



Expertise  
and insight  
for the future

Jere Raassina

# DevOps and test automation configuration for an analyzer project

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Software Engineering

Bachelor's Thesis

Author Title	Jere Raassina DevOps and test automation configuration for an analyzer project
Number of Pages Date	68 pages 31 October 2020
Degree	Bachelor of Engineering
Degree Programme	Tieto- ja viestintäteknikka / information technology
Professional Major	Software engineering
Instructors	Janne Salonen, Head of School (ICT)
<p>The goal of this thesis was to study and investigate different DevOps, software testing, project management and test automation subject matters and conduct a related project consisting of designing and implementing a continuous integration and test automation environment for a medical analyzer project.</p> <p>The project was mostly completed on an existing development and work management platform that was already in use by the company. It enhanced previous features by introducing better build and release management and established new functionality such as continuous integration pipelines, a test automation environment and reporting services.</p> <p>The goal of implementing a stable DevOps and continuous integration platform as well as creating a solid basis for the test automation environment was achieved and is available for further development and the official verification &amp; validation of the automated tests in the future.</p>	
Keywords	DevOps, Test automation, Software testing, Robot Framework

<p>Tekijä Otsikko</p> <p>Sivumäärä Aika</p>	<p>Jere Raassina DevOps and test automation configuration for an analyzer project</p> <p>68 sivua 31.10.2020</p>
<p>Tutkinto</p>	<p>insinööri (AMK)</p>
<p>Tutkinto-ohjelma</p>	<p>Tieto- ja viestintäteknikka / information technology</p>
<p>Ammatillinen pääaine</p>	<p>Software engineering</p>
<p>Ohjaajat</p>	<p>Janne Salonen, Tieto- ja viestintäteknikan osaamisaluepäällikkö</p>
<p>Insinööriyön tarkoituksena oli tutkia ja tarkastella DevOpsin, ohjelmistotestauksen, projektinhallinnan ja testiautomaation eri osa-alueita sekä toteuttaa aihepiireihin liittyvä projekti. Projektin tavoitteena oli suunnitella ja kehittää jatkuvan integraation sekä testiautomaation mahdollistava ympäristö lääketieteelliseen analysaattoriprojektiin.</p> <p>Suurin osa projektista toteutettiin jo yrityksen ennalta hyödyntämään kehitys- sekä työnhallintaympäristöön ja järjestelmään, jonka aiempaa toiminnallisuutta parannettiin parempien paketointi- sekä julkaisuominaisuuksien avulla. Tämän lisäksi kehitettiin uusia ominaisuuksia kuten jatkuvan integraation putkia, raportointitoimintoja sekä kokonainen testiautomaatioympäristö.</p> <p>Varsinaisessa projektityössä hyödynnettiin useita aiheeseen liittyviä työkaluja ja ympäristöjä, kuten Azure DevOps Serviceä, Robot Frameworkia, Pythonia sekä PywinAutoa.</p> <p>Vakaan DevOps ympäristön, jatkuvan integraation perustan sekä yhtenäisen testiautomaatioympäristön projektitavoitteet saavutettiin ja näin ollen mahdollistettiin myös niiden jatkokehitys sekä valmistelu tulevaisuuden virallista verifikaatiota ja validointia varten.</p>	
<p>Avainsanat</p>	<p>DevOps, Test automation, Software testing, Robot Framework</p>

## Contents

### List of Abbreviations

1	Introduction	5
2	DevOps	6
2.1	Meaning, purpose and flow	7
2.2	Continuous integration	10
2.2.1	Version control	10
2.2.2	Integration pipelines	11
2.3	Continuous delivery	12
2.4	Continuous deployment	13
3	Software testing and project management	15
3.1	Software life cycle and testing models	15
3.1.1	Waterfall model	15
3.1.2	V-model	17
3.1.3	Agile model	18
3.1.4	Iterative model	20
3.1.5	Spiral model	22
3.2	Testing process	25
3.3	Test artifacts	27
3.4	Testing approaches	29
3.4.1	Black box testing	30
3.4.2	White box testing	31
3.4.3	Grey box testing	32
3.4.4	Exploratory testing	32
3.5	Software testing levels, -types and techniques	34
3.5.1	Testing levels	34
3.5.2	Testing types and techniques	35
3.6	Automated testing	37
4	Project tools, requirements, and execution	37
4.1	Requirements and the goal	37
4.2	Servers, services, and other tools	38
4.3	Azure DevOps	40

4.3.1	Version control	40
4.3.2	Software agents	41
4.3.3	Continuous integration and deployment pipelines	43
4.3.4	Azure portal, reporting and other features	48
4.4	Automated testing and Robot Framework	50
4.4.1	Environment and prerequisites	51
4.4.2	Test structure	53
4.4.3	Keywords and syntax	55
4.4.4	Libraries and tools	56
4.4.5	Test results and output	58
4.4.6	Automation and other tools	60
5	Results, improvements and conclusion	62
5.1	Alternative tools	63
5.2	Future development ideas	64
	References	65

## List of Abbreviations

IVD	In vitro diagnostics. clinical tests that analyze samples taken from the human body.
V&V	Verification and validation. The process of vivifying and validating a correct product functionality and operation.
SDLC	Software development life cycle. A clearly defined process for creating software products containing of different phases or stages.
SAFe	Scaled Agile Framework. A set of practices and patterns for scaling lean and agile practices.
ARA	Application-release automation. Process of packaging and deploying an application to production across various environments.
CCA	Continuous configuration automation. Process of deploying and configuring the settings and software automatically for virtual and physical targets.
CVS	Centralized version control system. Version control system based on the centralized server ideology.
DVCS	Distributed version control system. Version control system based on the distributed cloning ideology.
STLC	Software testing life cycle. A clearly defined process for testing software products containing of different phases or stages.
AUT	Application under test.
UML	Unified Modeling Language. Broad-purposed and developmental approach on a modeling language.
API	Application Programming Interface. A computing Interface that defines the interactions between multiple software emissaries.

.NET	A Microsoft software framework for Windows platforms.
PAT	Personalized Access Token.
GUI	Graphical User Interface.
CLI	Command-line interface
REST	Representational state transfer. A set of architectural constraints used for creating web services.
PATH	An environment variable that defines the directories where executable programs are located.
HTML	Hypertext Markup Language. Markup language standard for documents displayed in web browser.
COM	Component Object Model. The standard binary interface for Microsoft software components.
SQL	Structured Query Language. A domain specific language for relational databases.

## 1 Introduction

The subject for this thesis covers a study and investigation on DevOps related topics such as continuous development practices, software testing and project management areas as well as a related project in a clinical analyzer project environment. The project work consists of the study on finding out the suitable tools for the project, planning and learning the preferred ways of using the tools and implementing and taking the DevOps related features, continuous integration and the test environment in use. The idea was to create a good basis for the mentioned services and features that could be easily expanded and developed for further usage.

This study and project work were conducted by and presented to Thermo Fisher Scientific as a part of the Cascadion SM Clinical Analyzer project. The company and the project managers introduced a need for a modern Continuous integration and test automation solutions as other product lines had already started to plan and implement them for their specific purposes. The testing practices are an important part of the working process as the company works in the IVD field and the some of the products are used to analyze human based samples. The verification and validation or V&V processes being such significant also means that they tend to often be very time consuming and repetitive while still maintaining the need of high accuracy. The eventual expectations of the project would include the increase in DevOps methodology and mentality as well as the official verification and validation of the test automation environment.



## 2 DevOps

The Agile ways of working have gained remarkable popularity in the past few decades and many of the companies working in the software field have either completely shifted or are slowly moving towards this new way of thinking. The Agile methodology focuses heavily on the new ways and ideologies on how the team works together and acts during the development process. One of the key driving points is the idea of cross-functional teams that aim to be as self-organized and community focused as possible. This means that the team is trying to steer away from the idea of specific roles and manager positions and make their own decisions by utilizing the Agile ideology and by following the provided frameworks. [1.] The fundamentals of the Agile workflow can be summarized in the so-called Agile manifesto defined by The Agile Alliance: Individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation and responding to change over following a plan. [2.]

The so-called Lean thinking is also a big and important part of the Agile culture. Lean emphasizes the idea of eliminating all unnecessary work and effort by aiming for the maximum value for customer and the team itself. This is often considered to be best achievable by experimenting, failing-, starting over- and delivering as fast as possible as well as learning and optimizing after each working cycle. [3.] Much like in the Agile manifesto, Lean also sums its key values in the list of alleged Lean principles: Eliminate waste, amplify learning, decide as late as possible, deliver as fast as possible, empower the team, build integrity in, optimize the whole. [4.]

These two frameworks can be referred as their own ideologies but are often considered to be very similar and consist of similar values. Both highlight the ideas of fast reaction and response to problems and the concept of changing over long-term planning. This leads into a conclusion that accurate and timeboxed planning, real time tracking of the work items and the smooth development, testing and deployment processes must be well thought out. The term and concept of DevOps was invented in order to accommodate all these requirements and the different parties working with them. [5; 6.]

## 2.1 Meaning, purpose and flow

The term DevOps being a manifestation of the Agile and lean concepts, aims to combine the collaborative actions to create and manage a successful design, development and delivery process. The idea is to incorporate teams such as the traditional development and operations teams that consist of software designers, developers, testers, delivery- and deployment engineers and other parties that take part in the DevOps process. This allows a smooth operations cycle without forming traditionally problematic silos and other historical problems between the participating teams. [5; 6.]

The term DevOps has however gained reputation as a globally undefined buzzword and has not therefore developed an academically unique definition. This means that multiple meanings and interpretations can be found and are commonly used for the term. [7, p. 2, p. 5.]

Figure 1. shows an illustration of the typical DevOps toolchain and the flow of the different phases.

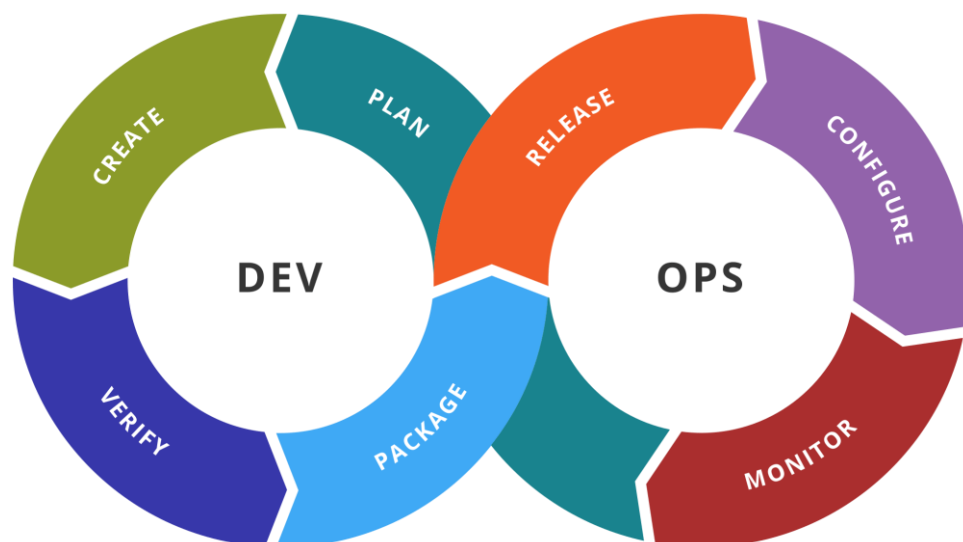


Figure 1. Illustration of the DevOps toolchain. [8.]

In order to achieve and maintain the DevOps workflow and unify the working methods, so called toolchains are commonly used. These toolchains are essential for the modern DevOps oriented way of software development and provide the fundamental tools for the concept of automation as well as the more traditional methods. [9; 10.]

The following chart (Table 1) displays a common DevOps toolchain and introduces examples of the possible tools or ways of working for each phase. The toolchains are usually defined by the team and can be modified case by case to follow the Software Development Life cycle or the SDLC.

Table 1. A common DevOps toolchain with stages and examples. [9; 10]

Phase / stage	Purpose	Examples and tools
Plan	Defining and planning with preferably the whole DevOps or project team.	Either a long-term plan or the use of Agile methods such as Scrum or SAFe.
Create and develop	Software design, development and possible unit testing.	Version control such as Git, Different development environments and software stacks, unit testing tools suitable for the language and the environment. Heavy focus on continuous integration.
Verify	Different testing methods and phases. Ensuring the quality of the code and the functionality of the software.	Integration-, acceptance-, regression-, security and system level testing. Test automation Etc.

Package	Configuring, packaging and staging the software for the release.	Various package managers.
Release	Deploying and releasing the software to its respectable target environments. Can include multiple sub phases from scheduling and provisioning to more testing features.	Various manual or automated processes across various environments and pipelines. Heavy focus on continuous delivery. Many application-release automation (ARA) solutions available.
Configure	Configuration of infrastructure storage and database, network and application provisioning.	Different configuration management, continuous configuration automation (CCA) and infrastructure as code solutions.
Monitor	Monitor and identify problems and analyze how they affect the end-users. Enhance methods and ways of working based on the information.	Use cases include performance and reliability monitoring, user response and experience feedback and production metrics and statistics.

## 2.2 Continuous integration

As the idea of centralized and unified DevOps practices have emphasized the need of the source code to exist in one robust place and be easily accessible for development or release purposes, the use of remote source control tools has become commonplace. The main purpose for continuous integration development practices focus on the ability for multiple developers to integrate and fetch software code into a shared remote repository at any time. These integrations are often verified by using automated building and testing of the software and therefore making it easier to find possible problems or defects and prepare it for release and delivery. It is common to also automate the deployment of the software at least on some level and depending on the scale of the project, whether it is just creating an internal release or preparing it for the official delivery. [11; 16.]

### 2.2.1 Version control

Whether the codebase needs to be modified by a developer for a new feature or a bug fix or a build or a patch is needed for a new release, it must be constantly available and ready for actions. In revision control systems this is done by using a data structure called repository to store metadata for a directory structure or a set of files. [12; 13]

There are two types of commonly used version control systems that utilize different technologies to store and distribute the source code and data. Centralized version control systems follow the principle of having a single centralized repository containing the project code stored on a server or on some other hosting service. Every commit made to the repository is recorded and will automatically update any changed files or change-sets upon pulling the contents. Some common centralized version control systems include Subversion, Perforce and CVS. [12; 13.]

Distributed version control systems or DVCS on the other hand apply the concept of each contributor having their own server and a working copy of the remote repository on their own workstation or server. This copy includes records of all repository branches and a complete history of the code allowing local access and the possibility to work without internet connection. The most popular distributed version control solutions include Git, Mercurial and Bazaar. [12; 13.]

Both systems have their pros and cons and are commonly used in software projects of all sizes and all around the world, while distributed systems have started to take a lead in the popularity. DVCS benefit from the ability to work offline, better performance for not being reliant on fast internet connection to avoid locks or other networking problems, easier branching and merging with powerful change tracking and not being dependent on the main server's performance or stability while retaining the possibility to get a backup from any development instances. The differences of the version control systems are displayed in figure 2. [12; 13.]

