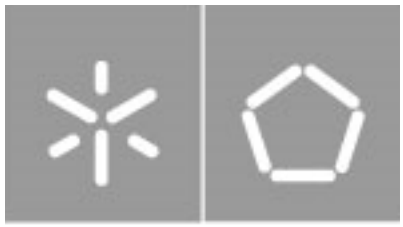
Universidade do Minho
Escola de Engenharia

Ana Margarida Oliveira Ferreira

Continuous Inspection of Software Quality in an Automotive Project

July 2023

Continuous Inspection of Software Quality in an Automotive Project
Ana Margarida Oliveira Ferreira
UMinho | 2023



Universidade do Minho
Escola de Engenharia

Ana Margarida Oliveira Ferreira

Continuous Inspection of Software Quality in an Automotive Project

Master's Work Dissertation Report
Integrated Master's in Engineering and Management of Information Systems

Work performed under the supervision of
Professor Doutor Miguel Abrunhosa de Brito

July 2023

COPYRIGHT AND TERMS OF USE OF THE WORK BY THIRD PARTIES

ACKNOWLEDGEMENTS

I want to start by expressing my gratitude to my parents for their consistent support throughout my journey, for providing me this opportunity, and always fighting for my best interest. Then, thank Rodrigo for being the most important person in my life and for being able to bother me like anybody else can.

A special thanks to my advisor, Professor Miguel Abrunhosa de Brito, for his support from day one and for helping me at any time. Your expertise, patience, and dedication have been instrumental in shaping my research and helping me navigate the challenges along the way.

Additionally, I want to thank my colleagues at CI&Tools for helping me out, teaching me, and turning me into an authentic mechanic, I am very pleased to work and learn with you. Above all, I want to express my sincere thanks to José Lima and José Marques for all the patient examining the entire dissertation, for giving me this opportunity, for their support, and for being great mentors.

I also want to show my gratitude to João for his unwavering support throughout my life's decisions, for his everlasting patience, and for being there for me on both good and bad days. Also, I'm grateful to my closest friends for their constant support and motivation, I hope we become rich together. Thanks to the course friends I've made over the past five years for adding to the beauty of this journey.

Lastly, even if they couldn't tell anyone the name of my course, I'd want to thank my grandparents and all my family for being there and for keeping me motivated, I think this is calling for a toast. Particularly those who are no longer present physically to celebrate, I hope they are proud of my trip. PS: I am an engineer, not a computer specialist, and much less of a home appliance expert!

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

Continuous Inspection of Software Quality in an Automotive Project

Due to the amount of software that is produced every day in the automotive industry, improve software quality became a necessity, especially in areas where safety is a critical point, such as autonomous driving. Following on from this, continuous inspection of software is important, timely and central, to guarantee software quality and avoid the worst scenarios related to the automotive industry, particularly with autonomous driving.

Even so, this proposal is motivated by the huge number of failures associated with low quality software. Additionally, extending a new feature could damage the success of the entire project because of potential software flaws or poor programming practices. After all, bad quality software leads to the need to review and rewrite the software and then, software's lifecycle is always going around, so this has associated costs in terms of money, time and resources and can lead to a bad reputation for the company, due to not meeting deadlines, delivering low quality software, among other factors.

In this way, the main objective of this research is to achieve a solution for continuous inspection of software quality in the context of the company Bosch Car Multimedia. For this, a tool called CIAnalyzer Tool was developed, and an architecture was designed and implemented in which CIAnalyzer acts as an intermediary between SonarQube and Jenkins providing an analysis of the software, every time someone delivers a software to the repository. This software passes through an analysis and, if it passes the analysis, is delivered to the repository, contributing to the continuous inspection of the software code.

After the implementation of this solution, there was a 77.7% decrease in bugs, a 64.6% reduction in code smells, and 100% in vulnerabilities existing in the repository. The results were good, and this dissertation helps in theory and in practice in CI/CD area, and the knowledge created was helpful, contributing with a solution capable to provide continuous inspection of the software quality, in an automotive project.

Keywords: continuous inspection; software quality; automotive industry; continuous delivery; continuous integration

RESUMO

Inspeção Contínua da Qualidade do Software num Projeto Automóvel

Devido à quantidade de *software* que é produzido diariamente na indústria automóvel, melhorar a qualidade do *software* tornou-se uma necessidade, especialmente nas áreas em que a segurança é um ponto crítico, como a condução autónoma. Neste seguimento, a inspeção contínua do *software* é importante, oportuna e central, de forma a garantir a qualidade do software e evitar os piores cenários relacionados com a indústria automóvel, particularmente com a condução autónoma.

Assim sendo, esta proposta é motivada pelo enorme número de falhas associadas a software com baixa qualidade. Além disso, estender uma nova funcionalidade poderia estragar o sucesso de todo o projeto, devido a falhas no software ou más práticas de programação. *Software* com má qualidade leva à necessidade de rever e reescrever o software e, por isso, o ciclo de vida do *software* anda sempre às voltas, o que tem associado custos em termos de dinheiro, tempo e recursos e pode levar a uma má reputação da empresa, devido ao incumprimento de prazos, entrega de *software* com baixa qualidade e outros fatores.

Neste sentido, o principal objetivo desta dissertação é obter uma solução para inspeção contínua da qualidade do *software*, no contexto da empresa *Bosch Car Multimedia*. Para isso, uma ferramenta chamada *CIAnalyzer Tool* foi construída e foi desenhada e implementada uma arquitetura, onde o *CIAnalyzer* age como intermediário entre o *SonarQube* e o *Jenkins*, providenciando uma análise ao *software*, sempre que é entregue código para o repositório. Esse *software* passa por uma análise e, caso passe nessa análise, é entregue para o repositório, contribuindo para a inspeção contínua do *software*.

Após a implementação da solução, houve uma diminuição em 77,7% dos *bugs*, 64,6% de *code smells* e 100% das vulnerabilidades existentes no repositório. Assim, os resultados foram positivos e esta dissertação ajuda na teoria e na prática na área de integração e entrega contínua, e o conhecimento criado foi bastante útil, contribuindo com uma solução capaz de providenciar a inspeção contínua da qualidade do *software*, num projeto automóvel.

Palavras-chave: inspeção contínua; qualidade do software; indústria automóvel; entrega contínua; integração contínua.

INDEX

List of Abbreviations and Acronyms

ADS	Autonomous Driving Systems
API	Application Programming Interface
CD	Continuous Delivery
CI	Continuous Integration
CQID	Code Quality Issues Density
CSS	Cascading Style Sheets
DSR	Design Science Research
GUI	Graphical User Interface
HTML	HyperText Markup Language
IBM	International Business Machines Corporation
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IT	Information Technology
Json	JavaScript Object Notation
LiDAR	Light Detection and Ranging
NPM	Node Package Manager
OEM	Original Equipment Manufacturer
S.A.	Sociedade Anónima
SAFe	Scaled Agile Framework
SPICE	Software Process Improvement and Capability Determination
Tapco	Test Automation Progression in Continuous Practices
XML	Extensible Markup Language
YAML	Yet Another Markup Language

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This chapter is intended to make a context of the subject under study, followed by a brief description of the company where the platform will be validated, the motivation and objectives for this dissertation. After that, the methodology is shown and, finally, the document structure is presented.

1.1 Context

With emerging technology and a multitude of factors such as business competitiveness, digital disruptions, and digital reinventions, the requirements of the markets have changed the environment for software companies (Gupta et al., 2022). Thousands and thousands of lines of source code are written every day to develop software. That software should be understandable by people other than its developers(Iqbal et al., 2017). According to (Lomio et al., 2022), developers spend more time fixing bugs and refactoring the code to increase maintainability than developing new features.

In the automotive industry, the amount of software in modern cars is significant and is growing with the release of new cars (Durisic et al., 2011). In their words, the permanently increasing complexity of in-car electronics and the rapidly growing amount of automotive software running on embedded electronic control units, places higher demands on quality assurance for the future. Quality cannot be implemented into software on embedded control units after their development, so methods for defects detection must be constituted to automatically stop development to fix a problem before the defect continues downstream (Farkas, 2008).

In this way, CI/CD have been shown to be very useful to improve the quality of software products (Zampetti et al., 2022). The sustainability of the utilization of CI/CD technologies is reflected within the detection of errors, which suggests the appliance of the law to attain profitability and therefore the long-term aspect of the assembly of autonomous vehicles. By using appropriate code analysis and applying standards, software quality improvement techniques are integrated (Dakic et al., 2022).

1.2 The Company

The solution will be validated by implementing a support platform in the context of Bosch Car Multimedia, S.A[1]. This unit belongs to the Automotive Eletronics division, and it is the group's largest in Portugal. Focused on the development and production of multimedia solutions and car sensors, Bosch Car Multimedia export almost all of its production, being on the podium of the Portuguese companies that most exports in the country and also one of the largest employers in the Braga region.

The platform will be validated in the context of the DevOps Team working in the LiDAR project. LiDAR (light detection and ranging) is a sensor that can be used in automobiles, thus making inroads into the key technology for fully automated driving.

1.3 Motivation

Due to the amount of software that is produced every day in the automotive industry, improving software quality became a necessity, especially in areas where safety is a critical point, such as autonomous driving. The sooner a software error is found, the easier it will be to change the code and less damage will be done to the project, avoiding compromising the whole project. Following on from this, continuous inspection of software is important, timely and central, to guarantee software quality and avoid worst scenarios related to the automotive industry, particularly with autonomous driving.

Even so, this proposal is motivated by the huge number of failures associated with low quality software. Additionally, extending a new feature could damage the success of the entire project because of potential software flaws or poor programming practices. After all, bad quality software leads to the need to review and rewrite the software and then, software's lifecycle is always going around, so this has associated costs in terms of money, time and resources and can lead to a bad reputation for the company, due to not meeting deadlines, delivering low quality software, among other factors.

In this way, the present proposal intends to contribute to the theme with a solution for inspecting software quality in the context of an automotive project, related to autonomous driving.

[1] https://www.bosch.pt/a-nossa-empresa/bosch-em-portugal/

1.4 Objectives

The main objective of this dissertation is to achieve a solution for continuous inspection of software quality, in order to ease the creation and maintenance of high-quality software. To achieve the main objective, it is important to highlight the following specific objectives:

- Study the existing scientific knowledge about the research topic.
- Define specifications for the support solution;
- Analyze tools and technologies in the market for continuous inspection of software quality;
- Implement a proof of concept of the support platform;
- Implement the solution for continuous inspection of software quality;
- Define and implement quality checks and quality gates in the support platform, based on specifications;
- Validate the support solution after the implementation, based on the results, the objectives, and the requirements.

1.5 Methodology

In order to achieve the goals for this dissertation, the Design Science Research (DSR) will be the methodology for research. The DSR is a methodological approach that aims to provide solutions to issues encountered in the real world while also making a prescriptive scientific contribution. As a result of this kind of study, an artifact is created that provides a solution to a problem, contributes to the advancement of science, and aids in the problem-solving efforts of organizations (Dresch Aline and Lacerda, 2015).

The process model of Design Science Research proposed by (Peffers et al., 2007)has six stages, and it is represented at Figure 1.

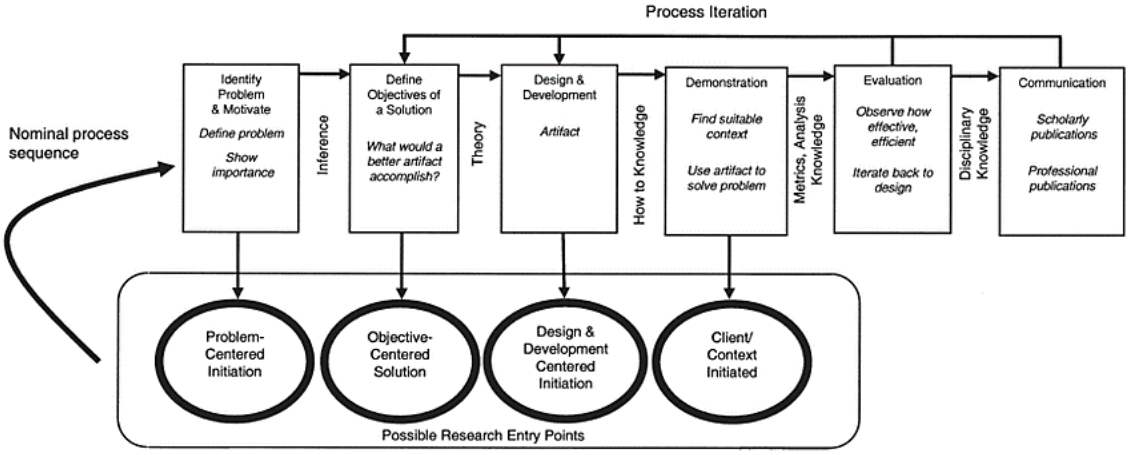


Figure 1 - DSR Process Model

In this dissertation, the DSR method was used. For the first stage, "Identify Problem & Motivate" the problem, context, and motivation for the study were established. The objectives were then specified for the second stage, "Define Objectives of a Solution." The methodology for the development includes three steps for the third phase, "Design & Development":

1. Study.
2. Design.
3. Implementation and validation.

In the first phase – study – the state of art was made, where the concepts about continuous inspection, integration, testing and delivery, agile and DevOps were extended. In addition, a literature review was carried out, verifying the existing studies in the study area. Also, the specific characteristics and properties of the Bosch project, in which the support platform that was developed within the scope of this dissertation was be integrated, were deeply analyze, always having in mind the current state and possible points of improvement.

Then, at the design phase, the specifications of the support solution were analyzed, and the architecture was constructed. Furthermore, a proof of concept of the support solution was made, according to the dissertation objectives and the defined solution specifications, the two analysis types were described, and the design decisions and quality gates were documented.

Finally, the implementation and validation phase is where the system was deployed, making a full implementation of the support solution and documenting that.

The fourth stage of the DSR Process Model is “Demonstration”, where the artifact was used to solve the problem, by testing it and validating the usefulness of the support solution to solve the identified problem.

The “Evaluation” stage has the observation and validation about how effective and efficient the artifact is to solve the problem, by observation, documenting the results of using the support solution. Finally, at the last stage “Communication”, is where the dissertation was delivered in order to communicate the artifact and its usefulness.

In this dissertation document, the DSR Process Model is represented along the existing chapters. The first chapter, Introduction, corresponds to the stage “Identify Problem & Motivate” of DSR, where the problem and motivation were identified, and the stage “Define Objectives of a Solution”, where the objectives for the solution were described.

Then, the State of Art chapter, represents the first phase of “Design & Development”, study, where a thorough research about the work areas was conducted, including a concept analysis and a literature review. The second phase of “Design & Development” is design, that matches with the Solution Specifications and Solution Design chapters of this document, showing the requirements and other specifications for the support solution, as well as the architecture, a proof of concept, the design decisions and the quality gates implemented. The last phase, Implementation and Validation, is represented by Solution Implementation and Solution Validation chapters, where the solution was deployed and the implementation was documented, as well as validated.

Then, the stage “Evaluation” of the DSR Process Model corresponds to the Results chapter, where is showed how effective and efficient the support solution was to solve the problem, documenting the results. Finally, the stage “Communication” corresponds to the articles that have been published within the scope of this dissertation.

1.6 Document Structure

Following on from the methodology, the content of this document is organized in eight chapters, following a logical order according to the DSR process.

In the chapter State of Art, an analysis of the central concepts was made and the literature review on the study area was carried out. Then, the Solution Specifications chapter shows the requirements, the files to be analyzed, an analysis of the existing tools and technologies for continuous inspection and a description of the selected ones.

The Solution Design chapter demonstrates the architecture of the support platform, a proof of concept, a description of the two analysis types, the design decisions, and the quality gates defined for the support

solution. Then, in Solution Implementation, the support platform's implementation is described, starting with SonarQube, CIAnalyzer Tool and, finally, Jenkins.

In Solution Validation is checked if the requirements were met and, in Results, the results are shown, compared and there is a discussion of the improvements. Finally, in Conclusion and Future Work chapter the conclusion of the dissertation and future work to be done within the scope of the dissertation.

2. STATE OF ART

This chapter is divided into two subchapters. The first one presents the theoretical concepts for this study, based on articles found in Scopus and IEEE. To conduct this research, some strings were used, like "continuous inspection", "continuous delivery", "continuous integration", "agile", "DevOps" and "automotive" and the relevant concepts were extracted.

In the same way, for the literature review, the second chapter, some services were used to search for documents related to the study areas covered, in particular Scopus, ResearchGate, Google Scholar and IEEE. In the literature review some keywords were used, including the words "continuous integration", "continuous delivery", "continuous inspection" and "software quality". The search was based on finding existing studies on the area and, in the first instance, all the articles found were selected. Then, the title and the abstract were analyzed to eliminate studies that were not relevant and, whenever not enough, the introduction and/or conclusion were read. Through this, it was possible to eliminate irrelevant articles or the ones that seemed relevant but were not, that is, false positives. Finally, the content of the articles that passed these filters was analyzed and based on them, the literature review was written.

2.1 Concepts

2.1.1 Continuous Inspection

Continuous Inspection is a process to detect the possible bugs in the source code which may occur due to bad code quality and lack of effective peer code review (Rajesh Kumar, 2021). According to the authors, the process of Continuous Inspection is to use available automated tools to continuously inspect code, generate a report on the overall code health, and point out if any violation was detected.

As mentioned by (Kosman & Restivo, 1992), the primary goal of inspection is to reduce software development costs and improve software quality by early defect detection. Early prediction of the quality of software modules prior to software testing and operations can yield great benefits to the software development teams, especially those of high-assurance and mission-critical systems (Khoshgoftaar et al., 2002). So, early indicators of software quality are beneficial for software engineers and managers in determining the reliability of the system, estimating and prioritizing work items, focusing on areas that require more testing, inspections and in general identifying "problem-spots" to manage for unanticipated situations (Nagappan et al., 2008).

In continuous inspection there is an important term: quality gate. Quality Gates indicates a status of approval or denial obtained for the snippet of code submitted to the (De Andrade Gomes et al., 2017).

Due to the ever-growing importance of software in almost all human activities and recognizing the potentially catastrophic consequences of quality defects (Kokol, 2022), companies and countries continue to invest a great deal of time, money, and effort in improving software quality. (Kitchenham & Pfleeger, 1996) believe that controlling the software's overall quality by ensuring code quality is a crucial aspect of software development. In this way, low quality software can cost money and resources to the organization. Quality costs are important because every dollar and labor hour not spent on rework can be used for making better products more quickly or for improving existing products and processes (Slaughter et al., 1998). In this way, low quality software can result in loss of reputation and loss of future business for the company (Tassey & Gregory, 2002). Assessment of software product quality can be achieved by analyzing and specifying the quality characteristics which is defined in software quality models and standards, based on the quality reports defined by the stakeholders. However, assessment of software quality is not an easy task and as time passes, the characteristics of new software systems and products will change (Forouzani et al., 2016).

Throughout the last century, the automobile industry achieved remarkable milestones in manufacturing reliable, safe, and affordable vehicles. Because of significant recent advances in computation and communication technologies, autonomous cars are becoming a reality (Hussain & Zeadally, 2019). The automotive industry continues to change the way we live today, creating innovation and transforming the whole society. These changes are due to the application of new technologies within the vehicles we use today (Dakić & Źivković, 2021). In this way, as mentioned by (Czarnecki, 2018), automated vehicles will eventually completely transform the automotive industry. In the past years, autonomous driving has gained steady improvements and is getting more and more intelligent to precisely sense environments in

the real world, quickly analyze the sensor data, and autonomously make complex decisions (Ren et al., 2020). Due to its direct impact on road safety, multiple prior efforts have been made to study it is the security of perception system (Cao et al., 2019) and, according to (Liu et al., 2019), safety is the most important requirement for autonomous vehicles, so modern vehicles are definitely "software-intensive" systems. Software is now implementing and/or controlling a growing number of traditional functions as well as new innovative functions, made possible only by software and is also taking charge of functions traditionally controlled by the driver (Panaroni et al., 2008).

Autonomous Driving Systems control the physical vehicle using sensors and actuators with intelligence provided by software and data. Due to the close interaction with highly open and dynamic environments, the ADS is expected to meet multiple safety requirements. Failures of ADS could result in accidents that cause damage to the environment, financial losses, injury to people, and loss of lives (Luo et al., 2022). Because of the increasing use of software in the processes and products of automotive companies, software has a significant impact on product quality and company productivity (Grimm, 2003). Hereby, within this industry more and more innovations are based on electronics and software to enhance the safety of the vehicles but also to improve the comfort of the passengers and to reduce consumption and emission. In this way, as the size and complexity of automotive software increases, software requirements specification is gaining importance in automotive software system development (Takoshima & Aoyama, 2015).

2.1.2 Agile and DevOps

The issue of how software development should be organized in order to deliver faster, better, and cheaper solutions has been discussed in software engineering circles for decades (Dybå & Dingsøyr, 2008). A company must adopt agile software delivery cycle to overcome these challenges (Gupta et al., 2022). As mentioned by (Dikert et al., 2016), agile methods have become an appealing alternative for companies striving to improve their performance, but the methods were originally designed for small and individual teams. This creates unique challenges when introducing agile at scale when development teams must synchronize their activities.

Cars include more functional safety systems such as line departure warning, which in the long run may develop towards full autonomous driving and the software also monitors more of the critical operations such as alert monitoring of the driver. In this way, due to the growth in software size and complexity, agile

methods are introduced to improve communication between developers and customers, shorten the time to market and facilitate upgrades of the software with low turn-around times (Myklebust et al., 2020).

Automotive companies must consider strategic initiatives such as agile manufacturing systems to compete globally and respond to dynamic customer demand. In the automotive industry, it is thought that agile manufacturing systems will permit fast cost-effective responses to unpredictable and ever-changing product demand and support rapid product launches for previously unplanned products tailored to meet changing customer desires (Elkins et al., 2004). However, scaling agile is not easy, as large projects often are globally distributed, and have many teams that need to collaborate and coordinate (Paasivaara, 2017).

In line with (Turetken et al., 2016), SAFe has gained rapid attention in the practice and has become an important choice for organizations that need approaches for scaling agile development. It addresses scalability not only by scaling up agile practices, but also by introducing new practices and concepts that integrate with basic and scaled agile practices. In the automotive domain, several dozen development teams work together in a highly coordinated fashion towards the delivery of a product. Systems and software engineering need to be combined in these cases to deliver a final product and the chosen process needs to scale across many teams and different engineering disciplines (Steghöfer et al., 2019).

Based on (Perera et al., 2017), DevOps is extended from certain agile practices with a mix of patterns intended to improve collaboration between development and operation teams. In order to manage the improve of product quality efficiently, classical development and operation tasks were combined which resulted in a development concept termed DevOps (Fitzgerald & Stol, 2017). Figure 2 shows the DevOps lifecycle, that is a loop because the need of collaboration and communication in the eight stages.

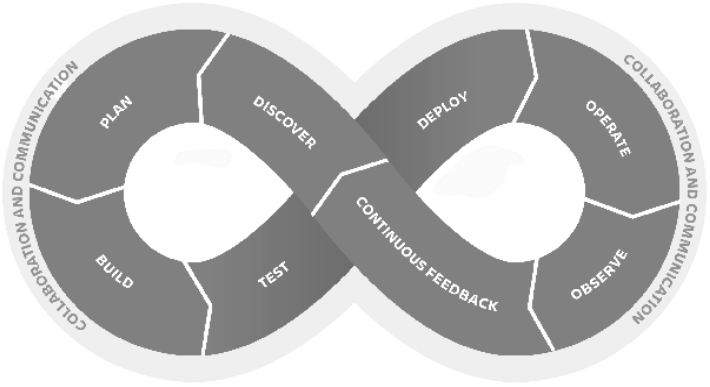


Figure 2 – DevOps. (https://www.atlassian.com/devops)

Essentially, DevOps is a set of methods, that can be seen in Figure 2, in which developers and operations communicate and collaborate to deliver software and services rapidly, reliably (Perera et al., 2017) while

maintaining a high software quality (Jonsson Wold, 2022) and, today, most of the organizations are switching to DevOps for faster and reliable delivery (Batra & Jatain, 2021).

