[1]1

Bachelor Thesis

Tebah, Wolfgang Azipon

Development of a Business Process and Test Automation for Continuous Integration and Continuous Deployment

Fakultät Technik und Informatik
Studiendepartment Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Tebah, Wolfgang Azipon

Development of a Business Process and Test Automation for Continuous Integration and Continuous Deployment

Bachelor Thesis submitted as part of the bachelors examination
for the Bachelor of Science in Information Engineering
Department of Information and Electrical Engineering
Faculty of Technology and Computer Science
at the Hamburg University of Applied Sciences

Supervising Examiner: Prof. Dr. rer. nat. Wolfgang Renz
Second Examiner: Prof. Dr. Annabella Rauscher-Scheibe

Submitted on: 22. August 2022

Tebah, Wolfgang Azipon

Thema der Arbeit
Entwicklung einer Geschäftsprozess- und Testautomatisierung für Continuous Integration und Continuous Deployment

Stichworte
DevOps, Continuous Integration, Continuous Delivery, Bereitstellung, Docker

Kurzzusammenfassung
Aufgrund des hohen Wettbewerbs in der Softwareentwicklungsbranche sind Unternehmen heute mehr denn je gefordert, qualitativ hochwertige Software in kürzester Zeit zu produzieren. Ein Teil der kontinuierlichen Software-Engineering-Prozesse zwischen den Entwicklungs- und IT-Operations Team wird auf sich wiederholende Aufgaben untersucht, die automatisiert werden können. Ein Geschäftsprozess wird implementiert, der die Probleme zwischen beiden Teams weiter entschärft. Das Entwicklungsteam produziert kontinuierlich Code, und das IT-Betriebsteam muss ständig die Auswirkungen des gepushten Codes auf die gesamte Anwendung überwachen. Dieser Prozess, der eine Reihe von Tests umfasst, erfolgt in wiederholten Zyklen. Das IT-Operations Team ist mehr an den Ergebnissen der Tests interessiert, aber es verbringt oft mehr Zeit damit, die Tests auszuführen, als die Ergebnisse zu analysieren.

Diese Tests sind in zwei Hauptkategorien unterteilt: statische Codeanalysen und dynamische Codeanalysen. Diese Arbeit soll die Automatisierung statischer Codeanalysetests vollständig implementieren und die dynamischen Codeanalysetests teilweise automatisieren. Es soll eine erweiterbare Automatisierungsinfrastruktur aufgebaut werden, in der zukünftige dynamische Codeanalysetests automatisiert werden können. Die Infrastruktur baut gleichzeitig das Projekt auf, führt statische Codeanalysen durch, stellt Anwendungen bereit, führt dynamische Codeanalysen durch und meldet Ergebnisse. Das Ergebnis dieser Bachelorarbeit ist eine Automatisierungspipeline, die einen Software-Qualitätssicherungsprozess automatisiert.

Diese Infrastruktur soll dann die Lücke zwischen den beiden oben genannten Teams schließen. Dies erhöht die Effektivität und Geschwindigkeit des IT-Operations Team und beschleunigt dadurch den gesamten Continuous Software Engineering (CSE)-Prozess. Die Testergebnisse müssen schließlich manuell analysiert werden, bevor bestätigt wird, ob die Anwendung für den Einsatz in der Produktion geeignet ist. …

Tebah, Wolfgang Azipon

Title of the paper
Development of a Business Process and Test Automation for Continuous Integration and Continuous Deployment

Keywords
DevOps, Continuous Integration, Continuous Delivery, Deployment, Docker

Abstract
Due to the high competition in the software development industry, Companies are now required more than ever to produce high-quality software in the shortest possible time. Part of the continuous software engineering processes between the development and IT operations teams are examined for repetitive tasks which can be automated. A business process is implemented, which further mitigates the issues between both teams. The development team continuously produces code, and the IT operations team must constantly monitor the effects of the pushed code on the entire application. This process which involves a series of tests, occurs in repeated cycles. The IT operations team is more interested in the results of the tests, but they often spend more time trying to execute the tests than analyzing the results.

These tests are divided into two major categories: static code analyses and dynamic code analyses. This thesis shall fully implement the automation of static code analysis tests and partially automate the dynamic code analysis tests. An extendable automation infrastructure shall be constructed, where future dynamic code analysis tests can be automated. The infrastructure shall simultaneously build the project, execute static code analyses, deploy applications, execute dynamic code analyses, and report results. The outcome of this thesis is an automation pipeline that automates a software quality assurance process.

This infrastructure shall then bridge the gap between the two teams mentioned above. This increases the IT operations team's effectiveness and speed, thereby accelerating the whole CSE process. The test results must finally be manually analyzed before confirming if the application qualifies to be deployed into production. …

Dedication

I dedicate this thesis to my Family and friends

Acknowledgement

I want to thank my supervising professors and my supervisor at work, Mr. Christopher Gudat, for their help throughout this thesis. I would also like to thank Mr. Carsten Polenski, who was also of great help.

Finally, I would like to thank Mr. Sebastian Krach for his tremendous help while writing this thesis.

Contents

List of Figures ix

List of Tables x

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Purpose . . . 1
- 1.3 Scope . . . 2

2 Requirements Engineering 5
- 2.1 Business Process (BP) . . . 5
  - 2.1.1 Business process without automation . . . 5
  - 2.1.2 Business process with automation . . . 6
- 2.2 Use case analyses . . . 7
- 2.3 Functional requirements . . . 11
- 2.4 Work packages . . . 13
  - 2.4.1 Present state of application . . . 13
  - 2.4.2 Required tools . . . 13

3 Continuous Integration Concepts 15
- 3.1 Technology-facing tests that support programming . . . 16
- 3.2 Static code analyses . . . 17
- 3.3 Continuous application deployment . . . 21

4 Continuous Delivery Concepts 26
- 4.1 Acceptance test . . . 26
  - 4.1.1 Business-facing test that supports programming . . . 26
  - 4.1.2 Technology-facing tests that critique the project . . . 27
  - 4.1.3 Business-facing tests that critique the project . . . 28

5 Implementation 30
- 5.1 System architecture . . . 30
- 5.2 Implementation of continuous integration . . . 31
  - 5.2.1 Build and unit test . . . 32
  - 5.2.2 Static code analyses SCA . . . 33
  - 5.2.3 Delivery package . . . 33
  - 5.2.4 Deployment . . . 34

5.3 Implementation of acceptance test 35
5.4 Alternative approach 37

6 Pipeline test results **39**
6.1 Static code analyses results 40
6.2 Acceptance test results 45
6.3 Functional requirements validation 46

7 Results and conclusion **47**
7.1 Requirements validation 48
7.2 Future Work 49

Bibliography **50**

Appendix **52**
1 Jenkinsfile 52
2 docker-compose.yml 58
3 setup.sh 60

Glossary **62**

Acronyms **63**

List of Figures

1.1 Continuous software engineering [1] 3
1.2 Relationship between continuous integration, delivery, and deployment [2] . . 4

2.1 Business process without CICD pipeline 6
2.2 Business process with CICD pipeline 7
2.3 Use case diagram 8
2.4 Detailed UC1 9
2.5 Detailed UC2 10
2.6 Detailed UC3 10
2.7 Detailed UC4 11

3.1 Testing quadrant diagram [3]. 16
3.2 Compilation steps of java code 20
3.3 Difference between VM and Docker container 22
3.4 High-level overview of major Docker components 23

5.1 General system architecture. 30
5.2 Continuous Integration flow chat 32
5.3 Selenium grid architecture 36
5.4 Alternate system architecture 38

6.1 Thesis Delivery Package 39
6.2 Overview of Pipeline Trend 41
6.3 Unit test report 42
6.4 Spotted PMD and Spotbugs High Severity Bug 43
6.5 Static analyses warnings 43
6.6 Bug results categories 44
6.7 Dependency check 44
6.8 Dependency-Check report suggestion 45

List of Tables

1.1 Requirements 2

2.1 Use Cases 7
2.2 Traceability Matrix 8
2.3 Functional requirements 12
2.4 Use cases and functional requirements traceability 12
2.5 Work packages 13

6.1 Validation of functional requirements 46

7.1 Project validation 48

Listings

5.1 CI results publishers 33
5.2 Groovy find versioned folder 33
5.3 populateDB.sh script in database container 34
5.4 Selenium Hub client setup 36

1 Introduction

1.1 Motivation

There is increased competition in the software development industry, hence the need for organizations to be more agile and allocate resources to develop and deliver high-quality software at a much-accelerated pace [2]. Organizations where the daily quality of the software is to be assured usually end up creating an environment of monotonous and inaccurate testing and deployment. Carrying out frequent or daily manual software Quality Assurance (QA) tests lead to less synchronization between the operations and development teams. Manual testing can be monotonous and hectic, making software release days tense [3]. The accumulation of errors and poor-quality code also causes long working hours during the release period.

One attempted approach to solving this is employing many QA engineers, making the project even more expensive and harder to manage. Nowadays, some companies are specialized just in testing. The outsourcing of tests is another costly approach to solving this issue. Large corporations like Amazon's AWS and Azure offer a cloud computing platform that attempts to a great extent to solve these issues. They are not free but also not too expensive, and the entire automation project and culture still need to be implemented and placed on these platforms. One of the less expensive options is to have a personal cloud infrastructure like VMware and then design an automation project on it. This also comes with some maintainability issues.

1.2 Purpose

The primary purpose of this thesis is to develop an infrastructure that attempts to bridge the gap between an organization's development team and IT operations team. It is done by continuously implementing static and dynamic tests on an application to have an idea of the state of an application constantly. The frequency and other parameters of this process's continuity depend on the organization's culture. The initial text to kick-start this project is "Develop a CICD Pipeline that builds the eCMS-Monitoring project from Bitbucket, deploys the server, and then runs a series of tests on it. The test should be executed every weekday before 6:00 am, and the result of the failed test is sent to the responsible developers and project manager. The results should be emailed to the actors if there are failures, but just an “OK” should be sent if it is successful. A nice-to-have will be to create a Jira ticket with the test errors to the respective developer(s)". From this text, the following requirements are extracted.

1.2.1 Requirements

#	**Requirements**
REQ1	The establishment of a test environment where software can automatically be deployed.
REQ2	An automated process for the dynamic quality assurance of the application deployed on the established test environment.
REQ3	An automated process for software integration and static software quality assurance.
REQ4	The project manager(s) and developer(s) shall be able to get daily notification about the status of the application. Success scenario: just a success message, failure scenario: an elaborate report of the executed tests and create a Jira ticket.

Table 1.1: Requirements

1.3 Scope

This thesis aims to develop continuous integration (CI) and continuous delivery (CD) pipeline, which automates tests and implements a business process that facilitates work between a company’s development and IT operations team. The pipeline shall automatically carry out the CI and CD processes. The CI phase is mostly static code analysis; meanwhile, the CD phase is mostly dynamic code analysis. The pipeline shall get the software project from its repository, build the project, run a series of static code tests on the project, gather the built artifacts, deploy them in a test environment, perform dynamic code analyses, and finally collect the test results.

To be more explicit about the scope of this thesis it will be clearer if we zoom out and have a bird’s-eye view of the entire system and its position in an organization. CSE is an organizational setup or methodology whereby software is developed, deployed, and quick feedback is gotten from software and customers in a very rapid cycle [1]. From Figure 1.1 below, CSE is divided into three phases, namely business strategy and planning, development, and operations.

Figure 1.1 below shows that this thesis falls between the development and the operation phases, famously known as DevOps. This means the solutions in this thesis shall be centered between these 2 phases. Making the processes in the DevOps phase continuous can be achieved by implementing three software development activities: CI, CD, and continuous deployment (CDE). These three activities or processes interact differently to achieve the desired outcome as in Figure 1.2 below. CD mainly focuses on ensuring the application successfully passes the automated tests (Acceptance test) and quality checks, i.e., ensuring that the application is always at a production-ready state. Meanwhile, CDE does the same but goes a little further to automatically deploy the application into the production environment [2]. In other words, the CD is a subset of CDE.

Furthermore, the exact scope of this thesis can be seen clearly in Figure 1.2 below. A Continuous Integration and Continuous Delivery (CICD) pipeline will be investigated, and the requirements above will be implemented. CDE is meant for organizations that build software and offer life services with their software, like Facebook.

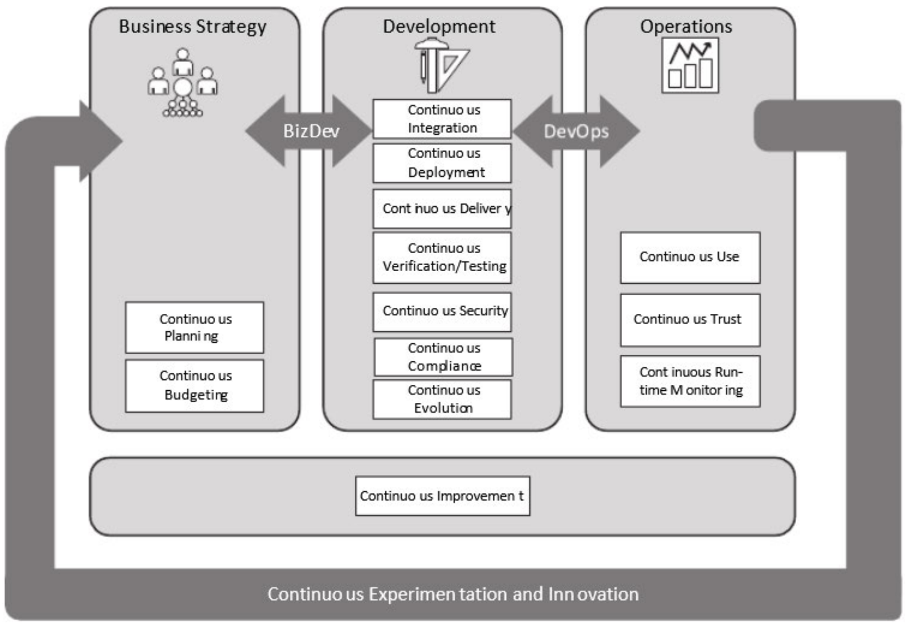


Figure 1.1: Continuous software engineering [1]

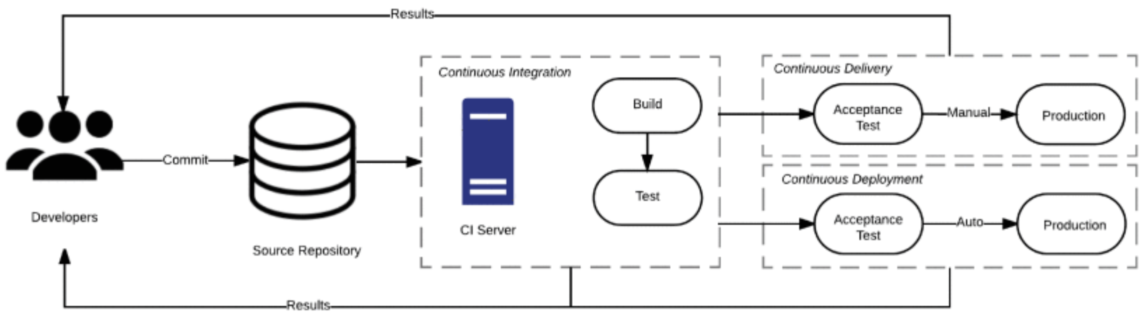


Figure 1.2: Relationship between continuous integration, delivery, and deployment [2]

2 Requirements Engineering

The initial kick-start text has been approved after multiple requirements elicitation. The listed requirements generated from the kick-start text have also been validated. This chapter shall thoroughly analyze the initial requirements to understand what is to be implemented. This process will lead to more questions and a deeper understanding of the project and the project manager's expectations. Let us initially analyze the business process and see how this thesis fits into the business project.