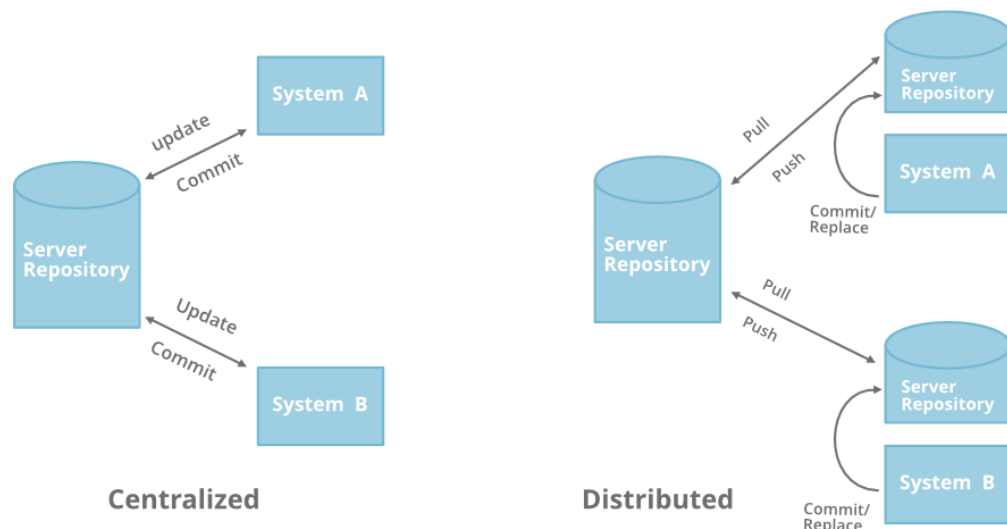


Figure 2. Graphical illustration of centralized- and distributed version control systems. [12.]

### 2.2.2 Integration pipelines

The concept of software pipelines was established to make the implementation, functionality and accessibility of the continuous integration development practices as clear and visible as possible. There are multiple use cases and solutions for the usage of the pipelines, while most common of them in the sense of continuous integration is the automation and running of the build and integration processes to test and verify the development product after each concrete addition, change or pull request made to the codebase. [14.]

The pipelines consist of different stages with different jobs based on the purpose of the pipeline. These stages and jobs are usually fully automated while providing logs and other possible visualization to the development team. The order of the stages is important as the tasks are often dependent on each other or on the actions performed in

the previous stages, or in some cases even on different pipelines. Every failure or error situation is recorded during the runtime of the pipeline and an appropriate message or notification is sent to the responsible persons through various channels such as email, instant messaging platforms or the continuous integration platform itself. [15.]

The stages of the pipelines are designed to be fully customizable depending on the tool and the needs of the project but most common software releases that implement the DevOps practices follow the same common stages of version control, continuous integration, continuous delivery and continuous deployment as displayed in the figure 3.

### 2.3 Continuous delivery

Continuous delivery is commonly seen as an extension of continuous integration with the focus on releasing and delivering the latest changes on the software to the customers as fast and smoothly as possible. The main thing to be added on top of the basic continuous integration and its automated building and testing is the automation of the release process. This kind of procedure combined with the passing of the tests in the earlier phases means that the software product can be deployed at virtually any given time if the developers so decide. A common practice is to decide on a release schedule that suits the business requirements and the development cycles of the software project. It is also advised to keep the deployments small enough and deploy them to production as early as possible to make the possible troubleshooting easier. [14; 15; 16.] There are multiple requirements and benefits introduced with the application of continuous delivery. These are displayed in table 2.

Table 2. The requirements and benefits of continuous delivery. [14; 16.]

Requirements	Benefits
<ul style="list-style-type: none"> <li>- Strong continuous integration infrastructure and large enough test coverage.</li> <li>- Manually triggered automated deployment without human intervention.</li> <li>- More dependent on stable release cycles and the sizes of the deployments.</li> <li>- Features and the development cycles need more attention so that the incomplete features do not affect the production and the customers.</li> </ul>	<ul style="list-style-type: none"> <li>- The software iterations will be faster due to the lowered risks and decision making related to small changes.</li> <li>- Faster test results and customer feedback due to the accelerated release pace.</li> <li>- Way less time and effort spent on preparing and commencing the software releases.</li> <li>- Less dependency on operations with enhanced security and integrated compliance.</li> </ul>

## 2.4 Continuous deployment

Continuous deployment is the next extension for continuous delivery and considered to be the final step of the modern DevOps pipeline structure. At this point all the integration, testing and release processes have been completely automated, and it is possible to deploy and deliver the software directly to the customer without any human intervention or other actions. If the complete process has been designed and implemented well enough and the tests can be completely trusted, the need for separate release schedules and procedures become insignificant. Therefore, the focus of the developers can be completely directed towards developing the software itself, while maintaining the DevOps practices on the side. A good example is a customer website or a service that runs through the whole process and is directly updated and the feedback is received straight away. [16.] The requirements and benefits are introduced in the table 3.



Table 3. The requirements and benefits of continuous deployment. [14; 16.]

Requirements	Benefits
<ul style="list-style-type: none"> <li>- The testing solution needs to be complete and as reliable possible since there is a direct relation to the deployment of the system.</li> <li>- Monitoring and planning of the development is especially important as all the passed builds have a direct impact on the complete system and the customer.</li> <li>- The documenting system other procedures need to be able to keep up with continuous deployments.</li> </ul>	<ul style="list-style-type: none"> <li>- Faster, smoother and more flexible development times and processes.</li> <li>- The automated processes make the improvements to the DevOps environment easier in the future.</li> <li>- The deployments tend to be smaller and therefore less risky and easier to fix if needed.</li> <li>- The continuous results and improvement can be directly seen by both the customer and the developers.</li> </ul>

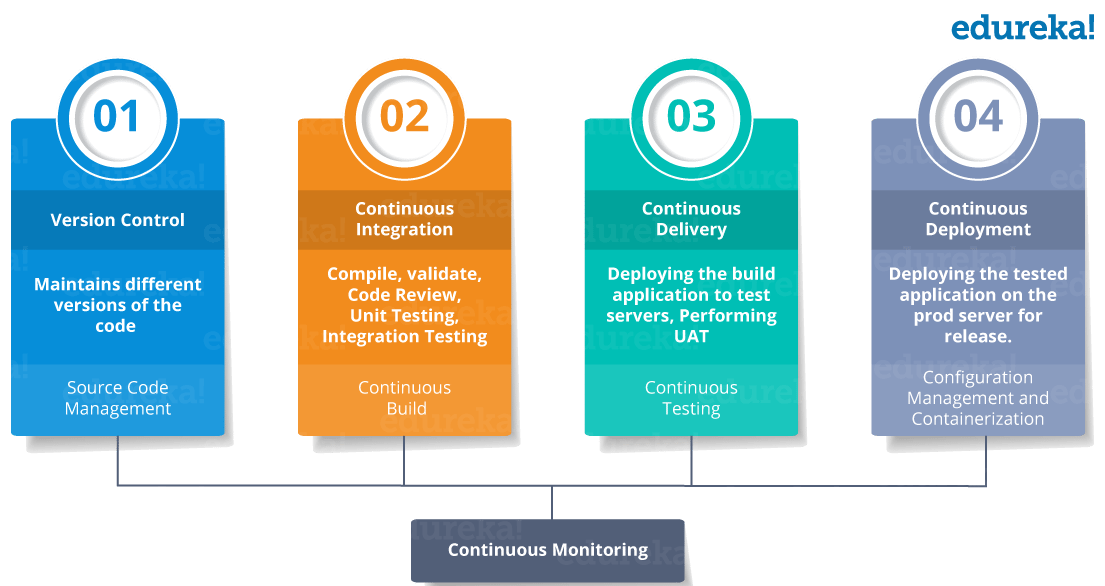


Figure 3. The complete process of modern DevOps development pipeline. [15.]

### 3 Software testing and project management

From the days of manual testing to the future of ever-increasing number of automated tests and smart processes, the concrete idea of software testing has remained simple, yet important. Keeping the quality of the software high by evaluating its performance and functionality by finding out whether it meets the given specifications and requirements, while also remaining as bug free as possible has been the main goal since the beginning of software testing. With the increased value seen in the testing, the development and release processes of every production or commercial level software company should and will nowadays incorporate some sort of testing practices in their ways of working. [17; 18.]

#### 3.1 Software life cycle and testing models

Without going too much in detail with the testing content itself, the project and testing team usually sets on following a specified testing model and approach depending on the needs of the current project or other regulations or requirements. These ways of working are also highly dependable on the higher-level company demands and other entities such as the development teams. Some fields such as medical, banking or law related ones require more specific documentation, fulfilment of given standards and other regulations depending on the severity, complexity and the needed deliverables of the product under development. Each of the common SDLC models has its advantages and disadvantages and the more complex and often time-consuming processes may not be as flexible and are usually harder to change or modify. [17; 18.]

##### 3.1.1 Waterfall model

The waterfall model is often considered the most traditional and basic software development- and testing practice. The model is still widely used and consists of a sequence of different stages or processes which each are dependent on the previous one and therefore form the shape of a waterfall flowing down to the goal and the completion of the project. [18.]

The structure and the different stages of the waterfall model are shown in figure 4.

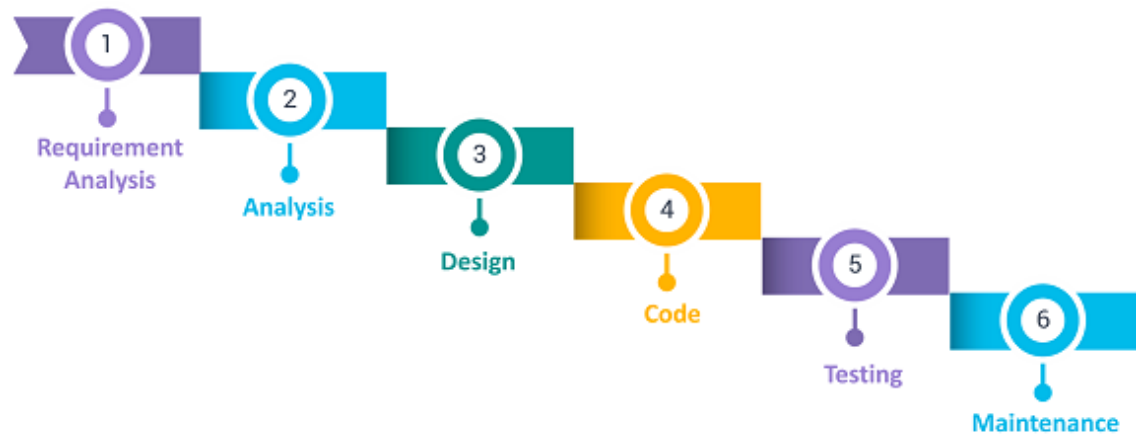


Figure 4. The stages of a software development waterfall model. [18.]

The waterfall model is most suitable for a project with the following conditions:

- Project duration is preferably short.
- Requirements are well defined and there are no undefined or ambiguous requirements.
- Resources, tools and manpower are available from the beginning in order to support the project.
- The product definition is stable, and the project is well documented.
- The technology and tool decisions are understood and in place.

Table 4. The advantages and disadvantages of the waterfall model. [18; 19.]

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>+ Easy to implement, follow and maintain</li> <li>+ Clearly defined stages with specific deliverables, review processes and elaborate documentation</li> <li>+ The requirements are clear as the phases are dependent on each other and the stages must be completed in order</li> <li>+ Very strict and quality assurance oriented</li> </ul>	<ul style="list-style-type: none"> <li>- Lack of adaptability across the stages</li> <li>- High amounts of risk and uncertainty as there is very little space for errors and delay</li> <li>- Very hard to change the original plan and requirements</li> <li>- High need for documentation that also eats other team resources</li> <li>- Testing period is located quite late in the waterfall process</li> <li>- Even small changes in the end product can cause a lot of problems</li> </ul>

### 3.1.2 V-model

The V-Model is often considered to be a remarkable upgrade or an extension for the waterfall model and regularly thought to be the solution for the disadvantages of the previous model. The biggest improvement compared to the waterfall model is the concept of associating each development stage with a corresponding test phase. This is done to combat the late testing stage and the laborious modification process after testing in the waterfall model. The V-model is formed by multiple verification and validation phases that are planned in parallel with the development and executed in a sequential format. These testing phases can also be directly associated to their counterparts on the other side of the model diagram, with development being the middle ground like displayed on figure 5. [20.]

Even with this approach it is still very much required to have as clear requirements as possible in the beginning of the project to minimize the possible unwanted or unscheduled changes to the final product. [20.]

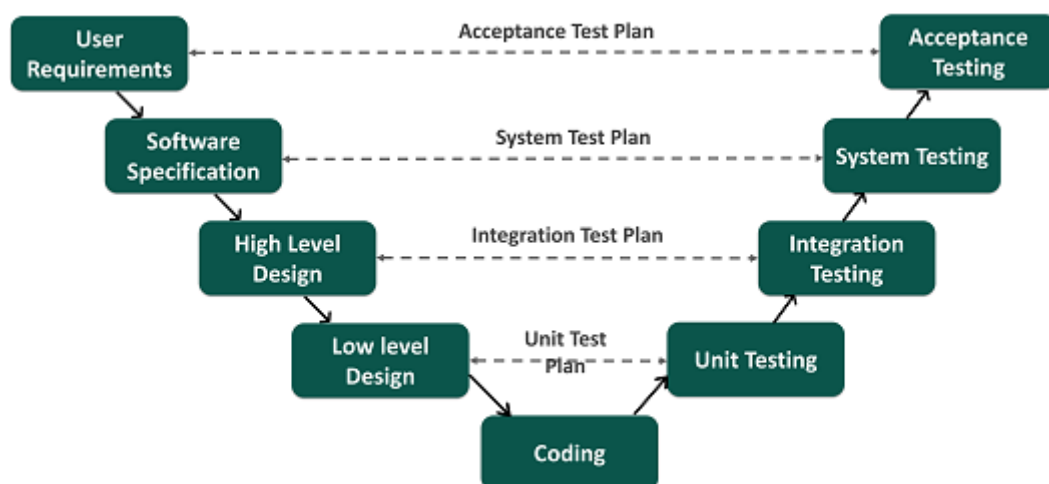


Figure 5. Visual representation of the V-model and the associations between the verification and validation stages. [18.]

The V-model is most suitable for a project with the following conditions:

- Preferably short project duration.
- Project within a strictly disciplined domain like the medical field.
- Well defined requirements, specifications and documentation similarly to the waterfall model since it is still not preferable to backtrack and make changes later in the project.
- The correct personnel and technology to accommodate the more complicated and laborious process.

Table 5. The advantages and disadvantages of the V-model. [17; 20.]

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>+ Easy to manage and highly disciplined due to the step-by-step structure and the rigidity of the model.</li> <li>+ Specific deliverables and review process for each phase.</li> <li>+ Good fit for smaller projects with clear requirements.</li> <li>+ Good success rate and defect tracking due to early planning and constant testing with multiple test phases.</li> </ul>	<ul style="list-style-type: none"> <li>- Not suitable for complex or long and ongoing projects.</li> <li>- Inflexible and difficult to respond to software or requirement changes during the testing phases.</li> <li>- Higher level of risk and uncertainty with less precisions.</li> <li>- Fully working software is produced only in the later stages and the development process is not clear to the client.</li> </ul>

### 3.1.3 Agile model

The concept of the Agile Model combines the incremental and iterative processes and focuses on process adaptability, flexible operations and customer satisfaction. This is achieved by rapid development, testing and delivery of working software in smaller increments completed in timeboxed iterations. Each iteration commonly lasts from one week to three weeks depending on the length and complexity of the project as well as the given resources. The flexible Agile functionality is carried out by using the concept of cross functional teams working simultaneously on all areas of the project and by comprehensively communicating and planning throughout the iterations. [21.] These areas are displayed in figure 6.



Figure 6. The working areas of the Agile focused used in the working iterations. [18.]

The Agile model is most suitable for a project with the following conditions:

- Projects with fixed schedule and the possibility to comply with the settled length of the iterations.
- Adjustable scope in order to preserve the set schedule.
- Adaptable project and team with cross functional teams.
- Compliant with organizing work into small deliverables.
- Committed to the idea of accepting and adapting to quick changes and delivering value incrementally over time.

