ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

JOHN VON NEUMANN
FACULTY OF
INFORMATICS

THESIS

OE-NIK
2023

HUNOR ISTVÁN NÁS
T/006664/FI12904/N

Contents

Abbreviations

ABS	Anti-lock Braking System
ALM	Application Lifecycle Management
API	Application Processing Interface
ASIL	Automotive Safety Integrity Level
BASH	Bourne-Again SHell
BCM	Brake Control Module
CAN	Controller Area Network
CI	Continuous Integration
CI/CD	Continuous Integration Continuous Delivery
CLI	Command Line Interface
DAG	Direct Acyclic Graph
DevOps	Software Development and IT Operations
DOS	Windows Disk Operating System
DSL	Domain-Specific Language
DVCS	Distributed Version Control System
EBCM	Electronic Brake Control Module
ECM	Engine Control Module
ECU	Electronic Control Unit
GUI	Graphical User Interface
HIS	Hersteller Initiative Software
HTTP	Hypertext Transfer Protocol
ISO	International Organization of Standardization
LIN	Local Interconnect Network
MISRA	Motor Industry Software Reliability Association
OSI	Open System Interconnection
PaaS	Platform as a Service
SCA	Static Code Analysis
SCM	Suspension Control Module
SDLC	Software Development Life Cycle

SFA	Single File Analysis
TCM	Transmission Control Module
TCM	Telematic Control Module
UI	User Interface
URL	Uniform Resource Locator
VCS	Version Control System
YAML	Yet Another Markup Language

Abstract

The automotive industry is changing rapidly and with it the onboard systems, including embedded software on Electronic Control Units (ECU), resulting in an influx of system complexity. This means ECUs must function properly and adhere to the Automotive Safety Integrity Level (ASIL) scheme. System complexity and proper functionality yields a significantly more dynamic sensor and software. To keep up with proper integration and testing of new functionalities, these processes must be automated. The concept of automating such processes seems simple, however, the optimization, maintenance, inner workings, and component communication is challenging to synthesize. After the automation process has been developed, the metrics and data of the current functioning architecture must be considered and monitored so that the automation process executes the job in a viable amount of time. The aim of this thesis is to build on an existing automated pipeline and develop the most optimized continuous integration and testing process within the constraints of the technology available.

1 Introduction

Everything in the technological field is advancing towards automation; whether that is full, semi, or hybrid automation. The point of automation is to increase a given systems efficiency while keeping the quality of service high. This criteria is especially achievable for monotone tasks. There are two main reasons worth noting to why such tasks are better off automated. The first reason being the health issues tied to monotonous work [24], that is, these types of tasks can be detrimental to mental health, leading to stress and ultimately wearing individuals out to a point of complete exhaustion. Though this point is a bit arbitrary and unrelated to the main topic at hand, it is important to note the humanitarian aspect in the reasoning. The second reason to automate repetitive tasks is to maximize the amount of work that can be completed. A software development model integrated with an automated pipeline will outperform any human contender. Automated tasks can run non-stop, assuming the proper computational resources are always allocated. In an automated development oriented project, both the integration and development of new software is important. Furthermore, in automobile embedded software projects, the proper integration and development of code is of utmost importance since human lives can be affected [17]. Automotive electronics need to function properly to achieve peak vehicle performance and meet safety measures. The different electronic components must be able to handle risky situations quickly and effectively, making risk classification schemes a large part of the automated development pipelines. Understanding the overarching connections, devices, and logical models' relationship to one another brings together the idea of a hybrid development environment. This thesis will focus on the automated development pipelines used to continually integrate embedded software. The environment that will be used to test the pipeline features will be Jenkins. The efficiency, optimization, and overall efficacy of the scripts and pipelines will be evaluated based on overall capability. This thesis aims to build on existing automated Jenkins pipelines that are thoroughly used in the embedded software department at Robert Bosch Kft. These pipelines are a crucial component to delivering validated embedded software updates in a timely fashion.

2 Automotive Electronics

Basic insight on automotive electronics is key in understanding the value of automated pipelines. The evolution of automobiles today is overwhelmingly dominated by computerization. Today's automobiles are equipped with more computers than to their purely mechanical ancestors. The amount of electronics in automobile has dramatically increased and continue to increase by the year [25]. Modern cars have a plethora of electronic systems including, but not limited to, engine electronics, transmission electronics, automotive control network protocols, motor starting electronics, navigational electronics, and electrical system electronics [18]. These electronics are usually present in automobiles with combustion engines. Hybrid and fully electric vehicles have even more electronics onboard. All these systems have to be properly calibrated, tested, and synchronized so that they serve their intended purpose. If one device, connection, or communication network is not functioning properly, the safety of the entire vehicle may be jeopardized [17]. These devices are programmed using low level embedded languages like C and C++. The Jenkins pipelines researched in this paper are working with the embedded software development and integration of components interacting with the different Electronic Control Units (ECUs). The functions and different types of ECUs will be discuss in Section 2.1. Alongside the ECUs, there are two automotive control network protocols that play a vital role in communicating the correct information to and from the different ECUs. One is the Controller Area Network (CAN) and the other is the Local Interconnect Network (LIN). Both are bus lines, which will be discussed in more detail in Section 2.2.

2.1 Function of different Electronic Control Units

An Electronic Control Unit (ECU), often times interchangeably referred to as an Electronic Control Module (ECM), is an embedded system component responsible for controlling and communicating with a given electronic system. It is very similar to any micro-controller board. On any given ECU, there is a processor running software, powered by a dedicated power source. There are many ECUs on a vehicle, each with its own purpose, software, functionality, communication lines, and power [8]. The purpose of each ECU having its own power source is so they can power on and off independent from other ECUs and work independently. Furthermore, each ECU receives different inputs and sends different outputs related to the electronic system it controls. To avoid confusion, this paper will refer to specific ECU types as control modules. Figure 2.1 shows the most well-known ECU types, including the Engine Control Module (ECM), Brake Control Module (BCM), Transmission Control Module (TCM), Telematic Control Module (TCM), and Suspension Control Module (SCM). Figure 2.1 is analogous to a Unified Modeling Language (UML) class diagram where the "Types of ECUs" is the superclass and the specific ECU types are the subclasses.

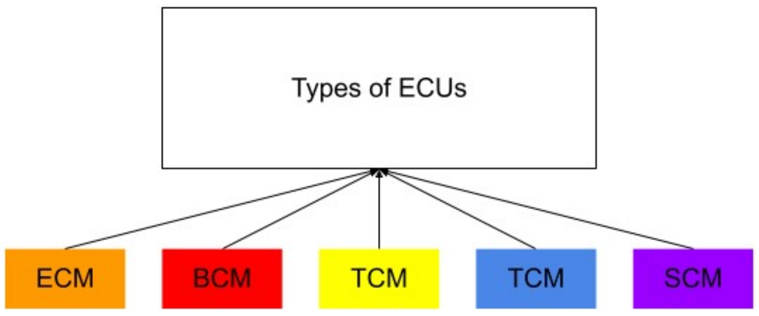


Figure 2.1: Basic Electronic Control Units

2.1.1 Engine Control Module

The ECM gets its input information from sensors placed all around the engine. Some examples of what the sensors measure include fuel flow rate, revolutions per minute, fuel economy, and oil pressure. The ECM uses all of this sensory data to determine the amount of fuel to inject into the combustion chambers at what times. Any information that has to do with engine performance, functionality, and operation is fed into the ECM, processed, and then reacted upon.

2.1.2 Brake Control Module

The BCM controls an automobiles braking system. Sometime this ECU is referred to as the Electronic Brake Control Module (EBCM) if the braking system is electronically driven. Here the BCM takes into consideration the traction control data along with the Anti-lock Braking System (ABS) data to determine how emergency braking should be applied.

2.1.3 Transmission Control Module

The TCM is in control of the automatic transmission. The TCM is only present in vehicles that have an automatic transmission. The TCM receives the revolutions per minute information from the ECM along with the cars actual acceleration and calculates when the vehicle should shift. This results in a very smooth and efficient shifting mechanism.

2.1.4 Telematic Control Module

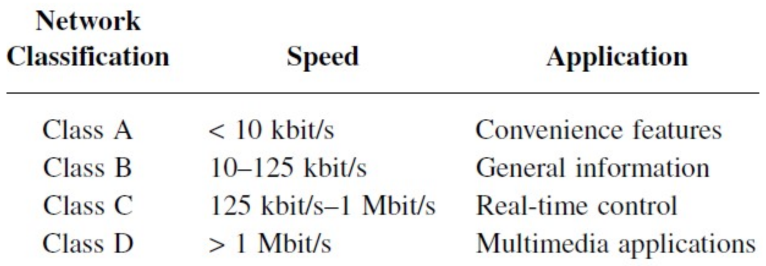
The TCM is in control of the infotainment system alongside the navigational system. It ensures the onboard services are up and running and sufficient satellite data is available for the driver.

2.1.5 Suspension Control Module

The SCM is in control of a vehicles suspension. Some vehicles have different modes, which yield suspension adjustment. This is only possible on vehicles with active suspension systems. The adjustments made by the SCM yield for a comfortable and secure driving experience.

2.2 Automotive Control Network Protocols

Automotive control network protocols, often times referred to as vehicle buses or power buses, are multiplexed communication buses used to connect the main electronic components of a vehicle for safe, secure, and quick communication between one another [18]. The proper functionality of any given ECU is important, but the proper communication between the ECUs and other onboard electronics is just as critical.



Network Classification	Speed	Application
Class A	< 10 kbit/s	Convenience features
Class B	10–125 kbit/s	General information
Class C	125 kbit/s–1 Mbit/s	Real-time control
Class D	> 1 Mbit/s	Multimedia applications

Figure 2.2: Classification of Automotive Networks [18]

These power buses are cabled around the vehicle and connected to various load points. These load points are controlled by smart switches. The entirety of the vehicle bus system communicates on or off loads between different components to achieve a fully in-sync system. Figure 2.2 shows how bus communication lines are split into classes depending on the type of information they must communicate. Class D is a high-bandwidth network mainly used to multimedia applications. This includes infotainment systems, onboard entertainment systems, and network connectivity. Classes B and C are used for systems that must exchange data at faster rates. Class A networks are used for convenience features like windshield wipers, electronically powered windows, or seat adjustment controls.

2.2.1 Controller Area Network

Controller Area Network (CAN) presents a two-wire serial communication system allowing for great system flexibility [38]. The CAN was developed by Robert Bosch in 1987 as a class C controller. The CAN bus can be applied in networks requiring high speed transfer rates as well as low cost multiplexing wiring, making it a versatile technology. CAN is capable of transferring complex data, meaning it is not just a binary communication wiring system. All automotive electronics like ECUs, sensors, and safety systems, often referred to as CAN nodes, are connected using CAN. Since the CAN system is rated to be class C, it can transfer with bitrates up to 1 Mbit/s. The problem CAN solves is having a universally compatible wiring system. Any two CAN nodes can implement the system and communicate efficiently.

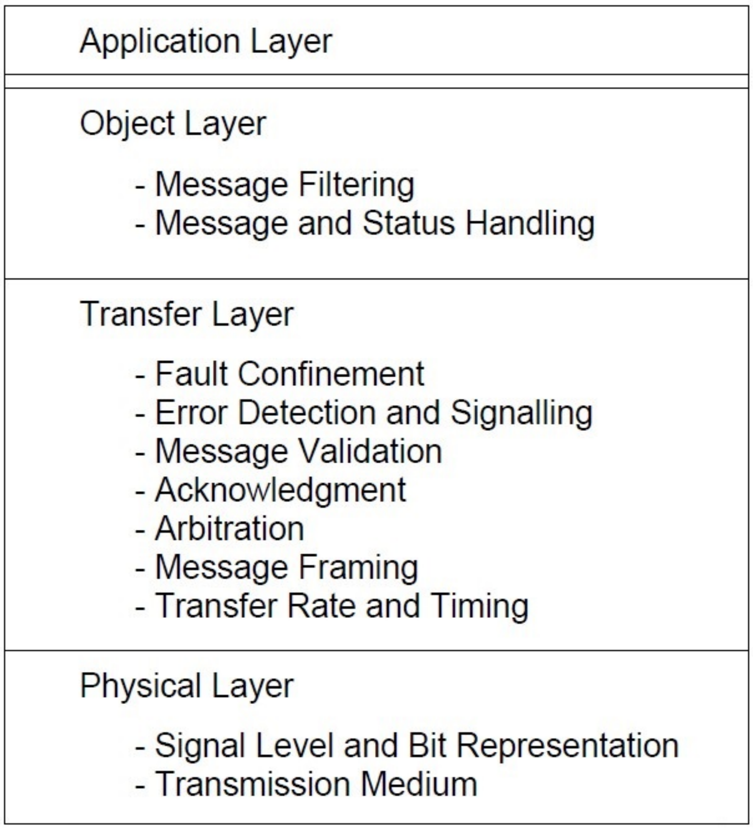


Figure 2.3: Layered Structure of a CAN Node [38]

It is important to note that the two-wire technology paired with twisted pair technology makes the CAN noise tolerant. Since CAN is a network protocol, it operates similar to protocols adhering to the Open System Interconnection (OSI) model. In Figure 2.3 the layers of a CAN node are presented. The Physical Layer defines the actual transmission of signals, how the signals are optimized, and how physical noise is avoided. The Transfer Layer is responsible for the incoming and outgoing messages. It manages bit timing, message framing, error detection and signaling, and fault confinement. The Object Layer filters and handles messages. The basics of CAN are presented to emphasize the importance of proper development of embedded software on the ECUs or CAN nodes.

2.2.2 Local Interconnect Network

Local Interconnect Network (LIN) was developed by the LIN Consortium, which consisted of five automakers: BMW, Volkswagen Group, Audi, Volvo Cars, and Mercedes-Benz. Technological help was provided by Volcano Automotive Group and Motorola. LIN is a class A protocol that allows a data transfer speed of 20 kbit/s [18]. It is a single serial wire transmission line that allows point-to-point communication between a master node and slave nodes. The LIN bus had to be developed since the CAN bus was too expensive to implement for every automotive component. It would not make sense to use a multiplexing bus for functions that only need binary communication. The LIN bus is useful for functions like locking the car, windshield wipers, seat adjustments, or any control communication that does not require a response.

2.3 Relevance of Automotive Electronics and Control Network Protocols in Continuous Integration

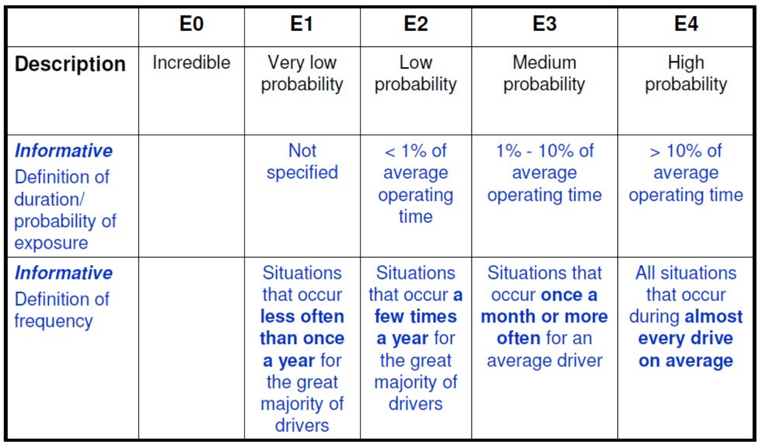
Automotive electronics and control network protocols are an integral part of this project since it provides the scope in which the software is written. All of the software the pipelines continuously integrate is embedded software that runs on different ECUs allowing for the ECUs to communicate through control network protocols. In the development phase, the continuous integration pipelines allow for quick embedded software testing and integration. The ECUs and control network protocols have to work as intended because their functionality ultimately determines overall safety. Continuous integration pipelines and process automation ensure the quick development of embedded software without jeopardizing the quality of electronic functionality.

3 Automotive Risk Classification Schemes

Automotive risk classification schemes are used to evaluate the importance of automotive components in an emergency. There are two main classification schemes applied in modern automotive designs: Automotive Safety Integrity Level (ASIL) and Quality Management (QM). It is important to note the existence of these schemes since the communication between ECUs and other electronic components in a vehicle rely on them. Classifying certain situations and accounting for all possible risks is crucial since human lives are at stake.

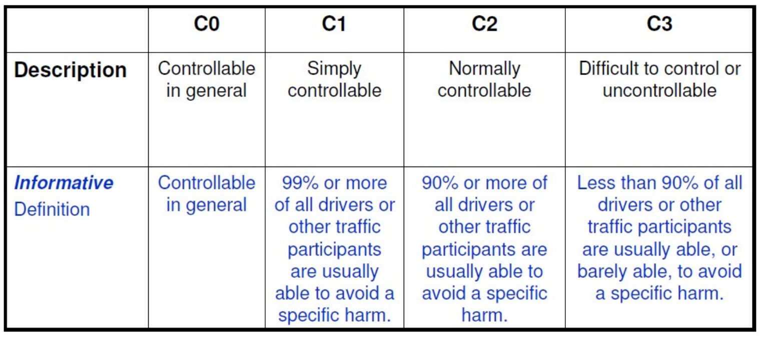
3.1 Automotive Safety Integrity Level

Automotive Safety Integrity Level (ASIL) is defined by the International Organization for Standardization (ISO) functional safety for road vehicle standards [32]. ASIL provides a standard evaluation and response taking into account the probability of exposure to risk (Figure 3.1), controllability of risk (Figure 3.2), and severity of automotive failure (Figure 3.3). From these three components, a risk estimation of possible outcomes' severity is what defines ASIL. ASIL is not a communication technology or a physical component, rather it is a logical and comprehensive system focused on the harm to all drivers. ASIL is divided into four categories; A, B, C, and D. Refer to Figure 3.4 to see how exposure, controllability, and severity levels are mapped to the different ASIL classes. ASIL class D has the most safety critical processes and strictest testing regulations [40]. Imagine a vehicle with an impaired driver has detected a possible collision in 10 meters. The electronics on board have detected this possible risk, and start to evaluate the actions that must be taken if the driver does not start slowing the vehicle down immediately. With ASIL standards, the ECUs communicate with each other and the sensors to prepare for stopping, that is, the power of non-essential components will be cut off (QM level devices, described in Section 3.2). This ensures that the steering servos, braking system, and traction control system will have plenty of resources to carry out the emergency stop. These set of standards are statically defined and applied to every electrical component. The classification of ASIL assigned to a component depends on the criticality or risk calculation measured from the probability of exposure, controllability, and severity of failure.



