University of Tartu

Faculty of Science and Technology

Institute of Technology

Dzvezdana Arsovska

Building an Efficient and Secure Software Supply Pipeline for Aerial Robotics Application

Master's Thesis (30 ECTS)

Robotics and Computer Engineering

Supervisors:

Assoc. Prof. Karl Kruusamäe

Illia Sheremet, MSc

Tartu, Estonia, 2018

Content

List of Figures v
List of Tables vii
Abbreviations viii
Abstract ix
Resümee ix
Acknowledgments x
1 Introduction 1
1.1 Practical Work and Objectives 2
1.2 Requirements and limitations 3
1.3 Previous Work 3
1.4 Thesis Outline 4
2 Wind Turbine Inspection 6
2.1 Current State of the Wind Energy Industry 6
2.2 Wind Turbine Inspection Solutions 9
2.2.1 Manual Rope Inspection 9
2.2.2 Ground-Based Camera Inspection 10
2.2.3 UAV Inspection 11
3 UAV System Architecture 15
3.1 General Overview 15
3.2 UAV 15
3.3 Autopilot 16
3.3.1 Hardware 16
3.3.2 Software 17
3.4 Gimbal and Cameras 17
3.5 ODROID XU-4 18
3.6 Power Supply 19
3.7 Lidar 20
4 Control Software 22
4.1 Software Architecture 22
4.2 Inspection Phases 23

5 Virtualization Environments 25
5.1 Virtualization 25
5.1.1 Hypervisor-based Virtualization 25
5.1.2 Container-based Virtualization 26
5.1.3 Containers vs. Virtual Machines 27
5.2 Docker 27
5.2.1 Docker Engine 28
5.2.2 Docker Images 29
5.2.3 Docker Volumes 29
5.2.4 Docker Containers 29
5.2.5 Dockerfile 30
5.2.6 Docker Compose 31
5.2.7 Docker Registry 32
5.2.8 Docker Swarm 32
5.2.9 Docker Security 33
6 Software Testing, Continuous Integration and Delivery 34
6.1 Continuous Integration 34
6.2 Continuous Delivery 35
6.3 Continuous Deployment 35
6.4 CI/CD Tools 36
6.5 CI/CD Pipeline 36
6.5.1 CI Pipeline 37
6.5.2 CD Pipeline 38
6.6 Practices for Successful CI/CD pipeline 39
6.7 Benefits and Challenges of Using CI/CD pipeline 40
6.8 Software Testing 41
6.8.1 Levels of Testing 42
6.8.2 Types of Testing 45
6.8.3 Prioritizing tests 47
7 Implementation of Secure Software Supply Pipeline 48
7.1 Test Implementation 48
7.1.1 Level 1 Tests 49
7.1.2 Level 2 Tests 50

7.1.3 Level 3 Tests 54
7.2 Log Analyzer 68
7.3 CI/CD Pipeline Implementation 68
7.3.1 CI/CD Architecture 68
7.3.2 Multi-Stage Docker Build 71
7.3.3 Security Consideration 72
8 Discussion, Results, and Evaluations 73
8.1 Field Testing 73
8.2 Benefits of the CI/CD Pipeline and Automated Testing 73
8.3 Considerations 77
8.4 Developer Survey 77
9 Conclusion and Future Work 78
References 79
Appendices 84
I. System architecture of the UAV control package 84
II. Guidelines for testing loops 85
III. Reporting test results and documenting test cases 86
IV. State Watcher – Lidar Integration Test 87
V. Sample FSM test file 90
VI. Publish state variable example 91
VII. States example check 92
VIII. Global variables 93
IX. Log Analyzer 94
X. Multiple Runners - One Machine 102
XI. Common issues 103
XII. Docker image update 104
Non-exclusive licence to reproduce thesis 106

List of Figures

Figure 1. Global annual installed wind capacity 2001-2017 [8] 6
Figure 2. (a) Lightning strike on a turbine blade, (b) blade damage from a lightning strike, adapted from [11]; 8
Figure 3. Leading edge erosion form least (a) to most severe (c), adapted from [15]; 8
Figure 4. Estimated share of wind turbine inspection methodologies in the USA in 2016, Source: Ariel Avitan, Perception [16] 9
Figure 5. Manual rope inspection [17] 10
Figure 6. Blade inspection using a ground-based camera [19]....11
Figure 7. Inspection using an UAV [17] 12
Figure 8. DJI M600. Motor 1, 3 and 5 rotates clockwise, while motor 2, 4 and 6 rotates counter clockwise. The IMU is located in the center of gravity. Adapted from [22] 15
Figure 9. Top view of the ODROID XU-4 board; Adapted from [24] 19
Figure 10. Principle of working of a lidar. The red wave is the light emitted from the source, and the green light is the reflected wave. Based on [30]. 20
Figure 11. Hokuyo UTM-30LX Scanning Laser Rangefinder; Courtesy to [31]....21
Figure 12. Cross-section of a wind turbine blade....23
Figure 13. Type 1 and Type 2 hypervisor-based virtualization....26
Figure 14. Container-based virtualization 27
Figure 15. Basic Docker architecture 28
Figure 16. Dockerfile-image relation 29
Figure 17. Docker Container 30
Figure 18. Comparison between Continuous integration, Continuous delivery and Continuous deployment (from top to bottom respectively); Based on [47] 35
Figure 19. CI workflow; Based on [47] 37
Figure 20. Deployment pipeline; Adapted from [47]....38
Figure 21. The four levels of software testing regarding complexity and development time frame....43
Figure 22. The four levels of software testing and the part of the code they are focused on 43
Figure 23. ROS Levels of Testing; Adapted from [60]....45
Figure 24. FSM diagram 55
Figure 25. Architecture of the FSM package 56
Figure 26. Visualization of the condition check (equivalently the condition can be set to change state instead of failing the test) 59
Figure 27. A hierarchical representation of a transition condition 60
Figure 28. GCS GUI....61
Figure 29. GCS station state messages....61
Figure 30. Simplified pipeline configuration 69
Figure 31. CI/CD configuration 70
Figure 32. Docker Stages Structure....72
Figure 33. Number of founds and resolved bugs 75
Figure 34. Test coverage 75
Figure 35. Number of merged pull requests....76

Figure 36. Number of commits 76
Figure 37. Production level defects 76
Figure 38. (a) Simple, (b) nested, (c) concatenated and (d) unstructured loop 85

List of Tables

Table 1. Wind turbine defects [13]...........7
Table 2. DJI M600 Pro components and specifications; Adapted from [22] 16
Table 3. Odroid-XU4 specifications; Courtesy to [24] 18
Table 4. TB47S battery specification; Courtesy to [22] 19
Table 5. Hokuyo UTM-30LX Scanning Laser Rangefinder specification; Courtesy to [31]...........21
Table 6. Common Dockerfile commands; Adapted from [40]...........30
Table 7. Directories in the project 51
Table 8. States definitions 62
Table 9. Priority codes used in Table 10 63
Table 10. List of states and messages...........63
Table 11. Implemented unary operations 64
Table 12. Implemented binary operations 65
Table 13. Average GitLab stage durations 71
Table 14. Testing types...........71
Table 15. Number of full-time developers in the team...........74
Table 16. Experienced benefits after adopting the pipeline...........77
Table 17. Experienced challenges during the implementation and after adopting the pipeline 77
Table 18. Documenting test cases 86

Abbreviations

CD - Continuous Delivery

CI - Continuous Integration

CLI - Command Line Interface

IMU - Inertial Measurement Unit

FSM - Finite State Machine

GCS - Ground Control Station

GPS - Global Positioning System

ROS - Robot Operating System

SBC - Single-Board Computer

SITL - Software in the loop

SSP - Secure Software Pipeline

UAV - Unmanned Aerial Vehicle

VCS - Version Control System

VM - Virtual Machine

Abstract

Building an Efficient and Secure Software Supply Pipeline for Aerial Robotics Application

Unmanned aerial vehicles (UAVs) used for wind turbine inspection need robust software testing and deployment strategies which go beyond the traditional pre-deployment validation on real hardware. The goal of this thesis is to implement secure software supply pipeline within a case organization and evaluate the results of the implementation. The pipeline leverages the Docker containerization environment coupled with the advancements in Continuous integration and Continuous delivery practices. The result of this thesis is an automated testing and delivery pipeline which can be used for testing single components as well as the system as a whole in an efficient way, effectively reducing the time and effort required to make a new release.

CERCS: T120 Systems engineering, computer technology; T125 Automation, robotics, control Engineering; P170 Computer science, numerical analysis, systems, control

Keywords: Docker, Continuous integration and delivery (CI/CD), Robot Operating System, Unmanned Aerial Vehicle

Resümee

Efektiivne ja turvaline tarkvaraarendusahel lennurobootika rakenduses

Tuulegeneraatorite inspekteerimiseks mõeldud mehitamata õhusõidukite tarkvara vajab testimis- ja kasutuselevõtustrateegiaid, mis võimaldaksid enamat kui tavapärane valideerimine pärast reaalse riistvara valmimist. Käesoleva töö eesmärgiks on näidisettevõttes turvalise tarkvarakonveieri (ingl *software supply pipeline*) juurutamine ning selle mõju hindamine. Konveieris kasutatakse Docker konteinerkeskkonda koos pidevkooste- ja pidevvalmidustavadega (ingl *Continuous Integration (CI) and Continuous Delivery (CD)*). Magistritöö tulemusena valmis automaatne testimis- ja valmiduskonveier, mille abil võimaldatakse nii üksikute osade kui ka kogu süsteemi tõhus testimine ning vähenevad tarkvara väljalaskmisega seotud kulud.

CERCS: T120 Süsteemitehnoloogia, arvutitehnoloogia; T125 Automatiseerimine, robootika, control engineering; P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Märksõnad: Docker, pidevkooste ja -valmidus (CI/CD), robotitarkvararaamistik ROS, mehitamata õhusõiduk

Acknowledgments

I would like to thank my supervisor Karl Kruusamäe for his sound advice, valuable feedback and for getting me interested in ROS. I would also like to thank the team of Pro-Drone for giving me the opportunity to work on this project. I would further like to thank my co-supervisor Illia Sheremet from Pro-Drone for the continuous help, support and patience, and my colleague from Pro-Drone Evangelos Mantas for his introducing me to the basics of CI/CD.

S. Арсовска

/Dzvezdana Arsovska on 25.12.2018/

1 Introduction

Today unmanned aerial vehicles (UAVs) are becoming increasingly popular and are widely used in various sectors. One area of application is the automated inspection of wind turbines. However, the development in this field has been slowed down by the costs of failure during the inspection. A mid-air failure might result in a damaged and in some cases unusable vehicle. Additionally, the pre-deployment testing needs to be carried out on a fully stopped wind turbine and in ideal weather conditions in order to decrease the chances of a collision between the turbine and the UAV due to high winds. The specific conditions required for a pre-deployment testing and the resulting cost of a critical mid-air failure imply that successful development of a fully autonomous wind turbine inspection solution heavily depends on the creation of a secure software pipeline (SSP). The SSP covers all steps of robotics software design: software development, simulation and the creation of specialized and automated building, testing and deployment strategies.

As a result, in this thesis, research towards the area of software testing, simulation and deployment was performed. The work was carried out according to the requirements and guidelines from Pro-Drone, a Lisbon based start-up focused on the development of a fully automated wind turbine inspection solution. The developed components were integrated into existing business processes and practices. The work focuses on UAV system developed in Robot Operating System (ROS).

Firstly, different testing practices are evaluated and testing guidelines are developed. A modern automated testing and delivery pipeline is developed and used for testing and validation of single components as well as the entire system as a whole in an efficient way. The testing is performed using simulated system components alongside pre-recorded real system data in the form of a rosbag. The development of the testing pipeline as well as the deployment of the resulting product required the creation of repeatable, clean and reproducible testing and development environments. This was achieved by making use of the recent advancements of the containerization technology, namely Docker. The effects of introducing Docker to the CI/CD are presented and analyzed. Finally, the security of the SSP is assessed, and threats are identified. As a result, methodology for securing the pipeline is proposed and discussed. Additionally, the challenges and the best practices related to successful a SSP implementation are presented.

The obtained results showcase several benefits among which improved release planning and more systematic way of catching bugs was created.

1.1 Practical Work and Objectives

In mid-2017, Pro-Drone has identified a need to improve its development process, focusing explicitly on the software testing. Prior to this thesis, the testing process was unstructured, manual and highly informal. Developers tested "what they deemed was necessary to be tested" without having a systematic approach towards this issue. Project members were left to solve problems as they arise, which resulted in project delays. Testing did not play a significant role in the development, where an accent was put on fast-track developing without properly focusing on testing the existing and the newly developed code.

Only few unit tests existed, and they were poorly documented or not documented at all, and no documentation regarding how the testing processes should be performed was created. Additionally, different testing libraries were used by the developers.

As part of Pro-Drone my work focused on solving the following problems:

- Create product-specific, risk-focused, diversified and practical tests;
- Create well documented and well-defined procedures that describe who performs the testing, how testing should be performed, which pieces will be tested and how much testing is adequate;
- Identify the most suitable solutions for creating a CI/CD pipeline;
- Implement a fully automated testing pipeline;
- Create a specific and detailed procedure for documenting the tests (that contains at least a simple classification of faults, documentation of test cases and documentation of test results);
- Analyze how well the implemented pipeline solves the company needs;

The work in this thesis was directed towards developing an automated testing and delivery pipeline, whose purpose is systematic checking of bugs and thus lowering the chances of a mid-air failure. The simulation environment means that fewer on-site tests will be required. For this purpose, the following work was done:

- Additional sensory safety check algorithms were implemented to notify the operator about the current state of the UAV via a modified version of the GCS GUI. The messages include useful information about the current inspection as well as potential safety warnings or emergencies that require immediate attention.
- Twelve rosbag Log Analyzers were implemented in order to create a quick initial debugging tool. The Analyzers use both the logs which contain all terminal output and the rosbag generated after the end of the survey to detect events of interest which may have caused the failure.
- Appropriate guidelines were created and testing libraries were identified.
- The control software was tested in three different levels. Where necessary, existing Level 1 tests were improved and modified, and new tests were added. Level 2 tests were implemented from scratch. The complexity of the Level 3 test resulted in a necessity to

create Python package which implements a black-box system-level testing technique known as Finite State Machines. FSMs were used for modeling and testing the system since they are a convenient way to model software behavior.
- Continuous integration and delivery pipeline was implemented and an effort to improve its security aspects was made.

1.2 Requirements and limitations

The pipeline and the tests developed in this thesis are specific to the requirements of Pro-Drone. The following requirements were defined by the developers at Pro-Drone:

- The developed pipeline should be compatible with C++ and Python. The development at Pro-Drone is and will be done mostly by using C++ and Python. Therefore, the developed pipeline and methodology should be fully applicable when these languages are used for development.
- Use open-source tools to achieve end-to-end automation. Use open-source tools to design an effective framework for CI/CD to automate the source code compilation, code analysis, test execution, deployment and notifications.
- The testing pipeline and the developed methodology should be applicable to different projects.
- The created methodology should be applicable in as many situations as possible. It should also be able to explain which units of the test should be tested first (which parts of the software should be prioritized).
- Although the team has grown in size recently, it is still relatively small. Thus, the created documentation should not require complicated or long updates, and the developed pipeline should be easy to maintain even by a small team.
- Keep update sizes as small as possible.
- Minimize the time and effort it takes to make a release.
- Make it easy to have multiple versions and developers on one UAV.
- Make developer, testing, and production environments as similar as possible.
- Make setting up a development and testing environment seamless and easy.

1.3 Previous Work

Docker and CI/CD practices have already been used in several robotics projects. MoveIt! [1], one of the most popular motion planning frameworks which runs on top of ROS, uses Docker for Continuous integration. Furthermore, it encourages collaboration between maintainers and allows developers and users to use various version of MoveIt! without breaking their local workspace.

White and Christensen provide a tutorial for integrating ROS inside a Docker container, list example use cases and discuss the main reasons for using Docker for robotics applications [2]. According to them the most beneficial aspect of Docker is repeatability and reproducibility in robotics research, industry and education.

In [3] and [4], the authors propose simulation-based validation of experimental algorithms, developed for deep-sea remotely operated vehicle. Docker in their case is used for packaging, virtualization, and deployment. According to [4] their approach can be extended for other robotics applications and is beneficial for any projects that involve limited high-cost equipment shared by a larger group of people, which needs to be tested under harsh conditions.

Several authors analyzed the security aspects of using Docker containers as part of the development process.

The whitepaper Understanding and Hardening Linux Containers [5] provides a detailed overview of the container technology, evaluates their capabilities and discusses their security features. This paper will be used for explaining Docker technology and its security problems.

Furthermore, [6] and [7] describe requirements and configurations for a secure Docker environment.

1.4 Thesis Outline

The thesis is divided into nine chapters. The reader is encouraged to read the chapters in ascending order.

Chapters 2 through 6 provide the necessary background including the use case of the work presented in this thesis, the hardware and software components of Pro-Drone's product for wind turbine inspection, the theoretical background behind the Docker technology as wells as the best practices for CI/CD and software testing. The rest of the chapters focus on the practical implementation and results.

Detailed chapter explanation follows:

- *Chapter 2: Wind Turbine Inspection*
  Chapter 2 gives an overview of the modern approaches for wind turbine inspection.
- *Chapter 3: UAV System Architecture*
  Chapter 3 summarizes the hardware components used in Pro-Drone's product.
- *Chapter 4: Control Software*
  Chapter 4 describes the wind turbine inspection process developed by Pro-Drone, as well as some of the components of the UAV control software.
- *Chapter 5: Virtualization Environments*
  Chapter 5 summarizes the theoretical concepts behind different types of virtualization technologies and the differences between them.
- *Chapter 6: Software Testing, Continuous Integration, and Continuous Delivery*

Chapter 6 describes software testing and the levels of software testing, as well as CI/CD principles, benefits, and challenges.

- *Chapter 7: Implementation of Secure Software Supply Pipeline*
  Chapter 7 describes the security improvements including the sensory check algorithms, the Log Analyzers, the developed tests, and the implemented CI/CD pipeline.
- *Chapter 8: Discussion, Results, and Evaluations*
  Chapter 8 presents and evaluates the results.
- *Chapter 9: Conclusion and Future Work*
  Chapter 9 summrizes the achieved results and shortly diccusses the future work.
- *Appendix I: System architecture of the UAV control package*
- *Appendix II: Guidelines for testing loops*
- *Appendix III: Reporting test results and documenting test cases*
- *Appendix IV: State Watcher – Lidar Integration Test*
- *Appendix V: Sample FSM test file*
- *Appendix VI: Publish state variable example*
- *Appendix VII: States example check*
- *Appendix VIII: Global variables*
- *Appendix IX: Log Analyzer*
- *Appendix X: Multiple Runners - One Machine*
- *Appendix XI: Common issues*
- *Appendix XII: Docker image update*

2 Wind Turbine Inspection

This chapter reviews the current practices for wind turbine inspection and the problems that UAVs aim to solve. The current approaches for wind turbine inspection are described in detail and compared. It is implied that UAVs are a promising tool for providing fast and accurate inspection. Additionally, the advantages and the open challenges associated with UAVs are discussed.

Briefly, the current state of the wind energy industry and the importance and value of the wind energy market as part of the global electricity supply is analyzed.

2.1 Current State of the Wind Energy Industry

According to the Global Wind Energy Council, wind energy is one of the fastest-growing sources of energy in the world with over 340 000 turbines installed worldwide [8]. The Global Wind Energy Council estimates that around 20% of global electricity will be supplied by wind the energy industry by 2030 [8]. Figure 1 indicates the installed cumulative wind energy capacity on a global level.

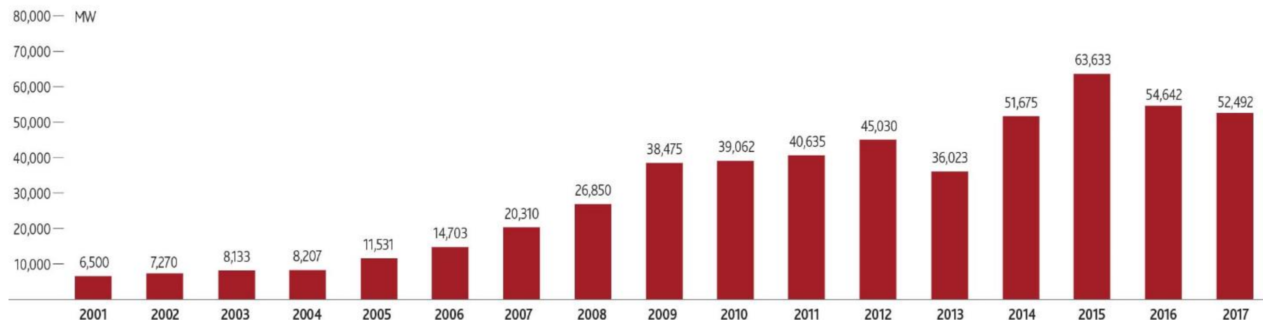


Figure 1. Global annual installed wind capacity 2001-2017 [8]

In [9] several benefits of the wind energy are presented. As a sustainable, cost-effective and clean energy source, it plays an essential role in decarbonizing the electricity supply and reducing the environmental damage caused by CO_2 emissions. The ever-increasing demand for wind energy resulted in many technological advancements like more reliable wind turbines and improved network infrastructure. Additionally, the wind energy sector related jobs, which include wind turbine manufacturing, operation and maintenance, are one of the fastest growing occupations in the US during the last decade.

However, according to [9] there are still many challenges that need to be faced before this source of energy become truly cost-competitive with traditional sources like coal or nuclear energy. One of the key issues is the associated cost of maintaining and repairing the wind turbines which comprises a significant share of the total annual costs.

For a new turbine, the share of these costs can vary between 10% to 15%, but this may increase to at least 20% to 35% as the turbine is getting older [10]. A key aspect of lowering the maintenance costs and making wind energy more affordable is finding an efficient way to perform wind turbine inspection [10, 11]. Wind turbine inspection is used for monitoring the structural health of the turbine, i.e., detecting and localizing damage, as well as for further evaluation of the degree of seriousness of the detected damage.

Table 1 presents some of the possible wind turbine defects.

Table 1. Wind turbine defects [13]

Assembly	**Possible defects**
Rotor blade	Surface damage and cracks due to weather conditions, structural discontinuities, damage to the lightning protection system
Drive train	Leakages, corrosion
Nacelle force and momentum transmitting components	Corrosion, cracks
Hydraulic system, pneumatic system	Leakages, corrosion
Tower and foundation	Corrosion, cracks
Safety devices, sensors, and braking systems	Damage, wear
Control systems and electrics including transformer station and switchgear	Terminals, fastenings, function, corrosion, dirt

Although all wind turbine parts are subjected to damage and require proper maintenance, the part that requires increasingly complex inspections is the blade [13]. According to [14] 30% of the total number of failures in wind farms was due to blade related issues, and this resulted in 34% of the total downtime.

When compared to other wind turbine parts, blade damage is one of the most expensive types of damage to repair and over time, repair costs increase non-linearly as degradation progresses [15]. This can have a huge impact on the performance and result in significant annual energy production reductions [15].

Numerous inspection reports and technical papers [11, 13, 15] indicate that the cause of blade failure is complex and often a result of a combination of factors such as loading conditions, material behavior, blade design, and weather condition. Weather factors that may cause damage on the blade's surface are airborne erosive particles (water, ice, sand, and dust), moisture ingestion, freeze and thaw cycles, UV degradation, and lightning strikes which are considered as the most dangerous factor [11, 13, 15]. An example of the damage caused by a lightning strike is shown in Figure 2. Damage caused by rain erosion is shown in Figure 3.

Since repair costs increase as degradation progresses blades continuously need to be monitored for detecting faults. This is known as preventive inspection. Preventive maintenance is performed routinely, on fixed time intervals according to the manufacturer's specification. At the moment the industry's practice is that blades must be inspected at least twice per year [15]. Alongside

preventive maintenance and inspection, reactive inspection must be performed in case of severe damage that forces the turbine to be shut down. For example, preventive maintenance would focus on detecting and repairing damage as the once shown in Figure 3 (a), opposed to reactive maintenance that needs to be done due to severe damage as shown in Figure 2 (b).

Figure 2. (a) Lightning strike on a turbine blade, (b) blade damage from a lightning strike, adapted from [11];

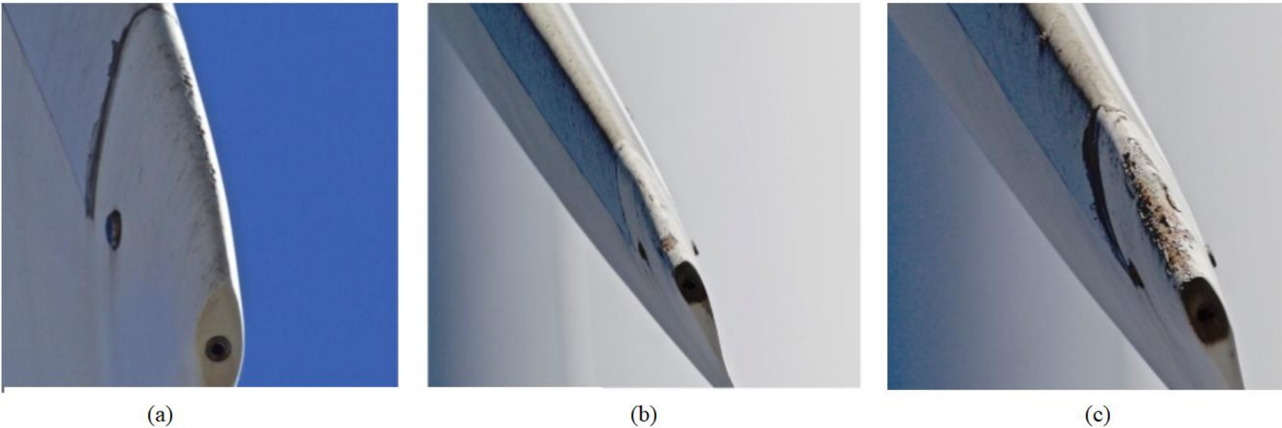


Figure 3. Leading edge erosion form least (a) to most severe (c), adapted from [15];

According to [11, 12, 13, 15] if done properly, regular inspection and maintenance can improve safety, prolong turbine’s lifetime, lower maintenance costs, minimize downtime and lower the chances of serious breakdowns. These benefits are a direct outcome from the early stage damage detections.

The main reason turbine owners postpone preventive inspection are the costs resulting from the proposed downtime and the maintenance cost [15].

2.2 Wind Turbine Inspection Solutions

Several wind turbine inspection approaches which vary greatly in quality and costs exist today. The three most common inspection methods are manual rope inspection, inspection using a ground camera and UAV inspection. Their market share is presented in Figure 4.

