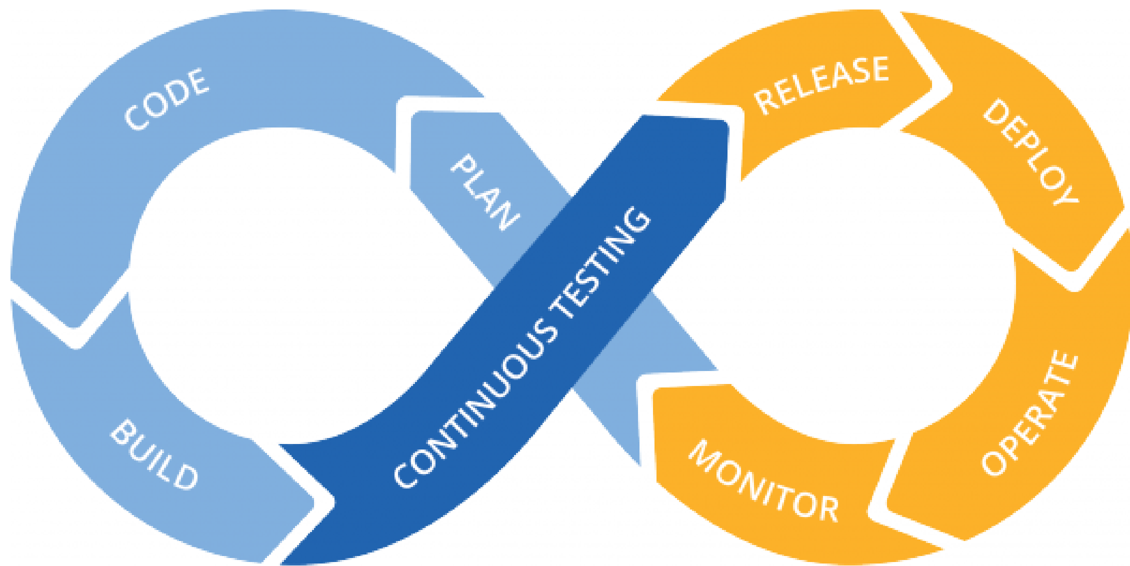
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Improving the Continuous Integration Process for the Vehicle Dynamics Software at VCC using Docker

Master's thesis in Computer Systems and Networks

Sara Fatih
Surafel Meheret Alamneh

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Improving the Continuous Integration Process for the Vehicle Dynamics software at VCC using Docker

Sara Fatih
Surafel Meheret Alamneh

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

Improving the Continuous Integration
Process for the Vehicle Dynamics Software at VCC using Docker

Sara Fatih

Surafel Meheret Alamneh

© Sara, Surafel, 2018.

Supervisor: Miroslaw Staron, Department of Computer Science and Engineering
Advisor: Tony Heimdhal, Volvo Car Corporation
Examiner: Elad Schiller, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2018

Improving the Continuous Integration
Architecture for the Vehicle Dynamics Software at VCC using Docker
Sara Fatih
Surafel Meheret Alamneh
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This research studies the implications that merging Docker with Continuous Integration has on automotive software architecture. This is important because continuous software quality assurance is crucial for safety-critical automotive software. Continuous integration promotes this quality assurance by fetching the test environment then running tests on the product every time there is a new software update. Docker is used in this study to containerize the test environment and the product under test to decrease the time it takes to finish these tasks. To evaluate the impact of Docker, a number of time criteria were used to measure the time it takes to get feedback from tests in three different Docker scenarios implemented in Continuous integration. The findings from these evaluations show that Docker can decrease time up to 50.38 per cent. This study concludes by explaining how using every one of these Docker scenarios can impact the software architecture of automotive software. We show this impact on a specific automotive product called Vehicular Dynamics software.

Keywords: Docker, Testing, VDSW,Continuous Integration, GoCD, Feedback loop

Acknowledgements

Firstly, we would like to express our sincere gratitude to our supervisor Miroslaw Staron, for his continuous support and motivation. We also want to extend our gratitude to Tony Heimdahl, Kashif Siraj and Magnus Rising in VCC for giving us the opportunity to do our thesis within this interesting area in the first place and also guiding us through different stages of the thesis project implementation.

Last but not least, we would like to thank the Swedish Institute for funding our studies and for giving us the opportunity to learn a lot in our Masters program.

Sara Fatih and Surafel Alamneh, Gothenburg, October 2018

Contents

List of Figures **xi**

List of Tables **xii**

1 Introduction **1**
1.1 Aim of study 1
1.2 Problem statement 1
1.3 Limitations 2
1.4 Related Work 2
1.5 Contributions 3

2 Background **4**
2.1 Docker Containers 4
2.1.1 Introduction to Docker 4
2.1.2 Containerization vs Virtualization 5
2.1.3 Clarification of Docker terms 5
2.1.4 How to use Docker images 7
2.1.5 How to use Docker for testing 7
2.2 Testing Procedures at Volvo Cars 8
2.3 Active Safety 9
2.4 Smoke Test 10
2.5 Sanity Test 10
2.6 Detailed Vehicle test 11
2.6.1 Unit Tests 11
2.6.2 System Tests 11
2.6.3 Acceptance Test 12
2.6.4 Regression Tests 12
2.7 Continuous Integration/Continuous Deployment 13
2.8 VDSW Software development 14
2.8.1 Introduction to VDSW 14
2.8.2 VDSW Project Loop 15
2.8.3 Continuous Integration and GoCD 16
2.9 Software Architecture 17
2.9.1 The theory of the monolithic and the component-based architectural styles 18

2.9.2 The current monolithic and component-based architectures of VDSW . . . 18
2.9.2.1 Current VDSW monolithic Architecture . . . 19
2.9.2.2 Current VDSW component-based Architecture . . . 19

3 Related Work **22**

4 Methodology **27**
4.1 Problem Investigation and Awareness . . . 27
4.2 Research Methodology . . . 28

5 Dockerization **30**
5.1 Testing scenarios and environment setup . . . 30
5.2 How can Docker be integrated into a Continuous Integration process? 35

6 Results **38**
6.1 Research questions and answers . . . 38
6.2 Our proposed Continuous Integration model for VCC . . . 40
6.3 Proposed component-based software architecture for vehicular dynamics 42
6.4 Implementation of Docker in VCC's Continuous Integration pipeline . 43

7 Evaluation **46**
7.1 Evaluation Environment . . . 46
7.2 Measurement criteria and methods . . . 46
7.3 Evaluation of the three Docker scenarios . . . 48
7.3.1 Scenario 1: Testing without Docker . . . 49
7.3.2 Scenario 2: Testing with a single shared Docker image for both the product and the test . . . 49
7.3.3 Scenario 3: Testing with separate Docker images for the product and the test . . . 50
7.3.4 Total time comparison for different scenarios . . . 51
7.4 Evaluation of the baseline pipeline and the proposed pipeline that uses Docker for the industrial partner VCC . . . 51

8 Conclusion **53**
8.1 Discussion of Docker's impact on testing . . . 53
8.2 Discussion of Docker's impact on software architecture . . . 54
8.3 Conclusion . . . 54

Bibliography **56**

A Appendix 1: Min.java **I**

B Appendix 2: MinTest.java **II**

C Appendix 3: script1.sh **III**

D Appendix 4: script2.sh **IV**

E Appendix 5: Test image Dockerfile **V**

F Appendix 6: Test dependencies Dockerfile **VI**

G Appendix 7: App image Dockerfile **VII**

H Appendix 8: docker-compose.yml for proposed Docker solution **VIII**

I Appendix 9: Dockerfile for proposed Docker solution **IX**

J Appendix 10: Scenario 2 Dockerfile **X**

List of Figures

2.1 Docker images and layers 6
2.2 Download of Busybox Docker image 7
2.3 Current CI architecture 9
2.4 Testing Procedures [1] 12
2.5 The difference between continuous integration and deployment. 14
2.6 Process refinement loop 15
2.7 VDSW Software build chain 16
2.8 Monolithic Architecture of VDSW 19
2.9 The current VDSW Software Architecture 20

5.1 Baseline scenario 32
5.2 Vanilla Ubuntu machines with a single shared Docker container for both the product and the test 33
5.3 Vanilla Ubuntu machines with multiple Docker containers 33
5.4 Single-pipeline GoCD server 36
5.5 Two-pipelines GoCD server 37

6.1 Proposed Continuous Integration model for VCC 42
6.2 Proposed Component-based VDSW architecture 43
6.3 Proposed Docker image for VCC's pipeline 44

7.1 Test Environment setup and execution without docker 49
7.2 Test Environment setup and execution with a single docker image . . 49
7.3 Test Environment setup and execution with multiple docker image . . 50
7.4 Total time comparison for Scenarios 51

List of Tables

2.1 Virtual Machines vs Docker containers [4] 5
3.1 The clusters derived from literature review [2] 23
4.1 Stages involved in Design Science methodology [2] 29
6.1 Mapping between Docker scenarios and software architectures 40
7.1 Evaluation of different scenarios 48
7.2 Time measurements for the baseline and proposed pipelines 52

1
Introduction

The first cars that were produced were purely mechanical systems [3]. But starting the 1970, electronics were introduced and concepts such as central locking and electronic fuel injection appeared. By the 1990s, the domain of infotainment, like the navigation system was used, and safety-critical solutions emerged, like Adaptive Cruise Control (ACC). As of this decade, notions like car to car communication, autonomous driving and wireless cars are becoming more common.
Software is currently providing an increasing amount of functionality in the domain of active safety in the automotive industry. Its scale is getting larger with the number of Electric Control Units (ECU), which introduces challenges in the development of software based automotive systems. Therefore, understanding the software architecture of every individual ECU as a whole is important for software design in the automotive industry [3].

1.1 Aim of study

This work aims at studying the impact of merging Docker containers with Continuous Integration on a product software architecture. Docker is a containerization engine which enables the automation of packaging and deploying software. It is a faster and lighter alternative to virtualization. Continuous Integration is the process of frequently integrating and merging source code of software. Continuous integration servers are used to trigger and run the different tasks in this process. The goal is to propose a Continuous Integration infrastructure based on Docker and to study its impact on the product architecture. The software architectures that are found to be compatible with Docker and Continuous Integration are applied on vehicular dynamics software.
Our results help to provide assistance to the architects of vehicular dynamics software by providing various possible solutions to its software architecture. These solutions may afterwards be evaluated in the contexts of specific projects, which would lead to a new architecture.

1.2 Problem statement

This study includes four research questions that the authors address:

- How can Docker affect a testing process?
  To answer this question, the authors started by studying the timing issues

that the industrial partner identified in their tests. The study of the effect of using Docker is important because the feedback loop of a testing process is crucial. The shorter it is, the better is the quality of the product. To address this, the authors experimented with Docker by implementing different Docker scenarios. To evaluate these Docker solutions, we needed to find the appropriate evaluation criteria and use them.

- How can Docker be integrated into a Continuous integration process?
  There are multiple ways to use Docker in Continuous Integration. Therefore, it is needed for this study to implement specific CI chains that are based on Docker. A CI that uses Docker needs a code repository, a Docker image repository and a CI server. So we worked on finding a more suitable layout of those three main components in the CI chain.
- How is it possible to integrate Docker into VCC's current CI process?
  Integrating Docker into VCC's current CI process can be limited due to the nature of the components that are tested and the environment where the tests are conducted. Docker can only be used in specific platforms. So the study of the limitations related to Docker usage is needed.
- What are the implications on the software architecture of vehicular dynamics software if Docker is used?
  Not all software architectures can fit into a certain Docker usage scenario. Hence, it is important to study how the software architecture of this product needs to change to work properly with Docker and CI.

1.3 Limitations

One of the challenges faced in this study are the compatibility problems between Docker and the other components. Docker works with both Linux and Windows but only on Windows 10. Therefore it is not possible to containerize all the software components. Also, Dockerizing the python scripts and all the files needed for tests is not possible. This is due to the fact that there is only one Python script that runs, which in turn invokes a number of other Python scripts. As part of those scripts, calls are made to external windows libraries, which are possible inside a Linux Docker image.

1.4 Related Work

A proposal of a Continuous Integration infrastructure that is based on Docker for our industrial partner Volvo Cars Corporation (VCC) . We based our model on the work done by Jan Bosch in [4], which is explained in detail in section 3. Bosch's model provides the three fundamental components of a Continuous Integration process that are input, scope activities, and result handling. However, it does not include Docker. Our contribution is in determining how Docker can be used in this model. Using Docker requires at least two steps: Building an image and then pulling it from the repository for usage. We decided to include the first step (the build) in the

input and the second step (pulling the image) in the scope activities. This solution fits with our final implementation of a new Continuous Integration process for VCC in section 6.4.

1.5 Contributions

This study contributes to the literature with the following:

1. An evaluation of different Docker usage scenarios using different time measurement criteria. As we were looking into how Docker affects the time it takes to get test results, also called as the feedback time, we implemented a sample app and a sample test and we used Docker in three different ways. Since caching is used, theoretically Docker will save time. The first time measurement that we made was for testing without Docker. The second measurement we made was for using a single Docker container for both the app and the test. The third measurement was for two Docker containers: one for the app and one for the test. Having two separate containers gives portability to the app and the test. The results we found did confirm our theory. When comparing to not using Docker, using one Docker container for both the app and the test split the time into almost half (0.5038). However, having two separate containers took more than double the time (2.7341) because building Docker images consumes time, and as the number of images increases, that amount of time increases.
2. A proposal of a mapping between software architecture and Docker usage scenarios. The two software architectures that we investigate on are the monolithic and the component-based architecture. The reasoning behind this investigation is that since the component-based architecture means that the components are independent, this would require a different Docker container for each component. On the other hand, a monolithic architecture would fit using one Docker container for the entire software instead of multiple containers. We implemented two Docker scenarios. The first one uses a single Docker container for both the app and the test. The second scenario uses two Docker containers: one for the app and one for the test. Having a dedicated Docker container for the app provides portability for the software. In addition to this, having a dedicated container for the test provides both portability for the test and concurrency of running tests on different software at the same time. The third Docker scenario splits the software into different Docker containers. In this case, the software architecture is the component-based architecture. We did not implement this scenario because it requires further investigation on how to deal with the communication between the different Docker containers. Kubernetes, which is a system that manages multiple containers, is one of the solutions that enables this orchestration.
3. A proposal of two software architectures for vehicular dynamics software. Vehicular dynamics software is one of the most important ones at VCC. In order for us to propose the two architectures which are the monolithic and the component-based one, we worked alongside the team of developers that is dedicated for this software.

2
Background

This chapter will start with explaining in detail about docker and the difference with virtual machines and will proceed to explain how docker containers are created from a dockerfile. Then, it will elaborate about the types of tests running in vehicle test. Finally, the chapter will explain more about continuous integration and deployment (CI/CD) in VDSW software architecture.

2.1 Docker Containers

Before we proceed to the details of Docker its important to understand the flow of events and also some key concepts in docker.
Dockerfile is the basic component of docker. It is simply a text file that contain a commands to create a docker image. By building the dockerfile docker image will be created. Docker image can be stored in private registry or public regstry like docker hub. The choice will depend on how secure the docker image will be. Running the docker image will create a docker container. Multiple application could run on the created docker container.

2.1.1 Introduction to Docker

Many new technologies emerge to provide more decisive automation and to overcome system complexities in the overwhelming usage of the IT domain across different industries. Virtualization was put forth to achieve portability and to optimize IT infrastructure. However, virtual machines reduce performance and they are slow and heavyweight. [5]
Containerization based on Docker was designed to overcome these challenges. Docker allows for accelerated and risk-free containerization. Raj proposes a definition that encapsulates all the aspects of Docker as follows:
"Docker is an open source containerization engine, which automates the packaging, shipping, and deployment of any software applications that are presented as lightweight, portable, and self-sufficient containers, that will run in any operating system without any dependencies."[5]
All the necessary components needed for software to run are encapsulated in a Docker container, which allows for software to run independently from the host machine. The components needed for the software to run are called dependencies and they can be scripts, libraries, binaries, jars, etc.
Docker containers can run fluently on systems that have a x64 Linux kernel. They

have their own network interface and process space, in addition to a different `/sbin/init` from the one of the host machine. Other operating systems like Windows and Mac can also run Docker containers but they need further workarounds to achieve that.
The primary components of the Docker solution are the Docker engine and the Docker Hub. The Docker engine is what allows for creating and running Docker containers. The DockerHub is a public repository that stores Docker images which can be combined to produce more complex containers. In the example of a Linux host machine, containers can run directly. The Docker engine can build and manage several containers at the same time.