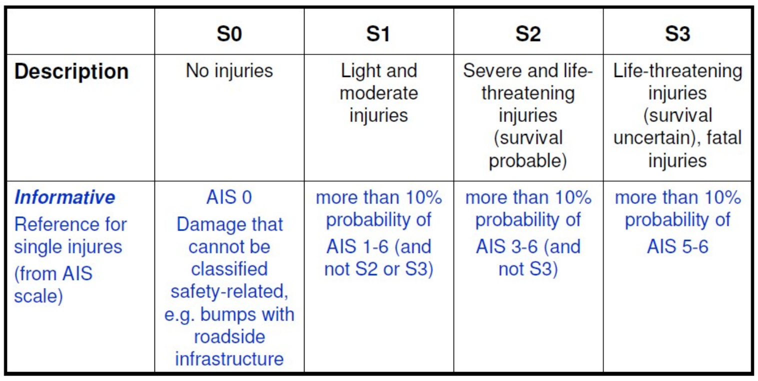
	E0	E1	E2	E3	E4
Description	Incredible	Very low probability	Low probability	Medium probability	High probability
Informative Definition of duration/ probability of exposure		Not specified	< 1% of average operating time	1% - 10% of average operating time	> 10% of average operating time
Informative Definition of frequency		Situations that occur **less often than once a year** for the great majority of drivers	Situations that occur **a few times a year** for the great majority of drivers	Situations that occur **once a month or more often** for an average driver	All situations that occur during **almost every drive on average**

Figure 3.1: Exposure Categories [32]



	C0	C1	C2	C3
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable
Informative Definition	Controllable in general	99% or more of all drivers or other traffic participants are usually able to avoid a specific harm.	90% or more of all drivers or other traffic participants are usually able to avoid a specific harm.	Less than 90% of all drivers or other traffic participants are usually able, or barely able, to avoid a specific harm.

Figure 3.2: Controllability Categories [32]

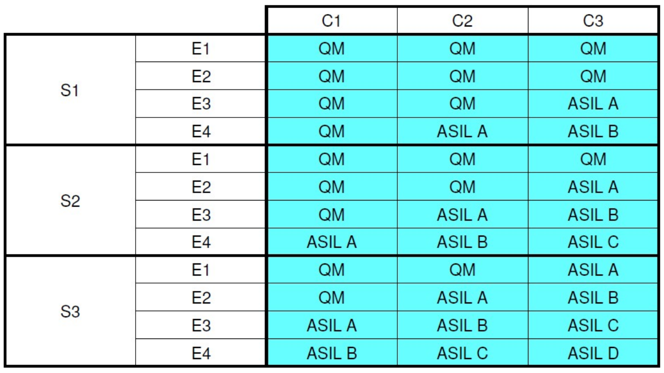


	S0	S1	S2	S3
Description	No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries
Informative Reference for single injures (from AIS scale)	AIS 0 Damage that cannot be classified safety-related, e.g. bumps with roadside infrastructure	more than 10% probability of AIS 1-6 (and not S2 or S3)	more than 10% probability of AIS 3-6 (and not S3)	more than 10% probability of AIS 5-6

Figure 3.3: Severity Categories [32]

3.2 Quality Management

Quality Management (QM) is defined as the lowest degree of automotive hazards. The potential risk of the hazard is deemed too low to consider from a component's functional aspect [32]. Electronics adhering only to QM standards are communicated to in a single direction; they do not send data back to their designated ECU. Instead, they get a binary signal to either do something, or not. Returning to the situation example at the end of Section 3.1, electronics deemed to belong to QM are the electronics to which the power will be cut off. It is a very important process in maximizing vehicle safety, and just as important to apply these standards in the software of each electronic component.



		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	ASIL A
	E4	QM	ASIL A	ASIL B
S2	E1	QM	QM	QM
	E2	QM	QM	ASIL A
	E3	QM	ASIL A	ASIL B
	E4	ASIL A	ASIL B	ASIL C
S3	E1	QM	QM	ASIL A
	E2	QM	ASIL A	ASIL B
	E3	ASIL A	ASIL B	ASIL C
	E4	ASIL B	ASIL C	ASIL D

Figure 3.4: Risk Graph [32]

3.3 Relevance of Automotive Risk Classification in Continuous Integration

It is great if the embedded software development is facilitated in a pipeline, however, it is useless if the criteria under which the software is developed does not adhere to automotive safety standards. Since ASIL and QM are directly correlated to the operation of all ECUs and other electronic components, they must be accounted for in the continuous integration pipelines process. Of course, the schemes themselves are not burned onto the ECU boards, but they play a huge role in the embedded software development process. In turn, the schemes are tested for reliability, execution, and ability to perform when the situations arise. Continuing the example of the impaired driver in Section 3.1, suppose the situation plays out well and the electronic components execute all emergency procedures without issues. The seemingly simple outcome is a result of a complicated process of communication and risk assessment. At some point, this loop of communication and execution had to be written into software and embedded on the different ECUs.

4 Version Control Systems

Version Control Systems (VCS), often times referred to as Source Control Systems, are tools used by software development teams to track the progression of software throughout its life cycle [3]. It is important to keep track of software versioning and development for successful completion of a software project. Usually software development teams consist of multiple developers who tweak, fix, and develop software simultaneously. If no VCS is used to manage the changes, each developers contribution to the project must be manually collected, reviewed, integrated, and finally compiled into one functioning project. Manual management of all code developed in a software team is very tedious, time consuming, and complicated. With a VCS, development teams are able to account for all contributions, while not having to integrate all new software manually. Each developer works on the project individually, solves a problem, and then commits their changes into the project for all the other developers to see. Often times, it may occur that one developer commits changes to a part of a project that another developer is working on. In these cases, the VCS plays a vital role in noticing such conflict and notifying the developer to manually review the source code to ensure the safe merging of both developers' code. There are two main VCS architectures used in modern day software development: distributed and centralized.

4.1 Distributed Version Control System

Distributed Version Control Systems (DVCS) are version control systems where the main repository is the core of a projects source code. Each developer has their own unique version of the main repository locally, called the local repository [16]. This setup can be seen in Figure 4.1. There are as many local repositories as there are developers in a team. As each developer works on a task, they make changes to their local repository first. After their software is confirmed to work, they push the updated software from their local repository onto the main repository. The other developers then synchronize their local repositories with the remote repository containing the new code. This action is referred to as pulling code.

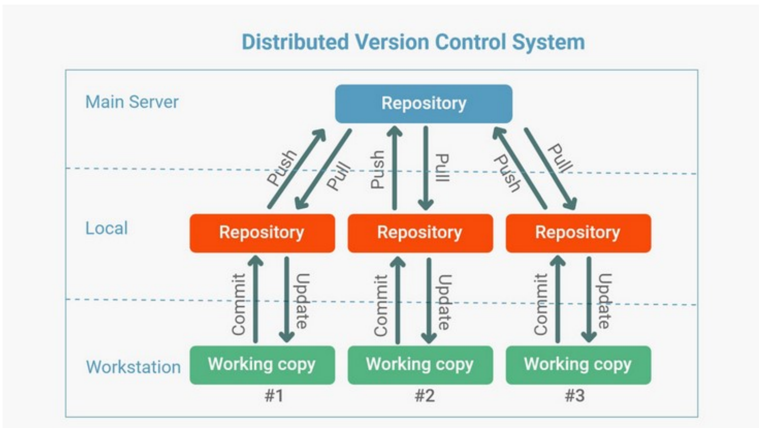


Figure 4.1: Distributed Version Control System Schema [11]

4.1.1 Advantages and Disadvantages of a DVCS

Since each developer has a local copy of the remote repository in a DVCS, they have the capability to work offline. This is a huge advantage because it does not limit the developers to a Virtual Private Network (VPN) or having to be present on a network on which the remote repository resides (including the internet itself) [12]. Furthermore, branching and working in parallel with other team members is very easy using a DVCS. Not only is it easy to create branches to signify a certain feature being added, but this allows for all team members to have access to the entirety of the code at all times. The most basic yet crucial disadvantage of DVCSs is that they are hard to learn [12]. The learning curve is steep in addition to the operations and conventions that must be obeyed for a successful production. Alongside a harder learning process, DVCS repositories with long histories take longer to download and occupy more space since there is a local copy of the repository. The two points are the main disadvantages to DVCS.

4.1.2 Git

Git is a DVCS that is written in C. Git is designed to be a version control system, which teams can use to manages all the changes to their project over time [27]. When a Git project is created, a remote repository is brought into existence. Anything that may be added to the repository is stored as a change in files or the software itself, instead of Git storing all of the software developed. Using a series of actions any developer can manipulate their changes to make sure their software will not be lost. Git also comes with a Command Line Interface (CLI), making the management of source code very robust.

4.2 Centralized Version Control System

Centralized Version Control Systems (CVCS) are version control systems where the central remote repository is a server that is solely responsible for tracking the added software [16]. This setup can be seen in Figure 4.2. Here, the developers do not necessarily have a local repository on which they individually develop, instead, they work directly with the repository present on the server.

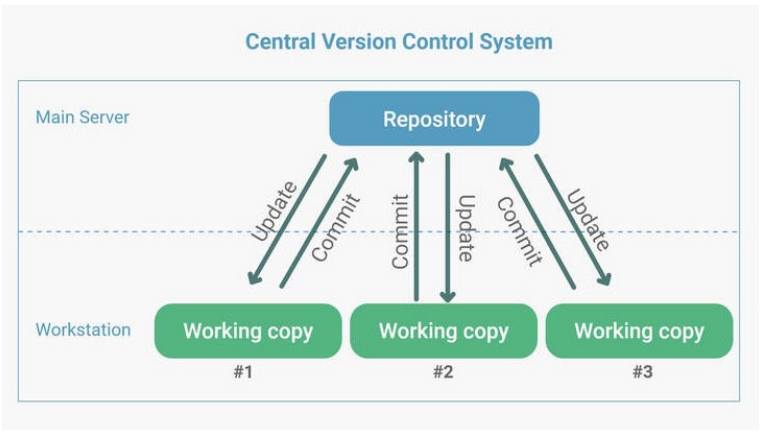


Figure 4.2: Centralized Version Control System Schema [11]

4.2.1 Advantages and Disadvantages of a CVCS

Since the CVCS architecture is significantly simpler, and it does not involve local copies of the remote repository, it is easier to learn [12]. Usually commits are achieve through a Graphical User Interface (GUI) so learning commands to commit code is not necessary. Not having to manage an entire local copy of the repository is also positive when it comes to the initial download of a project. Since CVCS does not require the developer to save the hostory of the project, CVCS projects are not a taxing on the local system. When it comes to software development, one of the hardest problems to manage is merge conflicts [12]. Merge conflicts occur when two developers change the same file. The CVCS then must decide what code snippets are kept, and what code snippets are not. Since developers in a DVCS environment have a local copy of the repository, they tend to face this issue less frequently. When they do, however, they can see the changes that have been pushed and the changes they have locally and review and merge their work accordingly. The largest disadvantage of CVCSs is code availability. If there is only one remote server and it happens to go out of service, nobody can get to the software. This is a huge issue because development then comes to a halt. When compared to its counterpart, DVCSs are significantly more robust. Each developer has their own local copy of the software and can keep working even if the remote server temporarily goes out of service.

4.2.2 IBM Application Lifecycle Management

IBM Application Lifecycle Management (ALM) is a CVCS in which everything is built around a central repository [41]. This repository is hosted on a secure server that can be accessed through a Uniform Resource Locators (URLs) using Hypertext Transfer Protocol (HTTP). The repository stores objects that can be categorized as workspaces, streams and versioned artifacts, which are regular files and folders. Workspaces are objects that are stored in the projects main repository [41]. A workspace stores project components and version-able artifacts, which a client can load into a sandbox present on a personal computer. Very simply put, sandboxes are files and folders in a file system. It is important to note that these components and version-able artifacts are placed under source control, meaning that users with access to the repository workspace can change the content of the items. The workspace itself has an owner who decides what change sets are allowed, in turn determining the modification, saving, and restoration configurations of the workspace (Figure 4.3). Here, flow targets also play an important role in IBM ALM. Flow targets are other workspaces and streams (covered in the next paragraph) that serve as guides to where a certain change set will arrive from or be delivered to. Changes made to the items by a user remain visible to the user only until the changes are delivered to the workspace. Changes from other users can be accepted into a sandbox by loading all change sets.

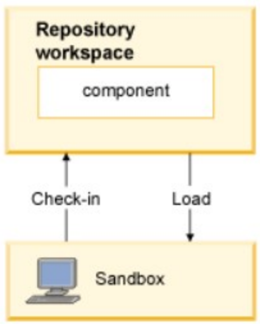


Figure 4.3: IBM ALM Workspaces and Sandboxes [41]

Streams are objects including the components, typically the same component(s) as the repository workspace. Streams can be thought of as branches of a repository workspace. Each stream, however, can only include a single version of a given component at a time. This is especially powerful because using multiple streams in a development team enables different stages of the project to be worked on. See Figure 4.4 to see how a stream fits into a workflow. Figure 4.4 only shows one stream, but imagine there is another parallel stream that is separate from the stream depicted in Figure 4.4. One stream can be used as a production stream and the other can be used as a development stream. On the production stream the latest stable version of software is controlled, whereas the production stream includes components that are being tested given the same environment.

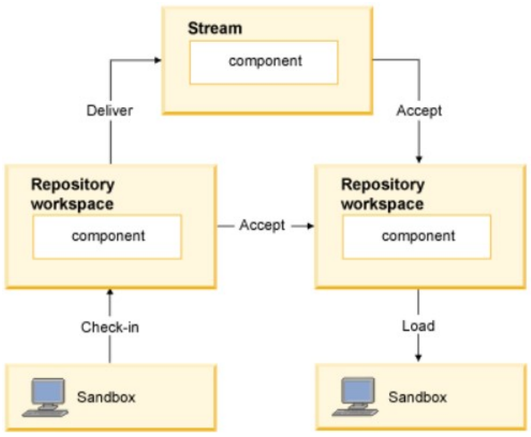


Figure 4.4: IBM ALM Stream [41]

4.2.3 Importance of IBM Rational Team Concert in Continuous Integration

The reason IBM ALM is important is because the project production workflow of which this thesis is a part of utilizes these concepts. Furthermore, these workspaces, components, and streams play a major role in the continuous integration development and how the pipelines are automated. The entire setup of the system used in the project is thoroughly explained in Section 9.

5 Software Development Models and Continuous Integration

For any software development project, it is vital to follow some type of plan in order to deliver promised results on time. There are different thought-out plans used in different software development periods called software development life cycles (SDLCs). The various models include the Waterfall model (W-Model), V-Model, Incremental model, RAD model, Iterative model, and the Spiral model [33]. Though all models present a unique solution to software development, the three models relevant to this thesis are the Waterfall model, V model, and Iterative models. Under the Iterative model falls a very well-adopted modern method called Agile. There is another method called Scrum that falls under the Agile workflow [37]. Some methods can be integrated with continuous integration pipelines better than others. A proper integration of continuous integration pipelines and a fitting development model can potentially result in a huge semi-automated model. With such a model, not only is the software quickly released, but the human communication is greatly improved.

5.1 Waterfall Model

The Waterfall Model, also referred to as the W-Model, is a sequential process where the different procedural steps are thought to follow on another [33]. Every step must be completed before starting the next step. To ensure the project is progressing in the intended direction, a review is conducted at the end of each step that evaluates the relevance of the project. Figure 5.1 shows the different generalized steps taken in sequential order. Since the steps do not overlap, the model is rigid, thus easy to understand and execute. Due to its simplicity, it is perfect in a small development environment. Putting all the positives aside, the Waterfall model presents key weaknesses, especially in the testing phase. If the model were to be followed precisely, it is hard to go back and change anything. This results in an uncertain workflow and is not good for complex projects. For these disadvantages, this model is not a great model to integrate with continuous integration pipelines.