2.1 Business Process (BP)

There are a series of steps required to produce high-quality software effectively. This chapter shall cover the present business process without automation and that with automation. This will further show how automation can influence an organization's business process.

2.1.1 Business process without automation

Let us look at the present process without a CICD server for automation. There is a close link between the customers and the organization, which is already a very positive move in an agile environment. The purple arrow indicates a circular process between the IT operations and the development teams in figure 2.1. This process sometimes involves long meetings, which are more about configuring the software environment and less about actual issues with the software. The more tickets are created, the more testing is required, but not necessarily, the more development is needed compared to testing. Sometimes, changes or tickets are just not tested due to time constraints. The assumption is that if the following tests work, the previous change must have been okay. All these assumptions are the "devils" that come back during delivery.

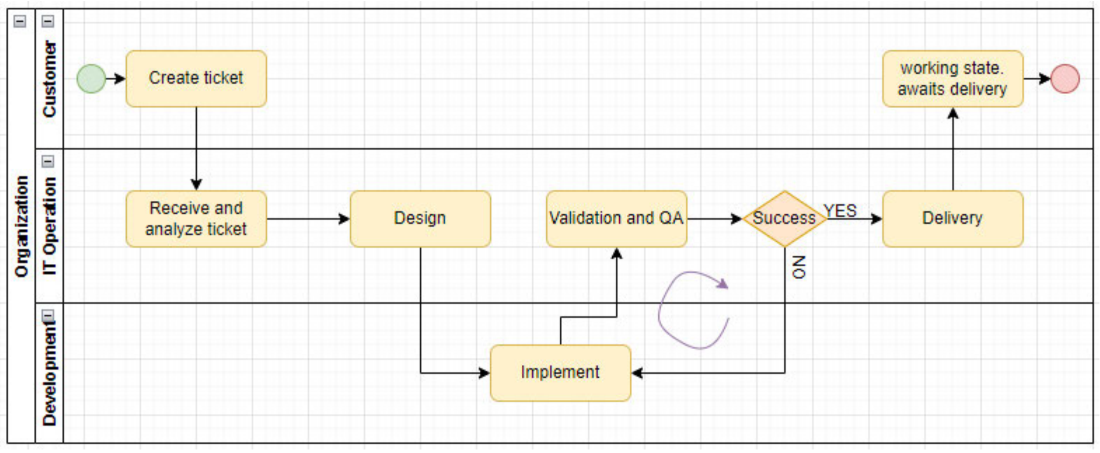


Figure 2.1: Business process without CICD pipeline

2.1.2 Business process with automation

Let us introduce an automation system between the IT operations and development teams and see its impact on the business process. In the previous section, we have seen the business process implemented by the organization where this thesis is being implemented. Figure 2.2 is an update of the business process after introducing an automation system (CICD-server). As seen on the diagram, the repetitive cycle has been shifted between the IT operations and development teams to between the development team and the CICD-server. It is important to note that the CICD-server does not entirely replace QA in the operations team but reduces monotonous tasks. This leads to high-quality and standardized testing because machines are much more accurate in testing than humans [4]. An excellent analogy to explain this logic is the evolution of the company Netflix. Netflix started by renting out DVDs to its customers via mail and then moved to offer the same services on the internet. The elimination of the overhead caused by mailing DVDs to a more automated platform on the internet, which implicitly changed their business process, provided room for Netflix to rent movies online and produce films and TV shows. This further shows that the automation system does not make the IT operations team have less work and thereby lay off some workers, but instead creates room for more sophisticated processes in the IT operations.

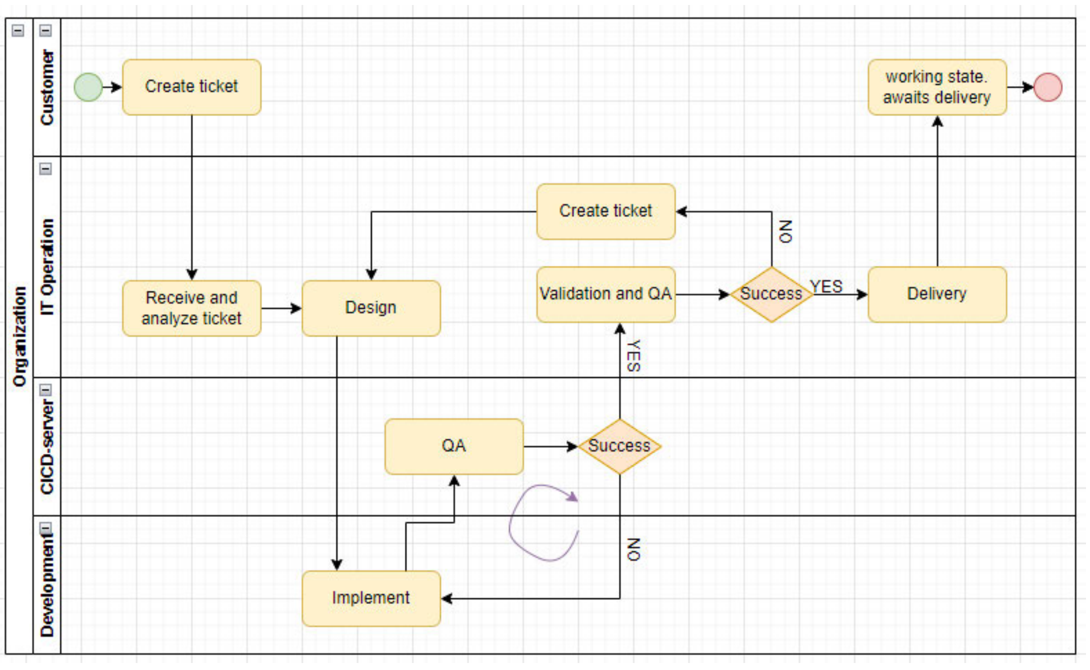


Figure 2.2: Business process with CICD pipeline

2.2 Use case analyses

2.2.1 Use Case

The following use cases are derived from the requirements. More details about the use cases can be found in section 5.1.4.

#	**Use Case**
UC1	Continuous integration (CI)
UC2	Continuous application deployment (CAD)
UC3	Dynamic test (DT)
UC4	Results

Table 2.1 Use Cases

2.2.2 Use case diagram

It is a distributed system. Therefore, different actors and sub-systems communicate in various forms to achieve different goals. It is essential to establish all the actors and sub-systems present.

That way, it will be easier to map the communication between the actors and sub-systems. Figure 2.3 below is the use case diagram of the system. This diagram broadens the knowledge about the interdependence of the actors, use cases, and sub-systems.

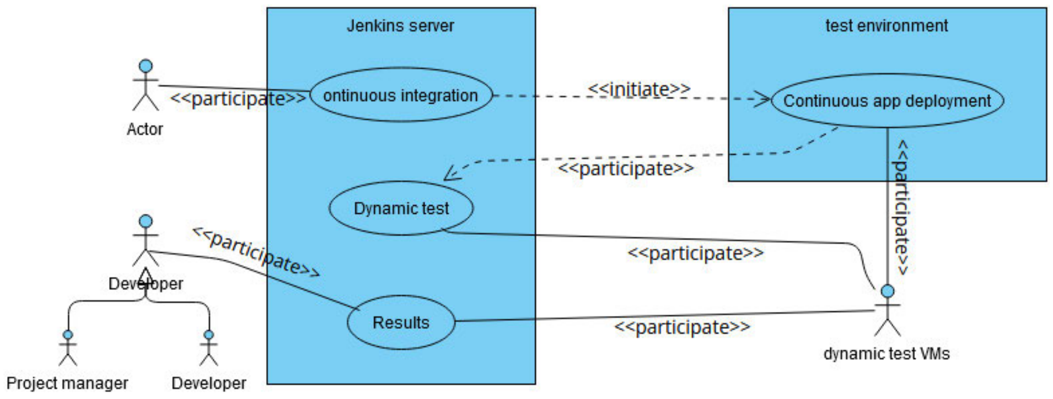


Figure 2.3: Use case diagram

2.2.3 Traceability matrix

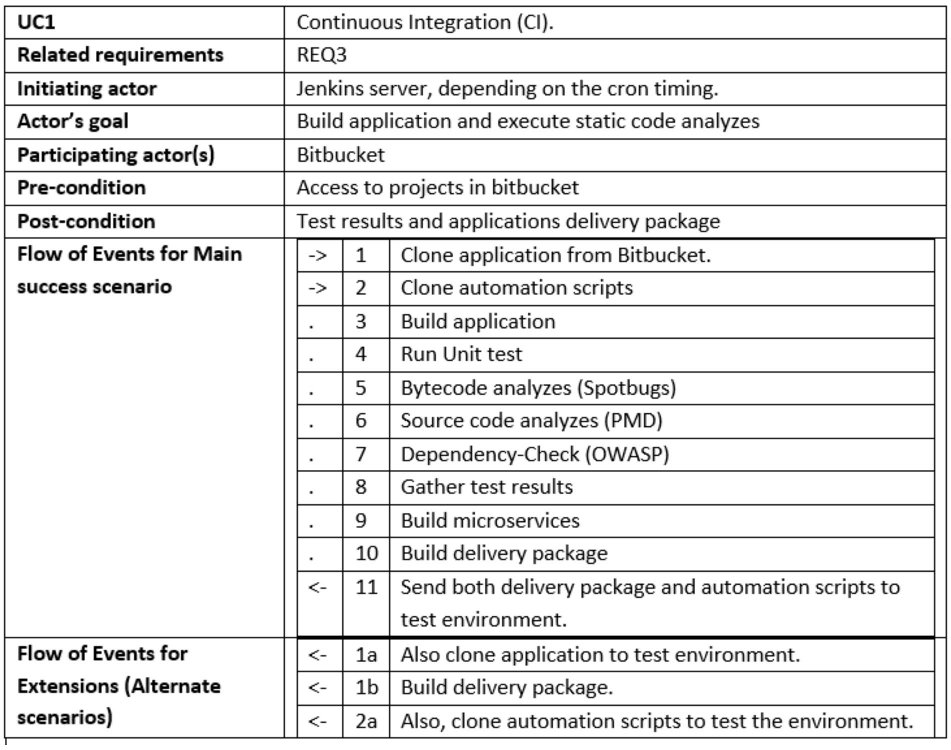
It is essential to ensure that the derived use cases adequately satisfy the requirements. A traceability matrix is used for the validation as in table 2.2. This reassurance is vital. Otherwise, a different product is archived as to what was intended.

#	UC1	UC2	UC3	UC4
REQ1		X		
REQ2			X	
REQ3	X			
REQ4				X

Table 2.2 Traceability Matrix

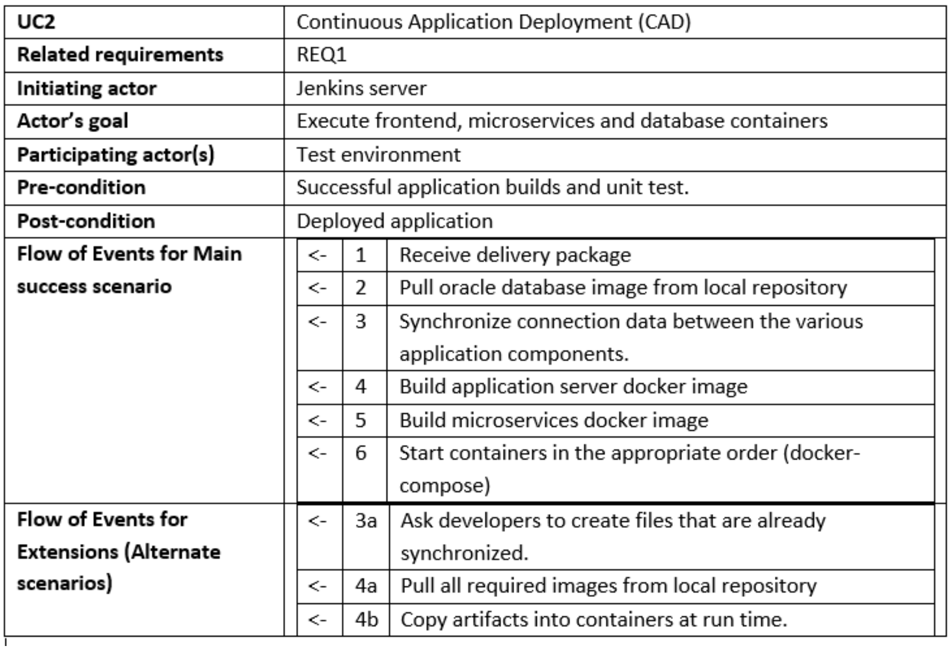
2.2.4 Detailed use case analyses

Use cases are phrases or words that describe a functional aspect or scenarios of an application or project. Each design is a combination of complex underlying tasks. To have a clear and deeper understanding of how to archive each use case, it is necessary to analyze the flow of events of each use case. This analysis also tells us what each use case receives and what it sends out. This will better help in establishing the functional and non-functional requirements of the system as well as estimating the amount of time required for the complete execution of the project. Below is a draft of the detailed use case analyses.



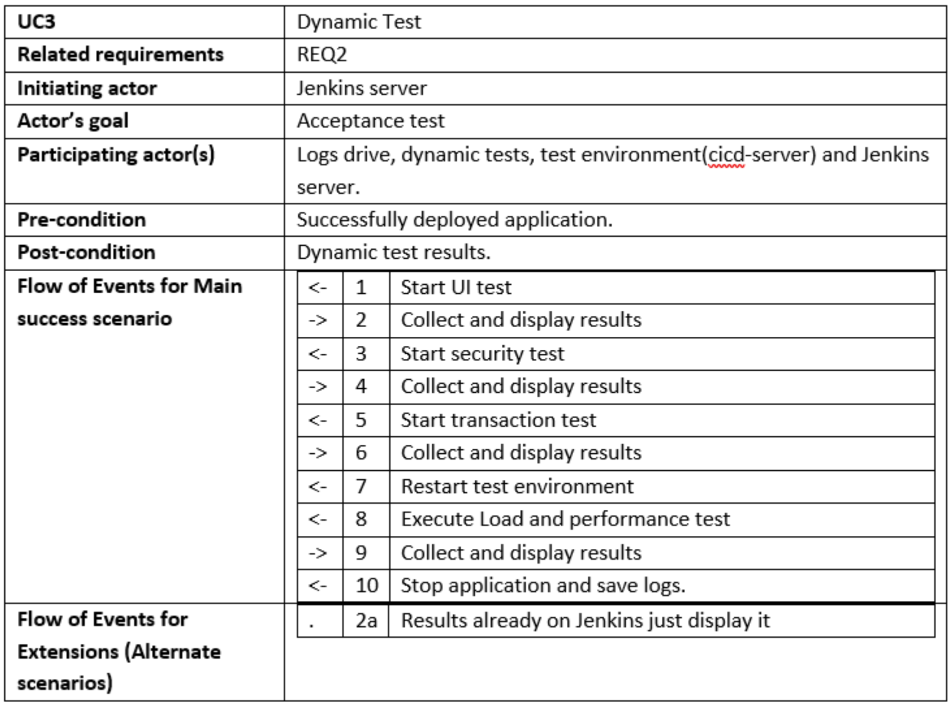
UC1	Continuous Integration (CI).		
Related requirements	REQ3		
Initiating actor	Jenkins server, depending on the cron timing.		
Actor's goal	Build application and execute static code analyzes		
Participating actor(s)	Bitbucket		
Pre-condition	Access to projects in bitbucket		
Post-condition	Test results and applications delivery package		
Flow of Events for Main success scenario	->	1	Clone application from Bitbucket.
	->	2	Clone automation scripts
	.	3	Build application
	.	4	Run Unit test
	.	5	Bytecode analyzes (Spotbugs)
	.	6	Source code analyzes (PMD)
	.	7	Dependency-Check (OWASP)
	.	8	Gather test results
	.	9	Build microservices
	.	10	Build delivery package
	<-	11	Send both delivery package and automation scripts to test environment.
Flow of Events for Extensions (Alternate scenarios)	<-	1a	Also clone application to test environment.
	<-	1b	Build delivery package.
	<-	2a	Also, clone automation scripts to test the environment.