2.1.2 Containerization vs Virtualization

Virtual machines are the replicates of an operating system [6]. Each virtual machine includes all the drivers and applications necessary to run the OS. To be able for a virtualized server to interact with the computer hardware, a virtual machine manager also known as hypervisor is used [7].
The main difference between virtual machines and Docker containers is that in the case of Virtual machines, the resources are matched with the operating system and hypervisor. That is the main reason different instances of operating systems are running in parallel. In the case of Docker, the containers are running inside the docker engine. Docker does not use hypervisor like virtual machines do. Therefore, containers are so small and they enable a faster start-up and better performance compared to the virtual machines. Table 2.1 shows the main differences between virtual machines and Docker containers.

Virtual Machines (VMs)	**Containers**
Represents hardware-level virtualization	Represents operating system virtualization
Heavyweight	Lightweight
Slow provisioning	Real-time provisioning and scalability
Limited performance	Native performance
Fully isolated and hence more secure	Process-level isolation and hence less secure

Table 2.1: Virtual Machines vs Docker containers [4]

2.1.3 Clarification of Docker terms

Docker images and layers
A Docker image is an assembling of the files that construct a software application. It originates from a base image and every future commit to the original image is stored in a new layer. To have a specific behavior or feature, different images can be derived by attaching additional modules to the base image. The different layers that are created through the commits do not make any changes to the original image and pre-existing layers.
Images are read-only and every time new modules are added, a new image with a

new name is created. Docker images are necessary to develop and deploy Docker containers, which will be explained later. Figure 2.2 illustrates a Debian base image with two images that lay on top of it (Emacs and Apache) for additional features. The container is the uppermost layer and it is writable. Another example could be that an image has an Ubuntu operating system with an Apache server and the web application installed.
Generally speaking, the base image is an operating system. It can be one of Linux distributions like Debian for example. Then additional modules are added through images. The combination of all those modules and the base image is what is necessary for a Docker container.
Concerning layers, a Docker layer can either be a read-only or a read-write image. However, the topmost layer in a container stack is always the writable layer which represents the Docker container.

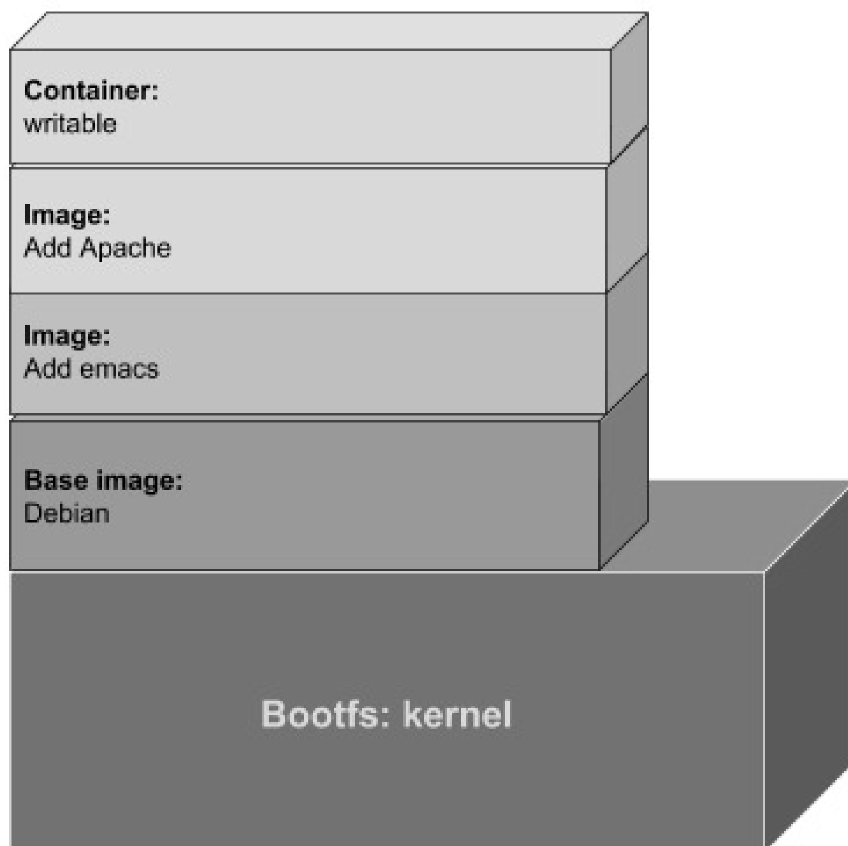


Figure 2.1: Docker images and layers

Docker containers
A Docker container can be thought of as a read-write layer that sits on top of one or several read-only images. When the container runs, the required images are merged together by the Docker engine. While it is running, the changes made into the container through interactions with the user (for example a web application) are

merged as well. This makes up a "self-contained, extensible, and executable system" [5].
The changes are combined with the subcommand `docker commit` to form a new image that constitutes a new layer on top of all the pre-existing layers.
Docker registry and Docker repository
The Docker registry is a public platform that enables registering Docker images created by developers from around the world. Those images are verified, validated and refined to make sure their quality is high. In order to dispatch a Docker image in the Docker registry, the `push` command can be used. The Docker registry is often confused with the Docker repository. The registry is only used for registering the images, while the repository is used to actually store them in a public storage system. Every user or account has their own unique repository.

2.1.4 How to use Docker images

If a machine has the Docker engine installed in it, images can be downloaded from the Docker registry. The Docker registry contains a large number of applications that can range from advanced applications to Linux images. To download images from the registry, the `docker pull` command is used. An example of a lightweight version of Linux is called busybox and it can be downloaded with the command `sudo docker pull busybox`. The displayed messages after the download are shown in figure 2.2.
After downloading the Docker image, the user can run the Docker container. In the example of the busybox, `Hello world!` can be echoed using the busybox image. This can be done by using the command:
`sudo docker run busybox echo "Hello World!"`

```
$ sudo docker pull busybox
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Important:
image verification is a tech preview feature and should not be relied on
to provide security.
Status: Downloaded newer image for busybox:latest
```

Figure 2.2: Download of Busybox Docker image

2.1.5 How to use Docker for testing

One simple way for testing an application is when both the application and its test suite are in the same Docker image. For this example, a java application and a junit test are used. The Docker image can be created by using a Dockerfile that contains the following lines of script :
`FROM openjdk`

```
COPY *.java .
COPY *.jar .
RUN javac *.java
CMD java MinTest
```

The first line fetches the openjdk image and uses it as the base image. The openjdk image is needed to compile and run java files.
The second line is used to copy all the java files in the current directory of the Dockerfile into the context of the image. The context of image is the storage made of the files needed by an image. The java files needed in this case are the application file and the test file.
The third line copies the jar files that are needed for the junit test to run. The hamcrest and junit jar files are fetched in this case.
The fourth line compiles the application file and the test file by using the texttt Javac command. The compilation is needed to run the test later.
Those four lines are the ones executed when the image is built using the command `Docker build .`
After the image is built, the test can run using the fifth line by running the Docker container. The command used is `Docker run dockerImageId`.

2.2 Testing Procedures at Volvo Cars

The current continuous integration process of Volvo Cars follows procedures as listed on figure 2.3, then eventually draws a report. In the first place software updates come from the software providers. After the software update is done, the continuous integration process of Volvo Cars is automatically triggered and conducts some tests like Smoke, Sanity and Vehicle tests. In figure 2.3, the feedback loop is the time from which the continuous integration tool is triggered until the final report is generated. The foundation of this study is based on this feedback loop. For the case of Volvo Cars specifically in active safety unit, the feedback loop takes 40 minutes, which is relatively long. The authors aim to integrate Docker into vehicular dynamics software to decrease the feedback loop of the continuous integration process. Docker does not only decrease the feedback loop but it also has the possibility to make the test environment portable. The continuous integration architecture is currently used by two different groups namely the software developers and testers. The software developers will do the code update on the shared repository and the testers run different tests as soon as there is a code update. The testers need to download necessary software for the software testing purpose to their local machine. The docker image contains all the necessary software needs for testing inside it. So, if there is a Docker image inside the CI process, the testers just need to pull the docker image and run it as a container to be able to run tests.

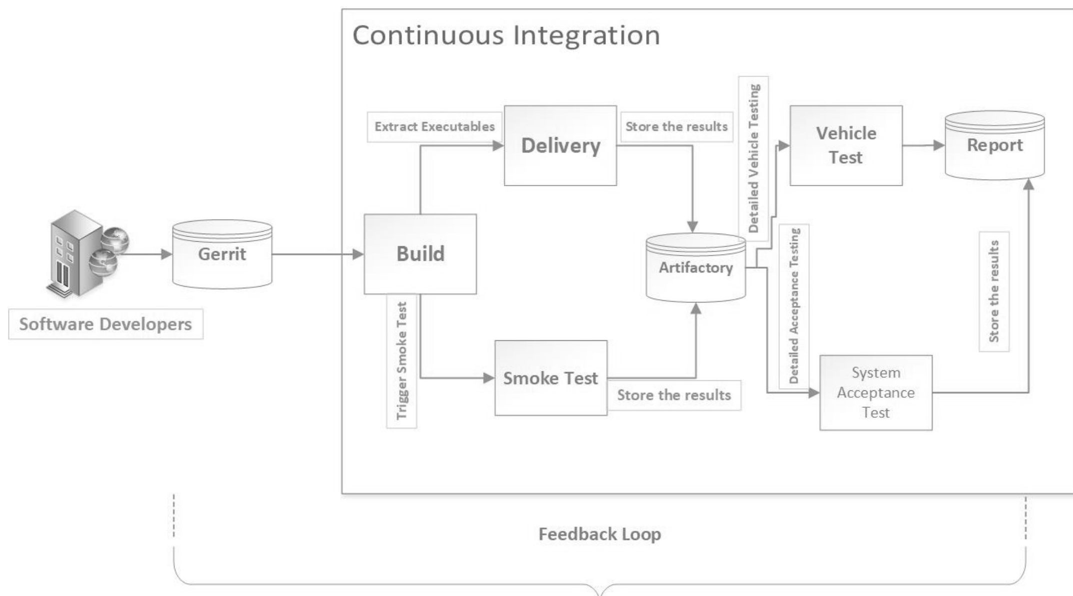


Figure 2.3: Current CI architecture

As can be observed from the figure 2.3 the software updates are coming from different software providers and they have a shared repository. As soon as a new commit happens on the shared repository, the build will be automatically triggered. A successful software build will trigger the smoke test and send the results of the test to the artifactory. The delivery phase also extracts the executable and stores it in the artifactory. Finally, the detailed vehicle tests such as Vehicle and system acceptance tests will be performed and the outcome of the tests will be reported.

2.3 Active Safety

Active Safety is an area where automotive companies like Volvo Cars launch software and hardware features that get triggered to protect the driver.
The VCC active safety department develop systems such as Collision Avoidance, Automatic Parking, Driver Support and Brake system Software. They develop and integrate system and functionality into a complete product, encompassing Software, Hardware and sensors as radar, camera, ultrasonic and Lidar, that build a comprehensive perception of the surroundings.
The department is responsible for the development and verification of Advanced Driver Assistance System, Protective Safety, Vehicle Motion Stability and last but not least Autonomous Drive Function, Software and Sensor Platform.
There are different safety features in the current modern vehicle specifically on Volvo Cars. Some of the safety features include:

- **Adaptive cruise control:** Adaptive cruise control (ACC) is one of the most demanding automated driving safety application [8]. It is a special car feature that enables the car to slow down or increase speed based on the car which is travelling ahead. ACC is responsible for insuring the safety of the vehicle by continuously tracking the vehicle distance from its immediate leader and also by keeping the distance between two cars in a safe range.

- **Automatic emergency braking:** Nowadays, the use of automatic emergency braking is becoming so common. It helps by automatic brake is also a safety feature that enables the car to automatically brake when an object which is in a near distance is appeared on the drives way.
- **Intelligent speed assist(ISA):** ISA is also one of the safety features in the modern cars that help the driver to adjust the speed according to the speed limit of the road. The speed limit information could be gathered from digital map or speed limit recognition technologies.

2.4 Smoke Test

The smoke test is a type of test used to check if the most important or critical parts of the software are working or not. In other words, it is a quick test to see if the software that goes to the smoke test catches fire when it runs for the first time. For example, to run a smoke test on a simple software program we just need to run the program and check weather if it is successfully running without crashing. In the case of Vehicular dynamics software, the smoke test is about checking whether powering on the hardware that is needed for this module and running the software on it will cause any physical damage or "smoke" as in a fire in the hardware.
The Smoke test branch at Volvo Cars is automatically triggered by using python scripts. On a successful software build, it fetches the executable and required test environment from a version control system to perform more critical tests on the actual target. After the smoke test is done on the deliverable ,the test results are uploaded to artifactory corresponding to every delivery.
The bench setup also consists of a dedicated desktop PC (hereby called Host PC) acting as GoCD Agent connected to GoCD Server over LAN. The software download shall also be initiated through Python scripts. The target ECU which is VDSW is connected to Vector Hardware over CAN and Flexray that simulates Vehicle environment and execute test cases. The Ethernet switch acts as an interface between the VDSW, Host PC and the vector hardware Ethernet interface. The Host PC shall control the programmable power supply via RS232 port to power target ECU and the Vector hardware independently.

2.5 Sanity Test

Sanity Test is also another way to perform fast and efficient tests before performing detailed software tests. The sanity check is implemented after each new software build. If there is any minor update or new functionality on the source code of the software, the sanity check is triggered automatically. The sanity check makes sure that the new software update does not introduce any bugs or that the previous bugs are solved. [9]
The main purpose of the Sanity check is to roughly determine that the new functionality of the software is working as expected. If the sanity check pass, more detailed vehicle tests are done on the software. But if the sanity check test fails, the

new software build is rejected. This saves more time and also costs in doing more detailed software testing.
The sanity test at Volvo Cars is also initiated by using the python script as soon as there is a software update on the shared repository. In the case of vehicular dynamics software, the sanity test is conducted by running the updated component alongside the old components to check if they function correctly together without introducing new bugs. After the test is done on the software the report will be sent to the artifactory.

2.6 Detailed Vehicle test

This section will explain about the tests that are conducted after the sanity and smoke tests are passed. In the smoke and sanity tests which is a subset of whole test suite, we verify most important functionality of the software and that it is stable and do not crash. So that tester can take the software over and start doing further extensive tests.

2.6.1 Unit Tests

This is one the detailed testing procedures that involves testing each unit or components of the software. The purpose of unit testing is to test and check each components are working as intended [10]. Basically unit test works by giving certain input to the software component that needs to be tested. Based on the inputs determine if the functions are returning the appropriate return values and also the errors should be handled gracefully. There are some advantages companies gain by using unit test in the development environment such as:

- To test the functionality of each component. As the testers do more and more unit tests on a software component, they will have a test suite that is very suitable for the tests that will made in the future.
- Adding unit test on the software development process or part of the continuous integration process may allow to catch the software failures early in the development process.

As explained on figure 2.9 there are different components on the vehicle dynamics software architecture and each software components are doing different functions. Unit test will be done independently at each component. Sometimes unit test can also be applied on some area of the code at some specific software component such as VehIncAg (Vehicle inclination angle). The results will then be stored on the artifactory for further analysis.

2.6.2 System Tests

System testing will test the software as a system. In another words, the the system testing will be done on the total and integrated software system. System testing will based on the software requirement specification (SRS) document. System testing has also a an important role on validating and verifying the software architecture.

It is very important to perform system testing before the product is released to the end users.

2.6.3 Acceptance Test

Acceptance test is the final step of the testing procedure where the software checks for the acceptability. At this stage the product is getting ready for the delivery to the end users. So the main purpose of this testing is to check weather it is really ready to be released or not. Acceptance test can also check if the software includes all the functionality that customers requested.

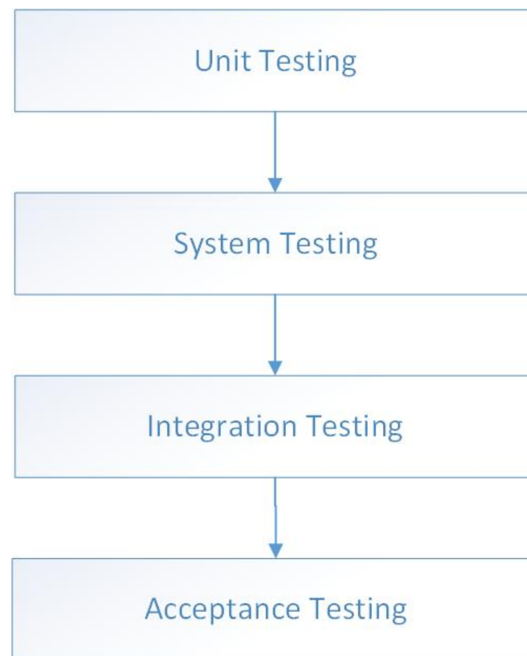


Figure 2.4: Testing Procedures [1]

From figure 2.4 we can observe that, Unit testing is done on the low level or it is performed on the functionality of the software product. After unit testing is applied on the software the next step is system testing which could be done on the complete software rather than software components. System testing will evaluate the entire software according to some specific requirements. Integration test will combine the components that are created on the unit test. After combination, the integration test will be applied on the group. Finally, the acceptance check will make sure the final software will comply the user requirements also also on the business perspective.