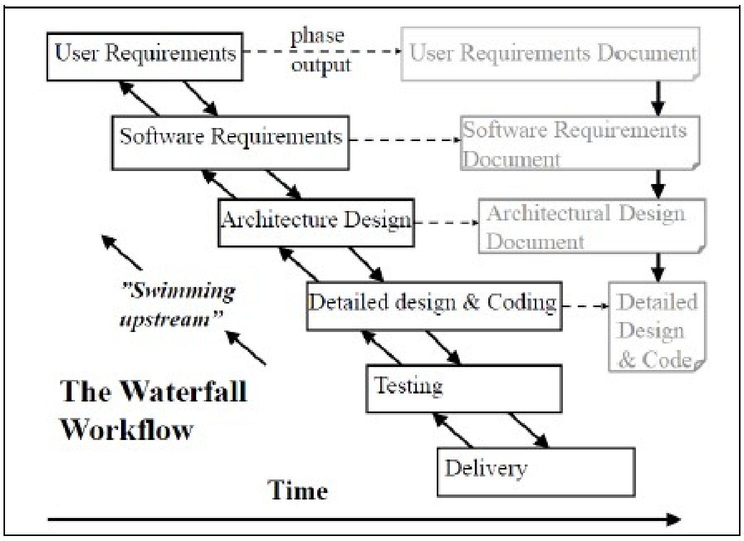


Figure 5.1: Waterfall Model [33]

5.2 Validation and Verification Model

The Validation and Verification Model, commonly referred to as the V-Model, is a variant of the W-Model, where the testing stage is extended into each previous stage of development [20]. See Figure 5.2 for an example. At each stage of development, the work done is thoroughly verified and validated, ensuring the possibility to proceed. Though the V-Model is a great improvement of the W-Model, it is still a rigid structure not taking into account the overlapping nature of certain stages. The sequential stages are not perfectly fit to serve a dynamic project. For example, sometimes the different testing strategies (unit, integration, and system testing) overlap. Applying such tests in the V-Model would result in redundant testing diminishing the efficiency of the model. The V-Model, however, is a great depiction of how a software development should flow, regardless of simplicity. The single V-Model can be modified so that each block represents the actual software to be tested. It can then be extended to a double V-Model or perhaps the triple V-Model; presenting tests on the software blocks and test verification, respectively [20]. Considering the potential software testing, it is possible to pair with automated development. The testing blocks can be executed as soon as there is a software change, furthermore, the different tests can be run in parallel.

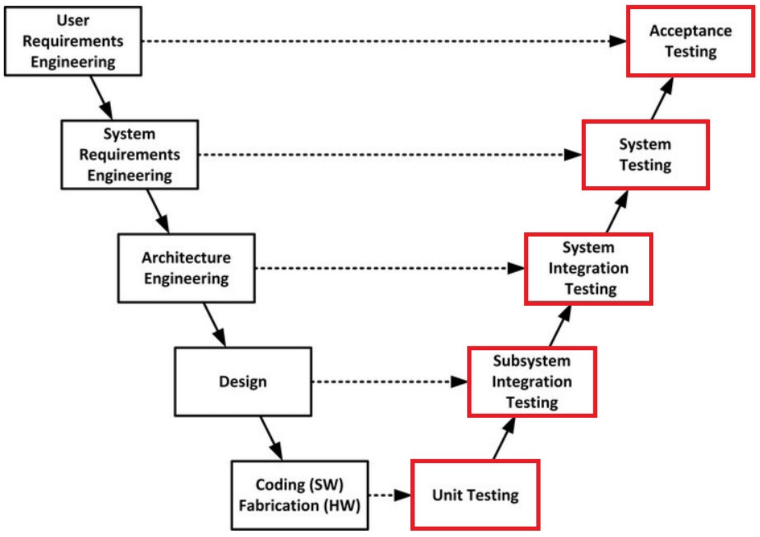

User Requirements Engineering
Acceptance Testing
System Requirements Engineering
System Testing
Architecture Engineering
System Integration Testing
Design
Subsystem Integration Testing
Coding (SW) Fabrication (HW)
Unit Testing

Figure 5.2: Validation and Verification Model [33]

5.3 Iterative Models

Iterative models are a generalized class of software development models in which the full scope of a project is not specified in the beginning. Instead, parts of the project are specified and implemented, then reviewed, and finally new requirements are specified. Figure 5.3 shows this progression of one step to another. The benefit to an iterative model is the scalability. Large projects can be split into granular iterative levels, progressing the project toward the end goal one step at a time. The largest downside of such models is the uncertainty of project life cycle and cost. Since the model does not put a timeline on the project itself, the different iterative steps can drag out and result in a costly project. This can be solved with a project manager and team leader positions. This is where three modern models come into play: Agile, Kanban, and Scrum.

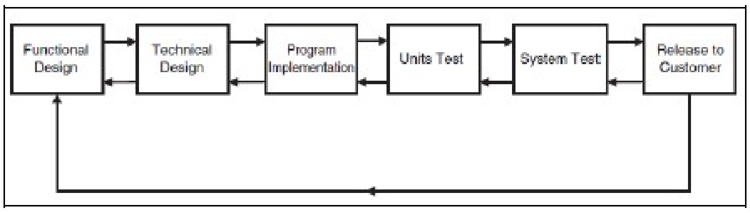


Figure 5.3: Iterative Model [33]

5.3.1 Agile and Other Models

Agile is project management model where the client of a development project is closely involved in the project cycles [37]. The main point of Agile is to provide a community-like structure valuing the client's perspective equally as much as the software developers. There are two specific models that apply the Agile philosophy in two different ways: Kanban and Scrum. Kanban seeks to improve the project workflow through a tool called Kanban board. Kanban board is a visualization tool depicting the specific stages in a project management process. Tasks are then assigned to the specific stages and are moved from one stage to the next until they are completed. To avoid incomplete tasks, the amount of tasks in progress at the same time is limited. Scrum, on the other hand, breaks down into smaller delivery periods called sprints. Within these sprints different development related tasks are completed. Another important distinction in Scrum is the different roles individuals take on. There are Scrum masters, product owners, and project members among whom the tasks are split up. The participation of all roles is what results in successful project deliveries. Integrating these Agile principles with continuous integration pipelines reduces the time feedback takes, thus providing a more fluid workflow [14]. Whether it is a single task in Kanban or a sprints worth of work, the new software has to be validated and tested in the context of the existing project. Integrating, testing, and releasing all the changes manually would yield significantly more work when compared to the continuous validation of software. Customizing given stages in pipelines and creating an environment of pipelines capable of applying different validation schemes. Adding test, compilation checks, and efficiency suggestions creates a hybrid model that can be applied to any project, regardless of size.

5.4 Choice of Software Development Model

The choice of development model in this project is the V-Model. The V-Model ensures safe and secure software releases while maintaining software quality as a whole. Furthermore, V-Model development can be paired very well with continuous integration. As the development cycle proceeds forward, the corresponding testing is executed on the developed code from granular tests to tests with a larger scope. Code belonging to a higher level can be developed in parallel to code belonging to a lower level, which provides a level of parallelism.

6 Deployment Automation Tools

The ultimate purpose of deployment automation tools is to be able to move software under development between testing and deployment environments automatically [39]. Efficiency is the consequence stemming from the ability to continuously develop newer versions of software while testing the previous improvements. This way a given software developer does not have to worry about paying attention to the granular details of software syntax and rule testing alongside functionality testing. Deployment automation plays a huge role in Continuous Integration and Continuous Delivery (CI/CD) pipelines.

6.1 Continuous Integration

Continuous integration is the process of constantly integrating the small changes to software as a project moves on [30]. Every time some software is modified, the continuous integration process checks and validates the syntax, structure, and overall correctness of the changes and then adds it to the entirety of the project. Of course the larger software integration's must be done manually, however, automating the small software integration's results in huge efficiency in the long run. A good manual analogy to such process is the constant pushing of project changes to a GitHub repository, but only when the software has been checked by a partner for errors.

6.2 Continuous Delivery

Continuous delivery is the process of delivering the software changes of a certain project work cycle to the customer or a testing environment automatically [30]. This part of the pipeline relies heavily on the proper testing and integration of the software. If the automatic testing and integration pipeline is not robust enough, the continuous delivery pipeline will run into major difficulties, potentially derailing a projects timeline drastically. In other words, continuous delivery is the automated extension of continuous integration.

6.3 Continuous Deployment

Just like continuous delivery is an extension of continuous integration, continuous deployment is also an extension of continuous integration. The main difference, however, is the releases happen automatically. Refer to Figure 6.1 for a clearer understanding.

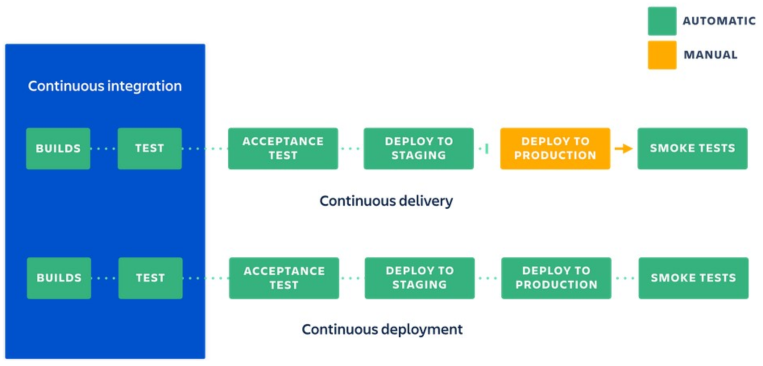


Figure 6.1: Relationship Between Continuous Integration, Delivery, and Deployment [30]

To bring all the concepts together, a CI/CD pipeline usually builds, tests, and deploys the software changes it receives. These processes go from analyzing broad criteria to more focused specifications. A CI/CD pipeline, if well designed, is a huge benefit to a development team. CI/CD pipelines tackle the "little streams make great rivers" concept. The many small integrations and testings done by the pipeline allow for the many quick small deliveries. Though this is a simple concept, there are many different tools an approaches to creating these CI/CD pipelines. Some tools focus on optimizing CI processes, while others focus on optimizing CD processes. Each tool exceeds the others in certain aspects, but has less functionality in the remaining aspects. To gain deeper intuition on the papers topic, it is important to become familiarized with the tools.

6.4 GitHub Actions

The huge benefit GitHub Actions provides is the ability to implement native capabilities right into an already existing GitHub flow [22]. It is very convenient that GitHub Actions can be used for free on all public repositories. On private repositories the free tier includes 2,000 minutes a month of hosted workflows. Figure 6.2 presents the basic components belonging to GitHub Actions.

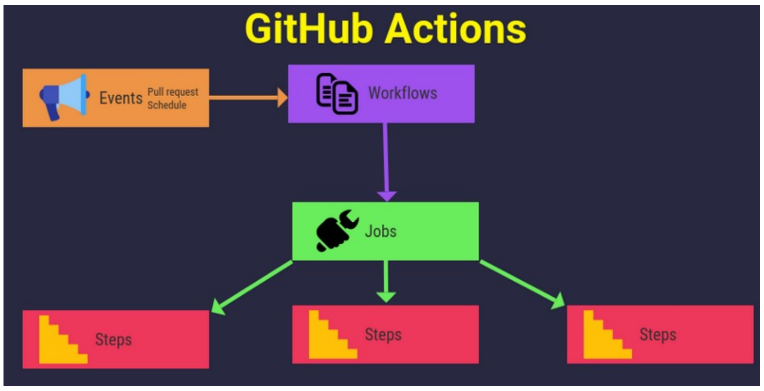


Figure 6.2: GitHub Action and GitLab Workflow [9]

6.4.1 Possibility to Automate Everything

In addition to the easy integration into a GitHub repository, there are tens of thousands of pre-written and tested automations and CI/CD actions available in the GitHub Marketplace. If none of the already established automations or CI/CD actions are fitting to a project, it is also possible to write individual specific actions using custom YAML (Yet Another Markup Language) files.

6.4.2 GitHub Actions Webhook Events

GitHub Actions responds to repository altering webhook events. This means certain action events can be connected to a GitHub repository and when triggered, a given (or multiple) CI/CD pipeline action(s) are automatically started. This function is a huge positive when it comes to having a pipeline connected to a project start on its own. This way the developer has one less component to worry about, creating a more smooth development process. On the other hand, if a given pipeline runs for a long time and requires significant computing resources, this function can be a huge con. Not to mention the small room for error because many times a developer forgets to push a new file or file modification that is important to a software change.

6.4.3 Available Continuous Integration and Continuous Delivery Templates

GitHub Actions also provides CI and CD developer templates designed for development environments. This is a great convenience to developers not well versed in the Software Development and IT Operations (DevOps) realm needing a CI/CD pipeline. GitHub Actions is very user friendly and a great option on the consumer level.

6.5 GitLab

GitLab offers a DevOps platform that provides an entire pipeline that is proprietary to GitLab [36]. Though similar in name to GitHub, GitLab is a completely different service. Gitlab is more developed and feature-rich. The most powerful aspect of GitLab is the ability for development teams to integrate complex tool-chains using a single platform. GitLab can be used as a source code management tool and a continuous integration and delivery tool. This eliminates the need to combine a source code management tool with a continuous integration and delivery tool.

6.5.1 Quicker Development on One Platform

Since GitLab offers all the tools needed to enable a DevOps workflow, it increases development significantly [36]. They also provide reports and metrics, including deployment frequency charts and monitoring, to break down a teams performance. These numbers are presented on a dashboard where the measurements can be analyzed to understand the strengths and weaknesses of a team.

6.5.2 Out-of-the-Box Security and Compliance

To support robust software, GitLab offers some security testing as well as modern software compliance [36]. Many times in a development workflow the emphasis is on getting work done as quickly as possible. This may result in non-compliant software full of bugs. GitLab adds this additional layer to enhance a development teams experience with code compliance and software robustness.

6.6 Atlassian Bamboo

The Atlassian Bamboo automated development tool is a continuous integration server used to create continuous delivery pipelines. It is commonly paired with Atlassian BitBucket repositories [6]. Bamboo is very similar to GitHub Actions in that it provides automated build and test processes to pushed repository code. The biggest advantage to Bamboo, however, is the ability to optimize build performance through parallelism and leveraging elastic resources. Resources of multiple computers can be combined and allocated to certain stages in the CI pipeline to ensure efficient use of those resources. This is achieved by having a central server, which schedules and coordinates all work. That is the Continuous Integration Server in Figure 6.1. This server pulls all changes from the source repository and goes through the build process. After the build process is complete, the outputted executable applications, configuration files, or build results are deployed. The web server interface is a graphical way to interact with the configuration and reporting status of the builds as seen in Figure 6.3.

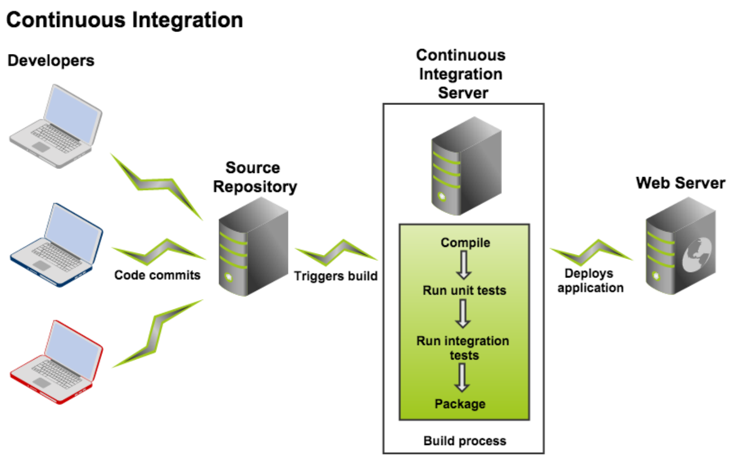


Figure 6.3: Continuous Integration with Bamboo [6]

6.6.1 Bamboo Workflow with a Continuous Integration Server

Bamboo uses a distinct file-like workflow structure, where each following structure in the workflow becomes a more specific job, ultimately resulting in tasks. This structure can be seen in Figure 6.4 (Figure 6.4 found on the next page). Any given software project will have one or more plans. These plans have one or more stages, where the different sequentially important tests and builds are classified. For example, one stage might be a compile test, the second a unit test, and the final stage a build. Under these stages there are jobs. If the above analogy is continued, the compile stage may have different compile criteria applied. Of course for each criteria, there must be specific set of tasks that must be properly executed for the compile to successfully pass. If there are multiple jobs in a stage that are independent of the others results, then the jobs can be run in parallel. This results in a very efficient CI workflow.

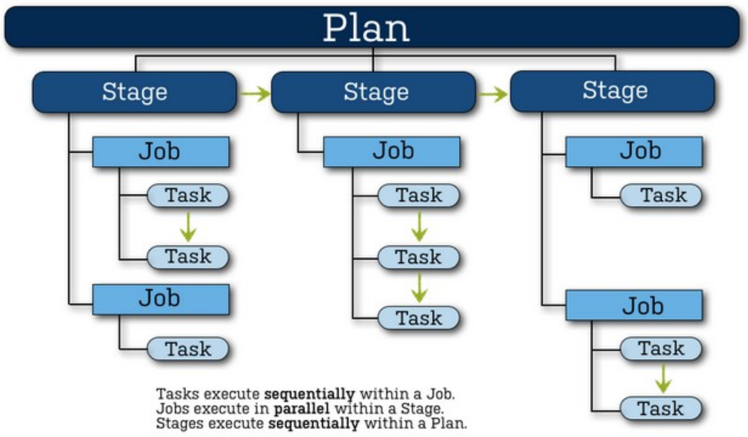

Plan
Stage
Stage
Stage
Job
Task
Task
Job
Task
Job
Task
Task
Task
Job
Task
Job
Task
Task
Tasks execute **sequentially** within a Job.
Jobs execute in **parallel** within a Stage.
Stages execute **sequentially** within a Plan.