Table 6. The advantages and disadvantages of the Agile model. [21.]

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>+ Little to no planning required with minimal resource requirements.</li> <li>+ Very realistic and suitable for software projects of multiple sizes, configurations and teams.</li> <li>+ Easy to manage with the ability to provide flexibility to the cross functional teams.</li> <li>+ Suitable for fixed or changing requirements and steadily developing environments.</li> <li>+ Rapid and concurrent development with the delivery of early functionality and partly working solutions.</li> </ul>	<ul style="list-style-type: none"> <li>- Makes the handling of complex dependencies hard.</li> <li>- High individual dependencies and depleted knowledge transfer due to the minimal documentation and the roles required to follow the Agile methodology.</li> <li>- Added risk of sustainability, maintainability and extensibility.</li> <li>- The scope, functionality to be delivered and the modifications of the project are dictated by the strict delivery management.</li> <li>- Highly dependent on customer and stakeholder interaction.</li> </ul>

#### 3.1.4 Iterative model

The iterative model is started with a small set of requirements needed for the functional part of the software which is then enhanced and developed in iterations until the complete system has been implemented and is ready for deployment. Each iteration produces a separate component or a version of the system which is then added or included to the earlier version or a specific function. Each of these components goes through all the iterative phases that include evaluation of the requirements, further design, development and implementation and finally the testing stage. The design and execution of the testing phase is a highly important part of the iterative process, and the meticulous and accurate verification and validation of the software components must be repeatable and extended throughout the process. [22.] The iterative process is illustrated in the figure 7.

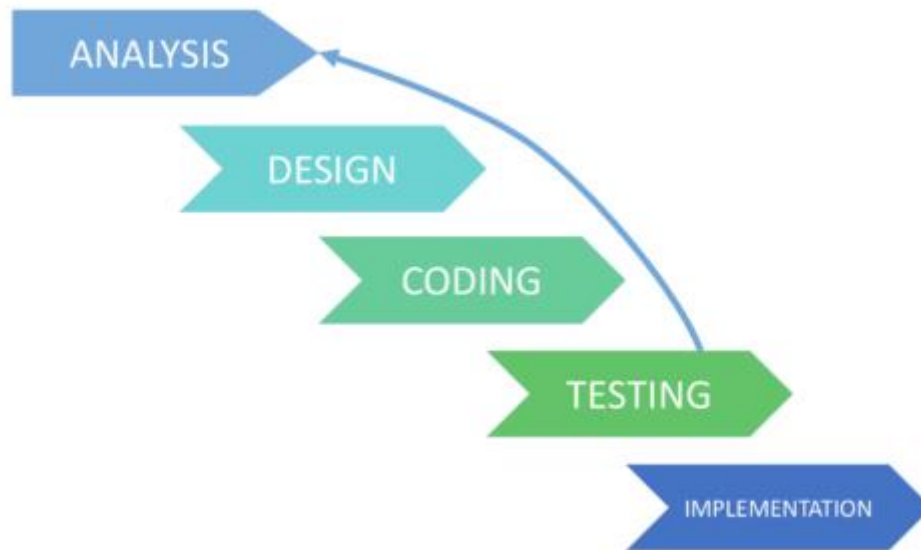


Figure 7. The order of the iterative model phases. [18.]

The Agile model is most suitable for a project with the following conditions:

- The prerequisites and specifications of the complete project are well understood and defined with minimal changes.
- A new technology is going to be applied and requires learning or configuration during the project.
- A changing need or availability in resources and skill sets based on iterations. For example, the use of contractors or consultants.
- Possible high-risk factors with increased probability of modifications.
- The market constraints are taken in account.



Table 7. The advantages and disadvantages of the Iterative model. [22.]

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>+ Early production of working functionality.</li> <li>+ Easily measurable early and periodical progress and results with clear milestones.</li> <li>+ Moderately easy and cheap to make changes to the requirements or the scope of the project.</li> <li>+ Very efficient testing, verification and validation progress that is easily split into iterations.</li> <li>+ Easier risk management with fast and early detection, decreasing risk factor and easy analysis.</li> <li>+ Suitable for larger scale and mission-critical projects.</li> </ul>	<ul style="list-style-type: none"> <li>- The process may require more project resources.</li> <li>- Changing requirements or project scope are still not recommended even with the lesser cost.</li> <li>- Higher attention from the management is required.</li> <li>- Architecture or design problems are likely if the requirements are not clear in the beginning.</li> <li>- Not very suitable for smaller projects.</li> <li>- The planning and definition of the iterations may require complete knowledge of the complete system.</li> <li>- High need for good risk analysis as the end goal may not be fully clear.</li> </ul>

### 3.1.5 Spiral model

The Spiral model is a combination of the systematic and controlled aspects of the linear models and the timeboxed approach of the iterative process models. It aims to fuse the ideas of sequential linear- and the iterative development together by having a high emphasis on risk analysis and by allowing the incremental releases and refinement over the spiral iterations. [23.]

The spiral model consists of four phases that are repeatedly passed through as the project progresses. The iteration starts with the identification and planning phase where the initial business requirements are gathered when the baseline spiral is formed for the first time. The need for prospective subsystem- and unit requirements are identified during the consecutive spirals and as the project develops. The monitoring and the understanding of the system requirements are also identified in close communication with system analysts and the customer. [23.]

The next phase focuses on design and risk analysis and outlines the baseline spiral with the conceptual design of the project and continues architectural design, physical product design, logical module design and the final design in the following spiral iterations. The Coding, building and implementation is the third Spiral model stage and starts with a baseline Proof of Concept that is developed and used to collect feedback for the further iterations. The later spiral repetitions with higher proof and knowledge of the requirements and desing are used to perform the actual development and testing. New build is prepared after each spiral cycle and sent to the customer for futher feedback. The fourth and final phase is called the evaluation and risk analysis phase where the estimating, identifying and monitoring of the management and technical risks are performed. After testing the software the customer actively evaluate the build from the previous iteration and produce feedback based on it. [23.]

The customer evaluation and feedback plays a huge role in the design and implementation decisions as the development and testing process moves from iteration to iteration in a linear fashion and by following the spiral model throughout the software life cycle as displayed in figure 8. [23.]



Figure 8. Representation of the flow of the spiral model. [18.]

The Spiral model is most suitable for a project with the following conditions:

- There is a high expectation of changes in the product during the development.
- If the product is new and should be released in phases in order to get enough feedback from the customers.
- The project requirements or specifications are complicated and need clarification and more evaluation.
- The requirements given are not clear or well enough defined by the customer.
- Preferably medium to high-risk projects.
- When budget constraints are probable and risk evaluation is needed.
- High and long-term commitment towards the project due to the possible changes to economic and other requirements during the execution of the model.

Table 8. The advantages and disadvantages of the Spiral model. [23.]

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>+ Good risk management due to the possibility to divide the development cycles into smaller parts and therefore leaving the riskier parts to be developed earlier.</li> <li>+ The proof of concept is developed early and is directly visible to the customer.</li> <li>+ Allows prototypes to be used and evaluated extensively.</li> <li>+ Accurate capturing and tracking of requirements.</li> <li>+ Possibility to accommodate changes in requirements and specifications.</li> </ul>	<ul style="list-style-type: none"> <li>- The length of the process may be unknown, and the spiral process may extend indefinitely.</li> <li>- Excessive documentation is required due to the large number of intermediate project stages.</li> <li>- Overall complex development and management processes.</li> <li>- Unsuitable for small or lower risk projects due to possible expenses.</li> </ul>

### 3.2 Testing process

Much like the bigger picture of the project level software development models, the testing parts or phases of the project can be sequenced in their own life cycles or STLCs. The testing models may vary depending on the chosen development model but often follow similar steps that produce similar output, documentation and deliverables. [24].

The testing life cycle starts with the *requirement phase* where the software requirements are analyzed and evaluated in order to find out whether the requirement in question can be tested or not. The goal is to help identify the scope of the testing needed for a certain feature or the development phase in question and develop a mitigation strategy for anything untestable. After the requirement phase the life cycle continues to the *planning phase*. The goal of this phase is to identify and define the activities and resources as well as to identify the tracking metrics in order to help meet the wanted testing objectives. The test strategy and the risk analysis for risk mitigation and management are also created during the planning phase. [24.]

The third phase is called the *analysis phase* that concretely defines what is to be tested. The test conditions are identified by going through the requirements, risks and other test bases while retaining the ability to trace the conditions back to the requirements. The conditions increase the test coverage and should be written with care since they will be later used as a basis for the test cases. There are multiple factors that affect the identification of the test conditions. These include the complexity of the product, the project risk factors, the development process chosen for the project, the management of the tests, the team and skill resources, the availability of the stakeholders and finally the overall depth and levels of testing. [24.]

The previous phase defined what needs to be tested and therefore the fourth stage called the *design phase* determines how the tests should be implemented and executed. The design phase consists of detailing the test conditions and breaking them down to multiple sub-conditions in order to increase test coverage, getting and identifying the test data, establishing and setting up the test environment and finally creating the test coverage- and requirement traceability metrics. After all the identifying and detailing it is time to create the actual test cases in the fifth phase known as the *implementation phase*. The comprehensive test cases are created and prioritized while also identifying which of them are going to be a part of the regression suite. Specified reviews are then held to ensure

the correctness and unity of the finished test cases. The possibility to automate any of the designed test cases is also evaluated during this phase. [24.]

The sixth phase marks the time to start the *execution phase* and put the implemented test cases in use. The tests go through the entry criteria check, the actual running of the tests, defects logging and other ways of checking any possible discrepancies, while filling the traceability metrics during the test execution. After the test have been carried out, it is time to move to the *conclusion phase* where the exit criteria and other test outcome are carried out with the help of reporting. The contents and the frequency of the reports change depending on the recipient of the reports. The direct testing background such as the managers are usually more interested in the technical details such as the number of failed or passes test cases and other defects while the stakeholders and other customers like to hear about the risks and how they are handled or mitigated in the project. The final phase of the STLC is called the *closure phase* and is used to conclude the whole testing process with the means of checking if all the planned and designed tests cases are properly executed, and if the test outcome is investigated and properly mitigated if needed. It is important to check whether any sever defects are open and whether everything is ready for the next possible testing process. The phase often includes feedback meetings and documentation with possible improvement ideas. [24.] The big picture of the software testing life cycle is illustrated in figure 9.

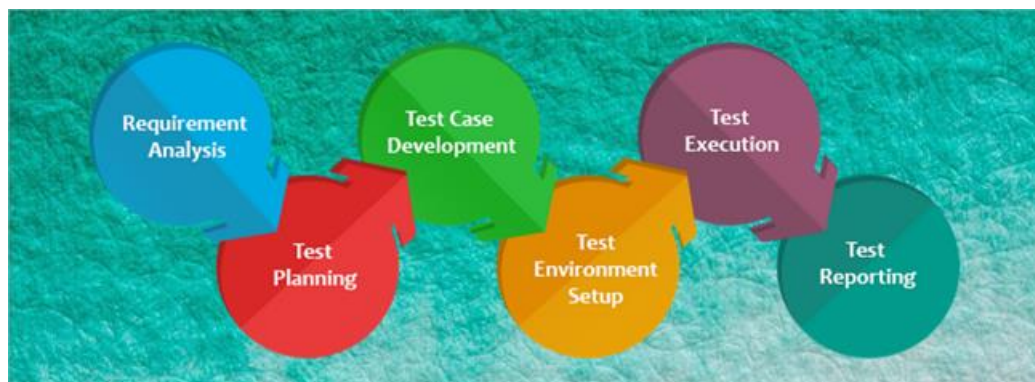


Figure 9. The progress of the software testing lifecycle. [24.]

### 3.3 Test artifacts

The so-called test artifacts are an important part of software testing and are used to provide transparency among the team members, clients, stakeholders and other lead and manager roles by enabling the sharing of requirements, tracking the changes in the software and following the testing activities in the form of different reports and other deliverables. Multiple different test artifacts are prepared and generated during the various phases of the testing process and life cycle. [25.]

*The test strategy* is provided in order to create guiding regulations that further enhance the test design and administer how the testing needs to be executed. Often prepared by the test manager and made available for anyone in the team, the test strategy mainly sheds more light on the areas of the testing tools, techniques, infrastructure, communication and the overall process. Although similar in naming and easily confused with the test strategy, the *test plan* differs quite a bit and supplies a different part of requirements of the software project. It focuses primarily on the approach, objective and the scope of the testing and is mainly used for the formal testing of the software. The test plan systematically covers the areas of identifying the tested features, the tasks to be performed during the testing process, the extent of test independence, the techniques for designing the tests and the usage of the entry and exit criteria. The plan and the focus areas are highly reliant on the implementation of the used processes, standards and the test management tools during the course of testing. [25.]

Another important and very practical deliverable is the *test case*. They are usually regarded as the beginning of the test execution and are primarily designed to verify the correct features, behavior and functionality against the given requirements. A test case contains a name for the test case, an test case specific unique identifier, a description of the test case with a reference to a requirement from the design specifications, the flow of the test execution with preconditions and the needed events, the predicted and the factual results and outputs as well as other reports and details about the test execution. In order to perform and report the results of the tests some *test data* is needed. This is required and stands for the manual or automatic input that is systematically given to the tested software. Poor design of tests cases and the input may affect the ability to run all of the test scenarios and therefore bring down the quality of the testing and the software itself. [24.]

An important part of test artifacts are the table formatted test matrices. The *requirement traceability matrix* illustrates the relation between the requirements given by the development team, or the client and the requirements used in the validation process. The goal of this matrix is to make sure that all system requirements are specified and are included in the test protocol by tracking the requirements against the concrete test cases. Another essential test matrix is the *test coverage matrix* that is used to measure the testing volume and performance of a set of test cases while providing software component data. The coverage matrix assists the testers on creating test cases that fulfil the test coverage requirements and helps them to seek for the requirement areas that are not resolved by the test cases during the testing process. [25.]

Today's testing conventions emphasize the role of automation and efficiency and have started to invest more and more resources in these areas. The *test script* artifacts define the information and instructions to perform in order to validate the performance of the software under test. The scripts are an especially important part of automated testing. [25.]

The reporting phase is as important part of software testing as the designing and running of the tests themselves. To begin, the *test log* is used to record all information regarding the completed tests that can be helpful to the testing team. It includes information such as the test naming, date and time, execution status and the noted errors. Another reporting related artifact is the *defect report*. It is generated during the software testing process and is constructed of all encountered errors, defects, bugs or any other divergencies detected. It is desired to define the found irregularities in a way that they can easily be reproduced, investigated and fixed by the developer team. The report is used to help the developers notify and act on the found defects in order to make the quality of the software better. To summarize the completed testing cycle, a *test closure report* is assembled. The report contains the process details such as the activities performed that are then used to clarify them to the project managers and other stakeholders. The document is prepared when the exit criteria is fully met and is usually created by the testing team lead. [25.]

### 3.4 Testing approaches

The testing approaches are regarded as the way of defining how the testing of a project or a part of it is performed. The commonly used approaches can be divided into static, dynamic and passive groups of which the static approaches are usually implicit, include verification and consist of different inspections, reviews and walkthroughs. The dynamic testing however also includes the validation phase and processes a set of test cases that are executed during software runs and that are divided into appropriate sections as well as administered to discrete modules or functions. In some suitable cases the passive testing is also applied and is carried out without any interaction with the software under test. The passive nature of this category of testing does not include any testing data but incorporates the investigation of system logs and other different traces that may reveal specific patterns or other related behavior to act upon. [33.]