2.6.4 Regression Tests

In an environment where an iterative software development approach is applied, software regression tests plays an important role by testing the most recent updates of the product and ensuring the integrity of the software [11]. A small code update on the source code of the product may bring unexpected consequence. Regression

test will make sure the new update on the software does not fail the entire product or some functionality.
Different developers merge code which is under development from one CI pipeline to the other pipeline. Regression test will have an enormous role in this kind of scenario [11] [12]. To be able to merge, regression tests will be applied to any code that needs to merge with the main pipeline.
A good example of regression tests in Volvo cars is the project loop as explained on section 2.8.2. The software product are refined according to the new software requirement specifications. The new refinement of the product may bring new components to be introduced into the product software architecture, two or more components to be merged and create a single component or removing a component. Either the case is the regression test must be applied before the new refinement approved. New functionality of the system is also applied through the refinement of the software. To be able for this functionality to be integrated to the system the regression test must also be applied.
As explained on figure 2.3, software updates are coming from different software providers. Updates by the software providers to the shared repository are applied around 10 times per a day. Every time the updates are done the regression test must be applied on it before the updates are accepted by Volvo cars.

2.7 Continuous Integration/Continuous Deployment

Continuous Integration (CI) and Continuous Deployment (CD) are common words in the software development industries [13]. They create an environment for the organizations that enable to frequently and reliably release a new version in the software development process.
In the continuous integration (CI) environment, a number of developers can integrate and merge the source code of the software frequently. CI enables software companies to have shorter and more frequent software development release cycles. It also highly increases the software quality and the productivity of the software developers [14].
Before defining Continuous Deployment, it is very important to have an idea about continuous delivery. Continuous Delivery is an extension of continuous Integration in which each software update or release must be delivered in a sustainable way. In other words, it is an automated process to release the new software updates in an organized way [15] .

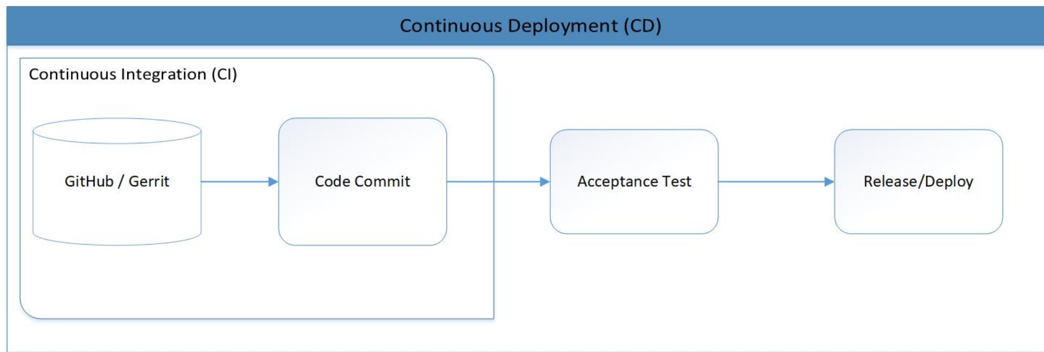


Figure 2.5: The difference between continuous integration and deployment.

There is still debate on the difference between continuous delivery and deployment [16]. Figure 2.5 shows both the processes. The main difference between the two is the production environment. In the case of continuous deployment, it is actual customers environment. The main goal of continuous deployment is to deploy every changes of the software to the production environment.

2.8 VDSW Software development

This section will briefly define what is meant by Vehicle Dynamics Software (VDSW) and the continuous integration process of this product.

2.8.1 Introduction to VDSW

The vehicle dynamics have a major role in the development of Vehicles in general [17]. Vehicles Dynamics in VCC is mainly concerns with studying the behaviour of the vehicle in motion and also the properties that the car shows in motion. Vehicle dynamics is controlled and managed by the VCC by using the vehicle dynamics software (VDSW).

2.8.2 VDSW Project Loop

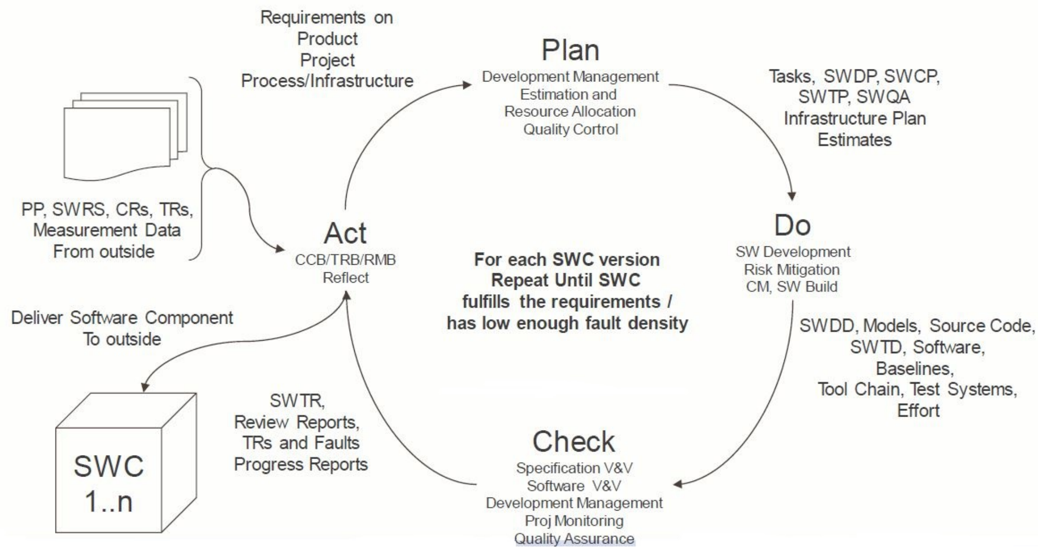


Figure 2.6: Process refinement loop

Volvo Cars apply the PDCA (plan–do–check–act or plan–do–check–adjust) to continuously refine the process based on the new software requirement. Constant improvement of evaluation of the process lets the company have an efficient process.
For any project, the Software requirement specification (SWRS) is very important because it clearly describes what need to be done and delivered in each phase. The SWRS contains requirements on what the software components must do. As described in figure 2.6, the project has input and also requirements on the product from the Software requirement specifications (SWRS), Change requests (CRs), Trouble requests (TRs) and also the measurement data which is not in the range of VDSW software development cycle.
After the inputs are provided to the project loop, the SWRS and measurement data is analyzed using the development plan, Quality assurance and infrastructure plan estimates. This phase is important to walk through the software requirements to make sure everything is going as per the plan.
Assuming the goals are clearly described, the next step is to produce the design and implementation of the VDSW. The design of VDSW is about defining the software architecture of the product. It is a crucial step before the actual development and implementation of the product. This study shows later the impact of using Docker on the architecture of the product. The final step of the project loop is the verification process or checking if everything is done right or not. Here the verification system is divided into two.

- Verification of Specifications: Verify the specification of the software by checking the outcomes.
- Verification of Software: Verify the software using the component, verification tests. This step in the loop can be improved by using Docker. The impact of Docker on a testing process is explained later in this study.

2.8.3 Continuous Integration and GoCD

Every software build phase of the VDSW starts in the GoCD environment and is triggered automatically as soon as any software update happens on the version control system. Once the build is successful on the GoCD environment, the delivery phase is triggered to extract the executable from the build environment and to store it in the artifactory.

As explained previously in section 2.4, the smoke test is mainly used to test the most important parts of the software. The need for the automatic smoke test on the vehicle dynamics software is very important in order to validate the executable on the actual target before the software is considered for the other detailed vehicle testing. It also has a great advantage by achieving efficiency in terms of time and resources for validating several successful builds. As shown in figure 2.7, after the successful triggering of the smoke test, the next step is to fetch the executable and all required test environments from the version control system to perform the more critical testing on the actual target. The test results are then uploaded to the artifactory corresponding to each delivery.

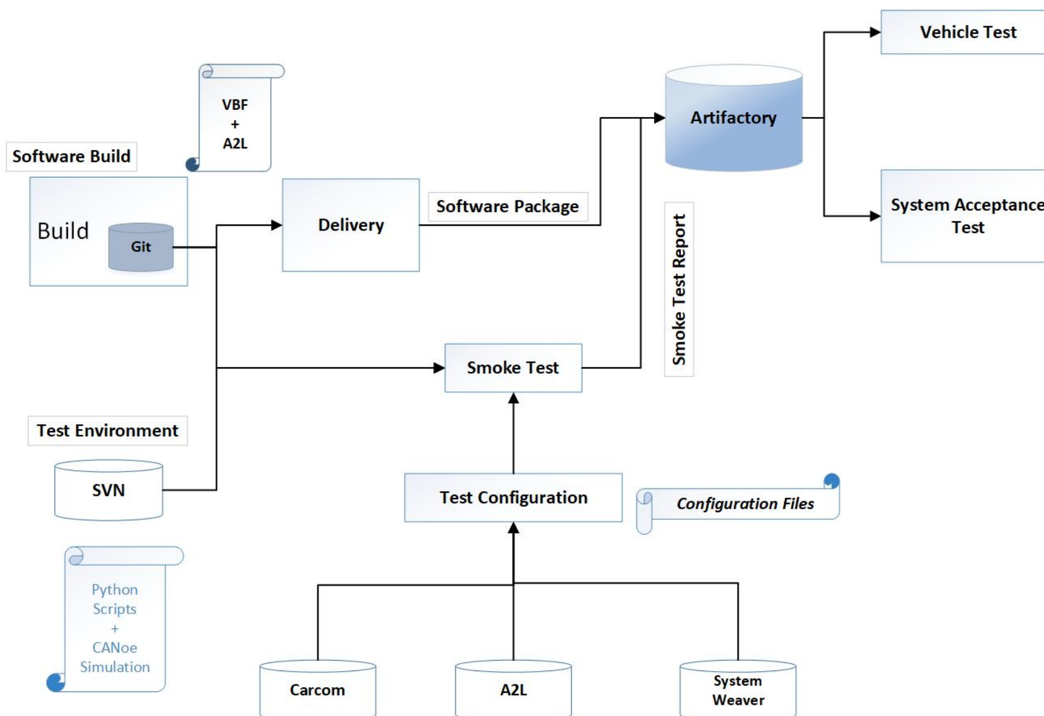


Figure 2.7: VDSW Software build chain

The continuous integration architecture of Volvo cars is automatically triggered whenever there is a new code commit on the shared repository. Most of the time the files that reside on the shared repository have an extension name of VBF (Volvo Binary files). But the extension names may differ from one software provider to another. Once the build is successful, the deliverable is fetched and stored on the artifactory called delivery.

The software build does not only trigger the artifactory delivery to fetch the deliverable. The smoke test is also triggered automatically once there is a successful

software build. But sometimes the Smoke test can also be triggered by using the Python scripts that are already stored on the Subversion (SVN). Then the smoke test runs on the software and the smoke test report is stored on the artifactory.

The test automation scripts created in CANoe CAPL programming language need inputs from multiple place like Systemweaver (for Product requirements related to Ethernet communication), Carcom (for requirements related to Diagnostics), A2L file (for information related to software variables), Software part numbers etc. In order to achieve complete automation without manual edit of test scripts, the concept of configuration file (INI) is developed. The information from multiple location is collected and updated in the configuration file which acts as an interface to the CANoe test scripts so that manual update of the scripts shall be avoided.
The final stage on the VDSW software build chain is to run detailed or critical tests such as vehicle test and system acceptance tests. As mentioned on section 2.4 and section 2.5 the smoke and sanity test will check only the most important parts of the software. After the most important parts are checked the detailed tests take over.

2.9 Software Architecture

A software architecture is a high-level representation of a system, and it is the foundation of software development [18]. Being a preliminary view of the system, it provides detailed information about the components involved in the system and the interactions between those components. Software architects rely on the requirement specifications provided in the previous stage of software development to create a comprehensive architecture.
Staron [3] presents the formal definition of a software architecture as "the high-level structures of a software, the discipline of creating such structures, and the documentation of these structures. These structures are needed to reason about the software system". This definition is taken from Wikipedia. The high-level structures mentioned here refer to the various entities involved in the architectural design. These structures can be:

1. Software components-software subsystems that are logically related to each other. Depending on the project and its programming language, they can be classes in the case of object-oriented projects or modules in the case of non-object-oriented projects. They can also be XML configuration files.
2. Hardware components-components of a computer system on which software runs. They can be either independent or inter-related elements such ECU, sensors or actuators. The communication buses that interconnect those units between each other are also considered hardware components.
3. Functions-tasks or goals that are conducted by one or more software components. The ensemble of the functions is what constitutes the logic of the system as a whole and its subsystems.

Depending on the project, architects may or may not be asked to include the hardware components in the architecture even though these are not part of the software. In that case, the software/hardware relationship is visible in the architecture. The

relationships between the components (associations) can be either one-directional or bi-directional depending on the logic of the system.
This chapter presents the theory of two architectural styles and explains why they were chosen for this study.

2.9.1 The theory of the monolithic and the component-based architectural styles

A Docker image can isolate its content from what is external to the image. Therefore, in designing the architecture of a system that uses Docker, clusters of elements that have the same characteristics need to be defined. For this reason, the component-based architecture is considered as a solution to this approach. On the other side, the monolithic architecture can be used when it is not needed to create containers within the product itself.

The theory behind the monolithic architecture is that the entire system is considered as one large component. The modules inside the large component can communicate and use each other [3]. In the case of Monolithic architecture, if there is a need to change a module within the system the whole system or product need to be changed. Component-based architecture is more flexible as compared to the monolithic architectural style. Components are the main ingredients in this style of architecture. Components are the small pieces that make up the whole system. All component communications must go through a well defined interface [3]. Thus, a component based architecture is made up of different components and the components communicate through a well defined interface.
The following are the main advantages of component-based architecture style as compared with the monolithic:

- Re-usability: The components are designed in a way to be used for different application.
- Encapsulation: Each components can interact each other through the interface. This will let the components hide the state and other characteristics.
- Independence: The components are designed in a way where the components has a minimal dependency.

2.9.2 The current monolithic and component-based architectures of VDSW

This section will discuss about the current monolithic and component-based architecture of vehicular dynamics software.

2.9.2.1 Current VDSW monolithic Architecture

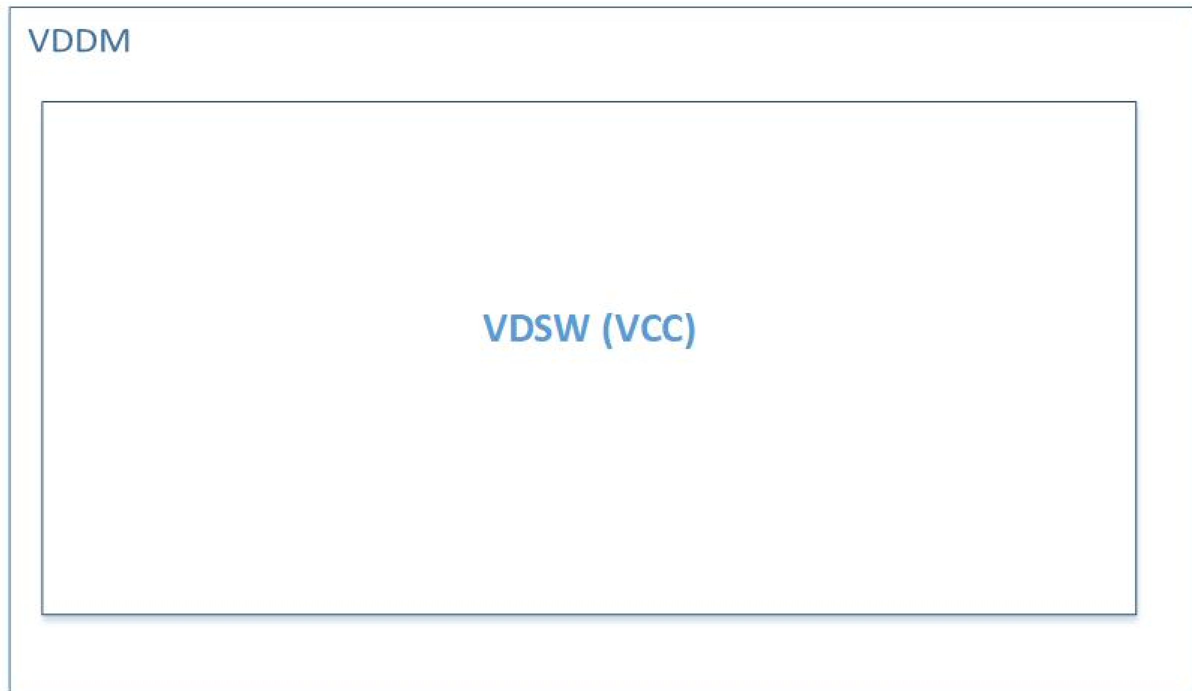


Figure 2.8: Monolithic Architecture of VDSW

2.9.2.2 Current VDSW component-based Architecture

This section will mainly discuss the current software architecture of VDSW in general, the components that are available in the product architecture and also the relationship between each software components.

