INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Automatização de Testes de Regressão para Verificação de Versões de Produtos ASIC

FRANCISCO JOSÉ ALBUQUERQUE DIAS
Junho de 2023

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Automation of Regression Tests for Verification of ASIC Product Releases

Francisco Dias

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Supervisor: Eng. Domingos Terra, PhD Nuno Bettencourt

Porto, June 30, 2023

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, June 30, 2023

Abstract

In the technological industry, performing tests on developed products is a crucial step in ensuring their quality before delivering them to any possible clients. Should a small mistake or bug get through the testing phase, it could create the potential for disastrous consequences waiting to happen.

Regression tests are among these tests that must be performed, with the goal of guaranteeing the correct functioning of new features, while assuring recent features do not compromise previous development. Some of the major downsides to these tests, however, is the extensive amount of time required to perform them, and the fact that most of them have to be run manually. This entails a high cost, both in the duration of the execution as well as the person-hours that have to be invested into this process. Additionally, users must also clean up the disks used for regressions frequently, and this may cause important information to be lost if it is not tracked. In the context of this project, the regression testing is being performed on designs created using SystemVerilog.

This thesis aims to present and document an approach taken to resolve this issue, via the automation of the testing flow by using Continuous Integration and Continuous Development tools. The system that was constructed takes into consideration the varied desires of its users, presenting a high level of configuration in order to facilitate this process, regardless of the projects utilized with it. It presents a novel approach on how to make use of automation tools to improve the process of regression testing, whilst also implementing new useful features for the team responsible for this process.

Keywords: Automation, Regression Testing, Application-Specific Integrated Circuit, Continuous Integration, Continuous Deployment

Resumo

Na indústria tecnológica, a realização de testes nos produtos desenvolvidos é uma etapa crucial para garantir a qualidade dos mesmos antes de os entregar a possíveis clientes. Caso um pequeno erro ou bug passe pela fase de teste, é criado um potencial para consequências desastrosas. Os testes de regressão estão incluídos neste grupo de testes que devem ser realizados, sendo o seu objetivo garantir o correcto funcionamento de novas funcionalidades, ao mesmo tempo que asseguram aos desenvolvedores que as funcionalidades anteriores permanecem inalteradas pelas modificações recentes. Algumas das principais desvantagens destes testes, no entanto, são a extensiva quantidade de tempo necessária para os realizar e o facto de que a maioria precisa de ser executada de forma manual. Isto acarreta um alto custo, tanto na duração da tarefa, quanto nas horas de trabalho humanas que devem ser investidas.

Esta tese tem como objetivo apresentar e documentar uma abordagem para resolver este problema, através da automação do processo de teste utilizando ferramentas de Continuous Integration e Continuous Development. O sistema construído tem em consideração os diferentes objetivos dos seus utilizadores, apresentando assim um alto nível de configuração para facilitar este processo, independentemente dos projetos utilizados em conjunção com o mesmo.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AHP	Analytical Hierarchy Process.
API	Application Programming Interface.
ASIC	Application-Specific Integrated Circuits.
CD	Continuous Deployment.
CDE	Continuous Delivery.
CI	Continuous Integration.
CRUD	Create, Read, Update, Delete.
EDA	Electronic Design Automation.
GUI	Graphical User Interface.
HTML	HyperText Markup Language.
HTTP	Hypertext Transfer Protocol.
IC	Integrated Circuits.
ISEP	Instituto Superior de Engenharia.
JSON	JavaScript Object Notation.
NCD	New Concept Development.
OAuth	Open Authorization.
QEF	Quantitative Evaluation Framework.
REST	Representational State Transfer.
RSS	RDF Site Summary.
SCM	Source Control Management.
SoC	System-on-Chip.
SQL	Structured Query Language.
URI	Universal Resource Identifier.
URL	Universal Resource Locator.
VCS	Version Control System.

Chapter 1

Introduction

Application-Specific Integrated Circuits (ASIC)s are a form of Integrated Circuits (IC)s, and are designed with specific applications in mind, such as for use in a computer, or a mobile device. The software implemented into these circuits has to go through a thorough testing process to ensure that a faulty product is not shipped worldwide to customers. Among these tests, regression testing is one of the most important types of tests to be executed.

Regression testing is a crucial component of verification of software that will later be implemented into products to be used by consumers. This process guarantees that all new features implemented into the software work accordingly, without compromising the functionality of previous features. It is extremely important to obtain 100% functional coverage during this phase to prevent the release of a faulty product, which could come with a heavy cost for the company to repair.

Synopsys, the company this project is being developed for, is no different. In this company, regression testing is used to ensure that the ASIC design components developed for System-on-Chip (SoC) are free of any problems to guarantee a robust product. To help contextualize the problem at hand, this section will provide some background on what Synopsys does and what ASICs are.

Synopsys is an American Electronic Design Automation (EDA) company that focuses on silicon design and verification, silicon intellectual property and software security and quality. Synopsys's products can be applied to a wide array of industries, such as [1]:

- Automotive
- Aerospace & Government
- Financial Services
- Optical
- Internet of Things
- ...

This kind of testing is an intensive, time-consuming task. Engineers have to make sure everything is verified in order to guarantee that maximum coverage is obtained on the SystemVerilog design. However, nowadays, there is a wide array of tools that can be used to make this process more efficient if applied. If a robust automated system is developed upon this already existing flow, and is capable of helping teams perform regression testing, it could save time, cost, and avoid human error. There

is already a company developed script being used to efficiently run the regressions, but active human intervention is still required. Most of this intervention, however, can be automated.

This chapter provides a detailed overview of the problem at hand, as well as the objectives and the methodologies applied in order to fulfil them.

1.1 Problem

The process of running regression tests requires human intervention, by setting up a new area for a specific release and then running a list of tests on the product to ensure its quality. This is often time-consuming and prone to human error. Setup aside, the execution of these tests also takes an extensive amount of time, and the constant monitoring of the status of the tests by an human user results in further time losses.

1.2 Goal

The aim of this project is to make use of automation tools in order to assist the verification team. The system to be developed must help the team set up new areas for testing, and afterwards run the tests that are specified. By performing these tasks, efficiency will increase by running tests and reporting the results without any human intervention throughout the testing. It should be noted that, in the context of this project, the regression testing is performed upon SystemVerilog designs.