2.1.3 Continuous Integration and Continuous Delivery

The popular agile practices of continuous integration and delivery (CI/CD) have become an essential part of the software development process in many companies (Ståhl & Bosch, 2017). Continuous Integration and Continuous delivery have emerged as a boon for traditional application development and release management practices to provide the capability to release quality artifacts continuously to customers with continuously integrated feedback (Soni, 2015). As mentioned by (Williams, 2018), CI/CD is a well-known practice in DevOps to ensure fast delivery of new features and it enables development teams to deliver code changes constantly and consistently in production, so CI/CD is widely used in DevOps communities, as it allows for teams of all sizes to deploy rapidly changing hardware and software resources quickly and confidently (Sampedro et al., 2018).

CI/CD are the software development industry practices that enable organizations to frequently and reliably release new features and products (Shahin et al., 2017). In the words of (van der Valk et al., 2018), the pressure of reducing time to market and increasing flexibility while keeping quality are leading motivations for these companies to embrace system-wide Continuous Integration and Delivery. According to (Debroy et al., 2018), CI/CD are widely considered to be best practices in software development, allowing significantly reduce deployment time and increase reliability (Di Penta, 2020).

Continuous Integration covers development and testing, and Continuous Delivery extends this with automated integration testing (Rangnau et al., 2020). In this way, Continuous Integration and Continuous Delivery are software engineering processes used in DevOps in order to improve the efficiency of projects, by helping to reduce the amount of time needed to release software to customers while also maintaining a high level of quality (Eddy et al., 2017). In Figure 3 it is possible to see the CI/CD pipeline workflow and all the phases of this method, as Code, Plan, Monitor, among others.

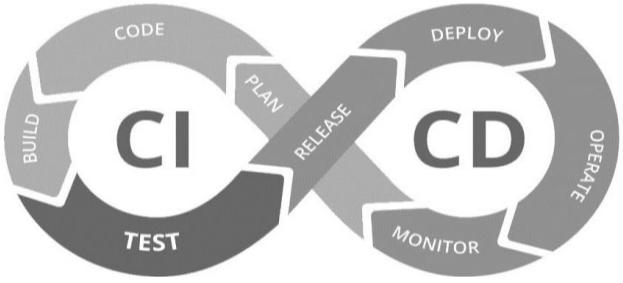


Figure 3 - Continuous Integration and Continuous Delivery. (https://www.mabl.com/blog/what-is-cicd)

In the words of (Zampetti et al., 2021),CI/CD pipelines entail the build process automation on dedicated machines and have been demonstrated to produce several advantages including early defect discovery, increased productivity, and faster release cycles. They also said that effectiveness of CI/CD may depend on the extent to which such pipelines are properly maintained to cope with the system and its underlying technology evolution, as well as to limit bad practices. The results of (Johnson et al., 2013) study confirmed that false positives and developer overload play a part in developers' dissatisfaction with current static analysis tools, so CI/CD platforms must be reliable and stable to avoid this kind of errors.

Basically, CI refers to integrate early, don't keep changes localized to your workspace for long, instead share your changes with team and validate how code behaves (Virmani, 2015). It is a software development practice that leads developers to integrate their work more frequently. Software projects have broadly adopted CI to ship new releases more frequently and to improve code integration.

The adoption of CI is motivated by the allure of delivering new functionalities more quickly (Bernardo et al., 2018). Developers can check-In code frequently, this allows developer teams to check if the code passes testing phase or not consistently. It helps development teams by faster issue discovery, which means the issues and errors in code are found quickly through the automated test. The other benefit is, it reduces the possibilities of integration issues, large scale integration issues are less familiar with the adoption of CI (Garg & Garg, 2019). Main benefits of CI practices are reducing the risk and make software bug free and reliable, which removes the barrier of frequent delivery (Arachchi & Perera, 2018).Figure 4 helps to see the stages of CI, which starts with developer and ends with the release.

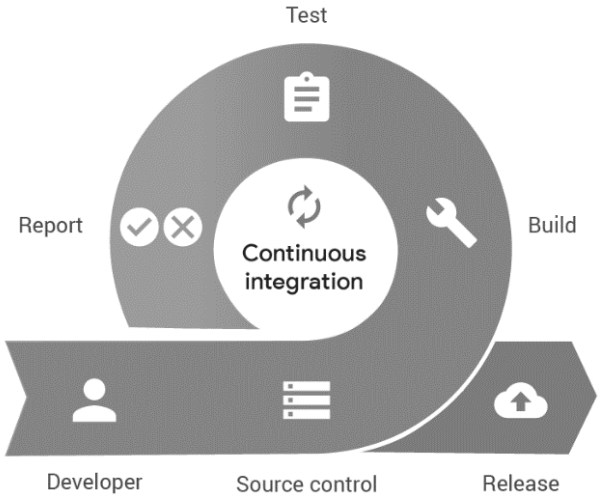


Figure 4 - Continuous Integration. (https://www.pagerduty.com/resources/learn/what-is-continuous-integration/)

Continuous Delivery is a software engineering approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time (Chen, 2015). Allowing swift release cycles, Continuous Delivery has become popular in application software development and is starting to be applied in safety-critical domains such as the automotive industry (Vost & Wagner, 2017). As mentioned by (Vassallo et al., 2016), CD is an agile software development practice in which developers frequently integrate changes into the main development line and produce releases of their software. CD tries to optimize the infrastructure management and the critical need to balance out time and resources (Virmani, 2015). The CD approach goes even further in software development automation, it aims to enable on-demand software release and the practice employs a set of automated stages including the acceptance tests and release process (Górski, 2022).

According to (Chen, 2017), companies that have adopted CD have reported significant benefits. Accelerated time to market, improve product quality, improved customer satisfaction, reliable release, improved productivity, and efficiency are key benefits which motivates companies to invest on CD (Chen, 2015). Basically, CD has five different stages, that happen sequentially and allows to create builds multiple times per day, as it is possible to see in Figure 5.

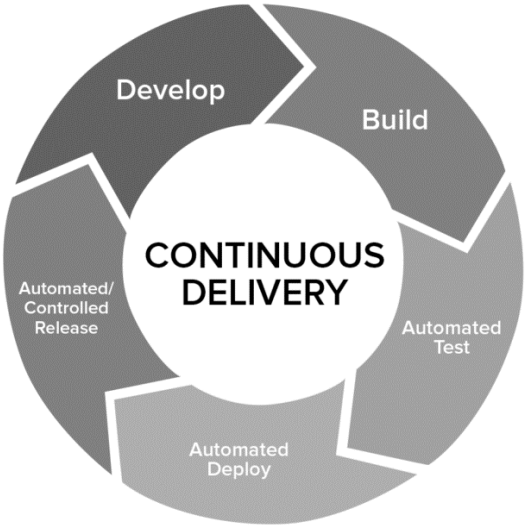


Figure 5 - Continuous Delivery. (https://www.altexsoft.com/blog/business/continuous-delivery-and-integration-rapid-updates-by-automating-quality-assurance/)

Figure 5 shows how Continuous Delivery works, demonstrating that it is a cycle and has many stages as Develop, Build, Automated Test, Automated Deploy and Automated/Controlled release.

2.2 Literature Review

2.2.1 Challenges of CI/CD

Notably, more and more companies are in the process of adopting modern continuous software development practices and approaches like continuous integration, continuous delivery, or DevOps. These approaches can support companies in order to increase the development speed, the frequency of product increments, and the time-to-market. To be able to get these advantages, especially the tooling and infrastructure need to be reliable and secure. Therefore, the goal of this research conducted by (Paule et al., 2019) was to identify which vulnerabilities are present in industry of CD pipelines and how they can be detected. They present the results of an industry case study which includes a qualitative survey of agile project teams regarding the awareness of security in CI/CD, the analysis and abstraction of two CD pipelines measuring the overall risk, matching the impact with the likelihood.

By conducting a survey and inspecting two CD pipelines from industry, they mentioned that found that the security of CD pipelines does not have high priority in development teams. Additionally, though most of the team members have access to the CD pipeline configuration, the lack of security awareness and background in the teams pose a risk to this increasingly business-critical development tool, both in terms of infrastructure, as well as in terms of application. The results of the case study, in their words, show that both investigated CD pipelines have included vulnerabilities which have an overall risk severity between medium and potentially high, because the project teams are dependent on the customers' infrastructure.

Based on metrics and interview results from a large-scale industry project, (Mårtensson et al., 2017) present the factors that, according to the developers themselves, affect how often they deliver software to the mainline. The authors said that the three factors found in this study which affect continuous integration behaviors are:

- The delivery process is time-consuming.
- It is too complicated to deliver.
- No evident value to deliver often to the mainline.

They also defend that build system capacity should be considered as an important factor, but other factors should be seen as at least as important.

In a similar way, because of challenges faced when shifting a primary web application to a micro-services-based architecture and adapting the software for more effective CI/CD, (Debroy et al., 2018) focus on two challenges: related to ling wait times for builds/releases to be queued and completed and the lack of support for tooling, especially from a cross-cloud perspective. Then, they present the solutions that they came up with, which involved rethinking DevOps as it applied to them, and re-building their own CI/CD pipelines based on DevOps-supporting approaches such as containerization, infrastructure-as-code, and orchestration.

They said they were able to get around these barriers by applying DevOps principles such as containerization, orchestration, and cross-collaboration among teams, to ultimately develop a lean and robust CI/CD pipeline which has shown to be very performant and suitable for our needs. In doing so, they also achieved de-coupling and portability – which they defend that are good software engineering principles.

To this extent, in the automotive, most of the innovation is nowadays coming from electronics and software. The pressure of reducing time to market and increasing flexibility while keeping quality are leading motivations for these companies to embrace system-wide continuous integration and delivery, which in the scope of complex automotive value-chains, implies inter-organizational CI/CD. (van der Valk et al., 2018) investigates the challenges and impediments posed by inter-organizational CI/CD in the automotive domain, focusing on legal contracts that regulate the agreements between these companies and transparency intended as the degree/level of information that is shared between the various companies in the value-chain.

As mentioned by (van der Valk et al., 2018), the main findings of this study show that:

- inter-organizational transparency is considered positive but not a necessary condition for inter-organizational CI/CD.
- transparency has positive effects on information sharing among different companies.
- legal contracts are an impediment for inter-organizational CI/CD.

They defend that the results of the study provide useful insights for practitioners that work in similar settings and show that more flexible contracts are needed, and that more transparency between the manufacturer and suppliers is considered as an enabler for interorganizational CI/CD.

2.2.2 Benefits of software quality

According to (Paliotta, 2015), statistics show that more than 50% of auto recalls are now due to software bugs, not mechanical issues. His paper intends to outline why software quality needs to be at the top of the list for automotive OEMs looking to preserve and elevate their brand status. He defends that for quality to improve, continuous integration and continuous software testing are a necessity and well-designed test allow regressions to be caught prior to product release, and lead to a reduction in branch damage and costs associated with product recalls.

The author conclude that are numerous process improvement initiatives that can easily be adopted by any development group regardless of size, as long as the entire team is committed to quality improvement. In his words, improving software quality results in the following business benefits:

- Fewer bugs go to integration, which shortens integration time;
- Shorter integration time means faster release cycles;
- Fewer bugs go to customers, leading to happier customers;
- Happier customers lead to increased revenue and brand loyalty.

Finally, he said that software is the primary controller of the human interface and the majority and internal processing of most electronic devices, and the automobile is the largest and most complicated electronic device that any consumer will ever buy and improving software quality has the power to transform the reputation of automotive brands.

2.2.3 Case Studies

With the increasing interest in the literature on continuous practices, it is important to systematically review and synthesize the approaches, tools, challenges, and practices reported for adopting and implementing continuous practices. (Shahin et al., 2017) revise the state of the art of continuous practices to classify approaches and tools, identify challenges and practices in this regard, and identify the gaps for future research. Their paper also reveals that continuous practices have been successfully applied to both greenfield and maintenance projects.

The results showed that the research on continuous practices is gaining interest and attention from software engineering researchers and revealed that continuous practices can be successfully applied to both greenfield and maintenance projects.

Years ago, automated test generation has been suggested as a way of creating tests at a lower cost. Nonetheless, it is not very well studied how such tests compare to manually written ones in terms of cost and effectiveness. This is particularly true for industrial control software, where strict requirements on both specification-based testing and code coverage typically are met with rigorous manual testing. To address this issue, (Mateen et al., 2018) conducted a case study to compare manually and automatically created tests.

Their results showed, according to them, that automatically generated tests achieve similar code coverage as manually created tests, but in a fraction of the time (an average improvement of roughly 90%). They also found that the use of an automated test generation tool does not result in better fault detection in terms of mutation score compared to manual testing. Specifically, manual tests more effectively detect logical, timer and negation type of faults, compared to automatically generated tests. They defend that the results underscore the need to further study how manual testing is performed in industrial practice and the extent to which automated test generation can be used in the development of reliable systems.

Continuous integration and delivery consolidate several activities, and, during these activities, software professionals seek additional information to perform the task at hand. Developers that spend a considerable amount of time and effort to identify such information can be distracted from doing productive work. A better understanding of the information needs of software practitioners has several benefits, such as staying competitive, increasing awareness of the issues that can hinder a timely release, and building a visualization tool that can help practitioners to address their information needs. In this way, (Ahmad et al., 2021) conducted a multiple-case holistic study to identify information needs in continuous integration and delivery. This study attempts to capture the importance, frequency, required effort (e.g., sequence of actions required to collect information), current approach to handling, and associated stakeholders with respect to identified needs. There were identified developer's information needs and discussed whether the information needs were aligned with the tools used to address them.

As stated by (Ahmad et al., 2021), the data produced in the continuous integration and delivery can provide significant insights such as teams performance, possible bottlenecks, or areas for improvements, etc. After the study, they observed that the identified needs were not aligned with the tools that are used to address them. Also, several information needs cannot be addressed through available tools, requiring manual inspections, and thus taking more time. Information needs that are related to code and commit, confidence level, and testing are marked as most important and most frequently sought. This study

concluded that companies do not put enough effort into development resources for building in-house tools, thus spending a considerable amount of time selecting unsuitable outsourced tools or plugins.

With a specific context, (Gmeiner et al., 2015) defends that companies running an online business need to be able to frequently push new features and bug fixes from development into production. Successful high-performance online companies deliver code changes often several times per day. Their continuous delivery model supports the business needs of the online world. At the same time, however, such practices increase the risk of introducing quality issues and unwanted side effects. Rigorous test automation is therefore a key success factor for continuous delivery. In this paper, they describe how automated testing is used in the continuous delivery pipeline of an Austrian online business company. The paper illustrates the complex technical and organizational challenges involved and summarizes the lessons from more than six years of practical experience in establishing and maintaining an effective continuous delivery pipeline, as shows Table 1.

Table 1 - Lessons learned in test automation (Gmeiner et al., 2015)

Establishing the Pipeline	Operating the Pipeline
• Ensure management commitment • Take collective responsibility for the pipeline • Establish test environment management • Improve testability • Manage test data provisioning	• Define the ownership of acceptance tests • Acceptance tests should not compensate for missing unit tests • Invest in the maintenance of automated tests • Combine automated and manual testing

They reported several lessons learned from more than six years of practical experience. These lessons, as can be seen in Table 1, from which it is possible to highlight take collective responsibility for the pipeline for establishing the pipeline and combine automated and manual testing for operating the pipeline, include success factors that had a huge impact on the setup and operation of the pipeline. In the beginning, as they said, the main purpose of the continuous delivery pipeline was to get the features of the software system ready for user acceptance testing in the best possible quality. As stated by the authors, few pre-conditions must be satisfied before this step can become reality. First, a fully automated capacity and security testing must be in place, unit test coverage must be increased, and the coverage has to be ensured in the commit stage. In addition, the plans for future are also driven by the fast-changing online business.

2.2.4 Methodologies, Processes and Guidelines

Using the practice for introducing system test activities in the agile team activities every iteration, (Sandu & Salceanu, 2019) defend that organizations can improve the quality of the delivered products, reduce

costs, become more predictable, reduce time to market, increase sustainability of the development process, improve employee's morale, strengthen their image, and increase customer satisfaction. According to them, only by adopting and implementing continuous software process improvement imposed and requested as well by the standard automotive SPICE, OEMs and their subcontractors which produce software based electronic components for automotive industry, will increase quality by reducing the number of unresolved bugs, unleashing the power of autonomous agile teams.

Due to overcome the problem associated with the traditional load test methods, that are unable to identify production performance behavior because of simulates traffic patterns are highly deviated from production, (Arachchi & Perera, 2018) has presented an approach that has extended CI/CD pipeline to have three automation phases. That approach minimizes the system interruption by using test and scaling and it minimizes the system interruption by using test bench approach when system benchmarking and it uses the production traffic for load testing which gives more accurate results.

According to (Arachchi & Perera, 2018), the proposed extended feature on CI/CD pipeline, has achieved goals through test bench approach, it enables continuous benchmark of each software version and go-replay has given access to same live traffic on benchmark and load test phases. So, consistent with their results, test bench approach has proved that after scaling system could process same or better load with new development.

With the same point of view, (Poornalinga & Rajkumar, 2016) defend that automating the integration, deployment and delivery of software development is one of the key solutions for attaining productive growth in software industries whereby which the ideas of agile and DevOps can be turned into practical solutions. In their paper, they demonstrated a newly structured cloud infrastructure in which the complete automation of the continuous integration, deployment and delivery are done and experimentally tested.

Because of the importance of software quality, is significant to have a high-quality software on every project. CI mitigate the risks of software integration, improving its quality. In this way, (Hamdan & Alramouni, 2015) presents a framework that identifies software quality characteristics of the development process when applying the practices of continuous integration in the process of software projects development. Then, they construct a table with comparison criteria and the results before and after continuous integration, as shows Table 2.

Table 2 - Quality before and after applying CI (Hamdan & Alramouni, 2015)

Comparison criteria	Before continuous integration	After continuous integration
Time to develop	Developers work on the whole release requirements	Developers work on a single requirement
Introduced bugs	As testing is done only after finishing a large part of code, the number of bugs is greater than when testing is done after implementing only small parts of code	Every feature is tested and fixed after it is completed and it doesn't have to wait until the whole release is finished
Time to deliver	After finishing the release and testing it	After finishing a single requirement and testing it
Test quality	Testing is done to the releases as a whole Testing environment is not the same as the production environment	Testing a feature is done once it is integrated Small parts are tested individually and testing results are sent immediately back to the developers Testing environment is almost exactly the same as the production environment
Documentation	The release is documented properly The release specification is documented before starting the work And the development work is documented in detail after finishing	The requirements are documented But there is a minimal amount of documents by the developers The automatic tools generate statistics and data about the developed features based on the requirements, the testing results and the generated bugs etc.
Change management	Change is only accepted after a long process and approvals Changes are introduced through a whole new patch or release	Change is accepted at any time only by adding new requirements by the owner to the business analysis and software development teams
Cost model	The time and effort of manually doing the software build and integration	The cost of getting continuous integration server and tools

The results of the study are demonstrated through Table 2, where it is possible, through the different criteria, to compare before and after the continuous integration. According to the authors, the results showed that after using CI the project took less development time and less time to fix defects, so many of the risks involved in the software integration process were mitigated as a result of integrating parts of the developed software continuously once it is ready.

As long as continuous delivery has become popular in application software development and starting to be applied in safety-critical domains such as automotive industry, a safe analysis has to be updated with every change to ensure the safety test suite is still up to date. In this way, (Vost & Wagner, 2017) propose that a safety analysis should be treated no differently from other deliverables, propose guidelines to adopt this and formulate implications for development process.

They have identified safety analysis and testing as a crucial point for the wide-scale establishment of continuous delivery in industries producing safety-critical systems. Thus, they formulated guidelines for dealing with these issues in the form of continuous safety builds and proposed to treat results of a safety analysis no different than any other artifact. They said this requires a high amount of automatization and good tool support in iterative safety analysis and testing.

In that vein, recent practices of continuous delivery, which bring new content to the end user in days or hours rather than months or years, have generated a surge of industry-driven interest in the release engineering pipeline. (Adams & McIntosh, 2016) argues that the involvement of researchers is essential. While release engineering technology has flourished tremendously due to industry, empirical validation of best practices and the impact of the release engineering process on (amongst others) software quality is largely missing and provides major research opportunities. They provide some advice in form of a checklist about how release engineering can impact software engineering researchers who need to mine software repositories:

- Not all releases are equal – each release has its own characteristics.
- There are branches, and branches – most software projects have more than one branch open at a time.
- Choose before you build – not every file in a project goes into a release.

Specifically in the automotive area, due to the growth of software in modern cars and new releases, changes are required to the existing systems and the changes might have a positive and negative impact on the quality of the final product and software architects working with the changes often conduct impact analyses using metrics. In this way, to identify potential risk caused by these changes, (Durisic et al., 2011) present metrics used to measure the complexity and size of software systems. The metrics support early phases of software development and help reduce costly late changes and their main goal is to identify parts that should be tested more to reduce the risk of deteriorated quality.

In that paper, two metrics to support measuring the impact of changed were proposed - one for measuring complexity and one for measuring coupling of automotive software system design and architectural components. If applied regularly when significant architectural changes in the system need to be made, they can be used to verify certain non-functional properties of the system (such as maintainability and robustness) and indicate parts which should be tested more. They say that this increases overall product quality and decreases production cost, and the measures can be applied early in the development process, before sending requirements for specific component realization to the suppliers, in order to reduce the time and cost.

In the same domain, due to the increasing of complexity of the automobile, (Sandu & Salceanu, 2019) presented a methodology for reducing the number of open defects for automotive software development when agile methodology is used. They exemplified a method used to increase the system testing awareness into the agile teams. In order to have fast feedback and to solve the defects discovered during

development as soon as possible, they showed the benefits of implementing system test activities on team level. They presented comparative results for implementing system test activities outside and inside the agile team. In their words, they could easily identify that executing the system test activities inside the agile team significantly improved the delivered products and reduced the development time, effort, and costs.

Within the estimations, they defend that teams should consider the system test activities and should solve the defects discovered during the iteration execution in the same iteration. As part of the features and user stories development, the agile team should fix the critical defect before the working increment is delivered to the internal or external customer. If in isolated cases, from various reasons, the team can't fix these defects, then the affected feature will not be delivered and the activities for defects fixing will be planned for the next iteration.

Still in the automotive field, is known that electronic content is increasing in automobiles day by day, many functionalities are related to automobile system safety and safety is one of the key issues of future automobile development and risk of system failure is high due to increasing technological complexity and software content, so the software shall be tested well to arrest almost all the defects. (Barhate, 2015) explains a test case development and execution strategy based on practical implementation. It explains how test case reduction using Taguchi method, prioritization of test execution and automation help to make testing effective. It also demonstrates how maximum defects are discovered in short time.

The authors defend that the paper showed that test case prioritization logic demonstrated the effectiveness: 20% of defects discovered in 5% test cases within 6 hours of test execution. Also, according to the author, increasing testing depth as per increase in maturity of software and writing new test scenarios to avoid pesticide paradox led to huge number of new test cases and it was achieved due to automation in writing test cases.

In particular, with the increased volume of data, IoT applications need to adapt frequently and rapidly to new requests to increase competition and rapidly change market needs required from companies, more and more organizations are relying on new IT technologies and software development processes. (Guşeilă et al., 2019) proposes a process, methods, and tools for introducing automation for continuous integration, deployment, and testing in agile software development process of IoT applications, that is shown at Figure 6. Through the figure, it is possible to visualize the entire proposed pipeline, as well as highlight the use of tools such as SonarQube and Nexus to assist in some phases.