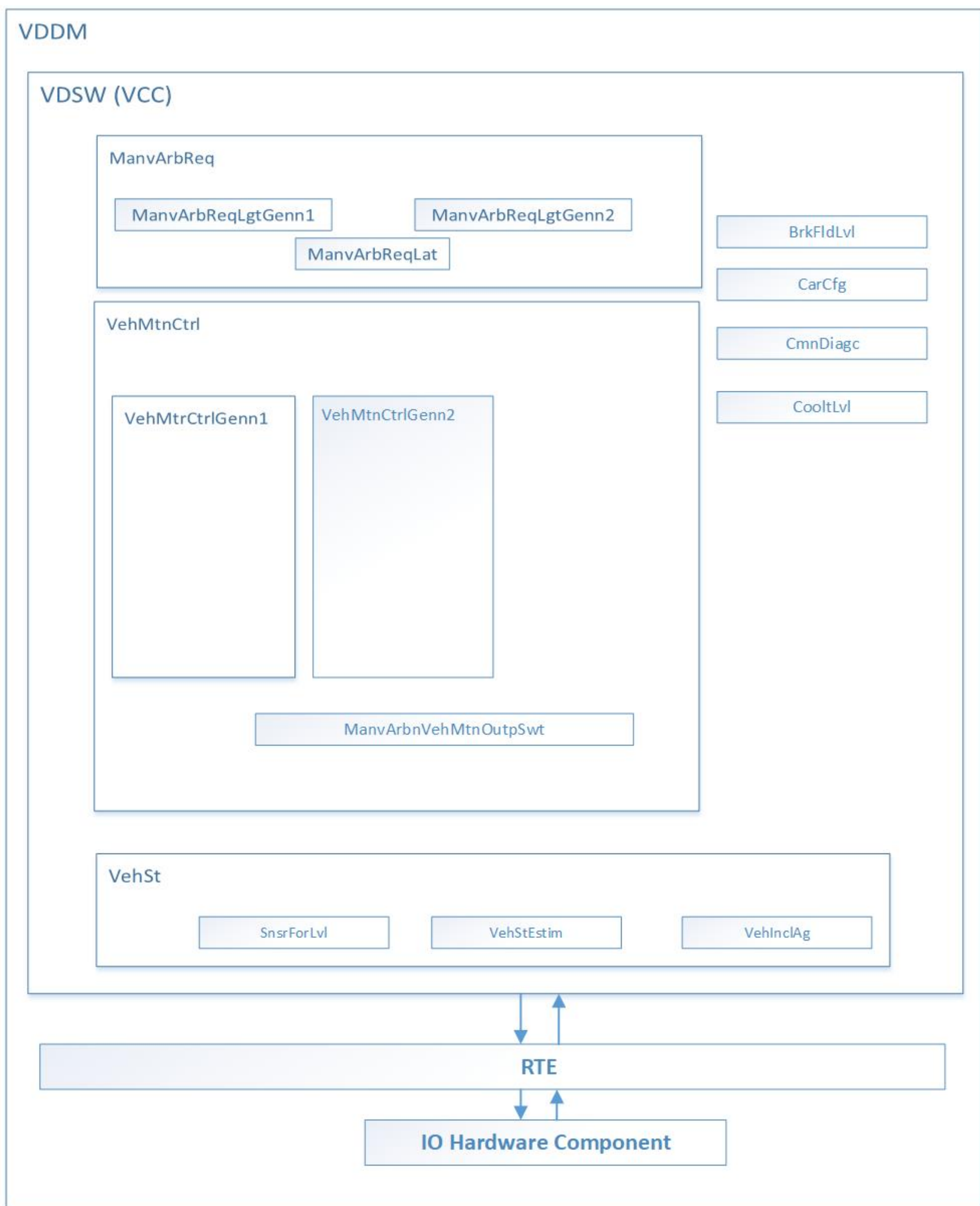


Figure 2.9: The current VDSW Software Architecture

As can be seen from the above Figure 2.9 the Vehicular dynamics software (VDSW) belongs to the parent ECU called the Vehicular dynamics model master (VDDM). The VDSW also has three major components namely the Manoeuvre Arbitration Request (ManArvinReq), Vehicle motion control (VehMtnCtrl) and the Vehicle state (VehSt).

Each software component in the VDSW has a direct connection with the Run time environment (RTE). If there is a need for each software components in the VDSW to interact with another one, the request must go through the RTE. The RTE has also a direct connection with the input output hardware components.
All the software components on VDSW are running on the Linux environment, which allows for Docker to be used.

3
Related Work

This chapter explains in detail the literature related to Continuous Integration. We deem this to be highly important as it is the basis for this project. As explained in the second point of section 1.4, the model proposed in the book by Ståhl [4] does not include Docker. However, it does give a basis on which we build our CI model that shows how Docker can be incorporated in CI. This model is presented later in section 6.2.

Ståhl and Bosch [4] studied the different implementations of Continuous Integration in order to see whether the differences between them can affect the benefits that Continuous Integration presents. In our literature review, we will first look into a number of sources to derive attributes that we can use for our CI Model, then we will construct a model that we see to fit for VCC's current way of CI.
The model that Jan Bosch proposes at the end of his study is comprehensive and includes parts that we will not need: the parts related to the developers.
Bosch's study focuses on the differences in the actual CI process rather than the differences in the tools used in the process. Tools are an important part of Continuous Integration. But Fowler suggests [19] that though they can affect the implementation of CI, CI does not require a specific tool. This study also does not regard contextual factors concerning the project, the business environment and other similar variables.
Bosch discusses 'variation points' and explains how these do not have consequences on the practice of CI as a "whole". They only affect a specific variant of CI. Variation points are defined as those aspects of CI where literature related to CI has differed in defining. They are regarded as important because they can create a variant of CI. Therefore, his model's goal is to unravel those variants in a comprehensive schema. Our method of work is to study those variants and their sources and to filter out the ones that we deem as not relevant to our industrial partner, then to create a CI model where the selected variants are going to define a new instance that is specific to VCC. This model will additionally include the steps related to Docker.

Cluster name	Statements	Unique sources	Contention	Claimed disparity
Build duration	10	9	Yes	Yes
Build frequency	10	8	Yes	No
Build triggering	32	29	Yes	Yes
Build version selection	2	2	No	No
Component dependency versioning	6	3	No	No
Definition of failure and success	8	6	Yes	No
Fault duration	5	5	Yes	Yes
Fault frequency	1	1	No	Yes
Fault handling	9	9	Yes	No
Fault responsibility	6	5	No	No
Integration frequency	7	7	No	Yes
Integration on broken builds	6	6	Yes	No
Integration serialization and batching	6	6	Yes	Yes
Integration target	8	6	Yes	Yes
Life cycle phasing	1	1	No	Yes
Modularization	17	11	Yes	Yes
Pre-integration procedure	16	12	Yes	Yes
Process management	1	1	No	No
Scope	50	40	Yes	No
Status communication	19	16	Yes	Yes
Test separation	11	9	Yes	Yes
Testing of new functionality	10	9	Yes	Yes

Table 3.1: The clusters derived from literature review [2]

In table 3.1, Jan Bosch lists the cluster statements that he found in his literature review. In the following sections we will describe each one of them and we will provide examples.

1. Build duration

The build duration depends on the scope of the build, which results in different build times. According to Downs et al. [20] who studied the use case of an agile team to derive the build time, it takes several minutes from the point of committing code to the repository to getting the build result. On the other hand, when CI was implemented in the context of a robust C4ISR (Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance) distributed system, it took approximately more than an hour to compile and run the tests [21]. Build duration is important because if it is too long, it might not be able to keep up with the commits. Also, the shorter it is, the easier it is for developers to remember what could have caused a test to fail [22]. In order to tackle the build time, some scholars propose to use an incremental feedback [23] by starting with the fast tests, then

moving to the slower ones and separating those two.

2. Test separation
Test separation is a method used to reduce the build time by moving slow tests to a separate schedule and not including them in the regular CI builds [24].

3. Integration frequency
Integration frequency refers to the frequency of performing the steps involved in compiling modules together after the introduction of new code changes in one or many modules. It depends on the number of developers, but it is generally considered normal to have integration occur "every few hours and at least once per day" [25].

4. Build frequency
Build frequency refers to the frequency of compiling and linking the entire product. Some projects trigger a build automatically after some time regardless of whether an integration happened or not. The build frequency might go from "every few hours" [25] to once a day [26].

5. Build triggering
Builds can be triggered either by a time clock or a code commit. There is a mixed approach where builds can be triggered by both code commits and a time span in the same project ([27]; [28]; [20]; [29]). This means that builds are automatically triggered every once in a while with no need of code commits involved. But if a code commit happens, then a build is triggered.

6. Build version selection
There seems to be no difference in CI adopters when it comes to build version selection. A build is always made on the last source code commit, as this is the main concept of CI [30]. Also, if a build fails, then the latest previous successful build is kept instead.

7. Component dependency versioning and modularization
Continuous Integration can be modularized when the product is modularized. This means that it has several atomic components that may or may not have dependencies between each other. According to Park et al., "the source code of each [component] is controlled independently" [31]. But when there is a co-relation between two or more components, CI "can be seen as a [directed acyclic graph], where nodes correspond to package builds and edges correspond to dependencies among packages" [32]. However, when a build happens in the middle of this chain, any further builds that happen after it need to be re-performed. The build that is responsible for the other builds is referred to by Van der Storm as the "topmost component in the dependency hierarchy" [30].

8. Definition of failure and success
A build may consist of several tests. However, whether the entire build is considered failed if only one test fails or if all tests fail is debatable. Ablett [33] and Rasmusson [34] suggest that one test failure is enough for the whole build to fail.

9. Fault duration Fault duration refers to the time span between detecting the fault and the handling of this fault. Some teams set a time limit of 30 minutes to either fix the fault or revert to a previous integration [21]. Others do not set a time limit, but they claim that most the faults were fixed "within an hour" [35].

10. Fault frequency
Fault frequency is the frequency of build failures.

11. Fault handling and fault responsibility
Fault handling and responsibility is related to who fixes the bug when a fault happens. It differs between organizations, and it can be either the responsibility of one developer or a group of developers. It can be regarded as the sole responsibility of the developer who committed the failed build [21] or the responsibility of the entire team [36]. If several faults happen at the same time, there may be prioritizing between fault handling [37].

12. Integration on broken builds When a build fails, there are different views as to whether it is allowed to commit a change before this broken build is fixed or not. Miller [35] for example is against integrating on a broken build, but Holck and Jørgensen [26] state that commits can be made at "any time". It can be arguable whether it is in the benefit of the team to allow commits without fixing the previous problem as it makes it harder to spot where the problem comes from as the number of failed builds increases.
This problem is similar to when the integration frequency is higher than the build frequency. It happens when builds are slow.

13. Integration serialization and batching
When builds are slow, the integration frequency is higher than the build frequency. One way to deal with this is to serialize the changes before committing them to avoid "integration conflicts" [34].

14. Integration target
Integration target is where exactly in the repository do the developers integrate their

code changes. Some, like Stolberg [38], use a single development branch referred to as the "mainline". Others, like Rogers and Owen [25], use several branches, one per team. Each team integrates in their private server before committing into the "mainline" when a build is successful.

15. Life cycle phasing and Process management
Those two aspects regard CI as a whole. Life cycle phasing implies in the practice of CI that it is implemented in the phase of development and tests; mainly for development [26]. Process management on the other hand refers to the "degree of control over continuous integration processes" (Bosch, 2013, source). According to Holck and Jørgensen [26], CI build process management can be either centralized or less structured.

16. Pre-integration procedure
This refers to the actions taken before committing/integrating code.

17. Scope
This is one of the most important attributes of CI that we want to highlight in our model. Scope refers to the activities included in the build. Whether it includes some code compilation, tests, packaging, etc. Sometimes, static and dynamic code analysis can be added to the tests ([31], [39]). The tests conducted in CI can go from unit tests [40] to integration tests [41], acceptance tests [38], or smoke tests which is the case at VCC in our thesis. CI can also include deployment and other steps, which we will look further into with VCC.

18. Status communication
This refers to how build success or failure is communicated to the developers. Bosch mentions different notification methods like dashboards [42] and web pages [30]. Status communication is important in Continuous Integration because it is the backbone of feedback for improving the quality of the product.

19. Testing of new functionality
Regression tests are performed to make sure that any modifications to the code are still working properly. They can test the introduction of new functionality or the fixing of an old bug.

4
Methodology

This chapter will start with the problem foundation of this thesis project and we will explain the research approach to draw a conclusion for the problems formulated in the beginning.

4.1 Problem Investigation and Awareness

The initial problem of the thesis project is the timing issue which is the feedback loop that the industrial partner identified in the beginning.The feedback loop is very important in the testing process. The quality of the product will be improved while the feedback loop is shorter. We choose to integrate docker into the testing process to improve the feedback loop. In doing, there are also other problems that we discover.

- There is no exact possible way to integrate docker inside the continuous integration process. The continuous integration process have multiple components such as GoCD, different private repositories and repositories for storing a docker image. It is very important to know where and how the docker should be integrated.
- The integration of docker with the current VCC CI architecture is also another problem to investigate. Docker is supported by only machines that run Linux or Windows 10 operating systems. We will study the current VCC CI architecture to integrate docker inside the architecture.

In order for us to build a CI model for VCC, we need to build an understanding of the different ways that were adopted in implementing the CI practice in industry. As simple as the Continuous Integration concept seems to be, it can be implemented and interpreted in different ways according to Bosch and Ståhl [4].
This latter points out to the following three main aspects where CI can differ: what the developers integrate with and the procedures involved both before and after integration. The way CI is implemented in those regards is not alike in the industry. For this purpose, we studied *number* resources that each helped us create a comprehensive view of CI when combined. We based our literature review on Bosch's work entitled Modeling Continuous Integration Practice differences in Industry Software Development because he made a fully comprehensive analysis of all the scholarly CI publications [4]. However, we explored those resources and select the ones that are not focused on the testing phase of the CI cycle. In this work, Jan Bosch defines

a number of clusters that each reflect an attribute of CI that creates a different CI interpretation. Some of those attributes are regarding the build process of CI, others are about the development stage and other ones are about the testing stage.

For the scope of this thesis, we are concerned with what comes right after a commit is made from the developers. Therefore, the attributes that concern the developers work before the commit are not going to be included in our analysis nor our proposed model. Also, the fault handling attribute and the procedures that happen after the test is finished will not be included.

4.2 Research Methodology

A research methodology is the systematic way of solving a problem [43]. Every research needs to follow a specific research methodology to collect, analyze and to reach to a conclusion.
We followed the design research methodology to conduct the thesis project. Design science is one of the major research methodologies especially in computer science. Design science is the design and investigation of artifacts in context [2]. We propose a design science research methodology to investigate the use of docker inside current VCC CI architecture and also for designing a new software architecture. The list of activities in the design science research methodology will guide us through the process of designing the new CI architecture.
Design science has a list of activities as observed on table 4.1 that are also very important while we design the new CI architecture. The list of activities will make it easier to examine whether the most important parts of the research are included or not. Here we list and explain the most important stages of the design science research and we give a description according to our research focus.

Stages	Description
Problem identification	Here we explain and motivate the initial research problem. The value of the solution must clearly also be justified. The feedback loop take too much time in the current VCC CI architecture is our main problem.
Define the objectives of a solution	Clearly explain how the identified set of problems will be solved. In addition, the criteria for the solution to solve the problem also be defined.
Design and development	Design and develop the solution for integrating docker inside the current VCC CI architecture. The design also includes the new monolithic and component based product architecture.
Demonstration	The Proof of the proposed solution works by solving a problem. Our demonstration starts with the proof of concept by implementing a simple Java program that use docker. Then demonstrate the proposed solution in the real VCC computing environment.
Evaluation	This stage is very important to exactly know the demonstration meets the measurement criteria that are set in the second stage.
Communication	After the successful completion of the research, there needs to be a way to share the artifacts to the research society. This stage helps the output of the research to be published.

Table 4.1: Stages involved in Design Science methodology [2]

We will follow the engineering stages that are illustrated on table 4.1 for the successful completion of the project.