1.3 Research Questions

Taking into consideration the goal presented above, the following research questions were formulated:

- **RQ1** - How useful would it be to have a system capable of automatically running configurable sets of regression tests on labels as they're released?
- **RQ2** - How useful would it be for tracking to automatically obtain the results from the regressions run using this system, and make them available on an online dashboard?
- **RQ3** - How useful would it be to have a system that can be used to schedule certain regressions, and thus automatically run them and obtain their results?

The goal of these questions is to understand if this system would be preferred over the current method of running regressions at the company, which is much more manual and not as optimized, and if the users would have any difficulties adapting to this new flow. It should be noted that, in this document, when labels are mentioned, it is referring to the labels used for the Source Control Management (SCM) system used by the company, Perforce. Files in the SCM system can be tagged using labels, which allows users to access them through the use of that same label. As a result, new releases are composed of a set of labels and a few other details used for regression.

1.4 Research Methodology

For this project, a research methodology of *Action Research* was adopted. The reason for this is due to the ever-evolving list of requirements provided by the design team. The base idea presented by the supervisor at Synopsys was to develop a system capable of reading through emails containing information about new releases, parse the required data, and automatically run all the configured tests after setting up an area for it. However, new opportunities and improvements for this system kept surfacing, requiring a methodology that cycles between research and action.

As new opportunities or requirements appear, research is conducted in order to find out the best methods of implementing these new features. This is then followed up by a plan of implementation, be it through charts or a set of instructions to follow (Planning). Afterwards, the new feature is implemented, following the plan (Acting). In the end, this new feature is tested and evaluated by the design team (Observing), and based on their feedback, more research is performed regarding the feature and a new plan is developed, or the project moves on to the next feature to be implemented (Reflecting), thus completing the *Action Research* cycle.

1.5 Work Plan

Having a structured, well-defined work plan is very important for the development of any kind of project, no matter the size. However, it should be kept in mind that plans are always subject to change, be it due to internal or external factors. The plan used throughout the development of this solution is as follows:

- **Research and Analysis** - Research is conducted about the requirement presented, evaluating the methods of implementing it and the tools available for this development;
- **Design and Planning** - In this phase, a plan is written detailing how requirements should be supported. This planning always takes into account the bigger picture - as new features are added on to the evolving system, these will be planned based on the best way of implementing them into the current flow;
- **Implementation** - After a plan is formulated, the requirement is then implemented following the best practices, taking special care to develop it in a simple yet robust way for the system to be utilized by other users;
- **Evaluation and Validation** - The last step in the *Action Research* cycle. This step consists of the testing and evaluation of the implemented requirement, both by the person developing it as well as the design team who wants to use this system in the future. If the requirement is fulfilled, then the next requirement is studied. Otherwise, more research is conducted about the current requirement. Either way, this step leads to the restart of the cycle;
- **Documentation** - This is a task that is performed in parallel with any of the steps in the *Action Research* cycle. At any time during the cycle, documentation is written down instructing how the system is being developed and how it can later be used by other personnel.

Another important aspect to note is the volatile nature of the project developed. As the project was being developed, new requisites and opportunities kept appearing, and so even though the main flow of the project remained the same, new side features and functionalities were developed. In this circumstance, the *Action Research* method became even more appropriate, as when a new requisite or opportunity appeared, new research was required for its development.

1.6 Thesis Structure

For easier comprehension of the contents presented by this thesis, this document is structured into several chapters that each focus on different parts of the work at hand.

These chapters are as follows:

- **Chapter 1 (Introduction)** - Presentation and contextualization of the problem under study, the goals, the methodology and plans followed, and the thesis contributions;
- **Chapter 2 (State of the Art)** - Study and explanation of the different concepts involved in the context of the problem under study;
- **Chapter 3 (Analysis)** - Analysis performed based on the requirements presented by Synopsys, as well as several key factors to have in mind for the design of the solution;
- **Chapter 4 (Design)** - The designs and plans constructed for the development of the solution to the problem under study;
- **Chapter 5 (Implementation)** - Detailed explanation of the steps and procedures taken to implement the solution.
- **Chapter 6 (Evaluation of the Solution)** - Tests performed to evaluate the quality of the solution and the results obtained.
- **Chapter 7 (Conclusion)** - Gathers all the information presented into conclusions that relate to the goal of this thesis. Future improvements to the solution developed will also be mentioned.

Chapter 2

State of the Art

In this chapter, the state of the art is presented and explained, introducing concepts that were applied during the planning, designing, and development of the project. In order to contextualize the problem, ASICs and SoCs will be mentioned first, as the project developed aims to automate the regression tests run on the software that will later on be applied to the hardware mentioned.

Furthermore, the flow of the software developing process will be presented, with examples of tools used with it, as well as the section of the flow where this project fits into. SCM tools will also be discussed, as they are the location where the files used for testing are stored. Related work is mentioned, comparing the solution under study to them.

Finally, the New Concept Development (NCD) Model is also explained, as well as the Analytical Hierarchy Process (AHP), both of them being applied to the project in Chapter 3. The Quantitative Evaluation Framework (QEF) model is also introduced as a means of evaluating the project, being used later in Chapter 6.

Most of the information obtained from research, and thus presented in this paper, originates from ResearchGate, an online archive containing many legitimate, unmodified articles. The remainder of the information obtained comes from either books, or searches using tools such as Google and Google Scholar, ResearchGate, and IEEExplore, with keywords related to the subject in focus, such as “Regression Testing”, “Continuous Integration (CI) / Continuous Deployment (CD) Automation Tools”, “Source Control Management Platforms”, and so on.

2.1 Application-Specific Integrated Circuits and System-on-Chip

ASIC stands for Application-Specific Integrated Circuits. As the name entails, ASICs are a form of IC, designed for a specific application. These ICs are usually fabricated on silicon wafers, said wafer containing hundreds of dicestructures. There are many types of ASIC components. The following list provides some brief insight into each one [2]:

- Full Custom ASICS - Designed if there are no standard libraries available, or when needed to fulfil the requirements of power of performance for a certain application;
- Semi-Custom ASICS - Customized in one or two areas;

- Standard Cell ASICS - Uses pre-designed logic cells (AND gates, OR gates, multiplexers, etc.);
- Gate Array ASICS - Partially finished with the rows of transitors and resistors required, but with the transitors unconnected. The customer designs the chip and the vendor's software generates the interconnection masks;
- Programmable ASICS - Can be programmed for specific applications.