Figure 6.4: Bamboo Workflow [6]

6.7 Jenkins Pipeline

Jenkins is an open source automation server used to automated tasks from building and testing to delivering and deploying software [23]. Jenkins is loaded with functionality, most important of which belong to continuous integration. The suite of continuous integration plugins is called Jenkins Pipeline. Jenkins Pipeline offers a vast set of tools for modeling simple-to-complex pipelines as software code using the Jenkins Pipeline Domain-Specific Language (DSL). This DSL code is written and contained in a Jenkinsfile (very similar to a text file), which is then paired with a project's repository. The code-like structure of Jenkins Pipeline makes it very malleable. It is easy to quickly modify processes (including build processes), review pipeline changes, and keep tabs on file modification history. Pipelines in Jenkins are very durable since both planned and unplanned pipeline restarts can be handled without breaking the pipeline. It is even possible to pause a pipeline in the middle of running, modifying certain parameters that have yet to be executed, and then continuing the pipeline to run its full course. It is also important to note that a pipelines progress can be monitored in the Jenkins console through a web UI. Here all of the console outputs and logs can be viewed for debugging and pipeline execution review.

6.7.1 Jenkins Pipeline Workflow

Any given Jenkins Pipeline is a developer defined model of a general pipeline. The pipeline code (DSL syntax Jenkinsfile code) defines the entire build process. Build processes include stages, and stages include steps. A stage includes distinct subsets of tasks like builds, tests, or deployments. The steps themselves are single tasks that happen in a stage. Steps may include running particular scripts, uploading metrics, or configuring given files. The steps in the stages can be run in parallel depending on whether or not the steps rely on each others outputs. The basic Jenkins CD flow can be seen in Figure 6.5.

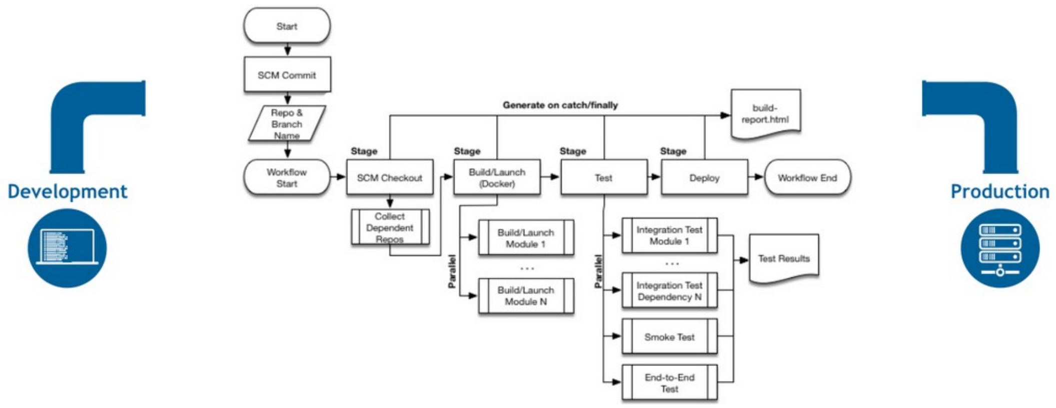


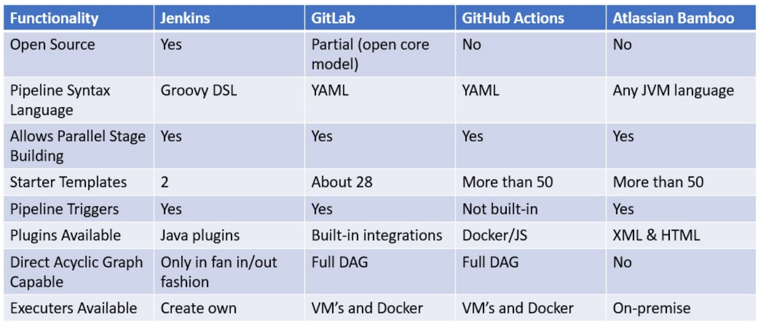
Figure 6.5: Example Jenkins CD Scenario [23]

6.8 Cloud-Based Development Automation Tools

There are other automated development tools like Spinnaker, AWS CodePipline, and Azure Pipelines that are cloud-based tools [4, 5, 35]. These tools allow for third party plugins, which means developers can integrate the GitHub, Bamboo, and Jenkins pipeline into the cloud-based pipeline solution. This is very useful if for a given project, a cloud-based solution is desired. It's tackles the same desires as having a cloud-based server instead of a proprietary server. There are also other not cloud-based automated development tools available like TeamCity (by JetBrains) and CloudBees. TeamCity is very similar to Jenkins, but it is more user friendly [31]. Jenkins is more centered around professional use and TeamCity around conventional development. TeamCity can be pair with TeamCity Cloud, where the TeamCity automated pipelines run on a third party cloud provider. CloudBees on the other hand, is more of a Platform as a Service (PaaS) and fully utilizes Jenkins to offer as a tool [15]. These development tools are all based on either GitHub Actions, Bamboo, or Jenkins and are intertwined with one another, making their functionality the same as the three explained in detail.

6.9 Choice of Automated Development Tool

In this thesis, Jenkins Pipeline will be used for the automated development environment, more specifically the continuous integration pipeline tool, since this thesis is more focused on continuous integration than continuous delivery and deployment. When compared to GitHub Actions, GitLab and Atlassian Bamboo, it is significantly more configurable and less constricted [13]. Figure 6.6 shows a side-by-side comparison of important capabilities of each technology.



Functionality	Jenkins	GitLab	GitHub Actions	Atlassian Bamboo
Open Source	Yes	Partial (open core model)	No	No
Pipeline Syntax Language	Groovy DSL	YAML	YAML	Any JVM language
Allows Parallel Stage Building	Yes	Yes	Yes	Yes
Starter Templates	2	About 28	More than 50	More than 50
Pipeline Triggers	Yes	Yes	Not built-in	Yes
Plugins Available	Java plugins	Built-in integrations	Docker/JS	XML & HTML
Direct Acyclic Graph Capable	Only in fan in/out fashion	Full DAG	Full DAG	No
Executers Available	Create own	VM's and Docker	VM's and Docker	On-premise

Figure 6.6: Comparing Jenkins, GitLab, and GitHub Actions Capabilities [1]

Though GitHub Actions and Bamboo are convenient alongside their respective repositories, they are not scalable enough to be used in large, complex, and specific work environments. Jenkins, however, provides similar possibilities in addition to the ease of creating a pipeline with code (Jenkinsfile). Jenkins is also open-source, meaning it is totally free with a large community of support behind it. It is important to understand the more restrictive a technology gets, the harder it is to implement for large project. The largest downside of Github Actions and Bamboo is their reliance on GitHub and Atlassian repositories. If a users pipeline is created in Github Actions or Bamboo, they will be tied to the GitHub or Atlassion repository workspace. At large, this restriction will cause compatibility issues, problems with pipeline functionality, and restriction on pipeline workflow integration. Furthermore, Jenkins has simply been around longer, allowing the platform to graduate to a well-known and well-respected pipeline workspace [19]. One downside of Jenkins, however, is that it only supports a fan-in fan-out direct acyclic graph (DAG) execution procedure. With a fan in fan out procedural flow of stages, all stages that another stage depends on must complete before moving on in the pipeline of steps. On the other hand, full DAG allows configuration on what to wait and when (almost analogous to asynchronous programming), which in turn saves time and resources. Regardless of this downside, Jenkins is still an overall great choice of technology due to its balanced nature and widespread support.

7 Scripts and Scripting Languages

Scripts are bits of light-weight code written to be executed during a larger systems run time. This means instead of being compiled during run-time, scripts are interpreted during run-time [21, 28]. The line between "real" programming and scripting is a blur since logically the two do not differ from on another. Two main categories of programming languages is worth noting: compiled and interpreted languages. Compiled languages are translated into "computer language" by the target machine, while interpreted languages are read line-by-line and executed on-the-fly. It is important to note some compiled languages can serve as interpreted languages and vice versa. Python is an example of such a language. To simplify the understanding of scripts in this thesis's context, scripts are used to automate repetitive task within a system that would otherwise consume vast amounts of time if done manually. Python and Windows Batch scripting are two scripting languages are commonly used. In the various Jenkins pipeline stages scripts serve to execute particular executables at a certain times, automate various on-the-fly tasks, ensure proper naming conventions, and so on.

7.1 Bourne-Again SHell

Bourne-Again SHell (BASH) is the native command interpreter on Linux systems [10]. It is most commonly used to automated system tasks. If one is familiar with Linux terminals (similar to Windows Command Prompt), then they know how tedious certain installations, configurations, and file handling can get within the terminal. Each line of commands does one thing and the syntax is of utmost importance. BASH scripts use these terminal commands along with system level functions to dynamically automate certain tasks. This does not mean that a full-blown application can be developed, but many system properties can be tracked and used to make using a computer a bit simpler. For example, if there was a computer that does not automatically connect to the internet when turned on, assuming the operating system does not provide the option to automatically connect, a BASH script could be written to carry out such function. In the long run it would save lots of time and allow the user to worry about less when first logging onto the computer.

7.2 Windows Batch Scripting

Windows Batch scripting enables command line instructions to be executed line-by-line as text files [7]. Windows Batch serves the same purpose as BASH does except on a Windows operating system. This is a great way to automate many tasks on a machine. Though Batch is old and was developed for Microsoft Disk Operating System (DOS), it is still currently in use. Batch files are usually saved with a ".bat" extension and are executed line-by-line just like an ".exe" file. Batch files are widely used in Jenkins pipelines because many files are being fetched, compiled, and moved around when a pipeline is working on a project, making it a good tool to automate tasks.

7.3 PowerShell

PowerShell is a more advanced version of Windows Batch. One huge difference between PowerShell and Windows Batch is that PowerShell treats shell outputs as object, while Windows Batch treats them as plain text or strings [10]. This becomes important as scripts become more dynamic since PowerShell is able to forward these object outputs onto anther "command". Otherwise all shell flavors are capable scripting languages.

7.4 Python

In modern scripting, Python is to go-to language for most script programmers [34]. The libraries available in Python are essentially endless. Python can be paired with many third party tools making it very compatible with many technologies, including automated development. Tools like file reading, deleting, editing, and other manually intensive tasks can be automated by Python. The simple-to-understand syntax also makes it easy to learn and use cross-platform (Windows, Linux, UNIX, Mac OS).

7.5 Scripting Languages Used

The scripting languages used in the developed Jenkins pipelines are Batch and Python. Some pipeline tasks may need to call certain compilers on the software, test software syntax, and even read, write, move, and delete files. Python is great due to all of its available libraries, and Windows Batch is great due to its low level nature. Although PowerShell is far more capable than Windows Batch, it is not needed since Python trumps both. With the Python Windows Batch combination, there is a simpler scripting language and a very advance and object oriented scripting language enabling the possibility for all types of scripts. Furthermore, Python and Windows Batch tend to have far greater documentation when compared to the others resulting in faster and easier scripts.

8 Software Testing Methods

Software testing is used to find software bugs, errors, and defects in developed software [2]. Finding software issues before the official release is a must, since vulnerabilities can be exploited. Undetected defects in software can have catastrophic effects, whether that is financial or physical. By running different types of tests targeting different layers of software operation, software vulnerabilities can be eliminated assuring the highest quality of software. Furthermore, pairing the different testing methods with different software development methods in Section 5, results in highly comprehensive and effective processes. There are four main types of testing strategies: unit testing, integration testing, acceptance testing, and system testing. In this thesis acceptance testing and system testing will be grouped into one testing strategy called smoke testing.

8.1 Unit Testing

Unit testing is used to test the smallest testable software components [2]. The smallest building blocks of software are the physical lines of code a developer writes, hence unit testing is usually carried out by a programmer who understands the programming language to the core. The programmer must make sure that the edge cases in the code are properly reacted to and handled. There cannot be any action that deviates from the proper functioning of code, even if such inputs and outputs are nearly impossible. It is important to note that unit testing is a super set of multiple specific unit testing methods like functional testing, structural testing, heuristics testing, and qualification testing. Though these specific tests may differ in technique, they achieve the same goal and ultimately test the smallest building blocks of software. Of the three main categories of tests discussed in this section, unit testing is arguably the most versatile since it is cost effective but also efficient due to its ability to test multiple modules simultaneously.

8.2 Integration Testing

Integration testing is used to reliably construct project structure while exposing interfacing errors [2]. After the individual building blocks of a project are unit tested and properly linked, they must be tested as a group to ensure their integrity when working in unison. Integration testing can be executed in a top-down or bottom-up fashion. When integration testing is preformed in a top-down fashion, the top layer of program modules are tested first. On the other hand, when integration testing is preformed in a bottom-up fashion, the most atomic program modules are tested first.

8.3 Smoke Testing

Software smoke testing is a specific test designed to to be an end-to-end initial verification of software quality [26]. Though it does not test software at a granular level, it is useful to detect fatal failures in the software before granular testing. Using smoke tests saves significant amounts of testing time because software that does not adhere to the initial test should not be sent on to be tested by more specific testing methods. Smoke testing is also known as build acceptance testing, which reveals a integral part of smoke testing; testing the buildability of new software. This is especially important when new versions of software are released. Many times new software introduces new dependencies, and new dependencies introduce previously unknown facets for failure. Without a smoke test, newly written software would be passed onto further testing without any initial build checks. As the more granular tests verify the quality of the newly written software, they eventually uncover the faults that the smoke test would have. The only exception, however, is the faults are uncovered significantly later. There are a couple reasons why this method is simply inefficient. The first being that the granular tests will only realize these failures in the software when it is sent to them to test. All of the software written after the unchecked software was added to the project is now reliant on software that does not pass initial build tests. Given the development periods length, such an issue can potentially shift a projects estimated time of delivery multiple weeks behind the promised deadline. The second reason as to why such a process is inefficient is because the granular tests should only test software that has been proven worthy enough to be tested open; that is software that passes initial build tests.

8.3.1 Smoke Testing and Continuous Integration

Smoke testing would become quite tedious if it had to be run manually over and over again. Since it is a baseline test, it has to run quite frequently. Furthermore, smoke testing a days worth of changes on software would be less efficient as smoke testing software on-demand as soon as it is committed. For this reason smoke tests are best if run as the development team makes changes, making it the perfect test to preform in a continuous integration workflow. Since these continuous integration workflows are run on designated servers with limited resources, it is important to utilize every bit of server resource maximally. So that resource is not wasted on testing bad software, the smoke test is integrated into the very beginning of continuous integration to ensure efficient usage of resources.

9 Project Scope

The specific project to which the practical work belongs to is an Audi project named PNG-ELVIS. The focus of this project is to develop a reliable framework for a network of sensors, ECUs, and communication channels that is very robust in all situations. In newer vehicles, there are tremendous amounts of electronic components to operate, which means the vehicle has to know what to operate when. For example, it is very important to have enough power to slam on emergency brakes if the car detects a collision. Furthermore, the windshield wipers functionality in such situation is not important whatsoever. This project aims to classify each components priority in an advanced system like Audi's, and create the most reliable system.

Within the automotive embedded software team at Robert Bosch Kft., there are several teams including but not limited to software developers, testers, system engineers, hardware architects, and DevOps engineers. Continuous integration pipelines and process automation belong to the DevOps sector of the embedded software wing. Though the software developers themselves focus on developing software, they are also heavily involved with the DevOps team since they must react accordingly to the feedback that is produced by DevOps. Since IBM ALM is the VCS that is utilized, there are three main streams to define three different stages of software development in this project. These three streams are depicted in Figure 9.1 as three blue rounded rectangles (Figure 9.1 found on the next page). The instance of objects that are being actively developed reside on the development stream (denoted as DEV in Figure 9.1). The Jenkins pipeline designated to testing these actively developed objects is denoted by the green oval-like shape labeled with DEV OnCommit. This pipeline is an event-base, meaning that it is run every time a developer commits a change to the development stream. After a set of change sets pass the tests and quality checks present in the DEV OnCommit pipeline, they are added to the continuous integration stream denoted by CONT_INT in Figure 9.1.

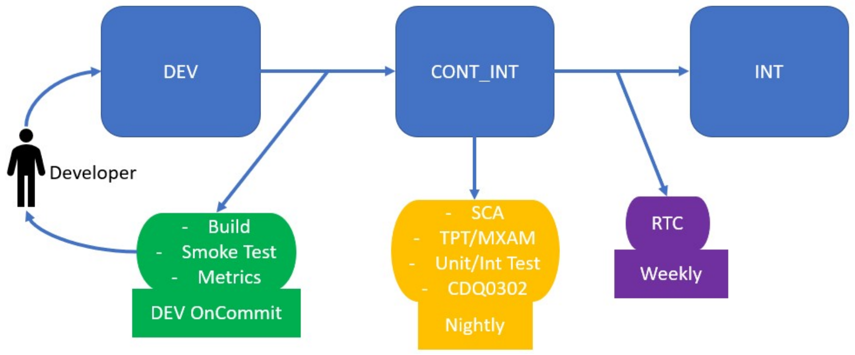


Figure 9.1: Jenkins Pipeline Development Workflow

The Jenkins pipeline that is associated with the continuous integration stream is denoted by the yellow oval-like shape labeled with Nightly. This pipeline is time-based, meaning that it is executed every night on weekdays, and includes more specific tests to further validate the quality of the software that had passed the DEV OnCommit pipeline. The final stream is the integration stream, comprised of software that has passed the DEV OnCommit pipeline and CONT_INT Nightly pipeline tests. There is one more pipeline denoted by the purple oval-like shape that is not actively used. This pipeline, once fully finished, would run on a weekly basis, making it a time-based pipeline as well. There is another pipeline call CONT_INT Internal Release, which operates on the CONT_INT stream, used to produce release packages when a final software integration happens. The CONT_INT Internal Release pipeline is not depicted in Figure 9.1 since it does not execute software tests. It does, however, serve a large role in the continuous integration process as it provides the stable release packages. The CONT_INT Internal Release pipeline is manually triggered when the need for it arises making it a manually triggered pipeline (as opposed to event-base or time-based). As the streams encompass more software, the corresponding pipelines becomes less automated. It is important to note that this workflow is up and running, however, not fully automated yet; meaning the integration is semi-automated. Tests are automatically run but the physical integration of software still happens manually. The environment to which contributions have been made throughout this thesis are the DEV OnCommit, CONT_INT Nightly, and CONT_INT Internal Release Jenkins pipelines. The physical server on which these pipeline executions reside is located in Germany. Furthermore, there are test pipelines that were developed that utilize a server in India. The Indian server is maintained by the software testing V&V team.