5
Dockerization

This chapter starts with a description of the scenarios and environment set-up needed to use Docker for testing. Then it presents a CI model that uses Docker to show how this latter can be integrated into a CI chain.

5.1 Testing scenarios and environment setup

The context of this work is Continuous Integration that involves testing. The final goal is to study whether using Docker in the testing process of software will affect the testing itself and the actual software architecture of the software concerned. The first step is to compare the feedback time between using and not using Docker in testing. The reason for this is that we want to see whether Docker improves the feedback time of testing to begin with. Therefore, we implemented a sample app (product) and a sample test, then we compared the testing feedback time in the case that uses no Docker with the case that uses Docker. The first scenario does not use Docker in testing, it is called the baseline scenario. The second scenario uses one Docker image that includes both the app and the test. The third scenario uses Docker differently from the second scenario, it uses separate Docker images for the test, the test environment and the app. The reasoning behind the third scenario was that we expected that using separate containers for each one of the app, the test and the test environment will provide portability for each one of them independently of each other. After implementing those three scenarios, a measurement of the feedback time was conducted over the three scenarios to see how using Docker differently can impact the feedback time. The comparison of those time measurements shows how using Docker images can decrease the feedback time by almost half. The second step after this was to show later in the report (section 6.1) whether using Docker could impact the software architecture of a software product. For every one of those three scenarios, we discuss which software architectures can be used.

Implementation details

- Machine specifications: Every machine used in this study has a vanilla 64-bit Ubuntu 16.04 Long Term Support (LTS) operating system (OS). Version 16.04 LTS was used because it was the latest stable version at the time of the implementation.
- Use-case: Three testers perform three different tests on a sample Java app called Min.java. The Min.java class is responsible for finding the minimum element in an array of integers. Java was chosen due to the experience of the authors with Java language and Java unit testing. The Min.java file content is shown in Appendix A
- Test environment: The test needs the Junit jar file, the Hamcrest jar file and

openjdk to run properly. The jar files contain the Java libraries necessary to write test cases and to assert whether the test succeeds or fails.

- Scenarios:
  1. The testers use Vanilla Ubuntu machines that run the tests natively.
  2. The testers use Vanilla Ubuntu machines that use a single shared Docker container for both the product and the test.
  3. The testers use Vanilla Ubuntu machines that use 3 Docker containers: A Docker container for the product that needs to be tested, a second one for the test environment and a third one for the test.

Scenario 1: Baseline scenario
This scenario does not use containerization. Every tester needs to set up the test environment on their machine before running the test, as shown in figure 5.1. The MinTest.java which is the test file is shown in Appendix B

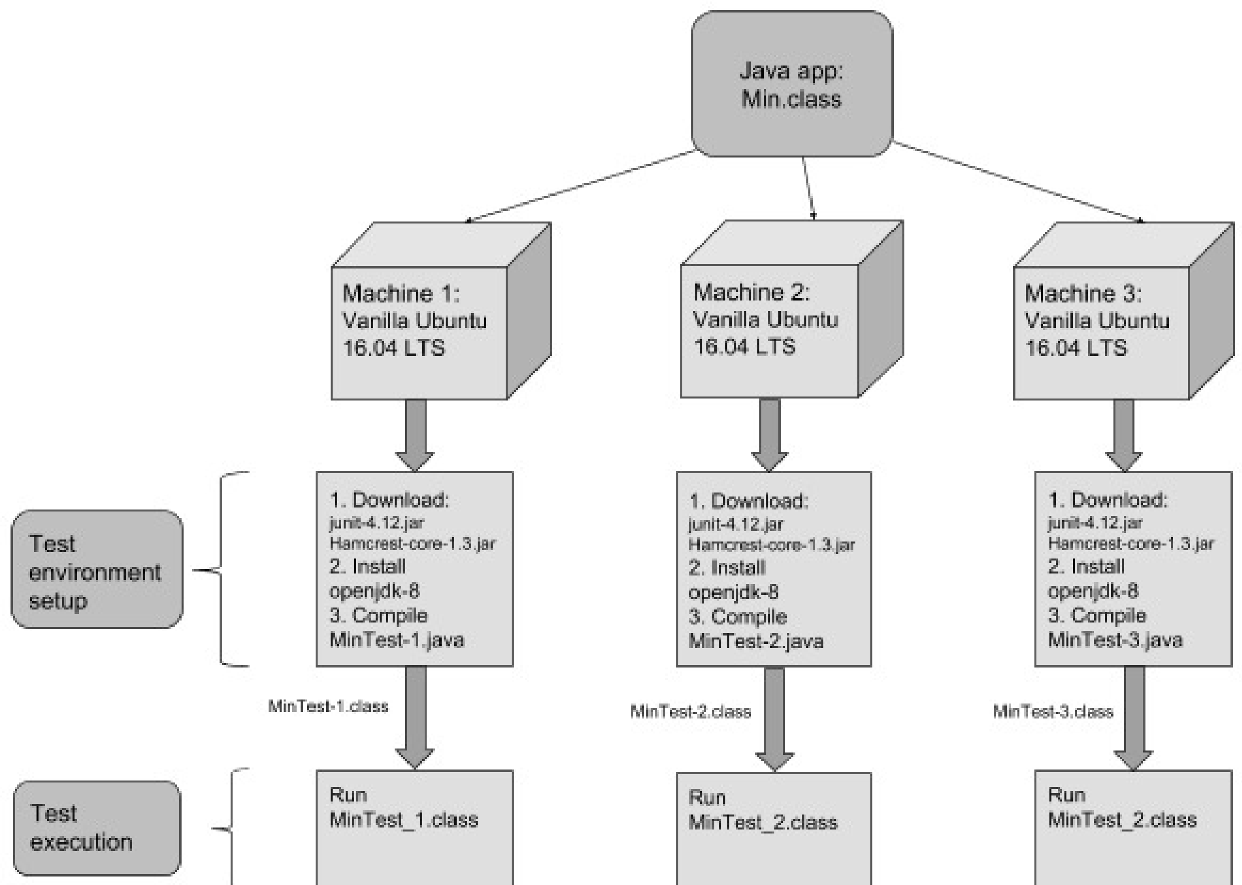


Figure 5.1: Baseline scenario

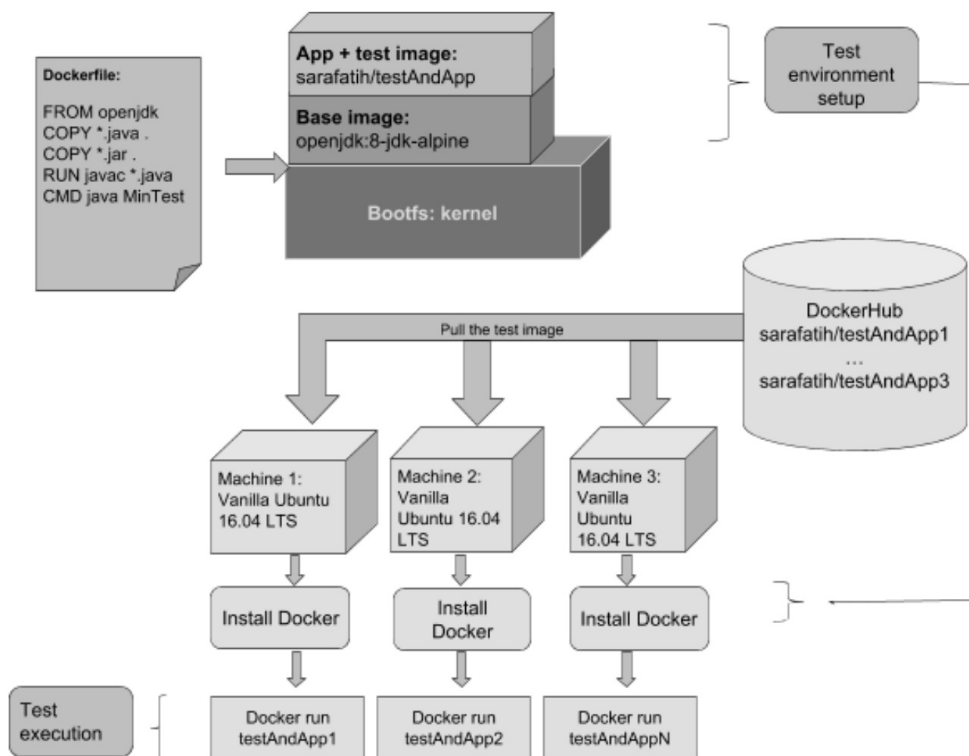


Figure 5.2: Vanilla Ubuntu machines with a single shared Docker container for both the product and the test

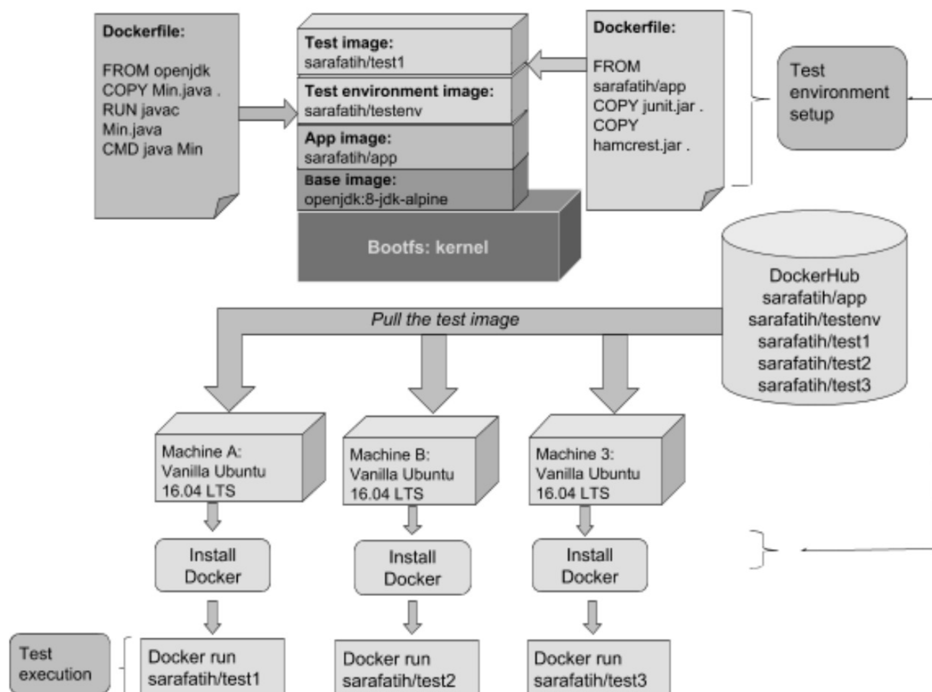


Figure 5.3: Vanilla Ubuntu machines with multiple Docker containers

Scenario 2: Vanilla Ubuntu 16.04 machines with a single shared Docker container for both the product and the test.
As shown in figure 5.2, every tester only needs to run the container to run the test. To run the test using a Docker container, a Docker image was built with a Dockerfile (see Appendix J) and a context containing the files needed for the test. The build context is the location path specified in the build command. When an image is built, it takes with it all the files located in this context using the Dockerfile. The Dockerfile is a script file that specifies the base image and the commands that should run when the image is built and when the container runs. A Docker container can only run one application at a time. Thereafter, when a single Docker image is used for both the product and the test, either the test runs or the application runs.
The Dockerfile was split into two main stages in addition to other ones:

- The Docker image build stage: This stage only does the compiling of the files and copies the needed files for running the container. The copying is done using the command COPY. The other important Dockerfile command is RUN. This command compiles the source files needed for running later.
- The Docker container run stage: In this stage, the test runs by using Dockerfile's CMD command and JUnitCore.

Docker provides a Docker public repository where users can push Docker images and pull them to run them locally. The test is executed when the container runs using `Docker run imageId`.
Since the Docker image contains both the product, the test environment and the test, every tester will have their own Docker image. The steps for every tester are as follows:

- Build the Docker image locally with a context containing both the application code and the test code, Junit and hamcrest jar files, and openjdk-8 as a base image. The image needs to have a tag that describes its version.
- Push the Docker image into DockerHub.
- Run the Docker container to see the result of the test.

Scenario 3: Vanilla Ubuntu 16.04 machines with separate Docker containers for the product, the test environment and the test
In this scenario, the components of the test process are separated into different images and stacked as follows:

- The application image: This image uses openjdk-8 as its base image and contains the product code as the container context. In the RUN command of the Dockerfile, the Min.java file is compiled into Min.class using javac. The output of this compilation (Min.class) is stored in the Docker image file system and it can be used by any other image that is based on this image. The Dockerfile used for this image is shown in Appendix G
- The test environment image: Since the testers need the same test environment, one Docker image was created containing the jar files of those libraries in order to make it reusable by all the testers. The Dockerfile used for this image is shown in Appendix F
- The test image: This Docker image is based on the test dependencies image. It contains only the test source file and during its build stage, it compiles the test file. The Dockerfile used for this image is shown in Appendix E

Every tester only needs to run the container of their test, same as scenario 2. Running the container will automatically pull the image from DockerHub and run the test. Figure 5.3 shows the components of this scenario
The evaluations performed to study the impact of Docker on testing are shown in section 7.3.

5.2 How can Docker be integrated into a Continuous Integration process?

After implementing Docker in a simple testing process, the authors implemented a Continuous Integration chain with Docker for the same product and test used previously. The goal behind this is to provide a full picture of the Continuous Integration process needed to keep quality assurance of the product.

GoCD was used as a CI server that runs the tests automatically whenever a new code push is made to the GitHub repository. To see how containerization works with CI and testing, the authors first implemented a one-pipeline GoCD server, then they implemented a two-pipeline GoCD server to solve the problems encountered with the first implementation. The details are explained in the Figure 5.4 and Figure 5.3.
Scenario 1: Implementation of a GoCD CI chain that uses a single pipeline

In this scenario of CI, there are three major components namely:

- Source code repository: which is git repository and all the source codes that are necessary for the development and test will be stored here.
- Continuous integration server: which is the GoCD and all the operation to pull and push the docker image will be performed here.
- Docker image repository: which is the docker hub or the jFrog. This will be used to store all the docker images that can be used later by the continuous integration server.

The first step in CI chain is a Code commit in the source code repository. The GoCD and the docker hub are configured in a way to be triggered automatically whenever there is a code commit the source code repository. So after the code commit, both the GoCD and the docker hub will be triggered and they will perform tasks that are configured to do.

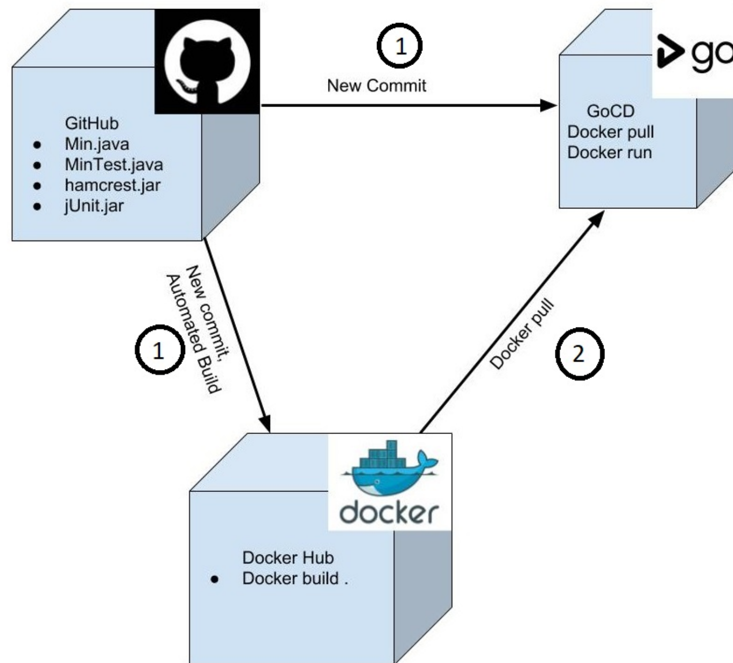


Figure 5.4: Single-pipeline GoCD server

As seen in figure 5.4, After a code commit both the GoCD and the docker are triggered and each perform different taska that are configured to do:

- Docker hub will perform a build of a Docker image.
- GoCD pulls the latest available version of the Docker image from Docker hub and runs it as a container.

But this layout fails in the following two cases:

1. When Dockerhub image build fails, GoCD retrieves the old Docker image from Dockerhub.
2. GoCD takes a shorter time in performing the tasks compared to Dockerhub. While Dockerhub is still building the new image, GoCD is done pulling the old Docker image. This is due to the fact that building a new image takes a longer time than pulling a Docker image.

Further time measurements are conducted and shown later in the evaluation section.