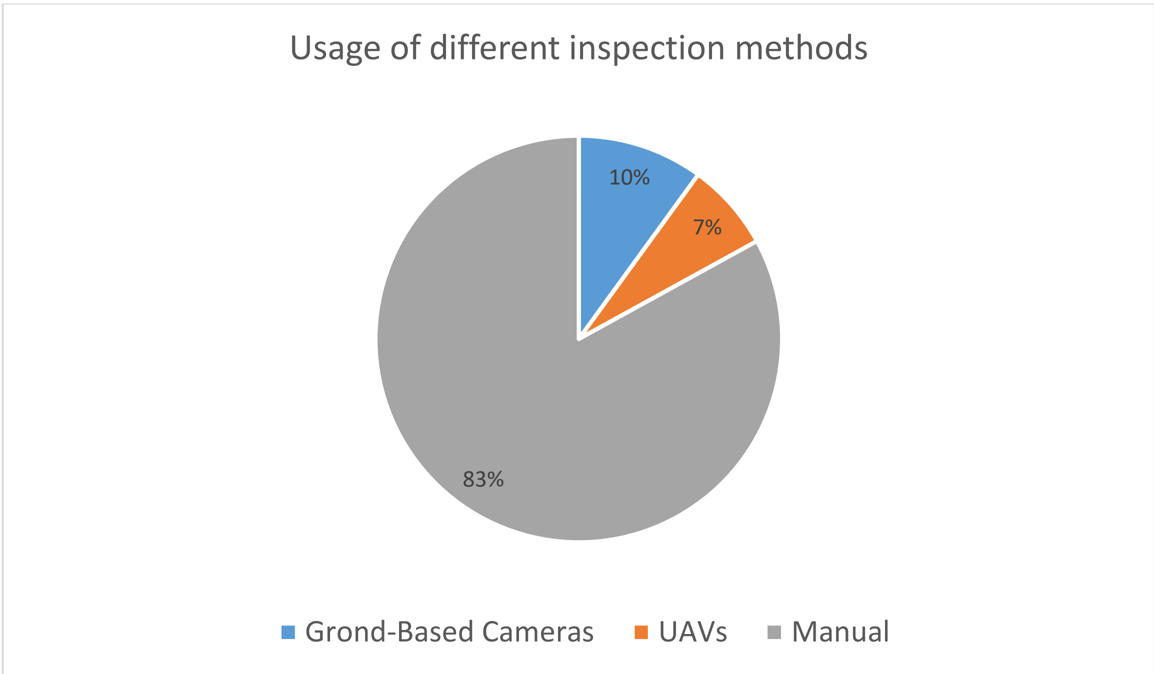


Figure 4. Estimated share of wind turbine inspection methodologies in the USA in 2016, Source: Ariel Avitan, Perception [16]

2.2.1 Manual Rope Inspection

Traditionally blades have been inspected through manual, rope-access inspection (Figure 5). During this inspection method, a team of technicians has to access the blade via the turbine ladders or lifts. The inspection is visual and followed by tapping the blade to get an idea of the structural integrity of the surrounding area [18]. The appearance of the found damage is documented by taking photos [18].

An obvious disadvantage of this method is that after the inspection is finished the exact position of the damage is hard to pinpoint [18]. The following localization of the documented damage is completely depended on the ability of the technician to correctly recall and describe the location, which makes this type of inspection prone to human error [18].

An additional disadvantage is that it is highly depended on the weather and it can be conducted only if the weather conditions do not pose a danger to the safety of the technician team [18]. Often, increase in the wind speed force the technicians to stop the inspections. This is especially common in offshore conditions, where the weather conditions change more rapidly compared to the onshore farms. This also poses additional risks to the technicians who need to be transferred by boats from turbine to turbine. Moreover, working at height is inherently dangerous, and there is an ever-present risk of falling due to equipment malfunction or the technician being struck by a falling object. Due to these risks, the technicians must be properly trained, and this results in increased cost rates [18].

Compared with the other methods, this type of inspection results in the highest downtime [16, 18]. The downtime is heavily influenced by the technician's experience, the size of the inspected blade and the weather conditions [18]. Additionally, the downtime is increased by the time it takes for the team to set up the specialized climbing platforms and ropes and the time it takes for the team to manually inspect the entire blades.

The costs for manual inspections vary greatly from country to country and range from 3000$ per-turbine in the US and Western Europe and can go down to 225$ per-turbine in parts of Eastern Europe [16]. On average, using this method only one turbine per day can be inspected [17]. Another factor that needs to be taken into consideration is that wind farms are often located in remote and difficult to access locations which result in high transportation costs if the team needs to go multiple times to the wind farm's site.

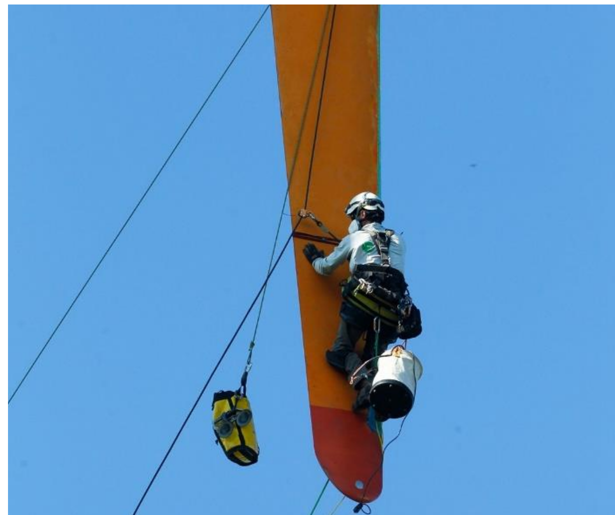


Figure 5. Manual rope inspection [17]

2.2.2 Ground-Based Camera Inspection

Alternatively, inspections can be carried out through high-quality cameras capable of capturing detailed images from a great distance (Figure 6). Using this method numerous segment images of the blade surface are created. Afterward, these high-resolution images can be combined into a

single image of the blade surface. This process requires a proprietary software which is often provided by the camera manufacturers at an additional cost [18, 19].

Compared with the rope access method, the ground-based camera method is significantly cheaper since it is easier to deploy [18, 19]. It does not require additional specialized equipment except the camera, and it is safer since there is no work at height related risks [18, 19].

The biggest disadvantage of this method is that even the images obtained by the high-quality camera can sometimes have poor quality. Weather conditions such as fog or rain and poor lighting may result in degraded images [18, 19]. Moreover, certain angles of the blade are hard to capture even when the inspection is conducted by an experienced photographer [18].

Ground-based camera inspection costs from 300$ to 500$ per-turbine [16]. On average, using this method five turbines per day can be inspected [17].

Figure 6. Blade inspection using a ground-based camera [19]

2.2.3 UAV Inspection

UAV is an airborne system that is capable of flying without an onboard human operator. It can be:

- operated remotely by a human operator,
- autonomously by an onboard computer,
- or semi-autonomously as a combination of the above-mentioned options.

Some of the biggest advancements to UAVs were achieved by the US military, and their development was fueled solely by their military applications [20]. However, recently UAV usage has expanded, eliminating low-level jobs which are considered too routine or dangerous [20].

UAVs are considered as a very promising tool for the automatic inspection of wind blades (Figure 7). Solutions with varying levels of autonomy are available on the market, and fully autonomous inspections are becoming more attainable and increasingly popular. A global market study for the

annual revenue for UAV sales and services for wind turbine inspections predicts that it will reach 6 billion USD by 2024 [21].

A fully automated UAV is capable of inspecting the entire surface of the blade from a very close distance without any human input. The UAV usually flies at the distance of 3 to 10 meters from the blade's surface, following an optimal path, and taking high-resolution images [16]. This data can be used to construct an accurate and highly detailed three-dimensional model of the blade.

Using this method, on average 10 or 12 turbines per day can be inspected. InspecTools, a company which is specialized in renewable energy plant inspections estimates that in the future this number might rise up to 15 and 20 turbines per day [16, 17].

There are varying estimates of the current cost of UAV inspection services. InspecTools estimated that UAV inspections cost 300$ to 500$ per turbine, whereas Percepto, an Israeli startup which focuses on the development of computer vision solution for drones, estimates that the costs are around 800$ per wind turbine [16].

Figure 7. Inspection using an UAV [17]

2.2.3.1 Advantages and Open Challenges in UAVs Used for Wind Turbine Inspections

The conventional problems of the traditional inspection methods can be mitigated or solved completely by utilizing UAVs for wind blade inspections. Several authors [12, 16, 17, 19] describe the advantages of using automated inspections. Some of the most important advantages are:

- *Labor cost reduction*. Small teams can inspect higher numbers of wind turbines. Compared with the rope inspection method, the price per-turbine for UAV inspection is much lower, and it has comparable cost when compared with the ground-based camera solution.

- *Reduced downtime*. UAVs are capable of inspecting higher numbers of wind turbines at a much faster rate, thus resulting in lower downtime.
- *Increased safety*. There is no working at height hazard, and the UAV operator can operate the UAV from the safety of a central service platform, eliminating the need for transferring the team from turbine to turbine using boats at rough sea.
- *Flexibility*. UAVs are lightweight and can be quickly deployed to a new location.
- *Failure avoidance*. Sudden breakdowns are avoided due to early detection of damage. Additionally, early detections allow the technician to repair rather than replace which lowers the overall maintenance costs.
- *Improved data quality*. Since UAVs are capable of flying at close proximity to the blade they can deliver high-quality data which contains detailed information about the damage and damage location. This data has higher accuracy and detail compared to the other methods. The UAVs are capable of collecting vast amounts of data in a short period of time. This data can be used for design optimization for the next generation of turbines.
- *Programmability.* Popular programming languages that are comprehensible for a large number of developers are used for UAV software development.

Even though using UAVs in wind turbine inspections resolves numerous problems, there are still several challenges that prevent their large-scale deployment. According to [16, 17, 18, 19] some of the major challenges that need to be resolved are:

- *Initial cost*. This method requires specialized equipment that should be able to operate in challenging outdoor conditions. Additionally, the autonomous navigation and the automatic detection and classification of blade damage are an active research area. This implies that the cost of research and development is higher for this method.
- *Developing fully automated inspection*. Manual piloting and inspection of several turbines whose height can exceed 100 m is a repetitive task which can be unpleasant for the pilot. Hence, a fully automated procedure is desired which adds additional development complexity.
- *Weather restrictions*. The UAV can operate as long as the wind speed is lower than 14 m/s and the temperature does not affect the battery [14].
- *Regulations*. Several countries have already passed laws and aviation regulations which are restricting the commercial and public use of UAVs. For example, the UAV must be monitored by a qualified operator on the ground, and it must fly only within the visual line of site. Some countries allow the inspection to be carried out only by a licensed operator with certified training. Also, some airspaces like the ones in the proximity to national borders or airports are strictly regulated and restricted, and flying there requires special permission.
- *Cost of testing*. The pre-deployment testing needs to be carried out on a fully stopped wind turbine. This presents additional cost for the wind farm owners. A mid-air failure may severely damage the equipment and make it unusable.
- *Image processing technology*. By using an UAV, it is possible to capture both video and high-quality images. However, high-quality data requires a large storage capacity and a pipeline that is capable of processing large quantities of data. This pipeline should be

capable of performing automated identifying, tracking and classification of damage type. Additionally, the data acquisition method must provide consistency. Otherwise, a lot of redundant data will be created.

- *Intuitive, user-friendly interface*. Since the UAV will be operated by an operator who does not necessarily have any background in software development, a user-friendly interface is needed. This interface should be able to effectively convey status, warning and emergency messages to the operator.

3 UAV System Architecture

This chapter presents an overview of the hardware used in this thesis, including the UAV, camera, autopilot, on-board computer and distance sensor.

3.1 General Overview

Pro-Drone's wind turbine inspection solution consists of a multicopter UAV which can perform inspection flights automatically and an industrial camera for obtaining inspection data. The integrated custom payload is optimized for blade inspection. A description of the individual system components follows.

3.2 UAV

When choosing an UAV whose specific application is wind turbine inspection an important factor is that it is easy to transport, store and prepare for flight. Additionally, the UAV must be able to reach high altitudes and successfully carry the payload, which in the Pro-Drone's case weights 2.3 kg.

Based on the above-mentioned requirements DJI's Matrice 600 (M600) Pro hexacopter was chosen (Figure 8). A hexacopter is a rotary-wing aircraft that has six rotors which are arranged in a circular shape. The six rotors which can be controlled independently make this aircraft very maneuverable, they enable it to fly steadily and to land safely even if two of the propellers fail. Due to its six rotors the hexacopter can reach very high altitudes. Its specifications are available in Table 2.

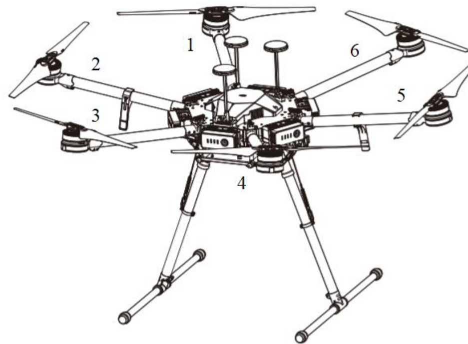


Figure 8. DJI M600. Motor 1, 3 and 5 rotates clockwise, while motor 2, 4 and 6 rotates counter clockwise. The IMU is located in the center of gravity. Adapted from [22]

Table 2. DJI M600 Pro components and specifications; Adapted from [22]

Component	Description
Autopilot	DJI and PixHawk
USB camera	Logitech c920 HD Pro Webcam
HQ camera	Sony Alpha 7R
Batteries	TB47S Intelligent Flight
On-board computer	ODROID-XU4
Transmission	DJI Lightbridge 2
Motors	DJI 6010
2D Lidar	Hokuyo UTM-30LX
Gimbal	DJI Ronin-MX 3-Axis Stabilizer

3.3 Autopilot

An autopilot is a configurable and programmable system consisting of both hardware (onboard processor, attitude sensors and I/O pins) and software and it is used for:

- *State observation*. The autopilot's processor collects all sensor readings in real time and passes them on for further processing.
- *Control*. Manual as well as autonomous guidance of the multi-rotor UAV.
- *Flight simulation for testing and debugging*. It is possible to simulate the behavior of the UAV and test new software without flying outdoors.

Pro-Drone's current system successfully supports two autopilots: DJI and the open source PX4 [23] flight stack.

3.3.1 Hardware

A minimal autopilot system includes sensors for measuring the location, velocity, and acceleration of the UAV. Available sensors include:

- *Global Positioning System (GPS)*. The signals sent by orbit satellites are used to determine UAV's specific geolocation.
- *Accelerometer*. Measures linear acceleration of a drone in the roll, pitch or yaw axis.
- *Gyroscope*. Measures the angular velocity at which a UAV rotates in the roll, pitch and yaw axis.
- *Inertial Measurement Unit (IMU)*. System which contains an accelerometer, gyroscope and in some cases a magnetometer.

- *Magnetometer*. Measures the Earth's magnetic field. This sensor is almost always present if the system has GPS input.
- *Barometer*. This measurement is used to determine UAV's altitude based on the measurement of the changes of the atmospheric pressure.
- *Distance sensors*. Used for measuring the distance from nearby objects. Typical examples include ultrasonic sensors, radars, and lidars.

It is also possible to connect additional sensors. Using a two-way communication link between the UAV and the GCS it is possible to monitor the sensor data in real time and tune parameters during the flight.

The USB support is used to connect the autopilot board with the more powerful ODROID XU-4 [24]. For the connection between ODROID XU-4 and autopilot board, the MAVLink protocol [25] is used.

3.3.2 Software

The sensor measurements are used for altitude, velocity and heading control. Based on the control techniques there are PID based autopilots, fuzzy logic autopilots, neural network, etc. The control algorithms used by Pro-Drone are described briefly in Chapter 0.

3.4 Gimbal and Cameras

The UAV is carrying:

- a Logitech c920 HD Pro webcam [26] used to stream video to the GCS, which is used for monitoring the process by the operator, and
- a high-resolution Sony alpha 7R camera [27] used for taking high-quality images of the blade.

The choice of the high-resolution industrial camera depended on the offered frames per second, resolution and the focus length. Additionally, a support system called gimbal is needed to stabilize the camera and remove vibration or shake. The gimbal used in the current payload is the DJI Ronin-MX 3-Axis Gimbal Stabilizer [28] which is compatible with the M600 Pro. A 3-axis gimbal stabilizes the camera on all three axes (roll, pitch, and yaw) and the camera can remain horizontal regardless of the motion around it.

3.5 ODROID XU-4

A single-board computer (SBC), namely the ODROID XU-4 is used for connecting and interacting with the different modules mounted on the UAV.

A SBC is a computer built on a single circuit board, and it contains all necessary features like a processor, GPU, memory, I/O ports, etc. In this setup, the task of the SBC is to run the control and computer vision software and to provide access to periphery components through its I/O interface. The ODROID XU-4 has to be chosen because it is lightweight, it has low power consumption and high computational power.

The ODROID XU-4 is the latest product of the ODROID line, and it is based on 8-core machine, with 2 GB RAM and it uses embedded Multi-Media Controller and/or micro SD for storage. It offers support for various Linux-based operating systems. The operating system needs to be installed on the external memory since the ODROID does not include NAND memory. It offers different expansion ports for connecting external sensors and equipment, and it can be used for GPU programming.

The full list of specifications is listed in Table 3. A top view of the ODROID XU-4 is shown in Figure 9.

Table 3. Odroid-XU4 specifications; Courtesy to [24]

Component	Description
CPU	Samsung Exynos5422 Cortex™-A15 2Ghz and Cortex™-A7 Octa core CPUs
GPU	Mali-T628 MP6(OpenGL ES 3.1/2.0/1.1 and OpenCL 1.2 Full profile)
RAM	2Gbyte LPDDR3 RAM PoP stacked
Storage	eMMC5.0 HS400 Flash Storage and/or MicroSD (TF) card
Display	Non MIPI DSI
Video In	Non MIPI Camera
Video Out	Standard Type-A HDMI, supports up to 1920 x 1080 resolution
Ethernet	The Realtek RTL8153-CG 10/100/1000M Ethernet
Wi-Fi	No Wi-Fi
Audio In	No Microphone
Audio Out	HDMI, Non 3.5 mm audio jack
USB	2 x USB 3.0 Host, 1 x USB 2.0 Host
GPIO	30Pin: GPIO/IRQ/ADC, 12Pin: GPIP/I2S/I2C
Power	5V/4A PSU (5.5 mm barrel connector)
Dimensions	82 x 58 x 20 mm approx.
Operating System	Linux Kernel 4.14 LTS

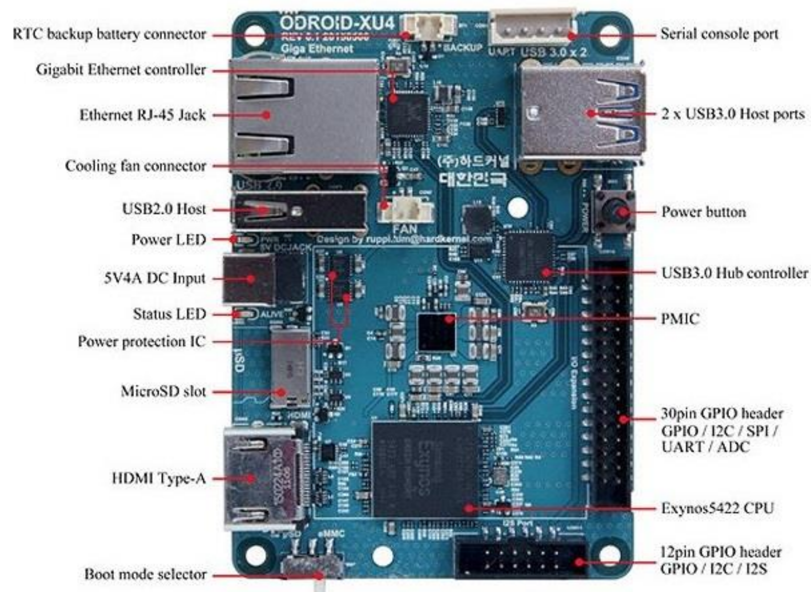


Figure 9. Top view of the ODROID XU-4 board; Adapted from [24]

3.6 Power Supply

The M600 Pro is equipped with six TB47S Intelligent Flight Batteries and a battery management system. The standard DJI Hex Charger can charge up to six Intelligent Flight Batteries and two remote controllers simultaneously. The hovering time with all six TB47S batteries without payload is 35 minutes while with 6 kg payload is 16 minutes [22]. The hovering time is based on flying at 10 m above sea level in a no-wind environment and landing with 10% battery level [22]. The relevant battery specifications are provided in Table 4.

Table 4. TB47S battery specification; Courtesy to [22]

Capacity	4500 mAh
Voltage	22.2 V
Type	LiPo 6S
Energy	99.9 Wh
Net Weight	595 g
Operating Temperature	-10 to 40 °C
Storage Temperature	Less than three months: -20 to 45 °C; More than three months: 22 to 28 °C
Max Charging Power	180 W

3.7 Lidar

A lidar is used to blade and tower detection and for continuous estimation of the distance between the UAV and the blade. By using a lidar, the distance to the target object is determined by emitting pulsed laser light with a known wavelength and measuring the time it takes to reach the target and return to the source.

The working principle of the lidar shown in Figure 10 is based on [29], the distance D between the source and the object is:

$$D = \frac{ct}{2} \tag{1}$$

where c is the speed of light and t is the time it takes for the light to hit the object and reflect back and it can be calculated as:

$$t = \frac{\phi}{\omega} \tag{2}$$

where ϕ is, the phase shift and ω is the angular frequency of the pulsated laser light. Substituting (2) in (1):

$$D = \frac{c\phi}{2\omega} = \frac{c}{4\pi f}(N\pi + \Delta\phi) = \frac{\lambda}{4}(N + \Delta N) \tag{3}$$

where N is the whole number of wavelengths, $\Delta\mathrm{N} = \Delta\phi/\pi$ is the residual of the wave and $\lambda = c/f$. From (3) it can be concluded that the distance is calculated as a function of the laser wavelength and the number of the cycles.

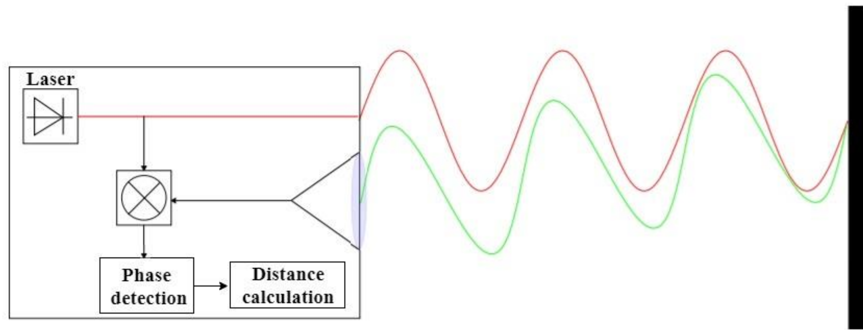


Figure 10. Principle of working of a lidar. The red wave is the light emitted from the source, and the green light is the reflected wave. Based on [30].

Figure 11. Hokuyo UTM-30LX Scanning Laser Rangefinder; Courtesy to [31]

The lidar used in the current payload is the Hokuyo UTM-30LX Scanning Laser Rangefinder, (Figure 11). Its specifications are presented in Table 5. According to [31], some of the main characteristics which make this particular lidar suitable for the wind turbine inspection are:

- Its ability to obtain measurement data while keeping a distance from up to 30 m. This means that the UAV will not have to get too close to the turbine and risk a collision.
- It can be used outdoors and is less susceptible to ambient light due to its internal filtering and protective housing.
- Due to its low power consumption, it can be used on battery-powered UAV.

Table 5. Hokuyo UTM-30LX Scanning Laser Rangefinder specification; Courtesy to [31]

Operating voltage	12 VDC ± 10%
Interface	USB 2.0
Scanning range	0.1 to 30 m
Angular resolution	0.25°
Scanning accuracy	0.1 to 10 m: ± 30 mm, 10 to 30 m: ± 50 mm
Scanning time	25 msec/scan
Resolution	1 mm
Weight	370 g
Dimensions	60 x 60 x 87 mm

4 Control Software

This chapter addresses various lower lever architectural decisions as well as the software development tools used by the Pro-Drone. Furthermore, Pro-Drone's procedure of wind turbine inspection is described. Additionally, the data collected during a successful inspection (including images, navigation data and other sensory measurements needed for the post-processing) is reviewed.

4.1 Software Architecture

This section covers the various software development tools used for developing the control software as well as the lower level architectural decisions.

Robot Operating System (ROS) [32]. ROS and its Jade distribution are used as a framework for developing robotic systems. ROS is an open-source framework which provides numerous advantages such as various communication patterns between nodes, package management, hardware abstraction, low-level device control, and modularity. ROS has a large and active community which is really beneficial since it offers additional support during the development process. The main reason for choosing ROS Jade was Ubuntu prerequisites.

Gazebo [33]. Gazebo is ROS compliant simulator which can be used for testing the code in complex simulated indoor or outdoor environments before running it on the real robot.

Operating system (OS). Ubuntu 14.04 [34] is the operating system used in Pro-Drone.

C++ programming language. C++ 11 was used to develop the control software. The motivation for using C++ is the fast runtime code and its real-time restrictions which offer performance optimization.

Python programming language. Python 3.4 was used for implementing for higher-level testing. The reasoning behind this is covered in detail in Chapter 7.1.

Version control system (VCS). VCS is needed for tracking code changes, keeping backup of the project and developer collaboration. During the development process in Pro-Drone, Git [35] was used as VCS alongside with Bitbucket web platform [36] as remote version control repository.

Integrated development environment (IDE). IDE is an application that provides a graphical user interface for used for code development. The IDE used in the development process is Clion by JetBrains [37]. Clion offers C/C++ and Python support, as well as support for Google Test, which was used for the test implementation. Furthermore, it provides VSC integration, syntax highlighting, refactoring, easy navigation and autocomplete of lines of code, which make the development process easier.

4.2 Inspection Phases

The overview of the system architecture is provided in Appendix I. Pro-Drone has developed semi-autonomous and robust blade inspection procedure. The aim is reaching a fully automated inspection in the future. Pro-Drone's proprietary software allows the UAV to navigate in relation to the blade itself and autonomously set and follow an optimal inspection path that covers the specific blade that needs to be inspected first. Then, the inspection is carried out by capturing high quality and consistent images. The average inspection time per blade is 6-10 minutes and the average full operation time is 55 minutes.

The typical cross-section of a blade and its sides is shown in Figure 12. The inspection can be carried out on both onshore and offshore wind turbines with an arbitrary number of blades.

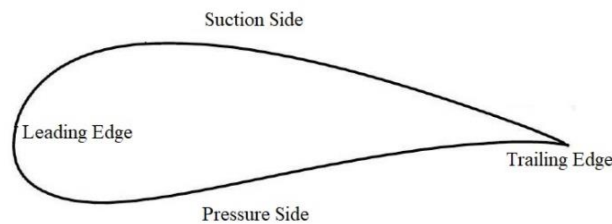


Figure 12. Cross-section of a wind turbine blade

During the inspection, it is assumed that the wind turbine is not rotating. One of the blades is rotated until is parallel to the turbine tower and is facing downwards. Before the inspection starts, the UAV is positioned directly below the blade that needs to be inspected. Pro-Drone's inspection procedure is divided into eight phases:

1. *Locking the blade*. This phase handles the initial positioning of the UAV. First, the operator manually lifts off the UAV and positions it in proximity of the suction side of the blade. Once the tower and the blade are detected the drone goes into autonomous mode. Then the UAV goes down until it loses sight of the tip of the blade.
2. *Suction side*. This phase starts once the UAV loses sight of the tip of the blade. Afterward, the UAV goes up again and saves the height value of the tip. In this way, the inspection starts at a known position. Then the suction side is scanned. Once the UAV is near the end of the blade, the ready for hub command is activated. As soon as the hub is reached and the UAV loses sight of the blade, the UAV stops and goes down until it relocks the blade.
3. *Rotation to the leading edge*. Once the blade is relocked the UAV rotates to the leading edge.
4. *Leading edge*. When the rotation is done, the UAV goes down scanning the leading edge until it loses sight of the tip. Once this happens, the drone stops and goes up until it relocks the blade.
5. *Rotation to the* pressure *side*. When the blade is relocked the UAV rotates to the pressure side.

6. Pressure *side*. Then the UAV starts the pressure side scan by going up. At the end of the pressure side of the blade ready for hub command is again activated. When the UAV reaches the hub and unlocks the blade the UAV stops and goes down until it relocks the blade.
7. *Rotation to the trailing edge*. Once the blade is relocked the UAV rotates to the trailing edge.
8. *Trailing edge*. After the rotation is done, the UAV goes down scanning the trailing edge until it loses sight of the tip. With this inspection finishes and manual control is taken to land the drone.