Based on a preliminary study conducted regarding the problem under study, these ASIC and SoC components are not of utmost importance for the development of the solution, as the systems for executing the tests on these devices are already set in stone, and so the solution constructed will call those tests with specific configurations. This study, however, serves as a means to demonstrate just how important it is to guarantee total coverage of the code implemented in these systems, as the smallest errors could cause catastrophic damages, especially considering the wide array of industries in which ASICs and SoCs are used.

2.2 Regression Testing

Regression testing consists of the use of written, often reusable test cases for the testing of new software features or modifications in order to evaluate their reliability and performance, whilst not compromising other features already implemented in the software. As such, regression testing is extremely important to ensure that the product is exported with the best quality possible, especially when the product goes through several changes during development and maintenance phases [3].

There is a wide array of reasons as to why automating the process of regression testing can be an invaluable improvement to the current flow of software testing. For starters, automating this process, if done correctly, increases efficiency and productivity by having automated systems handle the testing of the software while engineers can work on other projects and observe the test results later. Another advantage to note is the fact that regression testing consists of the retesting of software after changes are made to it to confirm that the new features as well as the old ones are functioning correctly, and so, having a system capable of applying test suites to test a certain software on a schedule and/or after new changes are published also contributes to minimizing the chance that a software bug survives the testing process.

To show the importance of guaranteeing full coverage on software before shipping it out to the public, an article dating back to 1994 is referenced here. Intel, the world's largest chip producer at the time, let an error make it through the testing phase when creating a chip named Pentium. This error consists of a bug in mathematical division operations involving numbers with over five significant digits. All of the chips shipped out to public market presented this error. According to Dataquest, a market research company located in San Jose, California, estimated that, at that time, Intel would have sold 5.5 million to 6 million Pentiums, a number that amounted to roughly 10 percent of the total of personal computers sold worldwide.

Upon discovering this error, Intel dismissed it, saying that they did not believe that recalling these chips was unnecessary and claiming that there was a minuscule

chance, roughly one in more than nine billion, that this error would cause an inaccurate result, and that due to this, there would be no noticeable consequences to users of business or home computers.

This statement caused public backlash, as these chips were also being used to perform calculations that should be exact, and an error like this would bring severe consequences. Scientists and engineers especially revolted against the company, as the product was indeed more cost-effective for the operations they were performing, but the existence of the error, despite the minuscule chance of it affecting a result, meant that the results of division operations could never be trusted fully. In the end, this led to Intel having to offer replacement chips to those affected, which caused massive losses for the company, around the $475 million mark [4].

There are various approaches to how regression testing is performed [5]. The following subsections each present a different technique, along with an explanation on how it works.

The **Retest All** method is the simplest one out of all of them, yet also the most expensive in terms of time spent in testing. As the name implies, all of the tests written for a product's test suite are retested in order to perform a full retest. While this is guaranteed to be the safest option in terms of fully verifying the reliability of the product, this method has the downside of a very long execution time and exhaustion of resources for larger products, making this method unreliable for larger scale project testing [6].

In an attempt to resolve the problem posed by the Retest All method, **Regression Test Selection** (RTS) consists of the selection of specific tests from the test suite for testing the new changes to the product. This should result in faster execution times and less usage of resources, however, this poses a few questions, such as which tests should be selected, and when do we know when additional tests are required. These decisions should be taken carefully, as the reliability of the product may be compromised. For a more correct selection of test cases, the characteristics of the previous, fully functioning program and the characteristics of the newly modified program should be taken into account [6].

Not to be confused with the RTS approach, **Test Case Minimization** consists of the removal of redundant tests that may be increasing both the execution time of the test suite as well as the resources used up. This differs from the RTS method in the sense that the full test suite is executed, except for tests that are considered redundant due to being covered by other tests already, thus consisting of a minimization of the tests required to evaluate the reliability of the product [7].

While the Regression Test Selection and Test Case Minimization approaches consist of a reduction in the amount of tests executed, be it by removing those that are redundant or selecting only a portion of the test cases for regression testing, the **Test Case Prioritization** approach makes use of the full test suite. However, the test cases present in the test suite are reorganized in a manner that shows increased reliability and cost-effectiveness. A good criteria to base this sorting on is the probability of a given test detecting a fault - after all, as soon as a bug or some other problem is detected, there is no reason in performing the rest of the tests if the goal is to achieve absolute coverage. Sorting the test suite this way would most likely guarantee a more efficient testing of the product, with reduced downtime and

resource use, and because the full test suite is used, all cases are guaranteed to be covered as well [7].

Finally, the **Hybrid Testing** method consists of a combination of the other methods. Several different hybrid methods have been proposed, namely one consisting of a combination of minimization, modification, and prioritization based selection, and a different technique which merges Regression Test Selection and Test Case Prioritization [5].

2.3 Continuous Integration and Continuous Development

CI and CD are deemed continuous practices, in the sense that they are continuously occurring in a loop. This loop can be seen in Figure 2.1, with CI taking the left side and CD taking the right one.

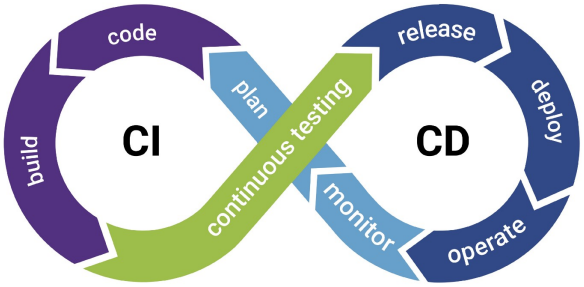


Figure 2.1: CI / CD Workflow [8]

On one hand, CI, as the name implies, consists of a practice in which several developers work together on a project, implementing and merging new code (new features, bug fixes, etc.) very frequently. Thus, CI makes it possible to shorten release cycles, make them more frequent, and in turn increase the team's overall productivity. With CI, the building and testing phases are automated [9]. The solution implemented handles this process - It will build the area for testing new releases by fetching the necessary files, and then run the tests on said releases, entirely automatically.