10 Process Automation in Jenkins

Process automation in DevOps is the elimination of human interaction from a pipeline of tasks. Such tasks may include software tests, file manipulation, or even script execution. A process can be called fully automated when there is no need to manually execute anything in regard to that process. Process automation plays a huge role in continuous integration pipeline's efficiency. Continuous integration pipelines execute smoother, experience less failures, and produce results of similar quality from consecutive executions when process automation scripts are added. It is only worth automating processes that will save time and resources in the future. Between the different stages of software development there are usually smaller tasks that must be carried out by hand simply because it is project specific. As a project grows, such tasks have to be carried out with more frequency or, perhaps, more accuracy. This results in an engineer having to spend their time making sure a certain task is properly carried out in a timely fashion. This is time potentially saved by simply automating the task itself. Furthermore, automating tasks eliminates a lot of error. If a process works well semi-automated, then it is guaranteed to work even better automated, since majority of the human involvement is eliminated. If human involvement in a pipeline of tasks is eliminated, human error is minimized, resulting in homogeneous results from one execution to another. Human error, however, is never fully eliminated because the automation scripts are developed by humans and the software being tested is also written by humans. It is safe to say that process automation applied in the right situations can have a massive impact on the pipeline efficiency and efficacy as a whole. In the work environment described in Section 9 there is plenty of room for process automation in multiple areas of the different pipelines themselves. It is also important to note that many times processes are not be automated due to the extensive amount of time and effort it may take to set up an automated version of such process. For this reason, it is important to evaluate the benefits and drawbacks of automating given processes so that a reasonable conclusion can be reached.

10.0.1 Current Internal Release Folder Content Deletion

On the development stream of the project, there is a directory in which all software internal releases are stored. From the development stream, they are moved to the continuous integration stream where they are used to run certain metrics and analytics. Internal releases are generated by a series of scripts when software development has reach a new stable stage. The stability of the release is then tested and the corresponding metrics analyzed to make sure that the developed software adheres to specified standards. The series of scripts that produce these files run on a remote server and thus, these files have to be uploaded to an artifactory. In this case JFrogs Continuous Integration (CI) Artifactory is used and serves as a cloud storage for the files produced by the CONT_INT Internal Release pipeline. Figure 10.1 shows how the artifactory serves as a storage between the remote server running the series of scripts and the developers who analyze the results (Figure 10.1 found on the next page). The files uploaded include a build output zip file, release zip file, and two other metrics related zip files. The problem before process automating scripts were added was the release zip file that was generated included all of the previous releases alongside the current release. There is absolutely no reason to include the previous releases in this zip file because they are not relevant to the current analysis.

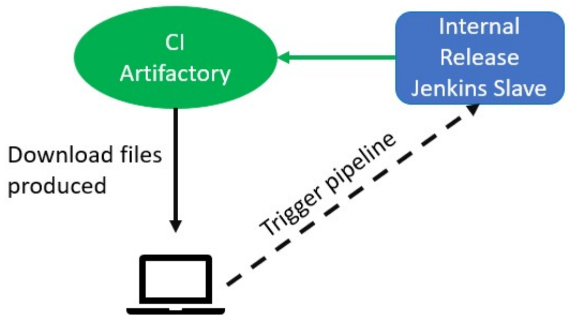


Figure 10.1: CONT_INT Internal Release Pipeline Remote Server Artifactory Relationship

Appendix A presents the script that was integrated into the CONT_INT Internal Release pipeline to automate the deletion of all unused release directories from the release zip file. Since paths are dynamic the script has to properly function without hard coding any paths or variables into it. The dynamic workspace is taken as the base path, and then the directory in which all previous releases are stored is specified. In the early stages of script development the pipeline exited with an error noting the deletion had failed. It turns out there is a maximal path length that can be fed in as a Batch command to delete a file. The solution was to link the release folder path to a drive letter to shorten the working path of all the directories being deleted. A for loop then loops over every existing folder and file in the directory linked to the drive letter and deletes everything. The linked drive is then unlinked to avoid future drive linking issues on the Jenkins server. The release directory on the remote server now strictly contains the current release folder. It is important to note that Jenkins only cleared this folder locally during this execution, meaning the deleted files and folders are not deleted from the stream itself. The script is called in the build stage of the CONT_INT Internal Release continuous integration pipeline:

```
dir('JWS/SW/ToolsAndConfiguration/CICD/scripts')
        {
            bat 'Cont_Int_Internal_Release.bat'
        }
```

Manually deleting the unnecessary release directories from the zip file is not an option because the process has to be executed within the pipeline. Furthermore, even if manual deletion was an option, it would be very time consuming to manually prepare the release directory for every single pipeline execution.

10.0.2 Fetching Software Version Based

On the CONT_INT Internal Release pipeline there was another process to automate. The directory inside the release zip file follows a specific naming convention, more specifically, the directory inside the release zip had to be named after the most recent stable software release. It is also important to be able to get this information dynamically so that the problem of versions not matching is eliminated. Though it seems quite trivial to update this information with a new stable software release, it is often forgotten, which then leads to software incompatibility issues producing errors in the pipeline. The software version based information was also integral to the CONT_INT Internal Release pipeline because the correct information has to be displayed within the pipeline process execution notification. The software developers involved in a release are notified if the pipeline fails, and the software version information has to be correctly forwarded so that a fix can be issued immediately. Appendix B presents the script written to allow the CONT_INT Internal Release pipeline to dynamically pick out the current software version information to use. The values acquired by the version based script are passed onto the pipeline and used in the build stage:

```
dir('JWS/SW')
    {
        env.simpleVersion = readFile(file: 'simple.txt')
        bat "del simple.txt"
        env.releaseVersion = readFile(file: 'release.txt')
        bat "del release.txt"
    }

dir('JWS/SW/ToolsAndConfiguration/CICD/scripts')
    {
        bat 'Get_Version_Based.bat'

        env.versionBased = readFile(file: 'version_based.txt')
        bat "del version_based.txt"
        env.almQuery = readFile(file: 'alm.txt')
        bat "del alm.txt"
    }
```

10.0.3 Final Thoughts on Process Automation

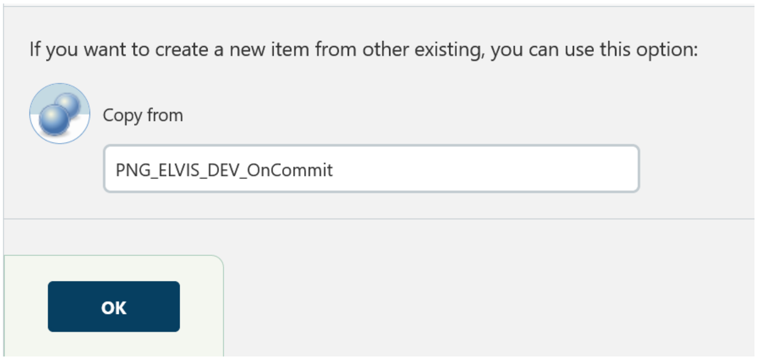
Though the processes presented in the previous two sections seem like petty processes to automate, a significant workload was lifted off the teams shoulders. With regards to deleting the unnecessary files the release zip file, significant amounts of space continue to be saved. The script was put into the live pipeline and is consistently saving 5,000 KBs of space every release. The software version based script was also integrated into the live pipeline, and the release version no longer has to be manually tracked.

11 Continuous Integration with Jenkins

Continuous integration is vital to process automation since process automation loses its value without continuous integration pipelines and vice versa. Setting up, configuring, and developing pipelines to execute the proper steps in a given amount of time is integral to a successful continuous integration environment. Process automation is integrated into the pipelines at certain points to make the entire pipeline execution process smooth and precise. To fully tie together all moving parts of this thesis, the different pipeline configurations are explained in the following subsections.

11.1 DEV OnCommit Test Pipeline configuration

The DEV OnCommit and Nighty pipelines presented in Figure 9.1 are live pipelines utilized on a daily basis. To avoid issues on live pipelines throughout the project, a DEV OnCommit Test pipeline is set up using a direct copy of the DEV OnCommit live pipeline (seen in Figure 11.1). Figure 11.2 shows how the test pipeline was named.

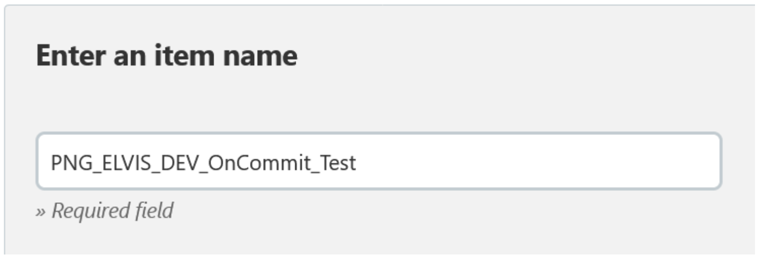


If you want to create a new item from other existing, you can use this option:

Copy from

PNG_ELVIS_DEV_OnCommit

Figure 11.1: DEV OnCommit Test Pipeline Copying



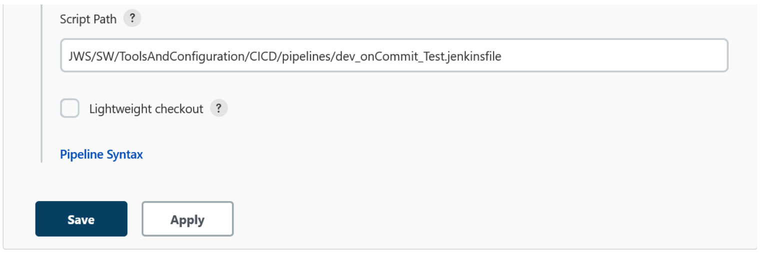
Enter an item name

PNG_ELVIS_DEV_OnCommit_Test

» Required field

Figure 11.2: DEV OnCommit Test Pipeline Naming

After the test pipeline was created, it had to be configured so that it uses the correct Jenkinsfile and environment variables for proper stage execution and pipeline processes. A newly created Jenkinsfile named dev_oncommit_Test.Jenkinsfile (Figure 11.3) is created so that the testing environment is completely separate from the live DEV OnCommit pipeline.


Script Path

JWS/SW/ToolsAndConfiguration/CICD/pipelines/dev_onCommit_Test.jenkinsfile

Lightweight checkout

Figure 11.3: DEV OnCommit Test Pipeline Assigning Jenkinsfile

Furthermore, the environment variables, stream and workspace properties, and other run time variables are set in the Jenkins pipeline configuration menu to make the DEV OnCommit Test pipeline fully functional.

```
rtcWorkspace=dev_workspace
rtcModuleIntStream=cont_int_stream
rtcCredentials=User_Credentials
loadRulePath=dev_workspace.loadrule
timeZone=Europa/Berlin
artifactoryKey=******

projectArtRepo=local_project_repo
projectArtCreds=User_Credentials
projectArtApi=******
pythonEnv=python_env
```

11.1.1 Static Code Analysis

Static Code Analysis (SCA) is automated process for checking embedded software for bugs [29]. The automated process includes internationally accepted guidelines that microcontroller embedded software must adhere to, to ensure human safety. The two main sets of guidelines used by the SCA script in this project are the Motor Industry Software Reliability Association (MISRA) test and Hersteller Initiative Software (HIS) test. SCA can be configured to use different rules to provide different metrics. In this specific example, SCA is used to test and check all components of software by analyzing the make files after building the target software. Figure 11.4 shows the SCA stage that was added to the DEV OnCommit Test pipeline. SCA can be broken down into smaller processes called Single File Analysis (SFA) where only the make files affected by a committed change are analyzed. SFA is significantly more efficient when compared to SCA since it builds and analyzes less make files. SFA is comprised of four main steps, the first of which is building all software that was subject to change by the development team. Building the software results in new object files. The second step is the production of the make files. The newly built make files are then analyzed by a script that runs through a series of tests, completing the third step in the SFA process. Once the analysis is complete, the fourth step is to generate and upload a report of the analysis to a dashboard, where the report metrics can be viewed on a web interface. Here the metrics are checked and filtered by individuals to either pass or fail a given commit batch. If the analysis as a whole fails, the changes are not accepted into the continuous integration stream. The last step in the SFA sequence is the only manual part of the SFA process.

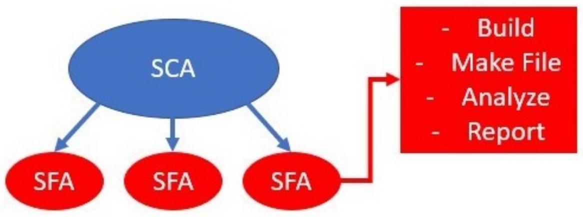


Figure 11.4: Static Code Analysis

11.1.2 Automating Single File Analysis Metrics Evaluation

To fully automate the SFA process, two scripts had to be written to evaluate the metrics produced by the analysis. The SFA stage implementation, that is, the stages added to the DEV OnCommit Test pipeline are presented in Appendix C. The scripts used to automate the SFA process are presented in Appendix D and Appendix E. The scripts are called from a new stage in the Jenkins DEV OnCommit Test pipeline. In the beginning of the Jenkinsfile, the local variables are set. To conduct a proper SFA, all current object files are packed into a zip file named make_file_objects.zip. SFA is then executed and the object files that include changes are checked for software and metric violations. The metrics reports are usually presented on the Axivion Dashboard, however, it is possible to reach the dashboard metrics through an application programming interface (API) call, which provides the metrics in JSON format. Since this process needs to run after the analysis in the DEV OnCommit Test pipeline, it was easiest to call a batch script for this step. After setting some variables in this batch script, it passes these variables to a Python script. Python is used to make the API request and parse the JSON because it has readily available libraries that make both processes simple. The Python script receives the arguments from the Batch script and loads them into local variables. In a try catch block, the script makes an API request to the proper URL and saves the response given a successful request.

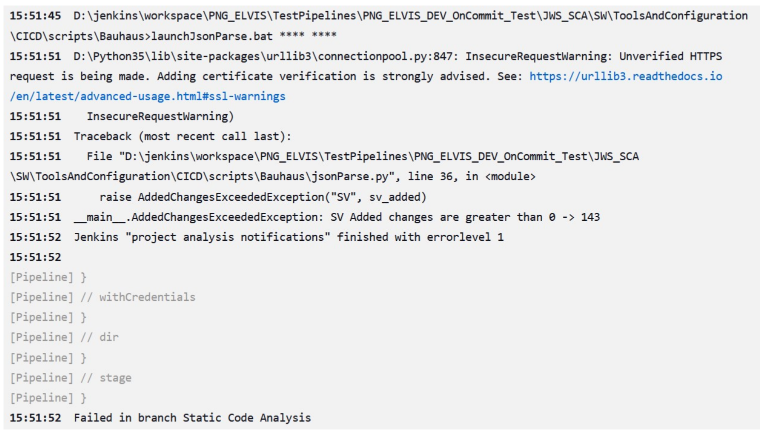
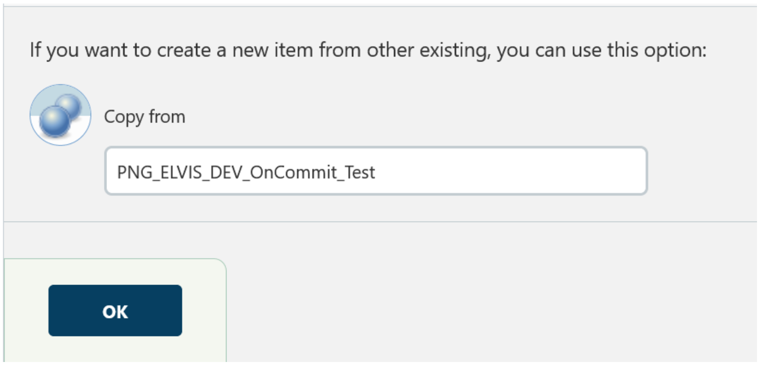

```
15:51:45  D:\jenkins\workspace\PNG_ELVIS\TestPipelines\PNG_ELVIS_DEV_OnCommit_Test\JWS_SCA\SW\ToolsAndConfiguration
\CICD\scripts\Bauhaus>launchJsonParse.bat **** ****
15:51:51  D:\Python35\lib\site-packages\urllib3\connectionpool.py:847: InsecureRequestWarning: Unverified HTTPS
request is being made. Adding certificate verification is strongly advised. See: https://urllib3.readthedocs.io
/en/latest/advanced-usage.html#ssl-warnings
15:51:51    InsecureRequestWarning)
15:51:51  Traceback (most recent call last):
15:51:51    File "D:\jenkins\workspace\PNG_ELVIS\TestPipelines\PNG_ELVIS_DEV_OnCommit_Test\JWS_SCA
\SW\ToolsAndConfiguration\CICD\scripts\Bauhaus\jsonParse.py", line 36, in <module>
15:51:51      raise AddedChangesExceededException("SV", sv_added)
15:51:51  __main__.AddedChangesExceededException: SV Added changes are greater than 0 -> 143
15:51:52  Jenkins "project analysis notifications" finished with errorlevel 1
15:51:52
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // dir
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
15:51:52  Failed in branch Static Code Analysis
```

Figure 11.5: Automated Single File Analysis Console Output

If the request is not successful then the script raises a general exception causing the batch script to exit with an error code of -1. When the script exits with an error code of -1, the batch script as a whole fails causing the pipeline execution to also fail. If, however, the API request and JSON parsing do not throw errors, the Python script pulls important metrics values from the response JSON and stores them in local variables. After the try catch block, the saved metric values are then evaluated by if statements. Of course, if the conditional statements are not satisfied the script execution comes to an end and the pipeline execution as a whole continues, meaning the metrics analysis comes back as positive. If, however, the if statements are satisfied, a custom exception is thrown. This custom exception is designed to print the information that caused the failure to the pipeline console while still causing the main batch script to exit with an error code of -1. Figure 11.5 shows an instance where the SFA metrics triggered an error, causing the pipeline execution to fail. It is important for the main script to exit with a error code of -1 because this is the error code that fails the pipeline execution in its entirety. This way further tests are not run on software that did not pass an initial metrics analysis test. It is important to bare in mind the single file analysis script itself was written by the testing team in India.