A phase ends when the conditions for that phase are fulfilled, and then the next phase is initiated. The process is fully autonomous with the exception of the drone's lift off and landing.

Additionally, there is an option for manual assistance. In that case, the operator can use the GCS and assist the autonomous mode by sending manual commands. If the UAV loses sight of the wind turbine's tower for longer than 5 seconds, manual control is taken to relocate it. Furthermore, emergency operating mode was implemented for cases when no proper communication exists with the remote control in order to prevent uncontrolled flight of the UAV.

Data acquisition

The blade images are taken with the high-resolution Sony alpha 7R camera which is capable of taking high quality and consistent 0.4 mm per pixel images. An automatic image capturing method capable of making automatic zoom, focus and photo taking frequency adjustments was implemented. The capture frequency was optimized to avoid overlap in the captured area. If necessary, the parameters can be adjusted manually via the GCS. Furthermore, a real-time video link from the webcam is transmitted to the GCS. The high-quality images are processed after the inspection, and they are not transmitted during the flight.

Additional sensory data from the flight needed for simulation and testing is saved in the form of a *rosbag*. This will be additionally addressed in Chapter 6.8.

Image processing pipeline

The data collected during the inspection (around 400 images per blade) is stored and processed on a cloud platform developed by Pro-Drone. This platform provides fault analysis in the form of a report which contains the blade status and accurate damage location. A report covering wind turbine with three blades takes approximately 15 minutes to be generated. It is also possible to link the images based on a matching point and create a stitched image of the entire blade. Additionally, automatic damage categorization software which can assist in identifying and planning the most critical repairs that need the most immediate attention is being developed.

5 Virtualization Environments

Firstly, this chapter provides an introduction to virtualization technologies, followed by an introduction to Docker and container technology. A comparison for different virtualization approaches is provided. Since Docker is an important part in the practical implementation of the testing and deployment pipeline, this chapter explains the ecosystem and some of the most commonly used features in detail. Additionally, practical examples are provided for some of the features.

5.1 Virtualization

Virtualization is the process of running multiple isolated guest operating systems on a host machine. The host machine provides the physical hardware components and resources.

There are two type of virtualization, hypervisor-based virtualization, and container-based virtualization [5]. They use different architectural approaches. Containers provide operating system level virtualization, while hypervisor-based virtualization provides virtualization at the hardware level.

5.1.1 Hypervisor-based Virtualization

In hypervisor-based virtualization, host machines and their resources are controlled by a hypervisor, which abstracts hardware resources for virtual machines. Hypervisors act as a platform for the VMs to be created on and the VM runs on top of the host machine using the hypervisor. Each VM, also known as a guest machine, contains a full copy of the operating system, supporting libraries, dependencies, and applications [38]. There are two types of hypervisors (Figure 13):

- A Type-1, native or bare metal hypervisor has its own operating system, and it can run directly on the host machine's hardware. Since the VM can have a direct access to any hardware component, and the hypervisor has its own drivers there is no need for a host operating system.
- Type-2 or hosted hypervisor runs on top of the host operating system. VM's that run on hosted hypervisors do not have direct access to hardware drivers, and the drivers are controlled and accessed only by the host operating system.

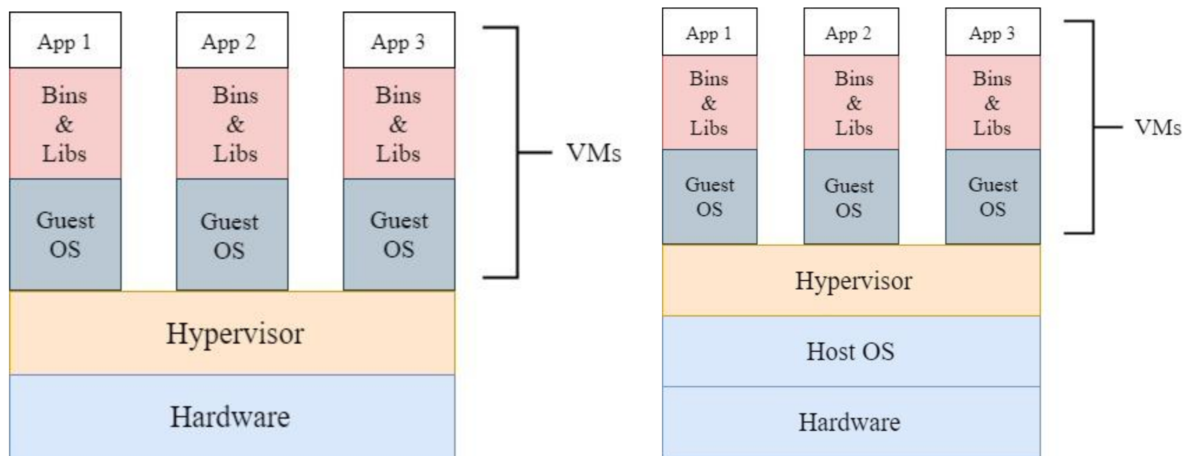


Figure 13. Type 1 and Type 2 hypervisor-based virtualization

5.1.2 Container-based Virtualization

Container-based virtualization or containerization uses isolated environments called containers to isolate applications from each other. These containers share the host system's kernel, so there is no need of a hypervisor. Containers only contain application specific binaries and libraries.

From Figure 14 it can be seen that each application runs on a separate container which means that each application is independent and it will not interfere with another application.

The containers are implemented by using different Linux kernel features like control groups (cgroups) and namespaces [5]. Cgroups are used to restrict or prioritize resource allocation like CPU or memory usage in order to prevent single container from overloading the host system [5]. There are different kinds of namespaces, like PID, networking and mount namespace, and they ensure process isolation by limiting what containers can see and access [5]. When a container is run, a namespace is created which will be used by the container. Each namespace appears to be its own separate environment to the processes within. Some of the namespaces used by modern container technologies are [39]:

- *Process ID (PID)*. Provides containers with scoped process view.
- *NET*. Provides containers with a scoped view of the network stack.
- *InterProcess Communication (IPC)*. Isolates IPC resources between processes running inside each container.
- *MNT*. Provides isolation of mount points for processes.
- *USER*. Used to isolate users within each container.
- *UNIX Timesharing System (UTS)*. Allows containers to have their own hostname and Network Information Service domain name that is different from the ones assigned to other containers and the host system.

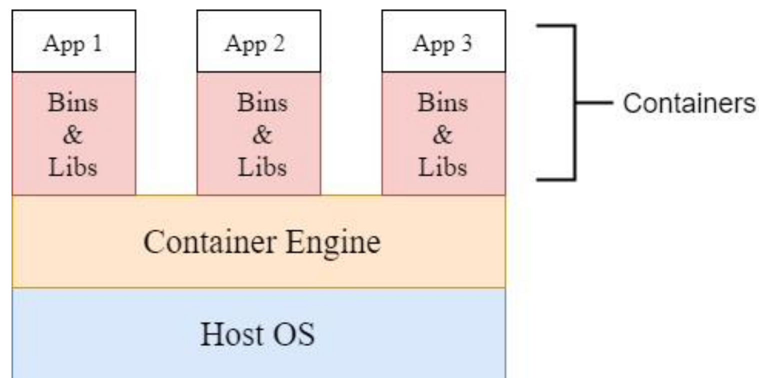


Figure 14. Container-based virtualization

5.1.3 Containers vs. Virtual Machines

Running several VMs on top of the same operating system causes performance degradation. Starting a VMs takes some time since it boots a full virtual operating system. Compared to VMs, containers are lightweight, have a better performance and a shorter start and stop time [38]. On the other hand, due to the hypervisor, VMs provide more isolation. In terms of security, in case of an attack, the attacker would have to break the operating system kernel and then the hypervisor in case of VMs, compared to containers where the only layer of security is the operating system kernel itself [5]. Containers are relatively new compared to VMs, and as a result, many companies are hesitant about solely relying on containers. Thus, hybrid systems of containers running inside VMs are common [38].

5.2 Docker

The container technology research began at the end of the 90s, with companies like Google, FreeBSD and Solaris Zones aiming to provide complete containerization technology [38]. The first successful implementation of what we today consider to be a container was the Linux Containers (LXC) project, which was created using cgroups and namespaces.

Docker was released in 2013, and it became the first mainstream container technology [38]. Docker is an open-source containerization platform that is used for packaging applications and its dependencies into containers. It can be used during the application's development and deployment. It extended the LXC's solution by adding, among other things, portable images and a user-friendly interface [38].

Docker's containers are lightweight and fast. They provide a possibility to create standardized environments, which means that the developers can develop and test application in the same environment as the one that is used during production and testing.

Furthermore, the containers can be integrated into a CI/CD pipeline which can be used for automatic testing and deployment of the images.

The main components Docker of are:

- Docker Engine (the core of the Docker ecosystem)
- Docker Compose (definition of one or multiple services in a single file)
- Docker Swarm (orchestration of containers on highly available clusters)
- Docker Registry (cloud storage of Docker images)

5.2.1 Docker Engine

Docker engine, the core of the Docker ecosystem is a client-server application, on which Docker runs [39]. It has three major components:

- A Docker Host or server running the Docker Daemon in the host computer.
- A command line interface (CLI) client that communicates with the Docker Daemon to execute commands.
- A REST API which specifies the interface between the client and Docker Daemon.

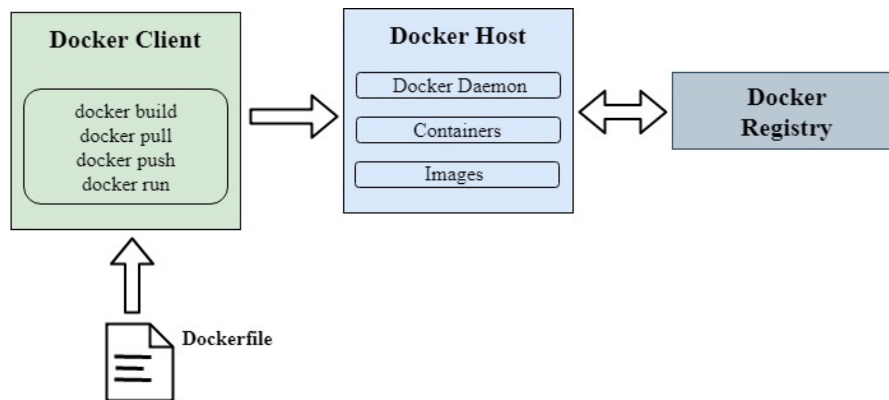


Figure 15. Basic Docker architecture

The Docker Daemon which runs within the Docker Host is responsible for listening and processing API requests from the Docker Client [39] (Figure 15).

The user never communicates directly with the Daemon. This communication is done via the Docker Client which communicates and sends user commands to the Docker Daemon. The Docker Daemon executes commands sent by the Docker Client, and it manages Docker Objects like building images and running and distributing containers. It also manages the network and the data volumes [39]. The building of the image is done by executing a Dockerfile. The image built by the Docker Daemon can be saved in a Docker Registry (Chapter 5.2.7), which is used for storing Docker Images. Finally, if a run command is issued by the client, a Docker Container will be created in the Docker Host.

5.2.2 Docker Images

Docker Images are read-only templates that are built from a set of instructions written in a Dockerfile. Images define the application and its dependencies as well as the processes it needs to execute when the application is launched. The Docker image is built using a Dockerfile [39]. The Dockerfile's instruction defines the steps needed for creating a Docker container and running it. Each instruction in the Dockerfile adds a new layer to the image. To make the build process faster, each layer is kept as a cache, which can be reused if no changes are made. [39]. Change in one layer causes all subsequent layers to be recreated. If another build command is executed, only the layers which have changed are rebuilt [39]. Docker containers use Union File Systems (UnionFs) as building blocks for containers [39].

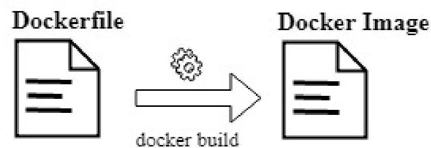


Figure 16. Dockerfile-image relation

5.2.3 Docker Volumes

Volumes are file systems, initialized when the container is created, which exists separately from the UnionFS. When the container is removed, the volumes are not automatically deleted unless it is explicitly specified. Consequently, volumes can be used for saving and sharing data between containers.

5.2.4 Docker Containers

A container is a runnable instance of an image (Figure 16). Each image is identified by a hash number, and it is just one of many possible layers of images that make up a container. A container is identified only by its top level image, which in return has references to parent images. When using the UnionFs, for example, if there are two containers with image layers a, b, c and a, b, d, both locally and in the repository, only one copy of image layer a, b, c, d will be stored. In Figure 17 Container 1 and Container 2 share the first three layers, but Container 2 has one more additional layer.

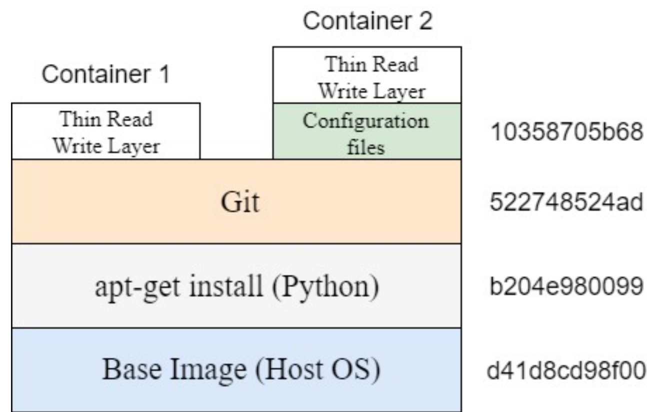


Figure 17. Docker Container

The creation, starting, pausing, stopping, moving, or deleting a container is done by using the client. When a container is created, Docker assigns an available IP address to it, and the container can be connected to a network, which allows it to communicate to the local host [39]. When a container is deleted, any changes to its state that are not stored in persistent storage are deleted [39]. The level of isolation between containers and the host machine can be controlled.

5.2.5 Dockerfile

A Dockerfile contains the instructions for building a Docker image (Figure 16). As described in Chapter 5.2.1, the Docker Engine is responsible for parsing and interpreting those instructions, then building the final Docker image from it.

Once the Dockerfile is created, it can be used to build an image using the docker build command:

```
$ docker build [OPTIONS] PATH | URL | -
```

Table 6 contains a list of the most common commands used in Dockerfile.

Table 6. Common Dockerfile commands; Adapted from [40]

Command	Description
FROM	Define the base operating system. Must be the first command in the file.
USER	Sets the default container user.
MAINTAINER	Identify the author and manager of the image. Images published on Docker-Hub, require identification of the author for future management.
RUN	Interpret and execute commands and saves the result as a new layer.
EXPOSE	Open port and expose container content.

CMD	Initial command to be executed when the image is started.
ADD	Copy files or entire directories, from the parent system into the container.
VOLUME	Adds a volume.
WORKDIR	Sets the default working directory for the container.
ONBUILD	Used when the image in the current Dockerfile is a base for another image.
ENV	Sets environmental variables.

Listing 1 shows a sample Dockerfile for a Python application, which is based on the official Ubuntu image, defined using the keyword FROM. Next, the maintainer is specified, and the port 22 is opened for communication. The dependencies are installed using the RUN command, the code file hello_world.py is copied with the ADD command into the container, and then the default directory is set. Finally, using CMD the command which needs to be executed first when the container is started is defined.

Listing 1. Example Dockerfile

```
# Use Ubuntu 14.04 as an OS
FROM ubuntu:14.04
# Specify the image maintainer
MAINTAINER Dana Arsovska <dana@prodrone.com>
# Open port 22
EXPOSE 22
# Install dependencies
RUN apt-get update
RUN apt-get install -y python python-pip wget
# Copy hellow_world.py into the container
ADD hello_world.py /home/hello.py
# Set the working directory
WORKDIR /home
# Run hello.py when the container launches
CMD [“python”, “hello.py”]
```

5.2.6 Docker Compose

Docker Compose is a tool that runs and defines a multi-container Docker application [41]. Compose is a three-step process [41]. First the environment is defined using a Dockerfile. Then the application's services, networks, and volumes are defined and configured using a YAML file called Compose file. A service can be for example a database, a frontend or a backend service. Finally, Compose is started, used to spin up the containers and run the application. Using Compose the service's status can be monitored, and the service can be started, stopped, and rebuild.

The Compose file can be used to describe the communication between services, the exposed ports, used Docker images, and environmental parameters [41]. To share data between containers, it is

possible to define data volumes. Listing 2 shows a basic Compose file which defines two services: database and nginxapp. The image keyword is a reference to the images from Docker Hub for the mysql and nginx containers. The database service has a volume and a port 5000 which is exposed to the outside applications. Finally, some environmental variables are specified.

Listing 2. Example Docker Compose file

```
version: '3'
services:
    database:
      image:mysql
         ports:
         - "5000:5000"
         volumes:
         - .:/code
       environment:
       - USER = new_user
       - PASSWORD = if495!#
    nginxapp:
      image:nginx
```

5.2.7 Docker Registry

Docker Registry is a central repository for Docker Images [42]. The Registry allows users collaboration by allowing users to share uploaded content. Images are built locally or on a build server, and they are pushed into the registry. Public images are readily available for use, usually with little-to-no modification afterward. They can be pulled and run in a container cluster like Docker Swarm [42]. The most commonly used cloud-hosted public registry is Docker Hub [42].

5.2.8 Docker Swarm

Docker Swarm is container orchestration platform which enables a group of machines that are running Docker to be joined into a cluster [43]. The machines in a swarm are referred to as nodes. There are two types of nodes:

1. managers handle tasks such as maintaining clusters states, scheduling services and serving swarm mode HTTP API endpoints, and
2. workers sole purpose is to run containers [43]. Each worker node requires at least one manager node. The swarm and its services can be controlled using the CLI.

5.2.9 Docker Security

Docker containers, compared with traditional hypervisors, have several security issues that must be addressed to make them suitable for running in production environments.

Use only trusted Docker images [5, 6]. When creating Docker containers, it is a common practice to use existing Docker images rather than building them for scratch. These images are often pulled from public repositories, and their creators, maintainers and exact content might be unknown. This may introduce vulnerabilities and configuration issues into the system. To avoid security risks, untrusted images should not be used. Container images need to be scanned before they are downloaded, or only official and verified images should be allowed to be used. Docker Store hosts containers from trusted partners, unlike Docker Hub, where anyone can push container images. Using Docker Content Trust image publishers can sign their images so that the image users can verify the identity of the publisher and the integrity of the contents of a container image [44]. The Docker Engine can be configured only to run signed images.

Manage secrets in Docker Images [5, 6]. Sensitive information like SSH keys, SSL certificates, usernames, passwords, etc. need to be handled carefully. Secrets should not be hardcoded in container images or in the application code since they will become exposed at the host level, accessible by an unauthorized team member or even external users. Docker Secrets provides centralized and programmatic way to manage sensitive information. The access to the secret should only be provided to the relevant container during runtime. Once the task is completed, the access to the secret is revoked from the container, and the secret is deleted.

Harden Docker Hosts using CIS Benchmarks [6, 7]. The Center for Internet Security (CIS) publishes documents describing security-related best practices for Docker containers. It is recommended to check the system against these benchmarks continuously. A useful tool for achieving this is *docker-bench-security*, an automated checker based on the CIS benchmarks.

Enforce Isolation with Namespaces and cgroups [5]. As described earlier, namespaces and cgroups are Linux kernel features, that limit what containers can see and how much system resources they can use. Namespaces should be configured in order to isolate containers from one another in order to prevent users or process running inside one container from spreading malicious behavior to neighboring containers. On the other hand, cgroups should be configured in order to prevent infected containers from using all the available system resources by themselves.

Privilege escalation [5, 6]. Privilege escalation happens when an attacker can gain root access on the host after gaining access to one of the containers. In traditional virtualization where the virtual environment is strictly abstracted from the host system, this issue does not exist. However, a process running inside a Docker container has the same namespace as the host system. This implies that if an attacker gets root inside a container, it is possible to then get a root access to the host system. To avoid this, the container should be run as non-root users. This is achieved by starting Docker containers with the -u flag so that they run as an ordinary user instead of root. Furthermore, only the specific capabilities that are needed by the containers should be assigned, i.e. it should have as least privileges as possible.

6 Software Testing, Continuous Integration and Delivery

This chapter reviews CI/CD and testing practices based on related literature. A focus was made on literature targeting CI/CD implementations in small businesses and startups. An explanation of how CI and CD relate to one another, and how they can transform software development and release practices is provided.

6.1 Continuous Integration

Traditionally, integration was done as one of the last steps of the project [48]. An article by Flower [48] describes the problems that occurred in a large project, that was developed over the course of a few years, where integration was done as the last phase of the project. The integration phase had taken a couple of months without any of the developers being able to pinpoint how long it will take until it was finished. Aside from being time-consuming, the integration was also error-prone.

Continuous integration (CI) is a software engineering practice, commonly used in agile software development processes, which tackles challenges experienced during software developing and release. In literature, it is defined as:

"Continuous Integration (CI) is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible." [48]

"CI stands for the collection of techniques, tools, and processes to automatically build and test software on each change." [47]

The goal of CI is to minimize the cost of integration by making it simple and repeatable process. If the developers do not integrate the changes frequently integration can become a long and unpredictable process, making it hard to predict how long it will actually take to finish it [48]. Moreover, if different parts of the software are not integrated there is no evidence that these different components will interact correctly [47]. However, by itself, integrating code frequently does not offer any guarantees about the quality and the functionality of the new code, or that it is bug-free [47, 55]. Successful CI relies heavily on robust and comprehensive automated tests. If automated tests are not implemented, new commits can introduce bugs to the codebase. Test automation is discussed later in this chapter.

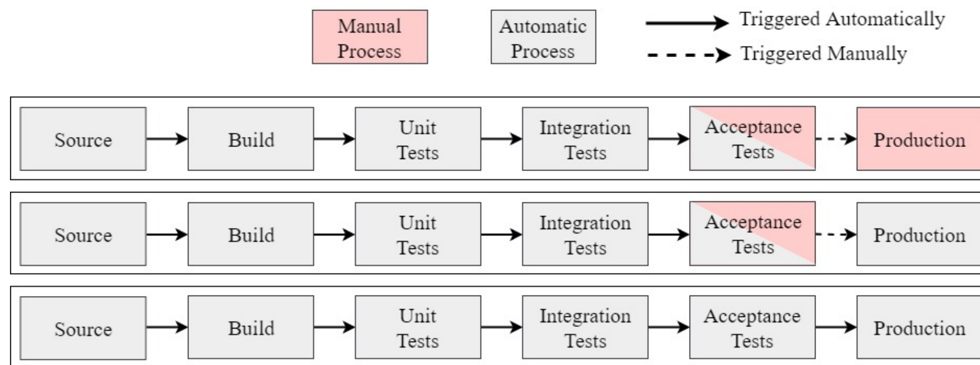


Figure 18. Comparison between Continuous integration, Continuous delivery and Continuous deployment (from top to bottom respectively); Based on [47]

Figure 18 shows the typical CI steps. It also provides a comparison between Continuous integration and two other closely related practices: Continuous delivery and Continuous deployment. From the image, it can be seen that the key difference between these three practices is the scope of the applied automation. Continuous delivery and Continuous deployment are discussed in more detail later in this chapter.

A key benefit of CI is that developers get quick feedback if the committed changes have introduced bugs to the codebase [47]. The code is proven to work with every new commit. Bugs and conflicts between the new, and the existing code can be discovered early, while they are easy to resolve. Additionally, CI makes software development and release process faster and more robust.

6.2 Continuous Delivery

According to Fowler [48], Continuous delivery extends CI by ensuring that the software is constantly in a deliverable state. This implies that a reliable CI setup is a prerequisite for implementing Continuous delivery. Each commit that successfully passes the entire pipeline can be deployed to production at any time, by anyone who has sufficient privileges. The deployment process needs to be triggered manually.

6.3 Continuous Deployment

Continuous deployment is the practice of automating the entire software release process. It extends Continuous delivery by automatically deploying to production each commit that has successfully passes the entire pipeline, without explicit approval from a developer.

6.4 CI/CD Tools

CI is largely tool agnostic, and there are numerous different tools that can be used within the CI pipeline. The optimal solution depends on many factors, from which some of them are: the type of the project, the skills and the size of the team, whether the company would prefer to use open source or proprietary software product, the features and the extensibility, the integration with existing tools, the learning curve, the supported platforms, etc. Choosing the appropriate tools is about finding a balance between the above-mentioned factors.

Although many tools can be incorporated in any CI system, the two must-have tools are the VCS and the CI server [47].

VCS is a software tool that enables the developer to keep a record of the changes of the source code. At the moment there are few popular VCSs that support CI such as Git [35]. VCSs are often coupled web-based hosting services for VC such as GitHub[1], Bitbucket [36], and GitLab[2] who additionally provide access control and collaboration features such as bug tracking, task management tools, feature requests, and wikis.

Build servers are used for building the software and running tests (for example unit tests and integration tests) automatically, on a regular basis. Various build servers are available, and they range from open source solutions like Jenkins[3], Travis[4] and GitLab CI [46] to commercial solutions like Bamboo[5] and TeamCity[6]. In addition to these, virtualization tools like Docker [39] are also commonly used to build an image for the test environment.

6.5 CI/CD Pipeline

A typical CI/CD pipeline is the implementation of an automated build, test, deploy and release pipeline. The pipeline is composed of few sequential stages or phases and it is linear and one-directional. The build is only passed to the next stage if it passes the previous stage successfully. The actual number of phases depends highly on the architectural choices of the developers, the used language, platform and tools. In order to assure the product's quality, Test automation is highly utilized in all CI/CD pipelines.

[1] GitHub, https://github.com/
[2] GitLab, https://about.gitlab.com/
[3] Jenkins, https://jenkins.io/
[4] Travis, https://travis-ci.org/
[5] Bamboo, https://www.atlassian.com/software/bamboo
[6] TeamCity, https://www.jetbrains.com/teamcity/

6.5.1 CI Pipeline

The integration pipeline described in this chapter is based on the work presented in Continuous delivery: Reliable Software Releases through Build, Test, and Deployment Automation [47]. Figure 19 shows a typical integration pipeline. The typical workflow in a CI pipeline is:

1. The developer clones the central repository from the VCS in their own local copy of the project.
2. He works on a task and commits changes locally. Once the task is completed, he builds and tests the code locally.
3. If the tests are successful, the developer pushes the local branch to the central repository.
4. The CI server monitors the repository and checks for changes. If a change is detected a new build process will be triggered on the CI server.
5. The CI server retrieves the latest copy of the repository. The developer is capable of setting how often the server will scan the repository for changes. The CI server carries out a build based on a build script. If the build is successful, the server starts a series of automated test, which checks if the software is ready to be released.
6. The server publishes whether the committed changes have passed or failed (either during the build or the testing phase). If the committed changes have passed the CI pipeline successfully, no further action is needed, and the changes can now be fetched from the VCS. On the other hand, if the committed changes cause failure the developer who triggered the build process is responsible to fix the issue at the earliest opportunity.
7. The developers check for results and take the necessary actions.