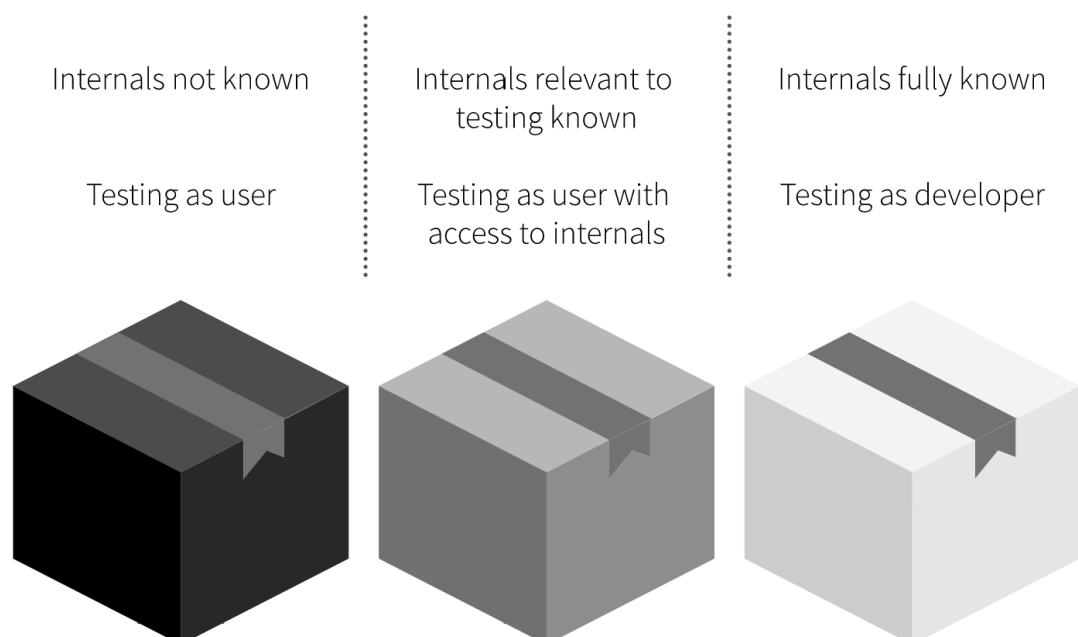


Figure 10. Illustration of the different box model testing approaches. [26.]



### 3.4.1 Black box testing

The black box approach is based on the idea of testing the Application Under Test (AUT) without any knowledge or specific information about the code structure, internal paths, connections or other implementation details. The whole testing process relies completely on requirements and specifications given to the software project and the focus is targeted towards the inputs and outputs of the Application Under Test. This type of testing is sometimes also referred as Behavioral Testing. [27; 28.]

The tested application can be pretty much any kind software system such as a website, a desktop application or a database, since the internals of the application do not need to be known or accessible and the evaluation is done by comparing the input values with the output values. The high-level goal of the black box testing is to test the functionality of the system under test as a whole and is often seen applied to integration-, system-, regression- and acceptance testing levels as functional or non-functional. Black box testing relies on different testing techniques such as boundary value analysis, state transition testing, comparison testing, error guessing, decision table testing, equivalence partitioning and different graph-based testing methods. [27; 28.] The relevant parts for black box testing are displayed in figure 10.

Table 9. The advantages and disadvantages of black box testing. [27.]

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>+ The knowledge of the software implementation or related programming languages are not needed.</li> <li>+ The designing of the test cases can be started when the specifications are complete.</li> <li>+ The developer relation and non-objective perspective can be minimized by using an independent party to run the tests.</li> <li>+ The user focused testing and perspective helps to discover any discrepancies in the software specifications.</li> </ul>	<ul style="list-style-type: none"> <li>- The tests can become redundant if the developers or designers have already run a specific test case.</li> <li>- Conflicts can happen if specific test cases end up requiring knowledge from the black box.</li> <li>- There is a limit for testable inputs meaning that multiple program paths are going to be left untested.</li> <li>- The test cases can be difficult to design due to the unclear nature of the black box testing specifications.</li> </ul>

### 3.4.2 White box testing

White box testing or as sometimes regarded as clear box or glass box testing is a software testing method where the inner details of the software under test, such as different structures, the design and the implementation are known and visible to the tester. It is usually required for the testers to be familiar and understand the functionality and implementation of the software as well as to be programmatically adept. The testing is mainly done by applying inputs that follow the flow and the implementation of the application and then determining applicable outputs for them. White box testing goes beyond the provided user interface and much further into the inner details of an open system, hence being called white box testing displayed in figure 10. [29; 30.]

White box testing is commonly applied to System level-, Integration- and unit testing levels of which the unit testing usually plays the biggest role. It is associated with testing techniques such as path-, branch- and statement coverage and extends into areas like data flow testing, path testing, loop testing and code-, segment-, compound condition and branch coverages. [29; 30.] The applicable parts for white box testing are included in figure 10.

Table 10. The advantages and disadvantages of white box testing. [29.]

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>+ It is possible to cover most of the testing paths due to the thorough testing and the transparent structure.</li> <li>+ The readiness and the implementation of the software does not hinder the testing and it can be started sooner.</li> </ul>	<ul style="list-style-type: none"> <li>- The fact that the testing is heavily dependent and bound to the application under test can lead to some implementations or platforms being unavailable.</li> <li>- The testing requires good knowledge of the software implementation and programming in general.</li> <li>- Any modifications to the software can also affect the test scripts or flow and may require maintenance.</li> </ul>

### 3.4.3 Grey box testing

Grey box testing is the middle ground of the black- and white box models. In this model the testers have limited knowledge and information about the inner functionality of the software system while still having access to requirements, specifications and other comprehensive design documents. Instead of direct access to the codebase and the implementation, the testers utilize different documents such as UML- and architecture diagrams or other state-based models. The most used testing techniques for grey box approach include regression-, matrix-, pattern- and orthogonal array testing. [31.] The appropriate components for grey box testing can be found in figure 10.

Table 11. The advantages and disadvantages of grey box testing. [31.]

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>+ A good middle ground in designing complex test case scenarios.</li> <li>+ Combines advantages of the black- and white box models.</li> <li>+ Based on multiple functional specifications and architectural documents.</li> <li>+ Acts as and helps to define the boundary in between the developers and the testers.</li> </ul>	<ul style="list-style-type: none"> <li>- The association of defects may prove difficult when testing specific types of software configurations, e.g. distributed systems.</li> <li>- Some parts of the more specific internal implementation provided in white box testing or the simplicity and user focused basis of black box testing are never fully reached.</li> </ul>

### 3.4.4 Exploratory testing

Exploratory testing is an alternative for the popular box model approaches and is commonly used in Agile models and projects where there is a need for an early iteration, there is a demanding application or if the testing resources include experienced or completely new testers. Instead of designing and creating the test cases in advance, the testers check the system and go through it with minimal previous details or notes. The approach relies on the concurrent processes of test design and execution and emphasizes investigation, analysis, learning and the responsibility and freedom of the testers. The exploratory testing process often consists of five phases that are also called the session-based test management cycle. [32.]

1. Classify and categorize the common types of faults and bugs from previous or similar projects and evaluate their root causes while finding possible risks and noting down related test ideas.
2. Create a small-scale test charter that proposes what to test, how to test and what needs to be noted down or investigated. The document can also include concrete test or application usage ideas and details.
3. Execute the time boxed testing individually or in a pair. Try to achieve a limited time session without interruptions, react to the functionality and output of the system and note down all findings.
4. Review and evaluate the defects while analyzing the test coverage areas. Try to learn from the testing process and consider what could be tested the next time.
5. Perform debriefing by compiling the test results and comparing them with the charter written in the beginning of the process. Find out the need for any further testing or supplementary tests.

Table 12. The advantages and disadvantages of grey exploratory testing. [32.]

Advantages	Disadvantages
<ul style="list-style-type: none"> <li>+ Rigorous, structured and easily taught and managed.</li> <li>+ Intellectual approach helps to improve the productivity and discover defects that might be ignored by other testing methods.</li> <li>+ Testing does not demand any requirement- or specification documents and can be performed in a short time.</li> <li>+ Generates new ideas and testing</li> </ul>	<ul style="list-style-type: none"> <li>- Highly dependent on the tester and their imagination and cognitive skills.</li> <li>- Not suitable for longer or more official testing processes or projects.</li> <li>- Defined by the tester's knowledge of the tested application and domain.</li> <li>- Some defects are likely to be missed since the testing is rarely thorough.</li> </ul>

### 3.5 Software testing levels, -types and techniques

#### 3.5.1 Testing levels

The practical software testing process is commonly divided into three main levels of unit testing, integration testing and system testing with the typical addition of acceptance testing level. These levels are used to group the smaller parts of the complete testing process into larger sections based on the specificity of the tests or their time of addition in the software development lifecycle. The chosen SDLC and its characterized phases like setting the prerequisites, project design, analysis, development, testing, and deployment are all tied together and undergo the process of the software testing levels. [34; 35.]

*Unit testing* focuses on verifying the functionality of smaller components or modules of the software. Each unit is isolated and used to perform tests to make sure that it fulfills the given requirements and its initial purpose. The tests are often the first or an early part of the complete software testing cycle and the development process. Unit tests are usually written by the developers during or after working on the software component in question. They are typically performed on the class level and commonly utilize at least constructors and destructors. Other common unit testing methods include code reviews, code coverage analysis as well as other techniques such as static code-, metrics- and data-flow analysis. The strength of unit testing lies in detecting and combatting the possible errors in the early development phases or during the development, and without having to return to the foundations of the code in the later phases or close to the software completion. [34; 35.]

The second testing level is called *integration testing*. It intends to test and combine distinct parts of the software and their interfaces to confirm the functionality of the unit groups or modules and their communication and interaction together. The testing can be commonly split into top-down and bottom-up integration methods from which the testers choose the desired option. The recommended bottom-up integration tests so called modules consisting of high-level sequence of smaller units in more intricate scenarios by utilizing the results of the completed unit tests. The other way is to follow the top-down method that is used to test and study the modules in complexity order, starting from the more complex modules and moving to the simpler ones. Integration tests are usually

larger and more complex than unit tests and therefore also produce larger traces that make the fault localization and detection harder. [34; 35.]

After testing the different modules of the software, it is time to move into *system testing*. Like the name suggests the goal is to test all the software components, modules, and interfaces as a whole system to find whether the complete software fulfills the specified requirements. The aim is to perform the testing in an environment that reassembles the end user or customer environment as well as possible. The testing team makes sure that the complete system runs fluently in the desired operating system and confirms that it meets the initial business requirements. System testing usually includes performance-, reliability-, security- and load testing. [34; 35.]

The final level of the four common testing levels is called *acceptance testing*. Now that the software should already be in a well-tested and verified state, it is time to determine whether it satisfies the end-user requirements and specifications and is ready for deployment. The tests are often conducted on a user system and done by following pre-written test cases and scenarios that range from simple cosmetic or other minor mistakes to bugs that may lead into major problems or malfunctions. There are often numerous contractual and legal reasons that require that the acceptance testing is carried out depending on the scale and the usage of the software product. [34; 35.]

### 3.5.2 Testing types and techniques

Multiple different concrete software testing techniques can be applied on different testing levels and depending on the type and needs of the project. These types and techniques are commonly split into two main categories of functional and non-functional testing. [36.]

Table 11. Illustrates a comparison between the two categories

*Functional testing* refers to the kind of testing techniques that make sure that the software functionality performs according to the requirement specifications. The goal is to test the user interfaces, databases, APIs, software security, client – server applications and the application as a whole and with the appropriate input and then compare the received results with the predicted results. The testing can be manual or automated and is mainly performed as black box testing and therefore the source code does not need to be visible to the testers. Common functional types techniques include smoke testing, integration

testing, unit testing, regression testing, sanity testing, acceptance testing as well as globalization, localization, and interoperability techniques. [36.]

On contrary to its opposite, *non-functional testing* focuses on the less practical sides of the software with the idea of explicitly making sure that the application is complete by the standards that the functional testing does not cover. Non-functional testing plays a big role in terms of customer satisfaction and therefore concentrates heavily on usability, reliability, and the overall performance of the software, with the baseline of efficient and smooth running under any given condition. As non-functional testing can often be found quite cumbersome and deals with aspects like accuracy, correctness, durability, stability and especially time, it is often heavily automated instead of performed manually. Typical non-functional testing techniques consist of usability testing, performance testing, load testing, compliance testing, stress testing, portability testing, volume testing, recovery testing, reliability testing and scalability testing. [36.]

Table 13. A comparison between functional and non-functional testing categories. [36.]

Criterion	Functional testing	Non-functional testing
Target area	following the customer requirements	Fulfilling the customer expectations
Requirements	Functional requirements are clear and easy to specify and are achieved using the functional specification	Non-functional requirements are difficult to specify and are achieved using performance specifications
Purpose	Validation of the correct software behavior	Validation of the performance, usability and reliability
Implementation	Executed before non-functional testing	Executed after the functional testing
Functionality	Illustrates what the software product does	Illustrates how the software product works
Testing form	Straightforward and commonly performed manually	Complicated to execute manually and typically automated.
Example	Validation of the login process	The estimated time it takes for a dashboard to load

### 3.6 Automated testing

In order to make the process of software testing as fast, easy, reliable and efficient, the interest in the usage and development of automated testing has gained a lot of attention in the recent years. Instead of manually executing the tests by using human testing resources, the tests are performed automatically with the help of different test automation software tools. The common tasks of the test automation consist of executing different test case suites, inserting test data into the system under test, analyzing and evaluating the test results and comparing them with the expected results as well as producing comprehensive test reports. Setting up the automation environment can often be difficult and time consuming and usually requires noticeable investments of both money and other resources. The essential goal of is to be able to automatically execute the test suites repeatedly and without human intervention while providing and recording the tests results. Test automation has become an important part of the software development- and testing cycles and acts as a significant addition alongside manual testing. [37.]

There are multiple types of automation testing that include the automation unit tests, API testing, UI based testing, smoke testing, integration testing, regression testing, security testing, acceptance testing, data driven testing and many other use cases. [37.]

## 4 Project tools, requirements, and execution

### 4.1 Requirements and the goal

The initial goal of the project was to enforce and create a new DevOps platform based continuous integration and internal release solutions with a heavy focus on test automation. The project already had a platform for the system as the old source code and work history had been transferred over from an older system.

The plan was to start with a stable automated system for inhouse software releases by utilizing custom build- and release pipelines with the addition of needed software agents, servers, and other tools. Another important addition to the system was going to be the utilization of test automation as well as the use of other verification methods and reporting tools. The idea was to start the test automation by verifying the functionality of the



software installation packages and begin the creation of the automated testing platform and the custom libraries that would allow further development and expansion for broader testing purposes. Most of the testing would be performed in a Windows environment and on an already existing instrument simulator made for the analyzer project.

The analyzer project itself is developed under strict IVD regulations and follows a very particular verification and validation process to make sure the product is safe and reliable to be used with real patient samples and data. This means that the actions performed by the test automation will mostly perform integration pre-testing that will help to detect and track down problems in an early phase of the development of a feature as well as help to improve the quality of the software by running regression and performance tests. The testing platform and the tests suites could be verified and validated separately at a later time in order to incorporate the automated testing as part of the official testing plan and requirements.

#### 4.2 Servers, services, and other tools

The complete environment for the continuous integration, automated testing and releasing of the analyzer software consists of the Azure DevOps Services infrastructure and related Microsoft-hosted agents running in cloud as well as two on premise servers which one of them is virtual and the other physical.

The hosting and configuration of the *cloud services* are mainly done by Microsoft as the service provider. The only needed manual configuration is related to the Microsoft-hosted agents and is mainly limited to deciding the type and amount of the agents as well as adding possible user-defined capabilities and defining how they work in parallel. Other small related things that can be changed are the retention times for the builds and releases as well as the build formats etc.