11.2 Smoke Test Pipeline Configuration

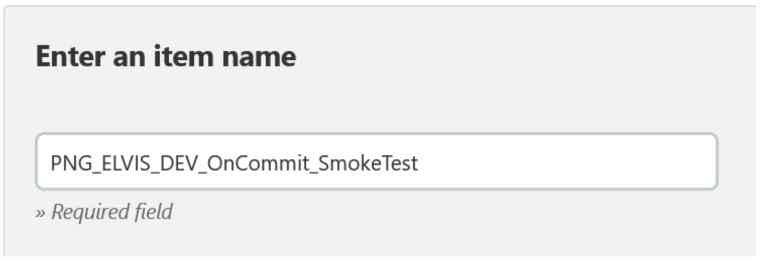
The first step to configuring the Smoke Test pipeline is to create the pipeline. Instead of creating a pipeline from scratch, the existing DEV OnCommit Test pipeline is used as a template for the Smoke Test pipeline. This made the configuration process quicker since the specific authorization and authentication tokens did not have to be manually added. The DEV OnCommit Test pipeline is used as a template (seen in Figure 11.6) and the Smoke Test pipeline is given a name (seen in Figure 11.7).


If you want to create a new item from other existing, you can use this option:

Copy from

PNG_ELVIS_DEV_OnCommit_Test

Figure 11.6: DEV OnCommit Test Pipeline Used as Template

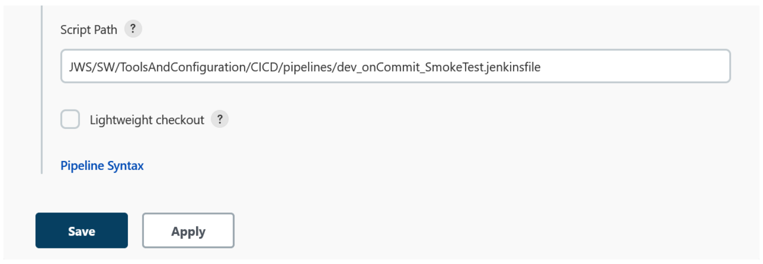

Enter an item name

PNG_ELVIS_DEV_OnCommit_SmokeTest

» Required field

Figure 11.7: Smoke Test Pipeline Naming

The proper Jenkinsfile is assigned to the newly create pipeline (Figure 11.8).


Script Path

```
JWS/SW/ToolsAndConfiguration/CICD/pipelines/dev_onCommit_SmokeTest.jenkinsfile
```

Lightweight checkout

Figure 11.8: Smoke Test Pipeline Assigning Jenkinsfile

Once the Smoke Test pipeline is ready, the following environment variables are added to the configuration to enable proper pipeline functioning.

```
rtcWorkspace=dev_workspace
rtcModuleIntStream=cont_int_stream
rtcCredentials=User_Credentials
loadRulePath=dev_workspace.loadrule
timeZone=Europa/Berlin
artifactoryKey=*****

projectArtRepo=local_project_repo
projectArtCreds=User_Credentials
projectArtApi=*****
pythonEnv=python_env
```

11.2.1 Separate Smoke Test Pipeline and Integrated Smoke Test Pipeline

Since the custom Jenkinsfile only included the smoke test stages, the configuration steps described above resulted in a pipeline strictly running the smoke test. A standalone setup was perfect for testing the functionality itself, however, it is not efficient when it comes to utilizing the smoke test. To fully streamline the test, it was integrated into the DEV OnCommit Test pipeline. Instead of just building the software and running the metrics on the DEV OnCommit Test pipeline, the smoke test was now added after the build stage. The Smoke Test pipeline alone requires a build anyway so it would be a waste of time to run the DEV OnCommit Test pipeline and Smoke Test pipeline serially. It is important to note that the smoke test stages were added to the live DEV OnCommit pipeline for a week. The stages were disabled after a week due to constant pipeline failure. More information on this integration in future works and application section (Section 13).

11.2.2 Smoke Test Integration into DEV OnCommit Test Pipeline

Smoke testing plays an integral part in continuous integration since it makes the development stream significantly more efficient in detecting problems in software at an early stage. Seen in Figure 9.1, smoke testing takes place after the build on the DEV OnCommit Test pipeline. Understanding the specifics of the smoke test is key to its integration into the DEV OnCommit Test pipeline. Figure 11.9 shows the architecture of the smoke test system. To start the smoke test, the PNG-ELVIS Jenkins slave residing in Germany builds all of the contributed software upon a commit.

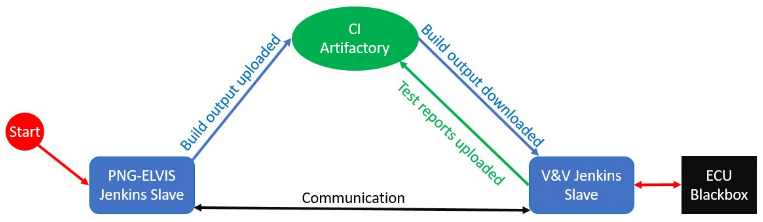
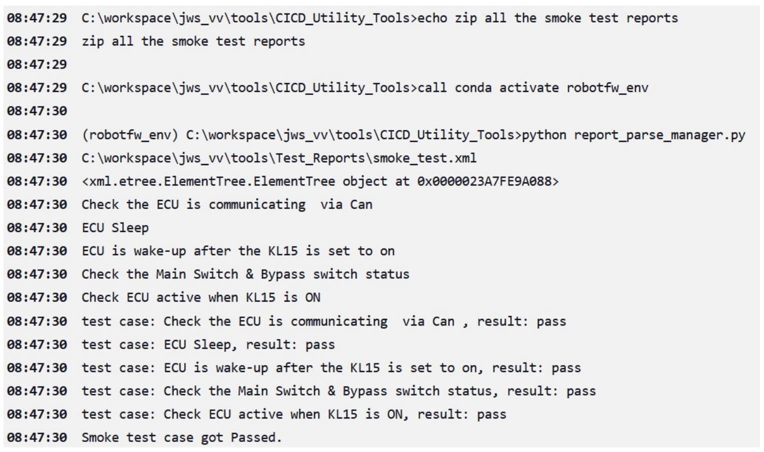


Figure 11.9: Jenkins Smoke Test System Architecture

Once the build is complete, the build results are uploaded to the CI Artifactory. Once the upload is complete, the PNG-ELVIS Jenkins slave notifies the V&V Jenkins slave about the successful upload. The V&V Jenkins slave is then triggered to download the build results and flash them onto the physically connected ECU. After the flash takes place, the ECU and Jenkins slave communicate with each other about the practical validity and effectiveness of the new software on the physical ECU. This is the smoke test. Once the tests and checks are complete on the V&V Jenkins slave and the ECU, the results are uploaded to the CI Artifactory. It is important to note that the PNG-ELVIS Jenkins slave does not terminate its pipeline process as the smoke test takes place, rather it idles and waits for the V&V Jenkins slave to finish smoke testing and uploading the results. At last, the PNG-ELVIS Jenkins slave processes the smoke test results and produces a pass or fail result. Figure 11.10 shows the console output produced by a passed smoke test (Figure 11.10 found on the next page). The five passed test cases that the V&V Jenkins slave tests on the ECU blackbox can be seen at the bottom of the console log. With the smoke test step, it is possible to practically test the newly developed software as it is committed. This is immensely powerful since the software is tested in small bits.



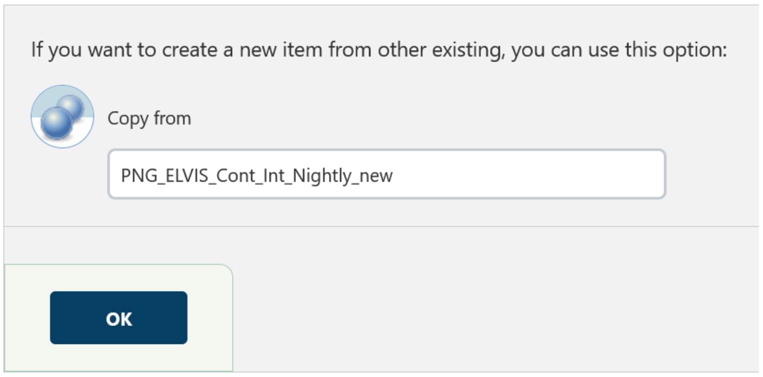
```
08:47:29  C:\workspace\jws_vv\tools\CICD_Utility_Tools>echo zip all the smoke test reports
08:47:29  zip all the smoke test reports
08:47:29
08:47:29  C:\workspace\jws_vv\tools\CICD_Utility_Tools>call conda activate robotfw_env
08:47:30
08:47:30  (robotfw_env) C:\workspace\jws_vv\tools\CICD_Utility_Tools>python report_parse_manager.py
08:47:30  C:\workspace\jws_vv\tools\Test_Reports\smoke_test.xml
08:47:30  <xml.etree.ElementTree.ElementTree object at 0x0000023A7FE9A088>
08:47:30  Check the ECU is communicating  via Can
08:47:30  ECU Sleep
08:47:30  ECU is wake-up after the KL15 is set to on
08:47:30  Check the Main Switch & Bypass switch status
08:47:30  Check ECU active when KL15 is ON
08:47:30  test case: Check the ECU is communicating  via Can , result: pass
08:47:30  test case: ECU Sleep, result: pass
08:47:30  test case: ECU is wake-up after the KL15 is set to on, result: pass
08:47:30  test case: Check the Main Switch & Bypass switch status, result: pass
08:47:30  test case: Check ECU active when KL15 is ON, result: pass
08:47:30  Smoke test case got Passed.
```

Figure 11.10: Console Output of a Passed Smoke Test

The ECU and V&V Jenkins slave is configured and managed by the V&V team in India. The smoke test scripts are written on that end to ensure proper smoke test functionality. Once the smoke test script was written, however, it had to be integrated into the DEV OnCommit Test pipeline. The DEV OnCommit Test Jenkinsfile with the smoke test integrated is presented in Appendix F.

11.3 Qualification Test Pipeline Configuration

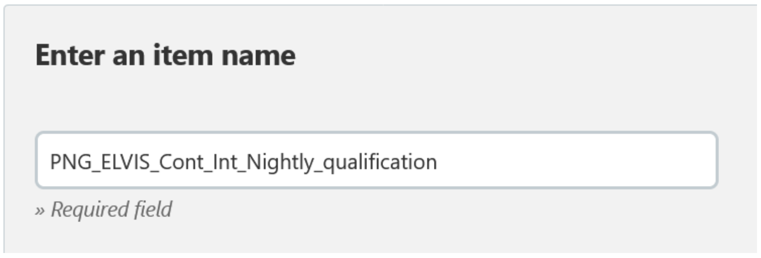
The qualification test is a type of unit test that is used to measure the integrity of developed software in the context of production requirements. Before software can be officially released, it must pass the qualification tests to confirm the software adheres to production standards. Configuring the qualification test pipeline was very similar to configuring the smoke test pipeline. Instead of copying the DEV OnCommit Test pipeline as a template, the CONT_INT Nightly pipeline was copied, since it runs unit tests (Figure 11.11). The pipeline was then named (Figure 11.12) and configured to use a unique Jenkinsfile (Figure 11.13).


If you want to create a new item from other existing, you can use this option:

Copy from

PNG_ELVIS_Cont_Int_Nightly_new

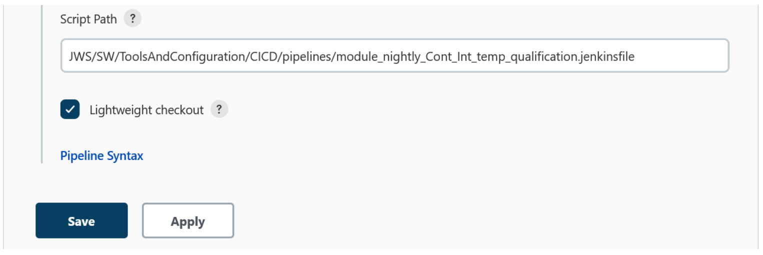
Figure 11.11: Nightly Pipeline Used as Template


Enter an item name

PNG_ELVIS_Cont_Int_Nightly_qualification

» Required field

Figure 11.12: Qualification Test Pipeline Naming


Script Path

JWS/SW/ToolsAndConfiguration/CICD/pipelines/module_nightly_Cont_Int_temp_qualification.jenkinsfile

✓ Lightweight checkout

Figure 11.13: Qualification Test Pipeline Assigning Jenkinsfile

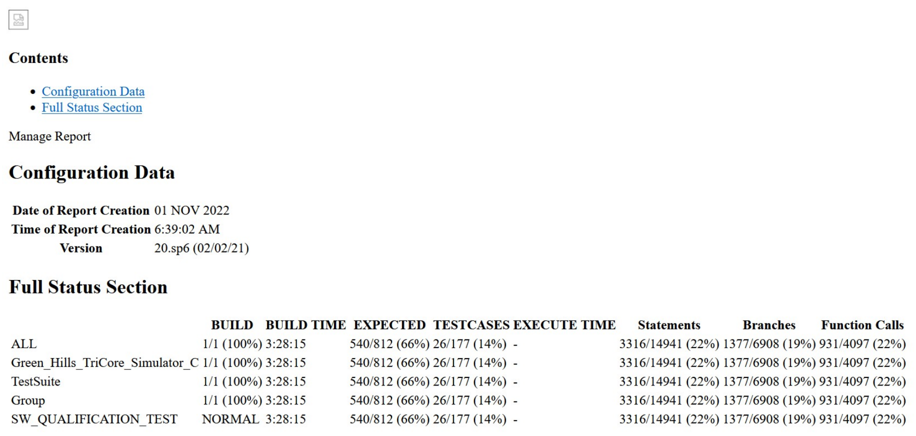
After the pipeline was successfully created, the following environment variables are added to the configuration to enable proper pipeline functioning.

```
buildNode=project_build_node

rtcStream=dev_stream
```

```
rtcWorkspace=dev_workspace
rtcCredentials=User_Credentials
loadRulePath=dev_workspace.loadrule

timeZone=Europa/Berlin

projectCode=*****
fossidCreds=*****
artifactoryKey=*****

projectArtRepo=local_project_repo
projectArtCreds=User_Credentials
projectArtApi=*****
pythonEnv=python_env
```

11.3.1 Standalone Qualification Test

The qualification test pipeline is configured to only run the qualification test since the tests themselves run independent of any other process. Since it is a unit test, it would be best to group it together with the CONT_INT Nightly unit tests, however, there are strict execution time limits under which the CONT_INT Nightly pipeline must run. The qualification test took too much execution time and thus was not added as a parallel stage to the CONT_INT Nightly pipeline. Integration of the qualification test into the CONT_INT Nightly pipeline would be done by simply adding the setup, test, and teardown stages into the CONT_INT Nightly pipeline and adding it to the parallel execution property. The standalone qualification test pipeline Jenkinsfile can be found in Appendix G. Figure 11.14 presents a report generate by the Qualification Test pipeline. This particular report shows that 540 of the 812 test cases were passed, resulting in a 66% passing rate. All of the report information is then analyzed by individuals to ensure proper qualification test functionality.