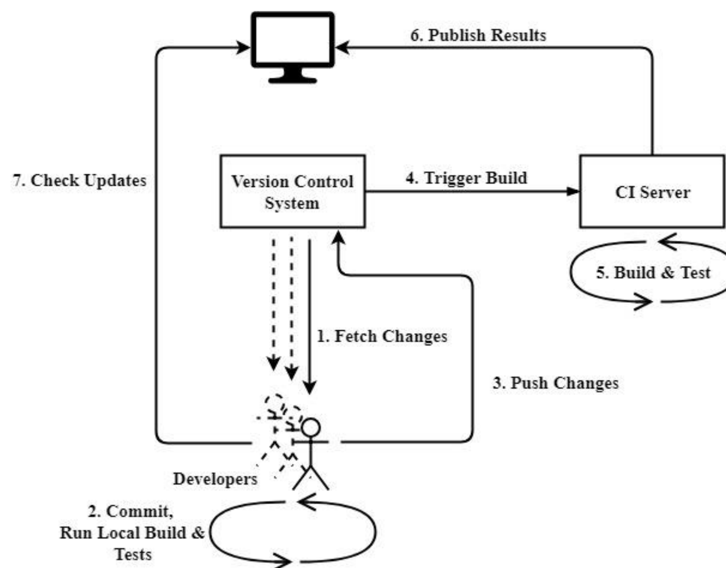


Figure 19. CI workflow; Based on [47]

6.5.2 CD Pipeline

The CD pipeline is a direct extension of the CI pipeline [45]. As in the case of the CI pipeline, the deployment pipeline is divided into a series of sequential stages. Each stage runs increasingly rigorous and time-intensive tests in increasingly like production environment. The tests must pass successfully before processing to the next stage. Each passed stage increases the confidence in the build. Feedback from each stage is sent back to the developer. The pipeline ends with a build that can be deployed to production at any time in a single step [45].

As described previously, CI/CD pipelines and their stages cannot be standardized in a way that would be applicable to all projects due to the selection of tools and the different architectural choices made by the team. However, according to Humble and Farley [47, p. 109-110], most CD pipelines contain the following stages:

1. *The commit stage.* The software is built, and a collection of automated lower level tests (such as unit tests) are started. The commit stage provides initial feedback to the developers [4].
2. *Automated acceptance test stage*. Acceptance tests are a series of automated tests used to test the functional and non-functional requirements, and they should be run in an automatically set up and configured production-like environment which replicates what the application will encounter in production.
3. *Manual test stage*. The software is tested manually in order to detect bugs that might have been missed during the automated testing phase.
4. *Release stage*. During this stage the software is released to the end users or deployed to a staging environment. In Continuous delivery, the decision to deploy is a manual process, i.e., the deployment has to be triggered manually, but the deployment itself should be performed automatically [47, p. 106]. As described previously, if Continuous deployment is used all changes are deployed without a manual approval by the developer.

The different CD stages are presented in Figure 20.

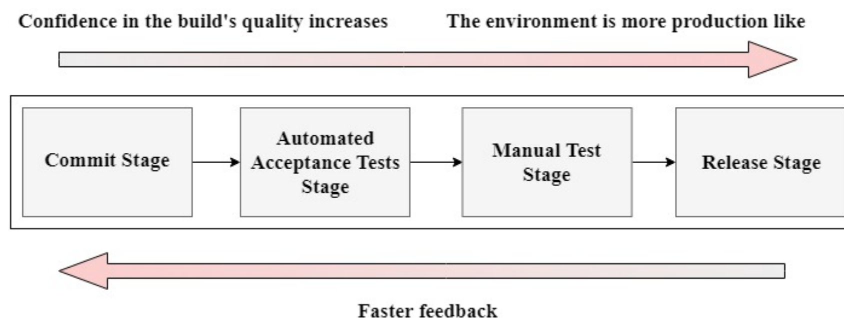


Figure 20. Deployment pipeline; Adapted from [47]

6.6 Practices for Successful CI/CD pipeline

According to [47], “CI is a practice, not a tool.” This implies that successful adopting of CI/CD requires implementation of a mixture of technical and organizational practices, as well as commitment from the development team to these practices. These practices offer a generic solution adaptable to most cases [47]. According to the relevant literature, these practices are:

Commit frequently. The frequency and the size of the commit are an important factor [47, 48]. It is recommended that the developer commits as frequent as possible and that each commit contains small change to the source code. The reason behind this practice is that the more complex each change is, the more difficult it is for the developers to identify the problems in case of a failed build and it is much harder to deal with them. Smaller commits make the issue tracking and fixing easier and more convenient.

Separate the testing into phases. The tests should be separate into different phases [48]. A positive test result from the previous phase should trigger the pipeline to move into a new phase. Each stage should contain increasingly complex tests. The reason for this separation is that the pipeline will halt the process at first sight of problems, i.e., more complex and thus time-consuming test will not be executed if earlier a simpler test has failed.

Test in a clone of the production environment. The test environment should mimic the production environment as closely as possible [48]. The reason behind this principle is that if the testing environment differs greatly from the production environment, the code might behave in an unexpected manner when deployed to production [48].

Update the tests regularly. The tests should reflect the code and should be updated as soon as new features are added to the code [48].

Keep the pipeline fast. This is accomplished by reviewing the tests frequently and ensuring that no redundant or unnecessary test conditions exist [48]. Unit and integration tests are typically very fast and require least maintenance time. In contrast, system or acceptance testing is often quite complex. To account for this, it is often a good idea to rely heavily on unit tests, integration tests, and then keep the number of the most complex tests minimal [49].

Do not allow the build to break. In CI/CD, development should always take place on a known and stable base [48]. As a result, if a code change breaks the build, it becomes a high priority task to fix it. This is event is called “stop the line” [47, p. 66].

Create a fast feedback mechanism. It is of great importance that the developers are notified and aware of the outcome of the latest build as soon as possible [47, 48]. The reason behind this is that the earlier a problem is identified, the easier and quicker it can be fixed.

Automate deployment. Ideally, the deployment pipeline should be as automated as possible, and all manual work related to the release should be minimized [48]. Good CI/CD practice allows the latest software updates to be deployed at any time, as fast as possible with little or no manual work.

6.7 Benefits and Challenges of Using CI/CD pipeline

Some of the benefits of using CI/CD pipeline are:

- Faster, easier and flexible deployment [47, p. 21, 48, 57]. Since the software can be released to production at any time in an automated way the release process is less prone to human error and it can be done by any employee.
- Increased product quality [47, p. 18, 48, 57]. CI/CD does not remove all bugs, but it does make the process of finding and removing them easier. This results in more trustworthy product and improved customer satisfaction.
- Increased productivity [47, p. 17, 48, 57]. CI/CD reduces the time it takes to release a newly implemented feature. Detecting faults at an early stage results in shorter development time. Additionally, the developers do not have to run all tests manually, and they can focus on other relevant tasks.
- The code can be updated or refactored more quickly [57]. Passing the tests verifies that no bugs were introduced while making small code changes or refactoring.
- Fast feedback [47, p. 11]. All bugs are reported immediately after they are detected.

Authors identify several challenges that may result in an unsuccessful implementation of a CI/CD pipeline:

- Technical challenges [47, 48, 55]. There are several technical challenges related to the implementation of CI/CD pipeline. First of all, the right tools need to be selected. Test cases need to be developed which can be challenging in case of a complex codebase which was developed with no testing in mind. Some of the challenges related to Test automation are described later in the text. Additionally, the pipeline and the tests need to be implemented by skilled developers.
- General resistance to change [55]. Introducing new practices requires adopting a new way of working and in some cases results in more responsibilities for some developers. Not presenting convincing reasons why the adoptation of CI/CD practices and automated testing is positive for the company, may cause skepticism and distrust towards those changes.
- Organizational practices [47, 48]. The organizational practices described in the previous chapter need to be adopted by the company.
- High short-term costs [47]. Writing non-trivial automated tests and implementing the pipeline has higher short-term costs.
- Environment inconsistencies [47]. Different production, test and development environments can cause the code not to behave in the same way.
- Poor test quality [55]. Developer's skill set and the time the team spends on writing tests play a significant role in how successful the pipeline will be. Poorly developed tests may result in unstable, long-running and low coverage tests.

6.8 Software Testing

Software testing is a widespread approach for software quality assurance. Authors provide the following definition for software testing:

“Testing is the process of executing a program with the intent of finding errors,” with the purpose of “raising the quality or the reliability of the program.” [50, p.6]

“Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended.” [51, p.8]

This definitions highlight that the main idea behind testing is to uncover evidence of faults in software, and not to show that they do not exist. The goal of software testing is ensuring that the product meets the expected quality and the requirements from the end users.

Test automation plays a crucial role in the CI/CD pipeline. It is a broad term which encompasses various activities. Consequently, authors tend to include different activities in its definition.

According to Dustin et al. [52], Test automation includes the automation of software testing activities like the development and execution of test scripts, verification of testing requirements, and the use of automated testing tools. This definition will be used throughout the thesis.

The following list of advantages is based on the work on Rafi et al. [53], who published a detailed list of advantages of automated testing based on relevant literature and the official ROS documentation [57]:

- Test automation improves product quality [53, 57].
- Implementation of test automating results in high test coverage [53].
- The time it takes to run the tests decreases [53].
- Since it takes less time to run the tests, they can be run more often [53]. Testers no longer need to run the tests manually so they can spend their time writing more comprehensive and better tests. This will result in improving the software quality, the confidence in the program and its reliability. Furthermore, since testers do not need to focus on repetitive tasks, Test automation enables better use of resources. However, it is important to note that having low-quality tests can lead to a false sense of security [54, 55].
- Increased fault detection [53, 57].
- Easier reproduction of faults [53].
- Test scripts can be reused by just making changes in the test data [53].
- A form of documentation [57]. Working on a large scale project with multiple contributors can be challenging since the developer is not familiar with all parts of the code and does not always know how a change will affect someone else’s code. So the tests become a useful “form of documentation” because a test failure precisely pinpoints which test needs to be read in order to understand the code violation that leads to failure.
- Simplified maintenance [57]. When updating packages that are not supported anymore with new packages, the tests enable quick check if all parts of the system behave correctly after the update.

Some of the challenges of implementing Test automation are:

- Automation cannot fully replace manual testing [53]. Not everything in the code base can be tested using automated tests.
- Maintenance cost [53]. Code redesign causes tests not to be valid anymore and fail. This failure occurs because the design of the test does not mirror the design of the tested unit anymore. As a result, tests need to be maintained or even completely recreated when a significant change in the design is made.
- Lack of proper test strategy and guidelines [55]. The strategy needs to explain clearly the scope, objectives, and schedule of the project, the roles, and responsibilities of the team members, the guidelines to follow, the tools that will be used, the parts of the code that will be tested, and the potential risks and benefits. The lack of a well-defined strategy and a clear vision of how and why the project will be implemented may result in unsuccessful CI/CD implementation.
- Not all tests should be automated [55]. Usually, only tests which are run often need to be automated.
- Training and skilled personnel are needed [55]. Test automation requires skilled testers.
- Technical challenges [56]. Some of the technical challenges include choosing the right tools and developing automated test cases for systems which are not designed with testability in mind.
- Low-quality test data [57]. The prerecorded sensor data used during some parts of integration and system testing must represent the real-life environment as close as possible.

6.8.1 Levels of Testing

There are four main levels of testing that need to be completed: unit testing, integration testing, system testing, and acceptance testing. The provided definitions are based on the work present in [51].

1. Unit test represents the first level of testing, and it focuses on the testing of the basic units or the basic components of the system in order to determine whether they function correctly. Depending on the type of programming the units may represent functions and procedures of the system, or independent classes and methods.
2. Integration testing focuses on combining units within a program into groups and determining if they are integrated properly (i.e., whether different components of the system can function together correctly). This testing level is used for finding interface defects between the units.
3. System testing is used for testing the final system as a whole, and it should be performed in an environment that closely mirrors the production environment. The goal of the System testing is to verify if the system satisfies the planned technical and functional requirements.

4. The final level, Acceptance testing, is conducted to determine whether a feature is correctly implemented. The focus in this type of tests in on executing scenarios from a user's perspective, by simulating the user's behavior.

Regression testing is not a separate level of testing, and it can be performed during any of the four main software testing stages. Regression testing is the repetition of previous tests in order to check whether the newly implemented feature introduced or reintroduced bug in the codebase.

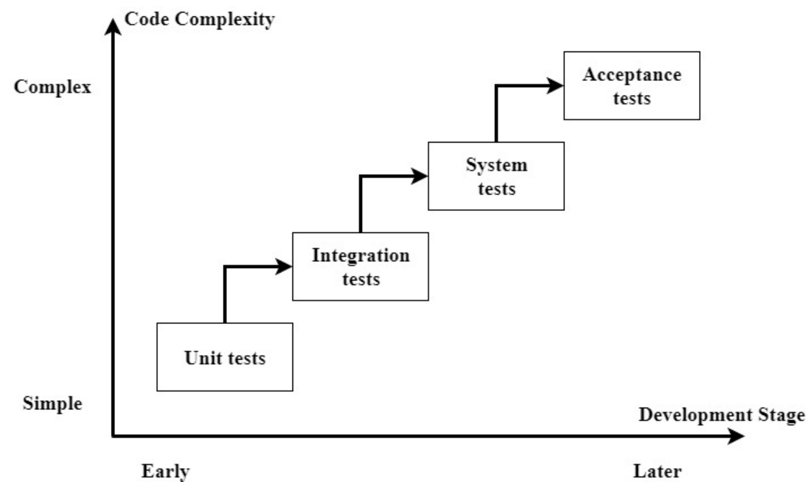


Figure 21. The four levels of software testing regarding complexity and development time frame

Figure 21 represents the four levels of software testing in terms of the stage of the software development and code complexity (in terms of the complexity of the product code and tests). Unit tests, for example, are created early in the development process and as mentioned, focus on the most basic code units. Figure 22 shows the part of the software on which the tests focus on.

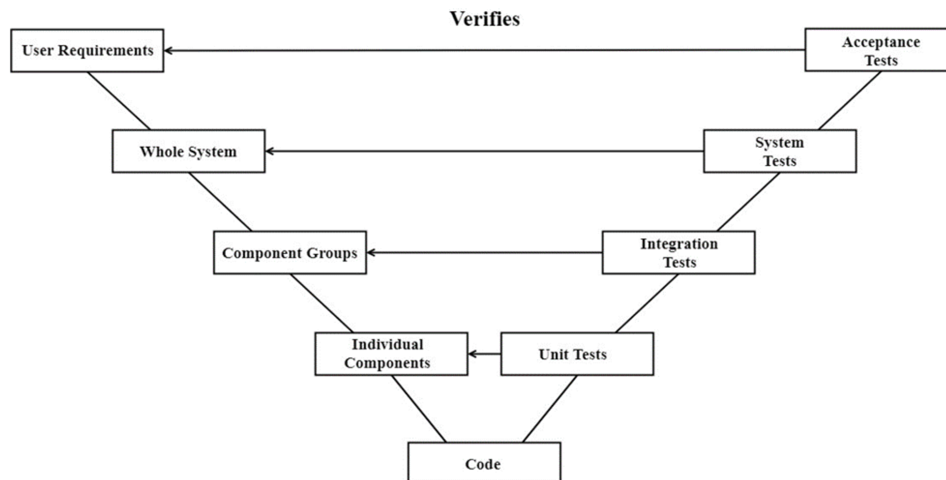


Figure 22. The four levels of software testing and the part of the code they are focused on

6.8.1.1 ROS Levels of Testing

The official ROS documentation [57] recommends a modified version of the testing levels presented in *Chapter 6.8.1*. The following text presents a summary of those recommendations.

The ROS system is comprised of a number of independent nodes, each of which communicates with the other nodes using a publish/subscribe messaging model [57].

The recommended testing tools are:

- *unittest* [61] for testing Python code at library level,
- *gtest* [62] for testing C++ code at library level, and
- *rostest* [63] for testing one or more nodes (in combination with unittest or gtest).

ROS services need to be tested at the levels 1 and 2. The level 2 tests are used for discovering threading or performance issues [57].

ROS nodes need to be tested at all three levels. Level 1 will test the underlying functionality. Level 2 tests the functionality at the ROS level and level 3 attempts to uncover race and deadlock conditions [57].

The testing of software that uses ROS should be separated into three levels (Figure 23):

- *Level 1: Library unit test.* This level includes all unit test for libraries that do not include ROS (for example simple math routine). The testing is most commonly done by using either unittest or gtest. If the unit tests cannot be written in a way that does not include ROS, then the code needs to be refactored. All functions must be well separated and tested for various conditions including all common, edge and error inputs. The behavior of the tested code should be well documented.
- *Level 2*: *ROS node unit test.* This level includes all node unit tests. The tests start the nodes and test its external API, i.e., published topics, subscribed topics, and services. The testing is most commonly done by using rostest in combination with either unittest or gtest.
- *Level 3: ROS nodes integration test.* Integration tests are used for testing whether multiple nodes work together as intended. An integration test is often the best way of uncovering bugs [57], deadlocks and race conditions. At the lowest level of integration tests, not many nodes will be tested so mock objects need to be implemented. As more and more nodes are included in the tests and the complexity of the tests grows, simulation or recorded real-life data should be used as input because mocking all inputs is not feasible. Tests continue until all nodes of the system are up and running. The tests on this level are implanted using a combination of rostest with either unittest or gtest.

For multiple reasons it is recommended that the regression tests are implemented on the lowest possible level:

- First, lower level tests are less time consuming during execution.
- Second, since lower level tests test a small section of the code, it is easier to localize which part of the code causes the bug.

- Finally, compared to higher level tests, lower level tests have fewer dependencies, so they are easier to maintain.

It is recommended that the tests are automated and integrated into the project scripts. The technical details about the implementation and the integration are provided later in this thesis.

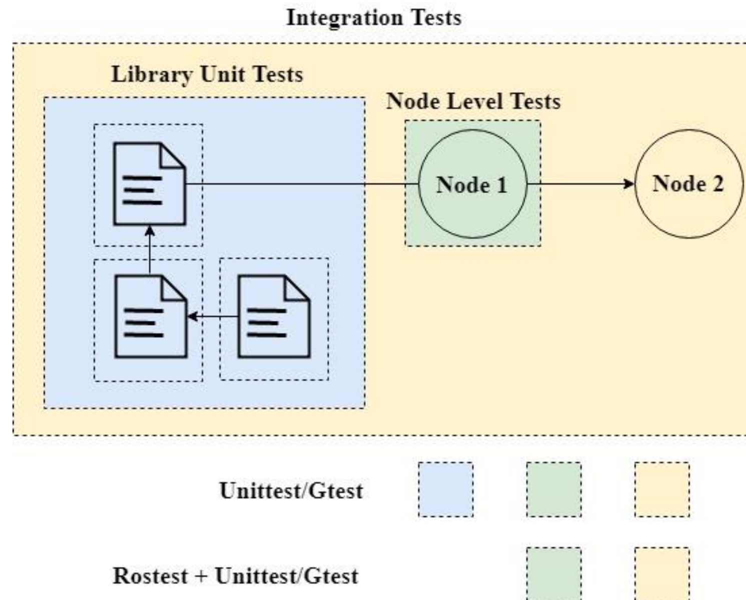


Figure 23. ROS Levels of Testing; Adapted from [60]

6.8.2 Types of Testing

The two most important testing techniques are white-box and black-box testing [58, 59]. *White-box testing* consists of testing the internal parts and logic of the system [58]. Different techniques of white-box testing are [59]:

1. Static white-box testing
   a. Desk checking
   b. Code walkthrough
   c. Formal Inspections
2. Structural white-box testing
   a. Control flow/ Coverage testing
      - Statement coverage
      - Branch coverage
      - Decision/Condition Coverage
      - Function Coverage
   b. Basic Path testing
      - Flow Graph Notation
      - Cyclomatic Complexity

- Deriving Test Cases
- Graph Matrices

c. Loop testing
- Simple Loops
- Nested Loops
- Concatenated Loops
- Unstructured Loops

d. Data flow testing

Static white-box testing is the process of reviewing the code in order to confirm whether the code fulfills all the requirements [59]. It also used to find faults. Structural white-box testing focuses on testing the internal implementation or the structure of the software [59]. The white-box testing techniques are briefly described in the following text [59]:

Desk checking: is an informal review where the code developer distributes his code to other developers for comments and review.

Code walkthrough: is an informal review meeting where the developers review each other's code for mistakes.

Formal inspection: is a formal review meeting, guided by a moderator, where the developers review each other's code for mistakes.

Control Flow/ Coverage Testing: It is a testing technique that uses the control structure of the program for developing the test cases. The test cases should cover the control structure of the program in a sufficient manner. The coverage can be measured based on the number of the executed statements, branches, functions, and the true or false outcomes of each Boolean expression (Decision coverage).

Basic Path Testing: It is a testing technique that derives the test cases by analyzing the control flows of the code. Basis path testing makes use of the Cyclomatic complexity measure and flows graphs to identify all independent paths.

Loop Testing: It is used to test all types of loops in the code (Appendix II).

Data Flow Testing: This testing technique focuses on the data variable and their values, i.e., the moment when the variables receive values and how and when these values are being used.

Black-box testing consists of testing whether the system fulfills the requirements from the client point of view [58]. During black-box testing, the tester does not have to have any knowledge of the internal logic of the system. It is also known as external testing, functional testing or specification-based testing [58]. The black box testing techniques are briefly described in the following text [58, 59]:

Equivalence Class Partitioning and Boundary Value Analysis: It is used for reducing the number of test cases by splitting the input range of values in smaller ranges and selecting one input value from each range.

Boundary Value Analysis: The analysis of the extreme lower and upper ends of the input values is known as Boundary Value Analysis. Additionally, error, empty, null, none and typical values are also tested.

Decision Tables: It is a cause-effect table used for designing test cases which represents the effects of combining different inputs.

State Transition Testing: It is a testing technique where the system under test is presented as a Finite State Machine. The testing of the system is based on checking whether the transition from one state to another is done in a correct manner.

Orthogonal Arrays: It is used when the number of inputs is small but too complex for exhaustive testing. Then combining the data to lower the number of test cases is beneficial.

All Pairs Technique: It is used for testing all possible discrete combinations of each pair of input values.

6.8.3 Prioritizing tests

Due to limited resources such as time, money or staff, it is not possible to test all units of a software system. Authors [50, 51] recommend prioritizing based on risk and test parts of the system where the most critical failures can occur. In addition to prioritizing based on risk it is also recommended to prioritize parts of the system that are more commonly used [50, 51]. The decision whether the unit will be tested independently or as a component of a larger system depended on the complexity of the unit and the amount of effort associated with mocking all the dependencies.

It is recommended to use both white-box and black-box testing techniques during testing [50, 58, 59]. However, if the company has limited resources, it is recommended to use black-box techniques instead of white-box techniques [51, 59].

A higher focus in Pro-Drone was put on developing end-to-end system tests since they offered the highest trade-off between confidence and expense.

Test coverage was used as a measure of how much testing has been performed. The test coverage was provided using GitLab Page code coverage tool.

7 Implementation of Secure Software Supply Pipeline

This chapter covers the practical implementation of the testing and release pipeline, focusing on its technical aspects. First, an overview of the implemented tests is provided, followed by a description of the implemented sensory check algorithms that are used to convey important flight and safery information to the UAV operator. Furthermore, the Log Analyzers used for initial analyzing of the inspection data are described. Finally, the overall pipeline architecture is discussed and its safety aspects are briefly covered.

7.1 Test Implementation

The requirements for UAV's safety and reliability can be guaranteed by creating comprehensive tests. As described previously, conducting these tests in real life is expensive and time-consuming, so an automated testing pipeline is needed. In this thesis:

- Level 1 and Level 2 tests were implemented for the sensory check algorithms described in Chapter 7.1.3.2, covering most of the logic behind the lidar, camera, gimbal and autopilot code. Monitoring and recording memory usage and CPU load was also implemented and used during performance testing.
- The main focus was put on Level 3 tests where SITL tests, as well as tests based on data playback, were used for testing newly developed algorithms. These tests were implemented so that the environment can be simulated as closely as possible. The real life inspection data was provided using a ROS tool called rosbag which can record and replay data from topics. The simulation dataset contained 80 hours of inspection data with a data volume of 15.8 GB in the form of 93 rosbags.
- In order to increase operation safety, a notification panel was added in the GCS GUI, which was used to notify the operator about the state of the UAV. The created states were used during the Level 3 tests.
- Log Analyzers used for quick initial debugging were implemented.
- Guidelines for all three testing levels were created.

The used software development tools are:

C++ programming language. Since the software was developed using C++, C++ 11 alongside Gtest were used for implementing Level 1 and Level 2 tests. Additionally, the logic behind the sensory check algorithms was implemented using C++ 11.

Python programming language. Level 3 tests were implemented using Python 3.4. The motivation for choosing behind was the concept of Python dictionaries which could be used for simple and

easy to understand implementation of the state-transition pattern needed for the Finite State Machines. Additionally, the Log Analyzers were also implemented using Python.

In addition, the tools covered in Chapter 4.1 were used during the development process. The detailed description of each testing level is given in the following subsections.

7.1.1 Level 1 Tests

The testing process starts with writing Level 1 tests which are used to test the behavior of individual units of the software that does not require ROS. The purpose is to validate that each software unit performs as designed. Software unit is a piece of code which cannot be further decomposed.

According to the recommendations in Chapter 6.8.1, Unit tests in Pro-Drone were written using the Google C++ Test Framework (gtest).

A minimum working example of a unit test using gtest (based on [64]), for a square root function follows. The function can be declared as:

```
double squareRoot (const double);
```

For negative numbers, this routine returns -1. For positive numbers, it returns the square root of the number.

This function is implemented in the following files:

```
prodrone/math_functions/include/SquareRoot.h
prodrone/math_functions/src/SquareRoot.cpp
```

The unit test example is shown in Listing 3. The two major components in the unit test script are:

- test cases set up, and
- run test cases.

First, a test hierarchy SquareRootTest, which contains two unit tests PositiveNum and ZeroAndNegativeNum, is created. TEST is a predefined macro from gtest that defines the hierarchy [64]. EXPECT_EQ and ASSERT_EQ are macros that resemble function calls [62]. ASSERT generates fatal failures, which abort the test. EXPECT generates nonfatal failures, which do not abort the test and additionally shows in how many test cases the square root function fails. For example, the following line checks whether the square root of 324.0 is calculated correctly as 18.0:

```
EXPECT_EQ (18.0, square-root (324.0));
```

::testing::InitGoogleTest sets up the environment and must be called before RUN_ALL_TESTS. RUN_ALL_TESTS macro should be called only once, and it runs all tests that were previously defined, outputs the results and returns 0 on success.

Instead of TEST, a TEST_F Fixture can be used. TEST_F repeatedly creates an isolating environment for each individual test.