On the other hand, CD is an extension of Continuous Delivery (CDE). CDE is responsible for ensuring that the product or application is ready to move on to the production phase, after passing through all of the automated tests. This brings many benefits, such as reducing the risk of deploying a faulty product. CD takes this a step further by automating the deployment phase, deploying all the changes made entirely automatically [9]. The solution follows the CDE practice, by informing the team regarding the results of the tests performed on the new release. Should there be any problems, the team can react more quickly. Should the release be free of any problems, the team can then decide to move on to production phase.

2.3.1 DevOps Methodology

By implementing automation of regression testing, this solution promotes a DevOps methodology, which consists of a set of practices that automates the processes between development (Dev) and operations teams (Ops), in order to improve speed and

reliability of building, testing, and releasing of products. This is a step up from the Agile methodology, which consists of small release iterations with customer reviews, thus being more focused in "sprints", yet focusing less on the deployment process. The DevOps methodology proposes a set of practices that encourages a closer collaboration between developers and operators - Development teams are responsible for continuously planning and developing new features to meet the requirements, while operations teams are in charge of managing the modifications made in production phases [10].

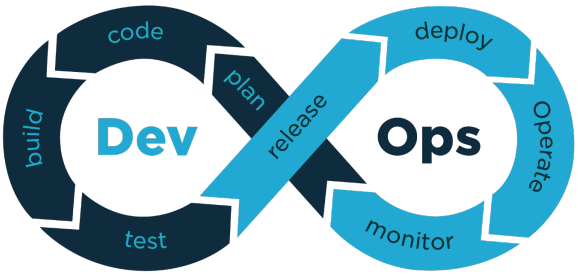


FIGURE 2.2: DevOps Methodology Workflow [11]

In this specific scenario, when the developers send an email regarding a new release, the solution will test the release based on the information gathered and inform the operations team regarding the tests run on the release and the coverage rate obtained. In this sense, this solution covers both the test and release phases from the flow indicated. This applies to both CI / CD and DevOps - As seen from Figure 2.1 and Figure 2.2, the stages presented are roughly the same between the two images.

Nowadays, there are plenty of CI / CD solutions available in the market for companies to use. The following subsections give information on a few of those tools, the selected tools to study being:

- Jenkins;
- Bamboo;
- GitLab.

2.3.2 Jenkins

Jenkins itself is simple to learn and use, it is also quite powerful and has an enormous potential, as the plugins available for use enable users to link the platform with other widely used tools, such as Source Control Management tools, code quality tools, deployment platforms, and so on [12]. This makes it one of the most used platforms when it comes to CI and CD. Another key factor in support of Jenkins is the community itself - It is large, quite active, and helpful, with members in the community helping each other with tutorials and customized plugins.

2.3.3 Bamboo

Bamboo is yet another CI / CD solution, this one being owned by Atlassian. Much like Jenkins, it allows automation of the building, testing, and deployment stages

of development, with the added advantage that it has direct integration with other Atlassian products, such as Bitbucket, JIRA, and Confluence [13].

2.3.4 GitLab

GitLab is an "all remote" company, meaning that it has no offices and all employees work remotely. The company created their own CI solution, named GitLab CI. This tool makes use of scripts to build and test applications automatically every time a change has been made, allowing all the remote users to collaborate on projects better by ensuring that coding bugs and other similar problems stay out [14]. It is also worth noting that GitLab also has developed their own CD solution, GitLab CD, which handles the process of deployment of the application into the production phase, also fully automatically [15].

2.4 Source Control Management

SCM, also known as Version Control System (VCS), are tools that allow the control of the different versions of software that are produced, like a backup system - If a user encounters an error or other problems with a certain version of the product, they may wish to go back to an older version, or compare the changes between the two versions to verify what might have been the cause of the problem. Another great upside is that most of the SCM tools available today facilitate collaboration between members of the same team, or even between the different teams themselves, by storing all the files of the project on a server and allow users to change the files at their own pace [16]. Even if two users change the same file at the same time, these tools help resolve these conflicts with merging tools. The following subsection presents the different types of SCM systems, with the subsections following it presenting examples of SCM tools.

2.4.1 Types of Source Control Management

There exist three main types of SCM tools, out of which the latter two allow for team collaboration [16]:

- Local Source Control Management
- Centralized Source Control Management
- Distributed Source Control Management

Nowadays, team collaboration is a highly important factor that companies need. Due to this, Local Source Control Management systems are rarely, if ever, used in companies, instead being used by single individuals. Due to this, more detail will be given to the other two types of SCM.

The first type of SCM that offers team collaboration is the Centralized SCM, demonstrated in Figure 2.3. In this type of system, the project is kept in a server and made available to the users to fetch files from it and also commit changes to the server. This way, different users can modify different files at the same time and upload their changes to the server seamlessly. As explained earlier, even in the event that different users modify the same files, the existence of merging tools helps resolve these conflicts [16].

An important detail to note is that this system functions by letting users fetch specific versions of the files they pretend to change, usually the latest, also allowing them to fetch only part of the files available on the server. For example, if an user wishes to fetch a file A from a server that contains files A, B, and C, they can obtain only the file they want without fetching the other two. A major flaw with Centralized SCM is that, in the event that the server goes offline, users are unable save their changes to the server, pull other files from it, and thus are unable to collaborate with each other [16].

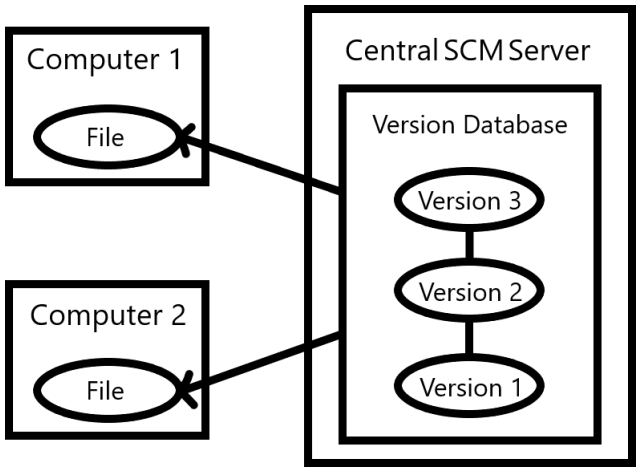


FIGURE 2.3: Centralized Source Control Management

Lastly, Distributed SCM, shown in Figure 2.4, aims to fix the problem that Centralized SCM presents. Whenever a user checks out the server repository for files, they fully mirror the files available, thus obtaining a full copy of the project files, including the file history. This means that, if the server is down, users can still continue working as they possess all the files they need as well as the complete history of the files. Whenever the server is back up, they can then submit the changes they have made, or if a team member requires it, the user can even send a copy of the files they possess [16].