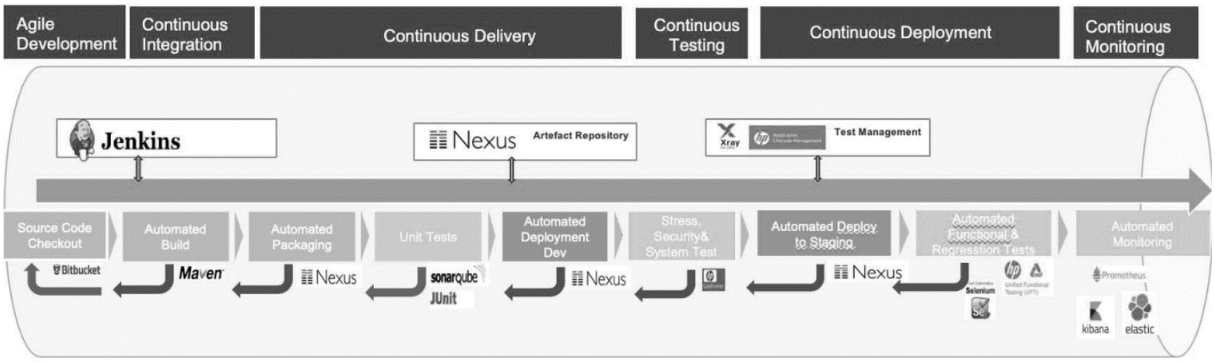


Figure 6 - Software Delivery Pipeline (Guşeilă et al., 2019)

According to the authors, the presented software platform for CI/CD along with the implemented modules for automated testing is in close alignment with the agile development methodology and practice, as it is possible to see in Figure 6. The approach presented for continuous testing, in their words, is providing a clear view on the quality of software delivered and supports the delivery of high quality, stable and robust products to the customer. They defend that the presented software deployment pipeline is an excellent platform for all mobile and web-based applications running in the cloud and supporting the interconnected devices to benefit of the next generation of the Internet, the Internet-of-Things.

In the same way, because of the necessity to defining the scaffold that is flexible enough to support multiple software products, but solid and durable when it comes to extensive usage in the context of continuous integration, (Khodiakova & Khodiakova, 2021) describes approaches that allow to implement a reusable CI infrastructure based on a chosen continuous integration system.

They defend it is possible to create a reusable CI infrastructure based on any software system with continuous integration support. The infrastructure itself and the process of its creation need to be robust and reliable for both infrastructure and development team. As mentioned by them, collaboration between these two departments improves the quality of the outcome, as opinion and experience exchange play major role in creating complex and distributed software and infrastructure systems. The principles of building a reusable CI infrastructure are valid for most of the popular solutions. As any system, the CI infrastructure will change with time adapting to the changes in organization structure, requirements of international standards or local regulations. The ability to grow, as they said, adapt and change is the most crucial feature of such system, as it will accompany the changes and development of related software products.

Due to the need of high-quality programming skills of college students and the increasing of continuous inspection paradigm utilization by developers on social coding sites, as an important method to ensure the internal quality of massive code contributions, (Lu et al., 2018) designed a specific continuous

inspection process for student's collaborative projects and conducted a controlled experiment to evaluate how the process affects their programming quality. The authors compared the CQIDs (Code Quality Issues Density) before and after adopting CI, as shows Figure 7.

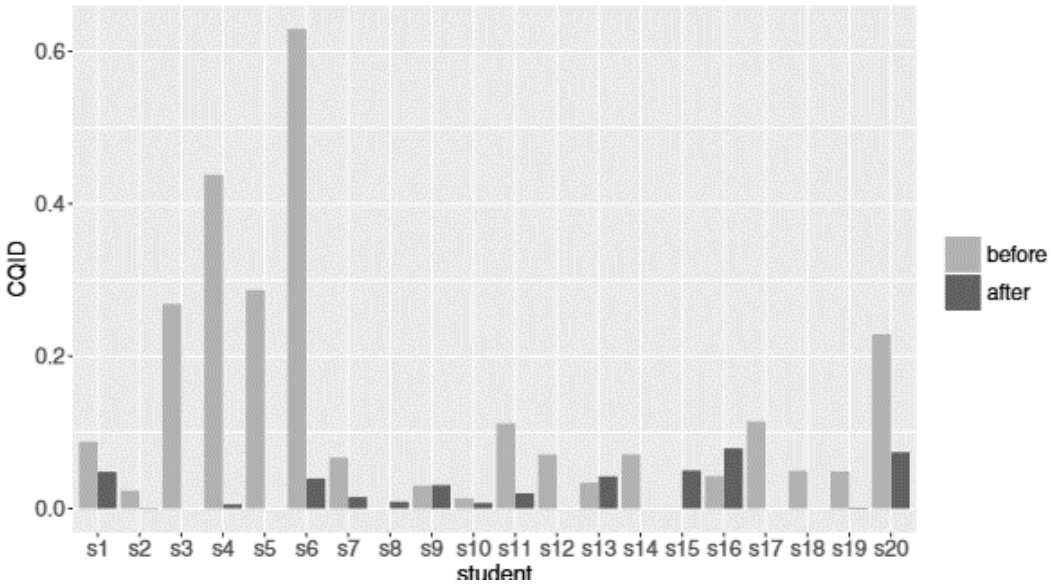


Figure 7 - Student's introduced CQIDs before and after adopting CI (Lu et al., 2018)

They defend that the results, verified in Figure 7, show that continuous inspection can significantly reduce the density of code quality issues introduced in the code per changed code line. As an example, it is possible to highlight student six, who showed significant improvements, according to the figure.

In turn, (Dakic et al., 2022) tried to determine the general indicators that are most often used in assessing the efficiency of sustainable production in companies to show the relationship between the financial element and the application of technologically innovative final products in the industry of autonomous vehicles. They defend that by producing new software components, it is necessary to meet the basic stability requirements without compromising the entire system with the best possible standards governing the area. In order to achieve this within continuous integration, continuous deployment (CI/CD) pipeline, it is necessary to define the stages and testing steps that are applied within the CI/CD environment. It is crucial to properly understand new ideas and the direction of technology development, which is accompanied by appropriate standards and processes.

As a conclusion of their work, authors defend that preservation of quality is achieved by various tools to improve software quality and, when creating final products, CI/CD development pipelines have been increasingly used in recent times. The idea of providing consumers with an equal experience is appealing in using a wide range of regional languages and many other innovations, setting new standards to be implemented in the automotive industry of autonomous vehicles.

As a consequence of the importance of CI in service lifecycle, due to demands of security and quality assurance at the stage of service deployment, (Lavriv et al., 2017) propose the infrastructure design and pipeline design for CI, which can be seen in Figure 8. The pipeline presented for continuous integration uses several tools, namely Jenkins as the core tool, SonarQube, PostgreSQL as a database, among others.

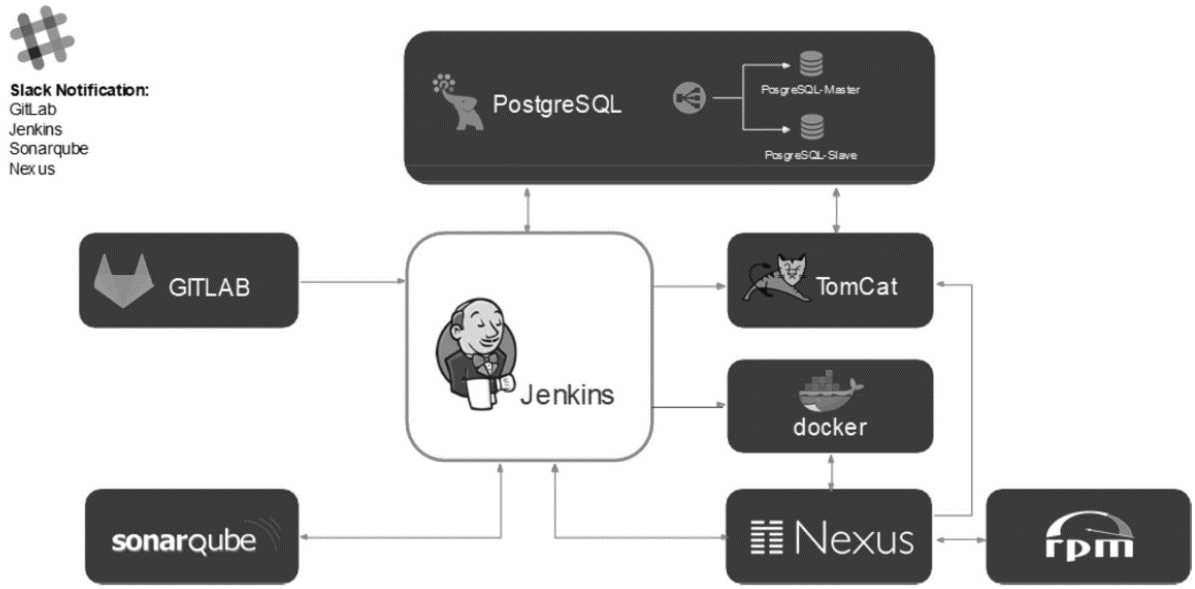


Figure 8 - Proposed CI pipeline (Lavriv et al., 2017)

According to the authors, the proposed scheme of Figure 8 is necessary to be implemented in modern information and communication systems because the service delivery includes service integration as well as service transferring to the end user. They need to consider the network related issues during the service integration process because build may be failed when code is written right but something went wrong in integration components interaction.

They defend that their analysis shows that the proposed pipeline providing a productive environment for the developing team to automate the build and deploy their code up to the production line and the automation system effectively helps in saving the time and cost by increased software quality and productivity. Furthermore, in their words, the complete automated pipeline worked magically over the testing and deployment process within expected short time and their report shows that the proposed infrastructure could provide more productive product in development for companies adopting to CI/CD based agile and DevOps practices.

Due to the problems associated with the adoption of continuous integration in industry because of social challenges, (Laukkanen et al., 2015) studied the adoption of continuous integration in a large-distributed organization.

In their case study, they found that a difficult architecture, lack of time, and the distribution of the organization can be the most influential factors hindering the adoption of CI. The authors give three guidelines to practitioners:

1. Understand that the product architecture has a significant effect on the adoption. However, do not let architectural problems keep you from implementing CI. You may adjust the architecture by changing technologies or components if needed.
2. Give enough time for the team members to overcome the initial learning phase. You must lower the priority of new features during the adoption.
3. Invest in the communication between different sites and avoid centralizing different competencies to certain sites. Understand the value of face-to-face communication and local support personnel when adopting CI.

Like this, (Ståhl & Mårtensson, 2021) present a set of under-reported challenges with continuous practices observed in multiple industry settings. Through a grounded theory approach, they construct the Tapco model — Test Automation Progression in Continuous Practices — identifying two distinct ways in which companies progress towards continuous delivery, into which the studied cases are mapped. The model is shown in Figure 9 and it is possible to see all the steps, in order by numbers or letters, for automation in the continuous practices.

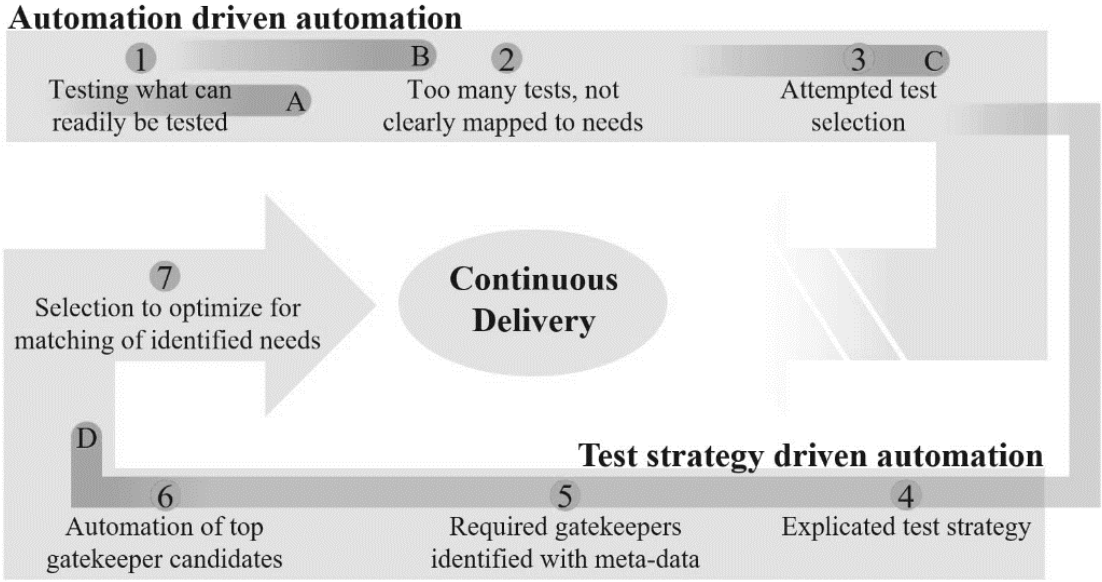


Figure 9 - Tapco model (Ståhl & Mårtensson, 2021)

According to them, they find that companies often approach automation as an end in itself (automation driven automation), rather than as a means to an end. By automating tests for automation's own sake

and starting where it is easy rather than planning for where it is hard, one can easily be led to forego certain capabilities when laying the groundwork for one's continuous delivery pipelines — capabilities the necessity of which only becomes evident further on, as requirements become increasingly demanding. As one nears the goal of reliable software releases in a continuous delivery fashion along this path, one may experience diminishing returns and increasing costs while never quite achieving the goal. On the other hand, companies may take a different approach (strategy driven automation), treating automation to achieve better testing, taking them closer to the goal of actual continuous delivery. By presenting the resulting model to a total of seven cases in multiple industries and collecting their reflections with regards to its accuracy, relevancy, understandability, and novelty, they defend that the model is a valuable contribution, serving as a guide to practitioners who strive at great effort and expense to realize the promises of continuous delivery.

2.2.5 Tools

Once software maintenance and testing are very expensive and time-consuming activities for developers, several researchers in the field of software engineering devoted their effort in conceiving tools for boosting developers' productivity during such development, maintenance, and testing tasks. In this way, (Panichella, 2018) discuss some empirical work he performed to understand the main socio-technical challenges developers face when joining a new software project.

As a result, he defends that recent research in software engineering observed an increasing adoption of summarization techniques for accomplishing simple or more complex, development, maintenance, and testing tasks.

With the same interpretation, (Deshpande et al., 2021) mentioned that test automation is the effective way to increase the test coverage and speed of the execution in software testing. As manual testing needs human effort and is prone to errors, CI is the best way to minimize risk. Software testing is becoming a compulsory process in assuring the quality of software product. Industries are facing challenges when developing software at different sites and testing on multiple platforms. Automating the build and the testing process is the best way to make continuous integration faster and more efficient. In this way, (Deshpande et al., 2021) discusses about the test automation execution setup, Jenkins's master and slave communication architecture, usage of various plugins available and test execution. They used Jenkins because they believe Jenkins is the most effective continuous integration and continuous delivery tool because it helps in automating the complete process, reducing the work of a developer and can

check the development at each and every step of software evolution. The results, according to the authors, show that the limitations of the Jenkins are it has lot of redundant plugins which sometimes are confusing to use the correct one.

In the same way, and because of the developed tools to facilitate such continuous activities, (Uzunbayir & Kurtel, 2018) identify, review, and reveal the characteristics of available tools to summarize their best features, as well as to identify which tools can be used for specific continuous software practices. They first identified the most popular tools according to their repository model to satisfy local, centralized, and distributed. Then, they explained how these models differ, and described them. To conform uncovering main characteristics of the tools, they identified 10 different characteristics, and presented a mapping between these characteristics and the tools, as it is possible to check through Table 3.

Table 3 - Comparison of source code management tools (Uzunbayir & Kurtel, 2018)

Tool	WBI	ONL	PLG	FRE	OPS	USG	DEV	COP	ENV	RPM
RCS			•	•	•	•	B	CI	UNIX	L
SCCS			•	•	•	•	B	CI	UNIX	L
AccuRev	•	•	•			•	I	CI/CD	Cross-Platform	C
Subversion	•	•	•	•	•	•	I	CI/CD	Cross-Platform	C
Autodesk Vault	•	•	•			•	I	CDE	Cross-Platform	C
CVS	•	•	•	•	•	•	B	CI	Cross-Platform	C
CA-Panvalet		•	•			•	B	CI	z/OS and z/VSE	C
Helix Core	•	•	•			•	I	CDE	Cross-Platform	C
IBM Rational Synergy	•	•	•			•	I	CI	Cross-Platform	C
Visual SourceSafe		•	•			•	I	CI	Windows	C
StarTeam	•	•	•			•	I	CI	Windows	C
PVCS		•	•			•	B	CI	Cross-Platform	C
TFS	•	+	•			•	I	CI/CD	Windows, Cross-Platform with VSTS services	C
VSTS	•	•	•			•	I	CI/CD	Windows, Cross-Platform with VSTS services	C
ArX	•	•	•	•	•	•	I	CI	Cross-Platform	D
BitKeeper	•	•	•	•	•	•	I	CI	Cross-Platform	D
Code Co-op		•	•				I	CI	Windows	D
Darcs	•	+	•	•	•	•	I	CI	Cross-Platform	D
Fossil	•	•	n/a	•	•	•	I	CI	Cross-Platform	D
GNU Bazaar	+	•	•	•	•	•	B	CI	Cross-Platform	D
Git	+	+	•	•	•	•	B	CI/CD	Cross-Platform	D
Mercurial		•	•	•	•	•	B	CI	Cross-Platform	D
Veracity	•	+	n/a	•	•		I	CI	Cross-Platform	D

In Table 3 it is possible to see the comparison that was made by the authors, who crossed the tools and the characteristics, and it is possible to see similar characteristics between some tools, such as ArX and BitKeeper. According to them, many tools were similar in their working mentality. However, they differed according to their characteristics, and they all support and enhance continuous software practices, but just five of them support CI/CD at the same time.

Similarly, due to the needs of maintaining a high quality of software releases and the costs associated, (Niæetin et al., 2018) argue that many open-source technologies and frameworks can be used in addition to field proven automotive development software so the total cost of the development, and time needed for testing and error checking can be reduced. Specifically, they extend the basic development process by using open-source build automation tools and servers to decrease the build and unit test time and incorporate code-based analysis tools for static and dynamic analysis.

In conclusion of their work, they defend that with servers for continuous integration, it is possible to decrease the time needed to test and integrate new changes into automotive software. In their words, with static analysis and dynamic analysis tools, it is possible to detect and document errors that can occur in automotive software and with unit testing frameworks, verify the validity of software on the unit and integration levels. So, they think it is shown that the approach is feasible and promising for more cost-effective development environments which are suitable for new companies and startups attempting to efficiently participate in the growing automotive industry.

So, automatic static analysis tools are tools that support automatic code quality evaluation of software systems. To shed light on the impact of such contexts on the warning's configuration, usage and adopted prioritization strategies of automatic static analysis tools, (Vassallo et al., 2018) surveyed developers and professionals in the area. In their words, the study highlights that 71% of developers do pay attention to different warning categories depending on the development context, and 63% of respondents rely on specific factors (e.g., team policies) when prioritizing warnings to fix during their programming.

According to them, their findings show that developers mainly use automatic static analysis tools in three different development contexts, i.e., local environment, code review and continuous integration, developers configure automatic static analysis tools at least once during a project, and although developers do not change configuration when working in different contexts, they assign different priorities to different warnings along the contexts.

Along the existing techniques, static analysis is a technique to identify and analyze software characteristics from source code, so it is possible to identify elements such as packages, classes, relationships, lines of code, bugs, complexity, coding violations and others. In this way, (Guaman et al., 2017) study, use as input the source code of the software applications written in different programming language for through static analysis identify metrics, characteristics, and technical debt with the aim to improve the quality when writing code. It also supports the identification of aspects such as correct apply of quality attributes, standards and best practices of programming that ensure the correct software development in terms of design and coding, using SonarQube.

Based in their results, they defend that is advisable use standards, quality models and best programming practices improve in maintainability, security, changeability, reliability, and testability as part of technical debt pyramid. In their words, the issues major and critical are associated to bag programming practices.

Software quality assurance is a sub-process that ensures that developed software meets and complies with defined or standardized quality specifications. Focusing on source code, there are characteristics

that can be used to evaluate the quality. From this perspective, (De Andrade Gomes et al., 2017) present a tool to source code quality evaluation aimed at supporting students to improve their source code and, consequently, their programming skills. The proposed tool uses quality reports (available to professional environment integrate with software repositories) to analyze students' source code and provide feedback about the student coding. They prepared a source code by introducing common defects, what decreases the quality of source code, and ask students to perform maintenance tasks to both eliminate the introduced defects and introduce new features. After each modification, students must evaluate their code using the proposed tool to obtain feedback about quality of source code.

The tool, named SMaRT, based on a well-known continuous inspection tool, aiming at supporting students to maintenance tasks – the analysis of source code allows the students to evaluate how the quality of the code present in their local repositories is, without compromises the project repository. The results, according to the authors, were positive to enhance the teaching-learning software quality assurance to the students.

With the context of a static analysis tool, (Bolduc, 2016) share the lessons learned during the integration of the static analysis tool Klocwork with the continuous integration system. In context, in order to detect defects faster, some analysis tools offer an integration with the integrated development environment of the developers at the cost of not always detecting all the issues. To detect defects earlier and still provide a reliable solution, one could think of running an analysis tool at every build of a continuous integration system.

The lessons learned, according to him, are:

- Focus on the process on new defects
- New defects management within the CI system should be driven by the types of defects sought out and the development process used
- Exploit more precise initial defect data
- Educate and engage the developers in the new process

They have discussed some lessons learned while integrating their static code analysis tool called Klocwork into their continuous integration system for the Klocwork development team at Rogue Wave Software. They put emphasis on focusing only on newly introduced defects, managing these defects depending on the types of defects that one is looking for, and on the development process used, exploiting more precise initial defect data, and educating and engaging the developers in the new process.

(Armenise, 2015) defends that the ongoing transformation in the software industry, which requires fast response to changes and thus, across team communication and collaboration, brought Jenkins to extend its functionalities to impersonate the role of orchestrator for all and different roles involved in the product lifecycle. The need of managing the whole continuous delivery pipeline required the implementation of new features in Jenkins, with the objective of facilitating the creation of complex workflows, allowing the traceability, reducing the time-to-market, and improving the productivity. In this way, (Armenise, 2015) illustrates how Jenkins evolved from being a pure continuous integration platform to a continuous delivery one, embracing the new design tendency where not only the build but also the release and the delivery process of the product is automated. In this scenario, Jenkins becomes the orchestrator tool for all the teams/roles involved in the software lifecycle, thanks to which development, quality assurance and operations teams can work closely together.

3. Solution Specifications

The goal of this chapter is to present the support solution's specifications. In this manner, the requirements imposed by the stakeholders were established after the programming and data exchange languages that would be examined. In addition, a market study of current tools and technologies was conducted. Finally, the tool was carefully selected based on the stated specifications.

3.1 Programming and Data Exchange Languages to be analyzed

The team's existing repository, which contains all the relevant software for a CI/CD process, both software developed by the team and third-party software, has several different programming and data exchange language files. From this complete list of files, it was selected with the stakeholders which ones are relevant to be inspected.

The relevant files to be analyzed are Groovy, PowerShell, JavaScript, Python, XML, CSS, HTML, JSON, YAML and IBM's Load rule[2] files. Markdown files don't need to be analyzed because they don't have a direct impact on the project. In the same way, GEN, STCFG and Configuration Source Files are third-party files, so they don't need analysis. Finally, NPM ignore files are references for source control, so it doesn't need analysis too. In a general way, the programming, and data exchange languages files relevant to be

[2] https://www.ibm.com/docs/en/elm/6.0.1?topic=build-component-load-rules

analyzed are the ones developed by the team and all the external libraries and third-party SW files are excluded. Also, all license files are third-party files, so they will be excluded too.

All this information led to Table 4, which demonstrates all the exclusions that are configured in SonarScanner properties file, "sonar-scanner.properties", with "sonar.exclusions" key, in order to narrow the focus of the analysis. For each exclusion, an ID and the motivation are explained.

Table 4 - Exclusions

ID	Exclusion	Motivation
1	**/*.npmignore	Exclude third-party files
2	**/*.**/*.conf	Exclude third-party files
3	**/*.stcfg	Exclude third-party files
4	**/*.gen	Exclude third-party files
5	**/*.md	Exclude third-party files
6	**/*.jar	Exclude third-party files
7	jenkins/jobs/Obsolete/**/*	Exclude obsolete directory
8	**/license.*	Exclude third-party files
9	**/.idea/**/*.*	Exclude third-party files

As an example, as can be seen in Table 4, the exclusion "jenkins/jobs/Obsolete/**/*" has the ID 7 and the motivation is to exclude the directory, because there are obsolete files that are save for a little period of time and, then, are deleted, so doesn't need to be included in the analysis since they take up memory and time to the system.