Figure 2.4: Detailed UC1



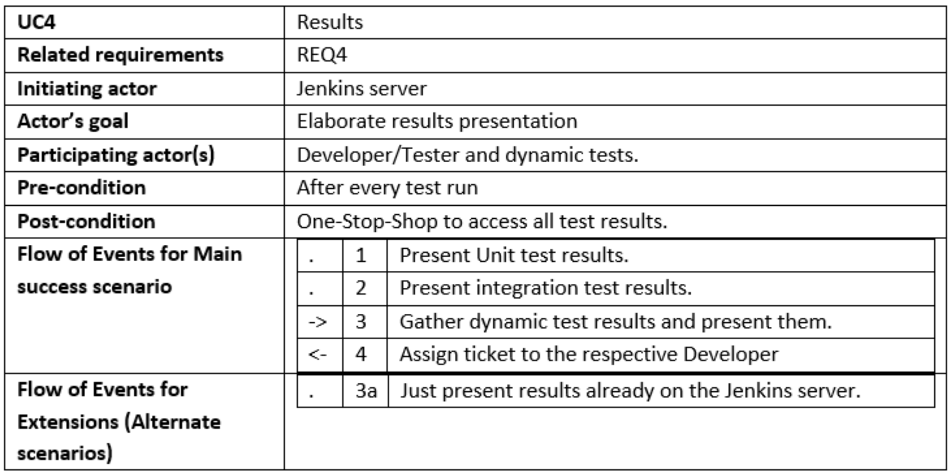
UC2	Continuous Application Deployment (CAD)		
Related requirements	REQ1		
Initiating actor	Jenkins server		
Actor’s goal	Execute frontend, microservices and database containers		
Participating actor(s)	Test environment		
Pre-condition	Successful application builds and unit test.		
Post-condition	Deployed application		
Flow of Events for Main success scenario	<-	1	Receive delivery package
	<-	2	Pull oracle database image from local repository
	<-	3	Synchronize connection data between the various application components.
	<-	4	Build application server docker image
	<-	5	Build microservices docker image
	<-	6	Start containers in the appropriate order (docker-compose)
Flow of Events for Extensions (Alternate scenarios)	<-	3a	Ask developers to create files that are already synchronized.
	<-	4a	Pull all required images from local repository
	<-	4b	Copy artifacts into containers at run time.

Figure 2.5: Detailed UC2



UC3	Dynamic Test		
Related requirements	REQ2		
Initiating actor	Jenkins server		
Actor’s goal	Acceptance test		
Participating actor(s)	Logs drive, dynamic tests, test environment(cicd-server) and Jenkins server.		
Pre-condition	Successfully deployed application.		
Post-condition	Dynamic test results.		
Flow of Events for Main success scenario	<-	1	Start UI test
	->	2	Collect and display results
	<-	3	Start security test
	->	4	Collect and display results
	<-	5	Start transaction test
	->	6	Collect and display results
	<-	7	Restart test environment
	<-	8	Execute Load and performance test
	->	9	Collect and display results
	<-	10	Stop application and save logs.
Flow of Events for Extensions (Alternate scenarios)	.	2a	Results already on Jenkins just display it

Figure 2.6: Detailed UC3



UC4	Results		
Related requirements	REQ4		
Initiating actor	Jenkins server		
Actor's goal	Elaborate results presentation		
Participating actor(s)	Developer/Tester and dynamic tests.		
Pre-condition	After every test run		
Post-condition	One-Stop-Shop to access all test results.		
Flow of Events for Main success scenario	.	1	Present Unit test results.
	.	2	Present integration test results.
	->	3	Gather dynamic test results and present them.
	<-	4	Assign ticket to the respective Developer
Flow of Events for Extensions (Alternate scenarios)	.	3a	Just present results already on the Jenkins server.

Figure 2.7: Detailed UC4

The above process brought different aspects of the project to light. After further requirements elicitation, the following functional requirements are derived and validated. This process also helped design the workload and estimate the time required to execute the project. Four main dynamic tests are supposed to be implemented. In this thesis, the GUI test will be implemented and automated, but the other tests will be theoretically analyzed to see how they fit in the pipeline.

2.3 Functional requirements

After numerous elicitation of the above process, more specific functional requirements can be constructed, which helps in project execution and time estimation [5]. These functional requirements are a more detailed description of the requirements. All test results must be presented in the same location, making analyses easier. A one-stop shop is a central location where all the test results can be found.

#	**Functional Requirements**	**Priority**
FREQ1	The establishment of a test environment where software can automatically be deployed.	high
FREQ2	An automated process for the dynamic quality assurance of the application deployed on the established test environment.	high
FREQ3	An automated process for software integration and static software quality assurance.	high
FREQ4	The project manager(s) shall get daily notification about the status of the application. Success scenario: just a success message, and for failure scenario: a detailed report of the executed tests.	high
FREQ5	In case of specific application errors, a Jira ticket should automatically be created and allocated to the developer(s) whose commit caused the error.	low
FREQ6	A one-stop shop to access all test results	high
FREQ7	Ensure pipeline starts around 10:00 pm, hence, will be done before morning	high
FREQ8	Email pipeline status	high

Table 2.3: Functional requirements

2.3.1 Traceability matrix

After the detailed use case analysis, a more elaborate list of requirements is generated. Let us map the use cases again to the generated requirements. This ensures that we are still on track.

#	UC1	UC2	UC3	UC4
FREQ1		X		
FREQ2			X	
FREQ3	X			
FREQ4				X
FREQ5				X
FREQ6				X
FREQ7	X			
FREQ8				X

Table 2.4: Use cases and functional requirements traceability

2.4 Work packages

#	**Work packages**	**Priority**
WKP1	Jenkins pulls application and builds	high
WKP2	Jenkins execute static code Analyses on the application	high
WKP3	Jenkins transfer artifacts over to the test environment	high
WKP4	Jenkins mail pipeline status after test run	high
WKP5	Jenkins setup docker web-server tomcat image	high
WKP6	Jenkins setup docker micro-services tomcat image	high
WKP7	Jenkins synchronize communication between applications services	high
WKP8	Jenkins start docker containers and deploy application	high
WKP9	Jenkins populate database with test data	high
WKP10	Create web server (UI) test	high
WKP11	Create transaction, LPT, and security tests	low
WKP12	Jenkins gather all test results and display	high
WKP13	Jenkins shall build a delivery package of the application	high
WKP14	Jenkins create Jira ticket on pipeline failure	low

Table 2.5: Work packages

2.4.1 Present state of application

The organization where this thesis is being implemented already has a particular environment and tools. This influences the choice of specific tools and how this project will be handled. The following points depict the present state of the organization regarding this thesis.

- There is a Jenkins server that is extensively used to automate other processes.
- Selenium is already being used for another UI testing in the organization.
- The application under test is relatively new and does not have well-established tests.
- The unit, component, and integration tests are available in the application.

2.4.2 Required tools

- Jenkins
  - Warnings Next Generation plugin

  - SSH pipeline plugin
  - OWASP Dependency-Check plugin
  - Groovy, scripted pipeline language

- Maven and Nodjs
- Bash
- Docker(Podman) and docker-compose
- Selenium
- Git and Bitbucket
- Oracle SQL

3 Continuous Integration Concepts

Continuous Integration (CI) is following specific tested methods to ensure that an application's assets can build correctly and "play nicely" with the rest of the system. Developers constantly push code onto a source code repository like Bitbucket. There are some standard errors that developers are prone to make. These common errors have been analyzed over time, and some tools have been made which can be used to automate the finding of such standardized errors. These tests are carried out on an application that has not been executed, hence known as static code analyses or static tests. Some usually expected errors are introducing corrupted open-source libraries into the project and bad programming habits that create bugs in software. Therefore, software needs to be continuously built and tested when changes are made. The frequency of test execution is different for different organizations, depending on their needs. Some organizations even trigger the start of this phase on every push to the version control repository. This method requires very robust and fast test servers. Each pipeline run takes about 2 hours, which means only a maximum of 4 tests for four pushes can be done in a day. But this is usually done using distributed servers and a queue for all pushes, which can avoid such an error. In the process described here, continuous will imply daily execution because that is a more productive way of doing it in the organization where this project is being executed. About 2 hours is required for the pipeline to run to completion, which means the most meaningful interval could be every 2 hours if the frequency of push to the version control repository happens at most every 2 hours. The latter is certainly not true hence the choice to execute daily.

Different concepts and tools are required to achieve static test automation. This chapter shall elaborate on some of those automation tools and concepts. There are other platforms in the market to which the software quality assurance process can be outsourced, like testlio. These platforms are costly, and there is also the risk of exposing sensitive company data. This thesis shall focus on some specific open-source tools and concepts which can work together to achieve our goals without the exuberant cost and risk of outsourcing. There are two main goals here: automated Testing and deployment.

Before advancing to test automation, it is essential to understand that many types of static tests exist. To ensure the delivery of high-quality applications, Brian Marick came up with Figure 3.1 below, which is widely used to model various software tests [3]. These tests will be analyzed if they cover our use cases and if they can be automated. The standards established by Figure 3.1 shall be the foundation of this thesis's software QA. When the tests executed by each quadrant are positive, an organization can estimate the quality of the application under test.

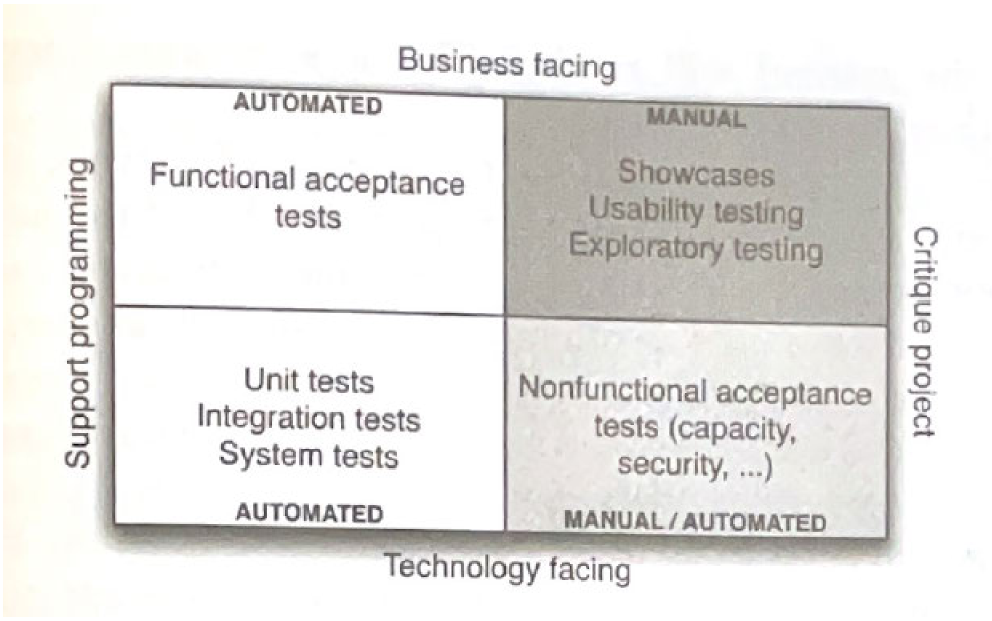


Figure 3.1: Testing quadrant diagram [3].

3.1 Technology-facing tests that support programming

These tests will be executed, and their results will directly support the development process. The processes in this section shall support the achievement of UC1, UC2, and part of UC4. The goal of UC1 is to carry out a static code analysis and obtain quantifiable results that estimate how good static software is. To know the quality of software, different aspects of the code must be tested. More focus will be on the Technology-facing tests that support programming, as in figure 3.1 above. Three types of tests fall into this category: unit test, component test (also called integration test), and System test (also known as deployment test). This thesis will refer to the deployment test as continuous application deployment. The aim is to automate this process. Therefore the concept of CI shall perfectly support some of these tests.

3.1.1 Unit test

The name unit test in some way already defines itself. It is a way of testing the smallest code unit that can be logically isolated in an application. The developers exclusively develop the unit test during the development of the application. It suffices to build the project using your application's package manager. The specific application under examination in this thesis uses both npm (Nodejs) and Maven (Java) package managers. Success in this test implies the application is built successfully, and every application unit works as defined by the developers. These tests can readily be automated. Using the package manager CLI of any project, the unit test can be automatically started via the command line.

3.1.2 Component test

It tests if the component's interface or interfaces behave according to specifications. Software components are often made up of several interacting objects [5]. This test is directed toward composite components. For example, components A, B, and C exist, and unit tests readily test each piece individually. But when these components combine in an interface to form a larger subsystem, behavior tends to change. Component tests test such changes, and this test is usually developed alongside the unit tests by the developers. The developers also design an integration test, usually packaged with the unit test. The same CLI principles of unit tests can also be used to automate these tests.

3.1.3 Deployment test

Here we test if the built application can successfully be deployed. It is possible that an application successfully passes the previous tests but still cannot be deployed due to errors in communication or connection between the database, micro-services, and the application's front-end. It is part of automated testing which means it is done on repeated cycles and, in some cases, on multiple parallel systems, hence the need for speed, standardized environment, and distributed deployment capabilities. More Information can be found under Continues application deployment section.

3.2 Static code analyses

This refers to static code analysis tools that highlight possible vulnerabilities in an application that is not being executed, using techniques such as data flow analysis, control flow graph, and lexical analysis[1]. These tests might not be directly implied in the quadrant diagram above, but they are essential and fits perfectly under Technology-facing tests that support programming.

3.2.1 Maintaining organizational standards with code audits

Having coding standards is always a brilliant idea for companies to adopt because it makes it easier to correct errors in code even when the person responsible for the error is not present. Trying to uphold these standards manually can be a cumbersome task. Hence automation tools like PMD and Spotbugs can and should be used.