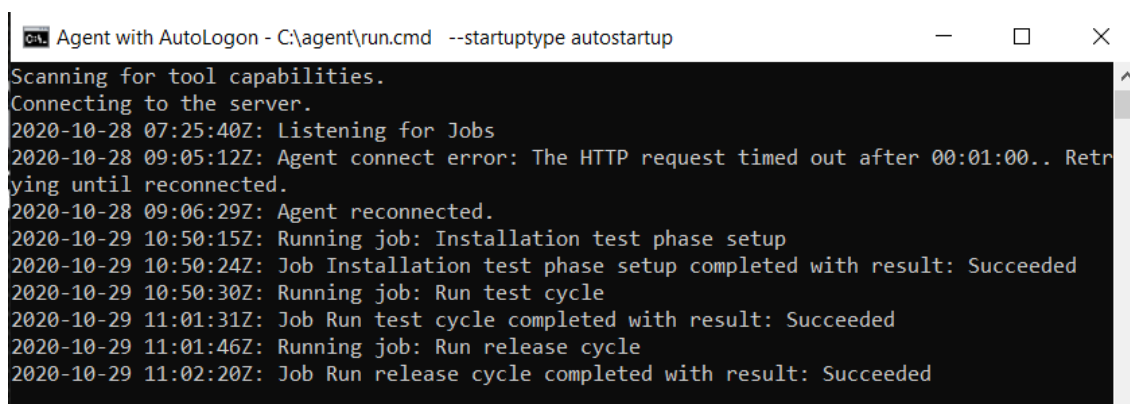
The creation of the installation package is done on Windows Server 2008 operation system based virtual *build server* located on-site and by using InstallShield. InstallShield has become one of the industry standards for creating software packages and installers for mainly Windows based server and desktop systems. The InstallShield project is highly configurable and customizable and includes multiple features regarding the easy installation of the developed software on the target systems. These features include for

example. Database services, prerequisite handling, version tracking, software patching and upgrading support etc. The IS project on the build server consists of custom configurations and set procedures how the installation packages are constructed. The creation of the installation package is started by running a specific build batch script and by gathering the software components from the build artifact that was built and created in the Azure DevOps Services cloud and then transferred to the build server. The InstallShield project and the build script also create multiple log files regarding the installation package creation. After the software version specific build script is ran successfully and the creation of the installation package is complete, it is transferred to the test server through the company network file share.

The *test server* is a physical on-premise desktop pc that is running a windows 10 operating system. The purpose of the server is to run automated integration- and smoke tests on the software installation package and the analyzer simulator software as well as release the tested package to its respectable destination as a part of the DevOps pipelines. When the release pipeline reaches the testing phase and the installation package is transferred from the built server, the self-hosted software agent begins running the tests on the server. After the functionality and integrity of the installation package have been tested, the server agent starts the release process and transfers the package to the release destination based on the given conditions. If the testing is unsuccessful or there are any failed tests, the pipeline process is stopped, and the results are reported to multiple destinations. In addition to running the tests during continuous integration pipelines, the server is also used for performing additional automated nightly integration tests.

The *test server* environment is kept as clean and simple as possible to save space, avoid clutter and to keep the performance as good as possible. There are however a few utility requirements on top of installed software agent, the test environment, and the tests packages. Most of the performed tests will be using the graphical user interfaces of either Windows or the instrument simulator and will therefore need a preferably uninterrupted interactive session. This is achieved with the correct service user account rights to overwrite the normal company user policies such as setting the server to a no sleep state and configuring the auto logon feature to make sure the server is ready when needed. Other requirements include disabling the user access control in order to disable security pop ups that cannot be handled by the automation, disabling Windows auto updates, enabling the Microsoft Message Queue and installing the analyzer software requirements such as .NET service package, SQL Server for the databases and the USB CAN

driver for the simulator. The software agent listening and executing jobs are displayed in figure 11.



```

Agent with AutoLogon - C:\agent\run.cmd --startuptype autostartup
Scanning for tool capabilities.
Connecting to the server.
2020-10-28 07:25:40Z: Listening for Jobs
2020-10-28 09:05:12Z: Agent connect error: The HTTP request timed out after 00:01:00.. Retrying until reconnected.
2020-10-28 09:06:29Z: Agent reconnected.
2020-10-29 10:50:15Z: Running job: Installation test phase setup
2020-10-29 10:50:24Z: Job Installation test phase setup completed with result: Succeeded
2020-10-29 10:50:30Z: Running job: Run test cycle
2020-10-29 11:01:31Z: Job Run test cycle completed with result: Succeeded
2020-10-29 11:01:46Z: Running job: Run release cycle
2020-10-29 11:02:20Z: Job Run release cycle completed with result: Succeeded

```

Figure 11. The self-hosted software agent listening and performing jobs on the test server.

### 4.3 Azure DevOps

The Azure DevOps Services by Microsoft was chosen to be the platform for the project as it was the direct successor to the previous solution called Visual Studio Team Services which was also formerly used as the main development service of the analyzer project in question. The Azure DevOps platform was introduced by Microsoft in 2018 to offer a uniform and comprehensive solution to serve the software industry in all scales and forms. The main functionalities of Azure DevOps include extensive work management and Agile tools, version control and repository system, continuous integration and continuous deployment pipeline management as well as different testing, monitoring and marketplace extendibility. The DevOps service is divided into cloud- and local solutions that are also designed to work together. The analyzer project uses the cloud solution for its versatility and easy accessibility. [38.]

#### 4.3.1 Version control

Azure DevOps offers a versatile version control solution in the form of Azure Repos that consists of a set of version control and management tools suitable for businesses and software projects of any size. The solution supports both Git distributed version control system and the centralized Team Foundation Version Control by Microsoft. The Git option is often chosen due to its reliability, ease of use and provided offline working functionality. Azure Repos provides standard Git as the default option but offers great

extendibility towards multiple Git clients and tools of choice. This also means a great connectivity and integration with multiple different development environments. All branching and pull request functionality together with Git API and semantic code search concepts are built in and conveniently available through the Azure DevOps services. [39.]

All the source code related to the Instrument control software of the analyzer project was previously located in VSTS source control repositories and was transferred over to the Azure Repos upon the release of the Azure DevOps services. Most of the development is done in .NET environment using Visual studio as the development environment with direct connection to the Azure Repos.

#### 4.3.2 Software agents

Azure DevOps provides two kinds of software agents that run and orchestrate pipeline or other automated jobs on either on self-hosted machines and servers or Microsoft-hosted virtual machines. The Microsoft-hosted agents are used to run tasks directly on Microsoft virtual machines or containers and therefore the updates and other maintenance of the agents are done automatically and do not require additional attention from the user. When a pipeline is run and a set to run on Microsoft-hosted agents, a free and suitable account is selected from the agent pool and then discarded after performing the given job. The self-hosted agents however are more customizable and require more work to set up, configure and maintain. They can be installed on multiple operating systems and can also be ran on containerized environments. The capabilities and demands as well as the amount of both agent types can defined by the user. [40.]

The DevOps solution for the analyzer project utilizes a pool of multiple Microsoft-hosted agents for building the software in the cloud and two self-hosted agents on the build and the release server to handle the creation of the installation package and the automated testing of the analyzer software. The installation and configuration of a self-hosted account on the test server running Windows 10 as the operating system follows the next steps. The configuration screen of a self-hosted Azure agent is displayed in figure 12.

1. Add the agent to the desired agent pool in the Azure DevOps project settings while making sure that the user has the correct rights to configure and manage the agent and is able to generate a personalized access token to ensure the connection between the agent machine and the Azure services.'
2. Download the agent package making sure that the correct target operating system and architecture are selected (64-bit Windows in this case).
3. Unpack and install the agent using the *config.cmd* file provided in the installation package. Provide the needed information such as the organization Azure DevOps Services URL and the corresponding PAT of the Microsoft account that was used to create the agent.
4. Configure the remaining options and set or install the needed capabilities based on the usage of the agent. Select whether the agent will be as a service or as in this case interactively, since the test server and the agent need to perform Graphical User Interface related tasks such as run some GUI tests. Install AzureRM modules to enable the Azure resource manager capabilities. Rest of the needed capabilities can be set and installed in the agent setting found in the Azure DevOps agent settings and by restarting the agent. The agent can then run by using the *run.cmd* file and should start automatically upon the machine or server restart.

```

Administrator: C:\WINDOWS\System32\cmd.exe

AZURE PIPELINES
agent v2.175.2 (commit 5c4925c)

>> Connect:
Enter server URL > https://dev.azure.com/cdssw
Enter authentication type (press enter for PAT) >
Enter personal access token > *****
Connecting to server ...

>> Register Agent:
Enter agent pool (press enter for default) >
Enter agent name (press enter for FIHEL-5YXH9X1) > Agent_1
Scanning for tool capabilities.
Connecting to the server.
Successfully added the agent
Testing agent connection.
Enter work folder (press enter for _work) >
2020-10-23 18:18:52Z: Settings Saved.
Enter run agent as service? (Y/N) (press enter for N) > n
Enter configure autologon and run agent on startup? (Y/N) (press enter for N) >

```

Figure 12. Azure pipeline self-hosted windows agent configuration screen.

#### 4.3.3 Continuous integration and deployment pipelines

Azure DevOps Services offers a multi-purpose cloud-based pipeline functionality for different continuous integration and continuous delivery needs. The pipelines support multiple different programming languages and offer options for building, testing, deploying, releasing, and delivering the software project. They are also very well integrated with other Azure services and support multiple deployment targets such as container registries, cloud solutions and physical- or virtual targets as well as different package formats like npm, Maven, NuGet or any other desired package management repository. [41.]

The pipelines mostly consist of different stages that include multiple tasks or jobs that are run by either the Microsoft-hosted or self-hosted agents. The stages are triggered by different triggers such as completed pull requests, finished builds, scheduled runs, continuous deployment or release triggers or any other custom pipeline trigger. Depending on the type of the pipeline, the sources for the pipeline objects can either be the source

code of the project that is going to be built, packaged etc. or an software artifact or another type of package that the pipeline actions will be performed on. The source code can be fetched directly from Azure Repos or any applicable version control system such as GitHub, and the artifacts from the built-in directory or any other provided source. The tasks and jobs inside the stages consist of different operations related or performed directly on the pipeline objects. The tasks can be either built-in or custom, and the functionality can include activities such as executing scripts or a piece of code, performing building, packaging or publishing actions, running unit tests, handling files or completing some monitoring or reporting operations. More ready-made jobs, tasks and integrations can be found from the provided Azure marketplace. Each pipeline also has the possibility to store and alter variables that can also be retained over the separate pipeline runs and stored as parts of variable groups. The custom variables can be used in combination with built-in variables to store information such as folder paths or version numbers. [41.]

The Continuous integration, testing and deployment of the analyzer project are divided into two types of pipelines. First the build pipeline that handles the collection of the source files from the Azure Repos, restoring the related NuGet packages, building the software, running the unit tests, copying all the required software components and publishing the build artifact. The build pipeline is triggered when a pull request is created or when it is completed and the code is pushed into the repository. Figure 13. displays the structure and the order of the tasks in the analyzer project build pipeline.

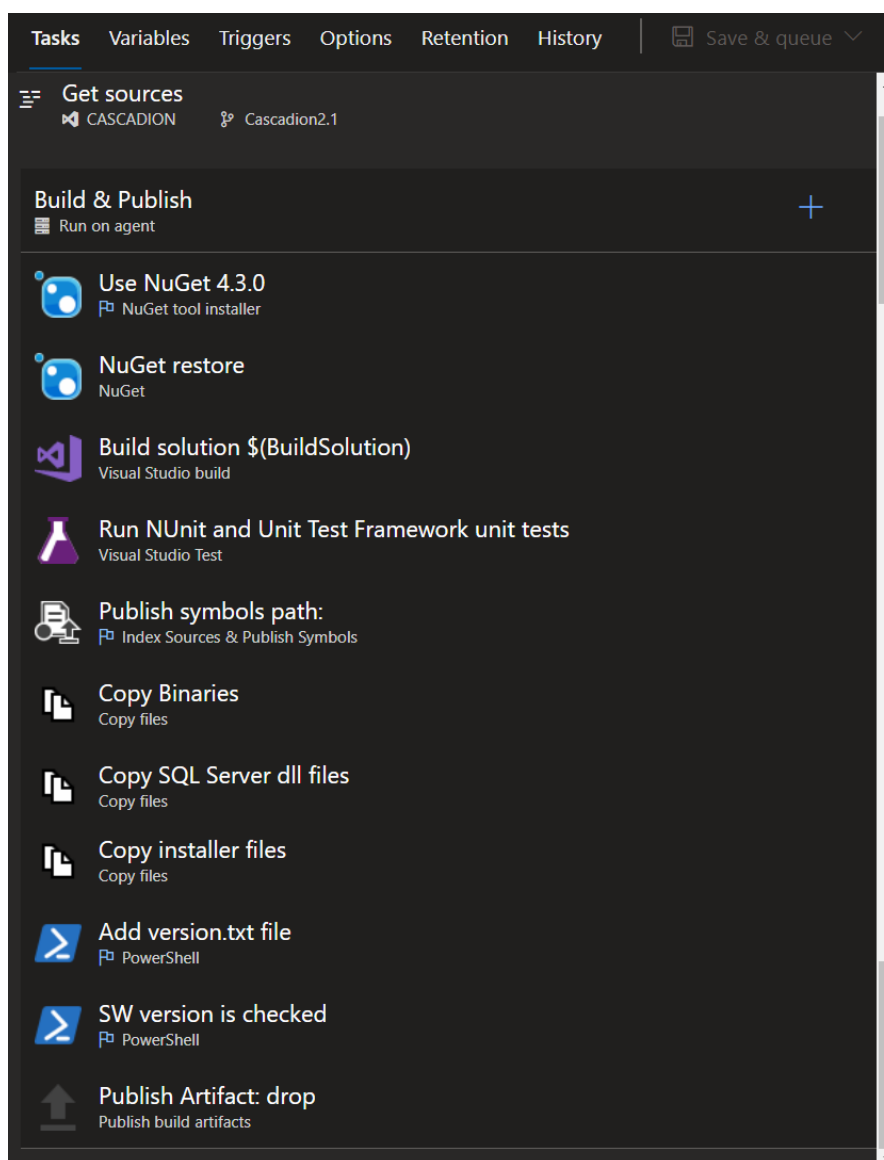


Figure 13. The structure and the order of the tasks in the build pipeline. (Some of the file copying tasks have been excluded from the image for more clarity).

The second pipeline is called the release pipeline and is triggered after a pull request in a respective branch is completed or an individual software installation package is needed. The release pipeline is split into three stages and is responsible for retrieving the build artifact, transferring it over to the build server, handling some prerequisites and triggering the InstallShield build. After a successful building of the software install package, the pipeline operates the needed files and starts the automated testing process on the test server as well as uploads and publishes the test results. If the automated tests are successful, the pipeline checks for different release conditions and publishes the tested installation package in the respectable location. The complete process consisting of three phases of the release pipeline are displayed in figures 14, 15 and 16.