3.2 Requirements

To design the solution, some stakeholders impose requirements for the solution to guarantee that the implemented support solution meets all the needs. In this way, the requirements are presented in Table 5. For each requirement was assigned an ID, the category, and a description. The category is divided into non-functional (how the system should perform), functional (features the system should offer) and domain (resulting from the domain of operation).

Table 5 - Requirements

ID	Requirement category	Requirement description
1	Non-Functional	The solution must be easy to deploy. If the machine does not have the tool installed, there must be a script that installs and configures it on that machine.
2	Non-Functional	The platform shall provide means to enable/disable the configured Quality Checks and Quality Gates.
3	Non-Functional	It shall be possible to analyze a project locally on developer workstation and in a CI/CD Pipeline.
4	Non-Functional	The feedback must be provided in less than one minute.
5	Functional	The platform should perform continuous inspections of SW code by implementing predefined Quality Checks and Quality Gates in a CI/CD environment.
6	Functional	It must be possible to perform an analysis of the complete project, components, or individual files.
7	Functional	Feedback shall be provided upon analysis completion with reference to the generated report.
8	Functional	The platform shall report the projects' code quality and depict the trend over time by saving historical data.
9	Functional	It shall be possible to do two different analyses: one every time someone delivers the code and the other once per night.
10	Functional	The platform shall provide means to exclude files or directories from the analysis.
11	Domain	The platform shall be developed adhering to the existing architecture and have a command line tool that can be used by external tools.
12	Domain	The Platform shall be integrated into the CI/CD Platform of the Bosch product.

The support solution that will be implemented must be aligned with these requirements. As shown on Table 5, the requirement with ID 4, is a Non-Functional requirement and it says that "The feedback must be provided in less than one minute.", which means that the feedback of the inspection must be provided in less than one minute.

3.3 Tools and Technologies Analysis

In accordance with the requirements, a collection and analysis of tools and technologies that exist in the market for software code analysis and continuous inspection was completed, searching for the ones that support the relevant programming and data exchange languages. After a long search, analyzing comments and evaluations about each one in several separate places, reached a set of tools and technologies. This led to Table 6, which presents the technologies and tools and some features, pros, and cons of each one, allowing to define which one is closest to the needs.

The information used was accomplished by analyzing the websites of each platform, as well as some websites that have analyzed them and addresses only the most relevant information according to the defined requirements and the programming languages to be analyzed.

Table 6 - Tools and Technologies

Tool	Programming language	Features[3][4][5]	Pros[3][4][5]	Cons[3][4][5]
Codacy	CSS, Groovy, JavaScript, JSON, PowerShell, Python, XML	Code review Issue tracking Collaboration tools Issue management Security testing Static analysis	Easy to use Quick to set up See grade for project and individual files See progress of fixing the issues	Disconnects repository when not in use Does not have integration with continuous integration automation tools
Coverity	Python, JavaScript	API Multi-language scanning Reporting/Analytics Application Security Software Development	Details of existing and potential vulnerabilities Scalable Open source	Could improve the ease of use Need to be faster Not stable
Deepsource	Python, JavaScript	Static Analysis Security	Easy to setup and customizable Problems are sorted by type/severity Open source	Criticality of the vulnerability is overrated Only cosmetic issues
Embold	Java, JavaScript, Python, HTML, CSS	Code review and quality monitoring KPI Refactoring support	Clean and easy to use UI Good support Web integration Fast review	More data flow analysis Multi-Language repository not supported

[3] https://www.g2.com/categories
[4] https://www.getapp.com/
[5] https://www.peerspot.com/categories/application-security-testing-ast

Tool	Programming language	Features[345]	Pros[345]	Cons[345]
		Manage technical debt effectively Security and compliance Multi-dimensional scan		
Klocwork	JavaScript, Python	Security Standards Security vulnerability detection Bug, quality issue and code smell detection	Mature, robust, and helpful Integrates well with CI/CD	Confusing getting started Could improve the REST API Report created can be improved Web UI look and feels old
SonarQube	JavaScript, Python, CSS, HTML, XML, Groovy, YAML	Detect bugs and vulnerabilities Review security hotspots Track code smells and fix technical debt Code Quality metrics and history CI/CD integration API	Extensible, with 50+ community plugins Easy to navigate and configurate Quality gates Quick report Well integrated within the pipeline Quality of support Ease of use Open source Web API	Bad exporting capabilities
Veracode	JavaScript, Python	End-to-End Static Scanning Auto-Tuning Accuracy Prioritization and Remediation Reporting & Analytics	Realtime resolution Detailed report Easy usability Integrating into CI/CD process to help catch and resolve new flaws	Details on documentation Scan takes too long False positives

Table 6 allow to evaluate the tools by showing the programming languages that each one examines (from the ones defined in Programming and Data Exchange Languages to be analyzed) as well as the features, pros, and cons of each one. As example, Veracode has the capability of analyze JavaScript and Python and the features of that tool are, for example, the end-to-end static scanning, auto-tuning accuracy. The real-time resolution and easy usability are pros and the fact that the scan takes too long is a con.

3.4 Selected Tools and Technologies

Following the requirements and the files to be analyzed and looking for the Tools and Technologies analysis, SonarQube was the tool selected. SonarQube[6] is a self-managed, automatic code review tool that systematically helps deliver clean code. As a core element of the Sonar solution, SonarQube integrates into the existing workflow and detects issues in the code to help perform continuous code inspections of projects. The tool analyses more than 30 different programming languages and integrates into the CI pipeline and DevOps platform to ensure that the code meets high-quality standards. Due to these features, mostly because of the multi-language analysis support and for the capability to analyze almost all the languages needed, the Web API and the capacity for extension with plugins, SonarQube was the selected tool.

According to the requirement with ID 12, "The Platform shall be integrated in the CI/CD Platform of the Bosch product", the platform will be integrated with Jenkins's pipeline.

4. Solution Design

This chapter aims to present the support solution's design. It starts by showing the architecture, followed by a proof of concept of the solution. Then, the analysis types are described, and the design decisions are shown. Lastly, the defined quality gates are displayed.

4.1 Architecture

SonarQube and Jenkins are two tools that can be used together to manage and automate the software development process. SonarQube[6] is a self-managed, automatic code review tool that systematically helps developers deliver clean code, that integrates into the existing workflow and detects issues in code to help perform continuous code inspections of projects. In turn, Jenkins[7] is an open-source automation server which enables developers to reliably build, test, and deploy their software. In addition, SonarLint[8] was used to integrate the existing IDE workflow and helps commit with confidence, by detecting coding issues

[6] https://docs.sonarsource.com/sonarqube/9.7/
[7] https://www.jenkins.io/
[8] https://www.sonarsource.com/products/sonarlint/

in real-time and getting clear guidance on how to fix them. Then, a tool named CIAnalyzer Tool was created to keep the communication between SonarQube and Jenkins or another external tool.

The architecture of the support solution has gone through two different versions, one in an initial phase, and the other after the proof of concept and after some design decisions.

Figure 10 shows the first version of the architecture of the support solution, where Jenkins is the midway point control for the process, working as an orchestrator. The process starts when the developer produces software locally and, with the help of SonarLint, connected to the SonarQube Server and the specific project, a local and real-time analysis is performed. Then, the developer delivers the code to the team's repository, that triggers a build in Jenkins. In turn, Jenkins triggers the analysis in CIAnalyzer Tool, that starts a SonarQube analysis, which at the end returns the results and then the CIAnalyzer Tool constructs the report and returns it to Jenkins. Finally, Jenkins returns to the developer the build status, which depends on the quality gate result, as well as the link for the results and analysis report, which contains information about the analysis of that software.

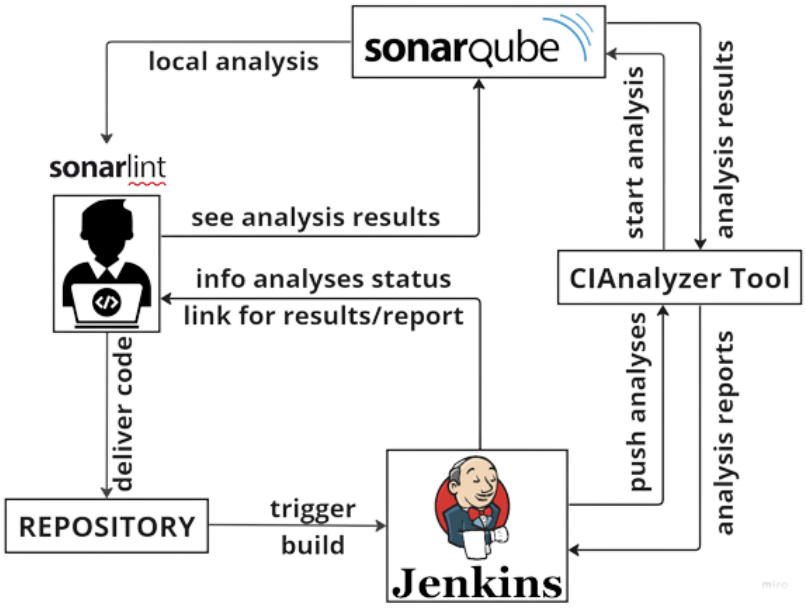


Figure 10 - Architecture First Version

Lately, it was realized that it makes sense to have two distinct analysis types, as shown 4.3 Analysis types, and, as a result, the architecture was updated to take this into consideration, by including the timer as other way to trigger Jenkins build. In this way, every night, a timer triggers a Jenkins build, and the analyzes are triggered. Additionally, the CIAnalyzer Tool was changed to make Json analysis, but it is not showed directly in the architecture, since it happens inside the CIAnalyzer Tool. In this case, the build

status depends on the quality gate result and the result of Json analysis. The last version of the architecture can be clarified with Figure 11.

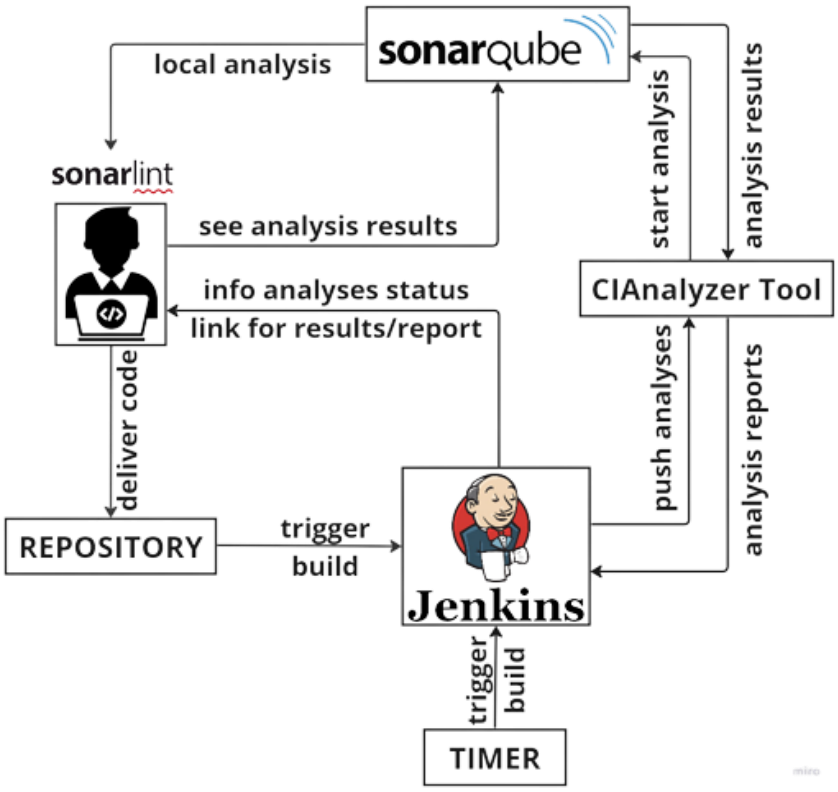


Figure 11 - Architecture Second Version

Figure 11 shows the last version of the support solution's architecture. It demonstrates how SonarQube and Jenkins can work together to provide and automate the development process of continuous inspection, by identifying code quality issues and then, improving software quality.

4.2 Proof of Concept

After defining the solution specifications, a small proof of concept was carried out. The proof of concept was used to test and validate the support solution and to identify potential issues or improvements earlier. For this, a section of the team's repository was analyzed, to explore the tools, understand the concepts, how it works, as well as its strengths and weaknesses.

In this sense, firstly SonarQube and SonarScanner were installed and configured on local machine, and the part of the repository was analyzed, allowing an initial analysis of the existing software. Firstly, initializing SonarQube provides a path to the server. Then, access the server and start using SonarQube GUI to create a new project, providing a name, a key and, after that, a token is generated. Following that,

SonarScanner was set up and initialized to do an analysis of the repository. Finally, the SonarScanner gives the path to access the results of the analysis, that can be seen in Figure 12.

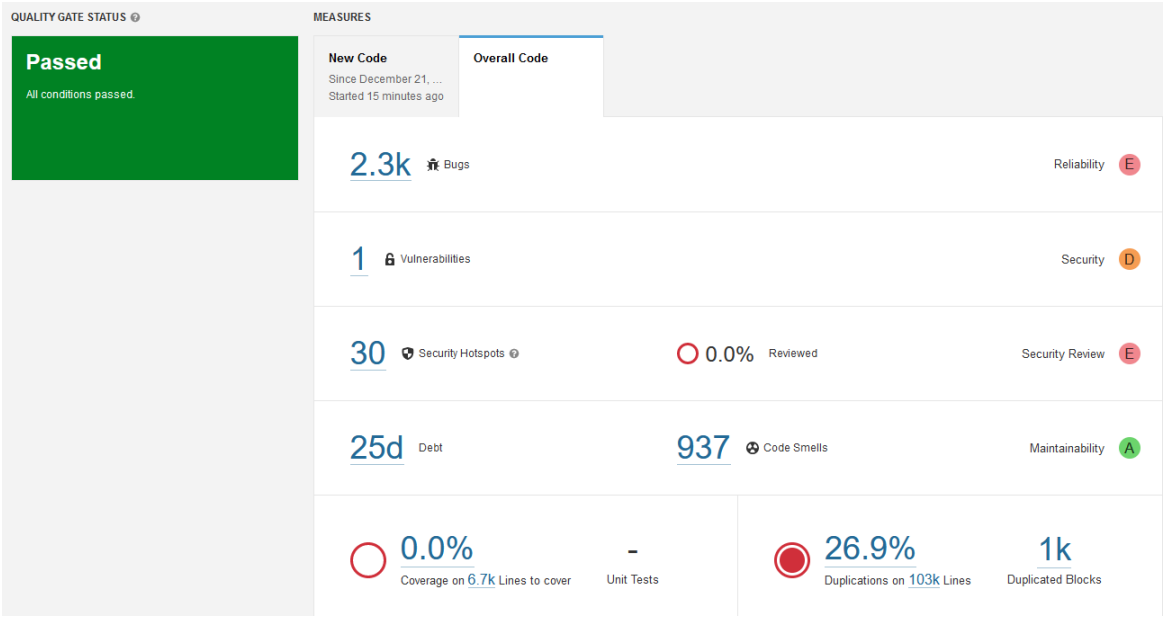


Figure 12 - SonarQube analysis result - Proof of Concept

Through Figure 12, in a very preliminary analysis with default quality profiles and quality gates, it possible to confirm that the existing software analyzed has 2.3k bugs, i.e., small errors can compound into larger ones, requiring software revisions and decreasing reliability. There also 30 security hotspots, which need to be reviewed but do not impact the entire project, one vulnerability, which should be changed as soon as possible because disturbs the security, and 937 code smells that, although they do not impact the project, they should not exist, since they are issues that make the code confusing and difficult to maintain. With this analysis it possible to have the information that there are 26.9% duplicate lines of code and that the time to fix all the bugs presented would be 25 days, a rather considerable number.

To meet requirement with ID 11, that says the platform should have a command line tool that can be used by external tools, a tool named CIAnalyzer Tool was developed and the SonarQube Web API methods were used. This tool is described with detail in 5.2 CIAnalyzer Tool.

After defining and testing the tool, SonarQube was integrated with Jenkins to automatically starts the analysis of the software that the user delivered, in order to keep the continuous inspection of software quality. In this way, a Jenkins Job was created with some pipeline stages, that are a block of code that defines a subset of tasks performed in the pipeline, to trigger the analysis and check if it passes or fails,

as it is showed in Figure 13. A Jenkins Job[9] is a sequential set of tasks that a user defines, a synonym to a project. The first stage "Loading sources" is where the last version of the team's repository is loaded to a folder, to be analyzed later, and the snapshot is created. In the second stage, "Checking for differences", the new snapshot is compared to the previous one, and it is checked if there are differences between them. Next, "Code analysis", is where the analysis starts and then the results are checked in "Gate status", to pass or fail the build and decide if the release will be created. If the analysis pass in all Quality Gate conditions, a release will be created in the last stage.

Figure 13 - Jenkins Job Proof of Concept

As it is possible to see in Figure 13, this build passed the analysis, and a release was created. After configuring SonarQube, creating the CIAnalyzer Tool and defining the pipeline, the proof of concept was complete.

In conclusion, the proof of concept was essential, because it allowed to identify issues and other potential problems associated with the support solution. So, through the proof of concept it was possible to check that some files were not analyzed by SonarQube. Groovy, YAML, PowerShell and JSON files cannot be parsed with the installed version of SonarQube, so three plugins were selected, for Groovy[10], YAML[11] and PowerShell[12], to complete the analysis of these files. However, because SonarQube does not support a built-in functionality to analyze the content of Json files, in the implementation phase a Python script was used to analyze these data. Initially, the idea was to analyze only the changed files but, after performing the proof of concept, it was concluded that it did not make sense, since changes made to one file can compromise other files, as it is detailed explained in Make two different analysis – Analysis content.

After the proof of concept, the support solution was presented to the team and the initial feedback from stakeholders was good, mostly because of the usability and adaptability of the support solution.

[9] https://anto.online/code/jenkins-an-introduction-to-jobs-and-projects/
[10] https://www.sonarplugins.com/groovy
[11] https://github.com/sbaudoin/sonar-yaml
[12] https://github.com/gretard/sonar-ps-plugin

Additionally, the proof of concept proved that the solution could be able to provide business value, by decreasing the time and resource costs associated with the need to review and modify the code, improving the overall quality of the software, enhancing the finished product, and demonstrating the effectiveness of the support solution. In this way, since the support solution has stakeholder support, business value since it appears to have the capacity to resolve the problem and the resources are available, it is appropriate to move forward with the implementation.

4.3 Analysis types

It was defined two analysis types: Full or Lightweight. Each one has distinctive characteristics, as it is possible to see in Table 7. In an initial phase, the goal is to ensure that software quality does not degrade, so that the team gets adapted to the solution, by applying quality gates that doesn't allow to add issues to the overall quality. Then, in a second phase, the goal is to implement more strict quality gates and improve software overall quality.

The team determined that the first phase should take around two weeks in order to provide time for adaptation. After that, the second phase is going to happen, and the analysis will be more restrict.

Table 7 - Analysis Types

Analysis type	Characteristics
Full	Build is triggered by timer, once per night.
	CIAnalyzer Tool analyzes all Json files and ask SonarQube to analyze all the other repository files.
	Quality Gate focuses on Overall Code.
	Stage "Checking for Differences" and "Create release" are skipped.
Lightweight	Build is triggered when someone delivers to the stream.
	CIAnalyzer Tool analyzes just the modified Json files and ask SonarQube to analyze all the other repository files.
	Quality Gate focuses on New Code.
	Stage "Create Release" is not done if the repository fails the analysis.
	Build stops if there are no changes between the actual and the previous snapshot.

The Full analysis, as shows Table 7, will be triggered by timer once per night and will focus on overall code in order to assess the overall status of the repository. The Lightweight analysis will be focused on New Code and triggered every time that one member of the team delivers something to the stream, thus, in order to create the release, the given code must pass the analysis. The goal of Lightweight analysis is that New Code delivered cannot compromise the software quality so if the delivered code has any issue it will fail.

4.4 Design Decisions

4.4.1 Manage Quality Checks and Quality Gates

Requirement: The platform shall provide means to enable/disable the configured Quality Checks and Quality Gates.

Solutions: For this requirement, two potential solutions have been found.

Solution 1 – SonarQube GUI

SonarQube GUI may be accessed by using the admin credentials and, through the Quality Profile section, enable or disable a quality profile from a project and create or remove a quality profile. Through the Quality Gate section, the user can create or destroy a quality gate, add, or remove gate conditions and enable or disable quality gates from projects, as it is represented in Figure 14.

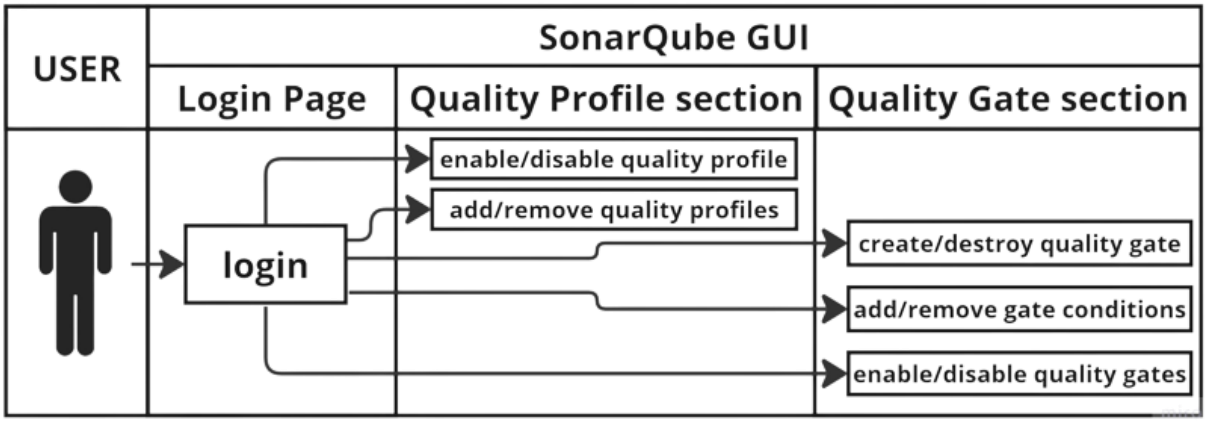


Figure 14 - Solution 1 Manage Quality Checks and Quality Gates

Solution 2 – CIAnalyzer Tool

The user can access the CIAnalyzer Tool and enter the commands to deselect, select, create, destroy, and create or delete conditions from the quality gates. The commands will call the SonarQube Web API

methods, that will enable or disable, create, or destroy and add or remove the gate conditions, depending on the command. This solution can be clarified through Figure 15.

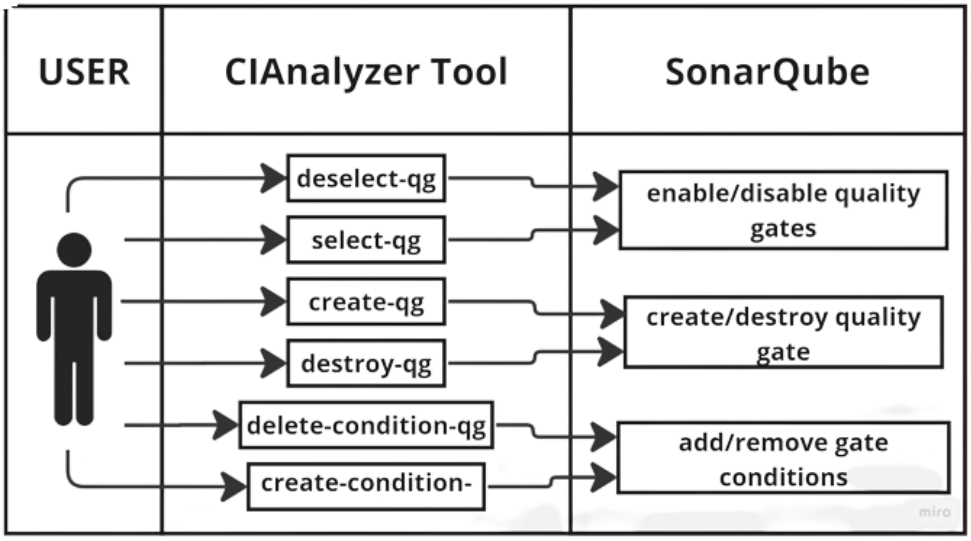


Figure 15 - Solution 2 Manage Quality Checks and Quality Gates

Decision:

As it is possible to see in Figure 14, solution 1, using SonarQube GUI, is more intuitive and clearer to enable/disable several Quality Checks and Quality Gates configurations. Using Solution 2, Figure 15, although it is possible, it is not intuitive enough, the user will be more confused, and it takes additional effort to do that. In this way, solution 1 was the decided one.