Scenario 2: Implementation of a GoCD CI chain that uses two chained pipelines

In this scenario of CI, the number and types of components that are participated in the CI process are the same with the first scenario. The basic difference of this scenario of CI with the first is the number of pipelines that are included in the continuous integration server. There are two pipelines that perform different tasks which are responsible to perform different tasks.

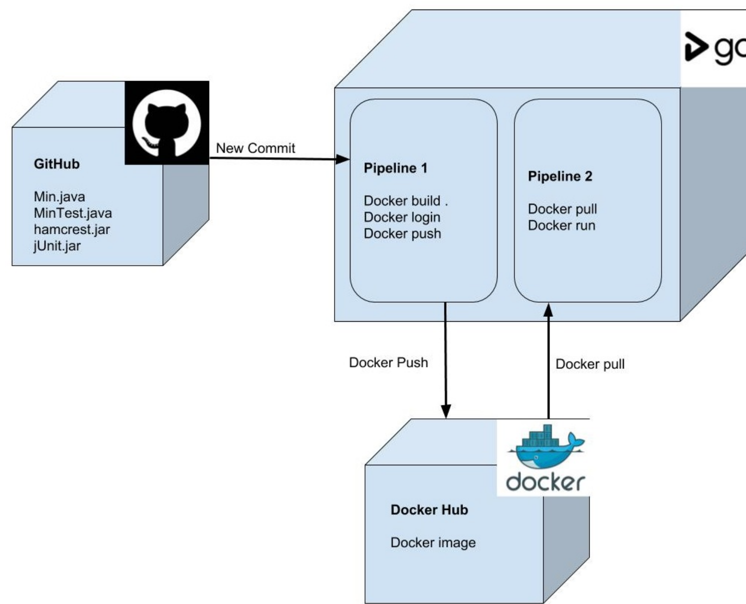


Figure 5.5: Two-pipelines GoCD server

As seen in figure 5.5, When there is a new commit, only GoCD pipelines will be triggered. GoCD has two sequenced pipelines inside it responsible for different tasks:

- The first pipeline is triggered when a new commit is pushed on GitHub. It is responsible for building the Docker image and pushing it to Dockerhub.
- The second pipeline is triggered after successful completion of the first pipeline. It is responsible for pulling the latest image from Dockerhub and running it as a container. The task of Dockerhub is only to store Docker images.

In the previous scenario, GoCD pulled an image from Dockerhub whether the build was successful or not. It also pulled an old Docker image because Dockerhub was not done building the new image yet. But on the second scenario, the first pipeline does the image build then triggers the second pipeline when it is successful. This allows GoCD to pull the latest version of the Docker image.

Overall, the integration of docker have three major components which are the Continuous integration server (GoCD in this case), the docker image repository and the source code repository. As we mentioned on Figure 5.4 having only a single pipeline in the GoCD server will create a synchronization problem between the docker hub and the continuous integration server. In result, the GoCD will pull the old image from docker image repository. On the other hand from the refined CI architecture illustrated in figure **??** the integration of docker with the continuous integration server have the correct synchronization because in the refined CI architecture any source code repository commit does not trigger the whole CI server.

As we can also observe from the evaluations of the continuous integration architecture in section **??** the refined architecture have a correct synchronization between the docker hub and the continuous integration server. The continuous integration server will also pull the latest image.

6
Results

In this chapter, the research questions determined previously are answered in the first part. The second part proposes a solution that maps two different software architectures to four different Docker scenarios.

6.1 Research questions and answers

Research Question 1: How can Docker affect a testing process?

To answer this question, the authors implemented a sample app and a sample test and used Docker in different scenarios. After the scenarios were evaluated using different time criteria, it could be included that using Docker can decrease the feedback loop. However, Docker should be used in the most optimized way when many Docker images are involved in the process.

Using Docker also affects the sequence of operations involved in the testing process and requires the use of Dockerhub for the storage of images. Hence, using Docker requires an additional setup to the native approach.

Docker provides portability to whichever content that it containerizes. This allows for test environment re-usability if the test environment is containerized. It also allows for parallel tests to run on the same module if the module is containerized.

Docker is capable of saving a large amount of time through its caching concept. In the first image build, Docker creates a layer for every element of the container. Future builds of the same image only fetch the new modified layers. This is due to the fact that Docker caches layers.

Research Question 2: How can Docker be integrated into a Continuous integration process?

To answer this question, the authors implemented a Continuous Integration chain using a GoCD server, Dockerhub and a Github repository for code version control. They implemented two different scenarios. The first scenario used only one pipeline in the GoCD server, which caused problems that were spotted by conducting time evaluations on this scenario. These problems were solved by the second implementation that answers this research question.

The layout of the CI process that uses Docker is as follows:

- The code repository allows for code storage. It is connected to the first pipeline.
- The first pipeline is automatically triggered whenever there is a new code commit in the code repository. Upon triggering, it builds the new image that contains the product and the test, then pushes the new image to DockerHub. Only after the new image is successfully built that the second pipeline is triggered

- The second pipeline pulls the new image from Dockerhub then runs the test.

Research Question 3: Is it possible to integrate Docker into VCC's current CI process?

Vehicular dynamics software is written in C, which makes it platform independent. VCC uses multiple agents based on Linux which makes it possible to use Docker. VCC also has access to a private Docker image repository. Therefore, it is possible to integrate Docker in the CI process of VCC, but it stays within the limits of Docker. Programs that work with Windows 7 as well as external python libraries cannot be containerized. Simulated ECUs and hardware cannot be containerized either.

Research Question 4: What are the implications on the software architecture of vehicular dynamics software if Docker is used?

Software architecture is the backbone of software design. Depending on the Docker scenario used, the architecture of vehicular dynamics software needs to change. As explained previously in section 6.3, the two architectures that the authors proposed are the monolithic and the component-based architecture.

As can be seen in table 6.1, both the monolithic and the component-based architecture can be used in the three first scenarios. However, the fourth scenario is the one that requires the software architecture to be component-based. The component-based architecture allows for creating self-contained blocks that interact with other blocks through coupling. Containerizing a component makes it portable and independent for easy testing and continuous integration. If necessary, the other components can be dummy or stubs. Coupling can also be handled by basing images on other images.

The authors propose this mapping for architects as a way to adopt a software architecture depending on the Docker usage in a project.

Scenario	Monolithic Architecture	Component-based architecture
Vanilla Ubuntu machines that run the tests natively	X	X
Vanilla Ubuntu machines that use a single shared Docker container	X	X
Vanilla Ubuntu machines that use 3 Docker images	X	X
Vanilla Ubuntu machines that use one container for each component		X

Table 6.1: Mapping between Docker scenarios and software architectures

6.2 Our proposed Continuous Integration model for VCC

The model presented below is a proposition by the authors for the industrial partner VCC. It is based on the literature review done in section 3. Following our analysis of the aspects and attributes of Continuous Integration in literature, we selected nine aspects for our model. The model we are proposing is suitable for the context of a team of testers whose sole responsibility is to take care of whatever build steps that come after receiving a new software update.
In the next sub-sections, we will describe the elements of our model, then we will present the model. We will also discuss how VCC's current model looks like. We classified the nodes of our model into three: Input, scope/build activities and result handling.

Node 1: Input, pre-integration procedure, and integration target
Input can be source code or a binary file. It is the triggering node for builds. Input usually resides in a version control repository which automatically triggers the build steps as soon as there is a new commit of code change. Input is already existing in the current model of VCC. But we propose the addition of both a pre-integration

procedure and an integration target in the new model for the purpose of containerisation.
The pre-integration procedure shall include the creation and building of Docker images. This would be the initial build that is essential for using the Docker image in the scope activities. Pushing the image into a private repository is also necessary in this step. It requires the configuration of ssh keys and authentication within the image as well.
The integration target is also important because it represents in which container the integration takes place. Depending on the choice of VCC, they can either have all the code in one container or in several containers.

Node 2: Scope and build attributes:
This is the stage that is triggered automatically after the code commit. But the build can also be launched automatically in a scheduled fashion. Whether it is a code commit or a time span is what we call build triggering. The next important attributes are build duration and execution frequency. The build can also either allow batching or not. Other than the attributes are the activities involved in the build, also called as the scope of the build. The scope can include testing (legacy testing or new functionality testing), packaging, and deployment. Legacy testing refers to testing the main product code [4] while new functionality testing is a new feature that is added but that is not complete yet to be included in the main product. Packaging refers to activities that make the product deployable. Pulling Docker images is what we propose as an additional step in the scope. It would make use of the images built during the pre-integration procedure.

Node 3: Result handling:
This is the last stage in Continuous Integration and it includes two elements: result definition and status communication. Result definition is about what makes a build fail. It can be a single test in a series of tests, or a group of tests, or all the tests. Status communication refers to test result notification which is targeted towards the developers.

The proposed Continuous Integration model for the industrial partner VCC:

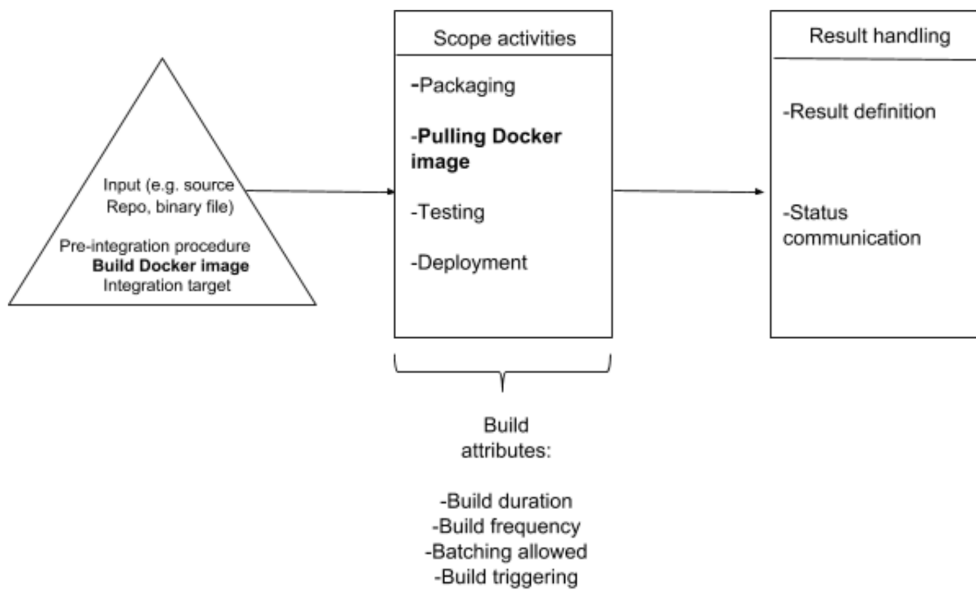


Figure 6.1: Proposed Continuous Integration model for VCC

6.3 Proposed component-based software architecture for vehicular dynamics

As explained in table 6.1, containerizing every software component or cluster in the product requires a component-based software architecture. However, the design of this architecture need not only be based on the logical meaning of a component, but also on the following criteria:

- Components that need to be tested together should be in the same cluster.
- Components that have the same critical level should be clustered to avoid unnecessary advanced testing for elements that are not critical.

Given these criteria, the proposed architecture introduces the following changes:

- Merge of two components for the controller into one component called Vehicular Dynamics Control (VDCtrl). The reason why those two components are merged is that they belong to the same critical level and they require the same tests.
- Merge of the four individual components for vehicular functions into one component called secondary vehicular functions. The reason for this is the same as the first suggestion.

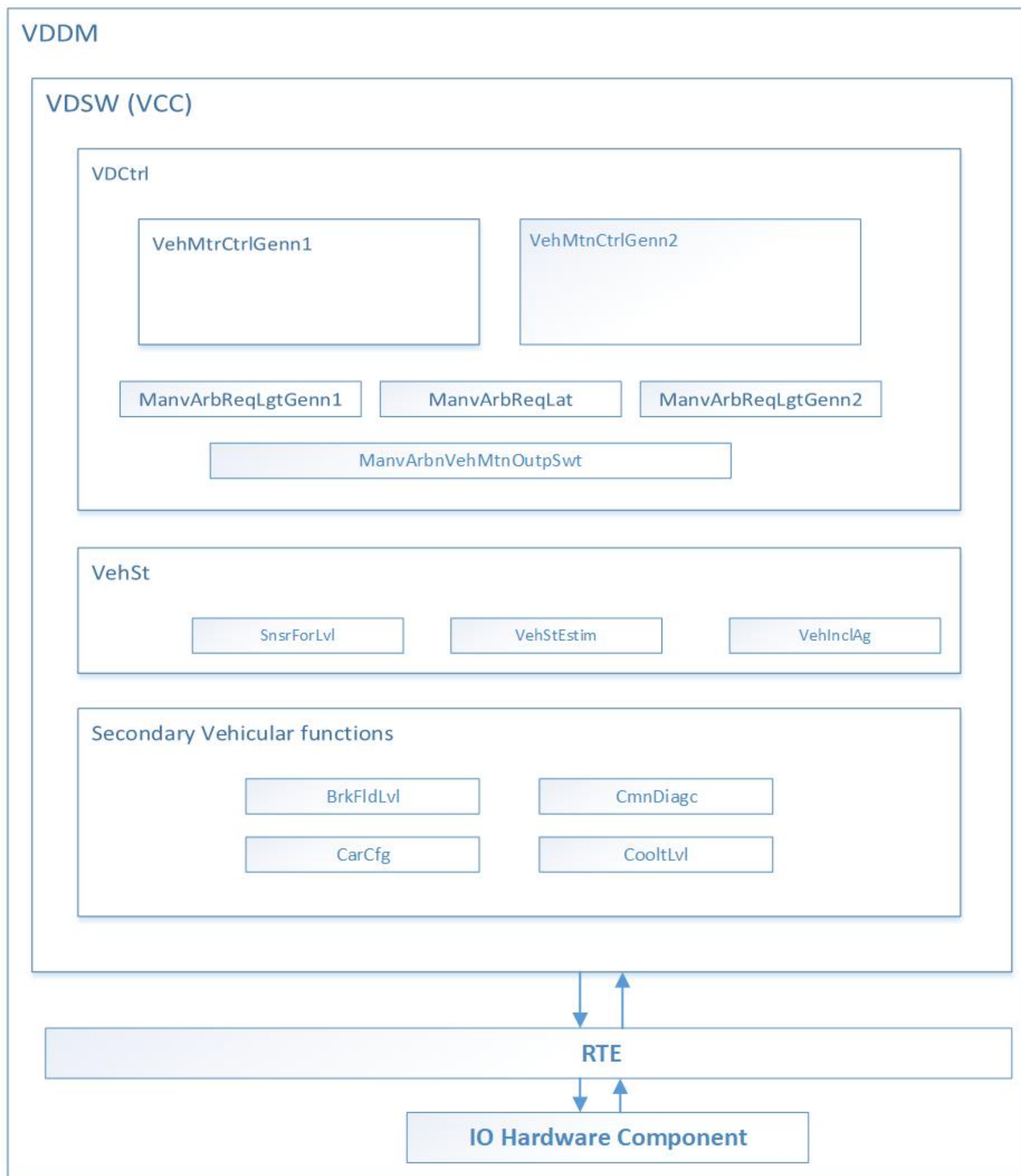


Figure 6.2: Proposed Component-based VDSW architecture

6.4 Implementation of Docker in VCC's Continuous Integration pipeline

In order to understand the benefits of using Docker, the authors built a Docker image, pushed it into an internal private Docker repository of VCC, then pulled it in one of VCC's continuous Integration pipelines. Below, the authors explain both the original baseline pipeline and the proposed Docker pipeline. Both pipelines fetch script files needed to run a test.

1. The baseline pipeline

The baseline pipeline called "fetch-submodules" runs the following command every

time it is triggered:
`git submodule update -init -recursive`

The `submodule` git command is used to fetch all the submodules that are present in the git repository. The `update` command downloads the content of each submodule. The `recursive` command goes through all the submodules.