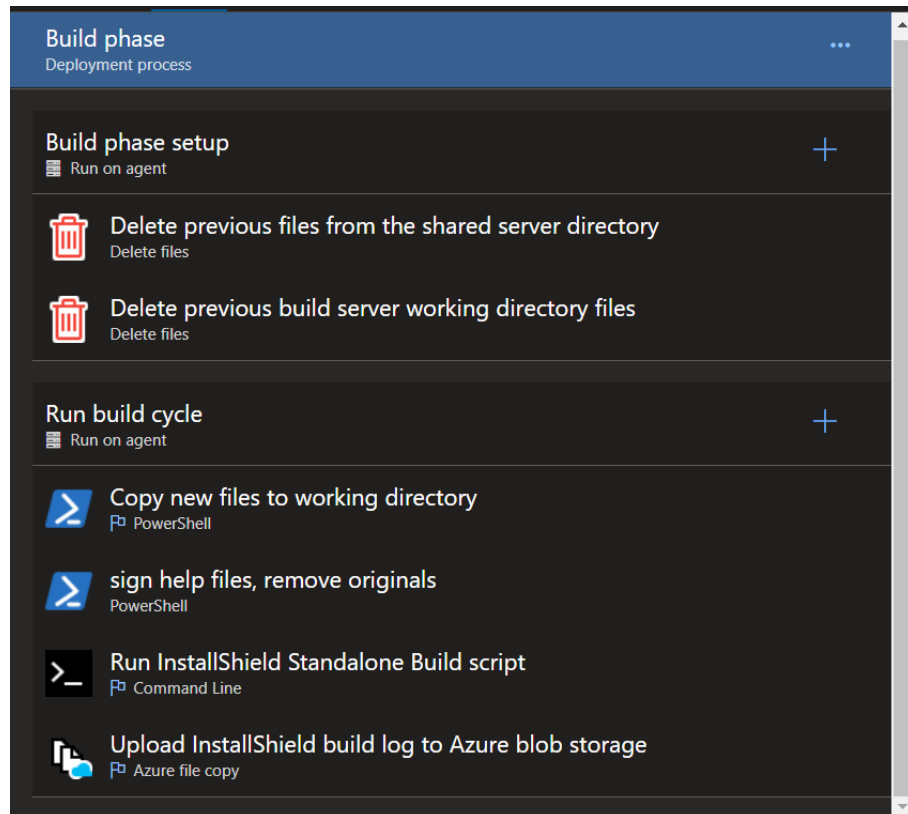


Figure 14. The structure and the order of the tasks in the first phase (build phase) of the release pipeline.

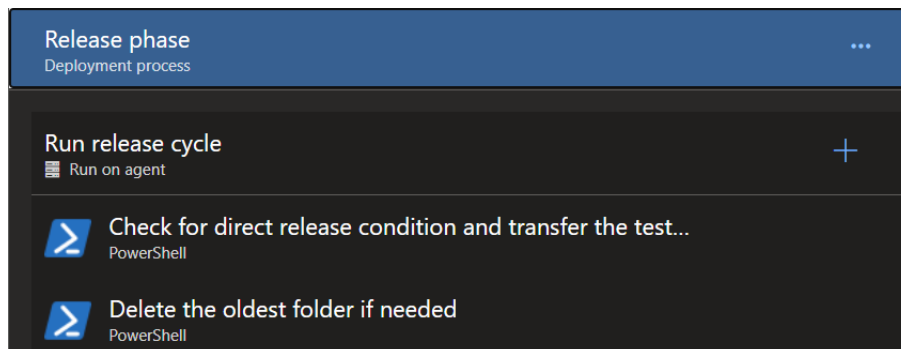


Figure 15. The structure and the order of the tasks in the third and final phase (release phase) of the release pipeline.

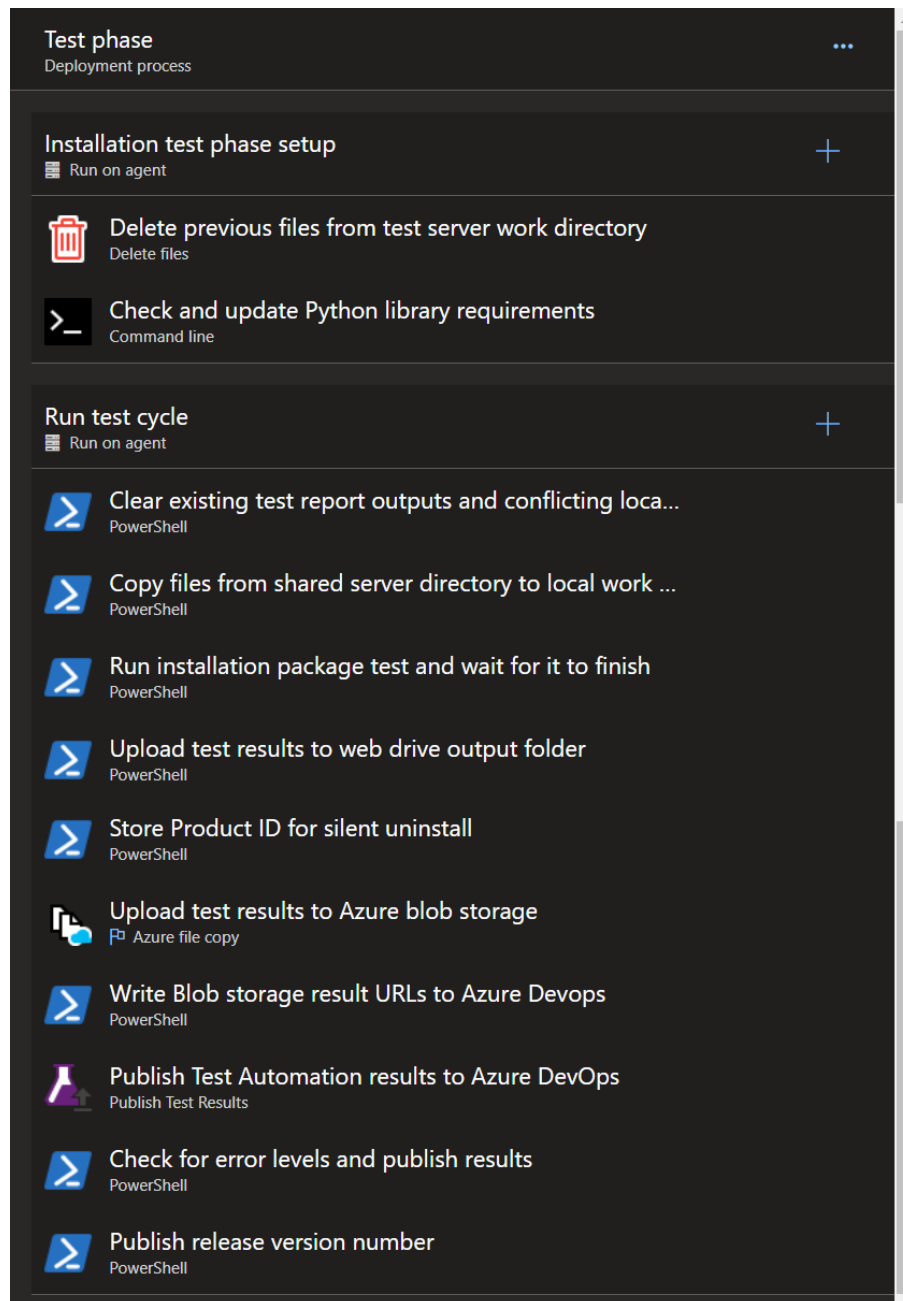


Figure 16. The structure and the order of the tasks in the seconds phase (test phase) of the release pipeline.

Both pipelines use multiple variables to define file paths and transfer information such as unique software version numbers between the servers and pipeline runs. Each version of the analyzer software has its own corresponding build and release pipeline. The pipelines can also be run manually at any time and in the case of the release pipeline, the test phase can be skipped in order to obtain an untested build without waiting for the automated tests to complete. The structure and a performed run of an example release pipeline is displayed in figure 17.

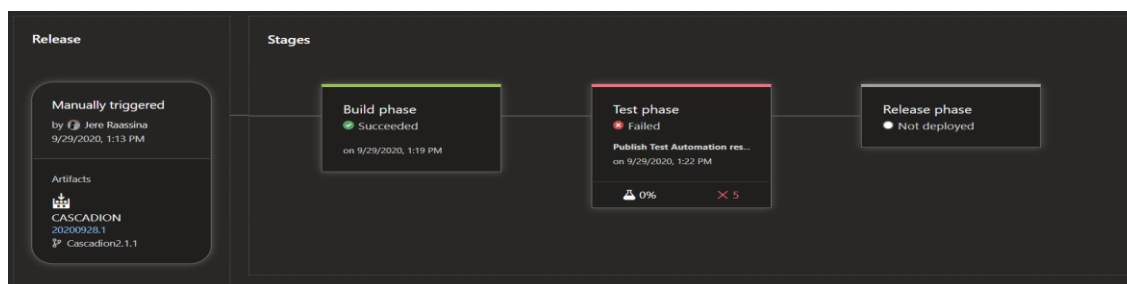


Figure 17. The structure and the order of the phases of the release pipeline, starting from the software artifact and continuing to the reorderable stages.

#### 4.3.4 Azure portal, reporting and other features

The Microsoft Azure environment offers multiple other tools, services and integration options through the Azure Portal. Some useful options include virtual machines, database- and storage services, monitoring and container services. Most of the utilities provide either a built-in or third-party support for pipeline tasks or the Azure DevOps service directly.

The analyzer project uses the cloud based Azure Blob object storage for publishing and storing the automated testing results for the pipeline and nightly tests as well as the software build logs and reports. The documents are stored into the project blob storage container and separated into virtual folders and further into blobs that are easily accessible through pipeline runs and shared inside the project team for smooth troubleshooting or monitoring. The Azure Blob Storage is designed and optimized for the storage of large number of unstructured data items that do not conform to any distinct definition or data model. Therefore, the storage system is optimal for performing actions on multiple types of data items such as storing items for distributed use, presenting documents for direct browser use, writing to log files, storing data for analysis or backup and restore situations or even streaming media files. The blob data objects can be accessed through

HTTP/HTTPS connection by either clients or applications via Azure CLI, Azure REST API, Azure PowerShell or Azure Storage client library and they support multiple different programming languages. [42.] An example of the contents of a virtual Azure Blob storage folder can found in figure 18.

Name	Modified	Access tier	Blob type	Size	Lease state
[.]					...
log-20201012-160446.html	10/12/2020, 4:10:08 PM	Cool (Inferred)	Block blob	229.54 KiB	Available
output-20201012-160446.xml	10/12/2020, 4:10:09 PM	Cool (Inferred)	Block blob	41.4 KiB	Available
report-20201012-160446.html	10/12/2020, 4:10:09 PM	Cool (Inferred)	Block blob	227.49 KiB	Available
test_Output.txt	10/12/2020, 4:10:10 PM	Cool (Inferred)	Block blob	5.96 KiB	Available
xunitOutput-20201012-160446.xml	10/12/2020, 4:10:11 PM	Cool (Inferred)	Block blob	1.03 KiB	Available

Figure 18. The analyzer project test automation result files inside an Azure Blob storage virtual folder.

Another way to have the test results conveniently available right after the tests have been performed is to use the Azure test services and publish the results directly to the cloud service. The job takes the generated universal XUnit file containing the test results from the test server and uses the information to store and display some of the test data directly on the pipeline runs. This step and file information are also used to define whether the tests have passed, and the pipeline can continue to the next phase or if there were any errors that need to be reported. Figure 19. Displays an example of a XUnit based test summary displayed in Azure DevOps.

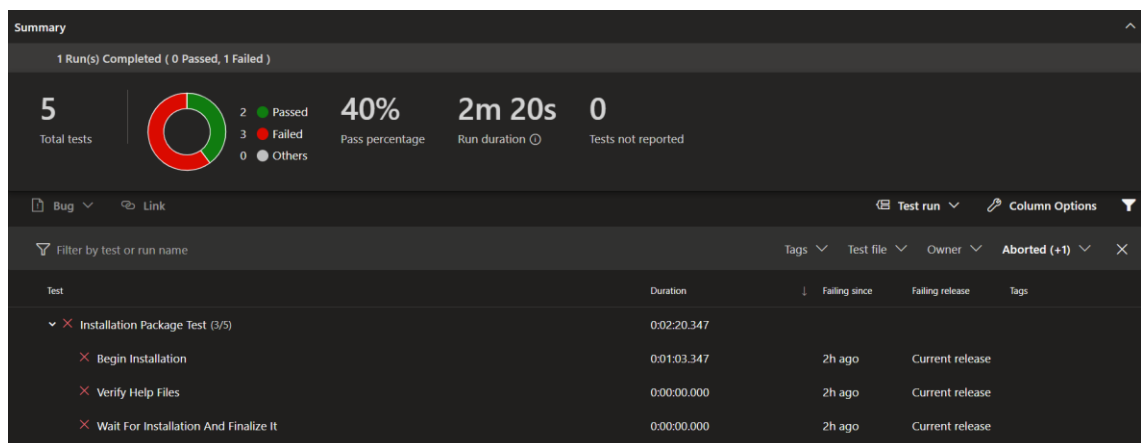


Figure 19. An example of the generated test summary based on the XUnit test report file information.

#### 4.4 Automated testing and Robot Framework

One of the biggest improvements alongside the DevOps tools and pipelines is the beginning and addition of the test automation functionality. It was important to consider and choose suitable and optimal tools for the test automation of the analyzer project, especially when starting from scratch and designing the baseline of the environment. The main attributes to be fulfilled were considered to be: easy to start with, easily expandable, simple enough to be operated with minimal knowledge, technically customizable and suitable for the project, compatible with the Azure DevOps environment, able to generate reports for further investigation, preferably open sourced with minimal costs and the possibility to verify and validate the environment later in its life cycle. Based on further evaluation and previous experience, the main tool for the project was chosen to be Robot Framework.

Robot Framework is an open source automation framework designed mainly for test automation and robotic process automation. The supportive funding and leading of the development are mainly done by the Robot Framework Foundation which is a consortium formed by multiple different supporting companies. The framework has gained a lot of attention in the past few years and is constantly growing together with its user base which ranges from individual hobbyists to large corporations. The source code with its official documentation are globally hosted on one central repository and being open source means its usage is completely license and cost free. Robot Framework is originally written in Python but is application and operation system independent and can be developed and extended in Java and .NET environments. [43.]

The main strengths of the tool include easy and open extensibility and usage, possibility and flexibility to be integrated with practically any other system or tool, easy and understandable human-readable keyword syntax, separately developed built-in and custom libraries and tools written in multiple programming languages, good documentation and knowledgeable open source user base, built-in and easy to read logs and reports, as well as its nearly endless use cases. These assets were also found to fulfill and support the needs of the testing environment for the analyzer project very well. [43.] The high-level architecture of Robot framework is displayed in figure 20.

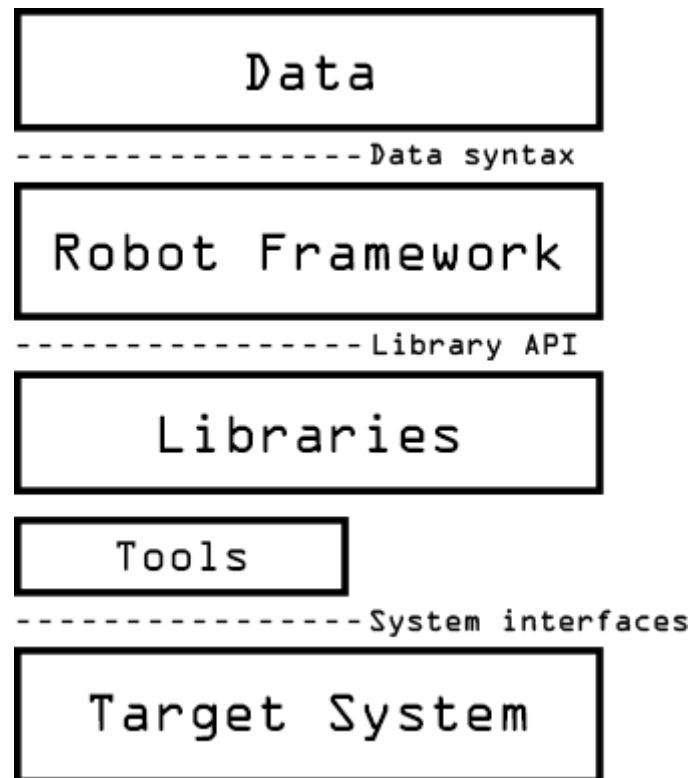


Figure 20. High-level architecture of Robot Framework. [44.]

