and it was connected to an Openstack server through a Gitlab runner that it was ran on. After trying one or two other technologies, Gitlab proved to be good enough. It had very basic setups that allowed a yaml file to run the pipeline by showing each step of the pipeline, even though the file was very sensitive as the slightest mishap became problematic.

The dynamic performance test had proven itself to be useful in more than just one way. It was difficult to set up as there were problems trying to get it to run, however it revealed itself to be useful. There are a lot of things one can use the dynamic performance tests to in a pipeline that benefits both the company that are using them in order to gauge how it will benefit their infrastructure, and developers who will catch bugs easier, or performance issues easier during the pipeline before it is sent out into production or deployed. Or in terms of the history part, see how it performed before to make sure if the performance of something was done worse in future iterations, they could go back to change it to a previous build.

Sources

[1] Evolution of Books[Infographic]. online-spellcheck.com. (n.d.). https://blog.online-spellcheck.com/general/the-evolution-of-books-infographic/

[2] Kienitz, P. (2019, August 27). Most expensive software mistakes. One Beyond. https://www.one-beyond.com/most-expensive-software-mistakes/

[3] Functional testing vs non-functional testing. Software Testing Help. (2023, June 24). https://www.softwaretestinghelp.com/functional-testing-vs-non-functional-testing/

[4] Vokolos, F. I., & Weyuker, E. J. (1998, October). Performance testing of software systems. In Proceedings of the 1st International Workshop on Software and Performance (pp. 80-87).

[5] Burke, J. (2022). "Throughput." from https://www.techtarget.com/searchnetworking/definition/throughput.

[6] Avritzer, A., Kondek, J., Liu, D., & Weyuker, E. J. (2002, July). Software performance testing based on workload characterization. In Proceedings of the 3rd International Workshop on Software and Performance (pp. 17-24).

[7] Courtemanche, M. "What is DevOps? The ultimate guide." from https://www.techtarget.com/searchitoperations/definition/DevOps.

[8] . "What Is DevOps?". from https://newrelic.com/devops/what-is-devops#toc-introduction.

[9] Gokarna, M., & Singh, R. (2021, February). DevOps: a historical review and future works. In 2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS) (pp. 366-371). IEEE.

[10] M, R. "Pros and Cons of the DevOps principles Explained Clearly." from https://www.kovair.com/blog/pros-and-cons-of-the-devops-principles/.

[11] Paul Duvall, S. M., Andrew Glover (2007). Continuous Integration: Improving Software Quality and Reducing Risk.

[12] Fowler, M., & Foemmel, M. (2006). Continuous integration.

[13] . "CI CD." from https://www.synopsys.com/glossary/what-is-cicd.html.

[14. "What is continous integration (CI)?". from https://about.gitlab.com/topics/ci-cd/benefits-continuous-integration/.

[15] Penington, J. (2019). "The Eight Phases of a DevOps Pipeline." from https://medium.com/taptuit/the-eight-phases-of-a-devops-pipeline-fda53ec9bba.

[16] (2022). "9 Tool Types for Your DevOps Toolbelt." from https://www.dbmaestro.com/blog/database-devops/9-tool-types.

[17] . "How to use Slack: your quick start guide." from https://slack.com/help/articles/360059928654-How-to-use-Slack--your-quick-start-guide.

[18] . "The best platform for cross functional work." from https://asana.com/?noredirect.

[19] . "Database Source Control Simplified." from https://www.dbmaestro.com/database-source-control.

[20] Simplilearn (2023). "What is JIRA? How to use Jira Testing Software Tool." from https://www.simplilearn.com/tutorials/jira/what-is-jira-and-how-to-use-jira-testing-software.

[21] . "Introduction to Puppet." from https://www.puppet.com/docs/puppet/6/puppet_overview.html.

[22] Rehkopf, M. "Continous Integration Tools." from https://www.atlassian.com/continuous-delivery/continuous-integration/tools.

[23] "Top 15 Automated Testing Tools." from https://katalon.com/resources-center/blog/automation-testing-tools.

[24] . "Azure DevOps." from https://azure.microsoft.com/en-us/products/devops/.

[25] . "Azure DevOps." from https://www.finn.no/job/fulltime/search.html?q=Azure+DevOps&sort=RELEVANCE.

[26] . "Database Source Control Simplified." from https://www.dbmaestro.com/database-source-control.

[27] . "Continous Delivery." from https://continuousdelivery.com/.

[28] Pittet, S. "Continous Integration vs Delivery vs Deployment." from https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment.