PMD

It has more than 180 customizable rules ranging from braces placement in conditional statements to naming conventions, design conventions, and even unused code [6]. For example, the code below is entirely okay. PMD also regards this as OK. Still, most companies do not because mistakes can quickly occur when adding more options in the "if" statement by omitting the braces.

```
1 if(status)
2     commit();
```

A developer can commit the code snippet below, where line 3 always executes irrespective of the status. Such errors are run-time errors and are hard to find; hence PMD offers the option to customize the rules. Note that the error below can easily be found by PMD in its default settings, though it will be classified under low severity, which can be neglected. But it could altogether be avoided if correct standards are put in place.

```
1 if(status)
2     log.debug(committing cicd)
3     commit();
```

Other than maintaining the code standards, PMD also reduces another problem known as **Code Duplication** by implementing Copy/Paste Detector (CPD) [6]. Developers have always not hesitated to copy-paste code during updates rather than re-using already existing code. This problem has existed since programming came into existence. Studies estimated that the Linux kernel (as of 2002) is 15% - 25% duplicated, and the Sun Java JDK is 21% - 29% duplicated [6]. These are some old problems we have learned to live with. It does not affect the running program but has the following consequences.

[1]https://owasp.org/www-community/controls/Static_Code_Analysis

- Increases the maintenance cost due to discovering, reporting, analyzing, and fixing the same bugs multiple times.
- Uncertainty about the results of the number of bugs found (it could just be the same bug multiple times).
- Increase testing cost for the additional corrections that will be done multiple times.

Using PMD-CPD helps to locate the areas with duplicated code and produces an XML report that can be viewed using the Next Generation Plugins on Jenkins.

Spotbugs

Spotbugs, as the name implies, is a tool that finds bugs in Java applications. It spots out the security flaws of an application with a high level of precision. For most applications, the Spotbugs XML reports enable analysts or software developers to further narrow down to the exact flaw in their code [7][2]. It handles issues like buffer overflow and division by zero. Below is a function that will likely throw the division by zero error.

```
int divide(int x, int y) {
        return x/y;
}
```

The function divide() above does not handle the exception if y is zero. Spotbugs will spot out functions like this and even go as far as suggesting a possible solution.

Spotbugs is like PMD. They differ in that Spotbugs runs its tests on byte code. Meanwhile, PMD does the same on source code. Figure 3.2 below clearly illustrates the relationship between Spotbugs and PMD.

[2]https://owasp.org/www-community/controls/Static_Code_Analysis

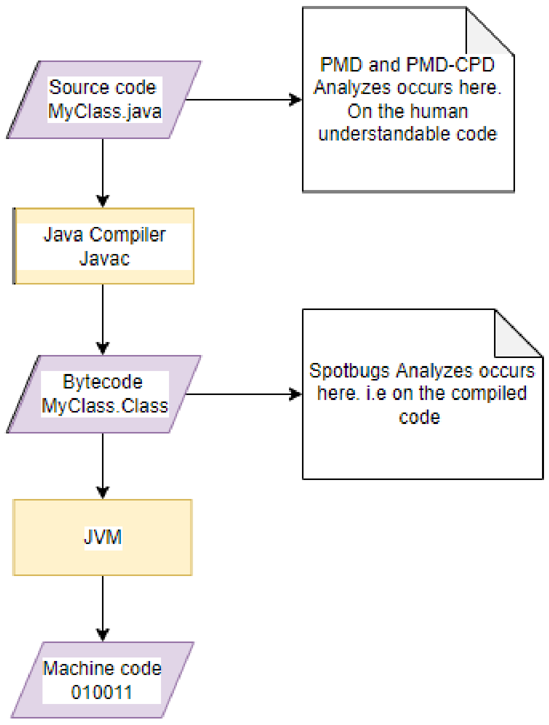

Source code
MyClass.java
PMD and PMD-CPD Analyzes occurs here. On the human understandable code
Java Compiler
Javac
Bytecode
MyClass.Class
Spotbugs Analyzes occurs here. i.e on the compiled code
JVM
Machine code
010011

Figure 3.2: Compilation steps of java code

Dependency-Check by OWASP

Software Composition Analyzes (SCA) is an automated process for finding open source software in a codebase. This is done for three main reasons: security, code compliance, and software quality [8]. *Dependency-Check* is a SCA tool offered by The Open Web Application Security Project (OWASP), that attempts to detect vulnerable libraries in an application's dependencies [9][3]. It creates a Software Bill of Material (SBOM) i.e., an inventory of all open source dependencies in an application or project. The resulting SBOM is then compared with a list of known vulnerabilities found in the National Vulnerability Database (NVD) offered by part of the U.S Department of Commerce known as National Institute of Standards and Technology (NIST). Dependency-Check then further classifies the vulnerabilities of all open-source libraries in the application. Each library's vulnerability in NVD is categorized into five domains: null, Low, moderate, high, and critical. Null implies that the dependencies have no known vulnerability. The range of the vulnerability increases from moderate to critical, with critical being the worst.

[3]https://owasp.org/www-project-dependency-check/

Furthermore, Maven can also carry out the dependency check analyses by adding the dependency-check-maven dependency in the applications pom.xml and running "mvn site" on CLI. The approach implemented here is by using the above plugin to create the SBOM, carry out the analyses, and generate an XML report. This plugin also has a publisher which nicely publishes the results file dependency-check-report.xml. By now, it should be clear that this plugin comprises two parts, the Analyzer, and the Publisher. The former cannot create SBOM for Nodejs packages. Meanwhile, the latter publish any results generated by the Analyzer. The SBOM for npm packages (package-lock.json) has to be generated using "npm clean-install." The Analyzer can analyze the package-lock.json.

This is presently one of the best solutions available when we consider that Ken Olsen, one-time CEO of Digital Equipment Corporation, referred to open-source code in 1987 as "snake oil." Just 14 years later, Steve Ballmer, former CEO of Microsoft Corporation, referred to it in 2001 as " cancer that attaches itself in an intellectual property sense to everything it touches." But today, in 2022, 97% of commercial code in the world contains open-source code [8]. This means the amount of open-source code in commercial code is steadily increasing and, therefore, the need for more visibility into open-source code components in applications. It is a must-have for most DevOps or DevSecOps teams. The Log4j incident in 2022 is an excellent example of a vulnerable library that caused panic around the software development industry. This is just an example of how much open-source software is present in the industry and how important it is for such analyses to be carried out.

3.3 Continuous application deployment

This section shall cover the continuous deployment of an application for testing purposes. Applications can also be deployed on tomcat in a VM, but it is not ideal due to some of the issues that will be covered in this section. On the other hand, Docker has features ideal for continuous application deployment. Fast application deployment can be achieved by using Docker and VMs hence. We must first understand how Docker and VMs function and complement each other. Docker is a tool for creating containers that make applications portable and self-contained. These containers are like Virtual Machines (VM) but because they are lightweight and can be booted up within seconds. One of the major differences between a VM and a docker container is the hypervisor used by VMs and the container engine used by containers. Figure 3.3 below throws more light on this difference. The hypervisor is a battled-hardened technology that has already gained trust in the industry [10]. Meanwhile, docker is relatively new and can not handle some complex tasks that the VMs can. The main aim of docker is not to replace virtual machines but rather to run single encapsulated applications [11]. Due to the benefits of both technologies, most organizations use them together, i.e., a docker engine running on a VM. This exact implementation shall be used in this thesis.

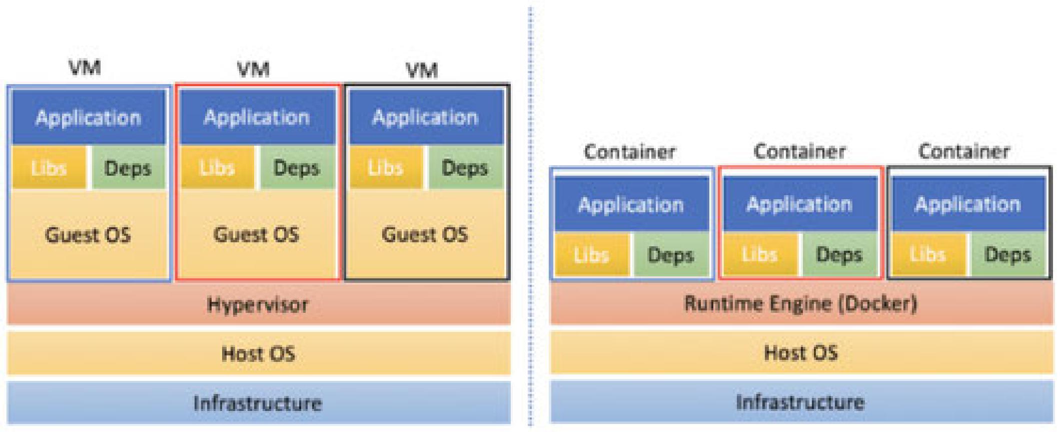


Figure 3.3: Difference between VM and Docker container
[12]

3.3.1 Docker architecture

As earlier mentioned, docker is a tool with different components which interact together to offer the fantastic service of the docker technology. Docker comprises five significant components; below is a brief description of each component and its use.

- ***Docker daemon*** is the "heart" of this tool. it is responsible for creating, running, and monitoring containers. It also builds and stores images [10].
- ***Docker client*** is used to talk to the docker daemon via HTTP. By default, it uses the Unix domain socket for communication, but it can also use the TCP socket for remote communication of the client and the docker daemon.
- ***Docker registry*** stores and distribute docker images. The default registry is the docker Hub. Organizations set up their registries for privacy, financial, and security reasons. Nonetheless, the docker Hub hosts thousands of public images and curated "official" images. The official images are owned by organizations using the paid services of docker.
- ***Docker images*** are used to build containers. It is a portable container that can be easily transferred and stored in registries and Hub.

- ***Docker container*** is the final interactive product of docker.

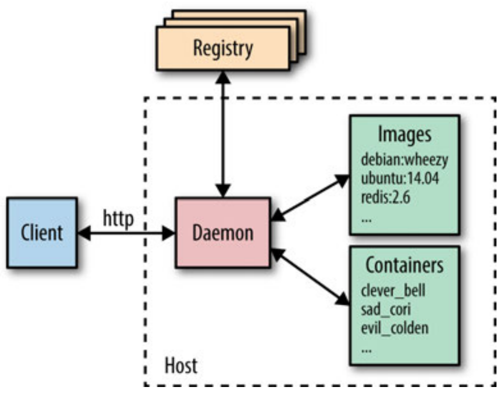


Figure 3.4: High-level overview of major Docker components
[13]

However, it is essential to note that this project is being executed on a Linux CentOS 9 Stream VM. This implies it runs podman and not docker. Podman CLI enables the execution of docker commands as if it was a docker. Both technologies are similar but for the fact that docker has a daemon and podman does not. Podman runs with the logged-in user and does not need the client daemon communication implemented by docker.

After a brief overview of docker and podman, we realize that they are lightweight, have images that are easy to transport, have fast boot-up time, and are easily re-configurable containers. Those are some primary reasons this project was done using docker/podman. As we advance in this document, the terms docker and podman will be used interchangeably. But it should always be considered that it is podman running in the background despite using docker commands.

3.3.2 Implementation of images

As will be seen in the following chapters in Figure 5.2, the database image is built and saved in the docker registry. Only the microservices and application server images will regularly be built. These tomcat images are built from Docker files. The already configured tomcat images found on the docker hub are not used here because those images are designed with the default settings and structure of tomcat. Most large organizations always customize tomcat to suit their needs. The same approach shall be used here. Hence the need for creating Docker files which will be used to create images from which containers will be built. These docker files can also be seen as documents containing all the standards for appropriate application deployment. This means if there are any changes in the application's environment, this change can be mirrored on the docker file, thereby maintaining smooth deployment. The goal is to be able to deploy a new application with all the latest delivery artifacts. As mentioned above, the database image was already built and stored in the podman registry. *Docker volumes* is a link between the VM operating system and a docker container. All the required database delivery artifacts are stored in a directory on the test environment VM; the directory is labeled and attached as a volume. This volume is then attached to the container, thereby giving the container access to resources residing on the VM. This sharing of resources is a two-way street depending on the attachment configuration, i.e., there are volumes where the container accesses resources of the VM and other volumes where the VM accesses the resources of the container. The latter is made use of for the storage of log files. The two-way communication can be ensured by using a ": z" for Podman-docker, see appendix 2 docker-compose.yml.

3.3.3 Docker compose

Docker-compose is designed to start up docker development environments [10] quickly. It is written in either YML or YAML files. Our application has different components to be automatically started in a specific order. Docker-compose is designed mainly for this purpose. It starts all the containers under the same docker network and allocates the defined volumes. It also provides the *HEALTHCHECK* option, which waits until a specific service is up and running before moving on to start the next as defined in the docker-compose file. This helps to start the different components or services in the desired order. Otherwise, docker-compose starts all services sequentially and does not wait for a service to be "up" before starting the next. This leads to failures due to no synchronization.

The *ports* exposed by docker containers are limited to the internal network created by docker-compose. These ports must be mapped to an external port on the VM on which the docker daemon resides. The mapped ports must not be identical to the containers. The deployed server in the container can be accessed from outside using the mapped port.

All the images and containers are built when the "docker-compose up" command is executed. On the execution of „docker-compose down, "all containers are killed, but their images and volumes keep existing. If the volumes are not deleted before starting the containers again, then the previous instance of the application will be launched. Hence test will be done on an application with the earlier artifacts. For this reason, all volumes and containers should be dropped before restarting docker-compose. By doing this, new and fresh content will be started.

Docker compose eases the overhead of creating a volume. Just define it in the docker-compose.yml file, and docker-compose will automatically create the volumes.

3.3.4 Expected outcomes

After successfully executing all the tests listed in this chapter, the following outcomes are expected.

- Unit Test surefire report
- Static Analyzes Warnings Report
  - Spotbugs report
  - PMD report
- Dependency-Check report
- Artifacts
  - Applications delivery package
  - Pipeline automation scripts from Bitbucket
- Deployed application
- Location of centOS9 container volumes: /var/lib/containers/storage/volumes

4 Continuous Delivery Concepts

Continuous delivery is more of a methodology or a way of working whereby software quality can be assessed at multiple intervals [3]. This allows quality software to be delivered much sooner than traditional methods. This should not be confused with DevOps, though similar DevOps is a way of working whereby the development team and IT operations team work collaboratively and harmoniously towards archiving a common goal. The pipeline to be implemented in this thesis shall achieve both the goals of continuous delivery and DevOps. CD is the phase after CI, i.e., after the application has been successfully deployed. A series of acceptance tests are then executed against the deployed application. The acceptance test shall verify if the application meets up the acceptance criteria of the requirements. This chapter focuses on dynamic software tests.

4.1 Acceptance test

Going back to the Brian Marick quadrant diagram in figure 3.1 above, we realize that the functional acceptance test should also be carried out to ensure quality software delivery. The quadrants which are directly concerned with the acceptance test will be investigated in this section.

4.1.1 Business-facing test that supports programming

GUI Tests

In this test, the customer tests the deployed application and confirms if the criteria for functional acceptance tests have been met. This is testing the functional requirements or user stories of a system. The standards of the Graphical User Interface (GUI) test will be tested with simulated customers and users. Many tools offer the capability to carry out this task, like Playwright, Gatling, and Selenium. Selenium will be used in this project because it is already widely used in the organization where this thesis is carried out. Besides, selenium offers the selenium grid infrastructure, which significantly fits test automation pipelines. Selenium is an open-source tool that also helps when looking at costs. An example of a GUI test of a function requirement could be testing if a user can log in to their account and access their personal information. Selenium does this by simulating the user and carrying out the tasks.

The downside of this test is that it often fails because of the change of name of a checkbox and not because of the functionality under test. This reason is enough for some companies to either omit the UI testing or replace it with a test bypassing the UI, basically API testing.