#### 4.4.1 Environment and prerequisites

There are minimal requirements for the usage of Robot Framework, with the most obvious one being the installation of either Python 2 or Python 3, with the optional addition of the needed interpreters for Java and .NET compatibility. For the easy installation of the additional Robot Framework libraries as well as other Python tools, it is also recommended to install the Python Package Installer (pip). The easiest way to install Robot Framework itself is to use pip, however it can also be installed using a manual download, a standalone Java archive or directly from the source code. Python needs to be added to the Windows PATH variable in order to use the installers and operate Robot Framework itself. [44.] Figure 21. Introduces the needed command line commands for installing pip for Python and Robot Framework itself.

```
python -m pip install robotframework // Installing pip for python
pip install robotframework // Installing Robot Framework using pip
$ robot --version // Verifying the installation of RF
```

Figure 21. The needed commands for installing pip and Robot Framework. [44.]

The test execution can be done either by running the test files directly from the provided robot script command line functionality or by including multiple test files into a directory that is run as a complete package. The test execution commands are able to consume multiple arguments and tags that define some of the main test properties such as output file paths, output types, timestamps and fail conditions. Starting the test runs opens a separate window for following the test flow which can also be piped directly into a text file if needed. The correct setup of the PATH variable is required to execute the tests using the command line. [43.] The different ways of executing Robot Framework tests from the command line are displayed in figure 22.

```
robot example_tests.robot // Executing a single robot test file
python -m robot example_tests.robot // Executing the robot module di-
rectly with Python
python path/to/robot/ example_tests.robot // Executing the robot di-
rectory
robot --outputdir path/to/output --xunit xunitOutput --timestampout-
puts --exitonfailure example_tests.robot > path/to/outputtext-
file/test_output.txt // Executing tests with optional arguments
```

Figure 22. Different ways to execute the test suites or directories. [44.]

The automated tests for the analyzer project are performed in a Windows environment and most of them utilize the analyzer software simulator that provides the functionality and the graphical user interface of the analyzer without a physical instrument. The tests are executed by running the corresponding robot directories and performed by the software agent on the test server. The test structure and code are located in their own repository in Azure DevOps Repositories and have their own integration pipeline for updating the test folders.

#### 4.4.2 Test structure

Robot Framework follows a hierarchical file structure when forming the test framework and data. The high-level test structure is built so that the test cases are constructed in test case files that automatically form the tests suites that correspond to the test file structure. Test directories are formed by combining multiple test case files or other nested directories and can also contain a special initialization file that is used to configure the test suites inside the directories. Robot Framework also introduces multiple additional files to support the functionality of the test suites. These files include test libraries that contain a collection of lower level keywords for building test cases, resource files with higher level keywords and custom variables, as well as designated variable files that introduce additional ways to create and manage variables. The files mostly follow the Robot Framework syntax with the exception of the test libraries and variable files that are usually created using traditional programming or scripting languages such as python. [44.]

The Robot Framework test file structure is divided into multiple different data sections. The file starts with the *settings* section that is used to define which test libraries or resource- and variable files are imported as well as mark down any metadata used in the test suites. The metadata can contain comments or documentation, setting up the suite setup or teardown along with applying the possible test template. The *variables* section is a local alternative for the more inclusive variable files and is used to define the variables used in the scope of the test file. The *test cases* segment is used to construct the individual test cases that will be run as a part of the test file. The test cases are created using the built-in or user defined keywords set in either the imported test libraries or the test file itself. The most extensive data section includes the Robot Framework *keywords* that contain the concrete test operations that manipulate and utilize the test data and are used to form the test cases. The individual tests cases can also be nested and used in other test cases as well as tagged in order to be run separately. The data sections are identified and distinguished from each other by using the triple asterisk header format shown in the figure 23. [44.]



The test structure supports a few different file formats to bring some diversity into designing the test environment and to making sure that the users can choose their preferred format. The most common approach is to use the *space separated format* that follows the concept of separating the data components such as the keywords and their arguments with two or more spaces. Another option is to use the *pipe separated format* where the pipe character acts as the separator. The third option is the *reStructuredText format* that is mostly used when the common Robot Framework data is embedded into code blocks and when mixing different types of documents with the test data. Robot Framework ignores certain data, sections and objects such as empty rows and all data that does not meet the structure specifications, in order to make the structuring and formatting simple and easy to read. [44.]

The high-level test environment of the analyzer project is divided into multiple lower level subcategories that form the complete folder structure. The *configuration folder* contains the global variables such as file paths and other configuration values. The *libraries folder* that is further divided into python based custom libraries, Robot Framework keyword libraries, other scripts that support the test functionality and the simulator specific configuration files. The *test suites* folder that includes all of the tests suites, separated by their test type or the area in the tested software. The top-level directory also contains all other configuration and requirement files as well as the script files that are used to run the correct test suite categories based on the different pipeline triggers.

```

test_suites > smoke_example.robot > ...
1  *** Settings ***
2  Documentation      Test suite for Cascadion automated smoke tests.
3  Resource           ../libs/robot_keywords/CMD_Keywords.robot
4  Resource           ../libs/robot_keywords/GUI_Keywords.robot
5  Resource           ../libs/robot_keywords/Combined_Keywords.robot
6  Variables          ../conf/variables.py
7  Library            OperatingSystem
8  Library            Process
9  Library            BuiltIn
10 Library            ../libs/python_libs/AutomationCommands.py
11
12
13 *** Settings ***
14 Suite Teardown     Close Application      ${SIMULATOR_GUI_LOCATION}
15 ...                AND                  Take Screenshot          Failed_test.jpg
16
17 *** Keywords ***
18
19 Start Simulator And Login
20     Start The Simulator
21     Login To Gui As          ${SUPER_USER}    ${SUPER_PASS}
22
23 End Test Case
24     Close Application      ${SIMULATOR_GUI_LOCATION}
25
26 *** Test Cases ***
27
28 #Vitamin D calibration test flow start.
29 Prepare And Start Simulation For Vitamin D
30     [Tags]    VIT_D_CAL_1
31     Prepare Simulation Dependency Files
32     Start Simulator And Login
33     Dismiss Cartridge Alert
34
35 Prepare Instrument Consumables
36     [Tags]    VIT_D_CAL_2
37     Prepare Instrument Consumables For Usage
38
39 Prepare For Vitamin D Testing
40     [Tags]    VIT_D_CAL_3
41     Prepare Assay Specific Tests
42     Prepare Assay Specific Consumables
43

```

Figure 23. The test structure of an example smoke test suite

#### 4.4.3 Keywords and syntax

The basis for the Robot Framework testing are the human readable and re-usable keywords that act as the construction material for the test cases and further test suites. The keywords can be split in two categories that include the *library keywords* that derive from the imported libraries and the *user keywords* that are created directly in the test suites or as a separate keyword libraries. [44.]

The *user-defined keywords* are thought of one of the most compelling features of Robot Framework and introduce the ability to combine other keywords to create new higher-level ones. The keywords include information and operations from the standard and build-in libraries as well as any other actions introduced by any user-defined keywords or libraries. They follow the same Robot Framework syntax as the test cases and offer basic functionality and setting such as separate documentation, keyword tagging, arguments that take information inside keywords, return values to send data out of the keywords, keyword teardown to specify what happens after the keyword is run, and the possibility to set a timeout for the keywords. Some of the features are demonstrated in figure 24. [44.]

The analyzer project testing utilizes user-defined keywords that are split into GUI-keywords that control the functionality of the graphical user interface part of the simulator, CMD-keywords that control the command line extension for the simulator as well as combined keywords that incorporate the keywords from the previous two libraries along with other functionality separate from the simulator software, such as the package installer software. Most of these keywords utilize functionality from the custom Robot Framework- and Python libraries that implement the needed automation frameworks, with the wide application of the standard- and built-in Robot Framework libraries. An example of a user-defined keyword is displayed in figure 24.

```

Login To Gui As
  [Arguments]          ${username}  ${password}
  Log To Console       Logging in as ${username}
  Switch To Gui Process If Needed
  Enter Text To Element  ${username}  class_name=TextBox  auto_id=LoginNameBox
  Enter Text To Element  ${password}  class_name=PasswordBox  auto_id=PwdBox
  Locate And Click Element  5          5s          auto_id=LoginButton

```

Figure 24. A user-defined keyword that performs the login functionality.

#### 4.4.4 Libraries and tools

The Robot Framework keyword libraries are another powerful feature that allow the easy and direct extendibility on top of the normal user-defined keyword functionality. The framework comes with multiple readymade libraries that can be split into three main categories. The *standard libraries* include universal tools such as BuiltIn, OperatingSystem, DateTime and Screenshot libraries that are distributed together with the core framework and therefore making them directly available as normal keywords. *External libraries*

however must be installed separately but still provide a lot of ready-made and directly available keywords and utilities created by other open source contributors. The third option is to create *custom libraries* if the desired functionality is not found from the other available choices. The libraries tend to be very specific and the lower level keywords that they introduce are usually implemented using more conventional programming languages like Python or Java and can therefore utilize all the functionality and extensions provided by them as well as directly combine them with the Robot Framework utilities. [44.]

The custom libraries are implemented with the help of the three different provided Application Programming Interfaces, and the remote library interface if any other programming languages are needed. The simplest way is to use the *static API* to implement a module or a class that has the methods that map directly to keyword names inside it. The methods are also able to use the same functionality such as arguments, return values and failure reporting, as the direct robot keywords. Another option is to use the *dynamic API* to create libraries that implement a method to resolve the names of the keywords they implement, in combination with another method that executes the named keywords with the given arguments. This also allows the dynamic establishing of the needed keywords and their execution during the runtime. The third option is to use the *hybrid API* which combines the functionality of the two other APIs and allows the identification of keywords to be done dynamically while still implementing direct method classes. [44.]

In order to utilize the libraries, they must be imported and taken into use by using the library setting. The standard libraries can be imported just by using their names, but the custom libraries require the specification of the library location such as a file path. Importing the libraries is demonstrated in the figure 25. [44.]

```
1  *** Settings ***
2  Variables    ../../conf/variables.py
3  Library      ../python_libs/GUIController.py
4  Library      OperatingSystem
5  Library      Process
6  Library      BuiltIn
7  Library      String
8  Library      Collections
9  Library      WhiteLibrary
10 Resource    GUI_Keywords.robot
```

Figure 25. The importing of variable files, resource files and libraries.

The analyzer project uses multiple custom libraries implemented with Python to achieve the automation functionalities required by the simulator software. These libraries mostly correspond to the ones on the Robot Framework side and contain the GUI controller class that includes the methods for automating the simulator software graphical user interface, the CMDController class that handles the usage of the simulator command line functionality, AutomationCommands class that contains all other automation related methods such as Windows automation and database related operations. An example of the custom Python library keyword is displayed in figure 26.

```
def start_the_simulator_py(self, simulator_location, sleep_time=20):
    '''Starts the watchdog.exe and the instrument simulator from a given location.'''
    try:
        os.startfile(simulator_location)
        BuiltIn().set_global_variable("${CURRENT_WINDOW}", "NOTSET")
        logger.console("The simulator found and started!")
        time.sleep(sleep_time)
    except Exception as exception:
        raise Exception("The application was not found or did not start properly. Reason: " + str(exception))
```

Figure 26. A custom Python keyword from the AutomationCommands custom library.

#### 4.4.5 Test results and output

Robot framework generates multiple different customizable result files in the desired file location after the test execution. The output is folder and its contents are by default created to the relative test suite location where the tests are ran from but can be directed to specific locations by using the output directory flag and defining either the absolute or relative path. Customization of the test reports allows changes such as title editing, timestamping and changing the report designs and colors. [44.]

The first generated result file is called the *output file* and it contains all the test execution and result information in XML format. The file is machine readable and is also used to generate the other result files that can be further combined and post processed. The second result file is the *log file* that consists of specific details about the tests that were run. The file has a hierarchical structure that includes information about the test suites, tests cases and keywords in a HTML format. The log file provides a way to investigate the results in higher detail and are especially useful when targeting and investigating the reason why a test failed. The third report file type is the *report file* that contains an analysis of the test results. The results are separated by the test suites, test tags and the executed test cases, and provide a good overall picture of the test execution status in

easy to read HTML format. On top of the three main test result files, it is possible to include an additional *XUnit file* that can be used as input data for compatible external tools, in order to generate different test reports and statistics. An example of the log result file is demonstrated in the figure 27. [44.]

The analyzer project environment generates separate result files for the continuous integration software installation smoke tests and the nightly simulator integration tests. All of the result files are stored both locally on the tests server and in the respectable release folder destinations located in the company network. The files are also uploaded into Azure Blob Storage containers during the pipeline runs and are retained there for the duration of two weeks. Additionally, the XUnit result file is published by the Azure pipeline task and used to form and display test report directly in the Azure DevOps Services cloud environment.

## Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	5	2	3	00:02:14	
All Tests	5	2	3	00:02:14	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
NOT robot:exit (combined)	3	2	1	00:02:14	
robot:exit	2	0	2	00:00:00	
SETUP-1	1	1	0	00:00:45	
SETUP-2	1	1	0	00:00:25	
SETUP-3	1	0	1	00:01:03	
SETUP-4	1	0	1	00:00:00	
SETUP-5	1	0	1	00:00:00	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Installation Package Test	5	2	3	00:02:20	

## Test Execution Log

<div style="display: flex; justify-content: space-between;"> <span>[-] <b>SUITE</b> Installation Package Test</span> <span>00:02:20.346</span> </div> <p>Full Name: Installation Package Test</p> <p>Documentation: The newest refactored version of the Cascadion installation package test with the silent installation of the help files. Separate variables can be found and modified from ../conf/variables.py Windows User Acces Control must be turned of in order to run the automation!</p> <p>Source: C:\TestAutomation\CascadionAutomatedTests\test_suites\installation_package_test.robot</p> <p>Start / End / Elapsed: 20201026 11:13:13.268 / 20201026 11:15:33.614 / 00:02:20.346</p> <p>Status: 5 critical test, 2 passed, <b>3 failed</b> 5 test total, 2 passed, <b>3 failed</b></p>
<div style="display: flex; justify-content: space-between;"> <span>[+] <b>TEARDOWN</b> Close Instances After Suite Execution</span> <span>00:00:05.860</span> </div>
<div style="display: flex; justify-content: space-between;"> <span>[+] <b>TEST</b> Uninstall Old Software If Needed</span> <span>00:00:45.207</span> </div>
<div style="display: flex; justify-content: space-between;"> <span>[+] <b>TEST</b> Start Setup</span> <span>00:00:25.067</span> </div>
<div style="display: flex; justify-content: space-between;"> <span>[-] <b>TEST</b> Begin Installation</span> <span>00:01:03.349</span> </div> <p>Full Name: Installation Package Test.Begin Installation</p> <p>Documentation: Runs through the first installation steps.</p> <p>Tags: SETUP-3</p> <p>Start / End / Elapsed: 20201026 11:14:24.403 / 20201026 11:15:27.752 / 00:01:03.349</p> <p>Status: <b>FAIL</b> (critical)</p> <p>Message: ERROR! The element was not found or could not be clicked after 10 retries</p>
<div style="display: flex; justify-content: space-between;"> <span>[+] <b>KEYWORD</b> BuiltIn.Log To Console Starting the installation...</span> <span>00:00:00.000</span> </div>
<div style="display: flex; justify-content: space-between;"> <span>[+] <b>KEYWORD</b> GUI_Keywords.Locate And Click Element 10, 5s, class_name=Button, auto_id=1, title=Next &gt;</span> <span>00:00:07.702</span> </div>
<div style="display: flex; justify-content: space-between;"> <span>[+] <b>KEYWORD</b> GUI_Keywords.Locate And Click Element 10, 5s, class_name=Button, auto_id=1, title=Install</span> <span>00:00:55.645</span> </div>
<div style="display: flex; justify-content: space-between;"> <span>[+] <b>TEST</b> Wait For Installation And Finalize It</span> <span>00:00:00.000</span> </div>
<div style="display: flex; justify-content: space-between;"> <span>[+] <b>TEST</b> Verify Help Files</span> <span>00:00:00.000</span> </div>

Figure 27. An example of an installation smoke test log result file.