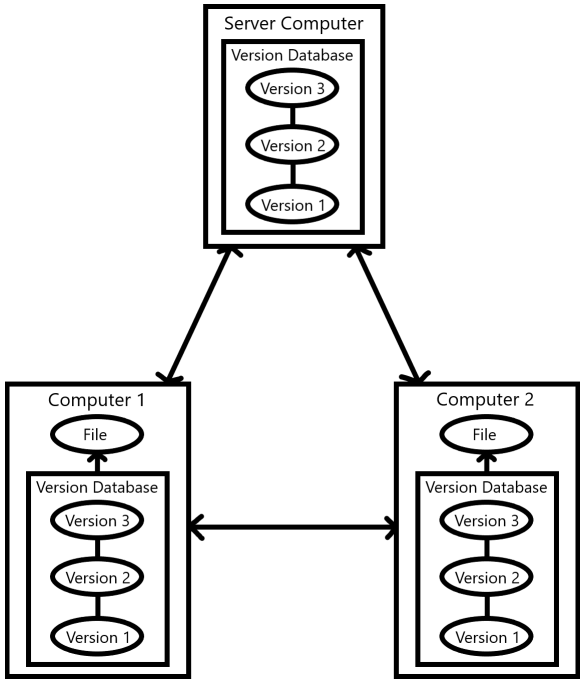


FIGURE 2.4: Distributed Source Control Management

With all of this in mind, the following subsections present different examples of SCM platforms.

2.4.2 GitHub

GitHub is the biggest Git repository platform, used by millions of users for all kinds of projects. A great amount of Git repositories are hosted on GitHub, and there are many open-source projects that make use of the platform for hosting, issue tracking, reviewing code, and so on [17]. GitHub is an example of a Distributed SCM. Its simplicity to use and worldwide recognition make it one of the best platforms for hosting projects for team collaboration. Its reliability, along with integration with several other tools, make it a great choice to use for a project like the one under study.

2.4.3 Bitbucket

Bitbucket, developed by Atlassian, is another highly popular tool when it comes to hosting projects for team collaboration. Much like GitHub, it is also a Distributed SCM - Generally speaking, all Git repository platforms are Distributed SCM platforms. While also simple to use and robust when it comes to hosting projects, Bitbucket has two added advantages over GitHub - It is used for a number of projects developed at Instituto Superior de Engenharia (ISEP), and has integration with other Atlassian tools, such as Bamboo, a CI / CD tool mentioned in Chapter 2.3.3, as well as JIRA and Confluence. Another upside is that it also has a built-in CI / CD solution, called Bitbucket Pipelines, which could be used and remove the need for a separate tool [18].

2.4.4 JIRA

Also developed by Atlassian, JIRA is an issue tracker tool that allows teams to collaborate in a more efficient manner. This tool allows users to create pages for their projects, and inside of them, track bugs in software under development, create tasks, and even handle customer support. These types of items will from now on be referred as issues. Issues can be assigned to specific members of a team in order to help them keep track of what needs to be done or corrected, whilst also helping the remaining members of the team identify who is in charge of a specific issue. Different statuses can also be assigned to these issues in order to help users identify what has been done, what is in progress, and so on. Ultimately, an issue is only considered closed when its status becomes "Closed" or "Done" [19].

In Synopsys's case, JIRA is used to keep track of the many labels involved in the products under development, with other custom statuses such as "Verifying" being available to show users what labels are being tested at the time. The information in the emails sent to the mailing list that the solution under study is responsible for monitoring can often be traced back to JIRA issues created specifically for each of the labels present in said email. In other words, the information present in the emails read by the solution originates from the JIRA pages.

2.4.5 Confluence

Developed by the same company, Confluence obviously presents integration with JIRA. Whilst JIRA is used to define tasks and attribute them to users, Confluence complements it by serving as a knowledge base, in which users can document their projects. A simple example of this is having a project's issues being tracked in JIRA, while all the documentation regarding how to use the project and how it works being all stored in Confluence. Unlike what many people think, Confluence and JIRA do not replace each other - Instead, they complement each other, proven by the fact that there are integrations and plugins available in both platforms to allow the establishment of a connection between the two tools [20].

Similarly to JIRA, Confluence allows the creation of spaces, with each space representing a project, a team, or anything else that may be relevant. Inside the spaces, users can then create pages to document the project or inform other users about any other important information. Tutorials on how to use a tool, notes on how it functions, or even notes taken from meetings are examples of information that can serve as documentation for a project and thus should be written down in Confluence [20].

For this project, one of the starting requisites was that the solution must post a report containing the details of the tests run to a Confluence page. In order to achieve this, the Confluence's built-in Representational State Transfer (REST) Application Programming Interface (API) must be used to allow scripts to create and / or update the information available on Confluence. Another requisite that appeared during the development of the project is to allow users to configure tests to run with specific labels and options in Confluence, and either run them manually, or schedule them to run at specific times or dates. For this, the REST API is also required. The following section explains this in more detail.

2.4.6 Perforce

Unlike the other presented SCM solutions, Perforce is an example of a Centralized SCM system. To make use of this system, an user must first create a client in a certain location and create a mapping view. A mapping view tells the server how the files should be synchronized to the client, inside of its root directory. For example, if a client named UserClient is created, with its root directory being /folder, and the server file at //depot/files/file1 is mapped to the client location /UserClient/file1, the file would be available in the client at /folder/file1. Files that are edited together and then submitted to Perforce are put into a changelist, which is also known as a "commit" in the other presented tools. All changes can be traced back to their changelists and the differences can easily be visualized. The "depot" is the name given to the location where the files are stored on the server, and each client created possesses a mapping of the files it requires, with each user being capable of having several clients [21]. Despite Perforce being a SCM like GitHub and Bitbucket, the way in which they operate is very different due to them being distinct types of SCM.