4.4.2 Exclude files or directories from analyses

Requirement: The platform shall provide means to exclude files or directories from SonarQube analysis.

Solutions: For this requirement, two potential solutions have been found.

Solution 1 – Sonar-Scanner configuration file

By adding the path from files or directories to be excluded, in "sonar.exclusions" property, these files or directories will be excluded in the next analysis. This should be added directly by the user in "%SONAR_HOME%\conf\sonar-scanner.properties" file, as it is possible to see in Figure 16.

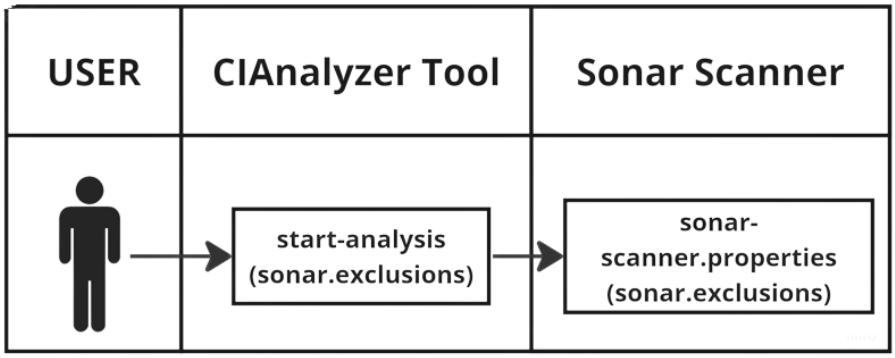


Figure 16 - Solution 1 Exclude files or directories from analysis

Solution 2 – CIAnalyzer Tool

When the user invoke "start-analysis" command from the CIAnalyzer Tool, he should configure "sonar.exclusions" argument, that will be configured in sonar-scanner, as demonstrated in Figure 17. The CIAnalyzer Tool is not capable of doing that, so it will take time and effort to implement this functionality.

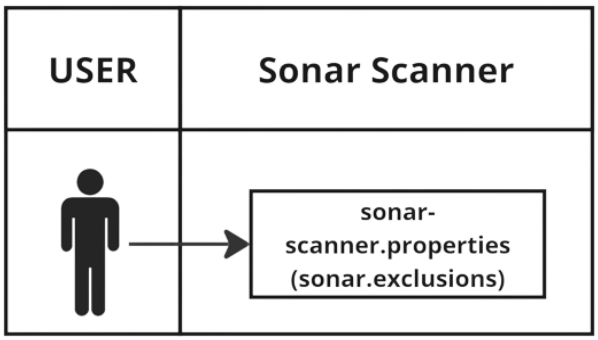


Figure 17 - Solution 2 Exclude files or directories from analysis

Decision:

Through Figure 16, it is possible to check that solution 1, changing it directly in sonar-scanner configuration file, is a more intuitive and explicit approach to add "sonar.exclusions" property, because it is less error-prone. Using Solution 2, shown in Figure 17, will require additional configurations and effort, what was considered unnecessary. In this way, solution 1 was the decided one.

4.4.3 Make two different analysis – Job definition

Requirement: It shall be possible to do two different analyses: one every time someone delivers the code and the other once per night.

Solutions: For this requirement, two potential solutions have been found.

Solution 1 – Using the same Jenkins Job

Using the same Jenkins Job for the two analysis types. When the build starts, it is verified if it was triggered by a timer or a user. If the build is triggered by User Id, it means it is a Light analysis, as it is shown in Figure 18.

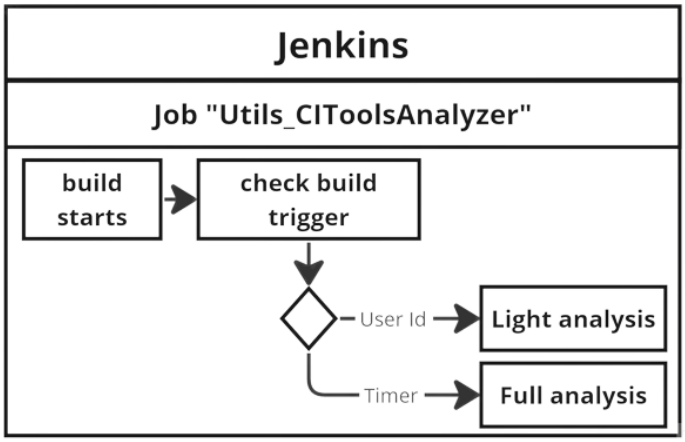


Figure 18 - Solution 1 Make two different analysis – Job definition

Solution 2 – Using two different Jenkins Jobs

Using two different Jenkins Jobs, one for each analysis type. The Job "Utils_CIToolsAnalyzer" is for the Lightweight analysis and the Job "Utils_CIToolsAnalyzer_Nightly" is for Full analysis. In Figure 19 and Figure 20 it is possible to clarify this solution.

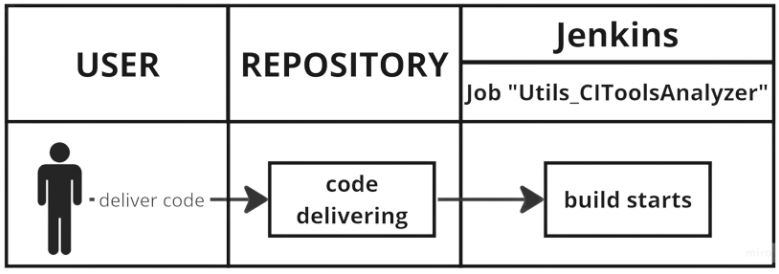


Figure 19 - Solution 2 Make two different analysis – Job "Utils_CIToolsAnalyzer"

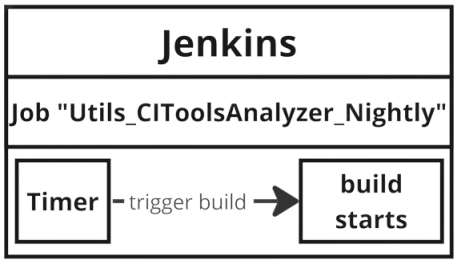


Figure 20 - Solution 2 Make two different analysis – Job "Utils_CIToolsAnalyzer_Nightly"

Decision:

Solution 1, using the same job, like it is showed in Figure 18, since "Full analysis" type does not create release, it will not allow to create the next release (when it is a Lightweight analysis) properly, because the last successful build doesn't have a snapshot (because the snapshot is not available when the release is not created). Using Solution 2, in Figure 19 and Figure 20, allows to separate the two analysis types, enabling the correct release creation, so it was chosen.

4.4.4 Make two different analysis – Analysis content

Requirement: It shall be possible to do two different analyses: one every time someone delivers the code and the other once per night.

Solutions: For this requirement, two potential solutions have been found.

Solution 1 – Analyze just the changes

When a user delivers software, the differences between the actual and last snapshots are compared to see whether there are any differences. After that, the changed files will be copied from a folder and only the content of this folder will be analyzed by SonarQube, as it is possible to see in Figure 21.

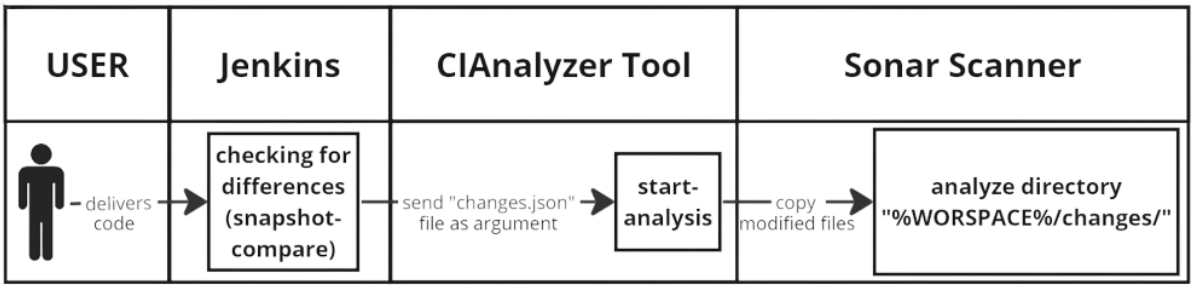


Figure 21 - Solution 1 Make two different analysis – Analysis content

Solution 2 – Analyze all repository

When a user delivers software, the differences between the actual and last snapshots are compared to see whether there are any differences. After that, the changed files are copied from a folder and, if there are just Json files, the content of this folder is analyzed. If there are other files, all repository is analyzed by SonarQube as can be clarified through Figure 22.

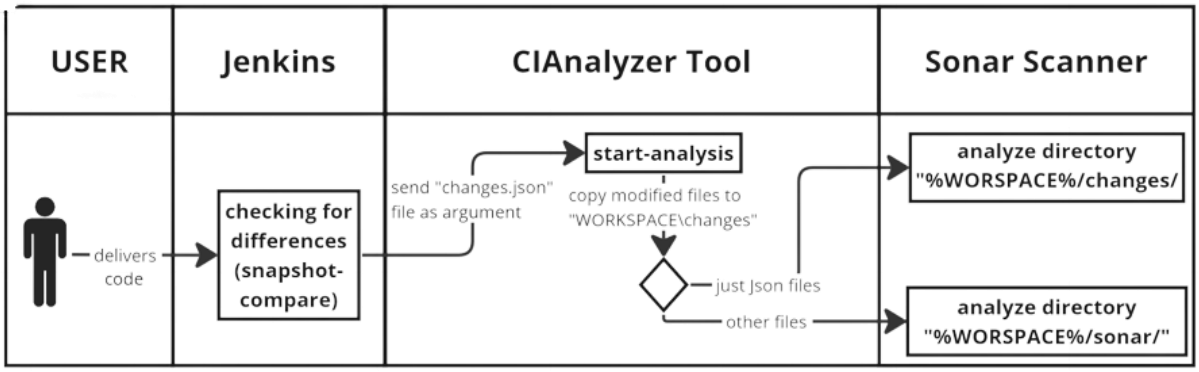


Figure 22 - Solution 2 Make two different analysis – Analysis content

Decision:

Using solution 1, as it is possible to see through Figure 21, analyze just the changes could compromise the veracity of the analysis, since changes made to one file could damage other files. For example, changes in one method can compromise other files that use that method. Thus, by analyzing only the files with changes detected, these issues will not be detected, since SonarQube community version does not allow pull requests analysis. In this way, the decision was to always analyze the entire code, focusing on new code or overall code depending on the analysis type, as shows solution 2, in Figure 22.

4.4.5 Analysis report and history

Requirement: The platform shall report the project's code quality and depict the trend over time by saving historical data.

Solutions: For this requirement, two potential solutions have been found.

Solution 1 – Using SonarQube GUI and the same project

By using SonarQube GUI and the same project for the two analysis types, it is possible to access the project history. The report for each analysis is sent to Jenkins, developed through the SonarQube Web API. This can be seen through Figure 23. Using the same project could be more confused because the results of the two analysis types will be showed in the same trend chart. Due to the fact that all analyses, even unsuccessful ones, will be displayed on the chart and won't be able to see the repository's current status.

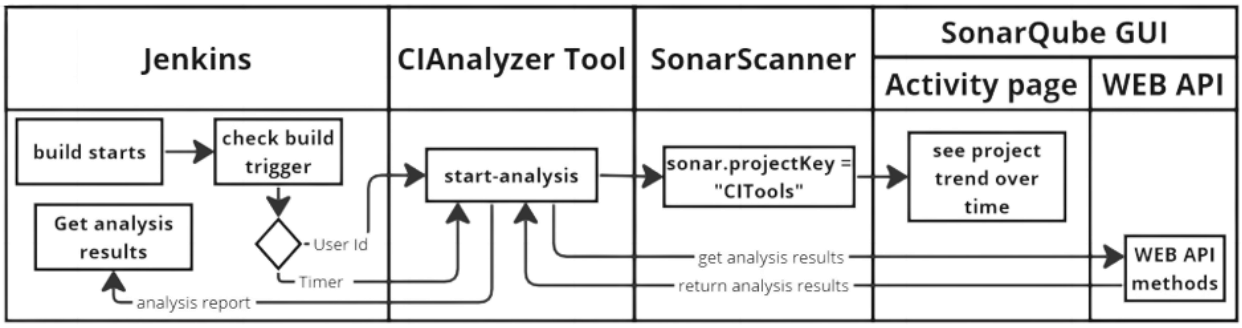


Figure 23 - Solution 1 Analysis report and history

Solution 2 – Using SonarQube GUI and different projects

By using SonarQube GUI and a different project for each analysis type, it is possible to see the history of each project, in the Activity page. The report for each analysis is sent to Jenkins, developed through the SonarQube Web API, as it is shown in Figure 24. Using different projects will provide a clearer and more precise view of the history and trend over time, because in the project "CITools" it is possible to see all analysis (failed or passed) and, with project "CITools_Nightly" it is possible to see the effective tend over time, because it runs once per night and aggregates all the results.

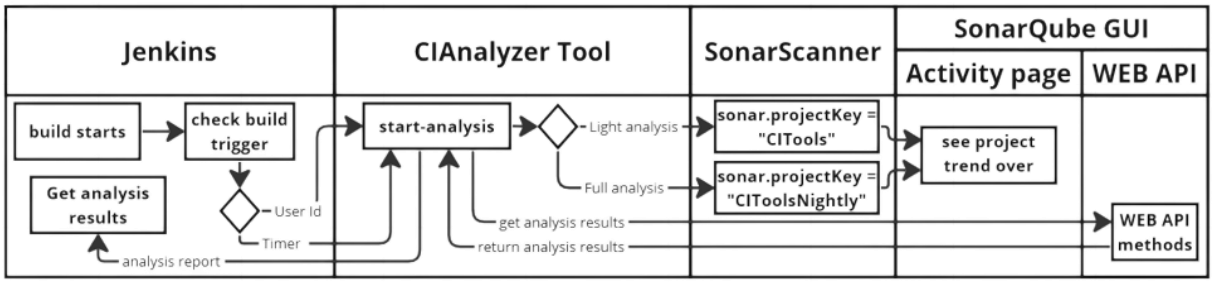


Figure 24 - Solution 2 Analysis report and history

Decision:

As it is showed in Figure 23, solution 1, using the same project, will not allow to see the history with precision, because the two analyses will run in the same project. With solution 2 it is possible to have more accuracy results and to see the history in a clearer way, so it is the chosen solution.

4.5 Quality Gates Definition

To check if the analysis passes or fails, SonarQube has many quality gates that can be implemented in the project and has the option to create new ones, according to project needs. In this way, a careful and thorough investigation of existing quality gates on SonarQube was carried out, searching for the most

appropriate according to the goals and specifications of the solution. After that, two different quality gates were created, one for the Lightweight analysis, and another for the Full analysis.

The Quality Gates defined for the project are shown in Table 8. The Quality Gate called "Commit" is for the Lightweight analysis and the other, with the name "Full" is for the Full analysis. For each analysis, some conditions were selected and, for each condition, the measure, the focus on new code or overall code, the operator and the value are shown.

The Quality Gates were defined for the first phase, that has the goal to not degrade de existing software quality, in Table 8, and for a second phase, in Table 9, with the objective to improve the existing software quality.

Table 8 - Quality gates – First Phase

Name	Measure	New Code/Overall Code	Operator	Value
Commit	Blocker Issues	on new code	is greater than	0
	Critical Issues	on new code	is greater than	0
	Major Issues	on new code	is greater than	0
	Minor Issues	on new code	is greater than	0
	Bugs	on new code	is greater than	0
Full	Maintainability Rating	on overall code	Is worse than	A
	Security Rating	on overall code	Is worse than	C
	Reliability Rating	on overall code	Is worse than	D

As showed in Table 8, in the Lightweight analysis, the developers cannot add more blocker, critical, major, and minor issues, as well as bugs. Because these are the starting values from the initial analysis and should stay the same or get better, the maintainability rating cannot be worse than A, the security rating cannot be worse than C, and the reliability rating cannot be worse than D in the full analysis.

In SonarQube, blocker issues are bugs with high probability to impact the behavior, critical issues are bugs with a low probability to impact the behavior or an issue that represents a security flaw and a major issue is a quality flaw that can highly impact the developer's productivity. Additionally, a minor issue is a quality flaw that can slightly impact the developer's productivity. Finally, a bug is a coding mistake that can lead to an error or unexpected behavior at runtime. The maintainability rating is related to code smells, that are issues that make the code confusing and difficult to maintain and the security rating is related to vulnerabilities, that are points in the code open to attack.

In a second phase, with the goal of improving the existing software quality, the Quality Gates for the Full analysis were modified to be stricter, as represented in Table 9.

Table 9 - Quality Gates – Second Phase

Name	Measure	New Code/Overall Code	Operator	Value
Commit	Blocker Issues	on new code	is greater than	0
	Critical Issues	on new code	is greater than	0
	Major Issues	on new code	is greater than	0
	Minor Issues	on new code	is greater than	0
	Bugs	on new code	is greater than	0
Full	Maintainability Rating	on overall code	Is worse than	A
	Security Rating	on overall code	Is worse than	A
	Reliability Rating	on overall code	Is worse than	A

The conditions for the second phase were upgraded to be more restricted than the first phase, so for the Full analysis, the conditions required the security and reliability rating to be better. The conditions for the Commit quality gate, for the Light analysis, are the same as the ones in the first phase because these conditions do not allow to add more issues to the code.

5. Solution Implementation

After understanding the most relevant concepts and searching for research related to the study areas covered, defining the specifications, and designing the solution, the implementation of the solution was done. This phase is a crucial step in the overall project lifecycle because it is where the defined plan is put into action and the solution is built and deployed. In this way, this chapter details the implementation, according to the design and solution specifications.

5.1 SonarQube

5.1.1 SonarQube configuration

The implementation started with the configuration of SonarQube. Firstly, the SonarQube Server and SonarScanner were installed on a virtual machine, and the server was configurated by defining the "sonar.web.port = port" and the "sonar.web.host = webhost", with the correct values. There was created an environment variable called "SONAR_HOME" with the path to SonarQube folder.

After that, the plugins for Groovy, PowerShell and YAML were added to "%SONAR_HOME%\extensions\plugins" folder, by downloading it. Then, SonarQube was defined as a service in Windows, by installing and starting the "SonarService.bat", to keep the server up all the time, using the commands "%SONAR_HOME%\bin\windows-x86-64\SonarService.bat install" and "%SONAR_HOME%\bin\windows-x86-64\SonarService.bat start" and SonarQube was ready to be accessed, trough the URL "webhost:port". Then, a token was created in SonarQube to authenticate with SonarQube through the CIAnalyzer Tool.

To meet the third requirement, "It shall be possible to analyze a project locally on developer workstation and in a CI/CD Pipeline", SonarLint was chosen for the local analysis and Jenkins for the CI/CD Pipeline. SonarLint is a plugin that connects with the SonarQube server and provides a real-time analysis in the IDE, with the same quality profiles implemented in the specific project. All team members added the plugin to IntelliJ, the IDE used by the team, and establish the connection by configuring the server, the token and associating a SonarQube project. To the CI/CD Pipeline, a Job was created in Jenkins in order to analyze the software code and return the results, what can be seen in more detail in 5.3 Jenkins.

SonarQube provides a Dashboard with the result of the analysis, based on the Quality Profiles and Quality Gates implemented for the project. As an example, Figure 25 shows the dashboard for the project "Test9", a test project that analyzes one part of the repository with predefined Quality Gates.

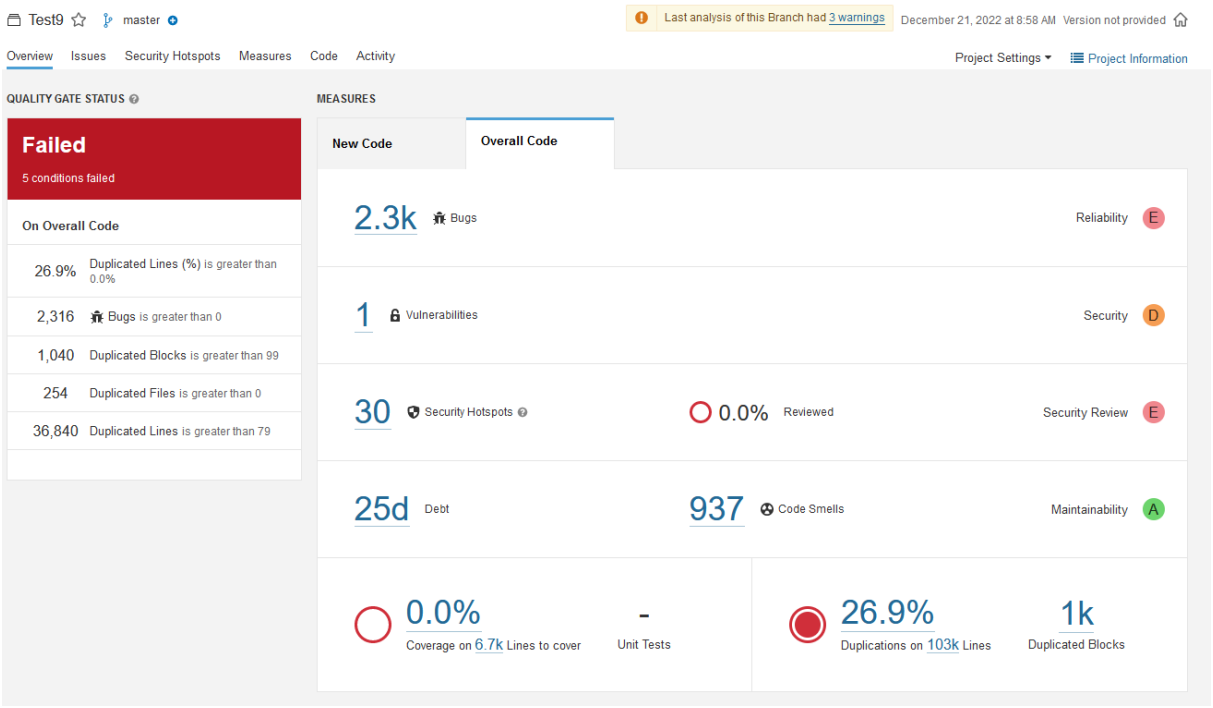


Figure 25 - SonarQube Dashboard

As it is possible to see through Figure 25, this project failed the analysis due to the failure of five quality gate conditions. The results of the SonarQube analysis to the complete team's repository can be seen with detail in 7.1 First analysis.

5.1.2 Quality Profiles

The Quality Profiles implemented for the SonarQube analysis have all existing rules, except the deprecated ones, for each language, since every rule appeared to be significant and necessary. In this way, Table 10 shows the Quality Profiles used for the SonarQube analysis, with the name given in SonarQube, language, and the number of rules.

Table 10 - Quality Profiles

Quality Profile Name	Language	Number of Rules
CSS	CSS	25
Groovy	Groovy	384
HTML	HTML	58
JS	JavaScript	258
PS	PowerShell	65
Python	Python	209
YAML Analyzer	YAML	19

For example, for the JavaScript files, the quality profile named “JS”, which has 65 rules, is used, as shows Table 10.

5.1.3 SonarQube issues

SonarQube divides the issues into five types: critical, blocker, major, minor and info. Some examples of issues reported in the analysis are represented in the figures below. In Figure 26 it is possible to see an example of a blocker issue in Python, that is a code smell and needs 2 minutes of effort to solve it. There is a good chance that this problem will affect how the application behaves.

Figure 26 – Example of Blocker Issue

In Figure 27 is showed an example of a critical issue, that has a low probability to impact the behavior of the application and will take 2 minutes to resolve it. This issue appears in a JavaScript file.