### 4.4.6 Automation and other tools

The test automation of the analyzer software utilizes a few different libraries to fulfil the Windows GUI automation and the other testing needs.

The main tool and addition to Robot Framework in terms of the project is the Python based Windows automation framework called *Pywinauto*. It Is composed of a set of different python modules that make it possible to automate and simulate GUI operations

such as mouse and keyboard actions. The tool is installed and imported as a part of the Python modules and requires the following packages to be installed on the test system: *PyWin32* that provides multiple Windows APIs to be used with Python, *Comtypes* that allows the COM interface functionality, and the *Six compatibility library* to provide smoothing between the Python versions. After the modules are available the tool itself can be downloaded, unpacked and imported into the custom libraries. The library uses either of the two Windows accessibility technologies called the Win32 API and the MS UI Automation to connect its backend. The automated application process and the graphical interface or operating window are connected to the backend and used to locate the automation elements based on their attributes such as names, values, classes, automation IDs or other control types. The attributes of the tested elements can be found using a GUI inspection tool such as *inspect.exe* or directly from the source code. After the successful locating the elements, multiple actions that operate or simulate the element's functionality can be performed and automated. The tool is used to access and automate most of the elements used in the software simulator integration tests as well as the installation smoke tests in the analyzer test environment. The library is imported into the custom libraries where the Python methods utilize its capabilities and then execute the formed keywords using Robot Framework as shown in figure 28. [45.]

```
def enter_text_to_element(self, input, **controls):
    try:
        element = self.locate_control(**controls)
        element.set_text(input)
        logger.console(input + " entered to the given element.")
    except Exception as exception:
        logger.console("Entering text failed. Reason: " + str(exception))
```

Figure 28. A user-defined Python keyword utilizing Pywinauto.

The secondary automation tool used in the test environment is a third-party external Robot Framework library called *WhiteLibrary*. It acts as a wrapper for the White automation framework and provides similar functionality as Pywinauto but in .NET environment. The keywords are directly available in Robot Framework by just downloading and importing the library, but do not offer as much customization and extendibility as Pywinauto and are therefore used only for simple automation tasks such as handling Windows based dialog windows. An example of the WhiteLibrary keyword is displayed in figure 29. [46.]



```

Add Sample to Rack
[Arguments]                ${sample_id}
Switch To Gui Process If Needed
WhiteLibrary.Click Item    text:${sample_id}
WhiteLibrary.Click Button  AddToRackButton

```

Figure 29. An example of a user-defined keyword utilizing the WhiteLibrary.

Along with the automation features needed in the testing, there are some additional test focuses and useful utilities that help the test structure and execution. The instrument software is heavily in relation with relational databases and stores a lot of related data into SQL databases. Therefore, the custom python libraries also utilize the database connections to perform various actions by running predefined SQL scripts that can be used to for example set up, initialize and modify the databases for the simulator usage. This makes the integration testing faster and easier since the automation does not have to wait for the simulated actions to happen in real time and can skip some parts of the laborious setup and execution. Apart from database operations, the test libraries include some additional file handling tasks that are used to transfer and operate instrument software related configuration files such as medical assays and other dependency files. Some of the database related functionality is demonstrated in figure 30.

```

def run_sql_script(self, script_path, database_name):
    logger.console(f"Running SQL script '{script_path}' on database '{database_name}'")
    process = subprocess.Popen([f"osql" , "-E", "-S", r".\CASCADION", "-d", database_name, "-i", script_path])
    process.wait()
    logger.console(f"SQL script completed")

```

Figure 30. A user-defined Python keyword utilizing the database functionality

## 5 Results, improvements and conclusion

The research and investigation towards DevOps and its concepts such as continuous integration, as well as the study on different software testing and project management models, processes and approaches helped to come up and end with a solid continuous integration and test automation platform and basis for extendibility towards the final goal of the official validation of the test automation process. The study also helped to understand the software testing in bigger picture as well as in the role of a software engineer.

The source code is stored into a robust location from where it can be easily and continuously unit tested, built and packaged into an installer which is then automatically tested and released for further usage, testing and delivery. The separate and nightly automated integration testing was able to be started and is ready to be extended with new tests and functionality created by either developers or the testers, in order to support the rigorous testing practices of the IVD field.

## 5.1 Alternative tools

The tools and platform used in the project were mostly selected based on the compatibility, efficiency and accessibility factors but there are multiple other DevOps, version control, continuous integration, continuous delivery and test automation solutions available on the market to consider.

Alternative options for DevOps and integrations tools include Jenkins for its easy availability, open source model and extendibility, Bamboo that includes many pre-built functionalities and the integration with Jira and Bitbucket, as well as GitHub actions for the new innovations and the direct GitHub platform that has become the industry standard on the version control side. Other good options to consider are TeamCity, GoCD, GitLab, Circle CI and many others from the continuously growing tool selection. [47.]

As for test automation, the considerable options in the case of Windows UI automation would be CodedUI with the seamless integration to Microsoft tool stacks and Visual studio services, TestComplete for its easy usage and wide range of supported technologies, and Ranorex for its long history and solid performance. There are obviously more notable tool options such as Selenium, and especially for the more common use case on web testing. Most of these tools need very specific configuration and customization to be suitable for every kind of systems under test and some of them are stand alone and do not provide the needed customization options or are solely based on automation tasks. [48.]

## 5.2 Future development ideas

As the concrete idea for the project was to plan and implement the start for the DevOps properties, Continuous integration and test automation, there are many clear improvements as well as some future development ideas to implemented.

One of the biggest improvements would be the addition of cloud based virtual machines, which would make the availability and distributing the test runs a lot easier and lower the execution times significantly, than with a single test server. Another option would be to implement a test coordinator that would spread the automated tests to multiple local virtual machines and gather the test results from them. Similar thing could be implemented with containers, although the containers not being able to access the UI would cause a problem with the GUI tests. All in all, any additional automation and robustness could be further developed as the project related things progress further. Whether it would be added Cloud or DevOps services or even some robotic process automation implemented with the constantly growing Robot Framework.

## References

- 1 Agile 101. [online] Accessed 30 August 2019. Available from: <https://www.agilealliance.org/agile101/>
- 2 The Agile Manifesto. [online] Accessed 30 August 2019. Available from: <https://www.agilealliance.org/agile101/the-agile-manifesto/>
- 3 7 Guiding principles of Lean development. [online] Accessed 2 September 2019. Available from: <https://leankit.com/learn/lean/principles-of-lean-development/>
- 4 Lean software development. [online] Accessed 2 September 2019. Available from: [https://en.wikipedia.org/wiki/Lean\\_software\\_development](https://en.wikipedia.org/wiki/Lean_software_development)
- 5 Ernest Mueller. 2010. What is DevOps? [online] Accessed 6 September 2019. Available from: <https://theagileadmin.com/what-is-devops/>
- 6 DevOps: Breaking the Development-Operations barrier. [online] Accessed 7 September 2019. Available from: <https://www.atlassian.com/devops>
- 7 Floris Erich, Chintan Amrit, M. Daneva. 2017. A Qualitative Study of DevOps Usage in Practice. [online] Accessed 10 September 2019. Available from: [https://www.researchgate.net/publication/316879884\\_A\\_Qualitative\\_Study\\_of\\_DevOps\\_Usage\\_in\\_Practice](https://www.researchgate.net/publication/316879884_A_Qualitative_Study_of_DevOps_Usage_in_Practice)
- 8 Mark Sigler. 2018. A Healthy DevOps Toolchain. [online] Accessed 14 September 2019. Available from: <https://electric-cloud.com/blog/healthy-devops-toolchain/>
- 9 Brett Johnson. 2019. Understanding a DevOps Toolchain: Use Cases and Fundamentals. [online] Accessed 14 September 2019. Available from: <https://www.networkcomputing.com/networking/understanding-devops-toolchain-use-cases-and-fundamentals>
- 10 DevOps Toolchain. [online] Accessed 14 September 2019. Available from: [https://en.wikipedia.org/wiki/DevOps\\_toolchain](https://en.wikipedia.org/wiki/DevOps_toolchain)
- 11 Martin Fowler. 2006. Continuous Integration. [online] Accessed 4 November 2019. Available from: <https://martinfowler.com/articles/continuousIntegration.html>
- 12 Anu Upadhyay. Centralized vs Distributed Version Control: Which One Should We Choose? [online] Accessed 6 November 2019. Available from: <https://www.geeksforgeeks.org/centralized-vs-distributed-version-control-which-one-should-we-choose/>

- 13 Ravi Verma. Centralized vs Distributed Version Control Systems [CVCS vs DVCS]. [online] Accessed 6 November 2019. Available from: <https://scmquest.com/centralized-vs-distributed-version-control-systems/>
- 14 Vince Power. 2019. What is a Continuous Integration and Delivery Pipeline and Why Is It Important? [online] Accessed 12 November 2019. Available from: <https://codefresh.io/continuous-integration/continuous-integration-delivery-pipeline-important/>
- 15 Samarpit Tuli. 2018. Learn How to Set Up a CI/CD Pipeline from Scratch. [online] Accessed 15 November 2019. Available from: <https://dzone.com/articles/learn-how-to-setup-a-cicd-pipeline-from-scratch>
- 16 Continuous integration vs. continuous delivery vs. continuous deployment [online] Accessed 17 November 2019. available from: <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>
- 17 Raj Kumar. 2015. What Is Software Testing – Definitions, Types, Methods, Approaches. [online] Accessed 6 February 2020. Available from: <https://www.softwaretestingmaterial.com/software-testing/>
- 18 Deb Sayantini. 2019. What are the Types of Software Testing Models? [online] Accessed 6 February 2020. Available from: <https://www.edureka.co/blog/software-testing-models/>
- 19 SDLC – Waterfall Model [online] Accessed 8 February 2020. Available from: [https://www.tutorialspoint.com/sdlc/sdlc\\_waterfall\\_model.htm](https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm)
- 20 SDLC – V-Model [online] Accessed 9 February 2020. Available from: [https://www.tutorialspoint.com/sdlc/sdlc\\_v\\_model.htm](https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm)
- 21 SDLC – Agile Model [online] Accessed 10 February 2020. Available from: [https://www.tutorialspoint.com/sdlc/sdlc\\_agile\\_model.htm](https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm)
- 22 SDLC – Iterative Model [online] Accessed 12 February 2020. Available from: [https://www.tutorialspoint.com/sdlc/sdlc\\_iterative\\_model.htm](https://www.tutorialspoint.com/sdlc/sdlc_iterative_model.htm)
- 23 SDLC – Spiral Model [online] Accessed 14 February 2020. Available from: [https://www.tutorialspoint.com/sdlc/sdlc\\_spiral\\_model.htm](https://www.tutorialspoint.com/sdlc/sdlc_spiral_model.htm)
- 24 8 Phases of Software Testing Life Cycle (STLC). 2020 [online] Accessed 15 February 2020. Available from: <https://www.softwaretestinghelp.com/what-is-software-testing-life-cycle-stlc/>
- 25 Test Artifacts. 2020 [online] Accessed 2 March 2020. Available from: <https://www.professionalqa.com/test-artifacts>

- 26 Black Box Testing Tools [online] Accessed 4 March 2020. Available from: <https://www.ranorex.com/black-box-testing-tools/>
- 27 Black Box Testing [online] Accessed 6 March 2020. Available from: <http://softwaretestingfundamentals.com/black-box-testing/>
- 28 Black Box Testing: An In-Depth Tutorial with Examples and Techniques. 2020 [online] Accessed 6 March 2020. Available from: <https://www.softwaretestinghelp.com/black-box-testing/>
- 29 White Box Testing [online] Accessed 7 March 2020. Available from: <http://softwaretestingfundamentals.com/white-box-testing>
- 30 White Box Testing: A complete Guide with Techniques, Examples, & Tools. 2020 [online] Accessed 8 March 2020. Available from: <https://www.softwaretestinghelp.com/white-box-testing-techniques-with-example/>
- 31 Grey Box Testing [online] Accessed 10 March 2020. Available from: [https://www.tutorialspoint.com/software\\_testing\\_dictionary/grey\\_box\\_testing.htm](https://www.tutorialspoint.com/software_testing_dictionary/grey_box_testing.htm)
- 32 What is Exploratory Testing? Techniques with Examples. [online] Accessed 16 March 2020. Available from: <https://www.guru99.com/exploratory-testing.html>
- 33 Software testing [online] Accessed 18 March 2020. Available from: [https://en.wikipedia.org/wiki/Software\\_testing](https://en.wikipedia.org/wiki/Software_testing)
- 34 Levels of Testing in Software Testing [online] Accessed 19 March 2020. Available from: <https://www.guru99.com/levels-of-testing.html>
- 35 Differences Between the Different Levels & Types of Testing [online] Accessed 22 March 2020. Available from: <https://reqtest.com/testing-blog/different-levels-of-testing/>
- 36 Functional Testing Vs Non-Functional Testing: What's the Difference? [online] Accessed 23 March 2020. Available from: <https://www.guru99.com/functional-testing-vs-non-functional-testing.html>
- 37 Automation Testing Tutorial: What is Automated Testing? [online] Accessed 8 April 2020. Available from: <https://www.guru99.com/automation-testing.html>
- 38 Azure DevOps [online] Accessed 23 April 2020. Available from: <https://azure.microsoft.com/en-in/services/devops/>
- 39 What is Azure Repos [online] Accessed 28 April 2020. Available from: <https://docs.microsoft.com/en-in/azure/devops/repos/get-started/what-is-repos?view=azure-devops>

- 40 Azure Pipelines agents [online] Accessed 14 August 2020. Available from: <https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/agents?view=azure-devops&tabs=browser>
- 41 What is Azure Pipelines [online] Accessed 15 August 2020. Available from: <https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops>
- 42 Introduction to Azure Blob storage [online] Accessed 18 August 2020. Available from: <https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction>
- 43 Robot Framework [online] Accessed 22 September 2020. Available from: <https://robotframework.org/>
- 44 Robot Framework User Guide [online] Accessed 22 September 2020. Available from: <https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html>
- 45 Pywinauto contents [online] Accessed 13 October 2020. Available from: <https://pywinauto.readthedocs.io/en/latest/contents.html>
- 46 Roboframework-whitelibrary [online] Accessed 16 October 2020. Available from: <https://github.com/Omenia/robotframework-whitelibrary>
- 47 20 Best Continuous Integration (CI) Tools in 2020 [online] Accessed 22 October 2020. Available from: <https://www.guru99.com/top-20-continuous-integration-tools.html>
- 48 12 Best Automation Tools for Desktop Apps in 2020 [online] Accessed 23 October 2020. Available from: <https://www.logigear.com/blog/test-automation/12-best-automation-tools-for-desktop-apps-in-2020/#section8>