2. The proposed pipeline
This pipeline makes use of Docker and it is called "fetch-docker". In order to have an advantage over a normal fetch as in the baseline pipeline, the authors created a Docker image that has three layers as follows:

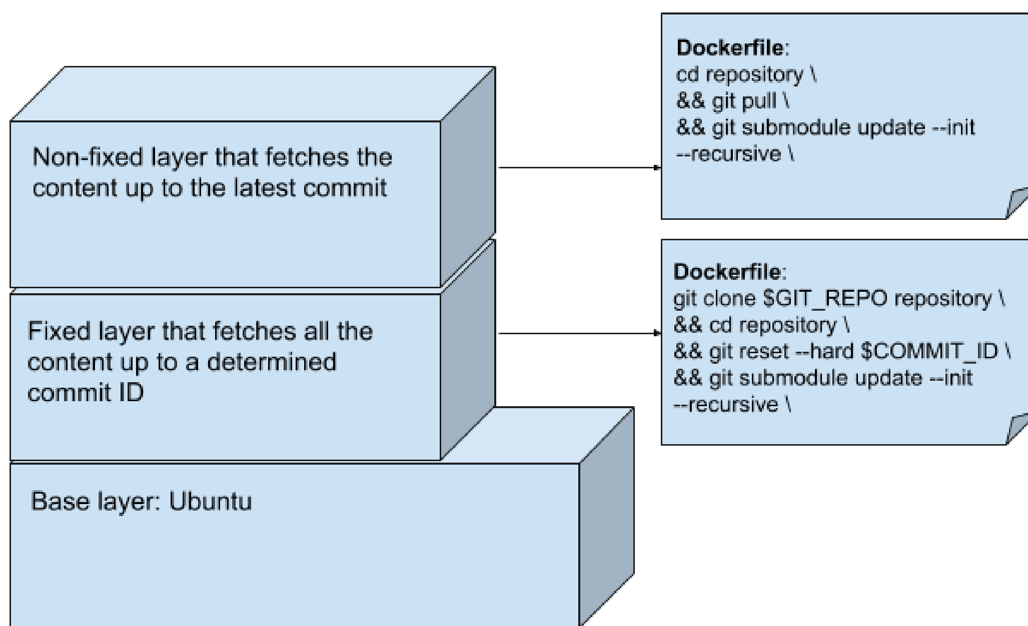


Figure 6.3: Proposed Docker image for VCC's pipeline

The reasoning behind the three layers is that Docker uses caching for every layer. This means that every subsequent build does not need to fetch the contents of a layer if the content of that layer did not change. The first layer, which is called the base layer is only fetched once during the first build. In the subsequent builds, the cache is used. Concerning the second layer, the commit ID that the authors chose for it belongs to one day before the time of the first build. This means that the second layer will always contain all the submodules up until that commit. This is due to the way Git works; the content of a repository at a fixed time (commit ID) does not change. Therefore, any build subsequent to the first build will use the cache of this second layer, in the same way as was done for the first layer. The third layer which pulls the content up until the present time is small because it only fetches the updated or added files that happened after the commit ID specified in the second layer. However, its content changes depending on whether new commits happen. Therefore, it will be fetched every time there is a change. The details

of the Dockerfile used for this image are shown in Appendix I and the content of the Docker compose YAML file used is shown in Appendix H. The Docker compose YAML file was used to define arguments such as the username, the ssh key, the Commit ID as well as the Docker image name.
The only command that the proposed pipeline will run is:
`docker pull DOCKERID`

3. Comparison between the baseline pipeline and the proposed pipeline

The first time that both pipelines run, they perform the same tasks. They both fetch all the content. The only difference is that the Docker image fetches the content in two layers in the proposed pipeline, as opposed to the baseline pipeline.
The second time that the pipelines runs is when there is a difference. The baseline pipeline does exactly what it did the first time after cleaning the directory. On the other hand, the proposed pipeline makes use of the second layer that is cached from the first build. As for the third layer, if there is a new commit since that last build, then the third layer fetches the updated files. If not, then it uses the cached content as well.
In every subsequent triggering, the baseline pipeline does the exact same thing. But the proposed pipeline performs different fetches depending on the commits. The time evaluations performed on the two pipelines are shown in section 7.4

7
Evaluation

In order for us to understand how Docker impacts testing, we implemented three different scenarios that all have the same goal in common. This goal is to fetch the testing environment, then run the first test on the app. After we made modifications to the app, we ran the same test again on the app. The feedback loop is what we measured to see how it changes depending on the usage scenario. The first scenario does not use Docker, we will call it the baseline scenario from now on. The second scenario uses one single Docker container for both the app and the test. We will call it the single container scenario. The third scenario uses three Docker containers: One for the app, one for the test and one for the test environment. We will call it the multi-container scenario. What we expect from our time evaluations is that time will be saved by using Docker. The reasoning behind this is that Docker uses caching. So, whenever we build a Docker image, any future build after modifications will only fetch the modifications. It is similar to how Git works in this sense. Our expectations were met, as will be shown below. The single Docker container scenario took half the time it took for the baseline scenario. However, the multi-container scenario took 2,74 longer than the baseline scenario. A detailed analysis of these results is provided below in section 7.3.4.

7.1 Evaluation Environment

For each docker scenario described below, the time measurement is performed according to the test environment setup mentioned in 5.1. Each scenario has a number of stages. A stage in every scenario may depend on the stage which is executed before it. For example stage two in figure 7.1 below needs the successful download of Libraries which is performed in stage one. For every stage, the time is measured and registered. Some stages such as downloading files and accepting an input from a user were not measured in time because those stages are dependent on external circumstances like Network speed, the speed in which the user inserts an input, etc. For those stages, we denoted the time to complete a stage as delta.

7.2 Measurement criteria and methods

The evaluation uses the measurement methods shown in table 7.1. Each scenario has its own sequence of tasks. The time it takes for each task in every scenario is measured and the total time is the sum of the task times.

Scenario	Measured entity	Description	Measurement Method
Scenario 1	Time to run `script1.sh`	`script1.sh` (appendix C) performs the following tasks: • Install openjdk-8 • Build the product code • Build the test code • Run the test	res1 = date *`commands`* res2= date totalTime=res2 - res1
	Time to run `script2.sh`	`script2.sh` (appendix D) performs the same tasks as `script1.sh`, except installing openjdk8.	res1 = date *`commands`* res2= date totalTime=res2 - res1
Scenario 2	Time to build the a Docker image	This Docker image contains both the product and the test	Linux command `Time` measures the time it takes for a command to execute as follows: `Time docker build .`
	Time to run the Docker container	This runs the test	Linux command `Time` measures the time it takes for a command to execute as follows: `Time docker run containerId`

Scenario 3	Time to build and push the product image	This builds the product code to make it available for testing	Linux command `Time`
	Time to build and push the test environment image	This image contains the needed dependencies for the test in its context	Linux command `Time`
	Time to build and push the test image	This builds the image that contains the test code	Linux command `Time`
	Time to run the Docker container	This runs the test	Linux command `Time`

Table 7.1: Evaluation of different scenarios

The discussion of the time evaluations of those scenarios is shown in both section 8.1 and the first research question of section 6.1.

7.3 Evaluation of the three Docker scenarios

Latency is the main and only requirement specified for this project. It should not take longer to get the test result if Docker is used. Therefore, time was measured for every scenario that used Docker, as well as the scenario that did not make use of Docker.
Below are the motivations behind each scenario and the time lines for each one.

7.3.1 Scenario 1: Testing without Docker

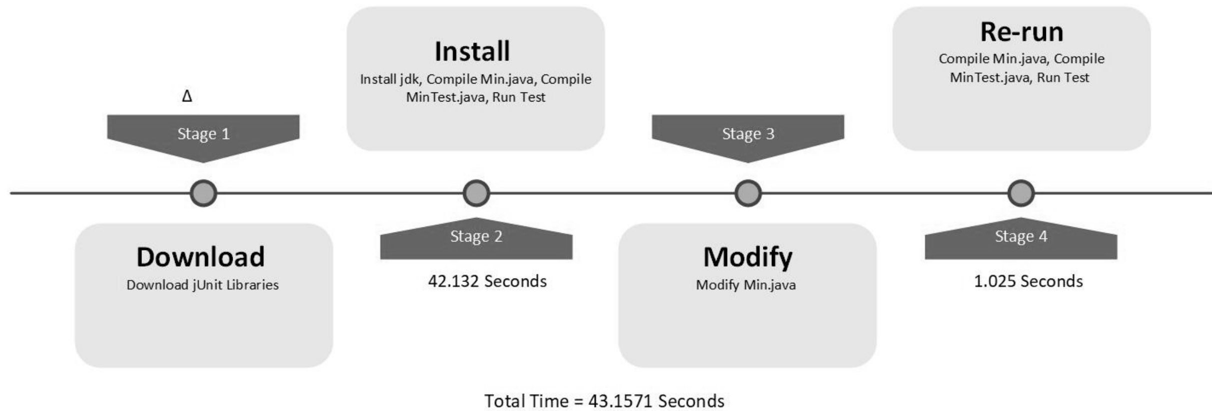


Figure 7.1: Test Environment setup and execution without docker

The above figure 7.1 shows the fist scenario of testing without using Docker. We needed to measure the time that it takes to install the testing environment, then run the test on the app for the first time. Then run the test again after making modifications to the app. The reason why we are running the tests twice is because we are not making use of Continuous Integration which would allow for automatic testing whenever changes are pushed to the repository. The stage where we modify the Min.java file (the app file)is not calculated because modifications of source code depend on circumstances like the developer's own speed of modification, etc. The same goes for this stage in the scenarios described in section 7.3.2 and 7.3.3. The total time it took for the feedback look in the baseline scenario is 43.15 seconds.

7.3.2 Scenario 2: Testing with a single shared Docker image for both the product and the test

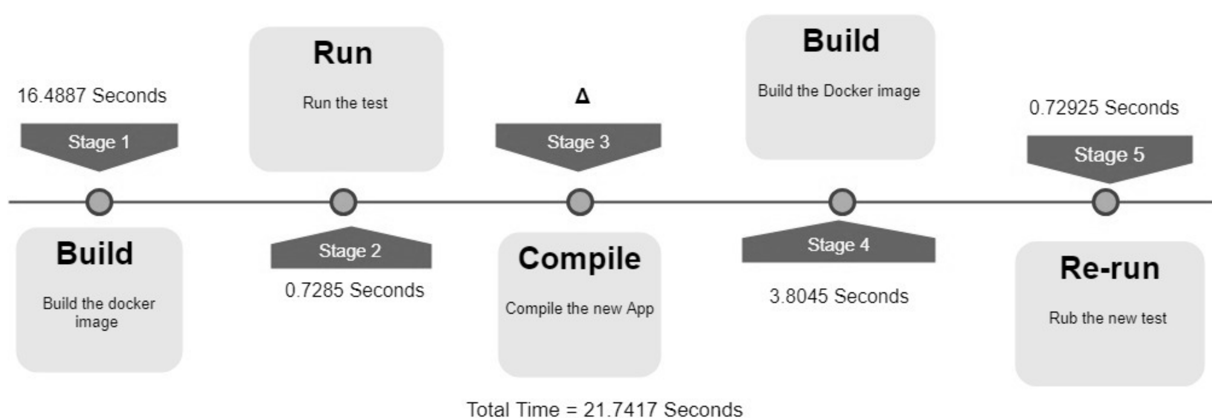


Figure 7.2: Test Environment setup and execution with a single docker image

In this single container scenario, we built a Docker image that contains both the app and the test. Therefore, to run the test, all we needed to do was to build the image and then run the container. The longest time needed was to build the Docker

image the first time (16.48 seconds). But after making modifications to the app, the second build takes only 3.8 seconds due to Docker's caching. Docker only fetches the parts of the image that changed. The rest is cached from the first build. The total time for this scenario was 21.76 seconds, which is about half of the the time it took for the baseline scenario.

7.3.3 Scenario 3: Testing with separate Docker images for the product and the test

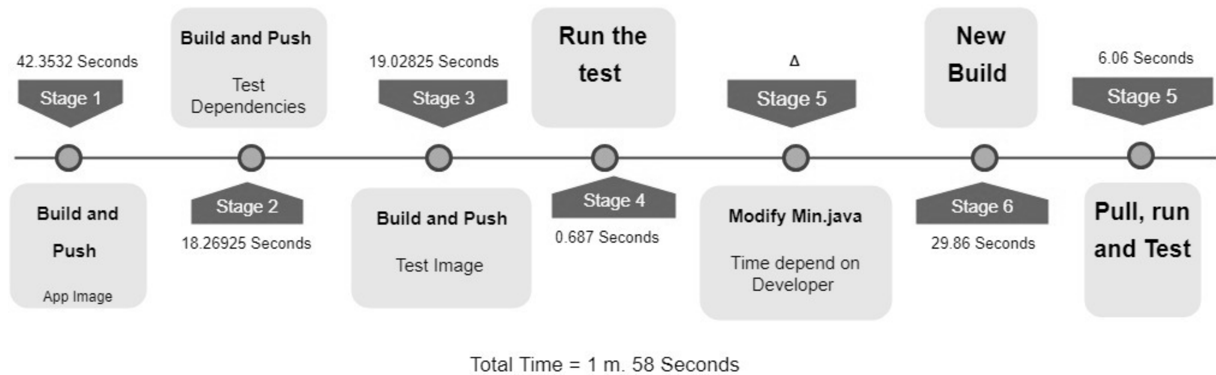


Figure 7.3: Test Environment setup and execution with multiple docker image

In this multi-container scenario, we built a separate image for the test environment, the test and the app. This provides portability for each of them. The test environment can be used by anyone independently of their machine, they just need to run the container for it. This makes it easier to run tests by different people or to run different tests in different machines. The app can also be used independently of the test by having its own container. However, despite this advantage, this scenario took more than double the time it took for the baseline scenario (multi-container-time = 2.74*baseline-time). The reason behind this is that having three different Docker containers means taking three times the time to build the Docker images. So the more Docker containers we have, the more time it takes.

7.3.4 Total time comparison for different scenarios

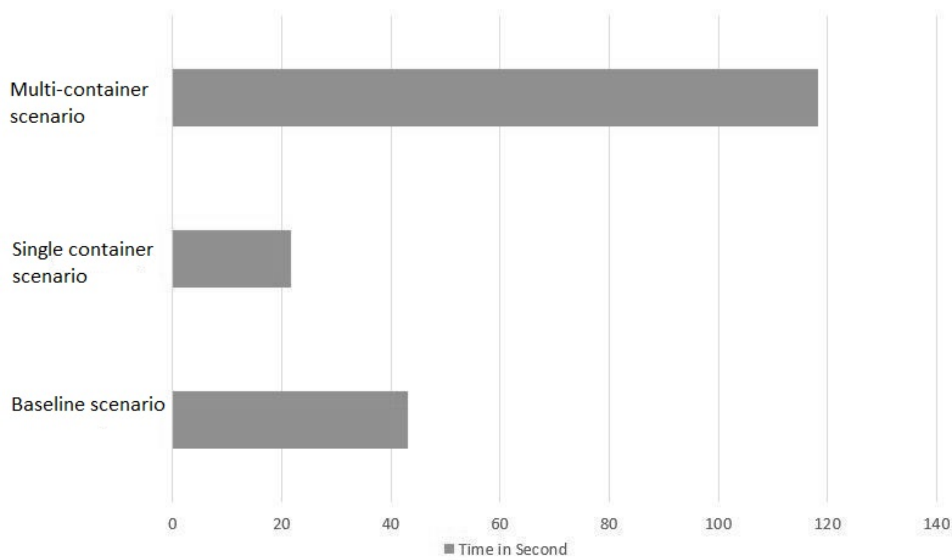


Figure 7.4: Total time comparison for Scenarios

The bar chart on figure 7.4 explains the time comparison for the scenarios that are explained on section 7.3.1, 7.3.2 and 7.3.3. As can be observed from the chart using a single Docker container took the least time(half that of the baseline scenario)

7.4 Evaluation of the baseline pipeline and the proposed pipeline that uses Docker for the industrial partner VCC

As can be seen from table 7.2 below, the proposed solution, that was explained in section 6.4, takes an additional 12 mins in the first iteration to push the Docker image into the VCC private Docker repository. However, it only takes a few seconds in the following iterations to build and push the Docker image. In the first iteration, all Docker related tasks are performed without any cache, that is why it takes a long time to build. In the second iteration, all Docker related tasks are performed with cache, which means that the Docker image does not fetch any new content. In the third iteration, the upper layer of the Docker image is fetched again because a new commit happened. However, the layer under it was not fetched again because it did not change. Overall, the time difference between the proposed solution and the baseline pipeline is noticeable due to the advantages provided by Docker's caching as well as the multi-layering used (two layers here).

	With Docker			**Without Docker**
Iterations	Build Docker image	Push image into Docker repository	Pull Docker image into Gocd server	Fetch the data from the repository
First iteration	15m16.612s	12m2.658s	4m 21.238s	17 m 3.769 s
Second iteration	0m1.310s	0m3.083s	3s 39	7m 8.74s
Third iteration	0m20.799s	0m5.763s	2s 33	7m8.21s

Table 7.2: Time measurements for the baseline and proposed pipelines

8 Conclusion

8.1 Discussion of Docker's impact on testing

Docker has the advantage of caching when building the image. This is due to Docker's use of layers in building the image. It gives a layer to every component or file that it adds to the image and every layer has its own hash. Future builds check the hash of every layer for modifications. If the layer is not modified, the new build uses the previous cache of the layer. But if it is modified, the new build replaces the old layer with the new one. This means that further updates to either one of the files of the context and building a new image will take less time after the first time, depending on the changes.