Listing 3. Unit test test/SquareRootTest.cpp for the square root function

```
#include "SqaureRoot.h"
#include "gtest/gtest.h"

TEST (SquareRootTest, PositiveNum) {
    EXPECT_EQ (18.0, square-root (324.0));
    EXPECT_EQ (25.4, square-root (645.16));
    EXPECT_EQ (50.3321, square-root (2533.310224));
}

TEST (SquareRootTest, ZeroAndNegativeNum) {
    ASSERT_EQ (0.0, square-root (0.0));
    ASSERT_EQ (-1, square-root (-22.0));
}

// Run all tests that were declared with TEST()
int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

The tests need to be added to the CMakeLists[7]:

```
rosbuild_add_gtest(test/SquareRootTest test/SquareRootTest.cpp)

rosbuild_add_library(SquareRootLib src/SquareRoot.cpp)
target_link_libraries(test/SquareRootTest SquareRootLib)
```

The tests can be run with the command:

```
catkin_make run_tests
```

By default, the results are printed to the standard terminal output. Additionally, a XML-file containing the results can be found in the build folder of the workspace.

Guidelines for documenting test cases, and reporting bugs are provided in Appendix III.

7.1.2 Level 2 Tests

Integration tests are handled through rostest which uses the roslaunch XML description [63]. Rostest is a wrapper around roslaunch that enables roslaunch files to be used as test fixtures [63]. Any behavior that requires ROS functionality should be tested using rostest. With rostest, the gtest test files are launched as ROS nodes alongside any other nodes which are specified using the tag <node> in the launch files. The tag <test> can be added anywhere inside a standard ROS launch

[7] CMakeLists, http://wiki.ros.org/catkin/CMakeLists.txt

file to add additional nodes that are to be launched specifically for the test. This tag is reserved for rostest and is ignored by roslaunch. Within the test suites, a unit testing framework like gtest is used to report to rostest the results of the test. With rostest, it is possible to:

- send messages to a node and test how it responds,
- subscribe to messages from a node,
- use a node's services.

A minimum working example with the following package structure is provided:

Listing 4. Package structure

```
simple_rostest
├── CMakeLists.txt
└── src
    └── add_two_ints_service.cpp
    srv
    └── AddTwoInts.srv
    test
    ├── add_two_ints_service_client.cpp
    └── add_two_ints_service_client.launch
```

Table 7 contains the description of the Listing 4's content. The following minimum working example is based on the ROS Service tutorials [65] and the example provided by [66].

Table 7. Directories in the project

Directory	Content
src	Service implementation
srv	Service definition
test	Test node and corresponding launch file

The test directory contains the implemented test and the launch file. The service definition is shown in Listing 5. Listing 6 contains the content of the launch file. The node "add_two_ints_service" is created which receives two integers and returns their sum (Listing 7).

Listing 5. simple_rostest/srv/AddTwoInts.srv

```
int64 a
int64 b
---
int64 sum
```

Listing 6. simple_rostest/test/add_two_ints_service_client.launch

```
<?xml version="1.0"?>
<launch>
  <node pkg="simple_rostest" name="add_two_ints_service"  type="add_two_ints_service"
/>
  <test pkg="simple_rostest" test-name="add_two_ints_service_client"
type="add_two_ints_service_client" />
</launch>
```

Finally, the tests need to added to CMakeLists (Listing 9).

Listing 7. simple_rostest/src/add_two_ints_service.cpp

```
#include <ros/ros.h>
#include <simple_rostest/AddTwoInts.h>

bool AddTwoInts(simple_rostest::AddTwoIntsRequest &req,
                simple_rostest::AddTwoIntsResponse &res)
{
  res.sum = req.a + req.b;
  return true;
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "add_two_ints_srv");
  ros::NodeHandle nh;

  ros::AsyncSpinner spinner(1);
  spinner.start();

  // Advertise service
  ros::ServiceServer service = nh.advertiseService("add_two_ints", AddTwoInts);

  ros::waitForShutdown();
  spinner.stop();
  return 0;
}
```

Listing 8. simple_rostest/test/add_two_ints_service_client.cpp

```
#include <ros/ros.h>
#include <ros/service_client.h>
#include <gtest/gtest.h>
#include <simple_rostest/AddTwoInts.h>

std::shared_ptr<ros::NodeHandle> nh;

TEST(TESTSuite, addTwoInts)
{
  ros::ServiceClient client = nh->serviceClient<simple_rostest::AddTwoInts>(
      "add_two_ints");
  bool exists(client.waitForExistence(ros::Duration(1)));
  EXPECT_TRUE(exists);

  simple_rostest::AddTwoInts srv;
  srv.request.a = 5;
  srv.request.b = 8;
  client.call(srv);

  EXPECT_EQ(srv.response.sum, srv.request.a + srv.request.b);
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_service_client");
  nh.reset(new ros::NodeHandle);
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Listing 9. simple_rostest/CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
project(simple_rostest)
add_compile_options(-std=c++11 -Wall -Wextra)

find_package(catkin REQUIRED COMPONENTS
  genmsg
  roscpp
  std_msgs
)
################################################
## Declare ROS messages, services and actions ##
################################################
add_service_files(
  DIRECTORY
  srv
  FILES
  AddTwoInts.srv
)

generate_messages(
  DEPENDENCIES
  std_msgs
)
###################################
## catkin specific configuration ##
###################################
catkin_package(
  CATKIN_DEPENDS
  std_msgs
)
###########
## Build ##
###########
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)

add_executable(
  add_two_ints_service
  src/add_two_ints_service.cpp
)
target_link_libraries(
  add_two_ints_service
  ${catkin_LIBRARIES}
)
add_dependencies(
  add_two_ints_service
  ${simple_rostest_EXPORTED_TARGETS}
)
#############
## Testing ##
#############
if(CATKIN_ENABLE_TESTING)
  find_package(rostest REQUIRED)

  add_rostest_gtest(
  add_two_ints_service_client
  test/add_two_ints_service_client.launch
  test/add_two_ints_service_client.cpp
  )
  target_link_libraries(
  add_two_ints_service_client
  ${catkin_LIBRARIES}
  )
```

```
  add_dependencies(
  add_two_ints_service_client
  add_two_ints_service
  ${PROJECT_NAME}_generate_messages_cpp
  ${catkin_EXPORTED_TARGETS}
  )
endif()
```

The tests are executed with the following command:

```
catkin_make run_tests
```

A brief overview of the test results is displayed in the terminal after the test finish. A detailed test result is provided is an XML-file in the user's .ros-folder.

7.1.2.1 Mocking Interfaces

Mock is an object that substitutes the real object by simulating its behavior in a controlled manner. Mockups are necessary because:

- it is hard, and sometimes even impossible to get the right value at the right moment during the test.
- the dependency might be buggy, and it might affect the test.

A minimum working example in which a fake Lidar data publisher is created for testing the interaction between the Lidar class and the State watcher is provided in Appendix IV. The test checks whether:

- the State watcher publishes a state when the Lidar is connected and publishes correct data,
- the Lidar is not connected, and
- when the Lidar is connected but blocked (the lidar is covered, its surface is dirty or it publishes incorrect data).

7.1.3 Level 3 Tests

The end-to-end System tests were based on a black-box technique called a Finite State Machine (FSM). FSMs were used to represent and control the test execution flow. FSMs are defined by their states and transitions, which can be represented by using a graph, where the nodes are the states, and the edges are the transitions [67]. Each edge has a label that defines the conditions that need to be fulfilled for a transition to happen (Figure 24).

For this tests, it was assumed that all moments of interest during the inspection flight could be aggregated and discretized. The moments of interest are called states, and the machine can be in exactly one from a finite number of states at any given moment. The testing environment will go from one to another state only when certain conditions are met. The change from one state to another in response to an external input is called a transition.

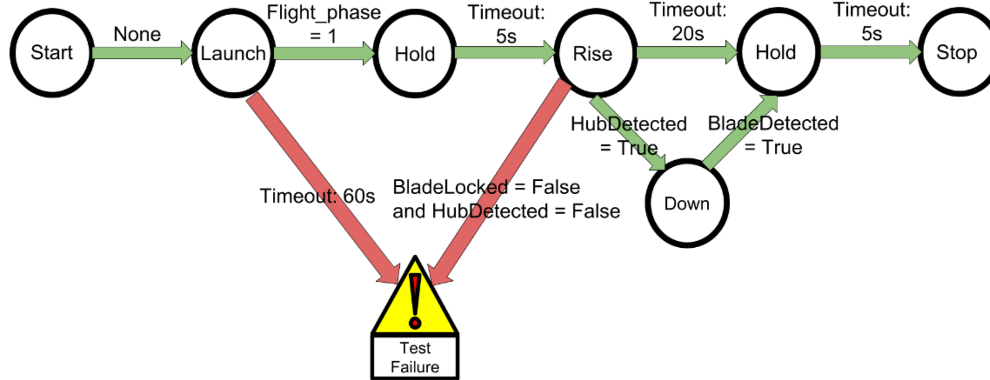


Figure 24. FSM diagram

Figure 24 represents the building blocks that make up a small test. The starting point is the *Start* state. Since the transition condition is set to None the current state is immediately transitioned to *Launch*. If the Flight phase is set to 1 in less than 60 seconds, the current state will transition to *Hold*. Otherwise, a general failure will be activated, and the test will end with failure. If no failure state is activated the state transitioning proceeds in the same manner until the *Stop* state is reached. If the *Stop* state is reached the test ends successfully. It is possible to define only one Start and Stop state. The flow of the system tests is based on the flight phases described in Chapter 4.2, and usually, a check is made whether all phases were finished successfully.

In total 9 SITL tests and 25 rosbag tests were implemented (10 were based on data gathered during inspection with the DJI autopilot and 15 with PX4). These rosbag were chosen in a way that maximizes the coverage of different scenarios (different wind parks, type of wind turbines, weather scenarios, type of the used autopilot etc.).

Figure 25 represents the architecture of the FSM package.

7.1.3.1 Test Creation

Depending on the type of the input data two types of FSM test exist:

- rosbag tests, and
- SITL tests.

Both of this tests follow the same workflow.

Rosbags were used to create a timeline to check whether the control code is able to handle the input generated by the rosbag file or the simulation. Initially, the tests were created by playing rosbag data from different inspections and taking notes on the important moments. Then the test was set to check whether the system is reaching the correct state in a correct time moment. Afterward, a rosbag analyzer script, located in the “tools” directory of the FSM package, was created. It can

export the times when the predefined moments of interest occurred. The script can also filter out topics that are not necessary for the test. Finally, the generated output presented in Listing 10 is parsed to its final form shown in Appendix V using the “rosbag_script_to_fsm_.py” helper script. Additional modification and check can be later added to this script. Using this, multiple tests are created, and they are stored in the “test_scripts” directory.

During the automated testing phase, the content of each test file stored in “test_scripts” directory is copied in “fsm.py” and then executed.

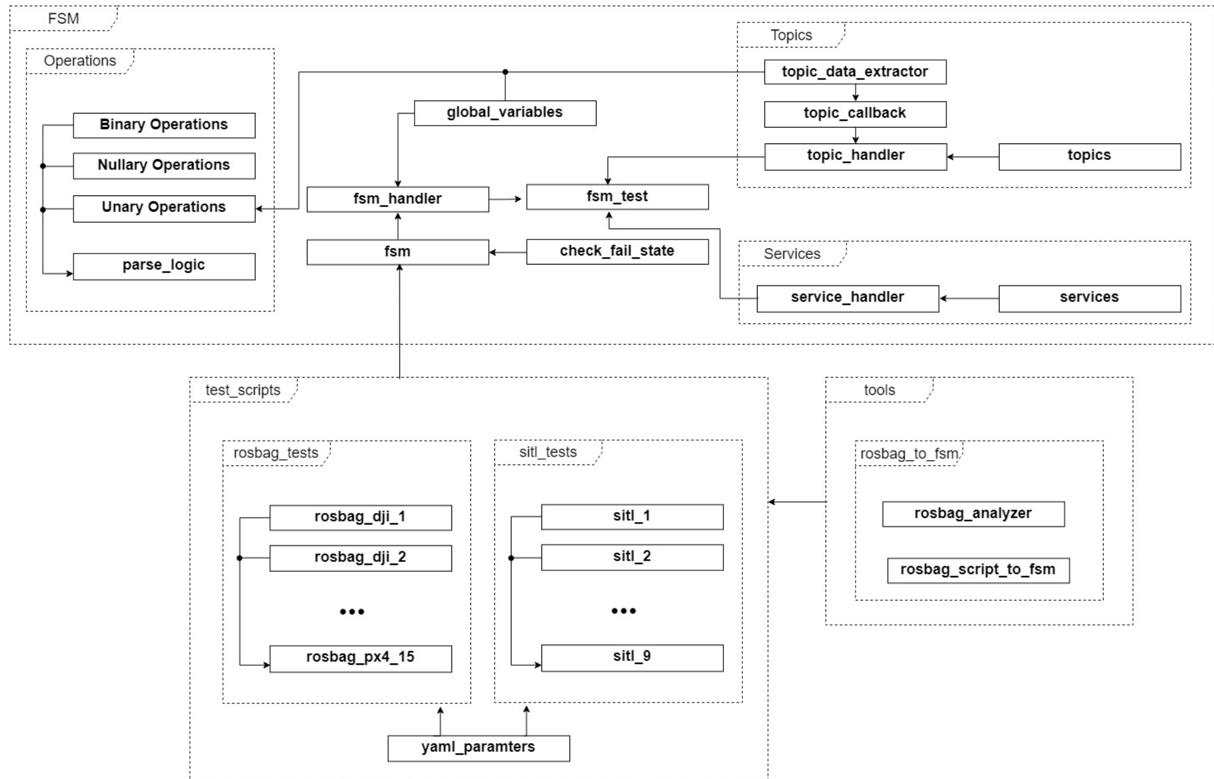


Figure 25. Architecture of the FSM package

Listing 10. rosbag tool output

```
rosbag_conditions = [
      #time #command
      [20, {'command': ["SRV_unlock_tower",""]}], #command
      #time #failure condition
      [60, ["T_monitor_data.blade_locked", "=", False]], #momentary check
      #start time #stop time  #failure condition
      [100, 120, ["T_monitor_data.blade_locked", "=", False] ]  #span
                        ]
```

The test script in Appendix V contains the scenario of the test, and it defines the:

1. the flow of the test,
2. which parts of the control code are tested, and

3. under what conditions the test is considered a failure.

The testing script is separated in the following steps:

1. First, each state is defined by specifying its name and a list of commands that need to be sent during the loading of the state.
2. Then the states are linked between each other. The names of the two states that need to be linked and the condition under which the state should change are defined.
3. Afterward, the failure conditions are specified. There are two types of failure conditions: state-dependent failures and state-independent failures. The first type is a failure linked to a specified state, which was previously defined in step 1. It occurs when the transition conditions are not fulfilled or when a failure condition occurs during this specific state. For this failure, the same formatting of conditions as for the transitions is used. The second type of failure is not linked to any state that was defined in step 1. In this case, the test fails if at any given moment an invalid GCS state is published by the GCS State Watcher. The GCS State Watcher and the states which are used during the state-independent failure check are described in detail in Chapter 7.1.3.2.
4. Lastly, a timeout is set in order to prevent infinite loops. The triggering of the timeout is considered as a test failure. The timeout can be disabled by simply putting its value to zero.

7.1.3.1.1 FSM States

States are moments of interest within the test. When creating a new state, a unique name for the state and the set of commands to send when the state is loaded needs to be provided. In practice, this means adding an entry to the dictionary “state_link.” Here the key is the name of the entry and the content is an array of commands.

The following formatting is used:

```
'state name':   [List of commands]
```

If no commands are given, the list of commands is initialized with None:

```
'state name':   None
```

The control commands are service calls which are defined in “services.py”.

A command consists out of two parts:

1. *Service name*: This is the name of the service which needs to be called.
2. *Payload*: The parameters that need to be passed along over the service call.

The following formatting is used:

```
["Service name",    ("Payload")]
```

If multiple commands need to be sent, they can be appended in an array:

```
[
      ["service 1", ("payload 1")],
      ["service 2", ("payload 2")]
]
```

7.1.3.1.2 Transitions

After the states are defined, they are linked together using transitions. A transition is defined by a source and a destination state and the conditions under which the transition should occur. The transition will happen only when the condition statement is a True, and refrain from transitioning when the condition is a False.

The transitions have the following format:

```
["source state",    "destination state",    Condition]
```

For an immediate transition between states the condition can be replaced with None:

```
["source state",    "destination state",    None]
```

A detailed explanation of the implemented conditions follows in Chapter 7.1.3.3.

7.1.3.1.3 Failures

There are two types of failures: state-dependent failures and state-independent failures.

State-dependent failures. A failure can be seen as another transition, where a source state, as well as the condition under which a failure should occur is defined. The destination state is always a general failure state where, when reached, the simulation will stop, and the test will be considered as a failure.

To add a new failure to a state a new entry to the “state_fail” dictionary needs to be added. Here the key is the source state name and the value is the condition array. This is achieved by using the following format:

```
'source state': conditions
```

State-independent failures. The state-independent failures occur if at any moment a specific GCS state is being published. To add this type of failure, a new entry in the “check_state_fail” dictionary needs to be added. This is achieved by using the following format:

```
global check_state_fail
check_state_fail = {
                    'CheckState': [lambda: fail_if_state_active('SafetyStop',
                                         get_field("T_states_data.states"))]
}
```

From the previous example, the test will fail if at any given moment the “SafetyStop” state is active.

7.1.3.1.4 Conditions

Conditions are used to change the state the simulation is into either the next state or a failure. These conditions can contain different sets of operations but always have to adhere to a certain set of rules. At the lowest level, the data-type should be a boolean, returning logical True if the conditions are met, or False if not. The hierarchy of operations always has to be met. This means that it has to be clearly defined which operation has to be executed first.

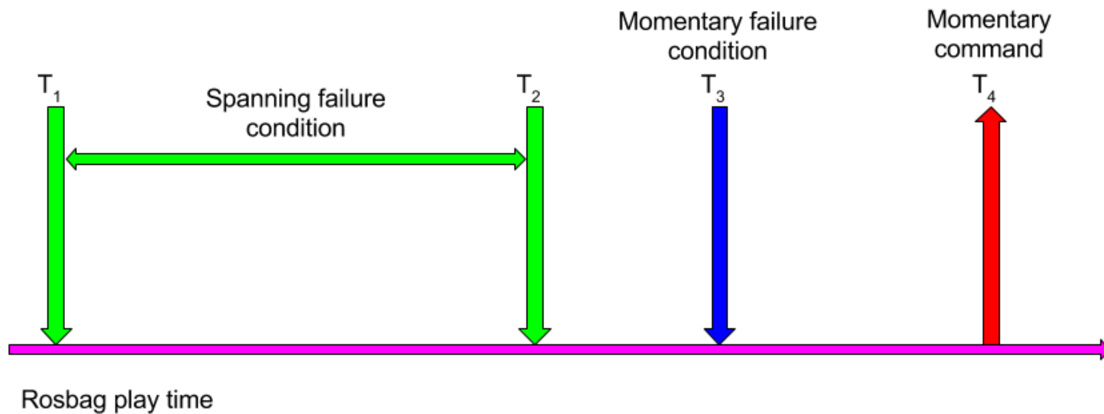


Figure 26. Visualization of the condition check (equivalently the condition can be set to change state instead of failing the test)

Figure 26 shows that the condition check can be done in two ways:

1. *Spanning check*: During this time the test will fail if the specific condition is met.
2. *Momentary check*: The test will check on that specific moment if the condition is met and if so, fail the test.

Additionally, a momentary command (for example lock the blade) can be sent.

For example, let's consider the following example of a condition:

```
[["T_monitor_data.flight_phase", "=", 1], '&', ["T_monitor_data.position_z",
"<=", [["LOAD_VAR", {"alt1": None}], '*', 1.2]]]
```

Figure 27 provides a hierarchical representation of the previous example.

This diagram can be approached in a top-down manner.

1. The first operation is the "LOAD_VAR". Here a previously saved variable "alt1" is loaded.
2. After the variable is loaded, it is multiplied by the constant 1.2.
3. When this multiplication is finished, the resulting value is compared with the latest value "position_z" from the topic "monitor_data" to check if it's greater or equal.
4. Now the left branch is analyzed. The latest value of "flight_phase" from the topic "monitor_data" is accessed, and a check is made if it equals to one.
5. The results from the left and the right branch are connected to a logic AND gate.

6. The resulting AND operation will give either a logic True or False. If the resulting value is True, the condition will be met, and either a transition or a failure will be triggered. If the resulting value is False, the condition will not be met, and the state of the simulation will remain the same.

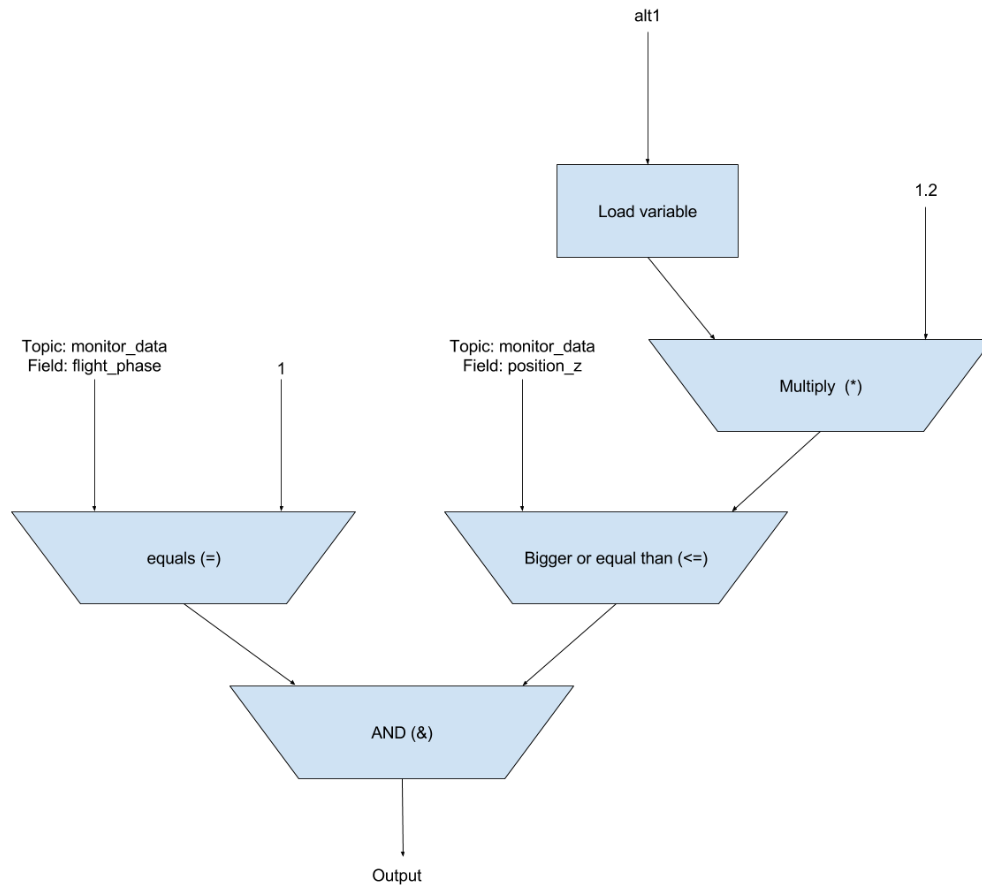


Figure 27. A hierarchical representation of a transition condition

7.1.3.2 GCS States and Messages

The GCS states, alongside with the FSM states are used to check for general failure in the CI. Unlike the FSM states, the check for the GCS is done continuously, and the test fails if one of the states is set active at any given moment. Initially the GCS States were created as an easy and intuitive way of notifying the operator about possible issues with the system. The UAV operator is usually not required to have knowledge about the internal logic of the system. For this purpose, the existing GCS GUI was modified to display the relevant safety information to the user (Figure 28). Later on the GCS States were extended (states 7 and 11 through 19 (Table 8) were added) and incorporated as part of the testing pipeline. The GCS states are described in Table 8. Table 9

contains a summary of the priority codes used in Table 10. The warning and information messages displayed in the GUI are shown in Table 10.

The GCS is notified about the current state of the UAV using the "StateWatcher," (Figure 29). Using the sensor topic data, the "StateVariablePublisher" publishes a "State Variable" (indicating, for example, the battery voltage level). The "StateWatcher" subscribes to the data published by the "StateVariablePublisher," and if certain conditions are met, it publishes a "StateGCSArray" message. Sample code is provided in Appendix VI and Appendix VII. The information/warning messages are used to notify the GCS user by using the GUI.

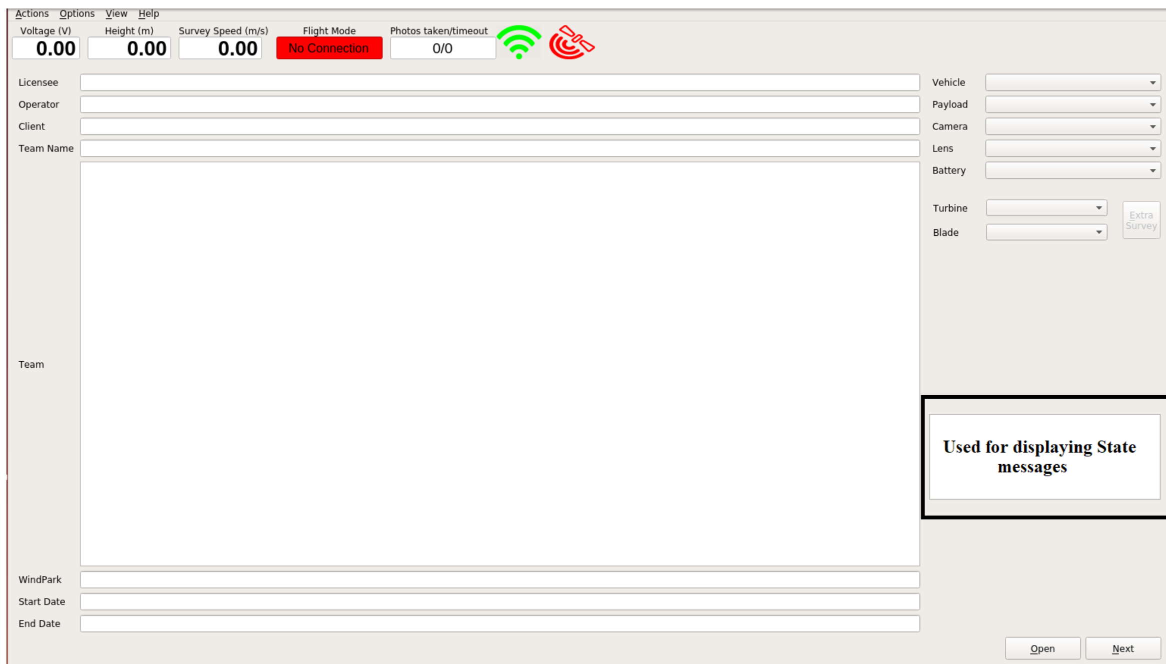


Figure 28. GCS GUI

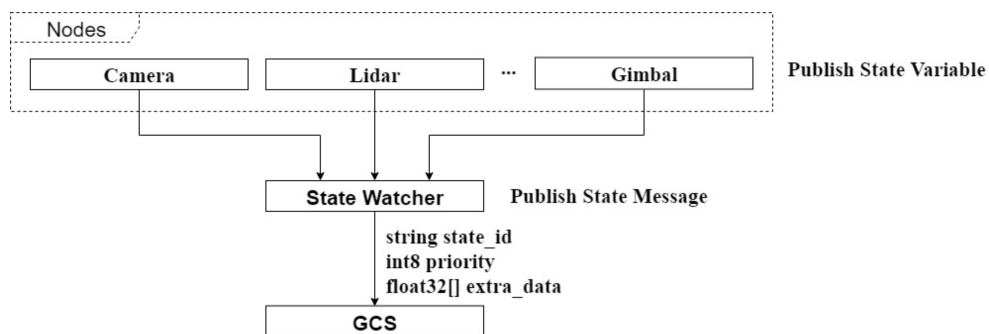


Figure 29. GCS station state messages

Table 8. States definitions

	State	Definition
1.	TowerIsLocking	The state is active if the set labeler is InitialTowerLock (tower is not initially locked) and within 2 seconds at least one body was being locked as a tower.
2.	TowerIsRelocking	The state is active if the set labeler is TowerRelock (tower is initially locked, blade is locked, but tower is not visible) and within 2 seconds at least one body was being relocked as a tower).
3.	BladeIsInitiallyLocking	The state is active if the set labeler is BladeInitialLock (tower is initially locked, blade is not locked, and within 2 seconds at least one body was being locked as a blade).
4.	BladeIsRelocking	The state is active if the set labeler is BladeRelockSet (tower is initially locked, blade was initially locked, and within 2 seconds at least one body was being relocked as a blade).
5.	SafetyStop	Is active when the SafetyStop Mode is on, and the flight mode is Autonomous.
6.	InitialLockNumberOfBodies	Is active if the set labelers are TowerInitialLockSet or BladeInitialLockSet and more than two visible bodies for 2 seconds.
7.	BladeIsLost	Blade was initially locked and was lost when ready for the blade tip wasn't set.
8.	SetpointOutOfFence	Setpoint is out of SetpointFence for more than 1 second.
9.	ReadyForBladeTip	ReadyForBladeTip is set.
10.	ReadyForAutonomous	This message is active when Blade is locked; Tower is locked and visible, the flight phase is ApprachingTheBlade.
11.	TowerNotVisibleAfterRotation	This state is active if, after rotation from LE to left or right side, the tower is not visible, thus the FlightPhase can't be changed. The user at this point should manually fly closer to tower to relock it and then proceed with normal operation.
12.	PackingSurveyData	Is active when the data on the UAV is being prepared to send to GCS.
13.	GimbalState	Returns 0 if the calibration failed or 1 if the gimbal is booting. Calibration fails if the gimbal is booting longer than 20 seconds.
14.	GimbalAngleLimits	Is active when (roll > 0.4 and roll < -0.4) or (pitch > 0.3 and pitch < -0.3).
15.	NoDataReceivedFromGimbal	Is active when no data is received from the gimbal.
16.	LidarBlocked	This state is active when the lidar is blocked, or its surface is dirty.
17.	NoDataReceivedFromLidar	This state is active when no data is received from the lidar.

18.	BatteryVoltageLow	Active when battery's voltage is too low.
19.	NoAutopilotConnection	Active when there is no connection with the autopilot.

Table 9. Priority codes used in Table 10

Priorities	
Code	**Meaning**
1	Critical (Land immediately)
2	Warning (Be extra careful)
3	Info (Useful info)

Table 10. List of states and messages

State name	Extra data	Priority	Message
TowerIsInitiallyLocking	0	3	Tower is initially locking
TowerIsRelocking	0	3	Tower is relocking
BladeIsInitiallyLocking	0	3	Blade is initially locking
BladeIsRelocking	0	2	Blade is relocking. Extra attention advised.
SafetyStop	0	2	Safety stop is on. Take manual control.
InitialLockNumberOfBodies	1	2	Only 2 bodies must be visible for initial lock, visible:
BladeIsLost	0	2	Blade is lost! Blade relock is required
SetpointOutOfFence	0	2	Set point is too far from the UAV
ReadyForBladeTip	0	3	Approaching the blade's tip
ReadyForAutonomous	0	3	Ready to go Autonomous mode
TowerNotVisibleAfterRotation	0	2	Tower in not visible - take manual control and relock tower
PackingSurveyData	0	3	Packing data
ReadyForBladeTipWhileGoingUp	0	2	BladeTip button is set for more than 15 seconds. Unset if it's not required.
GimbalState	1	3	Gimbal status:
GimbalAngleLimits	0	3	Pitch or Roll are out of range
NoDataReceivedFromGimbal	0	3	No data is received from the gimbal
LidarBlocked	0	2	Lidar is blocked
NoDataReceivedFromLidar	0	2	No data is received from the lidar
BatteryVoltageLow	0	1	Battery voltage is too low
NoAutopilotConnection	0	2	No autopilot connection

7.1.3.3 Operations

Operations are functions that are executed within the conditions. This can be any Python function which can have full access to any resource. The restrictions are that the function must return a True if the transition or failure should be triggered, and the function should be non-blocking as it can interfere with the checking cycle. The function is linked to a string that describes its operation and is used in the conditions of the FSM.

The operations are split into different files depending on the number of input variables. This is needed as the input ordering of these variables is different in certain circumstances.

Nullary operations do not require any input variable. They are often used for user-defined operations that handle variables internally. They use the following format:

```
["operation name"]
```

Unary operations require one input variable. They are often used for operations like saving, loading data from topics or other sources, timeouts and so on. They use the following format:

```
["operation name", var]
```

The implemented unary operations are shown in Table 11.

Table 11. Implemented unary operations

Unary operations	
NOT	Logic NOT-operation
STATE_RUNTIME	Check if the time the current state has run is more or less than the argument given.
SIMULATION_RUNTIME	Check if the time the node has run is more or less than the argument given.
SAVE_VAR	Save a set of variables with certain keys.
LOAD_VAR	Load a previously saved variable with a specific key.
STATE_MSG_CHECK	Checks if the state messages from drone are the same as the ones declared in the SITL test script. It is used in fail_state.
TO_LOCK_BLADE	Checks if a body has been recognized as a blade and returns true, so the `SRV_lock_blade` can follow without error.

For example, the SAVE_VAR operation uses the following format:

```
"SAVE_VAR", {"altitude 1": "T_monitor_data.altitude"}
"SAVE_VAR", {"my altitude": 45}
```

or

```
"SAVE_VAR", {"altitude 1": "T_monitor_data.altitude",
             "my altitude": 45}
```

You can save both topics and variables as the function will automatically parse them. Data is inputted as a dictionary, and multiple entries are allowed in the same function.

The LOAD operation uses the following format:

```
"LOAD_VAR", {"altitude 1", None}
```

Here the specific variable is loaded. Only one key is expected with “None” as content. The function will return the loaded variable.

Binary operations require two input variables. They are often used for comparing variables or doing an operation between them. They use the following formatting:

```
[var1, "operation name", var2]
```

where the middle part is always the instruction while the first and second parts are always the arguments. The implemented binary operations are shown in Table 12.

Table 12. Implemented binary operations

Binary Operations	
=	Equals
!=	Not equals
>	Bigger than
>=	Bigger than or equals
<	Smaller than
<=	Smaller than or equals
&	Logic AND-operation
|	Logic OR-operation
+	Numeric sum
-	Numeric subtraction
/	Numeric division
*	Numeric multiplication

7.1.3.4 Resources

It is possible to access several external resources. Currently, these are:

- ROS topic data
- ROS service calls
- Global variables

7.1.3.4.1 ROS Topics

Topics are the main source of data coming into the system. The topics that are used as external input in the system are configured and specified in the “topics.py” file.

The configuration file has the following structure:

```
"my topic name": {"name":"the ros name", "type":type of the topic}
```

To use the ROS topic the configuration file must contain:

- a name which will be used in the tests (any name as long as it doesn't have a "." in it.),
- the name ROS uses for the topic,
- and the type of the topic.

An example showing the configuration of the topic /prodrone/monitor_data follows:

```
from prodrone.msg import Monitor_data
global topics

topic = {
          "T_monitor_data": {"name": "/prodrone/monitor_data", "type":
                             Monitor_data}
        }
```

When data is received on a topic, a global variable is updated with the content of this topic. This data can be accessed and used for the transition and failure conditions. The data is accessed using a formatted string compromised of the name provided in the configuration file and optionally a specific field using "." as a delimiter. Using this format multiple levels (or fields) can be accessed.

The formatting of the string is as follows:

```
"My topic name.field_level1.field_level2. ..."
```

or in case of an actual topic:

```
"T_monitor_data.altitude" #returns the altitude value of the topic
monitor_data
"T_monitor_data.tower_pos.x" #return the x parameter of the tower position
field in the topic monitor_data
```

If the data cannot be found the function returns “None”.

7.1.3.4.2 ROS Services

Services used by the tests are defined in the "services.py" file. These services can then be used in the fsm.py as an initial command or within function under transition or failure conditions.

Using the configuration file, a name to a specific service can be assigned. To define the service, the ROS name and the data-type it uses has to be provided.

This can be done in the following way:

```
"SRV_service_name": {"name": "name in ros", "type": service type}
```

These can be aggregated as:

```
service = {
           "SRV_fly_up": {"name":
           "/prodrone/flight_controller/altitude_change_direction/set_up",
           "type": Trigger},
           "SRV_fly_down": {"name":
           "/prodrone/flight_controller/altitude_change_direction/set_down",
           "type": Trigger},
           "SRV_lock_blade": {"name": "/prodrone/lock_blade",
           "type": Trigger},
           "SRV_unlock_tower": {"name": "/prodrone/unlock_tower",
           "type": Trigger}
          }
```

Service calls can either be used as initial commands in states or within functions. The following format is used:

```
["my service name", payload]
```

or in a specific case:

```
["SRV_fly_up", ""]
```

7.1.3.4.3 Global Variables

The global variable file is mainly used as an aggregate of system information. Here unfiltered topic data, service functions, the parameters of the current state, timing information and variables that the user can use can be found (Appendix VIII).

7.1.3.5 FSM Handler

The Handler gets all the important and necessary components for the test simulation, in one place. It checks if the execution can move to the next state, changes the state if the transition's conditions are fulfilled and updates the topic values. In addition, it generates the FSM dictionaries and starts nodes which are necessary for the execution. Finally, it checks if any of the following failures occurred:

- failures defined inside the \`fsm.py\` script.
- wrong variable name defined inside the \`fsm.py\` script, which does not match either the predefined topic, service, failure, state or transition name.

7.2 Log Analyzer

The Log Analyzer is used to obtain important information related to the flight during an inspection. It can be used, for example, for quick and initial debugging if a failure occurred during an inspection. It uses both the logs which contain all terminal output and the rosbag generated after the end of the survey. The output will either be SUCCESS, or FAIL following specific details about what and when happened. Twelve analyzers were developed:

1. all_sides_inspected: Check if all sides were inspected
2. gcs_connection: Check if the connection to GCS had high ping
3. gps_quality: Check if the quality of the GPS was bad
4. high_vibr_drone: Check if the UAV experienced high vibrations
5. high_vibr_gimbal: Check if the gimbal experienced high vibrations
6. hub_reached: Check if the hub was reached during the flight
7. manual_control: Check if the pilot took manual control
8. node_crash: Check if a node crashed during the operation
9. odroid_shutdown: Check if shutdown command was sent
10. photo_counter: Check if any photo failed to be taken
11. setpoint_follow: Check if the UAV stayed between the setpoints
12. system_load: Check if the CPU and RAM load were too high

The Analyzers use a variety of topics and messages to extract data. Those topics and messages are declared at the beginning of each analyzer. During the execution of Analyzers, checks are made for specific values. The used topics and messages as well as the specific values used to check if a condition is met are provided in Appendix IX.

7.3 CI/CD Pipeline Implementation

7.3.1 CI/CD Architecture

Simplified configuration of the CI/CD pipeline and its components is shown in Figure 30. The three basic entities in the system are the developer who pushes changes made in his local environment, the Gitlab server, where all the repositories are stored and the Runner node(s) which is a special host, capable of running CI jobs. GitLab CI/CD was chosen as a service for building, testing and deploying. GitLab CI/CD was chosen because:

- it can be integrated with Bitbucket (which is used as remote VC repository in Pro-Drone), by creating a CI/CD project and connecting the repository via URL;
- is well documented;
- is open-source;
- supports Docker;

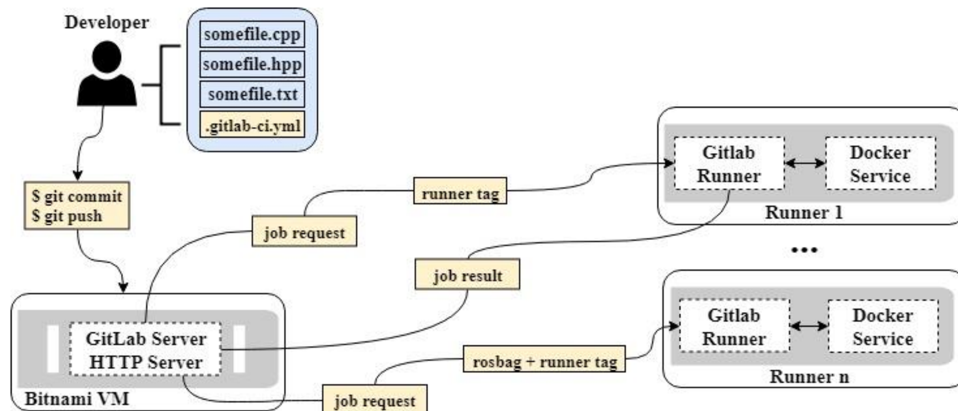


Figure 30. Simplified pipeline configuration

A decision was made to use Continuous delivery instead of Continuous deployment since manual hardware test are often required as a last testing step.

The pipeline consists of:

GitLab Server. Bitnami Virtual Machine [68], which contains a minimal Linux operating system with installed and configured GitLab CE is used as a GitLab Server. The virtual machine is set up using VirtualBox [69]. The virtual machine is also used as an HTTP server, to store and get rosbags, during the test runtime.

Gitlab-ci.yml file. GitLab CI/CD can be configured by using a .gitlab-ci.yml file, which needs to be placed in the root directory of the repository. It contains the stages of the pipeline, the tests that need to be executed, and variables such as the IP of the HTTP server and the Pro-Drone docker image. A pipeline is a group of jobs that are executed in stages. The jobs are executed in parallel and if all of the jobs end successfully the pipeline moves to the next stage. Otherwise, the pipeline fails. The jobs are executed using isolated machines called GitLab Runners.

GitLab Runner(s). The GitLab Runner is responsible for running the Continuous integration pipeline to the machine it has been registered. Multiple runners can be run on the same machine by registering a new one under a different name. This allows the user to run CI's from different branches in parallel, on the same machine. When the runner executes a rosbag job, it connects to the HTTP server to get the rosbag file declared in the .gitlab-ci.yml file.

Docker. Docker is the preferred way to set up and deploy the environment. Docker, when used with GitLab CI, runs each job in a separate and isolated container using the predefined image that is set up in .gitlab-ci.yml.

Repo Updater script. In case of frequent changes on multiple branches, the repo updater script can be used. It will search for branches which contain the word TEST and start CI pipeline automatically for each one of the branches, including master and development. The script checks every 15 seconds for changes, and this time can be increased or decreased.

The current pipeline strategy is Multiple Runners - One Machine (Appendix X). The code is compiled in the build directory and then is used as an artifact to be passed on the later jobs. Each job uses the dependencies tag to get the artifact-compiled code, which is placed in the same folder without any further action. That code is copied from the build directory and added to a new directory test_job/uav_control where the simulation and testing take place.

Figure 31 shows a more detailed depiction of the CI/CD pipeline.

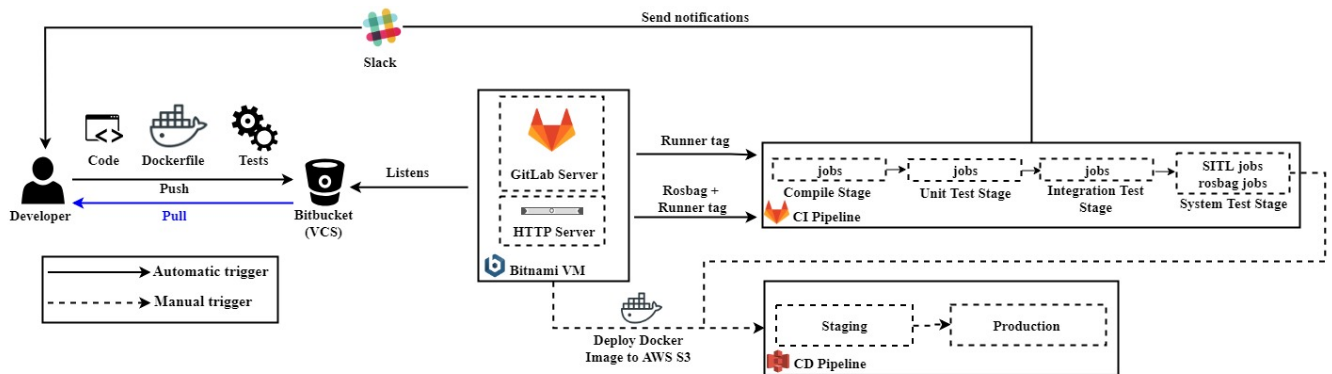


Figure 31. CI/CD configuration

The developer pulls the code from Bitbucket and creates a new branch. The branch should be named using the following format: BranchName_TEST. The developer implements a new feature, compiles the code locally and checks whether any compilation error occurred, then creates and runs the necessary unit and integration tests. If the tests are successful, the changes are pushed to the remote GitLab instance. The pipeline is automatically initiated. This starts a series of stages which trigger each other. In detail, these stages are:

1. Compile stage: Compiles the code. If the compilation is error-free, then artifacts are produced for later use. If an error occurs during the compilation, the pipeline finishes with a failure, and the error log is sent to the developers.
2. Unit Test Stage (Level 1 tests): Builds and executes the unit tests. It is triggered by the Compile stage, and it uses the artifacts produced by the Compile stage. During this stage, coverage data is collected. When a bug is found and fixed, a unit test is written to verify the fix. This is also part of regression testing which ensures that the defect will not be reintroduced in the future. If the stage finishes successfully, then the Integration test stage is triggered.
3. Integration Test Stage (Level 2 tests): Builds and executes integration test. It is triggered by Unit test stage, and it uses the artifacts produced by the Compile stage. For developing this tests, cross-team testing technique is used. This means that the integration test of module A is developed by a developer who has not worked on it. This creates a non-prejudiced approach which increases the possibility of detecting defects.
4. System Test Stage (Level 3 tests): Builds and executes system test. It is triggered by the Integration test stage, and it uses the artifacts produced by the Compile stage. These tests are changed rarely, and it is recommended to add new tests based on new inspection data.
5. Deploy Stage.

If during any of the stages a compilation error exists or if any test fails, the pipeline finishes with a failure and an error log (which can be a compilation error or test failure) is sent to the developers via Slack [70]. In this case, it is recommended that the developer should solve the introduced bug in less than an hour or revert the changes. The staged tests are executed automatically by CI Server every time a code is pushed to the remote repository. Although the last stage is more time consuming, at this moment it is still executed automatically every time a change is pushed. In the future, after additional tests are added, and if the duration becomes too long, it is recommended to execute them overnight. Static code analysis was also included in the pipeline by using GitLab's Code Quality.

If compile and test stages are completed successfully the Deploy stage starts. The Deploy stage contains two environments that deploy an image to a staging and production server. The staging and the production images are hosted on AWS S3 [71]. First, the code is deployed to the staging server for further testing. Performance requirements are checked, and Hardware in the loop (HITL) tests are run manually. If the software passes all these tests, a pull request from the feature test branch to the development branch is created. When the pull request is accepted and merged to the master branch a delivery release is triggered. The production script runs the build script, sets the Git tags, pushes them to Bitbucket, builds the production image from the Dockerfile, and pushes the image to a bucket hosted in AWS S3. Then the production image can be deployed at any time to the UAV.

Table 13 contains the average duration of each stage. Table 14 contains the number of tests at the beginning of this thesis and after. It also includes the type and the frequency of the running the tests. Some common issues encountered while using this pipeline and their solution are presented in Appendix XI.

Table 13. Average GitLab stage durations

	Compile stage	**Level 1 Tests**	**Level 2 Tests**	**Level 3 Tests**	**Total**
Time	01m 10s	03m 13s	8m 04s	30m 20s	42m 47s

Table 14. Testing types

	Type	**Frequency**	**Test Count (Before)**	**Test Count (After)**
Level 1 Tests	Automatic	Every Build	79	354
Level 2 Tests	Automatic	Every Build	0	86
Level 3 Tests	Automatic	Every Build	0	34

7.3.2 Multi-Stage Docker Build

One of the main challenges during building production Docker images is keeping the image size small. To achieve this, only artifacts that are needed during production need to be kept, and any unnecessary build and test artifacts need to be cleaned up. The initial approach was to use what is

referred to as the builder pattern and a development and production Dockerfiles were created and maintained.

In order to reduce the production image size and to maintain only one file, the multi-stage builds were used instead. Multi-stage build allows copying from intermediate stages without having those layers to show up in the final image. This means that with multi-stage builds it is possible to create single Dockerfile which chains all previously separated stages. Each stage in this Dockerfile can also be built individually using the --target flag while invoking docker build.

The created stages are shown in Figure 32. The *constant stage* contains the slowest changing dependencies, which need to be updated extremely rarely like the operating system and ROS. The next stage is the *base stage*. This stage contains the source code as well as all the dependencies. The building is handled during the *build stage*. The building is done in a separate stage since the compiling modifies a lot of files, and this creates a load which is not needed for example in the production image. After the building stage has passed the *development stage* starts. This stage inherits the dependencies from the base stage and copies the built binaries from the build stage. The *test stage* inherits from the development stage. Similarly, the *production stage* inherits the dependencies from the base stage and copies the binaries from the build stage. With this approach, the final size of the production image was reduced from 1.42 Gb to 319 Mb (if a change in the layers in the constant stage is not made).

Guidelines for creating new, or updating an existing image are provided in Appendix XII.

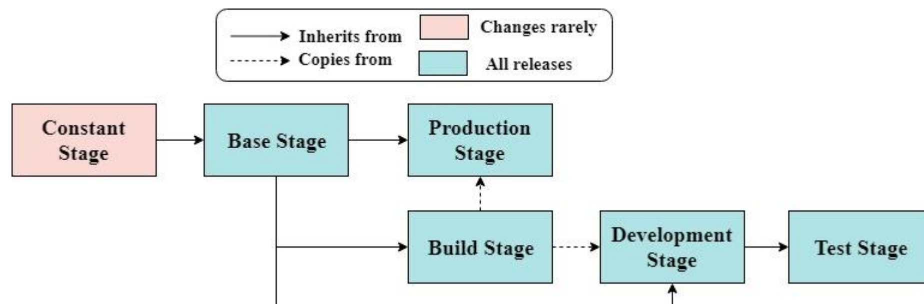


Figure 32. Docker Stages Structure

7.3.3 Security Consideration

The recommendations in Chapter 5.2.9 were closely followed during the pipeline implementation. Docker Secrets and GitLab Secret Variables were used to avoid hardcoding sensitive data in the Docker images. An additional threat was identified: untested, and thus possibly unstable image accidentally could be started in the production environment. This can be avoided by configuring the production environment to only execute images that use a specific versioning tag.

8 Discussion, Results, and Evaluations

In this chapter, the results of the field trials are presented. Later, a summary of the experienced benefits and challenges of CI/CD is presented. The results are based on the statistical data which can be accessed on Bitbucket and GitLab. In addition, to fully understand the influence CI/CD and Test automation had on the team and the project a survey was performed on several developers in Pro-Drone.

8.1 Field Testing

Several field trials on wind farms in Portugal and Brazil have already taken place where the production code has been successfully deployed on the UAV. The UAV was able to successfully finish the inspections.

During these trials, the sensory checks presented in Chapter 7.1.3.2 were also tested. The following tests were performed to check whether the correct state was shown in the GUI:

- Cover the lidar;
- Unplug the lidar;
- Turn on the UAV while the battery voltage is low;
- Shake the gimbal during calibration, so the calibration fails;
- After calibration, shake and move the gimbal;
- Activate the safety stop during flight and land the UAV manually;
- During the inspection, check whether the operator is notified when the tower and the blade are locked, relocked or lost;
- During the inspection, check whether the operator is notified when the blade tip is reached;
- During the inspection, check whether the operator is notified if the data is packaged and sent;
- After turning the UAV on, check whether the operator is notified when the autonomous mode is on;

8.2 Benefits of the CI/CD Pipeline and Automated Testing

To investigate the impact of implementing CI/CD pipeline on the team's productivity and software's quality the following metrics were used:

- number of bug reports (Figure 33);
- test coverage (Figure 34);
- number of merged pull requests (Figure 35);

- number of commits (Figure 36);
- number of production level defects (Figure 37);

The statistics data was obtained using Bitbucket (for data before implementing the CI/CD pipeline) and GitLab CI (for data after August 2017, when the company started using the pipeline). How the team's size varied during the last three years is shown in Table 15. With a strong CI/CD and Testing framework in place, the statistics and observations listed below prove that, although recently launched, the pipeline succeeded in achieving a robust, safer and continuously improving the software supply chain.

Table 15. Number of full-time developers in the team

	Team size
2016	4
2017	9
2018	8

Based on the GitLab and Bitbucket statistics data the following advantages were identified:

- *Code-base is more robust*. The success rate of the builds started with 67% and went up to 92%.
- *Increased number of tests and test coverage*. From Table 13 and Figure 34 it can be seen that there was a significant increase in the created tests and the test coverage.
- *Increased number of commits*. With an average total number of 37 commits per day in 2018, it's clear that developers are comfortable constantly refactoring the code base.
- *Increased number of pull requests*. From Figure 35 it can be seen that the number of pull requests increased after the implementation of the CI/CD pipeline.
- *Frequent releases*. There is no data about how frequently the releases were made before the implementation of the CI/CD pipeline. Now, minor releases are now done once a week and major releases once every other month.
- *Improved bug detection*. The peak in Figure 33 coincides with the moment when Pro-Drone started using the CI/CD pipeline and when an improved test coverage was created by implementing a significant number of tests.
- *Lower number of production level defects*. From Figure 37 it can be seen that the number of production level defects was lower in 2018.

Additional benefits are:

- The pipeline is transparent and easy to understand and modify.
- The entire production build pipeline is contained in just one Dockerfile, which replaced several scripts.
- Many cases and margins can now be validated during the tests before even involving real hardware or deployment under real conditions occurs. All components can be tested thoroughly using simulated data or rosbag data.

- Developers can now have different versions with mutually exclusive dependencies running on the same UAV. Several developers can test their code when only one instance of hardware is available.
- Setting up a complete local development or a production environment now takes ~2 hours. Previously, performing a clean installation and manual configuration could take up to several days. The previous setup time included manual configuration and dependencies installation, as well as manual building and compiling.

An additional benefit was identified from representing the system using FSM. By building the representation of the system using FSM, the developers were able to understand all parts of the system behavior effectively and in great detail. Additionally, they were able to understand the inspection process even if they have never attended an inspection in person.

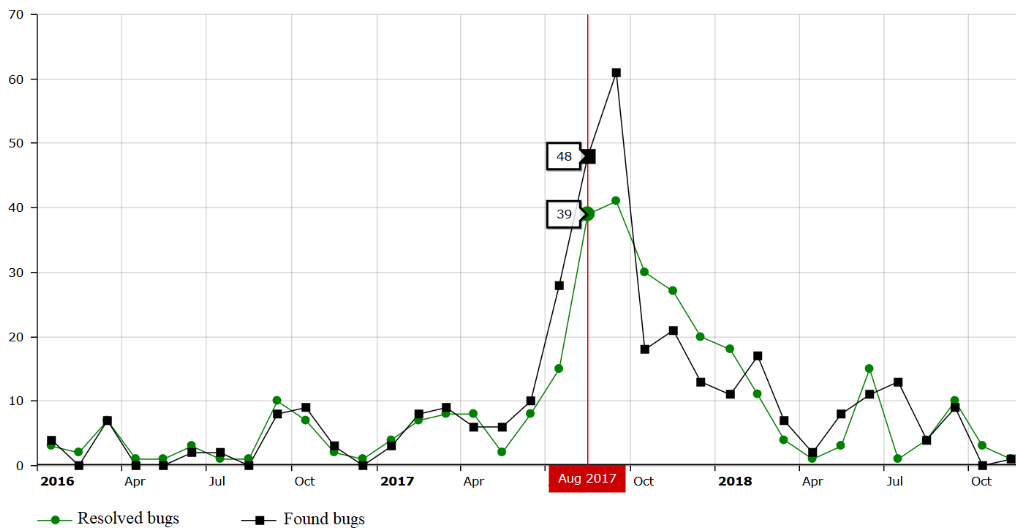


Figure 33. Number of founds and resolved bugs

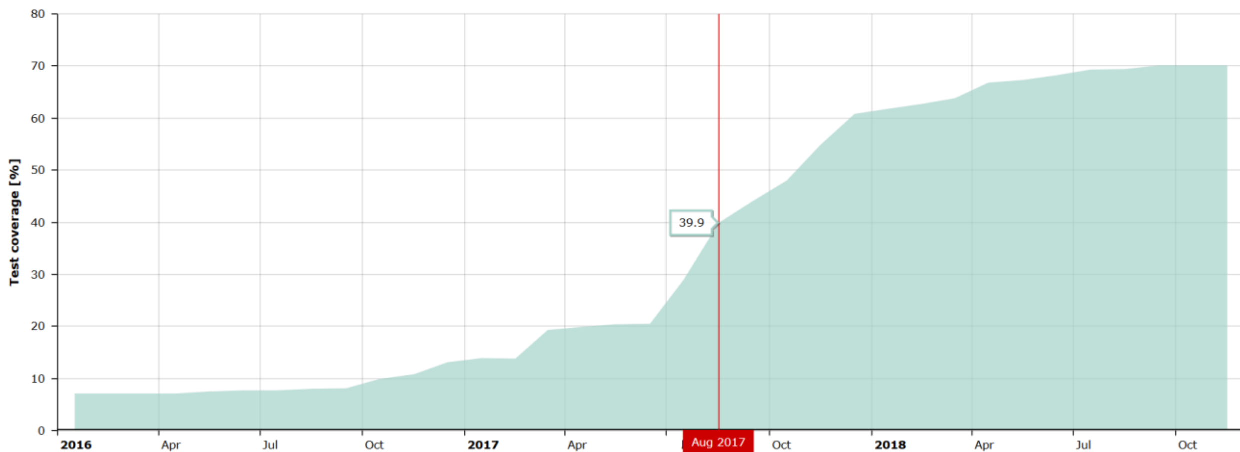


Figure 34. Test coverage

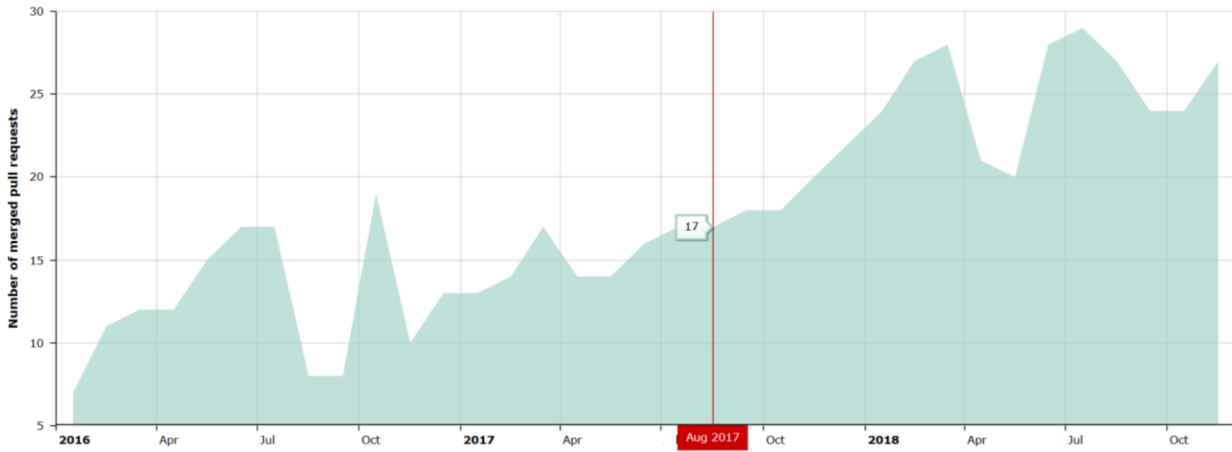


Figure 35. Number of merged pull requests

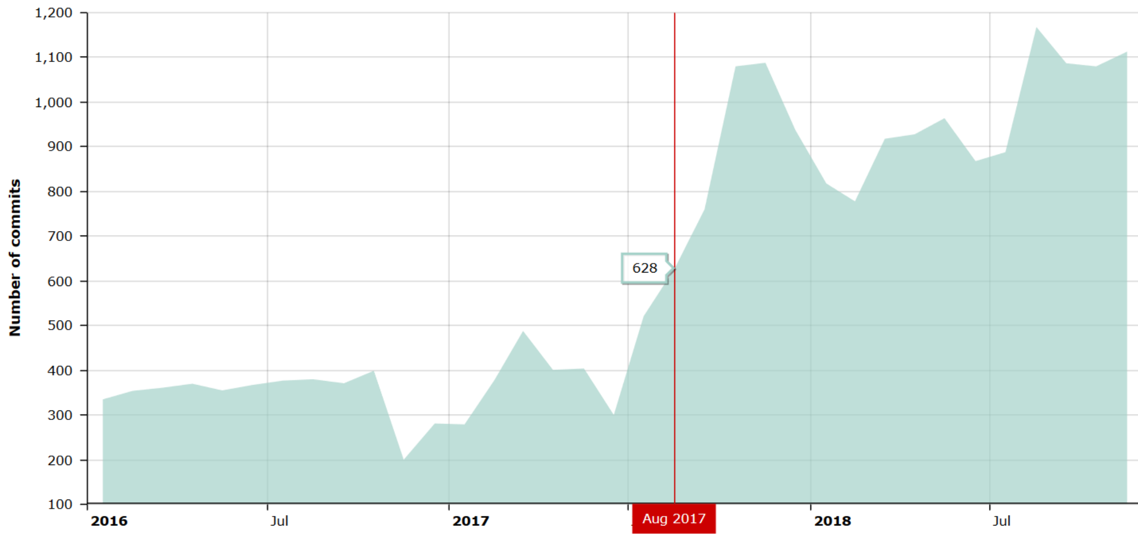


Figure 36. Number of commits

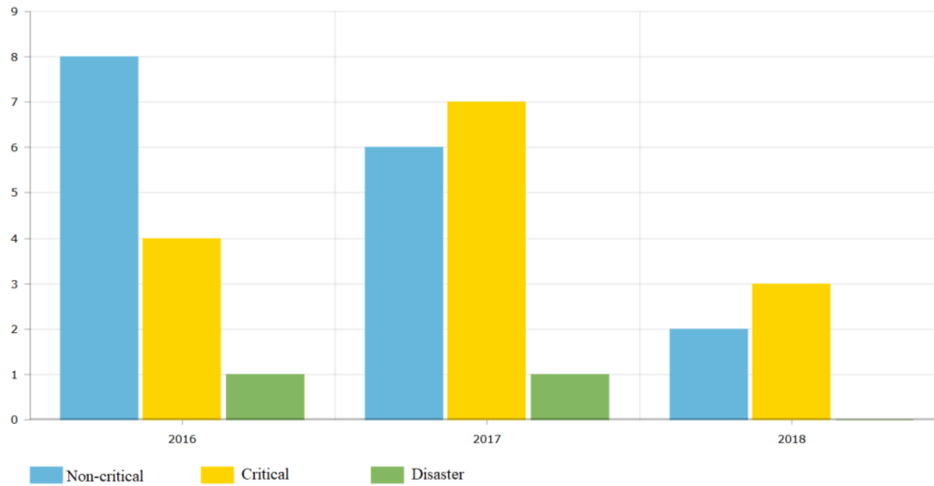


Figure 37. Production level defects

8.3 Considerations

The biggest concern is the production level image size. Although the image size was significantly reduced by using multi-stage Docker build, a change in the layer in the constant stage will cause large updates, sometimes up to 2.5 GB. In this case, pulling the image to the UAV can take a while especially in a wind farm where the Wi-Fi connection might be poor. However, this approach was still faster than pulling from source and recompiling the code.

8.4 Developer Survey

A survey on seven developers was done, focusing on the experienced challenges and benefits. The respondents were asked whether they experienced the challenges and the benefits listed in Table 16 and Table 17. The results of the survey supported the results presented in the literature. All of the respondents agreed that CI/CD increased the confidence in the code base and the deployment process.

According to most of the interviewed developers, the Test automation resulted in better use of the resource in the long run since most manual tasks were automatized. In addition, they felt that CI/CD resulted in better development practices and increased the transparency during the development process. The results show that the biggest challenge for the developers was the higher initial cost, especially during the test development. In addition, some developers felt that the test coverage should be higher in order to be sure that the release will not be error-prone.

Table 16. Experienced benefits after adopting the pipeline

Benefit	**Dev1**	**Dev2**	**Dev3**	**Dev4**	**Dev5**	**Dev6**	**Dev7**
Easier deployments	x	x	x	x		x	x
Better use of resources	x	x	x			x	x
Increased transparency	x	x	x	x	x	x	x
Increased confidence	x	x	x	x	x	x	x
Creates good software development practice	x	x	x	x	x	x	x
Easier development	x	x				x	x

Table 17. Experienced challenges during the implementation and after adopting the pipeline

Challenges	**Dev1**	**Dev2**	**Dev3**	**Dev4**	**Dev5**	**Dev6**	**Dev7**
Error-prone release				x	x		
High short-term cost	x	x	x	x	x	x	x
Technically challenging		x	x	x	x		
Organizational challenges	x		x				

9 Conclusion and Future Work

The UAV is a complex and a safety-critical system. During its operation the safety of the operator and the reliability and longevity of the device must be ensured. As a result of this thesis, a secure software supply pipeline used for testing and deployment was developed from scratch. The pipeline was demonstrated to be operational by testing it on Pro-Drone's product and it is actively used by Pro-Drone's team.

Several improvements towards the area of robotics software testing and deployment were made in this thesis. First, additional safety check algorithms were implemented and used to notify the operator about the current state of the UAV via a modified version of the GCS GUI. Several rosbag Analyzers were implemented in order to create a quick debugging tool. A research towards the best testing practices was performed and the control software was tested on all three levels. For the most complex Level 3 tests a custom Python package was implemented. Level 3 tests were performed using a black-box testing method known as Finite State Machines. Level 1 tests were modified and new tests were added, and Level 2 and Level 3 test were implemented from scratch.

Research towards modern CI/CD practices in order to effectively automate the compilation, building, and testing of software was performed. The pipeline leverages the Docker containerization environment and its multi-stage builds to build and deliver images in a simple, concise, and easily maintainable way.

The pipeline can be used for testing single components as well as the system as a whole in an efficient way, effectively reducing the time and effort required to make a new release. Additional benefits arising from this pipeline are rapid development of components and testing under controlled conditions and regular testing and deployments that require little to no user input. Finally, development, testing, and production environments are consistent and developers can work on shared assets.

There are several improvements that could be introduced to the pipeline. First of all, the coverage rate to needs to be increased by adding new unit tests. The next goal is reaching coverage of 75%. The number of integration tests should also be increased. Different simulated sensors can be shipped to different containers which would allow testing on a UAV which does not contain the complete hardware setup.

References

1. Ioan A. Sucan, Sachin Chitta, “MoveIt!”, [Online]. Available: http://moveit.ros.org. [Accessed: 30-October-2018].
2. Ruffin White, Henrik Christensen, "ROS and Docker", Robot Operating System (ROS): The Complete Reference, Springer, vol. 2. (2017)
3. R. Mabry, J. Ardonne, J. N. Weaver, D. Lucas, and M. J. Bays, “Maritime autonomy in a box: Building a quickly-deployable autonomy solution using the Docker container environment.”, Proceedings of OCEANS 2016 MTS/IEEE Monterey. Monterey, CA: IEEE (2016)
4. T. Fromm, C. A. Mueller, M. Pfingsthorn, A. Birk, and P. di Lillo, “Efficient Continuous System Integration and Validation for Deep-Sea Robotics Applications.”, OCEANS 2017 - Aberdeen (2017)
5. Aaron Grattafiori, “Understanding and hardening Linux containers.”, Whitepaper, NCC Group (2016)
6. Center for Internet Security. “Cis docker 1.13.0 benchmark. Technical report.”, (2017)
7. Andy Clemenko, “Docker Reference Architecture: Building a Docker Secure Supply Chain”, [Online]. Available: https://success.docker.com/article/secure-supply-chain., [Accessed: 30-October-2018].
8. “Global Wind Energy Outlook 2016.” [Online]. Available: http://gwec.net/publications/global-wind-energy-outlook/global-wind-energy-outlook-2016/#. [Accessed: 24-October-2018].
9. “Advantages and Challenges of Wind Energy.” [Online]. Available: https://www.energy.gov/eere/wind/advantages-and-challenges-wind-energy. [Accessed: 24-October-2018].
10. “Operation and Maintenance Costs of Wind Generated Power.” [Online]. Available: https://www.wind-energy-the-facts.org/operation-and-maintenance-costs-of-wind-generated-power.html. [Accessed: 24-October-2018].
11. Li Dongsheng, M Ho Siu-Chun, Song Gangbing, Ren Liang, Li Hongnan, “A review of damage detection methods for wind turbine blades.”, Smart Materials and Structures. 24. 10.1088/0964-1726/24/3/033001 (2015)
12. M.J. Schulz, M.J. Sundaresan, “Smart Sensor System for Structural Condition Monitoring of Wind Turbines.”, Subcontract Report NREL/SR-500-40089 (2006)
13. Chia Chen Ciang, Jung-Ryul Lee, Hyung-Joon Bang, “Structural health monitoring for a wind turbine system: a review of damage detection methods.”, Measurement Science and Technology, Volume 19, Number 12 (2008)
14. L. W. M. M. Radmakers, H. Braam, M. B. Zaaijer, and G. J. W. van Bussel, “Assessment and optimization of operation and maintenance of offshore wind turbines.” (2003)
15. Jon Salmon, Matt Sigala, Kelly Miller, “Blade Maintenance: Observations from the field and practical solutions.” [Online]. Available: https://www.windpowerengineering.com/business-news-projects/webinars/blade-maintenance-observations-from-the-field/. [Accessed: 24-October-2018].

16. Jason Deign, “Fully automated drones could double wind turbine inspection rates.”, 2016, [Online]. Available: http://newenergyupdate.com/wind-energy-update/fully-automated-drones-could-double-wind-turbine-inspection-rates. [Accessed: 24-October-2018].
17. Donna Dawson, “Service & repair: Optimizing wind power’s grid impact.”, 2018, [Online]. Available: https://www.compositesworld.com/articles/service-repair-optimizing-wind-powers-grid-impact-. [Accessed: 26-October-2018].
18. Bladena, Vattenfall, EON, Statkraft and KIRT x THOMSEN, “Instructions for blade inspections. EUDP Project LEX (2014-16) and EUDP Project RATZ (2016-18)”, 2018, [Online]. Available: https://www.bladena.com/uploads/8/1/6/3/81635550/instruction_-_blade_inspections.pdf. [Accessed: 26-October-2018].
19. Jon Salom, “Choosing the right blade maintenance regime.”, 2015, [Online]. Available: https://www.windpowermonthly.com/article/1369612/choosing-right-blade-maintenance-regime. [Accessed: 26-October-2018].
20. Ulrike Esther Franke, “Civilian Drones: Fixing an Image Problem?”, 2015, [Online]. Available: https://isnblog.ethz.ch/security/civilian-drones-fixing-an-image-problem. [Accessed: 26-October-2018].
21. Michelle Froese, “Navigant Research Report”, 2016, [Online]. Available: https://www.windpowerengineering.com/mechanical/blades/navigant-research-report-drones-for-wind-turbine-inspections-expected-to-hit-6-billon-by-2024/. [Accessed: 26-October-2018].
22. “Matrice 60 Pro User Manual V1.0”, 2018, [Online]. Available: https://dl.djicdn.com/downloads/m600%20pro/20180417/Matrice_600_Pro_User_Manual_v1.0_EN.pdf. [Accessed: 27-October-2018].
23. “PX4 Documentation”, 2018, [Online]. Available: https://px4.io/documentation/. [Accessed: 24-Devember-2018].
24. “ODROID-XU4”, 2018, Available: https://wiki.odroid.com/odroid-xu4/odroid-xu4. [Accessed: 27-October-2018].
25. “MAVLink Developer Guide”, 2018, Available: https://mavlink.io/en/. [Accessed: 24-Devember-2018].
26. “Logitech C920 HD PRO WEBCAM”, 2018, Available: https://www.logitech.com/en-us/product/hd-pro-webcam-c920. [Accessed: 24-Devember-2018].
27. “Sony Alpha 7R”, 2018, Available: https://www.dpreview.com/reviews/sony-alpha-a7r. [Accessed: 24-Devember-2018].
28. “DJI Ronin”, 2018, Available: https://www.dji.com/ronin-mx. [Accessed: 24-Devember-2018].
29. Dr. Rüdiger Paschotta, “Group Delay.”, [Online]. Available: https://www.rp-photonics.com/group_delay.html. [Accessed: 28-October-2018].
30. Metrology Resource Co., “Metrology Sensor Products - MRL3.”, [Online]. Available: http://www. metrologyresource.com/laser-sensor-MRL3.php. [Accessed: 28-October-2018].
31. “Hokuyo UTM-30LX Specifications Catalog.”, [Online]. Available: https://www.hokuyo-usa.com/products/scanning-laser-rangefinders/utm-30lx. [Accessed: 27-October-2018].

32. “ROS Jade.” [Online]. Available: http://wiki.ros.org/jade. [Accessed: 25-November-2018].
33. “Gazebo.”, [Online]. Available: http://gazebosim.org/. [Accessed: 25-November-2018].
34. “Ubuntu 14.04.”, [Online]. Available: http://releases.ubuntu.com/14.04/. [Accessed: 25-November-2018].
35. “Git.”, [Online]. Available: https://git-scm.com/. [Accessed: 25-November-2018].
36. “Bitbucket.”, [Online]. Available: https://bitbucket.org. [Accessed: 25-November-2018].
37. “Clion.”, [Online]. Available: https://www.jetbrains.com/clion/. [Accessed: 25-November-2018].
38. Adrian Mouat, “Using Docker.”, O'Reilly Media, Inc. (2015)
39. Docker Inc., “Docker overview.”, 2018, [Online]. Available: https://docs.docker.com/engine/dockeroverview/., [Accessed: 01-November-2018].
40. Docker Inc., “Dockerfile reference.”, 2018, [Online]. Available: https://docs.docker.com/engine/reference/builder/#usage., [Accessed: 01-November-2018].
41. Docker Inc., “Overview of Docker Compose.”, 2018, [Online]. Available: https://docs.docker.com/compose/overview/., [Accessed: 01-November-2018].
42. Docker Inc., “About registry.”, 2018, [Online]. Available: https://docs.docker.com/registry/introduction/., [Accessed: 01-November-2018].
43. Docker Inc., “How nodes work.”, 2018, [Online]. Available: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/., [Accessed: 01-November-2018].
44. Docker Inc., “Docker content trust.”, 2018, Available: https://docs.docker.com/engine/security/trust/content_trust/., [Accessed: 11-November-2018].
45. Justin Ellingwood, “An Introduction to Continuous Integration, Delivery, and Deployment.”, Digital Ocean, 2017, [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-continuous-integration-delivery-and-deployment., [Accessed: 09-November-2018].
46. “GitLab CI/CD”, 2018, [Online]. Available: https://about.gitlab.com/product/continuous-integration/., [Accessed: 09-November-2018].
47. J. Humble and D. Farley, “Continuous delivery: reliable software releases through build, test, and deployment automation.” Pearson Education, (2010)
48. Martin Fowler, “Practices of Continuous Integration.”, 2006, [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration., [Accessed: 09-November-2018].
49. Mike Wacker, “Just Say No to More End-to-End Tests.”, 2015, [Online]. Available: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html., [Accessed: 09-November-2018].
50. Glenford J. Myers, Corey Sandler, Tom Badgett, “The Art of Software Testing.”, John Wiley & Sons, Inc., (2012)
51. Michael E. Caspersen, Ole Lehrmann Madsen, “Testing Object-Oriented Software – COT/2-43-V1.0.”, 2001, [Online]. Available:

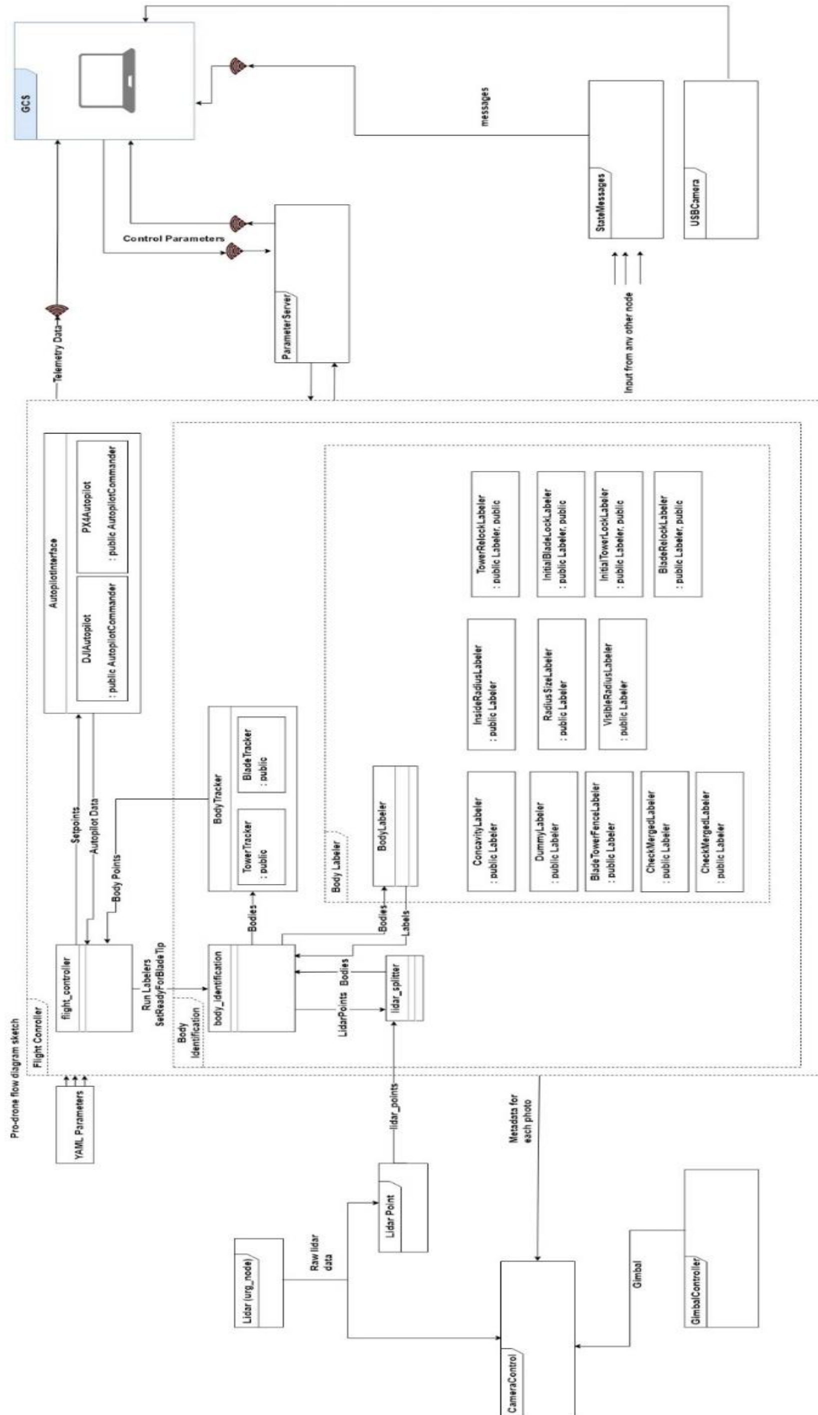
http://www.cit.dk/COT/reports/reports/Case2/43/cot-2-43.doc., [Accessed: 09-November-2018].

52. E. Dustin, J. Rashka, and J. Paul, "Automated software testing: introduction, management, and performance.", Addison-Wesley Professional, (1999)
53. D. M. Rafi, K. R. K. Moses, K. Petersen, M. V. Mantyl, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey." Proceedings of the 7th International Workshop on Automation of Software Test, ser. AST '12. Piscataway, NJ, USA: IEEE Press, pp. 36–42, (2012)
54. Dorothy Graham and Mark Fewster, "Software Test Automation: Effective Use of Test Execution Tools.", ACM Press/Addison-Wesley, (1999)
55. Manfred Rätzmann, Clinton De Young, "Software Testing and Internationalization.", Galileo Press GmbH, (2002)
56. Katja Karhu, Tiina Repo, Ossi Taipale, Kari Smolander, "Empirical observations on software testing automation. In 2009 International Conference on Software Testing Verification and Validation.", pages 201–209. IEEE, (2009)
57. Open Source Robotics Foundation, 2018, "Automatic Testing with ROS.", Available: http://wiki.ros.org/action/show/Quality/Tutorials/UnitTesting?action=show&redirect=UnitTesting. [Accessed: 11-November-2018].
58. Mohd. Ehmer Khan, Farmeena Khan, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques." (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 3, No.6, (2012)
59. Srinivas Nidhra, Jagruthi Dondeti, "Black box and white box techniques – a literature review.", International Journal of Embedded Systems and Applications (IJESA) Vol.2, No.2, (2012)
60. Víctor González-Pacheco, 2018, "Test driven development in ROS.", [Online]. Available: https://github.com/VGonPa/ros-testing-tutorial. [Accessed: 12-November-2018].
61. "unittest.", [Online]. Available: http://wiki.ros.org/unittest#ROS-Node-Level_Integration_Tests. [Accessed: 21-November-2018].
62. "GoogleTest.", 2018, [Online]. Available: https://github.com/abseil/googletest. [Accessed: 21-November-2018].
63. "Rostest", 2018, [Online]. Available: http://wiki.ros.org/rostest. [Accessed: 21-November-2018].
64. Arpan Sen, 2010, "A quick introduction to the Google C++ Testing Framework.", [Online]. Available: https://www.ibm.com/developerworks/aix/library/au-googletestingframework.html. [Accessed: 21-November-2018].
65. Open Source Robotics Foundation, 2018, "Writing a Simple Service and Client (C++).", Available: http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28c%2B%2B%29. [Accessed: 21-November-2018].
66. Institut Maupertuis, 2016, "Simple rostest.", [Online]. Available: https://gitlab.com/InstitutMaupertuis/simple_rostest. [Accessed: 21-November-2018].
67. T.S. Chow, "Testing Software Design Modeled by Finite-State Machines.", IEEE Trans. Software Eng., vol. SE-4, no. 3, pp. 178-187 (1978)

68. “GitLab CE.”, [Online]. Available: https://bitnami.com/stack/gitlab/virtual-machine. [Accessed: 25-November-2018].
69. “VirtualBox.”, [Online]. Available: https://www.virtualbox.org/. [Accessed: 25-November-2018].
70. “Slack”., [Online]. Available: https://slack.com/. [Accessed: 25-November-2018].
71. “Amazon S3.”, [Online]. Available: https://aws.amazon.com/s3/. [Accessed: 25-November-2018].

Appendices

I. System architecture of the UAV control package



II. Guidelines for testing loops

The following guidelines are proposed for testing different loops (Figure 38) with size n [59]:

Simple loops

1. Skip the entire loop.
2. Make one pass through the loop.
3. Make two passes through the loop.
4. Make m passes through loop where m<n
5. Make (n-1), n and (n+1) passes through the loop

Nested loops

1. Start at the innermost loop. Set all the other loops to a minimum value.
2. For the selected loop, perform a simple loop test.
3. Perform test for the next loop and work outward.
4. Continue testing until the outermost loop has been tested.

Concatenated Loops

If two loops are independent of each other then they are tested using simple loops. Otherwise, they are tested as nested loops.

Unstructured loops

Unstructured loops should be redesigned.

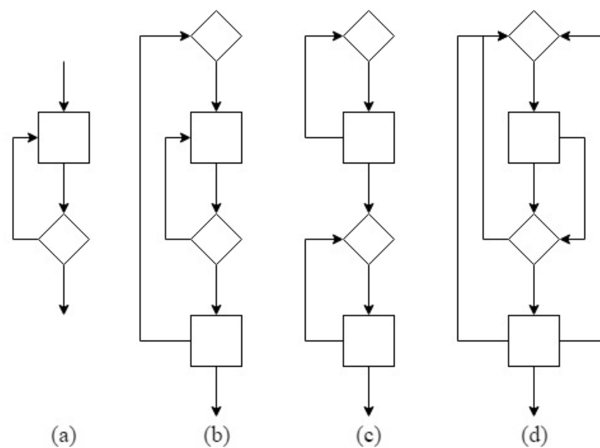


Figure 38. (a) Simple, (b) nested, (c) concatenated and (d) unstructured loop

III. Reporting test results and documenting test cases

Example for a documented test case is provided in Table 18. *Documenting test cases*

Table 18. Documenting test cases

Test case	PutNegativeData
Goal of the test	To check that the Gateway client does not respond and handles correctly the case of negative status code response.
Prerequirements	The Connection timeout in the conf file should be set to 30 seconds.
Test level goals reached	Yes
Action	**Expected result**
1. Prepare data for delivering. 2. Activate putData process. 3. Respond to putDataRequest with Status code -1. 4. Repeat steps 1 – 3, but use codes a. -2: System down b. -3: Contract suspended c. -6: Channel closed d. -7: Channel not found e. -8: Invalid file size	• Gateway terminates without sending the report.
1. Prepare documents for delivering. 2. Activate putData process. 3. Respond to putDataRequest with Status code -4 (Authentication error). 4. Repeat steps 1-3, but use code -5 (Unknown hardware hash).	• Gateway terminates without sending the report. • The data stays in the folder.

Any detected bug should be reported to TeamWork. The fault report must contain the severity of the detected bug:

1. Critical (S1) - The bug affects critical functionality of the software system. Example: complete failure of a feature which may endanger the operator and damage the equipment. Without fixing the bug, the system is unusable.
2. Major (S2) - The bug affects major functionality of the software system. Example: camera fails to take pictures which causes the inspection to fail. The system can be used but without some of its important functionalities.
3. Minor (S3) - The bug affects minor functionality of the software system. Example: bug in area which is used rarely, UI bug.
4. Trivial (S4) - The bug does not affect the functionality of the software system in any way. Example: spelling/grammar errors, unused variables, not following naming conventions, poorly structured code.

IV. State Watcher – Lidar Integration Test

FakeLidarPublisher.cpp