2.5 Related Work

It is important to mention that, when performing research about this problem and investigating the solution that could be implemented, similar studies were found regarding the automation of the regression testing process. Much like the solution presented in this document, the studies were made in conjunction with other companies, and thus developed for said companies.

2.5.1 Automation Using Visual GUI Tools

Much like the project under study, Sjöblom and Strandberg [22] aimed to automate the regression testing process due to its extensiveness, yet their scenario is quite different. The goal of their project, developed for Saab AB, a Swedish aerospace and defence company, was to automate the process of testing the Graphical User Interface (GUI) used at the company. At the time, the testing process of these interfaces was entirely manual, spending great amounts of person-hours for testing. So extensive in fact, that regression testing was done in a partial manner in order to save on resources and labor [22].

However, according to the facts presented earlier in this thesis, obtaining full coverage of a product before releasing it is crucial for the product's success, and by only running a partial testing of the system, the company was risking having a faulty product. Furthermore, Sjöblom and Strandberg [22] also refer that, due to the large cost of this testing process, caused by the requirement of several person-hours, the tests were executed less often that what is desired, which is obviously another concern.

Thus, Saab AB saw enormous benefits in automating this process. The lower levels of their systems were already being tested automatically (for example, the unit tests being run on the code), and extending this automation to the GUIs would prove to help the company with the big costs of the regression testing process [22].

In order to automate the testing of GUIs, Visual GUI testing tools are required, and this is the biggest difference between the solution under study and the work

of Sjöblom and Strandberg [22] - While both intend to automate the process of regression testing, different tools have to be used, as one aims to automate tests run on software code, while the other aims to automate the testing of the interfaces presented to the user.

To achieve this goal, Sjöblom and Strandberg [22] made use of Sikuli, now renamed SikuliX, a tool that allows the automation of visual tasks performed in systems, such as the use of a website for a specific purpose, or the use of a GUI, like in this scenario. The tool is capable of using image recognition in order to automate these tasks, and it is also able to use the keyboard and mouse to simulate an user's input, which may be required to test the different functionalities of a GUI [23].

2.5.2 Automated Cooperative API Regression Testing Using Jenkins

The project developed by Olsson and Ridderheim [24], for the company Axis Communications AB, also localized in Sweden, is more similar to the solution under study. The two projects aim to run regression tests on software, but the kind of software differs between the two. In this scenario, the solution under study runs regression tests on software that will then be integrated into ASIC chips, whilst the solution developed by Olsson and Ridderheim [24] runs these tests on releases of an API.

Furthermore, the solution developed by Olsson and Ridderheim [24] also has the goal of facilitating the cooperation between different teams when developing a product and executing regression tests on it, hence the term "Cooperative API Regression Testing". Different teams possess different experiences and backgrounds, and in order to reduce the amount of time regression tests take to be performed. This goal is similar to the one presented in this document, seeing as the solution under study aims to increase efficiency when running regression tests and simplifying the process of running regressions.

It is also worth noting that the solution produced by Olsson and Ridderheim [24] makes use of Jenkins, the platform presented in Chapter 2.3.2. Coincidentally, the same platform was used for the implementation of the solution under study. However, these solutions differ in their SCM technology, seeing as Olsson and Ridderheim [24] refer that Axis Communications AB makes use of Git, a technology that follows the principals of Distributed SCM, whilst Synopsys utilizes Perforce, that as explained in Chapter 2.4.6, is a Centralized SCM.

2.6 New Concept Development Model

The New Concept Development Model, designed by Koen et al. [25], was developed to help guide users through the process of developing a new product or idea. This model is utilized in Section 3.2 in order to perform an analysis regarding the value of the project under study.

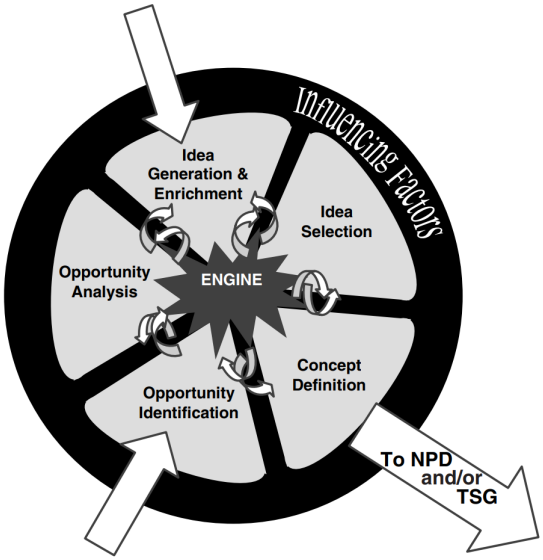


FIGURE 2.5: New Concept Development Model [25]

As seen in Figure 2.5, the model consists of three different key factors [25]:

- The center part, also known as the engine, represents the leadership, culture, and business strategy of the company. This engine is responsible for driving the five key elements controllable by the company.
- The inner area displays the aforementioned five activity elements controllable by the company, those being opportunity identification, opportunity analysis, idea generation and enrichment, idea selection, and concept definition.
- The outside ring represents the external factors, such as distribution channels, the law, government policy, customers, competitors, and also the political and economic climate, all factors that influence and affect the innovation process. These external factors, as the name implies, are uncontrollable by the company.

The following subsections explain the inner area of the model, the five key elements that the company can control, into more detail [25].

2.6.1 Opportunity Identification

In this phase, the company identifies the opportunities that it might want to pursue. These opportunities can be an entirely new product, or an upgrade or update to an already existing one. Opportunities are also not tied to just products - these opportunities can be new manufacturing processes, services to offer, marketing and sales approaches, or even a new direction for the company's business. Opportunity identification defines the market or technological area in which the company may wish to participate in [25].

2.6.2 Opportunity Analysis

In the opportunity analysis process, an opportunity is evaluated to ensure that it is worth to pursue it. To do this, additional information may be needed in order to extract specific business and technology opportunities from the opportunity

identification phase [25].

2.6.3 Idea Generation and Enrichment

The idea generation and enrichment process watches over the appearance, development, and thus maturation of a concrete idea. Ideas suffer many changes throughout their lifespan, thus having multiple iterations. Brainstorming sessions help create or develop new ideas for the identified opportunity [25].