[29] Santos, I. (2021). "The CI/CD Odyssey." from https://faun.pub/the-ci-cd-odyssey-1cc058088feb.

[30] Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. IEEE software, 32(2), 50-54.

[31] Prytulenets, A. (2022). "A Brief History of Software Testing." https://dogq.io/blog/a-brief-history-of-software-testing/.

[32] Singh, S. K., & Singh, A. (2012). Software testing. Vandana Publications.

[33] . "What is Performance Testing." from https://www.microfocus.com/en-us/what-is/performance-testing.

[34] . "Performance Testing, Best Practices, Metrics & More." from https://www.tricentis.com/learn/performance-testing.

[35] (2023). "Types of software Testing: Different Testing Types With Details." from https://www.softwaretestinghelp.com/types-of-software-testing/#2_Performance_Testing.

[36.1] Preston, M. "System Development Life Cycle Guide." from https://www.clouddefense.ai/blog/system-development-life-cycle.

[36.2]. "Jenkins." from https://www.jenkins.io/.

[37] . "CI/CD Pipeline: What,Why & How to Build The Best One." from https://katalon.com/resources-center/blog/ci-cd-pipeline.

[38] Kosnik, M. (2022). "Why do software projects fail? Most common reasons." from https://thecodest.co/blog/why-do-software-projects-fail-most-common-reasons/.

[39] Daly, D., Brown, W., Ingo, H., O'Leary, J., & Bradford, D. (2020, April). The use of change point detection to identify software performance regressions in a continuous integration system. In Proceedings of the ACM/SPEC International Conference on Performance Engineering (pp. 67-75).

[40] Waller, J., Ehmke, N. C., & Hasselbring, W. (2015). Including performance benchmarks into continuous integration to enable DevOps. ACM SIGSOFT Software Engineering Notes, 40(2), 1-4.

[41] Hershkovitch, D. (2021). "Write a stageless CI/CD Pipeline using GitLab 14.2." from https://about.gitlab.com/blog/2021/08/24/stageless-pipelines/.

[42] Tiagolo (2022). "unicorn-gunicorn-fastapi-docker." from https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker.

[43] . "Httperf." from https://www.mervine.net/performance-testing-with-httperf.

[44] . "httperf(1) - Linux man page." from https://linux.die.net/man/1/httperf.

[45] (2021). "Update." https://www.facebook.com/senpaisauces/photos/a.142863883749091/637121477656660/?type=3.

[46] "A brief history of CI/CD." https://jhall.io/archive/2021/09/26/a-brief-history-of-ci/cd/.

[47] Anzt, H., Chen, Y. C., Cojean, T., Dongarra, J., Flegar, G., Nayak, P., ... & Wang, W. (2019, June). Towards continuous benchmarking: An automated performance evaluation framework for high performance software. In Proceedings of the platform for advanced scientific computing conference (pp. 1-11).

[48] Hamilton, T. (2023). "Static vs Dynamic Testing: Difference Between Them." from https://www.guru99.com/static-dynamic-testing.html.

[49] Gaute Borgenholt, K. B., Paal Engelstad (2013). "Audition: A DevOps-Oriented service optimization and testing framework for cloud enviorments."

[50] Anzt, H., Chen, Y. C., Cojean, T., Dongarra, J., Flegar, G., Nayak, P., ... & Wang, W. (2019, June). Towards continuous benchmarking: An automated performance evaluation framework for high performance software. In Proceedings of the platform for advanced scientific computing conference (pp. 1-11).

[51] Reichelt, D. G., & Kühne, S. (2018, April). How to detect performance changes in software history: Performance analysis of software system versions. In Companion of the 2018 ACM/SPEC international conference on performance engineering (pp. 183-188).

[52] Heger, C., Happe, J., & Farahbod, R. (2013, April). Automated root cause isolation of performance regressions during software development. In Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (pp. 27-38)

[53] Bulej, L., Bureš, T., Horký, V., Kotrč, J., Marek, L., Trojánek, T., & Tůma, P. (2017). Unit testing performance with stochastic performance logic. Automated Software Engineering, 24, 139-187.

[54] Pratama, M. R., & Kusumo, D. S. (2021, August). Implementation of continuous integration and continuous delivery (ci/cd) on automatic performance testing. In 2021 9th International Conference on Information and Communication Technology (ICoICT) (pp. 230-235). IEEE.

[55] Moghadam, M. H. (2019, August). Machine learning-assisted performance testing. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1187-1189).

[56] Laaber, C. (2019, July). Continuous software performance assessment: Detecting performance problems of software libraries on every build. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 410-414).