Regression or transaction test)

It is also essential to test some functional requirements that can not directly be tested on the GUI. This test evaluates the overall behavior of the application. Some events on the GUI call some background processes, and nothing on the GUI confirms the successful execution of those events. It is doubtful that a GUI test with Selenium will be able to find errors like this. This test should be implemented to ensure the quality of an application. Tools like Cucumber, JBehave, Concordion, and Twist can be used. All these tools have subscription fees. Gatling, an open-source tool, can be used to create these tests, although the implementation will not be as trivial as the former tools. Gatling is already widely used in the organization for other purposes and shall be a good choice for this thesis because it is open source, which implies reduced costs. The transactions between the different application components like micro-services, database, and application server can also thoroughly be tested.

Part of these tests still needs to be done manually. These automated tests give a sense of the status of the application, but some functional requirements can be complicated to automate. Some of them might even not be worth implementing due to how unstable it will be, i.e., more time will be spent maintaining the test than ripping benefits from the automation.

4.1.2 Technology-facing tests that critique the project

So far in the acceptance test, most organizations focus only on the functional test and forget the non-functional tests. The non-functional test is critical, especially when it comes to systems that deal with sensitive data. In this modern society, where data is one of the most valued commodities, non-functional testing is critical to the project and is directly concerned with the technology used during development. Here we have tests like the Performance test and Security test. This is a must-have test for mission-critical systems.

Security test

In the modern software development industry, as well as the organization where this thesis is carried out, it is essential to know how secure or resilient to attack an application is. This test is carried out to determine how secure or vulnerable the running application is. Initially, a vulnerability test has already been carried out on the static code; this is not enough because it only analyses known vulnerabilities from known external libraries. But organizations also build in-house libraries and implement different communication protocols, which are not safe. The vulnerabilities introduced by such constructs can usually only be determined via well-implemented security tests using tools like Radamsa or OWASP ZAP. The best practice is to save all URLs used by the selenium test; these URLs are then passed to OWASP ZAP or Radamsa. The security tool then uses the URLs to conduct a fuzzing test against the deployed server. These tools are all open source and can be implemented in an automation pipeline. This test will only give an estimate of the security vulnerabilities.

Load and performance test (LPT)

Here, the resilience of the software is determined. The responsiveness of the UI can be affected negatively without causing any error that any of the other tests can determine. Selenium can measure the time between clicking on the UI and getting a response; this shows how fast the UI is. An example of a scenario that can be checked here is a call that can go into the database and get a specific name from a list of names, but it is made to call the whole list and then deliver the required name. This error can drastically increase the response time of the UI.

The breaking point of the application server is checked here. One of the most common hacks is to bombard a server until it cannot handle requests. But suppose the breaking point of a server is known. In that case, measures like load balancing can address the problematic situations of overloaded requests to the application server without breaking it.

This test also analyses how responsive the database is. Oracle databases keep records of their performance on the so-called AWR report; this report also contains suggestions for improving bad requests.

4.1.3 Business-facing tests that critique the project

This manual testing involves all project stakeholders to verify that the application meets its specifications. It is tricky to completely automate this process because the team is hoping to find the unexpected behavior of the software. The automation of this process is mainly done by large corporations that carry out minimal and specific incremental changes. When a change is limited to a particular area in an application, this area can be carved out, and speculations can be made on possible malfunctions. Test cases for hypothetical scenarios can then be developed. Meanwhile, corporations that only develop software for other corporations or make multiple changes on the whole application in one commit are limited to how far they can automate this process. Hence most of them settle for manual testing with the organization that offers services with the developed application. The results obtained from the pipeline developed in this thesis shall play a significant role during such evaluations. Therefore, the channel ends at deploying the application in the testing environment and running a series of tests against the application, but not automatic deployment into a production environment.

5 Implementation

5.1 System architecture

The system comprises parts executed at different time slots and in a specific order. Jenkins is used to controlling the entire process. The execution or flow of the pipeline follows a particular sequence. Initially, the software is pulled from Bitbucket and built, followed by the static code analyses, then dynamic code analyses, and finally the collection and presentation of results as in Figure 5.1 below. The test VMs on the architecture can be multiple VMs on which some of the dynamic tests will be executed.

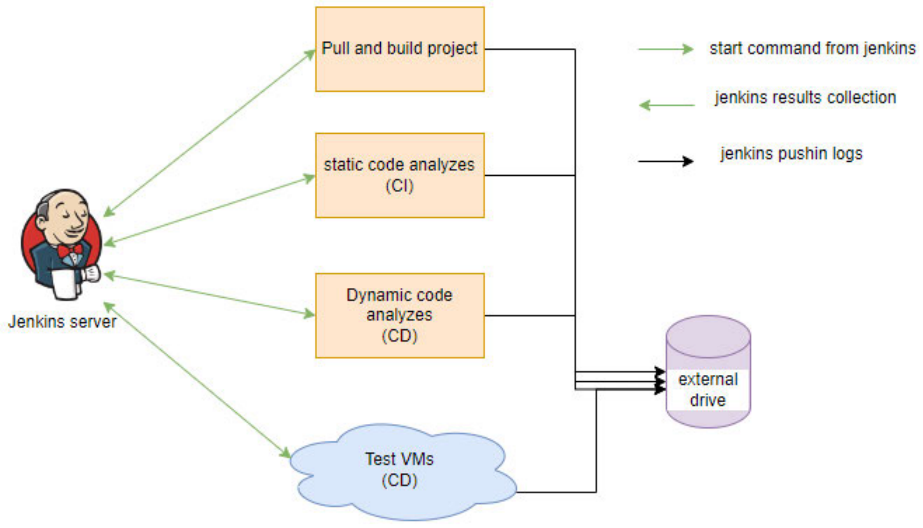

Pull and build project
start command from jenkins
jenkins results collection
static code analyzes (CI)
jenkins pushin logs
Jenkins server
Dynamic code analyzes (CD)
external drive
Test VMs (CD)

Figure 5.1: General system architecture.

The system is long and complex; therefore, we shall break the procedure into two parts: CI and CD. This will ease the implementation process and make error tracing easier. The entire pipeline is controlled by Jenkins, which executes a Groovy file, in this case, a Jenkinsfile.

5.2 Implementation of continuous integration

This is a continuous step-by-step process. Figure 5.2 below shows how the flow of events shall occur in this stage. This section shall satisfy UC1, UC2, and part of UC4. The initial plan was to collect all the results after execution, but it's more accurate and safer to collect the results after the execution of each test. This section shall be executed on both the test environment and the Jenkins server. The test environment is mainly for deployment with docker. All the other tests are executed on the Jenkins server, controlled by the Groovy Jenkinsfile. The Warnings New Generation plugin on Jenkins is used to trace the pipeline logs and report the information on the logs using different publishers.

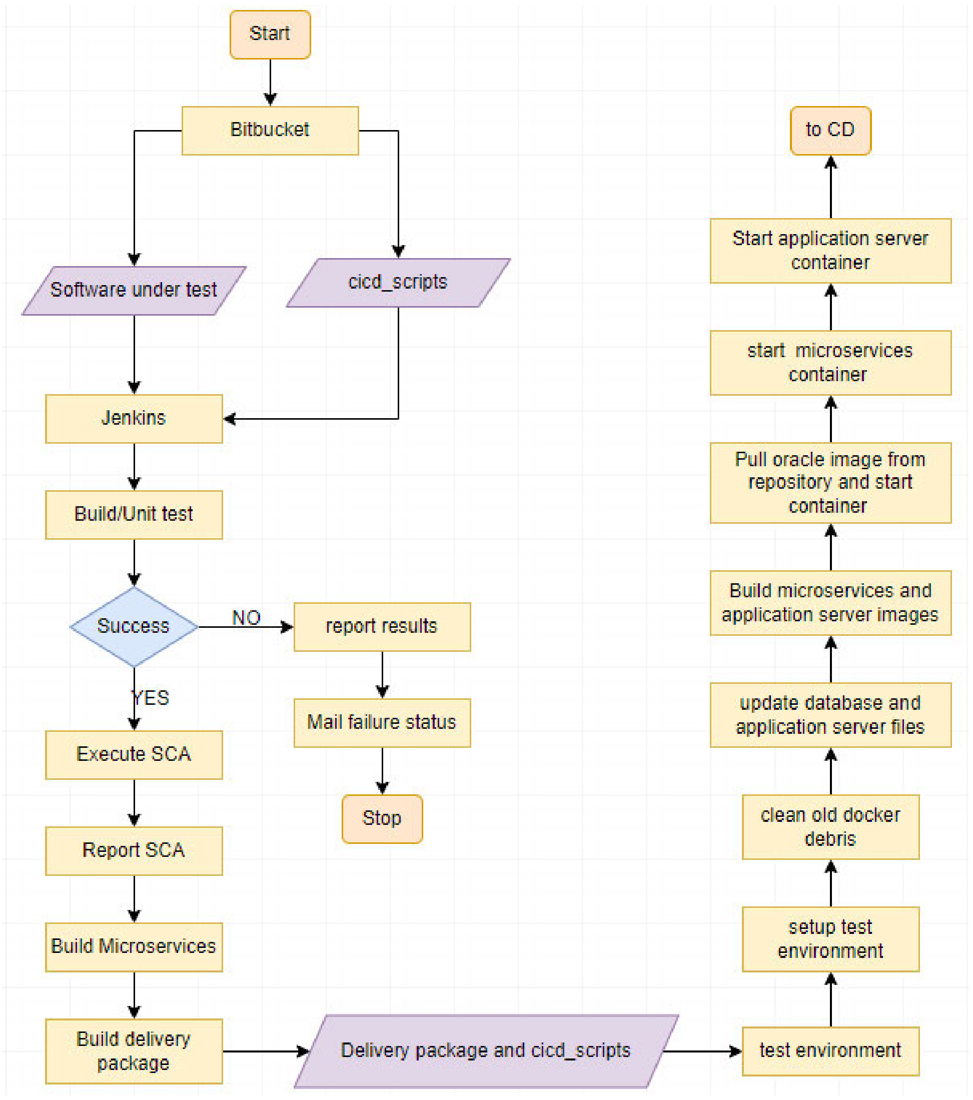


Figure 5.2: Continuous Integration flow chat

5.2.1 Build and unit test

After Jenkins has cloned the cicd_scripts (automation scripts and test projects) and the application under test, it builds and runs the unit test, including the component and integration tests. The developers exclusively develop the unit test, which is already part of the application under test. Maven is used to build the project and run the unit test. Surefire is a Maven plugin that is used to generate HTML test reports.

5.2.2 Static code analyses SCA

Furthermore, PMD results are generated, followed by SBOM for Java and SBOM for Nodejs packages. All these processes generate reports that are analyzed for errors, and the statistics are published in a "nice" user interactive manner on Jenkins. Below is a snippet of how the results are scanned and published. By default, the pipeline will be marked as a failure if there is a single new critical vulnerable library. The option "failedTotalCritical" on line 11 of the snippet is used to increase this threshold, and the same can be done with all the other vulnerability levels.

```
 1 script{
 2     def pmd = scan for issues tool: pmdParser(pattern: ’**/pmd.xml’)
 3     publishIssues issues: [pmd]
 4     def spotbugs = scanForIssues tool: spotBugs(pattern: ’**/spotbugsXml.xml
           ’)
 5     publishIssues issues: [spotbugs]
 6     publishIssues name: ’All Issues’, issues: [pmd, spotbugs]
 7 }
 8 
 9 //Dependency-Check Analyses (generates java SBOM by default)
10 dependencyCheck additionalArguments: ’--disableYarnAudit’, odcInstallation:
       ’<OWASP-version>’
11 dependencyCheckPublisher pattern: ’**/dependency-check-report.xml’,
      failedTotalCritical: 364
```

Listing 5.1: CI results publishers

5.2.3 Delivery package

After building all the other sections of the application, the generated artifacts need to be collected. These artifacts are then collected and stored under a single versioned folder. These application artifacts and automation scripts must be transferred to the test environment. This is an automation Groovy script that should be able to run on different platforms, so extracting the artifacts from a versioned folder is not trivial as on java or Bash. The snippet below is a function that can pick out a versioned folder on both Unix and Windows systems.

```
def getCommandOutput(cmd) {
    if (isUnix()){
         return sh(returnStdout:true , script: ’#!/bin/sh -e\n’ + cmd).trim
             ()
    } else {
       stdout = bat(returnStdout:true , script: cmd).trim()
```

```
        result = stdout.readLines().drop(1).join(" ")
        return result
    }
}
```

Listing 5.2: Groovy find versioned folder

Transfer of the delivery package and automation scripts is done using the SSH Pipeline Steps plugin. For information about the implementation, check out the Jenkinsfile in Appendix 1.

5.2.4 Deployment

The developers design the application to be deployed for manual deployment. This means the communication security between the micro-services, database, and application server is expected to be defined during setup. The software package comes expecting to use some predefined environment variables that suit the developer's environment. These environment variables are then defined in the Dockerfile, which eases the automation process. Standardized values like IPs, passwords, ports, and database SID shall be created to work well with the test environment. All files in the application responsible for connectivity are synchronized with the standardized values of the test environment. This eases the functioning of the application in the test environment. The file structure is also rearranged to suit docker norms. Deployment takes place in 3 significant steps: database, micro-services, and application server. We shall go through all of them chronologically as listed. setup.sh script, found in appendix 3, shall control this process.

The application under test uses Oracle XE. Hence we shall build an oracle XE image. Building the database image can take a very long time, and we do not change anything in the image at run time, so building the image on every pipeline run does not add any value to our system. The docker oracle image can be built with the aid of an online GitHub project[14][1]. This project comes with some default scripts to build the image, but some changes need to be made to make it run in our test environment. The first change is in the runOracle.sh script, where docker executes all scripts found in the mounted docker volume, but we want to run just some specific SQL scripts. This script is updated such that just some particular files are executed after the database starts. Secondly, a basic script by the name populateDB.sh is added, which runs an SQL script mounted on the oracle docker container. Below is the script's content.

```
#!/bin/bash

```

[1]https://github.com/oracle/docker-images/tree/main/OracleDatabase/SingleInstance

```
SCRIPTS_ROOT="/opt/oracle/scripts/startup";
if [ -d "$SCRIPTS_ROOT" ] && [ -n "$(ls -A $SCRIPTS_ROOT)" ]; then
  cd $SCRIPTS_ROOT
  echo ";
  su -p oracle -c "$ORACLE_HOME/bin/sqlplus / as sysdba @user_pass_extend.
      sql"; echo ;
  echo "DONE: Executing update script(s)";
  exit 0;
else
  echo "Problems updating DB!"
  exit 1;
fi;
```

Listing 5.3: populateDB.sh script in database container

After all these changes have been made, the image is then built and saved on the Podman-docker registry, which can be accessed from any machine connected to the same network. This must be done before starting the pipeline. The database artifacts are copied into the directory mounted as a docker volume to the database at the deployment stage. These artifacts are used during database start-up to set up the oracle XE database container.

The microservices and application images are created using the copied artifacts by executing the Dockerfiles. The microservices container is started, followed by the application server container.

These containers need to start in a specific order; docker-compose is used to start all the different services one after the other. HEALTHCHECK is used to wait until service is already working before moving to the next. There is a dummy alpine service at the end of the docker-compose file. This service allows docker-compose to wait until the application's UI is up and running. More details about docker-compose.yml are found in the appendix.

For any meaningful test to be done on the deployed application, we need some data in the database. The SQL script user_pass_extend.sql is used to execute SQL commands in the oracle container with the help of the populateDB.sh script. This SQL script can run other SQL scripts and can be extended as much as possible without changing the architecture. This makes updating the pipeline very easy.