Figure 27 – Example of Critical Issue

Figure 28 shows an example of a major issue. In this case also a code smell in PowerShell language. This issue represents a quality flaw that can highly impact the productivity of the team.

Figure 28 – Example of Major Issue

Figure 29 has an example of a minor issue in HTML, that is also a quality flaw that can slightly impact the team’s productivity and is a code smell because level H1 is skipped.

Figure 29 – Example of Minor Issue

Finally, Figure 30 shows an example of an info issue in Python, that is also a code smell and needs 0 minutes of effort to solve it.

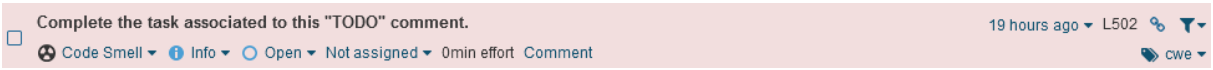


Figure 30 – Example of Info Issue

5.2 CIAnalyzer Tool

According to the requirement with ID 11 "The platform shall be developed adhering to the existing architecture and have a Command Line Tool that can be used by external tools", a tool called CIAnalyzer Tool was implemented, to be used by external tools. This tool is using the Web API of SonarQube, and, in this way, some methods were created in the CIAnalyzer Tool to manage the analysis, like create project, start analysis, select quality gate and other ones. To develop the CIAnalyzer Tool, Python was used to receive the arguments and call each chosen Web API method, acting as an intermediary between SonarQube and the user or other external tool, and have other methods to manage the analysis.

When calling each command, the user is required to enter the URL of the server, the authentication token, the path to the configuration file and the exceptions file as arguments as it is possible to see in Figure 31.

```
Options:
  -s, --server TEXT      Server web host  [required]
  -u, --user TEXT        User token  [required]
  -c, --conf TEXT        Configuration file  [required]
  -e, --exceptions TEXT  Exceptions file  [required]
  --help                 Show this message and exit.
```

Figure 31 – CIAnalyzer Tool options

Figure 31 shows the CIAnalyzer Tool options, so it is needed to do something like this before the specific command:

```
static-analysis-tool -s localhost:8080 -u sq2527sg26 -c C:/Desktop/config.json -e C:/Desktop/excep.json
```

The CIAnalyzer Tool has many features which provide several options to configure SonarQube information, like create projects, start analysis, among others. Figure 32 shows all the commands and their description and, to call each command, some arguments will be required to be provided.

```
Commands:
  create-analysis-event   To create a project analysis event
  create-condition-qg     To add a new condition to a Quality Gate
  create-project          To create a new project
  create-qg               To create a Quality Gate
  define-branch           To search a project analyses
  delete-condition-qg     To delete a condition from a Quality Gate
  delete-project          To delete a project
  deselect-qg             To remove the association of a project from a gate
  destroy-qg              To delete a Quality Gate
  get-conditions-qg       To display the details of a Quality Gate
  get-project-analysis    To search a project analyses
  get-project-components  To get the components of a project
  get-project-qg          To get the quality gate of a project
  get-project-qg-status   To get the quality gate status of a project
  get-projects            To search for projects
  get-qg                  To get a list of existing Quality Gates
  get-qg-projects         To search for projects associated (or not) to a gate
  select-qg               To associate a project to a quality gate
  start-analysis          To start a new analysis
  update-analysis-event   To start a new analysis
```

Figure 32 - CIAnalyzer Tool commands

Figure 32 displays the features of the tool and each one has different arguments and goals. In Table 11 it is possible to see the details about each command. For each one, the command, description, arguments, and an example are detailed. The necessary arguments are shown in bold type in the argument's column.

Table 11 - CIAnalyzer Tool Commands Information

Command	Description	Arguments	Example
create-analysis-event	To create a project analysis event	**-n analysis name** **-i analysis id** -c category (OTHER/VERSION)	create-analysis-event -n newanalysis -i 152 -c VERSION
create-condition-qg	To create a new Quality Gate condition	**-e error value** **-m metric** **-n quality gate name** -o operator (LT/GT)	create-condition-qg -e 2 -m bugs -n commit -p gt
create-project	To create a new project	**-p project key** -v visibility (private/public)	create-project -p key -v public
create-qg	To create a new quality gate	**-n quality gate name**	create-qg -n light
define-branch	To define an analysis as the branch	-p project key **-t type** -v value (analysis id, number of days, string)	define-branch -p key -t SPECIFIC_ANALYSIS -v 27163
delete-condition-qg	To delete a quality gate condition	**-i quality gate id**	delete-condition-qg -i 5262
delete-project	To delete a project	**-p project key**	delete-project -p key

Command	Description	Arguments	Example
deselect-qg	To deselect a quality gate from a project	**-n quality gate name** **-p project key**	deselect-qg -n light -p key
destroy-qg	To delete a quality gate	**-n quality gate name**	destroy-qg -n light
get-conditions-qg	To display the conditions of a quality gate	**-n quality gate name**	get-conditions-qg -n light
get-project-analysis	To display the details of project analysis	**-p project key** -a after date -b before date	get-project-analysis -p key -a 2022-05-01 -b 2023-03-22
get-project-components	To get the components of a project	**-p project key** -o order (ascending sort) -l limit	get-project-components -p key -o yes -l sonar
get-project-qg	To get the quality gate associated to a project	**-p project key**	get-project-qg -p key
get-project-qg-status	To get the status of the project quality gate	**-p project key** -a analysis id	get-project-qg-status -p key -a 152
get-projects	To get a list of existing projects	-d older than given date -p project keys -l limit	get-projects -d 2022-02-21 -p key,test -l sonar
get-qg	To get a list of existing quality gates	_____	get-qg
get-qg-projects	To get projects associated to a quality gate	**-n quality gate name** -l limit -s selected	get-qg-projects -n light -l sonar -s yes
select-qg	To associate a project to a quality gate	**-n quality gate name** **-p project key**	select-qg -n light -p key
start-analysis	To start a new analysis	**-p project-key** **-d directory for analysis** **-t type (Light/Full)** -c path to changes file -j job name -b build number	start-analysis -p key -d C:/Desktop/project/ -t Commit -c C:/Desktop/changes.json -j Utils_CIAnalyzer -b 451
update-analysis-event	To update an analysis event	**-e event id** **-n event name**	update-analysis-event -e 16627 -n newevent

Table 11 explains how to use each command. For instance, the command “delete-project” may be used to remove a project by using “delete-project -p key”, which asks for the key to the project to be deleted. For this example, a complete command will be something like that:

```
static-analysis-tool -s localhost:8080 -u sq2527sg26 -c config.json -e excep.json delete-project -p key
```

This command will call the “delete_project” function, that receive the project “key” and send it as an argument for the function “delete_project_action”, as it is possible to see in Figure 33.

```
@cli.command()
@click.option('-p', '--project_key', required=True, help='Key of the project')
@pass_config
def delete_project(config, project_key):
    """To delete a project"""
    actions.delete_project_action(config=config, project_key=project_key)
```

Figure 33 - Function delete_project

Then, in "delete_project_action", showed in Figure 34, there is developed the URL to call the SonarQube WEB API method to delete the specific project.

```
def delete_project_action(config, project_key):
    """To delete a project, by providing the project key"""
    data = {"project": project_key}
    construct_url(url=config, endpoint="projects", key="delete", data=data)
```

Figure 34 - Function delete_project_action

This function, "construct_url", receives the arguments and develops the URL to call the specific Web API method. In the case of the function shown in Figure 34, the method of Web API called has de URL "server_url/api/projects/delete?project=key" and it asks for the key to the project to be deleted. In Figure 35 it is possible to see the function "construct_url" and its utility, since almost all functions of the CIAnalyzer Tool uses this one to call the Web API methods.

```
def construct_url(url, endpoint, key, data=None, params=None):
    url_join = (url.server, 'api', endpoint, key)
    url_construct = '/'.join(url_join)
    if params is not None:
        url_construct = url_construct + params
    if data is not None:
        response = requests.post(url_construct, auth=(url.user, ''), data=data, timeout=5)
        if not response.ok:
            response = exceptions.retry(config=url,
                                        func=(requests.post(url_construct, auth=(url.user, ''),
                                                            data=data, timeout=5)))
    else:
        response = requests.get(url_construct, auth=(url.user, ''), timeout=5)
    return response
```

Figure 35 - Function construct_url

To see what arguments are necessary to pass in each command, it is possible to do the following command in the command line:

```
static-analysis-tool -s localhost:8080 -u sq2527sg26 -c C:/Desktop/config.json -e
C:/Desktop/excep.json start-analysis –help
```

By introducing it in command line, you will get the arguments needed to start the analysis, that are already showed in Table 11, as it is possible to see in Figure 36.

```
  To start a new analysis

Options:
  -p, --project_key TEXT   Key of the project  [required]
  -d, --directory TEXT     Base Directory of the project  [required]
  -t, --type TEXT          Analysis type (Full or Light)  [required]
  -c, --changes TEXT       Path to changes File
  -j, --job TEXT           Job Name
  -b, --build_number TEXT  Build Number
```

Figure 36 - Arguments required start-analysis

According to the arguments showed in Figure 36, to start the analysis the user should insert a similar command to the command line, as shown below. Because is a Lightweight analysis, the path to changes file needs to be provide.

static-analysis-tool -s localhost:8080 -u sq2527sg26 -c C:/Desktop/config.json -e C:/Desktop/excep.json start-analysis -p Tools -d C:/Desktop/project/ -t Light -c C:/Desktop/changes.json -j Utils_CIToolsAnalyzer -b 938

When calling this command, the function "start_analysis" is requested and, consequently, starts "check_analysis". The function "start_analysis" can be checked in Figure 37.

```
@cli.command()
@click.option('-p', '--project_key', required=True, help='Key of the project')
@click.option('-d', '--directory', required=True, help='Base Directory of the project')
@click.option('-t', '--type', required=True, help='Analysis type (Full or Light)')
@click.option('-c', '--changes', required=False, help='Path to changes File')
@click.option('-j', '--job', required=False, help='Job Name')
@click.option('-b', '--build_number', required=False, help='Build Number')
@pass_config
def start_analysis(config, project_key, directory, type, changes, job, build_number):
    """To start a new analysis"""
    actions.check_analysis(config=config, project_key=project_key,
                           directory=directory, type=type, changes=changes, job=job, build_number=build_number)
```

Figure 37 - Function start-analysis

Using click, a Python package, as can be seen in Figure 37, the function receives the parameters and call "check_analysis" function, passing the received arguments. In function "check_analysis", the changes are checked and then, files for Json analysis are separated from the others for SonarQube analysis and the analyses are started, passing the required parameters.

After that, if there are files for SonarQube analysis when checking for differences, all sonar-scanner's configurations needed are set, like host, login, project name, base directory, exclusions and then a report with the results of the analysis is returned. In Figure 38 it is possible to see an example of the report, with the analysis overview of the SonarQube analysis, showing the job info, the overview with times, measures, and results and also the quality gates implemented.

```
"CITools": {
    "Job Info": {
        "Job name": "Utils_CIToolsAnalyzer",
        "Build number": "1658"
    },
    "Analysis Overview": {
        "Analysis result": true,
        "Total time": "507 seconds",
        "Analysis time": "356 seconds",
        "Quality gate": "This project passed the analysis. ",
        "Measures": {
            "Duplicated Blocks": 0.0,
            "Security rating": 1.0,
            "Duplicated lines(%)": 0.0,
            "Reliability rating": 1.0,
            "Total time debt": "0 minutes",
            "Security hotspots": 0.0,
            "Maintainability rating": 1.0
        },
        "Issues": {
            "Bugs": 0,
            "Vulnerabilities": 0,
            "Code smells": 0
        }
    },
    "Quality gates": {
        "Name": "Commit",
        "Conditions": [
            {
                "id": "AYdMhn56K98ZFMkQPSAy",
                "metric": "new_blocker_violations",
                "op": "GT",
                "error": "0"
            },
            {
                "id": "AYdMhpxhK98ZFMkQPSAz",
                "metric": "new_major_violations",
                "op": "GT",
                "error": "0"
            },
            {
                "id": "AYdMhryvK98ZFMkQPSA0",
                "metric": "new_critical_violations",
                "op": "GT",
                "error": "0"
            },
            {
                "id": "AYdMhtQoK98ZFMkQPSA1",
                "metric": "new_minor_violations",
                "op": "GT",
                "error": "0"
            },
            {
                "id": "AYdRoO1zxw3w1U6ejx5a",
                "metric": "new_bugs",
                "op": "GT",
                "error": "0"
            }
        ]
    }
}
```


Figure 38 - Analysis Report start-analysis

In the case of Figure 38, it was a Light analysis, and the report shows the results on new code. The project “CITools” passed the analysis, and it is also possible to see that build number 1658 of the Job “Utils_CIToolsAnalyzer” was the one that triggered this analysis.

The CIAnalyzer Tool is also capable to generate and return a report with the results of Json analysis, as shows Figure 40. The command to analyze Json is the same for the SonarQube analysis, start-analysis, because the tool can detect if the file is a Json or not and analyze depending on it. The Json analysis is made with the help of the “json.load()” Python function, that returns the first error that it encounters in each file, as shows Figure 39.

```
def do_analyse(file_to_analyze):
    """To do the analysis"""
    is_ok = False
    message = None
    with open(file_to_analyze, encoding='utf-8') as file:
        try:
            _ = json.load(file)
            is_ok = True
        except ValueError as err:
            message = err.args
    result = JsonValidation(file=file_to_analyze, is_ok=is_ok, error=message)
    return result
```

Figure 39 - Json analysis function

The report generated after Json analysis is similar to the report generated for SonarQube analysis, with the information about the Job, an overview of the analysis and the specific errors that it founds, as it is showed in Figure 40.

```
"result": {
  "Job Info": {
    "Job name": "Utils_CIToolsAnalyzer_Nightly",
    "Build number": "24"
  },
  "Analysis result:": false,
  "Message:": "Failed",
  "Total time": "4.1 seconds"
},
"clones.json": {
  "Path": "C:\\clone.json",
  "Is ok?": false,
  "Error:": [
    "Expecting property name enclosed in double quotes: line 37 column 5 (char 624)"
  ]
},
"config.json": {
  "Path": "C:\\configure.json",
  "Is ok?": false,
  "Error:": [
    "Expecting property name enclosed in double quotes: line 7 column 2 (char 212)"
  ]
}
```


Figure 40 - Analysis report Json

In this analysis, the project failed Json analysis because the “clone.json” and “configure.json” files have errors, as it is possible to check in Figure 40. The Job Utils_CIToolsAnalyzer_Nightly and the build number 24 were the ones that triggered this analysis.

5.3 Jenkins

A Jenkins Job, with eight stages, was created to analyze the software code and get the results, as can be seen in Figure 41. Jenkins offers a simple way to set up a continuous integration or continuous delivery (CI/CD) environment for almost any combination of languages and source code repositories using pipelines, as well as automating other routine development tasks (Heller Martin, 2023). This tool is already used by the team as the CI/CD Platform of the Bosch product. After the proof of concept and some design decisions, the Jenkins Job had changes, comparing to the Job defined in the proof of concept.

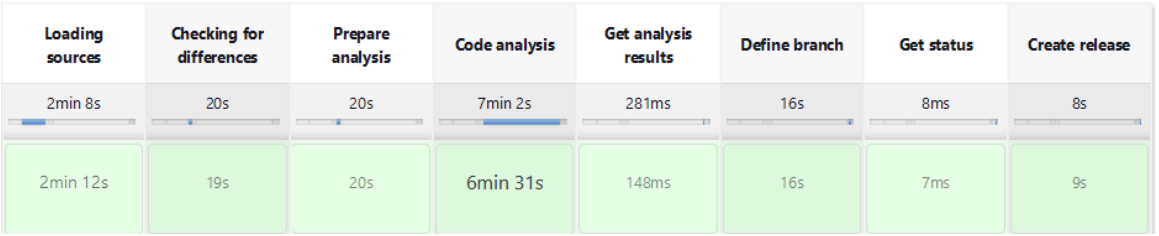


Figure 41 - Jenkins Job

Figure 41 shows the Job and all stages were developed using Groovy programming language. There is one Job for each analysis type, but they are equal, only the name and the analysis type differ, as stated in 4.4.3 Make two different analysis – Job definition. If it is a Full analysis, the stages "Checking for Differences" and "Create Release" are skipped. On the other hand, the focus of the Lightweight analysis is on the new code and all the stages are fulfilled, unless if the analysis is failed. Comparing to the Job defined in the proof of concept, this Job has three more stages: "Prepare analysis", "Get analysis results" and "Define branch".

In the first stage of the Job, "Loading sources", the sources, i.e., the content of the team's repository, is downloaded to a machine. It starts with the definition of the key to the project, in case it is the first analysis, and the checking of the build cause, i.e., what triggered the build (a commit made by a user, or a timer), in order to define what is going to be the analysis type. Next, a new snapshot with the current image of the repository is created, and the content of the snapshot is loaded to a folder in order to be analyzed later.

The purpose of the "Checking for differences" stage is to check if any changes have been added to the code. In this way, firstly a file with the list of snapshots is loaded and, if it has at least one snapshot, the new snapshot is compared to the last successful one, and the result of this comparison is saved in a file. After that, if there are no changes between the snapshots, which means that no changes have been added to the repository, the process is stopped, because there is no need to build. Otherwise, if there are changes, the snapshot for this release is named and a file with snapshot's information is loaded.

In "Prepare analysis" stage, if there are changes, the quality gate is selected based on the analysis type: Full or Lightweight. The stage "Code analysis" is where the analysis is started, by passing the directory, the project key, analysis type, job name, build number and, in case of a Lightweight analysis, the path to changes file. After that, it will start the specific analysis (SonarQube analysis, Json analysis or both) and then the reports of the analyses are constructed and saved in a file. In Figure 42 it is possible to see what were described before, the stage "Code analysis".

```
stage('Code analysis') {
    if (noChanges) {
        println('skipping stage...')
        Utils.markStageSkippedForConditional('Code analysis')
    } else {
        try {
            println('Starting code analysis!')
            timeout(time: 1, unit: 'HOURS') {
                String directory = String.format('%s\\sonar\\', env.WORKSPACE)
                if (fullAnalysis) {
                    startAnalysis = String.format('.\\needs\\static-analysis-tool.exe -s %s -u %s -c %s -e %s' +
                            'start-analysis -p %s -d %s -t %s -j %s -b %s', config.sonarServer.server,
                            config.sonarServer.token, config.sonarServer.configFile, config.sonarServer.exceptionLink,
                            env.project_key, directory, env.type, env.JOB_BASE_NAME, env.BUILD_NUMBER)
                } else {
                    String change = String.format('%s\\changes.json', env.WORKSPACE)
                    startAnalysis = String.format('.\\needs\\static-analysis-tool.exe -s %s -u %s -c %s -e %s' +
                            'start-analysis -p %s -d %s -t %s -j %s -b %s -c %s', config.sonarServer.server,
                            config.sonarServer.token, config.sonarServer.configFile, config.sonarServer.exceptionLink,
                            env.project_key, directory, env.type, env.JOB_BASE_NAME, env.BUILD_NUMBER, change)
                }
                bat(returnStdout: true, script: startAnalysis)
                println('Analysis successful!')
            }
        } catch (exception) {
            println('Problems in sonar scanner occured! no data will be available on dashboard!')
        }
    }
}
```

Figure 42 - Code analysis stage

The results of the analysis that has started in the stage represented in Figure 42 are showed in the stage "Get analysis results" and the results are read from the report files, that were loaded in the previous stage. Finally, an analysis event in SonarQube is created and the job name and build number are set as the event name, in order to append this information in SonarQube GUI. This information will appear in activity page, to provide an easy identification of which build triggered each analysis, and the report files are copied to a shared folder.

"Define branch" stage is where, if the code passes SonarQube analysis, trough the CIAnalyzer Tool the ID of the analysis is sought, and this analysis is defined as comparison for New Code on the next SonarQube analysis. If the code fails the analysis, this stage is skipped. Then, in stage "Get status", the results of the analysis are shown, in order to decide if the build passes or fails. This way, the results of the analysis are checked, and it is defined whether the build passed, if both of the analyses passed, or failed, if one of the analyses failed. For each decision, a message is sent to a team's channel on Microsoft Teams and the description of the build is set with the build result and the associated message. Finally, the last stage "Create release" is the creation of the release, only if the code passes the analyses, and if it is a Lightweight analysis. In this stage, the snapshot is provided, and the owner is set.

6. SOLUTION VALIDATION

To validate the solution, it is crucial to verify if all the requirements were met. To outline the required features and functionalities that affect the success of the solution, the requirements for the support solution were defined in Solution Specifications chapter. In this approach, it is crucial to double-check the requirements once the solution has been implemented, by carefully reviewing and validating each need to see whether it has been met. In this manner, Table 12 displays the cross-checks for the requirements and describes how it is checked.

Table 12 - Requirements Cross-Check

ID	Requirement category	Requirement description	Checked?	How it is checked?
1	Non-Functional	The solution must be easy to deploy. If the machine does not have the tool installed, there must be a script that installs and configures it on that machine.	Yes	Sonar Scanner has been introduced to the pipeline as a resource, making it automatically accessible at all times. SonarQube server is allocated in one machine, which operates as a service, and it is always accessible.
2	Non-Functional	The platform shall provide means to enable/disable the configured Quality Checks and Quality Gates.	Yes	Detailed in 4.4.1 Manage Quality Checks and Quality Gates.
3	Non-Functional	It shall be possible to analyze a project locally on developer workstation and in a CI/CD Pipeline.	Yes	Detailed in 5.1.1 SonarQube configuration, when talking about SonarLint.
4	Non-Functional	The feedback must be provided in less than one minute.	Yes	The feedback it is showed in 5.2 CIAnalyzer Tool and is provided in 57s after the analysis.
5	Functional	The platform should perform continuous inspections of SW code by implementing predefined Quality Checks and Quality Gates in a CI/CD environment.	Yes	Through SonarQube analysis, integrated in the CI/CD pipeline, the continuous inspection is done, and the Quality Checks and Quality Gates were previously implemented in SonarQube.

ID	Requirement category	Requirement description	Checked?	How it is checked?
6	Functional	It must be possible to perform an analysis of the complete project, components, or individual files.	Yes	It is possible to analyze the content that the user want's, by providing the path to the project, component or to the individual file to be analyzed as an argument in "start-analysis" function of CIAnalyzer Tool.
7	Functional	Feedback shall be provided upon analysis completion with reference to the generated report.	Yes	The feedback is made available as output in the pipeline together with links to the SonarQube analysis findings and the SonarQube and Json analysis reports made with the CIAnalyzer Tool. Additionally, a message with the overall result and a link to the analysis results is delivered to a Microsoft Teams channel.
8	Functional	The platform shall report the projects' code quality and depict the trend over time by saving historical data.	Yes	Detailed in 4.4.5 Analysis report and history.
9	Functional	It shall be possible to do two different analyses: one every time someone delivers the code and the other once per night.	Yes	Detailed in 4.4.3 Make two different analysis – Job definition and in 4.4.4 Make two different analysis – Analysis content.
10	Functional	The platform shall provide means to exclude files or directories from the SonarQube analysis.	Yes	Detailed in 4.4.2 Exclude files or directories from analysis.
11	Domain	The platform shall be developed adhering to the existing architecture and have a command line tool that can be used by external tools.	Yes	The platform is adhering to the existing architecture, as detailed in 4.1 Architecture, and has a command line tool, as detailed in 5.2 CIAnalyzer Tool.

ID	Requirement category	Requirement description	Checked?	How it is checked?
12	Domain	The Platform shall be integrated into the CI/CD Platform of the Bosch product.	Yes	The Bosch product CI/CD Platform is Jenkins, so this platform was integrated in there. Detailed in 5.3 Jenkins.

As an example, the requirement with ID 4, "The feedback must be provided in less than one minute." It is checked and can be seen with more detail in 5.2 CIAnalyzer Tool. So, through Table 12 it is possible to confirm that after the implementation all the requirements are met, so the support solution can be validated.

7. RESULTS

The findings of the support solution are presented in this chapter, along with a thorough examination of the outcomes of its use. The findings of the last analysis are presented first, followed by the findings of the first analysis, and the results are compared and discussed. After the findings, it is important to assess if the goals have been achieved since they are essential for validating the solution.