Contents

- Configuration Data
- Full Status Section

Manage Report

Configuration Data

Date of Report Creation	01 NOV 2022
Time of Report Creation	6:39:02 AM
Version	20.sp6 (02/02/21)

Full Status Section

	BUILD	BUILD TIME	EXPECTED	TESTCASES	EXECUTE TIME	Statements	Branches	Function Calls
ALL	1/1 (100%)	3:28:15	540/812 (66%)	26/177 (14%)	-	3316/14941 (22%)	1377/6908 (19%)	931/4097 (22%)
Green_Hills_TriCore_Simulator_C	1/1 (100%)	3:28:15	540/812 (66%)	26/177 (14%)	-	3316/14941 (22%)	1377/6908 (19%)	931/4097 (22%)
TestSuite	1/1 (100%)	3:28:15	540/812 (66%)	26/177 (14%)	-	3316/14941 (22%)	1377/6908 (19%)	931/4097 (22%)
Group	1/1 (100%)	3:28:15	540/812 (66%)	26/177 (14%)	-	3316/14941 (22%)	1377/6908 (19%)	931/4097 (22%)
SW_QUALIFICATION_TEST	NORMAL	3:28:15	540/812 (66%)	26/177 (14%)	-	3316/14941 (22%)	1377/6908 (19%)	931/4097 (22%)

Figure 11.14: Full Report Produced by Qualification Test Pipeline

11.4 Final Thoughts on Continuous Integration

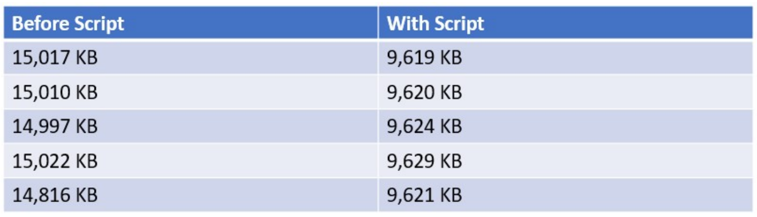
The three test pipelines created could easily be integrated into the live DEV OnCommit and CONT_INT Nighty pipeline. The smoke test and SFA stages would need to be copied over to the DEV OnCommit pipeline Jenkinsfile and added to the stage execution list. Adding the qualification test stages to the CONT_INT Nightly pipeline is just as easy. If the three test pipelines functionalities are implemented in the live environment, it would greatly increase the efficacy of the pipelines. Firstly, the DEV OnCommit pipeline would have an extra smoke test layer capable of noticing minute faults in software quality. Furthermore, the DEV OnCommit pipeline would be able to pass and fail specific commits since the fully automated SFA would enable it to weigh the metrics of all added software. Finally, the CONT_INT Nightly pipeline would have the qualification test results ready at the end of each pipeline run, ensuring the passed software is ready for production. Enabling these three advancements in the live pipelines would result in tested, quality, and safe automotive embedded software.

12 Measurements

Between the major topics of this thesis, process automation and continuous integration, concrete measurements were not trivial to collect. Both the test and live pipelines are run on a remote server to which I had no physical access to. Furthermore, the continuous integration portions of the project were not integrated into the live pipelines due to PNG ELVIS project specific limiting factors. Regardless of these limitations, I was able to collect quantitative results from one of the process automation scripts alongside some abstract but undeniable evidence of efficacy of both the scripts and continuous integration.

12.1 Process Automation Script Measurements

From the process automation portion of the thesis, the deletion of the release folder content script made quite an impact on the size of the result zip file stored. Figure 12.1 shows five successive release zip file sizes before and after the release directory script is added to the pipeline. Though a 5,000 KB difference does not seem like much, it takes up plenty of space on the CI Artifactory when there are multiple hundreds of release zip files stored there. Furthermore, the package upload and download speed is also decreased since the Jenkins server needs to move a smaller file size.



Before Script	With Script
15,017 KB	9,619 KB
15,010 KB	9,620 KB
14,997 KB	9,624 KB
15,022 KB	9,629 KB
14,816 KB	9,621 KB

Figure 12.1: Comparing Release Zip File Sizes Before and After Script

Considering the abstract yet undeniably effective aspects into consideration within the scope of this script, it is clear the script eliminates tremendous amounts of human working hours when considering the time it would take a colleague to delete the unnecessary release directories before every execution. Not to mention, human error will inevitably occur when manually deleting the directories. Considering the quantitative results, observations, and the amount of time it took to develop the script, the outcome is undoubtedly positive. The automatic fetching of software version based script also offers similar benefits with the exception of the quantitative data.

12.2 Continuous Integration Measurements

Though the work on the continuous integration pipelines was not put into full production, it did produce positive results. One of the most influential parts of the continuous integration work was the DEV OnCommit Test pipeline on which the single file analysis (SFA) script automatically executed on. Automating the SFA process greatly improves the development workflow and software quality. If SFA is not executed at small developmental commits, the potential for further issues is greatly increased. It is an even greater complication if flawed software becomes the foundation of newly developed software. Unfortunately it is complicated to quantify the positive results the automated SFA introduces into continuous integration, but the positives are undeniable. The smoke test and qualification tests make up the rest of the continuous integration work presented throughout the thesis. Likewise, both of these tests filter out possible software flaws that may otherwise be overlooked. The unique quality of the smoke test makes it possible to run every commit, making it a massive asset in continuous integration. The qualification test, on the other hand, generates a huge report of tests that passed and failed for given units of software. The positive effects of a smoke test and qualification test are clearly present, however, providing quantitative data proving this claim is not so simple.

13 Application and Future Works

Throughout this thesis work, as time passed, the avenues of improvement became increasingly evident. The first being the main development workflow around which the pipelines where designed. The V-Model development workflow is not fit to be used alongside continuous integration pipelines because it limits the continuous integration process. Continuous integration pipelines are designed to be dynamic, elastic, and future oriented software development tools and the software development workflow should compliment these features. The V-Model forced a pipeline to be constricted to one area of testing. To enhance the continuous integration pipelines, a Git-based development workflow should be used like Agile or Scrum. In a Git-based workflow, commit prechecks can be run to ensure only quality compliant software even gets to the continuous integration pipelines for further evaluation. The continuous integration pipelines can now scale vertically and horizontally, not limiting one pipeline to a given type of test or a series of a given type of test. The possibility to test per ASIL or QM class becomes possible with a Git-based workflow. Vehicles have become very complex electronically and the development workflow to uphold such advancements must reflect the possible capabilities of such systems.

The second improvement would be the use of a single scripting language instead of the plethora available. When the process automation scripts were developed, it wasn't clear what language to use. Though the freedom of using any scripting language available to the execution environment was great, it did not offer a uniform convention. Furthermore, every scripted process has a different flow and executes using different libraries. Using a single and modern scripting language would greatly reduce the scripting complexity and development time. Not to mention it makes the project cleaner and more organized. In this project a combination of Batch and Python was used to write the scripts. The Batch scripts took significantly more time to develop when compared to their Python counterparts. Batch documentation is not nearly as vast as Pythons and considering legibility, Batch scripting is harder to understand and debug. Python scripts were easy to develop mainly because of the vast documentation publicly available. Python is also an interpreted high-level language supporting programming conventions like object oriented programming so it was easy to include custom error handling alongside libraries that parse JSON objects. Error handling and parsing JSON objects would have been hard to implement with Batch.

The third improvement worth mentioning is the failure of the smoke test in the live DEV OnCommit live pipeline. Once the smoke test was integrated into the live pipeline, it worked very well for a week. The smoke test stage execution time stayed at a healthy level and it was consistently passing and failing as expected. After a week of functioning, a new software release was added to the system and the smoke test started failing for unexpected reasons. The live DEV OnCommit pipeline failed for about a week, stopping all the committed changes on the DEV stream from being integrated into the CONT_INT stream. The V&V team was unable to fix the smoke test script and so it had to be removed from the live pipeline. A possible solution to this issue would have been to set the importance level of the smoke test below the need to integrate committed changes from the DEV stream to the CONT_INT stream. Given the right amount of time, such a solution would greatly improve the smoke test, however, it could not be implemented because the priorities were set elsewhere.

The application on process automation and continuous integration is only bound by a project itself. The automotive industry is a great example of how continuous integration can be molded around a project environment to offer unique optimization in software development. Add process automation into the unique mold and a system is created that works around the clock with great precision. In the automotive scope, continuous integration and process automation could be used to created a very robust real-world environment used to test new functionalities quickly. Testing software in the automotive industry is very important due to the safety standards. Having a development stream, testing stream, and a production stream that are well synchronized by a series of pipelines is the best example of how powerful continuous integration can be. By adding cloud computing into the mix of such technologies the automotive software industry can become a universally synchronized infrastructure. The outcome would generate the safest vehicles with the latest capabilities at a rate otherwise unobtainable.

14 Conclusion

Throughout this thesis, continuous integration and process automation development was explored in the context of the automotive industry. Different ECUs were briefly mentioned and explained, safety classification schemes were presented, and electronic network protocols were introduced. Version control systems and deployment automation tools were discussed to help understand modern software development processes flow. It is no easy task to manage the software development for a single function on any given ECU. For that matter, handling the proper development of hundreds of ECUs while still encompassing the latest technological advances is a daunting task. To further explore the role of modern technology in current development environments, different scripting languages were evaluated considering the suitability of each in the project work. With the help of Jenkins, three test pipelines were created to enhance the software development process in a small network of automobile electronics. The practical work brought to fruition new test pipelines which were derived from preexisting pipelines. The three test pipelines executed smoke testing, qualification testing, and single file analysis, potentially enabling live pipelines to grow into truly powerful software development tools. These software development methods ensure quality software is continuously integrated into multiple streams. Process automation scripts were also written to increase pipeline autonomy. This eliminated the need for human labor on repetitive yet crucial tasks. Furthermore, the process automation scripts enabled the smooth transition of pipeline tasks from one to another. The two concepts together created a robust system in which new software was reliably and automatically integrated into preexisting software. A continuous integration model designed to evaluate ECU software, given operational constraints, holds the potential to be a powerful asset. A system designed to operate at the scale of an entire vehicle would close the gap between state-of-the-art technology and the application of it.

15 Summary in Hungarian

A gépjármű gyártási, illetve fejlesztési világa jelentősen halad előre; egyre több elektronikus vezérlőegységre (ECU – Electronic Control Unit) van szükség. Újabb gépjárművekben akár több száz elektronikus vezérlőegység is kommunikálhat párhuzamosan egy adott pillanatban. Ezek a vezérlőegységek biztosítják a vezetéshez szükséges információ helyes feldolgozását. Emiatt indirekt módon emberi életek múlhatnak a helyes működésüktől. Mivel magas a tét, a lehető legnagyobb pontossággal kell fejleszteni a beágyazott szoftvereket, amik a vezérlőegységekre kerülnek. Az állandó integráció segíti a beágyazott szoftverfejlesztés folyamatát és biztosítja a szoftver elvárt minőségét. Bár egyszerű feladatnak tűnik egy ilyen rendszert létrehozni, mégsem triviális. Az egész folyamat hatékonyságát szem előtt tartva a manuális integráció helyett az automatizálás nyújt valós megoldást. Ez sokkal dinamikusabbá teszi a beágyazott szoftver integrálását, illetve kialakít egy fejlesztési sablont amit bármilyen jövőbeli projektre fel lehet használni. Emellett lehetőséget nyújt különböző szoftverfejlesztési szabályok meghatározására is melyeket alkalmazni lehet az összes fejlesztett szoftverrel szemben. A frissen fejlesztett beágyazott szoftverek automatikus tesztelésen esnek át, ahol jelentős mennyiségű metrikának kell megfelelniük mielőtt az adott elektronikus vezérlőegységekre kerülnek. A szakdolgozatomban egy automatizálási szkript létrehozásának és tesztelésének folyamatát mutatom be, egy hatékony és biztonságos beágyazott szoftverfejlesztési környezet kialakításával.

List of Figures

16 References

[1] Albertsen, Sofus. *War of the CI servers – GitLab vs. github vs. Jenkins.* June 2022. URL: `https://www.eficode.com/blog/war-of-the-ci-servers-gitlab-vs-github-vs-jenkins`.

[2] Anwar, Nahid and Kar, Susmita. “Review Paper on Various Software Testing Techniques & Strategies”. In: *Global Journal of Computer Science and Technology* (May 2019), pp. 43–49. DOI: `10.34257/GJCSTCVOL19IS2PG43`.

[3] Atlassian. *What is version control: Atlassian Git Tutorial.* URL: `https://www.atlassian.com/git/tutorials/what-is-version-control`.

[4] *AWS Codepipeline | Continuous Integration & Continuous Delivery.* URL: `https://aws.amazon.com/codepipeline/` (visited on 05/08/2022).

[5] *Azure pipelines.* URL: `https://azure.microsoft.com/en-us/services/devops/pipelines/` (visited on 05/08/2022).

[6] *Bamboo support.* URL: `https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html` (visited on 05/08/2022).

[7] *Batch scripting.* URL: `https://www.seobility.net/en/wiki/Batch_Scripting` (visited on 05/08/2022).

[8] Baua. *What is Electronic Control Unit and what does it do.* Jan. 2022. URL: `https://bauaelectric.com/cars/what-is-electronic-control-unit-and-what-does-it-do/` (visited on 05/01/2022).

[9] Bolmér, Percy. *GitHub actions in action.* Feb. 2021. URL: `https://betterprogramming.pub/github-actions-in-action-3b10083cc700` (visited on 05/04/2022).

[10] Brown, Korbin. *Bash scripting vs PowerShell.* Mar. 2022. URL: `https://linuxconfig.org/bash-scripting-vs-powershell` (visited on 05/10/2022).

[11] *Centralized vs distributed version control system.* Jan. 2021. URL: `https://www.htown-tech.com/blogs/centralized-vs-distributed-version-control-system`.

[12] *Centralized vs distributed version control: Which one should we choose?* Sept. 2021. URL: `https://www.geeksforgeeks.org/centralized-vs-distributed-version-control-which-one-should-we-choose/`.

[13] Choudhury, Ambika. *5 reasons why Jenkins is the most-used open source tool by developers.* Sept. 2020. URL: `https://analyticsindiamag.com/5-reasons-why-jenkins-is-the-most-used-open-source-tool-by-developers/`.

[14] *CI/CD in Agile Development: Teamcity CI/CD guide.* URL: `https://www.jetbrains.com/teamcity/ci-cd-guide/agile-continuous-integration/` (visited on 05/08/2022).

[15] *Cloudbees vs Jenkins: What are the differences?* May 2022. URL: `https://stackshare.io/stackups/cloudbees-vs-jenkins` (visited on 05/08/2022).

[16] Dar, Renana. *Top version control systems.* Aug. 2022. URL: `https://www.incredibuild.com/blog/top-version-control-systems`.

[17] Ebert, Christof and Jones, Capers. “Embedded Software: Facts, Figures, and Future”. In: *Computer* 42.4 (2009), pp. 42–52. DOI: `10.1109/MC.2009.118`.

[18] Emadi, Ali. *Handbook of Automotive Power Electronics and motor drives.* Taylor & Francis, 2005.

[19] Eshiett, Joseph. *Github actions vs Jenkins: Which should you use?* Nov. 2021. URL: https://faun.pub/github-actions-vs-jenkins-which-should-you-use-d7ba6800e8bb (visited on 05/10/2022).

[20] Firesmith, Donald. *Using V Models for Testing.* Nov. 2013. URL: https://insights.sei.cmu.edu/blog/using-v-models-for-testing/ (visited on 05/08/2022).

[21] freeCodeCamp.org. *Interpreted vs compiled programming languages: What's the difference?* Jan. 2020. URL: https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/ (visited on 04/27/2022).

[22] GitHub. URL: https://docs.github.com/en/actions/guides (visited on 05/03/2022).

[23] *Jenkins.* URL: https://www.jenkins.io/ (visited on 05/08/2022).

[24] Johansson, G. "Job demands and stress reactions in repetitive and uneventful monotony at work". en. In: *Int J Health Serv* 19.2 (1989), pp. 365–377.

[25] Klier, Thomas H. and Rubenstein, James M. "Making cars smarter: The growing role of electronics in automobiles". In: *Chicago Fed Letter* Oct (2011). URL: https://ideas.repec.org/a/fip/fedhle/y2011ioctn291a.html.

[26] McConnell, Steve. "Daily Build and Smoke Test". In: *IEEE Softw.* 13.4 (1996), pp. 143–144. DOI: 10.1109/MS.1996.10017. URL: http://doi.ieeecomputersociety.org/10.1109/MS.1996.10017.

[27] Milu. *Git explained: The basics.* May 2020. URL: https://dev.to/milu_franz/git-explained-the-basics-igc.

[28] Morin, Rich and Brown, Vicki. *Scripting Languages.* 1999. URL: http://preserve.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html (visited on 04/27/2022).

[29] Nikiforova, Ekaterina. *Misra coding standard and static code analyzers.* Apr. 2020. URL: https://embeddedcomputing.com/technology/software-and-os/ides-application-programming/misra-coding-standard-and-static-code-analyzers.

[30] Pittet, Sten. *Continuous Integration vs. delivery vs. deployment.* URL: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment (visited on 05/08/2022).

[31] Rassokhin, Alexander. *Introducing TeamCity Cloud – A Managed CI/CD Service by JetBrains.* Apr. 2021. URL: https://blog.jetbrains.com/teamcity/2021/04/introducing-teamcity-cloud-a-managed-ci-cd-service-by-jetbrains/ (visited on 05/08/2022).

[32] Rivett, Roger S. "Hazard identification and classification: ISO26262- the application of IEC61505 to the automotive sector". In: *2009 5th IET Seminar on SIL Determination.* 2009, pp. 1–24. DOI: 10.1049/ic.2009.0220.

[33] Sarker, Iqbal et al. "A Survey of Software Development Process Models in Software Engineering". In: *International Journal of Software Engineering and its Applications* 9 (Nov. 2015), pp. 55–70. DOI: 10.14257/ijseia.2015.9.11.05.

[34] Sharma, Akshansh et al. "Python: The Programming Language of Future". In: *IJIRT* 6.12 (May 2020), pp. 115–118.

[35] *Spinnaker.* June 2021. URL: https://spinnaker.io/ (visited on 05/08/2022).

[36] Staff, DevPro Journal. *Gitlab 14 delivers modern DevOps in one platform.* July 2021. URL: https://www.devprojournal.com/news/gitlab-acquires-unreview-to-expand-its-open-devops-platform-with-machine-learning-capabilities-2/.

[37] Stobierski, Tim. *Agile vs. Scrum: What's the difference?* Mar. 2021. URL: https://www.northeastern.edu/graduate/blog/agile-vs-scrum/ (visited on 05/08/2022).

[38] Szydlowski, Craig P. "CAN Specification 2.0: Protocol and Implementations". In: *SAE Technical Paper 921603* (1992). DOI: https://doi.org/10.4271/921603. URL: https://www.sae.org/publications/technical-papers/content/921603/.