5.3 Implementation of acceptance test

This section assumes that the application has been successfully deployed. According to this project's scope, the focus will be on the UI test. However, the pipeline is constructed with the other tests in mind.

5.3.1 Selenium GUI test implementation

The deployed application is now up and running, but we are unsure if areas like the menu elements are working correctly. Selenium is an open-source tool for UI testing; it can simulate real-time user browsing through web pages. It offers a Hub and client architecture as in figure 5.3[2] below that will be fully exploited for this test.

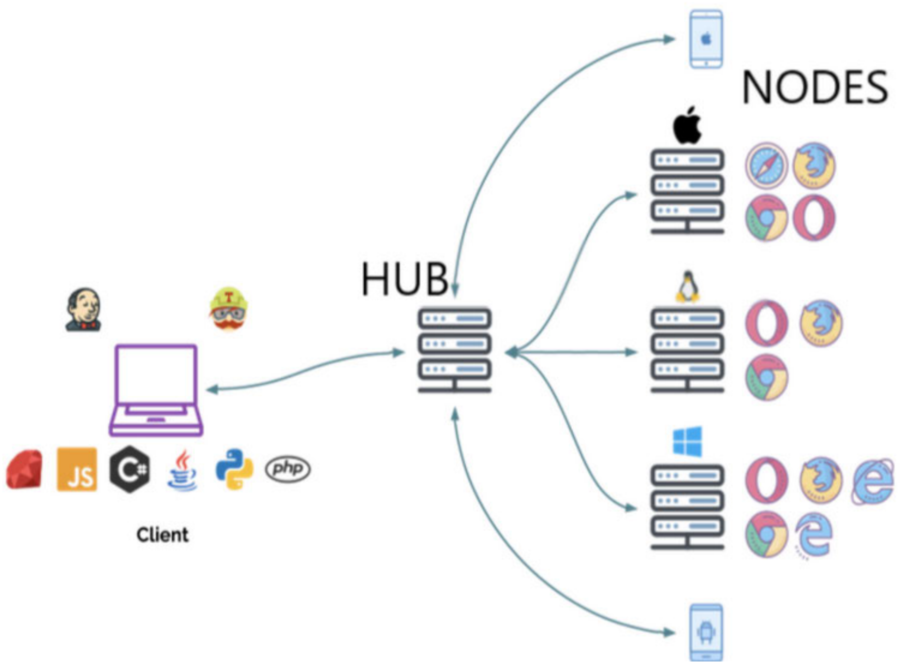


Figure 5.3: Selenium grid architecture
[15]

The selenium java code runs on the Jenkins server but connects itself as a client to a dedicated selenium Hub server. A node is then started on the Hub, which executes the UI test, and the final test report is sent back to the Jenkins server where the actual java code is located. Before starting the test, configure the Hub as on the Selenium online page [15][3]. Line 2 below starts an Edge browser node on the Hub, line 3 connects as a client to the selenium Hub via the Hubs URL, and line 7 can then call the application's web page under test.

```
1 DesiredCapabilities capability = new DesiredCapabilities();
2 capability.setBrowserName("MicrosoftEdge");
3 capability.setPlatform(Platform.WINDOWS);
```

[2]https://www.browserstack.com/guide/selenium-grid-tutorial
[3]https://www.browserstack.com/guide/selenium-grid-tutorial

```java
driver = new RemoteWebDriver(new URL(hubUrl), capability);
js = (JavascriptExecutor) driver;
driver.manage().window().maximize();
driver.get("<web page url of application under test>");
```

Listing 5.4: Selenium Hub client setup

The surefire plugin is added in the project's pom.xml, which is used to generate the test results and to choose the desired TestNG XML to execute. TestNG is a plugin to group test cases and control the execution by creating an XML file that Maven can use on a CLI. TestNG uses, amongst others, @Test before function declaration in java to compartmentalize different test cases. These TestNG XML files can automatically be generated on an IDE like IntelliJ after declaring the appropriate "@<commands>."

Finally, Surefire generates the test results in the target/surefire-reports directory. The results package contains detailed results in XML, HTML, and txt formats.

5.4 Alternative approach

The architecture can be slightly changed by introducing a Red Hat automation tool called Ansible. It is used to orchestrate multiple Linux VMs. This approach will not bring any fundamental change to the pipeline. But it shall provide the opportunity for parallel application deployment as illustrated in figure 5.4. The two major differences are: All images will be pushed to a central registry after creation, and Ansible will then use the images to perform the parallel deployment. This approach outsources load from the Jenkins server to the Ansible server. Ansible executes a YAML playbook which contains all the configuration commands [4].

[4] https://docs.ansible.com/ansible/latest/user_guide/playbooks.html

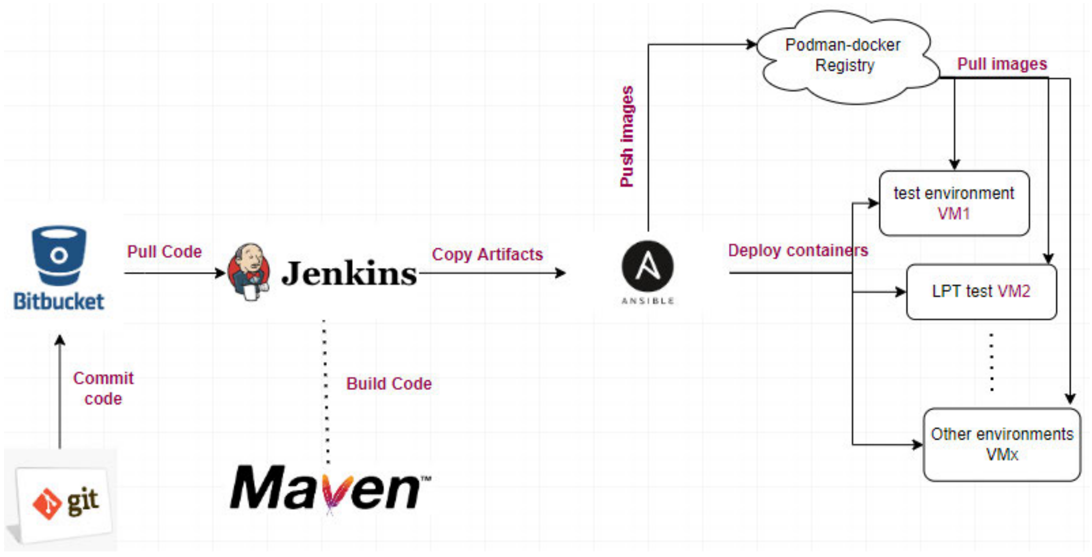


Figure 5.4: Alternate system architecture

6 Pipeline test results

A package made up of SQL, bash, groovy and yml scripts are the software outcome of this thesis. These scripts successfully execute the entire project (CICD pipeline). The tree or file structure in figure 6.1 is known as the automation package.

```
cicd_scripts
├─ database
│  └─ user_pass_extend.sql
├─ microservices
│  ├─ Dockerfile
│  ├─ start_microservices
│  └─ dos2unixScripts.sh
├─ webserver
│  └─ Dockerfile
├─ Jenkinsfile
├─ setup.sh
└─ docker-compose.yml
```

Figure 6.1: Thesis Delivery Package

Each script has a specific function. The setup.sh script sets up the test environment by rearranging the folder structure to suit the docker standards as mentioned in the former chapters. It also updates the configurations in the database and tomcat files to synchronize information and ensure connectivity. The respective docker images on the above structure are built using the Dockerfiles. The database docker image has been built and is readily available in the docker registry. docker-compose.yml, which is triggered by the setup.sh, starts all the containers in chronological order as expected, i.e., database, micro-services, and finally, the application server. After a successful start-up, we have a virgin application with no data in the database. The SQL script contains some data that will be filled in the database to ensure a more accurate UI testing.

Different sets of results are obtained at the end of this pipeline. Poor results in all the other tests will allow the pipeline to run till the end, except for the unit test. Build failures and unit test errors will break the pipeline and report the results.

6.1 Static code analyses results

6.1.1 Technology-facing tests that support programming results

There are individual and aggregated test results. As previously explained, the unit test was a success if we had the results of any other test. At the end of the CI phase, it is more convenient to have an overview of the results before going into detail to analyze every individual result. Each execution of the pipeline is called a build, and some publishers are used to publish the results of each build. The Dependency-Check publisher tracks every build and establishes a trend of the builds. The current build is compared with the previous build, as shown in Figure 6.2 below. With just a glimpse at the graph, it is easy to tell if the quality of the dependencies has improved, deteriorated, or remained constant. The same build trend can also be seen in the aggregated PMD, Spotbugs, and Unit test results. The aggregation and trend analyses are done by the Warnings Next Generation plugin, which is similar to that of the Dependency-Check publisher. Figure 6.5 below is an example of such a graph.

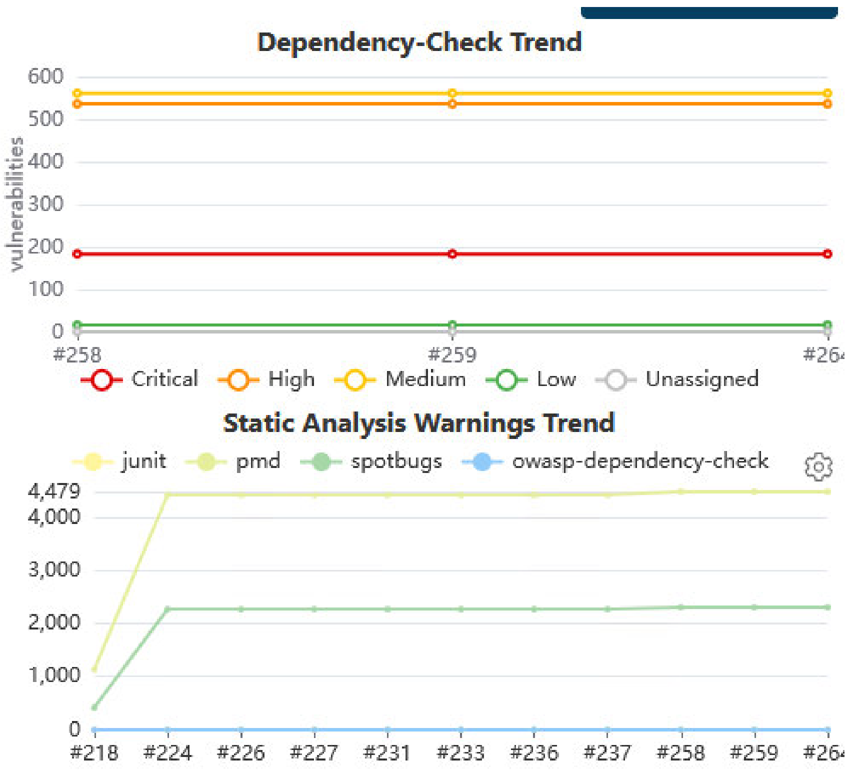
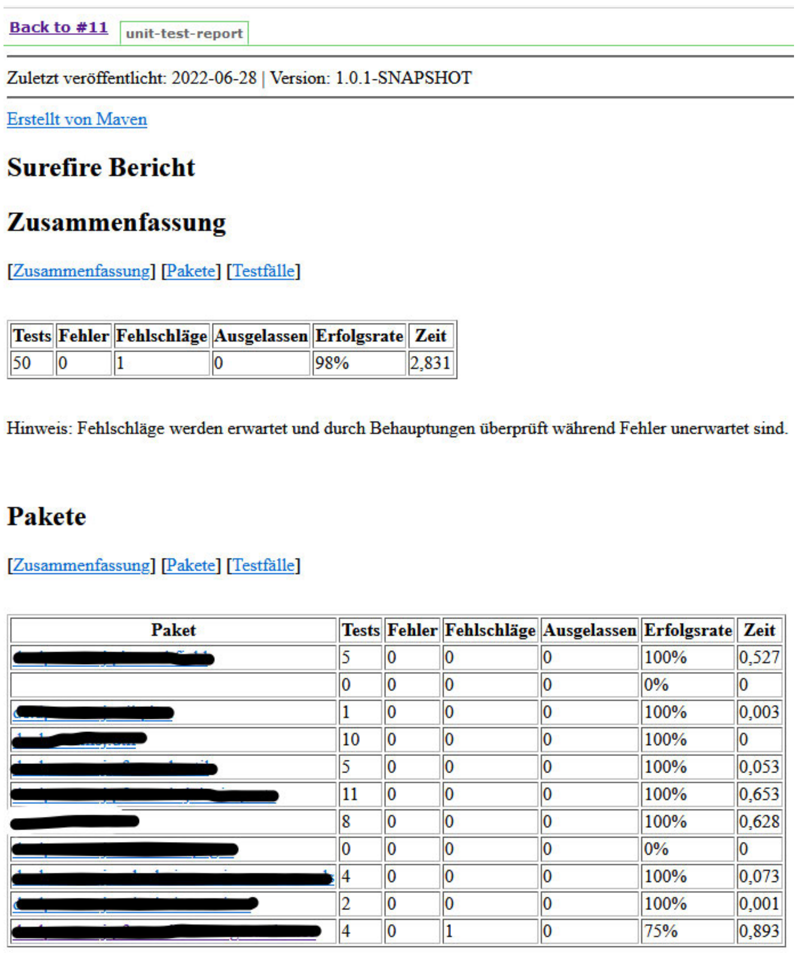


Figure 6.2: Overview of Pipeline Trend

Unit and component test results

It produces an HTML surefire report. This report shows, among others percentage of passed and failed tests. The area where the error occurred and the actual error message can be found on the report. Below is an example of the HTML report. Figure 6.3 below shows part of the results of a failed test. This is just part of the report, the actual report is very long, and because of that, just an extract of the report is shown here. The surefire-reporting maven plugin can be configured on how it reports the tests; in this project, it is configured not to create a detailed report for a successful test. Only the first table on Figure 6.3 containing all zeros will be outputted for a successful test.



Back to #11 | unit-test-report

Zuletzt veröffentlicht: 2022-06-28 | Version: 1.0.1-SNAPSHOT

Erstellt von Maven

Surefire Bericht

Zusammenfassung

[Zusammenfassung] [Pakete] [Testfälle]

Tests	Fehler	Fehlschläge	Ausgelassen	Erfolgsrate	Zeit
50	0	1	0	98%	2,831

Hinweis: Fehlschläge werden erwartet und durch Behauptungen überprüft während Fehler unerwartet sind.