Another advantage with using Docker is that it provides portability for the test environment. Any tester can just pull the Docker image of the test from DockerHub and run the container to run the test without having to setup the environment locally, as shown in both figure 5.2 and 5.3. The test becomes machine-independent when using Docker because all its dependencies are stored within the image.

One issue with using Docker and Dockerhub is that the repository does not store the context of every image that is pushed into it. Pulling an image to run a test does not show the associated context with that image. This complicates the task of version controlling which application code causes which test failure. This calls for using an SVN like Github, but it would still need some work from the tester to map the images in the repository with commits in Github. This problem can be solved by using the automated build feature of DockerHub. DockerHub shows the commit message that triggered the image build. This makes it easier for the tester to pull a certain image that corresponds to a test.

Concerning the single Docker container scenario and the multi-container scenario presented in Chapter 5, there is a difference in portability between the two. The second scenario provides the test portability but it does not provide the product portability. This is due to the fact that when a container runs, it can run only one application. Scenario 3 uses separate Docker images for the application and the test, which provides portability for both the product and the test and enables them to run independently.

Another difference concerns version control. In the single container scenario, version control of code and images is not obvious when both the test code and the application code are in the same container. This problem can be solved by the multi-container scenario. In this scenario, every test image is based on a specific app image that is characterized with a version tag. Thereafter, the tester can see which version of the product code corresponds to a certain test image. The advantage of stacking the

images in scenario 3 does not only reside in versioning the application and its test.

8.2 Discussion of Docker's impact on software architecture

As was discussed in table 6.1, Docker can affect the software architecture of a software if there is a need for several Docker containers for the components of the software. If a software has a monolithic architecture, then using one Docker container for the entire software is the most fitting. However, when there is a need for a special Docker container for each component, then the software needs to be designed in a component-based architecture. This entails using an orchestration service like Kubernetes. Future work regarding our thesis could be implementing a component-based software and studying how that affects the testing of the system and the feedback loop of the system.

8.3 Conclusion

One of the learnings of this thesis work is that using Docker can make testing and software development more convenient and machine-independent. It takes off the hassle of setting up test environments on every machine and looking into the system properties. Entire dependencies can be encapsulated in containers and can be used for different purposes. In addition to portability, Docker can save time in testing. This was seen both in the single container scenario implemented and the pipeline implemented for the industrial partner. Industries that rely heavily on testing can benefit of Docker to decrease the time it takes to fetch test dependencies every time tests need to executed. However, it was concluded also from our implementations that increasing the number of Docker containers increases the time it takes to build the images. It also requires setting up orchestration for the different containers. Therefore, these points need to be taken into consideration before using Docker.

This report studied several problems. The first one was about figuring out how Docker is used for testing, and how it affects it. To answer that, the authors implemented different Docker scenarios and evaluated them. Another problem was to understand how Docker can be integrated into a Continuous Integration process. For that, the authors used a sample code repository, a Docker image repository and a CI server. Then, the authors studied how it is possible to integrate Docker into the industrial partner's current CI process. That had some limitations because Docker works only on certain Operating Systems. So it is not possible to containerize all the software components. The industrial partner also relies on many physical components that cannot be containerized. The last problem that this report studied was about the effects of Docker on software architecture. If several Docker containers are to be used for different components, then the component-based architecture is necessary and it would need orchestration to manage the containers.

Our work contributed to the field of Docker and testing by providing an evaluation of different Docker usage scenarios using different time measurement criteria to see the effects of Docker on the feedback loop of testing. We also provided a proposal of a Continuous Integration infrastructure that is based on Docker for our industrial partner. A general infrastructure that can be used by different industries was also provided. We then proposed a mapping between software architecture and Docker

usage scenarios. The two architectures that were investigated on are the monolithic and the component-based architectures. The last part of the contributions was two software architectures for vehicular dynamics software that we developed by working alongside the team of developers that is dedicated for this software.
The findings of this study can help industries that need to decrease the feedback time of their Continuous Integration process. A lot of industries rely on a fast feedback time and have tests that take hours to finish. The Continuous integration model that we proposed can be developed further to suit a specific industry, but it can still serve as a starting point. Also, this study can serve as the ground or starting point to future studies regarding orchestration of different Docker containers when dealing with component-based software architectures.

Bibliography

[1] G. Jombo, Y. Zhang, J. D. Griffiths, and T. Latimer, "Towards an automated system for industrial gas turbine acceptance testing," in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, Oct 2017, pp. 4746–4751.

[2] G. L. Geerts, "A design science research methodology and its application to accounting information systems research," *International Journal of Accounting Information Systems*, vol. 12, no. 2, pp. 142 – 151, 2011, special Issue on Methodologies in AIS Research. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1467089511000200

[3] M. Staron, "Automotive Software Architectures: An Introduction," June 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0747563216308317

[4] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *J. Syst. Softw.*, vol. 87, pp. 48–59, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.08.032

[5] P. Raj, J. S. Chelladhurai, and V. Singh, *Learning Docker*. Packt Publishing, 2015.

[6] R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, April 2016, pp. 1204–1210.

[7] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 171–172.

[8] H. Kazemi, H. N. Mahjoub, A. Tahmasbi-Sarvestani, and Y. Fallah, "A learning-based stochastic mpc design for cooperative adaptive cruise control to handle interfering vehicles," *IEEE Transactions on Intelligent Vehicles*, pp. 1–1, 2018.

[9] "Smoke and Sanity Testing - Introduction and Differences," http://www.guru99.com/smoke-sanity-testing.html\T1\textquotedblrightaccessedon12/08/2013., accessed: 2010-09-30.

[10] A. Deuter, "Slicing the v-model – reduced effort, higher flexibility," in *2013 IEEE 8th International Conference on Global Software Engineering*, Aug 2013, pp. 1–10.

[11] R. H. Rosero, O. S. Gómez, and G. Rodríguez, "An approach for regression

testing of database applications in incremental development settings," in *2017 6th International Conference on Software Process Improvement (CIMPS)*, Oct 2017, pp. 1–4.

[12] S. M. P. M. Duvall and A. Glover. (2007) Continuous Integration: Improving Software Quality and Reducing Risk.

[13] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[14] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 123, pp. 176 – 189, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215001430

[15] E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery—a systematic literature review," *Information and Software Technology*, vol. 82, pp. 55 – 79, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916302324

[16] I. Weber, S. Nepal, and L. Zhu, "Developing dependable and secure cloud applications," *IEEE Internet Computing*, vol. 20, no. 3, pp. 74–79, May 2016.

[17] S. Yang, Y. Lu, and S. Li, "An overview on vehicle dynamics," *International Journal of Dynamics and Control*, vol. 1, no. 4, pp. 385–395, Dec 2013. [Online]. Available: https://doi.org/10.1007/s40435-013-0032-y

[18] K. Qian, X. Fu, L. Tao, C.-w. Xu, and J. Diaz-Herrera, *Software Architecture and Design Illuminated*, 1st ed. USA: Jones and Bartlett Publishers, Inc., 2009.

[19] M. Fowler, "Continuous integration," 2013. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[20] J. Downs, J. Hosking, and B. Plimmer, "Status communication in agile software teams: A case study," in *2010 Fifth International Conference on Software Engineering Advances*, Aug 2010, pp. 82–87.

[21] H. M. Yuksel, E. Tuzun, E. Gelirli, E. Biyikli, and B. Baykal, "Using continuous integration and automated test techniques for a robust c4isr system," in *2009 24th International Symposium on Computer and Information Sciences*, Sept 2009, pp. 743–748.

[22] S. Dösinger, R. Mordinyi, and S. Biffl, "Communicating continuous integration servers for increasing effectiveness of automated testing," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Sept 2012, pp. 374–377.

[23] M. Roberts, "Enterprise continuous integration using binary dependencies," in *Extreme Programming and Agile Processes in Software Engineering*, J. Eckstein and H. Baumeister, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 194–201.

[24] A. Holmes and M. Kellogg, "Automating functional tests using selenium," in *Proceedings of the Conference on AGILE 2006*, ser. AGILE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 270–275. [Online]. Available: https://doi.org/10.1109/AGILE.2006.19

[25] R. O. Rogers, "Cruisecontrol.net: Continuous integration for .net," in *Extreme*

Programming and Agile Processes in Software Engineering, M. Marchesi and G. Succi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 114–122.

[26] J. Holck and N. Jørgensen, "Continuous integration and quality assurance: a case study of two open source projects," *Australasian Journal of Information Systems*, vol. 11, no. 1, 2007.

[27] P. Tingling and A. Saeed, "Extreme programming in action: A longitudinal case study," in *Human-Computer Interaction. Interaction Design and Usability*, J. A. Jacko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 242–251.

[28] B. Hoffman, D. Cole, and J. Vines, "Software process for rapid development of hpc software using cmake," in *2009 DoD High Performance Computing Modernization Program Users Group Conference*, June 2009, pp. 378–382.

[29] C. Woskowski, "Applying industrial-strength testing techniques to critical care medical equipment," in *Computer Safety, Reliability, and Security*, F. Ortmeier and P. Daniel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 62–73.

[30] T. van der Storm, "Backtracking incremental continuous integration," in *2008 12th European Conference on Software Maintenance and Reengineering*, April 2008, pp. 233–242.

[31] S. Kim, S. Park, J. Yun, and Y. Lee, "Automated continuous integration of component-based software: An industrial experience," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sept 2008, pp. 423–426.

[32] O. Beaumont, N. Bonichon, L. Courtès, E. Dolstra, and X. Hanin, "Mixed data-parallel scheduling for distributed continuous integration," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 91–98.

[33] R. Ablett, E. Sharlin, F. Maurer, J. Denzinger, and C. Schock, "Buildbot: Robotic monitoring of agile software development teams," in *RO-MAN 2007 - The 16th IEEE International Symposium on Robot and Human Interactive Communication*, Aug 2007, pp. 931–936.

[34] J.Rasumsson, "Long build trouble shooting guide. extreme programming andagile methods," 2004, pp. 557–574.

[35] A. Miller, "A hundred days of continuous integration," in *Agile 2008 Conference*, Aug 2008, pp. 289–293.

[36] A. Janus, R. Dumke, A. Schmietendorf, and J. Jäger, "The 3c approach for agile quality assurance," in *2012 3rd International Workshop on Emerging Trends in Software Metrics (WETSoM)*, June 2012, pp. 9–13.

[37] R. O. Rogers, "Scaling continuous integration," in *Extreme Programming and Agile Processes in Software Engineering*, J. Eckstein and H. Baumeister, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 68–76.

[38] S. Stolberg, "Enabling agile testing through continuous integration," in *2009 Agile Conference*, Aug 2009, pp. 369–374.

[39] F. Cannizzo, R. Clutton, and R. Ramesh, "Pushing the boundaries of testing and continuous integration," in *Agile 2008 Conference*, Aug 2008, pp. 501–505.

[40] J. Bowyer and J. Hughes, "Assessing undergraduate experience of continuous

integration and test-driven development," vol. 2006, 01 2006, pp. 691–694.

[41] W. M. Watanabe, R. P. M. Fortes, and A. L. Dias, "Using acceptance tests to validate accessibility requirements in ria," in *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*, ser. W4A '12. New York, NY, USA: ACM, 2012, pp. 15:1–15:10. [Online]. Available: http://doi.acm.org/10.1145/2207016.2207022

[42] J. Baumeister and J. Reutelshoefer, "Developing knowledge systems with continuous integration," in *Proceedings of the 11th International Conference on Knowledge Management and Knowledge Technologies*, ser. i-KNOW '11. New York, NY, USA: ACM, 2011, pp. 33:1–33:4. [Online]. Available: http://doi.acm.org/10.1145/2024288.2024328

[43] R. Crepon, "Application of design research methodology to a contex-sensitive study in engineering education," in *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, Oct 2014, pp. 1–4.

A

Appendix 1: Min.java

```
public class Min {
   public static int find_min(int[] a) {
      int x, i;
      x = a[0];
      for(i = 1; i < a.length; i++){
         if(a[i] < x)
            x = a[i];
      }

      return x;

   }
   public static void main(String args[]) {
      int[] intArray = new int[] {4,5,6,7,8};
      int res = Min.find_min(intArray);
      System.out.println("The minimum is: " + res);
   }


}
```

B

Appendix 2: MinTest.java

```java
import static org.junit.Assert.assertTrue;
import org.junit.Test;
public class MinTest {

      @Test public void test_find_min() {
         int[] a = {5,1,7};
         int res = Min.find_min(a);
         assertTrue(res == 1);
      }

}
```

C

Appendix 3: script1.sh

```
#!/bin/bash
res1=$(date +%s.%N)

sudo apt-get install openjdk-8-jdk
javac Min.java
javac -cp .:junit-4.12.jar:hamcrest-core-1.3.jar MinTest.java
java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar
   org.junit.runner.JUnitCore MinTest

res2=$(date +%s.%N)
dt=$(echo "$res2 - $res1" | bc)
dd=$(echo "$dt/86400" | bc)
dt2=$(echo "$dt-86400*$dd" | bc)
dh=$(echo "$dt2/3600" | bc)
dt3=$(echo "$dt2-3600*$dh" | bc)
dm=$(echo "$dt3/60" | bc)
ds=$(echo "$dt3-60*$dm" | bc)

printf "Total runtime: %d:%02d:%02d:%02.4f\n" $dd $dh $dm $ds
```

D

Appendix 4: script2.sh

```
#!/bin/bash
res1=$(date +%s.%N)

javac Min.java
javac -cp .:junit-4.12.jar:hamcrest-core-1.3.jar MinTest.java
java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar
   org.junit.runner.JUnitCore MinTest

res2=$(date +%s.%N)
dt=$(echo "$res2 - $res1" | bc)
dd=$(echo "$dt/86400" | bc)
dt2=$(echo "$dt-86400*$dd" | bc)
dh=$(echo "$dt2/3600" | bc)
dt3=$(echo "$dt2-3600*$dh" | bc)
dm=$(echo "$dt3/60" | bc)
ds=$(echo "$dt3-60*$dm" | bc)

printf "Total runtime: %d:%02d:%02d:%02.4f\n" $dd $dh $dm $ds
```

E

Appendix 5: Test image Dockerfile

```
FROM sarafatih/app:latest

COPY MinTest.java .
COPY junit-4.12.jar .
COPY hamcrest-core-1.3.jar .

RUN javac -cp .:junit-4.12.jar:hamcrest-core-1.3.jar MinTest.java
CMD java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar
    org.junit.runner.JUnitCore Mintest
```

F

Appendix 6: Test dependencies Dockerfile

```
FROM sarafatih/app:latest

COPY junit-4.12.jar .
COPY hamcrest-core-1.3.jar .
```

G

Appendix 7: App image Dockerfile

```
FROM openjdk:8-jdk-alpine

COPY Min.java .

RUN javac Min.java
CMD java Min
```

H

Appendix 8: docker-compose.yml for proposed Docker solution

```
version: "2"

services:
  docker-thesis:
    image:
        swf1.artifactory.cm.volvocars.biz:5013/docker-thesis-image:layers
    build:
      context: .
      args:
        -
           GIT_REPO=ssh://haddock.got.volvocars.net:29418/arccore_delivery.git
        - SSH_USERNAME=hadbuild
        - SSH_KEY_URL=http://10.0.2.15:1337/id_rsa
        - COMMIT_ID=b9761f8300c5b6f5f9041833262cd52625e35f34
```

I

Appendix 9: Dockerfile for proposed Docker solution

```
FROM ubuntu

RUN apt-get update -y && apt-get upgrade -y
RUN apt-get install -y git wget

RUN mkdir -p ~/.ssh

ENV GIT_SSH_COMMAND="ssh -o UserKnownHostsFile=/dev/null -o
   StrictHostKeyChecking=no"

ARG SSH_USERNAME
RUN printf 'Host *\n User 'SSH_USERNAME > ~/.ssh/config

ARG GIT_REPO
ARG SSH_KEY_URL
ARG COMMIT_ID

RUN wget -O ~/.ssh/id_rsa SSH_KEY_URL && chmod 600 ~/.ssh/id_rsa \
&& git clone GIT_REPO repository \
&& cd repository \
&& git reset --hard COMMIT_ID \
&& git submodule update --init --recursive \
&& rm ~/.ssh/id_rsa

RUN wget -O ~/.ssh/id_rsa SSH_KEY_URL && chmod 600 ~/.ssh/id_rsa \
&& cd repository \
&& git pull \
&& git submodule update --init --recursive \
&& rm ~/.ssh/id_rsa
```

J

Appendix 10: Scenario 2 Dockerfile

```
FROM openjdk
COPY *.java .
COPY *.jar .
RUN javac *.java
CMD java MinTest
```