2.6.4 Idea Selection

Most of the time, companies don't have any struggles when it comes to creating new ideas, no matter the size of the company. The problem appears when attempting to select which ideas to pursue to achieve the most business value. In most scenarios, the idea selection process consists of a repetitive sequence of steps that involve going through the process of opportunity identification, opportunity analysis, and idea generation and enrichment, the previously mentioned steps, usually with a different set of influencing external factors and new directives from the engine [25].

2.6.5 Concept Definition

The concept definition is the last step of the model, thus being the only exit to the new product development. To pass this step, the innovator must make a compelling case for investment in the proposition. This case consists of both qualitative and quantitative information that helps make a decision on the proposition [25].

2.6.6 Analytical Hierarchy Process

The AHP, developed by Dr. Thomas Saaty back in 1980, is a tool to help solve technical problems, such as the decision making process. To do this, the AHP takes an approach based on scale to quantify priorities in a relative manner for a given set of options. It does so by deconstructing a problem to the level of pairwise comparisons (for example, comparing option A with option B regarding a specific criteria), and merges the results systematically [26].

The first step to use this method is to define the different options available as well as different criteria against which we should evaluate each option. These factors could vary, but they should all have a role in the selection process [26].

With the criteria defined, the next step is to perform a relative evaluation between the criteria. This is done using a scale system, in which the number 1 indicates equal importance between the criteria, whilst the number 9 is the maximum importance a criteria can have over another. Obviously, when a criteria is evaluated against itself, it results in the number 1 [26].

In order to calculate the weight of each of the criteria, the matrix needs to be normalized. This is done by dividing each of the values by the sum of its column. After this, the overall relative priority of each of the criteria can be calculated as the average of their respective rows. The highest priority value represents the most important criteria [26].

With these values calculated, it is necessary to validate the consistency of said values through the Consistency Ratio (CR). The values are considered consistent if

the CR is lower than 0.10, or 10%. CR is calculated through the division between the Consistency Index (CI) and the Random Consistency Index (RI), like shown in Equation 2.1.

$$CR = \frac{CI}{RI} \tag{2.1}$$

The CI can be calculated through Equation 2.2.

$$CI = \frac{\lambda_{max} - n}{n - 1} \tag{2.2}$$

Where n represents the number of criteria (4) and λ_{max} represents the principal eigenvalue. To calculate it, it is required to first multiply the non-normalized pairwise comparison matrix by the criteria priority vector.

Next, it is needed to divide each cell value of the resulting vector by the corresponding criteria priority vector cell. The average of these values is the principal eigenvalue. With the value calculated, it is then possible to calculate the Consistency Index through the formula presented above.

The Random Consistency Index can be obtained from Table 2.1, with n standing for the number of criteria.

Table 2.1: Random Consistency Index Values

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Random Consistency Index	0	0	0.58	0.9	1.12	1.24	1.32	1.41	1.45	1.49

With both the CI and the RI, the Consistency Ratio can then be calculated. If the value obtained is lower than the limit of 0.1, the values are considered consistent.

The next step is to evaluate the different platforms presented relative to each other regarding each of the specific criteria. This is done in the same manner that the criteria were compared against each other.

The final step is to gather all of the information obtained in a table and utilize it to determine the best option to take. This is done by multiplying the resulting matrix by the vector of the criteria relative priorities. The highest value obtained represents the best option to take.

2.7 Value Proposition

Nowadays, there are many kinds of definitions as to what a value proposition is. The most widely used one is that a value proposition is a promise made by a company to their customers, a promise that they will deliver a group of benefits that create value for the customer. In other words, it is a written statement that focuses the company's market activities onto customer elements that heavily influence the customers' decisions, thus leading them to prefer the company's product(s) over their competitors [27].

In order to help companies visualize the value of a certain product or set of products, the following value proposition canvas can be used, shown in Figure 2.6 [28]:

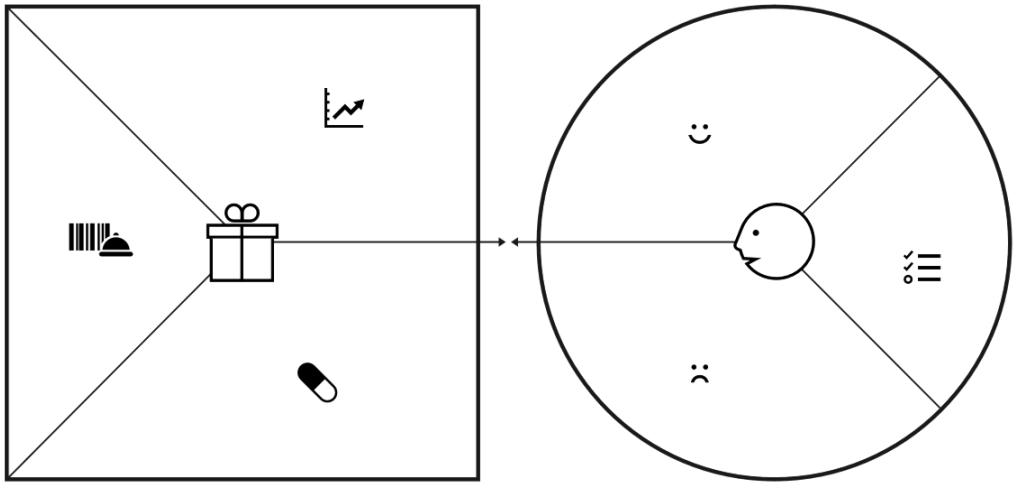


FIGURE 2.6: Value Proposition Canvas [28]

The left side represents the product or service, whilst the right side represents the customer. The two sides of the canvas are explained in the following subsections, starting with the product / service.

2.7.1 Value Map

The left side of the canvas, the value map, represents the features and characteristics of a specific value proposition. It contains the different products and services included in the proposition, the benefits they bring, and the pains they relieve [28].

Products / Services - On the left side of the value map, a list containing the products and services around which the value proposition is being built is presented. It can be one product or service, or a group of them [28].

Gain Creators - On the top side, the gain creators are presented, consisting of a description of how the products and services referenced are able to create customer gains and benefits [28].

Pain Relievers - On the bottom, the pain relievers are detailed, showing how the products and services and help alleviate the customers' pains [28].