7.1 First Analysis

An initial analysis was made in order to compare the changes over time. In this way, the dashboards of SonarQube were used to track if the results were getting better or staying constant. So, the initial analysis got the results showed in Figure 43, that has an overview of the initial analysis.

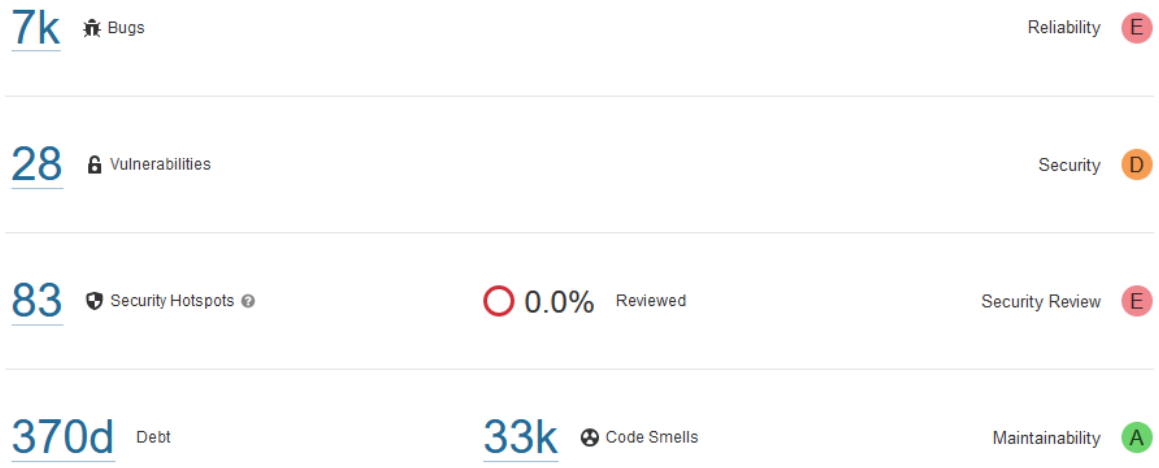


Figure 43 - Initial Analysis Overview

In this initial analysis, through Figure 43 it is possible to check that the repository has 7k bugs, 28 vulnerabilities, 83 security spots, 370d of debt time, i.e., time that will be needed to solve the code smells, 33k code smells. In Figure 43 it is showed that the reliability and security review have the worst value, E, the security has a rating of D, and the maintainability is good, with the best possible value, A. This analysis was triggered by the Job "Utils_CIToolsAnalyzer" with the build number 1, on April 12, as it is possible to check in Figure 44.

Figure 44 - Initial Analysis Job Info

Detailing the results, the reliability, showed in Figure 45, has a general rating of E and the graph represents the bugs' operational risks, sorted by volume of bugs per file. Each bubble in this graph represents a file, and the color symbolizes the reliability rating. The closer a bubble's color is to red, the more severe the bugs are; the closer it is to green, the better. The bubble size represents the bug volume in each file, so if the file has many bugs, the bubble will be bigger, and the position represents the estimated time to resolve the bugs.

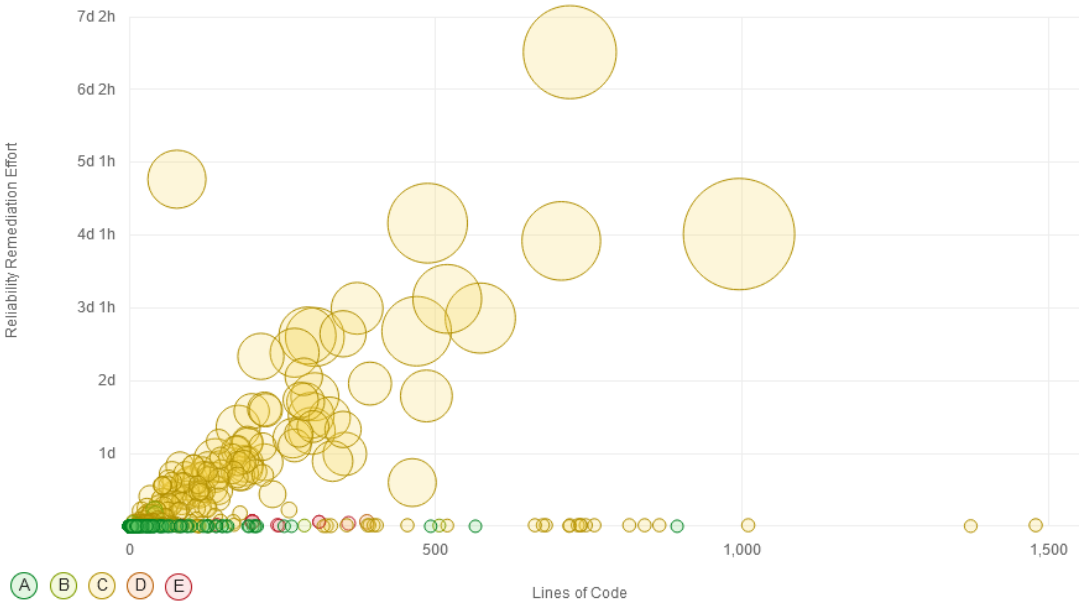


Figure 45 - Initial Analysis Reliability

Because the bubbles in Figure 45 are large and the remediation effort is also quite significant, it is possible to observe that the reliability rating of the files in repository is not good. Also, there are a lot of files with a reliability rating of C, which is not a good value, and a lot of bugs.

In the security level, there are represented the vulnerabilities' operational risks, as it is detailed in Figure 46. The graph is constructed like the one in Figure 45, so the explanation is the same, but now the size of the bubbles is the number of vulnerabilities, instead of bugs.

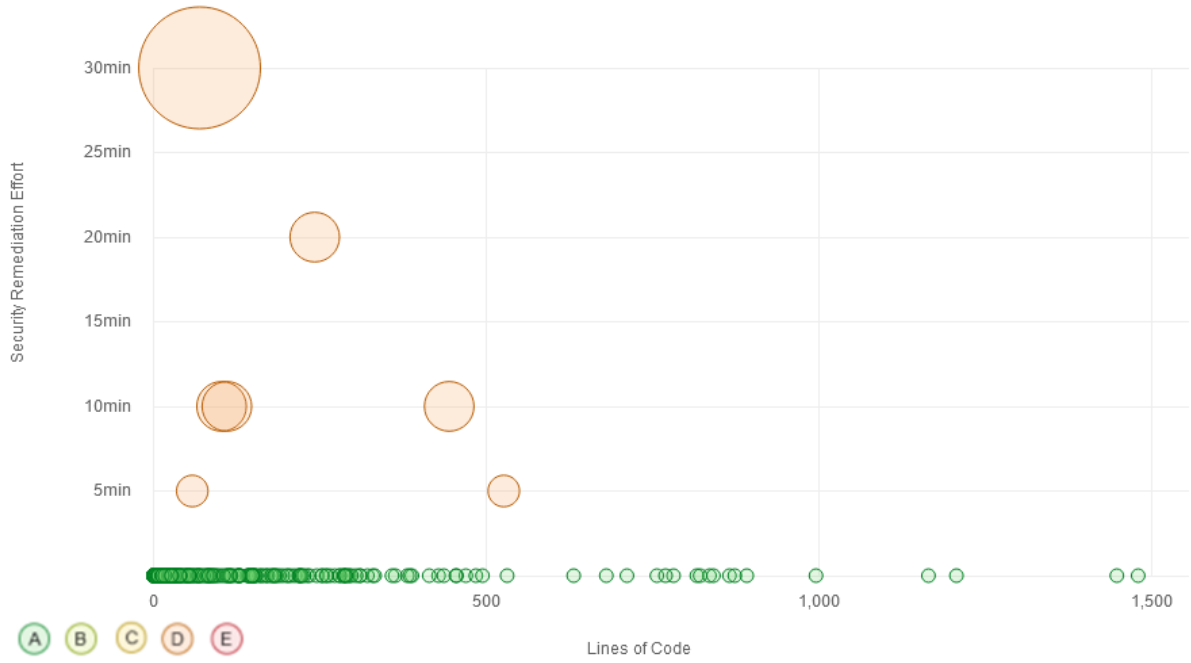


Figure 46 - Initial Analysis Security

In a general way, the security as a rating of D and, through Figure 46, it is possible to check that are some files with a big number of vulnerabilities, but it will not take too long to address them.

The maintainability, represented in Figure 47, shows the code smells' long-term risks, so the size of the bubbles represents the number of code smells. The maintainability is rated A overall, which is a great score.

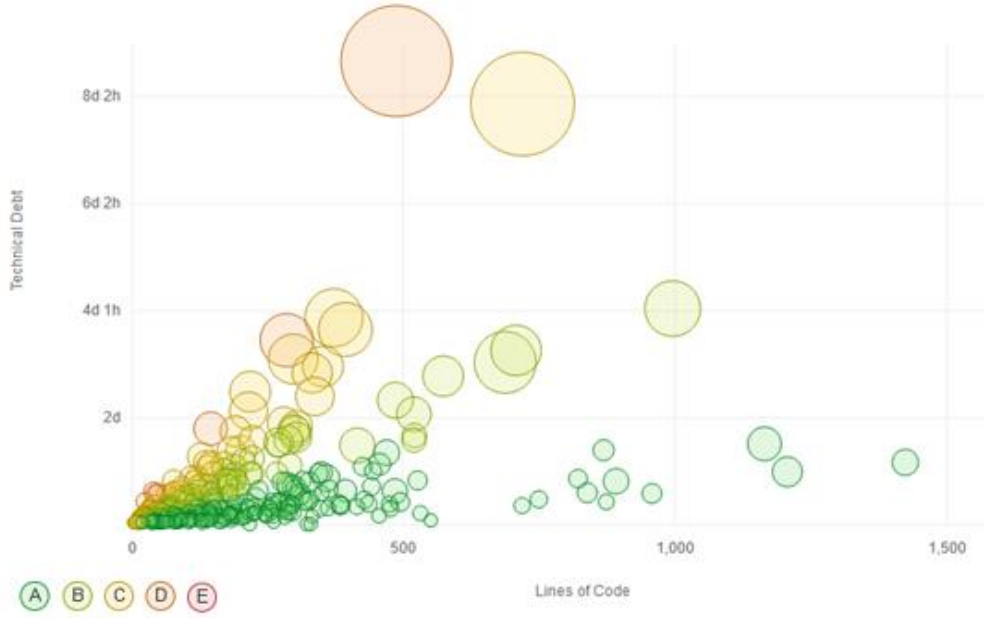


Figure 47 - Initial Analysis Maintainability

It shows from Figure 47 that there are some files with a lot of code smells and with a big number of technical debt time.

7.2 Last Analysis

Two months after the support solution's implementation, the results seem to be much better. The overview of the analysis, made on June 28, 2023, demonstrates a significant improvement in the results. In Figure 48 it is possible to see the overview of the results in the last analysis that was made.

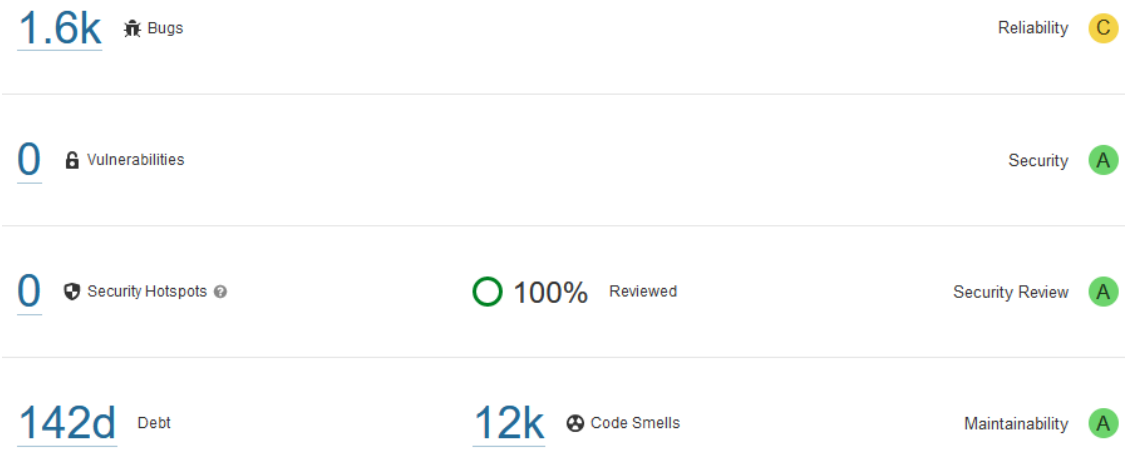


Figure 48 - Last Analysis Overview

With the results showed in Figure 48 it is possible to check that the security and maintainability rating have a value of A, the best possible value, and the reliability rating is C. In a general way, the project has 1.6k of bugs, 12k code smells and it will take 142 days to resolve the current code smells. This analysis was triggered by the Job "Utils_CIToolsAnalyzer" with the build number 84, on June 28, 2023, as can be seen in Figure 49.

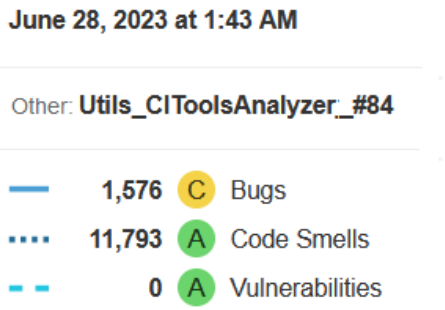


Figure 49 - Last Analysis Job Info

Looking for the reliability, it has a rating of C and, as explained before, the graph represents the bugs' operational risks, sorted by volume of bugs per file. In the graph showed in Figure 50, the color represents the reliability rating, and each bubble represents one file. The color of the bubbles indicates the reliability rating, so the closer a bubble's color is to red, the more severe the bugs are, and the greener they are, the better. The bubble size represents the bug volume in each file, so if the file has many bugs, the bubble will be bigger, and the position represents the estimated time to resolve the bugs.

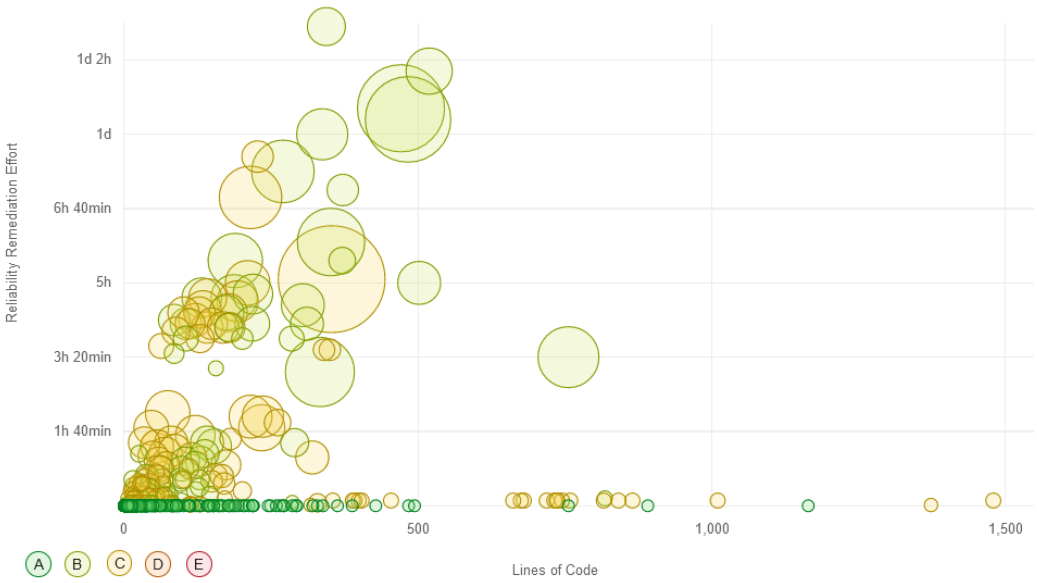


Figure 50 - Last Analysis Reliability

In Figure 50 it is possible to see that are many files with a rating of A, but also files with a rating of B and C, and some files has a big bubble, which indicates that the file has many bugs, but the reliability remediation effort is not so high.

Looking for the security, there are represented the vulnerabilities' operational risks, as it is detailed in Figure 51. The graph has the same logic of the one in Figure 50, so the explanation is the same, but now the size of the bubbles is the number of vulnerabilities, instead of bugs.

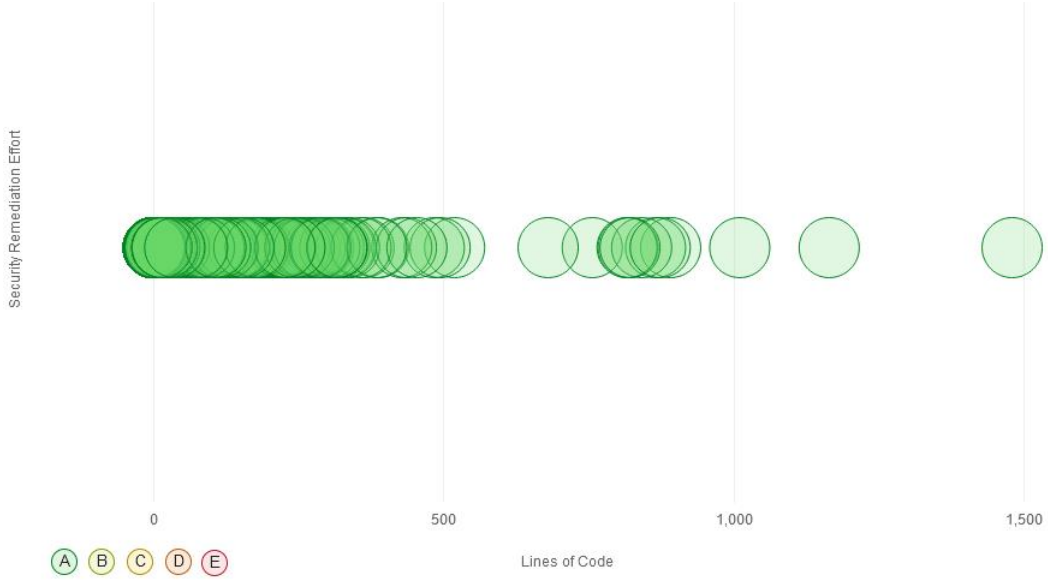


Figure 51 – Last Analysis Security

Through Figure 51 can be seen that the security as a rating of A and there are no files with vulnerabilities, which is great.

The maintainability, represented in Figure 52, as explained before, shows the code smells' long-term risks, so the size of the bubbles represents the number of code smells. The general rating of the maintainability is A, which is a great value.

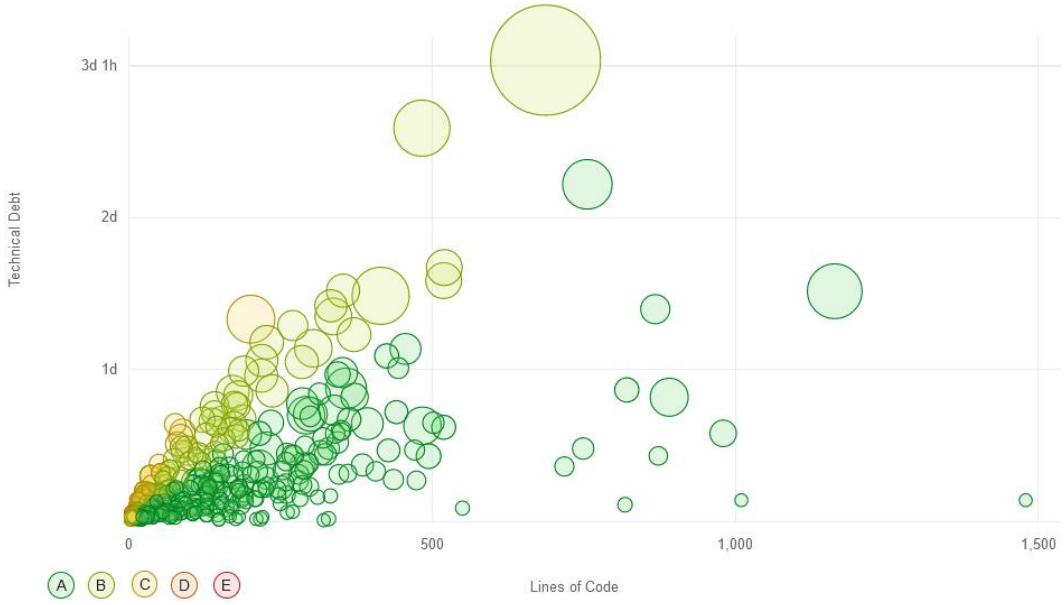


Figure 52 - Last Analysis Maintainability

Through the Figure 52 seems that the technical debt is a little high for some files, but, in general, the files have little bubbles, which means that they have few code smells, and almost all have rating of A and B, which are good values.

In a general way, in Figure 53 it is possible to see all the history of the number of issues, from April 12 to June 28.

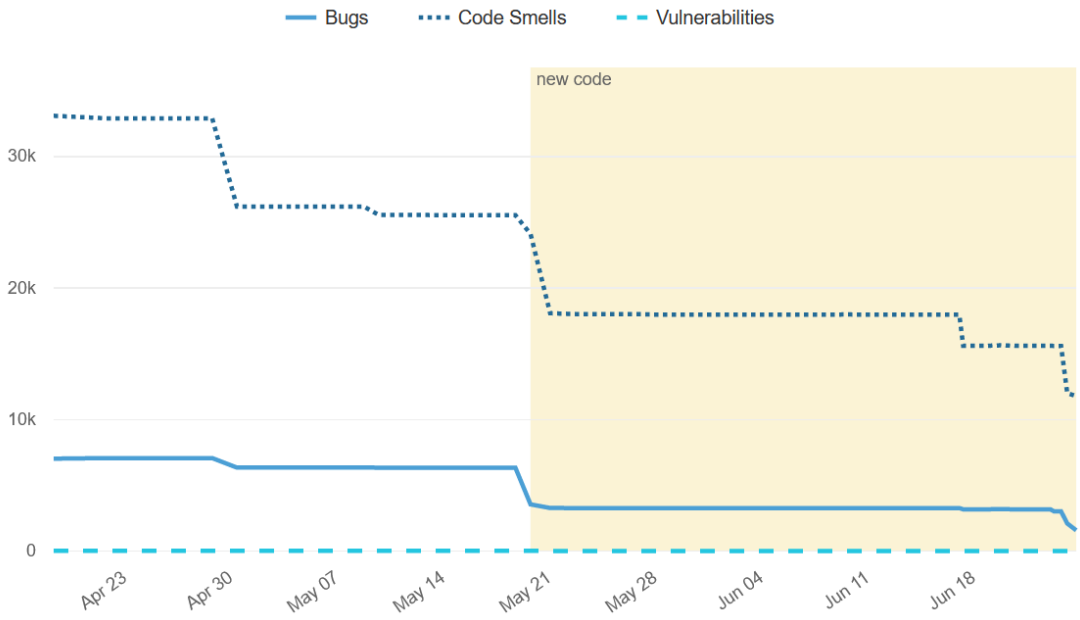


Figure 53 - Issues History

The chart in Figure 53 shows that the number of issues was either dropping or remaining constant, implying that once the solution had been implemented, no additional issues were ever added to the code.

7.3 Discussion

That are several improvements that may be found by searching for the results of the last analysis and contrasting them with those from the initial analysis. The results from the second analysis appear to be significantly better when comparing the two analyses. Overall, the reliability rating improves from E to C, the security rating improves from D to A, and the maintainability rating remains at A, which are great values. Figure 44 and Figure 49, which contrast the findings of the first analysis with the last, show that there was a 77.7% decrease in bugs, a 64.6% reduction in code smells, and 100% in vulnerabilities and these are all very encouraging results.