[39] *What is deployment automation?* Sept. 2020. URL: https://www.redhat.com/en/topics/automation/what-is-deployment-automation (visited on 05/03/2022).

[40] *What is the ISO 26262 functional safety standard?* June 2021. URL: https://www.ni.com/hu-hu/innovations/white-papers/11/what-is-the-iso-26262-functional-safety-standard-.html#toc2 (visited on 05/08/2022).

[41] *Workspaces and Sandboxes.* URL: https://jazz.multichoice.co.za/clmhelp/index.jsp?re=1&amp;topic=%5C%2Fcom.ibm.team.scm.doc%5C%2Ftopics%5C%2Fc_workspaces.html&amp;scope=null.

Appendices

A Current Release Folder Content Deletion Appendix

This appendix presents the current internal release folder content deletion process automation script.

```
SET RELEASE_FOLDER=%WORKSPACE%\JWS\SW\Release

SUBST t: %RELEASE_FOLDER%
SET NEW_RELEASE_FOLDER=T:

IF EXIST "%NEW_RELEASE_FOLDER%" (
    CD /d %NEW_RELEASE_FOLDER%
    FOR /F "delims=" %%i in ('dir /B') do (
        IF NOT "%%i" == "TEMPLATE" (
            RMDIR "%%i" /S/Q || del "%%i" /S/Q
        )
    )
)

SUBST t: /d
CD /d %RELEASE_FOLDER%
```

B Fetching Software Version Based Appendix

This appendix presents the fetching version based process automation script.

```
@ECHO OFF
SETLOCAL ENABLEDELAYEDEXPANSION
SET RB_ECUSWVERSION_PATH=%WORKSPACE%\JWS\SW\MM\MCore\AIM\CDD\CpxSL\rbd_ECUVERSION
    \inc\RB_ECUSWVERSION.h
SET count=1

FOR /f "delims=" %%a IN ('FINDSTR /v /c
    :"-----------------------------------------------------------------------"
     %RB_ECUSWVERSION_PATH%') DO (
   IF !count! equ 25 (set "text=%%a" & goto :next)
   SET /a count+=1
)

:NEXT
ECHO %text%>version_based_line.txt
FOR /f "tokens=8,10" %%a in (version_based_line.txt) do (
  SET version=%%a
  SET base=%%b
  ECHO S!base!>>alm.txt
  ECHO S!base!_!version:~1!>>version_based.txt
  )

DEL version_based_line.txt
ENDLOCAL
```

C Single File Analysis Appendix

This appendix presents all SFA code written.

```
def override_bauhaus(env) {
    artInstance = new ArtifactoryHelpers(this, 'aebe-png-elvis-local', '', env.
        rtcCredentials)
    rtcHelpers = new bosch.aebedo.RtcHelpers(this)
    rtcHelpers.credentials = "${env.rtcCredentials}"
    rtcHelpers.loadRulePath = "${env.rtcLoadRule}"
    rtcWorkspace = "${env.rtcTestWorkspace}"

    String dataVersion = artInstance.getLatestVersion('onMerge_Test',
                                                              ['type': 'bin'],
                                                              env.projectArtApi)
    echo 'Version:'
    echo dataVersion
    Map binProbs = ['version': "${dataVersion}",
                'type': "bin"]
    try {
                artInstance.download('onMerge_Test',
                                    "C:\\workspace\\jws_vv\\Objects",
                                    binProbs, true, true, true)
                } catch (e) {
                    echo 'No Memory metrics from Artifactory available'
                    echo e.toString()
                }
    dir('C:/workspace/jws_vv/Objects'){
        unzip zipFile: 'make_file_objects.zip'
    }

    dir('JWS_SCA/SW/ToolsAndConfiguration/Bauhaus')
    {
        *** SINGLE FILE ANALYSIS SCRIPT ***
    }

    // User violation list notification

    dir('JWS_SCA/SW/ToolsAndConfiguration/CICD/scripts/Bauhaus'){
        withCredentials([usernamePassword(credentialsId: 'b9fbfe74-aa64-4401-bbd5
            -3cabb2c0fcad',
            passwordVariable: 'PASSWORD', usernameVariable: 'USERNAME')]) {
                bat "launchJsonParse.bat %USERNAME% %PASSWORD%"
        }
    }

    ...
    MORE TESTS AND SOFTWARE ANALYSIS

    ...

    dir('JWS/SW/Build/_output/obj')
        {
            zip dir: './', zipFile: "make_file_objects.zip"
            stash name: 'Make_File_Objects', includes: "make_file_objects.zip"
        }

}
```

D Automated Single File Analysis Batch Script Appendix

This appendix presents the Batch script used to automated the SFA process.

```
@echo off

:: Batch parameters
:: set BCMF_SOFTWARE_BASE=%workspace%
:: set BCMF_TEMP_OUT=%OUTPUT%
set PYTHON_EXE="D:\\Python35\\python.exe"

:: For local script testing
:: set PYTHON_EXE="C:\toolbase\python\3.8.1.0.6\python-3.8.1.amd64\python.exe"

:: set BAUHAUS_PROJECT=PNG_ELVIS_OnCommit
:: set SENDER=PAN3_PNGELVIS.Jenkins@hu.bosch.com
set URL="https://rb-aebe-bauhaus.de.bosch.com/axivion/projects/PNG_ELVIS_OnCommit
    /issues?kind=SV&user=ANYBODY&namedFilter=p_axivion_default&format=json"
set SCRIPT_PATH=%~dp0
set TechUser=%1
set pwd=%2

%PYTHON_EXE% %SCRIPT_PATH%jsonParse.py %URL% %TechUser% %pwd%
if %ERRORLEVEL% neq 0 goto END
echo.

:END
echo Jenkins "project analysis notifications" finished with errorlevel %
    ERRORLEVEL%
echo.
exit /b %ERRORLEVEL%

pause
```

E Automated Single File Analysis Python Script Appendix

This appendix presents the Python script used to automated the SFA process.

```
import os
import sys
import json
import requests

class AddedChangesExceededException(Exception):
    def __init__(self, violation, added, message="Added changes are greater than
        0"):
        self.violation = violation
        self.added = added
        self.message = message
        super().__init__(self.message)

    def __str__(self):
        return self.violation + " " + self.message + " -> " + str(self.added)

if __name__ == ’__main__’:
    URL = sys.argv[1]
    USERNAME = sys.argv[2]
    PASSWORD = sys.argv[3]

    try:
        returned_response = requests.get(URL, verify=False, auth=(USERNAME,
            PASSWORD))
        returned_json = returned_response.json()

        sv_added = returned_json["endVersion"]["issueCounts"]["SV"]["Added"]
        # sv_removed = returned_json["endVersion"]["issueCounts"]["SV"]["Removed"]
        # sv_result = abs(sv_added - sv_removed)

        mv_added = returned_json["endVersion"]["issueCounts"]["MV"]["Added"]
        # mv_removed = returned_json["endVersion"]["issueCounts"]["MV"]["Removed"]
        # mv_result = abs(mv_added - mv_removed)
    except Exception as e:
        print(e)

    if (sv_added > 0):
        raise AddedChangesExceededException("SV", sv_added)

    if (mv_added > 0):
        raise AddedChangesExceededException("MV", mv_added)
```

F Smoke Test Appendix

This appendix presents the smoke test pipeline Jenkinsfile.

```
@Library('AEBEDevops@release/latest_stable_release') _

import bosch.aebedo.ArtifactoryHelpers
import bosch.aebedo.RtcHelpers
import bosch.aebedo.KpiHelpers
import java.time.LocalDateTime
import bosch.aebedo.WarningHelpers

def override_build_setup(env) {
    ***
    Build Setup - Workspace configured
    ***
}

def override_build(env) {
    ***
    Build Scripts
    ***
}

def override_build_teardown(env) {
    ***
    Build stage teardown
    ***
}

def override_upload_setup(env) {
   ***
   Upload setup - Unstash data from zip files
   ***
}

def override_upload(env) {
    ***
    Upload zip file data that was unstashed
    ***
}

def override_upload_teardown(env) {
    ***
    Upload stage teardown
    ***
}
```

```
void override_uploadPackage(env) {
    ***
    Upload package stage (Workspace and stream)
    ***
}

def override_get_test_data() {
    rtcHelpers = new bosch.aebedo.RtcHelpers(this)
    rtcHelpers.credentials = "${env.rtcCredentials}"
    rtcHelpers.loadRulePath = "${env.rtcLoadRule}"
    rtcWorkspace = "${env.rtcTestWorkspace}"

    dir ('C:/workspace/jws_vv/tools/CICD_Utility_Tools'){
    bat """
                    echo delete old build files
            call conda activate ${env.pythonTestEnv}
            ::python main.py F
            """
    }

   String dataVersion = artInstance.getLatestVersion('onMerge',
                                                            ['type': 'bin'],
                                                            env.projectArtApi)
    echo 'Version:'
    echo dataVersion
    Map binProbs = ['version': "${dataVersion}",
                'type': "bin"]
    try {
                artInstance.download('onMerge',
                                    "C:\\workspace\\jws_vv\\Builds",
                                    binProbs, true, true, true)
                } catch (e) {
                    echo 'No Memory metrics from Artifactory available'
                    echo e.toString()
                }
    dir('C:/workspace/jws_vv/Builds'){
        unzip zipFile: 'build_output_C1.zip'
    }
}

def override_run_test() {
    ***
    Smoke Test scripts run
    ***
}

def override_test_upload(env) {
    ***
    Upload smoke test results
    ***
}

node("master") {
        String projectName = 'PNG_Elvis'
    String splunkIndex = 'aebedevops'
    String pipelineClass = 'onCommit'
    withCredentials([string(credentialsId: 'SplunkToken', variable: 'token')]) {
            //kpiHelpers = bosch.aebedo.KpiHelpers.getInstance(this, "${token}",
                splunkIndex, projectName, pipelineClass)
        }

    TIMESTAMP = new Date().format('yyyyMMdd_HHmmss', TimeZone.getTimeZone(env.
        timeZone ?: 'Australia/Melbourne'))
    currentBuild.displayName = "png_elvis_dev_onCommit_${TIMESTAMP}"
}

node("PNG_Elvis_Build") {
    timestamps {
        timeout(time: 1, unit: 'HOURS') {
            cleanWs()
            def version = "0.0.${BUILD_NUMBER}".toString()
                        artInstance = new ArtifactoryHelpers(this, 'aebe-png-elvis-
                            local', '', env.rtcCredentials)

            def buildStage = new bosch.aebedo.CustomStage(setup: this.&
                override_build_setup, stageMethod: this.&override_build, teardown:
                 this.&override_build_teardown)
            def uploadStage = new bosch.aebedo.CustomStage(setup: this.&
                override_upload_setup, stageMethod: this.&override_upload,
                teardown: this.&override_upload_teardown)
            def uploadPackageStage = new bosch.aebedo.CustomStage(stageMethod:
                this.&override_uploadPackage)

                        def testStage = new bosch.aebedo.CustomStage(node: "
                            PNG_Elvis_Wid_CICD_New_PC", setup: this.&
                            override_get_test_data,
                                                   stageMethod: this.&
                                                       override_run_test,
                                                   teardown: this.&
                                                       override_test_upload)

            def skippedStage = new bosch.aebedo.CustomStage(skip: true)

            withEnv([ENV_VARIABLES]) {
                try {
                                rtc = new RtcHelpers(this)
                exec_module_on_merge this, [setupEnvironment: skippedStage,
                                    build: buildStage,
                                    buildPackage: skippedStage,
                                    unittest: skippedStage,
                                    extraTests: skippedStage,
                                    staticCodeAnalysis: skippedStage,
                                    uploadMetrics: uploadStage,
                                    uploadPackage: skippedStage
                                   ]

                          exec_module_on_merge this, [setupEnvironment:
                               skippedStage,
                                    build: skippedStage,
                                    buildPackage: skippedStage,
                                    unittest: skippedStage,
                                    extraTests: testStage,
                                    staticCodeAnalysis: skippedStage,
                                    uploadMetrics: skippedStage,
                                    uploadPackage: skippedStage
                                   ]

                          currentBuild.result = 'SUCCESS'

              } catch(exc) {
                currentBuild.result = 'FAILURE'
                echo exc.toString()
                def FAIL_MAIL=""
                emailext attachLog: true, body: '${SCRIPT, template="pipeline.
                    groovy"}', to: "$FAIL_MAIL", subject: '$DEFAULT_SUBJECT'
                throw exc
              } finally {
                          try {
                                 cleanWs()
                          } catch(e) {
                                 echo "E Deleting WS"
                                 bat "dir /a"
                          }

              }
          }
      }
  }
}
```

G Qualification Test Appendix

This appendix presents the qualification test pipeline Jenkinsfile.

```
@Library('AEBEDevops@release/latest_stable_release') _

import bosch.aebedo.ArtifactoryHelpers
import bosch.aebedo.FossidHelpers
import bosch.aebedo.CustomStage
import bosch.aebedo.RtcHelpers
import bosch.aebedo.VectorCast
import bosch.aebedo.AebeDevOps
import bosch.aebedo.KpiHelpers
import java.time.LocalDateTime


def override_qualificationtest_setup(env) {
       kpiHelpers.startTimestamp('Unittest')
       rtcWorkspace = "${env.rtcWorkspace}"
       rtc.checkoutRtc(rtc.getBuildTypeWorkspaceMap(rtcWorkspace))

       withEnv(ENV_VARIABLES) {

       vcastObj = new VectorCast(this, artHelpers)
              vcastObj.setupVcast()
    }

}

def override_qualificationtest(env) {
    withEnv(ENV_VARIABLES) {

        vcastObj.executeVcast()
    }
}

def override_qualificationtest_teardown(env) {

    withEnv(ENV_VARIABLES) {

    vcastObj.cleanVcast()
    }
}

node("master") {
    TIMESTAMP = new Date().format('yyyyMMdd_HHmmss', TimeZone.getTimeZone(env.
        timeZone ?: 'Australia/Melbourne'))
    currentBuild.displayName = "png_elvis_dev_cont_int_${TIMESTAMP}"
```

```
        String projectName = 'PNG_Elvis'
        String splunkIndex = 'aebedevops'
        String pipelineClass = 'Metric'
        String pipelineName = 'ep002_dev_nightly'
        withCredentials([string(credentialsId: 'SplunkToken', variable: 'token')])
             {
                kpiHelpers = bosch.aebedo.KpiHelpers.getInstance(this, "${token}",
                     splunkIndex, projectName, pipelineClass, pipelineName)
        }
}

node("PNG_Elvis_Build") {
    timestamps {
        timeout(time: 20, unit: 'HOURS') {
            cleanWs()
            def version = "0.0.${BUILD_NUMBER}".toString()

            artHelpers = new ArtifactoryHelpers(this, 'aebe-png-elvis-local', '',
                env.rtcCredentials)

            abeDevOpsobj = new AebeDevOps(this)

            def qualificationtestStage = new bosch.aebedo.CustomStage(node: "
                PNG_Elvis_Build", setup: this.&override_qualificationtest_setup,
                stageMethod: this.&override_qualificationtest, teardown: this.&
                override_qualificationtest_teardown)

            parallel_stages_def = [
                'Qualification_Test' : qualificationtestStage
            ]

            List unittest_common_env = ["scmPath=jws/SW/ToolsAndConfiguration/CICD
                /scripts", "dirVCAST=C:/VCast20SP6", "nodeName=PNG_Elvis_Build"]

            List qualificationtest_env = [
                        "vcastProj=",
                        "importScript=Execute_SW_QUALIFICATION_TEST_GreenHills.bat
                            ",
                        "uploadDir=${currentBuild.displayName}/
                            Cont_Int_Qualification/Unittest",
                        "reportResults=Qualification_Test_Report.zip"] +
                            unittest_common_env

            withEnv([ENV_VARIABLES]) {
                try {

                    rtc = new RtcHelpers(this)
                    rtc.lscmUtil.login()

                    parallel_stages = abeDevOpsobj.prepareParallelStages('Checks',
                        parallel_stages_def)
                    parallel(parallel_stages)

                    currentBuild.result = 'SUCCESS'
                    def SUCCESS_MAIL=""
                    def TEXT=""
                    emailext attachLog: false, body: TEXT, to: "$SUCCESS_MAIL",
                        subject: '$DEFAULT_SUBJECT'

                }
                catch(exc)
                {
                    currentBuild.result = 'FAILURE'
                    echo exc.toString()
                    def FAIL_MAIL=""
                    emailext attachLog: true, body: '${SCRIPT, template="pipeline.
                        groovy"}', to: "$FAIL_MAIL", subject: '$DEFAULT_SUBJECT'
                    throw exc
                }
                finally
                {
                    kpiHelpers.addData(KpiHelpers.SplunkMetrics.PIPELINE_STATUS,
                        currentBuild.result)
                    kpiHelpers.forwardData('JenkinsData')
                }
            }
        }
    }
}


\begin{lstlisting}[caption=Qualification Test Execution Script, captionpos=b]
set MP=%WORKSPACE%\jws\SW\ToolsAndConfiguration\CICD\scripts\PNG_ELVIS_GreenHills
    .vcm
set VECTORCAST_DIR=D:/VCast20SP6

call %WORKSPACE%\jws\SW\ToolsAndConfiguration\CICD\scripts\executeManage-
    ELVIS_GreenHills_new.bat %WORKSPACE%\jws\SW\ToolsAndConfiguration\
    Qualification_test\SWQT\SW_QUALIFICATION_TEST.env

%VECTORCAST_DIR%\manage --project=%MP% --build-execute
```