```
#include <ros/ros.h>
#include <gtest/gtest.h>
#include <sensor_msgs/LaserScan.h>
#include <prodrone/sensors/Lidar.hpp>
#include <prodrone/autopilot/AutopilotInterface.hpp>
#include <prodrone/autopilot/AutopilotFactory.hpp>
#include <prodrone/state_watcher/StateVariableInterface.hpp>
#include <prodrone/StateGCSArray.h>
#include <prodrone/State_gcs.h>
#include <prodrone/sensors/LidarPoint.hpp>

namespace prodrone
{
class LidarNotConnectedTest
{
public:
    ros::NodeHandle node1;
    ros::Subscriber subscriber1_;
    ros::Publisher publisher1_;
    void NotConnectedPublish();
    void Callback(const sensor_msgs::LaserScan::ConstPtr& scan);
    bool message_received_ = false;
    uint16_t states = 0;

    LidarNotConnectedTest()
    {
        this->subscriber1_ = node1.subscribe("/scan", 1, &LidarNotConnectedTest::Callback, this);
        this->publisher1_ = node1.advertise<sensor_msgs::LaserScan>("/scan", 1);
    }

};

void LidarNotConnectedTest::Callback(const sensor_msgs::LaserScan::ConstPtr& scan)
{
    this->message_received_ = true;
    for (int i = 0; i < scan->ranges.size(); ++i)
    {
        const float &range = scan->ranges[i];
    }
}

void LidarNotConnectedTest::NotConnectedPublish()
{
    unsigned int num_readings = 1080;
    float laser_frequency = 40;
    float ranges[num_readings];
    float intensities[num_readings];
    int count = 0;

 while (count < 10)
    {
        // generate fake data for laser scan
        for (unsigned int i = 0; i < num_readings; i++)
        {
            ranges[i] = 0.01;
            intensities[i] = 0.001;
        }

        sensor_msgs::LaserScan scan;
        scan.header.stamp.sec = ros::Time::now().toSec();
        scan.header.stamp.nsec = ros::Time::now().toNSec();
        scan.header.frame_id = "px4";
        scan.angle_min = -1.57;
        scan.angle_max = 1.57;
        scan.angle_increment = 3.14/num_readings;
        scan.time_increment = (1 / laser_frequency) / (num_readings);
```

```
        scan.range_min = 0.0;
        scan.range_max = 60.0;
        scan.ranges.resize(num_readings);
        scan.intensities.resize(num_readings);

        for (unsigned int i = 0;  i < num_readings; i++)
        {
            scan.ranges[i] = ranges[i];
            scan.intensities[i] = intensities[i];
        }

        publisher1_.publish(scan);
        count++;
        ros::Duration(0.11).sleep(); // State watcher update rate is 0.1 HZ
    }
}
} // namespace prodrone
```

LidarStatesTest.cpp

```
#include <ros/ros.h>
#include <gtest/gtest.h>
#include "FakeLidarPublisher.hpp"
#include <prodrone/state_watcher/States.hpp>
#include <prodrone/sensors/Lidar.hpp>

namespace prodrone
{
class LidarNoDataStatePublishTest : public ::testing::Test
{
public:
    ros::NodeHandle node_;

    AutopilotStatus autopilot;
    Lidar& lidar;

    bool message_received_states = false;
    bool has_no_data_lidar_state = false;
    bool has_lidar_blocked_state = false;

    void stateCallback(const StateGCSArray::ConstPtr& msg);
    ros::Subscriber state_subscriber;

    uint16_t states = 0;

    LidarNoDataStatePublishTest() : autopilot(Autopilot::factory(node_)), lidar(Lidar::getInstance
                                                                                (node_, autopilot))
    {
        this->state_subscriber = node_.subscribe("/prodrone/gcs/states", 50,
&LidarNoDataStatePublishTest::stateCallback, this);
    }

    FRIEND_TEST(LidarNoDataStatePublishTest, lidar_not_connected_test);
    FRIEND_TEST(LidarNoDataStatePublishTest, lidar_blocked_test);
};

void LidarNoDataStatePublishTest::stateCallback(const StateGCSArray::ConstPtr& msg)
{
    this->message_received_states = true;

    for (int i=0; i < msg->states.size(); i++)
    {
        const State_gcs &data = msg->states[i];
        states = data.id;
        ROS_INFO("States: %d", states);
    }

    for (auto &i: msg->states)
    {
        if(i.id == static_cast<uint8_t>(states::StateID::LidarBlocked))
        {
            ROS_INFO("State Lidar Blocked Received");
```

```cpp
            this->has_lidar_blocked_state = true;
        }

        if(i.id == static_cast<uint8_t>(states::StateID::NoDataReceivedFromLidar))
        {
            ROS_INFO("State NoDataReceivedFromLidar Received");
            this->has_no_data_lidar_state = true;
        }
    }
}

TEST_F(LidarNoDataStatePublishTest, lidar_not_connected_test)
{
     /// Wait to get notification that Lidar message is not received    ros::Duration(6).sleep();
    ros::spinOnce();

    ASSERT_TRUE(this->message_received_states);
    EXPECT_TRUE(this->has_no_data_lidar_state);
}

TEST_F(LidarNoDataStatePublishTest, lidar_blocked_test)
{
    LidarNotConnectedTest lidarNotConnectedTest;
    lidarNotConnectedTest.NotConnectedPublish();

    ros::Duration(1.2).sleep();
    ros::spinOnce();

    ASSERT_TRUE(this->message_received_states);
    ASSERT_TRUE(lidarNotConnectedTest.message_received_);
    EXPECT_TRUE(this->has_lidar_blocked_state);
}
} // namespace prodrone
```

unit_test_with_launch.cpp

```cpp
#include "LidarStatesTest.cpp"

int main(int argc, char **argv)
{
    ROS_INFO("MAIN");
    ros::init(argc, argv, "unit_test_sensors");
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

unit_test_sensors.launch

```xml
<launch>
    <rosparam command="load" file="$(find prodrone)/cfg/init_parameters.yaml" />
    <node pkg="prodrone" name="state_watcher"  type="state_watcher" output="screen"/>
    <node pkg="prodrone" name="fake_workorder_sender"  type="fake_workorder_sender" />

    <param name="/prodrone/survey_type" type="string" value="ROSBAG"/>
    <node pkg="prodrone" name="flight_controller"  type="flight_controller" output="screen"/>

    <test test-name="unit_test_sensors" pkg="prodrone" type="unit_test_sensors" />
</launch>
```

Modify CMakeLists.txt

```cmake
## test_sensors
add_rostest_gtest(unit_test_sensors
        test/launch/unit_test_sensors.launch
        test/sensors/unit_test_with_launch.cpp
)
target_link_libraries(unit_test_sensors
        prodrone_sensors_lib
        ${catkin_LIBRARIES}
        ${GTEST_LIBRARIES}
        prodrone_autopilot_lib
)
add_dependencies(unit_test_sensors
        prodrone_generate_messages_cpp
        ${catkin_EXPORTED_TARGETS}
)
```

V. Sample FSM test file

```
from check_fail_state import fail_if_state_active
from topic_data_extractor import *
global state
state = {}

states =        {       #state                  #send command on load

                        'start':                [['start']],
                        'LaunchDrone':          None,
                        'HoldToTip0':           [["SRV_fly_hold", ""]],
                        'RSScan':               [["SRV_fly_up", ""]],
                        'Level_with_hub':       [["SRV_fly_down",""]],
                        'HoldToHub0':           [["SRV_fly_hold", ""]],
                        'SwitchToLE':           [["SRV_flight_phase", "2"]],
                        'LEScan':               [["SRV_fly_down",""]],
                        'SwitchToLS':           [["SRV_flight_phase", "4"]],
                        'LSScan':               [["SRV_fly_up", ""]],
                        'HoldToHub1':           [["SRV_fly_hold", ""]],
                        'Stop':                 None
                }

global state_link
               # Start   # Target condition # Condition to change state
state_link = [
               ['start', 'LaunchDrone', None],
               ['LaunchDrone', 'HoldToTip0', ["T_monitor_data.flight_phase", "=", 1]],
               ['HoldToTip0', 'RSScan', [["STATE_RUNTIME", 1], "&", ["SAVE_VAR", {"tip":
"T_monitor_data.altitude"}]]],
               ['RSScan', 'Level_with_hub', ["T_monitor_data.blade_locked", "=", False]],
               ['Level_with_hub', 'HoldToHub0', ["T_monitor_data.blade_locked", "=", True]],
               ['HoldToHub0', 'SwitchToLE', [["STATE_RUNTIME", 1], "&", ["SAVE_VAR", {"hub":
"T_monitor_data.altitude"}]]],
               ['SwitchToLE', 'LEScan', ["T_monitor_data.flight_phase", "=", 3]],
               ['LEScan', 'SwitchToLS', ["T_monitor_data.altitude", "<", ["LOAD_VAR", {"tip": None}]]
],
               ['SwitchToLS', 'LSScan', ["T_monitor_data.flight_phase", "=", 6]],
               ['LSScan', 'HoldToHub1', ["T_monitor_data.altitude", ">", ["LOAD_VAR",
{"hub": None}]]],
               ['HoldToHub1', 'Stop', None]
             ]

# fail if it doesn't re-lock
global state_fail
state_fail = {   #state    #conditions
                 'RSScan': ["T_monitor_data.flight_phase", "=", 1]
             }

# fail if a gcs state is active
global check_state_fail
check_state_fail = {  #key          #gcs state conditions
                          'CheckState': [lambda: fail_if_state_active('SafetyStop',
                          get_field("T_states_data.states"))]
                   }
global node_timeout
node_timeout = 2000
```

VI. Publish state variable example

```
#include <prodrone/state_watcher/StateVariableInterface.hpp>

namespace prodrone
{
class BatteryVoltageReader
{

public:

BatteryVoltageReader(ros::NodeHandle& node_);
        void publish(); ///< Publish the voltage
        float getVoltage();

private:

void checkBatteryVoltageLow(float battery_voltage); ///< Check if battery's voltage level is < voltage_min
void publishBatteryVoltageLow(bool is_voltage_low); ///< Publish state variable if the voltage is < voltage_min
    prodrone::StateVariablePublisher pub_var; ///< Handle
};
}

///< Constructor
prodrone::BatteryVoltageReader::BatteryVoltageReader(ros::NodeHandle& node_) : node(node_), pub_var(node_)
{
   ...
}

float getVolatage()
{
        ...
        return voltage_value;
}

......

void prodrone::BatteryVoltageReader::publish()
{
    std_msgs::Float32 msg;
    msg.data = this->voltage_value;
    checkBatteryVoltageLow(msg.data);
    this->publisher.publish(msg);
}

void prodrone::BatteryVoltageReader::checkBatteryVoltageLow(float battery_voltage)
{
    if(battery_voltage < battery_voltage_min) {is_voltage_low = true;}
    else {is_voltage_low = false;}

    publishBatteryVoltageLow(is_voltage_low);
}

void prodrone::BatteryVoltageReader::publishBatteryVoltageLow(bool is_voltage_low)
{
       pub_var.publishVariable(prodrone::StateVariableID::BatteryVoltageLow, is_voltage_low);
}
```

VII. States example check

```
class BatteryVoltageLow : public State
{
    const bool &is_battery_voltage_low = getBoolVariableByID(StateVariableID::BatteryVoltageLow); ///< Get the
published state variable

public:

    BatteryVoltageLow(states::StateID id) : State(id){}

    void check()
    {
        if (this->is_battery_voltage_low) this->setActive(); ///< Set the state active and publish
*BatteryVoltageLow* state
    };

    bool needCheck()
    {
        return true;
    };
};
```

VIII. Global variables

```
# Topic data
# This contains a list of the most up-to-date values received on topics.
# These aren't normal python objects so it's best to use the get_field() function in topic_data_extractor.py
global topic_data
topic_data = {}

# Buffered topic data
# This contains a list of values received on topics. These values are updated once every cycle
# These aren't normal python objects so it's best to use the get_field() function in topic_data_extractor.py
global buffered_topic_data
buffered_topic_data = {}

# The service interface variable contains function-pointers to the service proxy functions
# These are used in the initial_commands() of the fsm_handler.py. It is possible to use them in any
operation.py file.
global service_interface
service_interface = {}

# The service_response variable is used to store the latest data reveived from a service
# It is possible to use them in any operation.py file.
global service_response
service_response = {}

# The current_state variable holds the transitions, failures and initial commands of the current state
# These parameters are all used in fsm_handler.py
global current_state
current_state = None

# The state_transition_time is the UNIX timestamp that gives us the moment we swithed states.
# This can be read out in any of the operation.py file.
global state_transition_time
state_transition_time = None

# The node_time is the UNIX timestamp that gives us the moment we started the node
# This is used in the timout parameter of fsm.py for the timeout failure.
# It can also be used in any of the operation.py file
global node_time
node_time = None

# The user_variables variable is a dictionary that holds custom user variables.
# It isn't used by the system but can be used in conditions and in any operations.py file.
global user_variables
user_variables = {}

# list of executed states
global executed
executed=[]

# var to enable or not transition according to the executed services
global move_to_next_state
move_to_next_state = None

# bool to control the spelling check
global loaded
loaded = None

# bool to control topics spelling
global topic_name_error
topic_name_error = None

#bool to stop the simulation f the safety stop is enabled
global safety_stop
safety_stop = None

# variable to control the blade lock
global ready_to_lock_blade
ready_to_lock_blade = False
```

IX. Log Analyzer

Topics & Messages

The Analyzers use a variety of topics and messages to extract data. They are listed in the following section:

topic:

- '/prodrone/monitor_data'

messages:

- 'autopilot_flight_mode'
- 'flight_phase'
- 'autopilot_gps_quality'
- 'altitude'
- 'blade_locked'

topic:

- '/prodrone/system_monitor/gcs_response_time_ms'

messages:

- 'data'`

topic:

- '/prodrone/camera_monitor_data'

messages:

- 'number_of_photos_made'
- 'number_of_photos_failed'

topic:

- '/prodrone/system_monitor/system_load'

messages:

- 'cpu_utilization_percentage'
- 'used_memory_MB'

topic:

- '/prodrone/gimbal/orientation'

messages:

- 'quaternion'

topic:

- '/dji_sdk/attitude_quaternion'

messages:

- 'q0', 'q1', 'q2', 'q3'

topic:

- '/mavros/local_position/pose'

messages:

- 'position.position.x'
- 'position.position.y'

topic:

- '/tf'

messages:

- 'transforms.child_frame_id.transform.translation.x'
- 'transforms.child_frame_id.transform.translation.y'

Parameters - Values inside resources

During the execution of Analyzers, checks are made for specific values. They are collected here:

Analyzer: InspectedSides

log_file_data_to_check = 'Blade tip altitude:'

source: Logfile, Rosbag

topic/msg:

Taken from topic: '/prodrone/monitor_data'

messages: 'flight_phase'

parameters:

sides_executed - list with the sides that have inspected

normal_operation_number_sides_executed - constant, define number of executed sides in an inspection

description:

Check if during the operation how many sides have been inspected.

First, we find the time range when 'blade_tip_altitude' appears in the log file

Then we use that time range to look which states where executed (we do not care about transitions).

A correct inspection should have 4 sides executed (phases 1-3-6-7).

Analyzer: GpsQuality

bad_quality = 0

source: Rosbag

topic/msg:

Taken from topic '/prodrone/monitor_data'

messages: 'autopilot_gps_quality'

parameters:

bad_quality - constant, define the unworkable quality

description:

Check if the autopilot GPS quality is 0.

Analyzer: PhotoCounter

source: Rosbag

topic/msg:

Taken from topic: '/prodrone/camera_monitor_data'

messages: 'number_of_photos_made', 'number_of_photos_failed'

parameters:

None

description:

Check if during the operation, some photos have failed to be taken.

Analyzer: ManualControl

dji_manual_autopilot = 'MANUAL'

px4_manual_autopilot = 'POSIX'

source: Rosbag

topic/msg:

Taken from topic: '/prodrone/monitor_data'

messages: 'autopilot_flight_mode' ,'flight_phase'

parameters:

dji_manual_autopilot - constant, define the manual state of autopilot

px4_manual_autopilot - constant, define the manual state of autopilot

Depending on the autopilot one of those will be assigned to manual_autopilot

description:

Check if during the operation manual control was taken, and show the side and how many times manual intervention happened.

Analyzer: NodeCrash

to_find = '__name:='

has_died = 'has died!'

source: Log file

parameters:

to_find - constant, define how the name of the node appears on the log

has_died - constant, define how the dead node appears on the log

description:

Check if a node fails and output the name and the line when it crashes

Analyzer: OdroidShutdown

to_find = 'Shutdown requested ...'

source: Log file

parameters:

to_find - constant, define how the shutdown command appears on the log

description:

Check if shutdown command was sent.

Analyzer: SetpointFollow

tf_frame_name = 'setpoint'

source: Rosbag

topic/msg:

Taken from topic: '/prodrone/monitor_data' | topic:'/dji_sdk/local_position'

messages: 'flight_phase' | messages: 'x', 'y'

topic: '/tf' | topic:'/mavros/local_position/pose'

messages: 'transforms' | messages: 'position'

parameters:

max_allowed_drift = 2 - constant, define maximum drift from setpoint allowed during normal operation

tf_frame_name - constant, how the setpoint name frame appears inside the /tf.transforms.child_frame_id

phase_eu_dist = variable, contains all the drifted positions for each phase

description:

For each flight phase, get the local_position and the setpoints.

Make a check for each local_position with the setpoints of 0.15 sec margin, if the Euclidean distance is bigger than the one allowed.

For now, the plot of the distance of the points is given.

Analyzer: HubReached

source: Rosbag

topic/msg:

Taken from topic "/prodrone/monitor_data"

messages: 'flight_phase', 'altitude', 'blade_locked'

parameters:

leading_operation_outcome - constant, True if the hub has been reached before switching to leading edge

trail_operation_outcome - constant, True if the hub has been reached before switching to trailing edge

description:

Check if the hub has appeared before leading and trailing edge:

1. Get the timestamp and alt when switching to LE
2. Get the timestamp of the first appearance of that altitude
3. Given that time-lapse (first appearance-switch) check for blade unlocked
4. Get the last timestamp of leading edge and look for the first timestamp of trailing

(with this approach, we are independent of the beginning inspection side)

5. Get the first timestamp of trailing (if exists)
6. Get the timestamp of the alt (of step 1) with an offset equivalent to the max_alt
7. Given that time lapse (alt appearance-trailing) check for blade unlocked

Analyzer: GCSConnection

source: Rosbag

topic/msg:

Taken from topic 'prodrone/system_monitor/gcs_response_time_ms'

messages: 'data'

parameters:

max_ping - constant, define maximum ping

description:

Check if the ping exceeds the defined max_ping.

Analyzer: SystemLoad

source: Rosbag

topic/msg:

Taken from topic '/prodrone/system_monitor/system_load'

messages: 'cpu_utilization_percentage', 'used_memory_MB'

parameters:

max_cpu_load - constant, define maximum cpu usage allowed during normal operation

max_ram_load - constant, define maximum memory usage allowed during normal operation

description:

Check how much CPU power is user

Check how much RAM memory is used.

Analyzer: VibrationsGimbal

source: Rosbag

topic/msg:

Taken from topic: '/prodrone/gimbal/orientation'

messages: 'quaternion'

parameters:

yaw_margin - constant, define the margin of yaw

pitch_margin - constant, define the margin of pitch

vibrations - list of timestamp and yaw or pitch value

description:

Check if during the operation yaw and pitch exceeded the margin value.

Analyzer: VibrationsDrone

source: Rosbag

topic/msg:

Taken from topic: '/dji_sdk/attitude_quaternion'

messages: 'q0', 'q1', 'q2', 'q3'

parameters:

yaw_margin - constant, define the margin of yaw

pitch_margin - constant, define the margin of pitch

vibrations - list of timestamp and yaw or pitch value

description:

Check if during the operation yaw and pitch exceeded the margin value.

X. Multiple Runners - One Machine

Compilation job

```
init:compilation:
 stage: compilation
 script:
  - rm /cache/* -rf
  - source /root/.bashrc
  - source /opt/ros/jade/setup.bash --extend
  - source /root/prodrone/supported_autopilots/install/setup.bash --extend
  - source /root/prodrone/supported_autopilots/devel/setup.bash --extend
  - ls /builds/root/ci-uav_control
  - cd /builds/root/ci-uav_control
  - catkin_make
  - catkin_make run_tests
 tags:
  - workstation
 artifacts:
   paths:
     - /builds/root/ci-uav_control
   expire_in: 1 day
```

SITL job

```
sitl_name:
 stage: test
 script:
  - INIT_YAML="px4_init_parameters.yaml"
  - mkdir -p /test_job/uav_control
  - mkdir -p /builds/root/ci-uav_control/survey_data
  - cd /
  - ln -s /builds/root/ci-uav_control/survey_data /root/survey_data
  - cp -arf --preserve=all builds/root/ci-uav_control/* /test_job/uav_control
    - ls /test_job/uav_control/src/prodrone/tools/common_gcs_uav
  - rm /test_job/uav_control/src/prodrone/tools/common_gcs_uav
  - ln -s $(readlink -f builds/root/ci-uav_control/src/prodrone/tools/common_gcs_uav)
/test_job/uav_control/src/prodrone/tools/common_gcs_uav
  - ls /test_job/
  - ls /test_job/uav_control/src
  - . /test_job/uav_control/src/prodrone/test/continuous_integration/tools/launch_sitl.sh
 dependencies:
  - init:compilation
 artifacts:
  when: on_failure
  name: "$CI_COMMIT_REF_NAME"
  paths:
   - /builds/root/ci-uav_control/survey_data
  expire_in: 1 day
```

Rosbag job

```
rosbag_name:
 stage: test
 script:
  - to_get_rosbag="rosbag_to_execute.bag"
  - INIT_YAML="AUTOPILOT_init_parameters.yaml"
  - mkdir -p /test_job/uav_control
  - cd /
  - ls builds/root/ci-uav_control/
  - cp -arf --preserve=all builds/root/ci-uav_control/* /test_job/uav_control
  - rm /test_job/uav_control/src/prodrone/tools/common_gcs_uav
  - ln -s $(readlink -f builds/root/ci-uav_control/src/prodrone/tools/common_gcs_uav)
/test_job/uav_control/src/prodrone/tools/common_gcs_uav
  - mkdir -p /test_job/uav_control/src/prodrone/test/continuous_integration/tools/CI_control/rosbags
  - ls /test_job/uav_control/
  - ls /test_job/uav_control/src
  - . /test_job/uav_control/src/prodrone/test/continuous_integration/tools/launch_rosbag.sh
 dependencies:
  - init:compilation
```

XI. Common issues

Troubleshooting commands.

For troubleshooting use: sudo gitlab-ctl tail or less /var/log/gitlab/prometheus/current. To check the status of GitLab services, run: sudo gitlab-ctl status.

Too many artifacts.

The artifacts are deleted automatically after one day. Rarely running a lot of pipelines in a short period results in creating too many artifacts resulting in the following error:

```
ERROR: Uploading artifacts to coordinator ... failed
```

By default, the artifacts are stored in /var/opt/gitlab/gitlab-rails/shared/artifacts. To fix this error remove the content of this folder manually.

Corrupted LevelDB database.

sudo gitlab-ctl tail shows the following error:

```
2017-08-27_20:04:18.79395 time="2017-08-27T22:04:18+02:00"
level=error msg="Could not open the fingerprint-to-metric index for archived series. Please
try a 3rd party tool to repair LevelDB in directory
\"/var/opt/gitlab/prometheus/data/archived_fingerprint_to_metric\".
If unsuccessful or undesired, delete the whole directory and restart Prometheus for crash
recovery. You will lose all archived time series." source="persistence.go:213"
```

and sudo gitlab-ctl status shows that Prometheus is down.

To solve this:

- Install leveldb Python module using pip.
- Run:

```
sudo -u gitlab-prometheus python -c "import leveldb;
leveldb.RepairDB('/var/opt/gitlab/prometheus/data/archived_fingerprint_to_metric')"
```

Wrong IP address on startup.

Check the IP address when you start bitnami-gitlab on the VM. If it's different than 198.168.1.37 the changes made while configuring the external_url will not have an effect. To fix this, change the IP address which is assigned by the router.

XII. Docker image update

This manual will discuss specific workflows needed to maintain and update the CI testing environment.

Updating Docker images

Login to Pro-Drone's private repository on Docker by executing the following command:

sudo docker login

and enter the credentials.

Now the CI image an existing image can be updated or a new image can be created.

Creating a new image

To build the new image run:

sudo docker build -t "prodrone/prodrone:tag-name_of_image"

After the new image is built, upload it back to the Docker repository. This can be done by using the following command:

sudo docker push prodrone/prodrone:*image name*

Updating an existing image

Launching the image

An existing image can also be updated to cut down on time. First, pull the latest image that is used by the CI. This can be done by the following command:

sudo docker pull prodrone/prodrone:*image name*

At the moment of writing the name is "runner_image". With the image downloaded a new container can be opened from it with a bash instance running inside it. This can be done via the following command:

sudo docker run -it prodrone/prodrone:*image name* /bin/bash

If we want to open extra terminals, we first need to know the container ID. We can find this by executing the following command:

sudo docker ps -a

Now we have the container ID we can open more bash instances on the container by executing the following command:

sudo docker exec -i -t *container id* /bin/bash

This can be done multiple times.

Saving and uploading the image

After all changes have been made, the Docker container can be closed by executing:

sudo docker stop *container id*

Now the container can be converted to a new image. This is quite simple with the following command:

sudo docker commit *container id* prodrone/prodrone:*new image name*

The newly made image can be uploaded to the Docker repository:

sudo docker push prodrone/prodrone:*new image name*

After the upload is complete, it can be used in the CI. To use the new image, the ".gitlab-ci.yaml" file needs to be updated. The first line in this files defines the used image:

line 1: image: prodrone/prodrone:*new image name*

Non-exclusive licence to reproduce thesis

I, Dzvezdana Arsovska,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for the purpose of preservation in the DSpace digital archives until the expiry of the term of copyright,

Building an Efficient and Secure Software Supply Pipeline for Aerial Robotics Application,

supervised by Assoc. Prof. Karl Kruusamäe and Mr. Illia Sheremet,

Publication of the thesis is not allowed.

2. I am aware of the fact that the author retains the right specified in p. 1.

3. This is to certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Tartu
25.12.2018