Pakete

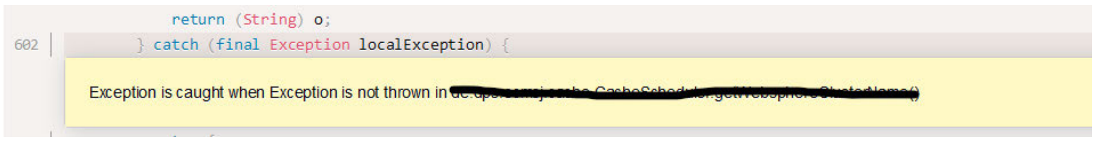
[Zusammenfassung] [Pakete] [Testfälle]

Paket	Tests	Fehler	Fehlschläge	Ausgelassen	Erfolgsrate	Zeit
[redacted]	5	0	0	0	100%	0,527
	0	0	0	0	0%	0
[redacted]	1	0	0	0	100%	0,003
[redacted]	10	0	0	0	100%	0
[redacted]	5	0	0	0	100%	0,053
[redacted]	11	0	0	0	100%	0,653
[redacted]	8	0	0	0	100%	0,628
[redacted]	0	0	0	0	0%	0
[redacted]	4	0	0	0	100%	0,073
[redacted]	2	0	0	0	100%	0,001
[redacted]	4	0	1	0	75%	0,893

Figure 6.3: Unit test report

Spotbugs and PMD results

The results of the tools Spotbugs and PMD can be aggregated and reported together. This only makes sense because they carry out similar activities. These tools produce an extensive and elaborate set of results. Figure 6.5 below is an example result gotten after the test. Spotbugs and PMD are the tools used in this project to scan for bugs in the application. Their outcome can be seen by clicking through an extensive report aggregated from their XML reports. All details about bugs and possible bug-prone areas and packages of the application are shown in this report. The exact problematic line of code and hints about the issue can also be found in the report as in figure 6.4 below.

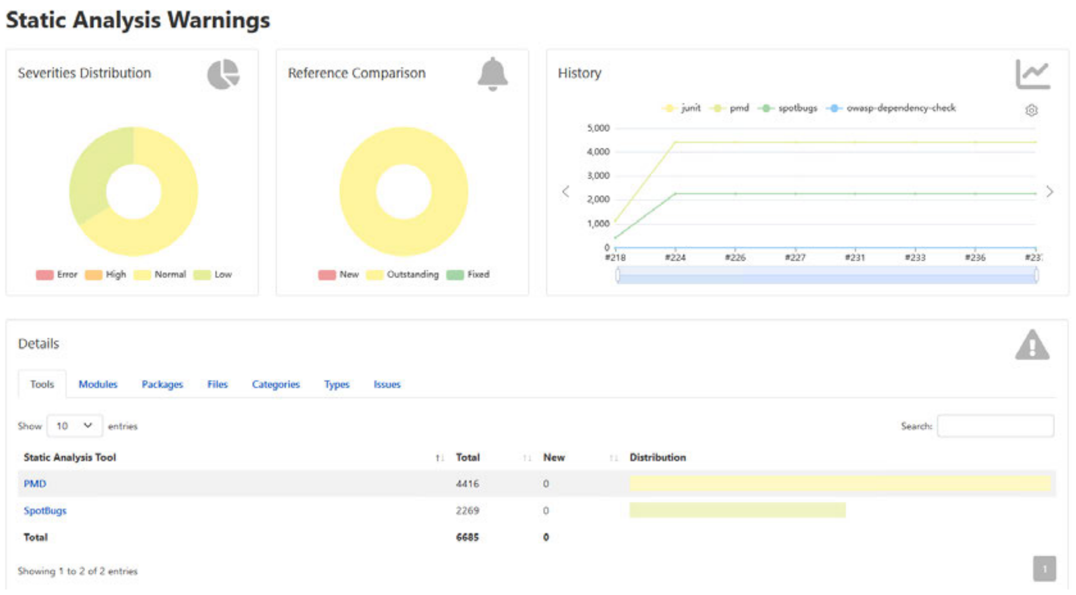


```
            return (String) o;
        } catch (final Exception localException) {
```

Exception is caught when Exception is not thrown in [redacted]

Figure 6.4: Spotted PMD and Spotbugs High Severity Bug

The bugs are categorized into critical, high, normal, and low. The trend of the bugs is also tracked, and in case of an increase or decrease in the number of bugs, it will clearly show itself on the trend at the top right of the report. The exact package and location of the code that changed can also be seen in this report. Below is an overview of the content of the report.

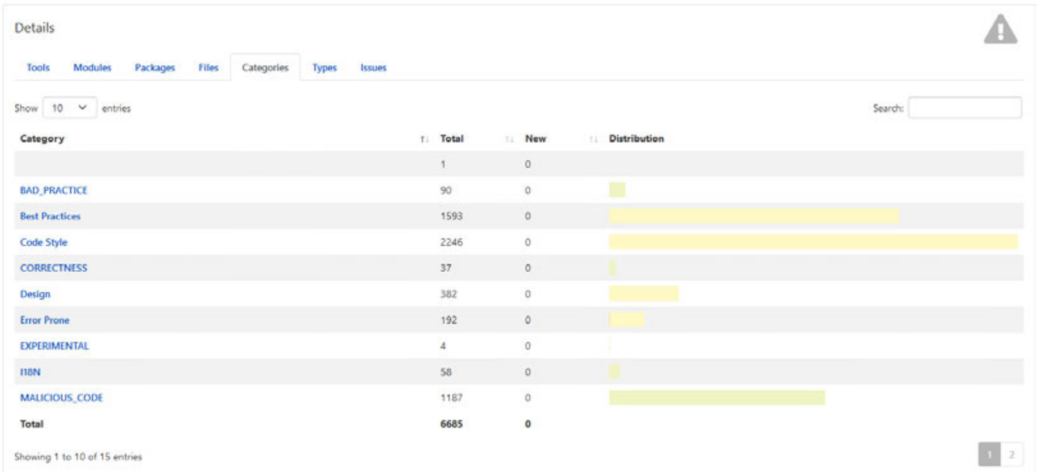


Static Analysis Tool	Total	New	Distribution
PMD	4416	0	
SpotBugs	2269	0	
Total	6685	0	

Showing 1 to 2 of 2 entries

Figure 6.5: Static analyses warnings

The image below shows how the different bugs of the application are categorized. There are also different menu points with different results groups. This allows the developer to choose exactly what type of bug to start fixing. The reported trend also helps to evaluate progress when bugs are fixed.


Details

Tools | Modules | Packages | Files | Categories | Types | Issues

Show 10 entries | Search:

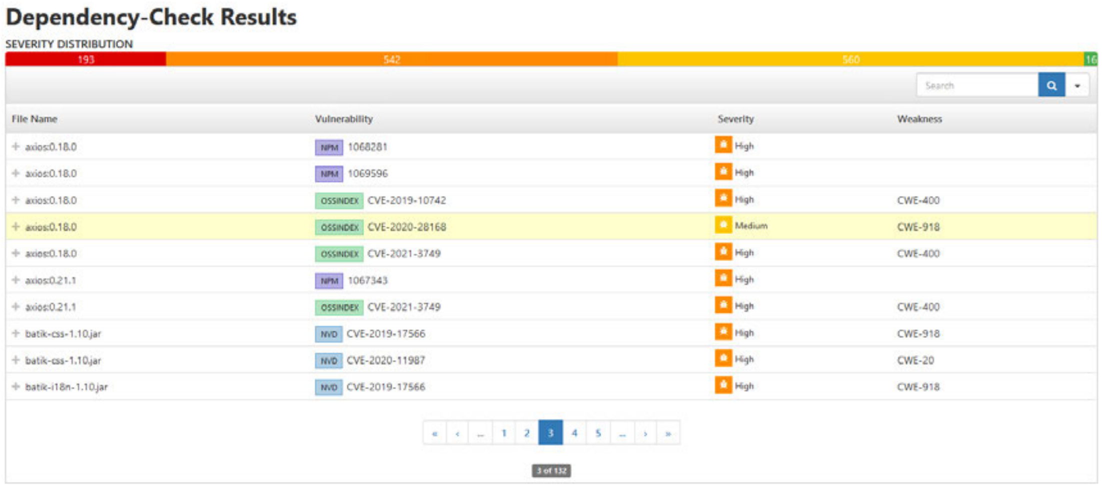
Category	Total	New	Distribution
	1	0	
BAD_PRACTICE	90	0	
Best Practices	1593	0	
Code Style	2246	0	
CORRECTNESS	37	0	
Design	382	0	
Error Prone	192	0	
EXPERIMENTAL	4	0	
I18N	58	0	
MALICIOUS_CODE	1187	0	
Total	**6685**	**0**	

Showing 1 to 10 of 15 entries

Figure 6.6: Bug results categories

Dependency check

The OWASP dependency check tool produces an XML report. This report contains all the applications' libraries, dependencies, or packages and their health results. It displays how weak, vulnerable, and severe the application packages are. This tool also tracks all the packages and makes a record of packages that change their state. The trend of the dependency check can be seen aggregated in Figure 6.2 above. Figure 6.7 below shows an example of the dependency check results. A very interactive report with all the necessary information about the dependencies in the application can be found in this report.


Dependency-Check Results

SEVERITY DISTRIBUTION

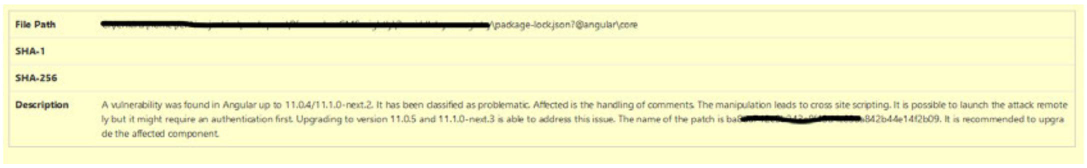
193 | 542 | 560 | 16

File Name	Vulnerability	Severity	Weakness
axios:0.18.0	NPM 1068281	High	
axios:0.18.0	NPM 1069596	High	
axios:0.18.0	OSSINDEX CVE-2019-10742	High	CWE-400
axios:0.18.0	OSSINDEX CVE-2020-28168	Medium	CWE-918
axios:0.18.0	OSSINDEX CVE-2021-3749	High	CWE-400
axios:0.21.1	NPM 1067343	High	
axios:0.21.1	OSSINDEX CVE-2021-3749	High	CWE-400
batik-css-1.10.jar	NVD CVE-2019-17566	High	CWE-918
batik-css-1.10.jar	NVD CVE-2020-11987	High	CWE-20
batik-i18n-1.10.jar	NVD CVE-2019-17566	High	CWE-918

3 of 132

Figure 6.7: Dependency check

The report goes as far as giving suggestions on how to fix the vulnerabilities as in figure 6.8 below. This publisher fails the build if either new vulnerabilities or vulnerabilities higher than the threshold values are detected.



File Path	[redacted]\package-lock.json?@angular\core
SHA-1	
SHA-256	
Description	A vulnerability was found in Angular up to 11.0.4/11.1.0-next.2. It has been classified as problematic. Affected is the handling of comments. The manipulation leads to cross site scripting. It is possible to launch the attack remotely but it might require an authentication first. Upgrading to version 11.0.5 and 11.1.0-next.3 is able to address this issue. The name of the patch is ba[redacted]842b44e14f2b09. It is recommended to upgrade the affected component.

Figure 6.8: Dependency-Check report suggestion

6.2 Acceptance test results

According to the scope of this thesis, these results are limited to the UI test.

6.2.1 UI test results

The Selenium UI test produces an HTML surefire report similar to the unit test report. It shows the percentage of passed, failed, and skipped tests. It also shows the source of the error. A successful test shows that all the menu points of the deployed application can be interacted with. The error logs are also found on the report in case of test failures. Below is an example test results output.

6.3 Functional requirements validation

#	**Functional Requirements**	**Priority**	**Implemented**
FREQ1	The establishment of a test environment where software can automatically be deployed.	high	✓
FREQ2	An automated process for the dynamic quality assurance of the application deployed on the established test environment.	high	✓
FREQ3	An automated process for software integration and static software quality assurance.	high	✓
FREQ4	The project manager(s) shall get daily notification about the status of the application. Success scenario: just a success message, and for failure scenario: a detailed report of the executed tests.	high	✓
FREQ5	In case of specific application errors, a Jira ticket should automatically be created and allocated to the developer(s) whose commit caused the error.	low	X
FREQ6	A one-stop shop to access all test results	high	✓
FREQ7	Ensure pipeline starts around 10:00 pm, hence, will be done before morning	high	✓
FREQ8	Email pipeline status	high	✓

Table 6.1: Validation of functional requirements

7 Results and conclusion

This thesis aimed at creating a test environment where an application can automatically be deployed. It is achieved by using Podman-docker and Linux CentOS VM. This process will otherwise require a considerable amount of time if manually done.

Furthermore, an application is automatically deployed in this test environment. Any dynamic software test can then be executed on the deployed application. This dramatically reduces the tester's work, thereby accelerating the development process. Automating such processes reduces the repetitive nature of quality assurance in continuous software engineering. The possibility of frequent testing implies testing on small incremental changes. Errors found in these changes can easily be handled compared to a situation with an accumulation of errors. The software release period will therefore be less tense.

With just a glimpse at trends created by the pipeline, a reasonable estimate of the daily quality of the static code of an application can be known. Code pushed to Bitbucket by all developers is automatically integrated, and its quality is determined. Over an extended period, it is easy to see if the team is generally improving or decreasing the software's quality.

Looking back at figure 1.1, facilitating the interactions between the Development and IT Operations (DevOps) teams is one of the significant outcomes of this thesis. In other words, the CICD pipeline is more of a business process aiming to increase the productivity of the Development and IT operations teams in an organization that implements continuous software engineering. The IT operations team can quickly know the application's state by looking at the results of the pipeline. Meanwhile, the Development team can get quick feedback on issues caused by their commits. The issues can readily be fixed without significant communication between the developer and the tester. This reduces the accumulation of errors and pressure on release days. It also reduces the miscommunication between both teams by solving one of the most common issues: one party says the code works on my machine, and the other says no, it didn't work on mine.

The major requirements of this thesis were all met. Theoretically, one of the expected outcomes of this thesis is to give an excellent daily estimate of the quality of both the static and dynamic state of an application. An Infrastructure has been developed that will be able to achieve all the expected outcomes in the future. Going back to the initial quadrant diagram in figure 3.1 in the CI concepts chapter, four main groups of tests should be carried out to ensure the delivery of a high-quality application. The following status of these test groups in this thesis can be reiterated as follows.

Technology-facing tests that supports programming: The criteria under this section are well implemented in this thesis, and because of the fast pace of change in this industry, there are more sophisticated static code analyses methods that are not accounted for by the diagram. But these methods are however well implemented in this thesis.

Business-facing test that supports programming: This thesis also accounts for this quadrant, though not fully implemented.

Technology-facing tests that critique the project: Accounted for in the infrastructure but is not implemented in this thesis.

Business-facing tests that critique the project: This is made up of a series of tests with the application owners and analysts, and testers, hence organizations like the one where this thesis is implemented can hardly automate such tests.

Successful implementation of the pipeline described so far can tell the static code quality with high certainty, can also continuously deploy a delivery state of an application on which multiple acceptance tests can be carried out, and can tell if the GUI of the application is functioning correctly.

7.1 Requirements validation

#	**Requirements**	**Implemented**
REQ1	The establishment of a test environment where software can automatically be deployed.	✓
REQ2	An automated process for the dynamic quality assurance of the application deployed on the established test environment.	✓
REQ3	An automated process for software integration and static software quality assurance.	✓
REQ4	The project manager(s) and developer(s) shall be able to get daily notification about the status of the application. Success scenario: just a success message, failure scenario: an elaborate report of the executed tests and create a Jira ticket.	✓

Table 7.1: Project validation

7.2 Future Work

The project's skeleton has been constructed, and like most pipelines, it will only keep growing and getting new features. Three significant tests should be implemented in the future to enjoy the full benefits of the DevOps culture. Some improvements can also be made to the UI test, especially when significant UI changes are made, for example, if there is a new functional requirement. The following are substantial tests to be introduced in the pipeline.