When comparing the reliability overview of the second analysis to the first analysis, the second analysis looks better and contains much less bugs. Looking for Figure 45 and Figure 50, it is possible to check how the security remediation effort differs from a maximum of 7d2h to a maximum of 1d2h. The number of issues in each file decreased, and each file's rating is better in Figure 50, as opposed to Figure 45, where there are fewer files with a rating of A. Then, the security overview results show that, in the second analysis, all security issues, i.e., vulnerabilities were resolved and now there are no vulnerabilities in the repository. Figure 46 and Figure 51 can be compared to determine whether there has been a full upgrading, with all the vulnerabilities solved. Looking for the maintainability, the highest technical debt from Figure 52 is 3d1h, which is lower than the number from Figure 47, which is upper than 8d2h. As a result, looking at maintainability comparing both figures, it is also possible to detect an increase in the software quality. Additionally, the number of green bubbles increased, which is positive because it means that more files overall have a higher maintainability rating.

Through the results, it is possible to say that, with the implementation of the solution, the number of issues in the software has improved and in comparison to the outcomes of the initial analysis, it is currently considerably better. In this way, it is possible to determine if all of the goals for this dissertation have been achieved by looking at 1.4 Objectives. It is possible to emphasize that the support solution has contributed to the creation and maintenance of high-quality, since the number of issues decreased, as a result of the implementation of the support solution. With this, it is possible to confirm that the solution is capable of carrying out a continuous inspection of the software quality.

8. Conclusion and Future Work

This Chapter aims to provide an overview of the work completed for this dissertation, highlighting its strengths and contributions while also drawing attention to its shortcomings. There are also some suggestions for future study that might be conducted to extend this research.

8.1 Conclusion

This work provides a thorough understanding of how to achieve continuous inspection of software quality in an automotive project, in order to ease the creation and maintenance of high-quality. A support solution was built to do this, and some issues that already existed were manually corrected.

It was a good choice to use Design Science Research as the dissertation's methodology since it offers an organized process for creating artifacts and solutions that successfully address real-world problems.

According to the results, it is possible to verify that the support solution and the entire dissertation have effectively supported reaching the main goal, which was to achieve a solution for continuous inspection of software quality, in order to ease the creation and maintenance of high-quality software. With the implementation of the solution, the developers can no longer deliver software with issues, and the team has already manually fixed 66.7% of the issues that were initially presented.

Saying that the ideal situation is to resolve every issue is impractical since the team would have to put development on hold for a certain period of time, which is not expected to occur. In this way, while not all of the problems were resolved, a significant number of them were within the aim of this dissertation. In general, there was no rise in the number of issues in the repository following the adoption of the solution, and it is evident that the number of issues has decreased, as it is possible to see through the results, and it is really advantageous for the team, the product, and the company.

Additionally, it is now (after this implementation) possible to observe an improvement in the way that all team members produce cleaner software code, use best practices, and stay careful to any potential issue they may be adding to the software that they are producing.

Looking for the objectives, for the objective “Study the existing scientific knowledge about the research topic” a study of the concepts and a literature review was done in the research topic based on 90 articles, obtained through the keywords “continuous integration”, “continuous delivery”, “continuous inspection” and “software quality”. To achieve the objective “Define specifications for the support solution”, there

was a meeting with the involved stakeholders to define the programming and data exchange languages to be analyzed and the requirements. Then, a study of seven currently available tools and technologies linked to continuous inspection and related to the requirements was done for the second objective "Analyze tools and technologies in the market for continuous inspection of software quality" and one was selected. With the creation of a proof-of-concept for the support platform, the objective of "Implement a proof of concept of the support platform" was accomplished, as was the objective of "Determine whether it makes sense to continue with the implementation of the support platform, after the proof of concept", which was verified following the proof-of-concept to see whether the solution could help in achieving the objectives. With the definition of the quality gates to be implemented on the solution, as well as the quality profiles defined, the objective "Define and implement quality checks and quality gates in the support platform, based on specifications" was also done. Next, to "Manually decrease the number of legacy issues", some existing issues were manually resolved, using the SonarQube report and the knowledge to see the issue and check how to resolve it. Finally, the objective "Validate the support solution after the implementation, based on the results, the objectives, and the requirements" was completed following the solution's validation, which determined whether the results were good and if the support solution met the requirements and the dissertation's objectives. This makes it feasible to claim that all of the objectives set forth for this dissertation were achieved.

During this work, the lack of technical understanding and the time-consuming initial ramp-up to comprehend all the principles and operation of the organization were two of the challenges that were encountered during the dissertation. On the other side, the dissertation's strengths might be highlighted as the company's team members' and the dissertation supervisor's assistance, knowledge gained through time, the learning of new programming languages and the fact that the dissertation has a real-world application. Furthermore, the time spent correcting SonarQube issues in the team's repository, allowed to learn and understand the rules and best practices of each programming language, as well as an overview of the work done by the team, which can be considered quite positive.

In sum, since the requirements and objectives were accomplished, the methodology was effective and the results are promising, it can be said that this dissertation adds in theory and in practice to the CI/CD area, and the knowledge created during the dissertation was helpful, contributing with a solution capable to provide continuous inspection of the software quality, in an automotive project.

8.2 Future Work

Although the main goal of this dissertation — contributing to continuous inspection of software quality — has been accomplished, there is always opportunity for improvement. In this manner, additional improvements to this work can be made to improve this dissertation in the following ways:

- Continue correcting the existing issues in the repository, in order to improve even more the quality of the software.
- Apply this solution to other projects because it worked effectively, received positive team input, and produced good outcomes after adoption.
- Include the development of new specific rules and best practices and their adaptation for each project, according to the project specifications.
- To afford an in-depth analysis for Json files, SonarQube may be integrated with another analysis tool that supports this programming language.
- Build a dashboard that includes more detailed data and useful visuals to promote improved decision-making.

REFERENCES

Adams, B., & McIntosh, S. (2016). Modern Release Engineering in a Nutshell – Why Researchers Should Care. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, *5*, 78–90. https://doi.org/10.1109/SANER.2016.108

Ahmad, A., Leifler, O., & Sandahl, K. (2021). Software Professionals' Information Needs in Continuous Integration and Delivery. *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 1513–1520. https://doi.org/10.1145/3412841.3442026

Arachchi, S. A. I. B. S., & Perera, I. (2018). Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. *2018 Moratuwa Engineering Research Conference (MERCon)*, 156–161. https://doi.org/10.1109/MERCon.2018.8421965

Armenise, V. (2015). Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery. *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, 24–27. https://doi.org/10.1109/RELENG.2015.19

Barhate, S. S. (2015). Effective test strategy for testing automotive software. *2015 International Conference on Industrial Instrumentation and Control (ICIC)*, 645–649. https://doi.org/10.1109/IIC.2015.7150821

Batra, P., & Jatain, A. (2021). Hybrid model for evaluation of quality aware DevOps. *International Journal of Applied Science and Engineering*, *18*(5), 1–11. https://doi.org/10.6703/IJASE.202109_18(5).013

Bernardo, J. H., Da Costa, D. A., & Kulesza, U. (2018). Studying the impact of adopting continuous integration on the delivery time of pull requests. *Proceedings - International Conference on Software Engineering*, 131–141. https://doi.org/10.1145/3196398.3196421

Bolduc, C. (2016). Lessons Learned: Using a Static Analysis Tool within a Continuous Integration System. *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 37–40. https://doi.org/10.1109/ISSREW.2016.48

Cao, Y., Xiao, C., Cyr, B., Zhou, Y., Park, W., Rampazzi, S., Chen, Q. A., Fu, K., & Mao, Z. M. (2019). Adversarial Sensor Attack on LiDAR-Based Perception in Autonomous Driving. *Proceedings of the*

2019 ACM SIGSAC Conference on Computer and Communications Security, 2267–2281. https://doi.org/10.1145/3319535.3339815

Chen, L. (2015). Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*, *32*(2), 50–54. https://doi.org/10.1109/MS.2015.27

Chen, L. (2017). Continuous Delivery: Overcoming adoption challenges. *Journal of Systems and Software*, *128*, 72–86. https://doi.org/https://doi.org/10.1016/j.jss.2017.02.013

Czarnecki, K. (2018). Requirements Engineering in the Age of Societal-Scale Cyber-Physical Systems: The Case of Automated Driving. *2018 IEEE 26th International Requirements Engineering Conference (RE)*, 3–4. https://doi.org/10.1109/RE.2018.00-57

Dakic, P., Todorovic, V., & Vranic, V. (2022). Financial Justification for using CI/CD and Code Analysis for Software Quality Improvement in the Automotive Industry. *2022 IEEE Zooming Innovation in Consumer Technologies Conference, ZINC 2022*, 149–154. https://doi.org/10.1109/ZINC55034.2022.9840702

Dakić, P., & Źivković, M. (2021). An Overview of the Challenges for Developing Software within the Field of Autonomous Vehicles. *7th Conference on the Engineering of Computer Based Systems*. https://doi.org/10.1145/3459960.3459972

De Andrade Gomes, P. H., Garcia, R. E., Spadon, G., Eler, D. M., Olivete, C., & Correia, R. C. M. (2017). Teaching software quality via source code inspection tool. *Proceedings - Frontiers in Education Conference, FIE*, *2017-October*, 1–8. https://doi.org/10.1109/FIE.2017.8190658

Debroy, V., Miller, S., & Brimble, L. (2018). Building Lean Continuous Integration and Delivery Pipelines by Applying DevOps Principles: A Case Study at Varidesk. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 851–856. https://doi.org/10.1145/3236024.3275528

Deshpande, A., Veenadevi, S. V, & Aleti, S. (2021). Test Automation and Continuous Integration using Jenkins for Smart Card OS. *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, 1–5. https://doi.org/10.1109/ICCCNT51525.2021.9580021

Di Penta, M. (2020). Understanding and Improving Continuous Integration and Delivery Practice Using Data from the Wild. *Proceedings of the 13th Innovations in Software Engineering Conference on Formerly Known as India Software Engineering Conference*. https://doi.org/10.1145/3385032.3385059

Dikert, K., Paasivaara, M., & Lassenius, C. (2016). Challenges and success factors for large-scale agile transformations: A systematic literature review. *Journal of Systems and Software*, *119*, 87–108. https://doi.org/10.1016/j.jss.2016.06.013

Dresch Aline and Lacerda, D. P. and A. J. A. V. (2015). Design Science Research. In *Design Science Research: A Method for Science and Technology Advancement* (pp. 67–102). Springer International Publishing. https://doi.org/10.1007/978-3-319-07374-3_4

Durisic, D., Staron, M., & Nilsson, M. (2011). Measuring the Size of Changes in Automotive Software Systems and Their Impact on Product Quality. *Proceedings of the 12th International Conference on Product Focused Software Development and Process Improvement*, 10–13. https://doi.org/10.1145/2181101.2181104

Dybå, T., & Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and Software Technology*, *50*(9–10), 833–859. https://doi.org/10.1016/j.infsof.2008.01.006

Eddy, B. P., Wilde, N., Cooper, N. A., Mishra, B., Gamboa, V. S., Shah, K. M., Deleon, A. M., & Shields, N. A. (2017). A Pilot Study on Introducing Continuous Integration and Delivery into Undergraduate Software Engineering Courses. *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, 47–56. https://doi.org/10.1109/CSEET.2017.18

Elkins, D. A., Huang, N., & Alden, J. M. (2004). Agile manufacturing systems in the automotive industry. *International Journal of Production Economics*, *91*(3), 201–214. https://doi.org/10.1016/j.ijpe.2003.07.006

Farkas, T. (2008). Quality improvement in automotive software engineering using a model-based approach. In *Model-Driven Software Development: Integrating Quality Assurance*. https://doi.org/10.4018/978-1-60566-006-6.ch015

Fitzgerald, B., & Stol, K.-J. (2017). Continuous software engineering: A roadmap and agenda. *J. Syst. Softw.*, *123*, 176–189.

Forouzani, S., Chiam, Y., & Forouzani, S. (2016). *Method for Assessing Software Quality Using Source Code Analysis*. 166–170. https://doi.org/10.1145/3033288.3033316

Garg, S., & Garg, S. (2019). Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security. *2019 IEEE Conference on Multimedia*

Information Processing and Retrieval (MIPR), 467–470. https://doi.org/10.1109/MIPR.2019.00094

Gmeiner, J., Ramler, R., & Haslinger, J. (2015). Automated testing in the continuous delivery pipeline: A case study of an online company. *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 1–6. https://doi.org/10.1109/ICSTW.2015.7107423

Górski, T. (2022). Continuous delivery of blockchain distributed applications. *Sensors*, *22*(1). https://doi.org/10.3390/s22010128

Grimm, K. (2003). Software technology in an automotive company - Major challenges. *Proceedings - International Conference on Software Engineering*, 498–503. https://doi.org/10.1109/ICSE.2003.1201228

Guaman, D., Sarmiento, P. A.-Q., Barba-Guamán, L., Cabrera, P., & Enciso, L. (2017). SonarQube as a tool to identify software metrics and technical debt in the source code through static analysis. *2017 7th International Workshop on Computer Science and Engineering, WCSE 2017*, 171–175.

Gupta, S., Bhatia, M., Memoria, M., & Manani, P. (2022). Prevalence of GitOps, DevOps in Fast CI/CD Cycles. *2022 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON)*, *1*, 589–596. https://doi.org/10.1109/COM-IT-CON54601.2022.9850786

Guşeilă, L. G., Bratu, D.-V., & Moraru, S.-A. (2019). Continuous Testing in the Development of IoT Applications. *2019 International Conference on Sensing and Instrumentation in IoT Era (ISSI)*, 1–6. https://doi.org/10.1109/ISSI47111.2019.9043692

Hamdan, S., & Alramouni, S. (2015). A Quality Framework for Software Continuous Integration. *Procedia Manufacturing*, *3*, 2019–2025. https://doi.org/https://doi.org/10.1016/j.promfg.2015.07.249

Heller Martin. (2023, March 15). *What is Jenkins? The CI server explained | InfoWorld*. https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html

Hussain, R., & Zeadally, S. (2019). Autonomous Cars: Research Results, Issues, and Future Challenges. *IEEE Communications Surveys & Tutorials*, *21*(2), 1275–1313. https://doi.org/10.1109/COMST.2018.2869360

Iqbal, T., Iqbal, M., Asad, M., & Khan, A. (2017). A source code quality analysis approach. *SKIMA 2016 - 2016 10th International Conference on Software, Knowledge, Information Management and Applications*, 142–145. https://doi.org/10.1109/SKIMA.2016.7916211

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? *2013 35th International Conference on Software Engineering (ICSE)*, 672–681. https://doi.org/10.1109/ICSE.2013.6606613

Jonsson Wold, S. (2022). *Evaluation of different runner set-ups for CI/CD pipelines*. 98.

Khodiakova, H., & Khodiakova, N. (2021). Creating a reusable infrastructure for continuous integration in software engineering. *2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT)*, *1*, 357–360. https://doi.org/10.1109/CSIT52700.2021.9648613

Khoshgoftaar, T. M., Geleyn, E., Nguyen, L., & Bullard, L. (2002). Cost-sensitive boosting in software quality modeling. *7th IEEE International Symposium on High Assurance Systems Engineering, 2002. Proceedings.*, 51–60. https://doi.org/10.1109/HASE.2002.1173102

Kitchenham, B., & Pfleeger, S. L. (1996). Software quality: the elusive target. *IEEE Software*, *13*(1), 12–21. https://doi.org/10.1109/52.476281

Kokol, P. (2022). Software Quality: How Much Does It Matter? *Electronics*, *11*, 2485. https://doi.org/10.3390/electronics11162485

Kosman, R. J., & Restivo, T. J. (1992). Incorporating the inspection process into a software maintenance organization. *Proceedings Conference on Software Maintenance 1992*, 51–56. https://doi.org/10.1109/ICSM.1992.242559

Laukkanen, E., Paasivaara, M., & Arvonen, T. (2015). Stakeholder Perceptions of the Adoption of Continuous Integration – A Case Study. *2015 Agile Conference*, 11–20. https://doi.org/10.1109/Agile.2015.15

Lavriv, O., Buhyl, B., Klymash, M., & Grynkevych, G. (2017). *Services continuous integration based on modern free infrastructure*. https://doi.org/10.1109/AIACT.2017.8020087

Liu, S., Liu, L., Tang, J., Yu, B., Wang, Y., & Shi, W. (2019). Edge Computing for Autonomous Driving: Opportunities and Challenges. *Proceedings of the IEEE*. https://doi.org/10.1109/JPROC.2019.2915983

Lomio, F., Moreschini, S., & Lenarduzzi, V. (2022). A Machine and Deep Learning analysis among SonarQube rules, Product, and Process Metrics for Faults Prediction. *Empirical Software Engineering*, *27*. https://doi.org/10.1007/s10664-022-10164-z

Lu, Y., Mao, X., Wang, T., Yin, G., Li, Z., & Wang, H. (2018). Poster: Continuous Inspection in the Classroom: Improving Students' Programming Quality with Social Coding Methods. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 141–142.

Luo, Y., Zhang, X.-Y., Arcaini, P., Jin, Z., Zhao, H., Zhang, L., & Ishikawa, F. (2022). Hierarchical Assessment of Safety Requirements for Configurations of Autonomous Driving Systems. *2022 IEEE 30th International Requirements Engineering Conference (RE)*, 88–100. https://doi.org/10.1109/RE54965.2022.00015

Mårtensson, T., Hammarström, P., & Bosch, J. (2017). Continuous Integration is Not About Build Systems. *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 1–9. https://doi.org/10.1109/SEAA.2017.30

Mateen, A., Afsar, S., & Zhu, Q. (2018). Comparitive analysis of manual vs automotive testing for software quality. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3330089.3330121

Myklebust, T., Stålhane, T., & Hanssen, G. (2020). *Agile Safety Case and DevOps for the Automotive Industry*. 4652–4657. https://doi.org/10.3850/978-981-14-8593-0_3495-cd

Nagappan, N., Murphy, B., & Basili, V. R. (2008). The influence of organizational structure on software quality: An empirical case study. *Proceedings - International Conference on Software Engineering*, 521–530. https://doi.org/10.1145/1368088.1368160

Niæetin, S., Šandor, R., Stupar, G., & Tesliæ, N. (2018). Maximizing the Efficiency of Automotive Software Development Environment Using Open Source Technologies. *2018 IEEE 8th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, 1–3. https://doi.org/10.1109/ICCE-Berlin.2018.8576212

Paasivaara, M. (2017). Adopting SAFe to scale agile in a globally distributed organization. *Proceedings - 2017 IEEE 12th International Conference on Global Software Engineering, ICGSE 2017*, 36–40. https://doi.org/10.1109/ICGSE.2017.15

Paliotta, J. (2015). The quality of your code is the quality of your brand - and it's time to pay attention to software testing. *2015 IEEE AUTOTESTCON*, 186–193. https://doi.org/10.1109/AUTEST.2015.7356487

Panaroni, P., Sartori, G., Fabbrini, F., Fusani, M., & Lami, G. (2008). Safety in Automotive Software: An Overview of Current Practices. *2008 32nd Annual IEEE International Computer Software and Applications Conference*, 1053–1058. https://doi.org/10.1109/COMPSAC.2008.139

Panichella, S. (2018). Summarization techniques for code, change, testing, and user feedback (Invited paper). *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, 1–5. https://doi.org/10.1109/VST.2018.8327148

Paule, C., Düllmann, T. F., & Van Hoorn, A. (2019). Vulnerabilities in Continuous Delivery Pipelines? A Case Study. *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 102–108. https://doi.org/10.1109/ICSA-C.2019.00026

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*, 45–77.

Perera, P., Silva, R., & Perera, I. (2017). Improve software quality through practicing DevOps. *17th International Conference on Advances in ICT for Emerging Regions, ICTer 2017 - Proceedings*, *2018-Janua*, 13–18. https://doi.org/10.1109/ICTER.2017.8257807

Poornalinga, K. S., & Rajkumar, Dr. P. (2016). *Continuous Integration, Deployment and Delivery Automation in AWS Cloud Infrastructure*.

Rajesh Kumar. (2021, April 24). *What is Continuous Inspection?* https://www.devopsschool.com/blog/what-is-continuous-inspection/

Rangnau, T., Buijtenen, R., Fransen, F., & Turkmen, F. (2020). *Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines*. https://doi.org/10.1109/EDOC49727.2020.00026

Ren, K., Wang, Q., Wang, C., Qin, Z., & Lin, X. (2020). The Security of Autonomous Driving: Threats, Defenses, and Future Directions. *Proceedings of the IEEE*, *108*, 357–372.

Sampedro, Z., Holt, A., & Hauser, T. (2018). Continuous integration and delivery for HPC: Using Singularity and Jenkins. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3219104.3219147

Sandu, I.-A., & Salceanu, A. (2019). System Testing in Agile SW Development of the Electronic Components Based on Software from the Automotive Industry. *2019 11th International Symposium on Advanced Topics in Electrical Engineering (ATEE)*, 1–4. https://doi.org/10.1109/ATEE.2019.8724968

Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, *5*, 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

Slaughter, S., Harter, D., & Krishnan, M. (1998). Evaluating the Cost of Software Quality. *Commun. ACM*, *41*, 67–73. https://doi.org/10.1145/280324.280335

Soni, M. (2015). End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery. *Proceedings - 2015 IEEE International Conference on Cloud Computing in Emerging Markets, CCEM 2015*, 85–89. https://doi.org/10.1109/CCEM.2015.29

Ståhl, D., & Bosch, J. (2017). Cinders: The continuous integration and delivery architecture framework. *Information and Software Technology*, *83*, 76–93. https://doi.org/10.1016/j.infsof.2016.11.006

Ståhl, D., & Mårtensson, T. (2021). Won't Somebody Please Think of the Tests? A Grounded Theory Approach to Industry Challenges in Continuous Practices. *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 70–77. https://doi.org/10.1109/SEAA53835.2021.00018

Steghöfer, J.-P., Knauss, E., Horkoff, J., & Wohlrab, R. (2019). Challenges of Scaled Agile for Safety-Critical Systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Vol. 11915 LNCS*. https://doi.org/10.1007/978-3-030-35333-9_26

Takoshima, A., & Aoyama, M. (2015). Assessing the Quality of Software Requirements Specifications for Automotive Software Systems. *2015 Asia-Pacific Software Engineering Conference (APSEC)*, 393–400. https://doi.org/10.1109/APSEC.2015.57

Tassey, & Gregory. (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing*.

Turetken, O., Stojanov, I., & Trienekens, J. (2016). Assessing the adoption level of scaled agile development: a maturity model for Scaled Agile Framework (SAFe). *Journal of Software: Evolution and Process*, *29*. https://doi.org/10.1002/smr.1796

Uzunbayir, S., & Kurtel, K. (2018). A Review of Source Code Management Tools for Continuous Software Development. *2018 3rd International Conference on Computer Science and Engineering (UBMK)*, 414–419. https://doi.org/10.1109/UBMK.2018.8566644

van der Valk, R., Pelliccione, P., Lago, P., Heldal, R., Knauss, E., & Juul, J. (2018). Transparency and Contracts: Continuous Integration and Delivery in the Automotive Ecosystem. *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 23–32.

Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., & Gall, H. C. (2018). Context is king: The developer perspective on the usage of static analysis tools. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 38–49. https://doi.org/10.1109/SANER.2018.8330195

Vassallo, C., Zampetti, F., Romano, D., Beller, M., Panichella, A., Di Penta, M., & Zaidman, A. (2016). Continuous Delivery Practices in a Large Financial Organization. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 519–528. https://doi.org/10.1109/ICSME.2016.72

Virmani, M. (2015). Understanding DevOps & bridging the gap from continuous integration to continuous delivery. *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, 78–82. https://doi.org/10.1109/INTECH.2015.7173368

Vost, S., & Wagner, S. (2017). Keeping Continuous Deliveries Safe. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 259–261. https://doi.org/10.1109/ICSE-C.2017.135

Williams, L. (2018). *Continuously integrating security*. https://doi.org/10.1145/3194707.3194717

Zampetti, F., Geremia, S., Bavota, G., & Di Penta, M. (2021). CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study. *Proceedings - 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME 2021*, 471–482. https://doi.org/10.1109/ICSME52107.2021.00048

Zampetti, F., Nardone, V., & Di Penta, M. (2022). Problems and Solutions in Applying Continuous Integration and Delivery to 20 Open-Source Cyber-Physical Systems. *Proceedings - 2022 Mining Software Repositories Conference, MSR 2022*, 646–657. https://doi.org/10.1145/3524842.3527948