- Load and Performance test (LPT)
- Transaction or regression test
- Security test

These tests will run against the deployed application. Once these tests are implemented, the pipeline will undergo a minor restructuring. All the other tests will run simultaneously on the deployed application, except the LPT. The application server will be restarted before the LPT test to ensure unique and accurate results. It could also run parallel to the tests, but new VMs will be needed. It's a bit complicated when using VMware but works a little easier and more flexible when using Amazon's AWS. Kubernetes is another sophisticated container orchestration system that can also be introduced to take the project to another level.

After executing the tests, the results should either be an XML or HTML. The test execution should also be designed to quickly transfer the results to the Jenkins server for analyses and presentation. Security and transaction tests can be carried out on the test environment VM. Still, due to its high system performance requirements, the LPT test needs a whole infrastructure dedicated to it. The pipeline shall only trigger the LPT start and collects its results for analyses and presentation.

Bibliography

[1] T. Karvonen, "Continuous software engineering in the development of software-intensive products : towards a reference model for continuous software engineering," Book, University of Oulu, 10 2017.

[2] M. Shahin, M. Ali Babar, and L. Zhu, *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*, 2017, vol. 5.

[3] J. Humble and D. Farley, *Continuous Delivery.* Pearson Education Inc, 2015.

[4] M. Taylor, A. Ravichandran, and P. Waterhouse, *DevOps for Digital Leaders: Reignite Business with a Modern DevOps-Enabled Software Factory.* Springer Science+Business Media LLC, 2016.

[5] I. Sommerville, *Software Engineering.* Pearson Education Limited, 2016.

[6] S. M. Duvall, Paul with Mtatyas and A. Glover, *Continuous Integration.* Pearson Education Inc, 2007.

[7] R. Dewhurst. Owasp static code analysis. Accessed: 2022-06-09. [Online]. Available: https://owasp.org/www-community/controls/Static_Code_Analysis

[8] Synopsys, "2022 open source security and risk analyzes report," 2022.

[9] OWASP. Owasp dependency-check. Accessed: 2022-07-01. [Online]. Available: https://owasp.org/www-project-dependency-check/

[10] A. Mouat, *Using Docker.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media Inc, 2015.

[11] R. McKendrick, *Extending Docker.* Packt Publishing Ltd, 2016.

[12] Techplayon. Virtual machine vs container. Accessed: 2022-06-30. [Online]. Available: https://www.techplayon.com/virtual-machine-vs-container/

[13] O'Reilly. O'reilly learning platform. Accessed: 2022-06-30. [Online]. Available: https://www.oreilly.com/library/view/using-docker/9781491915752/ch04.html

[14] abhisbyk. Oracle docker images. Accessed: 2022-07-21. [Online]. Available: https://github.com/oracle/docker-images/tree/main/OracleDatabase/SingleInstance

[15] G. Tiwari. Owasp dependency-check. Accessed: 2022-07-01. [Online]. Available: https://www.browserstack.com/guide/selenium-grid-tutorial

Appendix

1 Jenkinsfile

```
def temp_artifacts = '<artifact.name>-'
def server_ip = 'test env IP'
def source_dir = 'EMPTY'
def testEnv_credentialsId = '<cred1>'
def bitbucket_credentialsId = '<cred>'

pipeline {
agent any
 tools {
 // Install the Maven version configured as "M3" and add it to the path.
 maven "maven-3-6-3"
 }
stages {
    stage('Checkout') {
        steps {
            //cleanup
            dir('SELENIUM'){
                deleteDir()
            }
            dir('cicd_scripts'){
                deleteDir();
            }
            //clone application project
            git branch: '0-parent-ecms', credentialsId: "${
                bitbucket_credentialsId}", url: 'https://dps-bitbucket.intra
                .dps-online.de:8443/scm/em/ecms_j.git'
            //clone application-cicd-scripts
            dir('cicd_scripts') {
                checkout scm
            }
            //clone webadmin test
            dir('SELENIUM') {
                git credentialsId: "${bitbucket_credentialsId}", url: 'https://dps-bitbucket.intra.dps-online.de:8443/scm/emi/application-webadmin-selenium.git'
            }
        }
    }
    stage('Build') {
        steps {
            //Also contains unit test
            bat 'mvn clean install surefire-report:report -Daggregate=true -Papplication'
        }
        post {
            always {
                publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: 'target/site', reportFiles: '*.html', reportName: 'Unit Test Report', reportTitles: 'unit-test-report'])
            }
        }
    }
    stage('Static code Analyzes') {
        steps {
            //pmd (checks source code)
            bat 'mvn pmd:pmd -Papplication'
            bat 'mvn site -Papplication'
            //generate SBOM for Nodejs packages, used below by dependencyCheck
            nodejs('NodeJS_Jenkins') {
            dir('<directory to Nodejs project>') {
                bat 'npm clean-install'
            }
        }
        script{
            def pmd = scanForIssues tool: pmdParser(pattern: '**/pmd.xml')
            publishIssues issues: [pmd]
            def spotbugs = scanForIssues tool: spotBugs(pattern: '**/spotbugsXml.xml')
            publishIssues issues: [spotbugs]
            publishIssues name: 'All Issues', issues: [pmd, spotbugs]
        }
        //Dependency-Check Analyses (generates java SBOM by default)
        dependencyCheck additionalArguments: '--disableYarnAudit', odcInstallation: '5.3.2'
        dependencyCheckPublisher pattern: '**/dependency-check-report.xml', failedTotalCritical: 364, failedTotalHigh: 838, failedTotalMedium: 810, unstableTotalCritical: 364, unstableTotalHigh: 838, unstableTotalMedium: 810
        }
    }
    stage('Build Middlelayers') {
        steps {
            //build middlelayers
            dir('<directory to application middle layer>') {
                bat 'mvn -Pprod clean install package -DskipTests'
            }
        }
    }
    stage('Delivery Package') {
        steps {
            //Build delivery package
            dir('application-ecms-delivery') {
                bat 'mvn clean install -Pmonitoring exec:java'
            }
            withCredentials([usernamePassword(credentialsId: "${testEnv_credentialsId}", passwordVariable: 'password', usernameVariable: 'username')]) {
            script{
                def remote = [:]
                remote.name = 'cicd-server'
                remote.host = "${server_ip}"
                remote.user = username
                remote.password = password
                remote.allowAnyHosts = true
                temp_artifacts = getCommandOutput('FOR /D /r %%G IN ("application-ecms-delivery\\sb.ecms.dps.delivery-3.3.31.*") DO @echo %%~fxG' )
                source_dir = "${temp_artifacts}\\components"
                 stage('Copying artifacts') {
                    //cleanup
                    sshCommand remote: remote, sudo: true, command: "rm -rf /root/cicd_scripts"
                    sshPut remote: remote, from: 'cicd_scripts/.', into: '/root'
                    sshCommand remote: remote, sudo: true, command: "rm -rf /root/cicd_scripts/database /root/cicd_scripts/microservices /root/cicd_scripts/webserver"
                    sshPut remote: remote, from: 'cicd_scripts/database/.', into: '/root/cicd_scripts'
                    sshPut remote: remote, from: 'cicd_scripts/microservices/.', into: '/root/cicd_scripts'
                    sshPut remote: remote, from: 'cicd_scripts/webserver/.', into: '/root/cicd_scripts'
                    sshPut remote: remote, from: source_dir, into: '/root/cicd_scripts'
                    sshCommand remote: remote, sudo: true, command: "chmod +x /root/cicd_scripts/*"
                }
            }
        }
    }
}
stage('Deploy') {
    steps {
        //setup file structure on cice-server and starts docker containers
        node("cicd-server") {
            dir('/root/cicd_scripts'){
                sh 'sed -i -e \'s/\r$//\' *.sh' //ensure setup.sh is a linux script
                sh 'bash setup.sh'
            }
        }
    }
}
stage('Populate DB') {
    steps {
        node("cicd-server") {
            //executes sql script in database folder
            sh 'docker exec -u 0 database-server bash -c "cd \'/opt/oracle\'; ./populateDB.sh;"'
        }
    }
}
stage('Security Test') {
    steps {
        //OWASP ZAP test or Radamsa
```

```
            echo 'Empty stage! update me'
        }
    }
    stage('Webadmin Test') {
        steps {
            //Selenium tests
            dir('SELENIUM'){
                bat 'mvn clean test -DtestngFile="webadmin.xml"'
            }
        }
        post{
            always {
                //Webadmin Test results
                publishHTML([allowMissing: true, alwaysLinkToLastBuild: true, keepAll: true, reportDir: 'SELENIUM/target/surefire-reports', reportFiles: 'index.html', reportName: 'UI Test Report', reportTitles: ''])
            }
        }
    }

    stage('End') {
        steps {
            node("cicd-server") {
                dir('/root/cicd_scripts'){
                    sh 'docker-compose down'
                }
            }
        }
    }
}
post {
    failure {
        emailext body: '!! FEHLER !! "\n" Fur mehr infos: $BUILD_URL', subject: 'CICD Nightly Error', to: 'wolfgang.tebah@dps.de'
    }
    success {
        emailext body: '### ERFOLG! ### "\n" Fur mehr infos: $BUILD_URL', subject: 'CICD Nightly Feedback', to: 'wolfgang.tebah@dps.de'
    }
}
}
//get exactly the folder name from filter
```

```
def getCommandOutput(cmd) {
    if (isUnix()){
        return sh(returnStdout:true , script: '#!/bin/sh -e\n' + cmd).trim
            ()
    } else {
       stdout = bat(returnStdout:true , script: cmd).trim()
       result = stdout.readLines().drop(1).join(" ")
       return result
    }
}
```

2 docker-compose.yml

```
version: '3.7'

services:
  database-server:
    container_name: database-server
    image: localhost/oracle/database:18.4.0-xe
    ports:
      - "1521:1521"
      - "5500:5500"
    healthcheck:
      test: [ "CMD", "/opt/oracle/checkDBStatus.sh" ]
      interval: 1m30s
      timeout: 10s
      retries: 9
      start_period: 40s
    volumes:
      - oradata:/opt/oracle/oradata:z
      - db_log:/opt/oracle/diag/rdbms/xe/XE/trace:z
      - /path/to/db/artifacts:/opt/oracle/scripts/startup:z
    environment:
      - ORACLE_PWD=testDB123

  microservices:
    container_name: microservices
    environment:
      SPRING_JPA_HIBERNATE_DDL-AUTO: create
      SPRING_PROFILES_ACTIVE: docker-oracle
    build:
      context: ./location/of/Dockerfile
      dockerfile: Dockerfile_microservices
    command: bash -c "./start_microservices.sh; tail -F anything"
    ports:
      - "<external port>:<container port>"
    healthcheck:
      test: ["CMD-SHELL", "curl --fail -s http://localhost:<container port>
          || exit 1"]
      interval: 1m30s
      timeout: 10s
      retries: 4
      start_period: 40s
    volumes:
      - microservices_logs:/path/to/tomcat/logs:z
    depends_on:
      database-server:
        condition: service_healthy

  application-server:
    container_name: application-server
    build:
      context: ./components/02_appserver
      dockerfile: Dockerfile_appserver
    command: bash catalina.sh run
    ports:
      - "<external port>:<container port>"
    volumes:
      - webserver_logs:/path/to/tomcat/logs:z
    healthcheck:
      test: [ "CMD-SHELL", "curl --fail -s http://<test env IP>:<test env
          port>/application/login || exit 1" ]
      interval: 1m30s
      timeout: 10s
      retries: 7
      start_period: 40s
    depends_on:
      database-server:
        condition: service_healthy
      microservices:
        condition: service_healthy
  end:
    image: alpine:latest
    command: echo 'Just waiting for application!'
    depends_on:
      application-server:
        condition: service_healthy

volumes:
  microservices_logs:
  webserver_logs:
  oradata:
  db_log:
```

3 setup.sh

```bash
#!/bin/bash

db_directory="/dir/to/database/artifacts"
exists=$(grep -c "oracle:" /etc/passwd)
#### rearrange folder structure to comply with the docker-compose norms####
sleep 10
## some rearrangement here, if needed.
sleep 10
#### create user ####
function createUser {
  sudo adduser --disabled-password oracle
  mkdir /home/oracle/Deliverables
  mkdir /home/oracle/oradata
}
#### update sql scripts to enable connectivity ####
function updateDefines {
  cd /location/of/database/artifacts || return
  filename="<filename>.sql"
  # Assign test environment variables
  HOST="HOST=<test-environment ip>"
  PORT="PORT=<test-environment port>"
  SID="SID=XE"
  usrOwner="usrOwner=<db-test-username>"
  usrOwner_Pwd="usrOwner_Pwd=<db-test-password>"
  #make database changes accordingly
  sed -i "s/HOST=[0-9.]*/$HOST/" $filename
  sed -i "s/PORT=[0-9]*/$PORT/" $filename
  sed -i "s/SID=[a-Z]*/$SID/" $filename
  sed -i "s/userOwner=[a-Z_]*/$usrOwner/" $filename
  sed -i "s/usrOwner_Pwd=[a-Z_]*/$usrOwner_Pwd/" $filename
  echo 'alter session set "_oracle_script"=true;' >> $filename   #enable
      scripts to be excuted on DB externally
  mv -f /source/database/artifacts /destination/in/test/environment
}
##### update application files to enable connectivity ####
function updateServerXml {
  cd /path/to/tomcat/conf || return
  serverFile="server.xml"
  sed -i "s/<unwanted IP>/<test environment IP>/g" $serverFile
  sed -i "s/<unwanted DB SID>/XE/g" $serverFile
}
```

```
#### main setup ####
if [ "$exists" == "1" ]; then
    [[ -d "$db_directory" ]] && rm -rf $db_directory
    updateDefines;
    updateServerXml;
else
    createUser;
    updateDefines;
    updateServerXml;
fi
#### Build docker images and containers ####
cd /test/dir || return
docker-compose down
docker volume prune --force
docker rmi -f docker.io/library/cicd_scripts_application-server:latest
docker rmi -f localhost/cicd_scripts_application-server:latest
docker rmi -f docker.io/library/cicd_scripts_microservices:latest
docker rmi -f localhost/cicd_scripts_microservices:latest
docker rmi -f docker.io/library/alpine:latest
echo '#################################################################'
echo '############### Done cleaning old docker objects #################'
echo '## Please wait for server to be alive! this might take a while. ##'
echo '#################################################################'
docker-compose up --remove-orphans -V --detach
sleep 30
#### try again if server is up ####
[[ $( curl --head --silent --fail http://<test environment IP>:<test env
    port>/<application/login> 2> /dev/null ) ]] || exit 1
sleep 30
echo '####################################'
echo '#### Application Up and alive! ####'
echo '####################################'
```

Glossary

CICD Pipeline A series of steps executed on Jenkins, which tries to automate DevOps practices. 1

DevOps Set of practices that combine both Development and IT Operations. 2, 3

DevSecOps DevOps with integrated security. 21

Acronyms

CI Continuous Integration. 15, 16

CICD Continuous Integration and Continuous Delivery. 3

CSE Continuous Software Engineering. iii, iv, 2

GUI Graphical User Interface. 26

NIST National Institute of Standards and Technology. 20

NVD National Vulnerability Database. 20

OWASP The Open Web Application Security Project. 20

QA Quality Assurance. 1, 16

SBOM Software Bill of Material. 20

SCA Software Composition Analyzes. 20

VM Virtual Machines. 21

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

[redacted]

Hamburg, 22. August 2022   Tebah, Wolfgang Azipon