As seen from this explanation, the value map is a response to the customers' pains and desires. And so, the following subsection explains the other end of the canvas in more detail.

2.7.2 Customer Profile

On the other side of the canvas, the customer profile is presented. It describes a specific customer segment in a structured and detailed manner. It can be separated into the customers' jobs, their gains, and pains [28].

Customer Jobs - On the right side of the customer segment, a list of tasks that the customers are attempting to perform in their work or even their daily lives is presented [28].

Customer Gains - On top, the customer gains list a group of benefits that customers either require or expect, or possibly even additional benefits for them. Gains can include functional utility, social gains, positive emotions, and cost savings [28].

Customer Pains - The bottom side of the customer segment presents different aspects that trouble or annoy the customers before, during, and even after trying to get a job done. Bad outcomes associated with the poor or no execution of a job are also included [28].

2.8 Quantitative Evaluation Framework Model

the QEF model is utilized in order to measure and evaluate the quality of the solution that has been developed. This method of evaluation, as proposed by Escudeiro and Bidarra [29], allows users to verify and keep track of the quality of the project during its development, and in doing so, showing where the users should focus their attention in order to detect and fix errors in advance.

To apply the QEF model to a project, the concept of quality must be thought of as a three-dimensional space. Each of the dimensions has a criteria assigned to it that will be used to evaluate the project in order to assess its quality, and these criteria can be grouped into different sets that possess different themes. These sets are referred to as a domains, and as such, there are three different domains present in this model [29]:

- **Technical Domain** refers to the characteristics of the project in terms of its features and capabilities.
- **Ergonomic Domain** refers to the usage of the project, the tools, and the features provided by it.
- **Pedagogic Domain** refers to the learning and knowledge acquired through the use of the project.

In order to evaluate the project, criteria belonging to these domains must be chosen in order to properly measure the performance of the project. Examples of criteria are technical aspects, adaptability, usability, interactivity, audio, video, and so on. The chosen criteria are referred to as dimensions. After choosing the criteria to use for the different dimensions, the requisites must then be separated according to the dimension they relate to the most.

Then, they are separated once more, this time into different factors inside of the dimension, regarding the role they play in the project. After doing this, each requisite needs to be given a relative importance, which consists of a value from 0 to 10, where 0 indicates that it is irrelevant, and 10 means that it is crucial to the project. This value is considered to be the requisite's weight in the overall rating of the project [29].

After all the weights are defined, the user then needs to evaluate the completion rate of each of the requisites. This is done with a percentage, ranging from 0% to

100%, usually done in steps of 25%, with set criteria in order to give this evaluation. After each requisite has their completion rate, the factor they belong to can then be evaluated using Formula 2.3 [29].

$$factor\ n = \frac{1}{\sum_m \times w_m} \times \sum_m (w_m \times c_m) \tag{2.3}$$

Where m is the number of relevant requisites within the factor, wm is the weight of the requisite and cm is the completion rate of the requisite [29].

The relative evaluation of the dimension for the overall rating of the project can then be calculated with Formula 2.4 [29].

$$dimension\ d = \sum_k (p_k \times factor_n),\ \sum_k (p_k) = 1,\ and\ p_k \in [0, 1] \tag{2.4}$$

Where k is the number of relevant factors and pk equals the weight of the factor within the dimension, calculated by dividing the sum of the weights of the factor's requisites by the total weight of the requisites in the dimension [29].

The global deviation is calculated through Formula 2.5 [29].

$$D = \sqrt{\sum_j (1 - \frac{Dim_j}{100})^2} \tag{2.5}$$

Where j is the number of dimensions. Finally, the overall quality of the system, in percentage, is given through Formula 2.6 [29].

$$Q = (1 - \frac{D}{\sqrt{n}}) \times 100,\ Q \in [0, 100] \tag{2.6}$$

2.9 Summary

In this chapter, a study regarding the context of the problem and the goal is presented, including the available technologies for the development of a solution that meets the requirements presented at the start of the project as well as additional requirements that have appeared throughout the development of the solution. Furthermore, related studies were also mentioned, detailing how they relate to the solution under study, whilst also showing that other companies besides Synopsys have shown interest in automating the regression testing process, both a few years back in 2014 [22] as well as more recently in 2019 [24], thus showing that this goal of automating regression tests has been, and still is, a general interest of tech companies.

Chapter 3

Analysis

This chapter presents a value analysis as well as a list of all of the requirements of the system under study. For a better understanding of the information presented in this chapter, it is recommended to refer to Chapter 2 for information regarding the concepts mentioned.

3.1 Customer Value

The concept of customer value is one that holds many definitions, as it depends from person to person. Some of the definitions state that the customer value is a comparison between the benefits and sacrifices of acquiring and utilizing a certain product or service. The benefits represent the utility or resources that can be obtained from the use of the product / service, while the sacrifices encompass the downsides, such as adaptations that are required, costs, risks, and so on. This is what is called the perceived (customer) value - What advantages and disadvantages, benefits and sacrifices, the product or service presents or demands, from the customer's perspective. This evaluation influences the customer's decision regarding whether or not to acquire and use the product or service [30].

For the project under study, the benefits consist of the automated running of regressions in order to verify the integrity of new releases, and the setup of regressions to run on a schedule without any further human interaction. As for the sacrifices, these include the use of a strict email format in order for releases to be automatically tested, and the requirement of a setup process in order to make use of the regression schedule configuration. Since the sacrifices consist of only a set of rules to follow and an initial setup, the benefits far outweigh the sacrifices in the long run, which translates into a positive perceived value for the customer.

3.2 Value Analysis

This section focuses on the analysis of the solution opportunity being studied, as well as its value. This analysis will be performed according to the New Concept Development Model, designed by Koen et al. [25], and presented in Section 2.6.

3.2.1 Opportunity Identification

Regression testing is a major part of the development of a product, as it ensures the quality of its features, both old and new. However, this process is manual most of the time, which means that an immense amount of person-hours is required in

order to verify and ensure the quality of a product. This alone makes automating the regression testing process a major benefit for many companies, as the cost of the production for a solution will naturally outweigh the testing costs, and the system will provide an increase in efficiency and productivity.

3.2.2 Opportunity Analysis

In order to better understand the value of this opportunity at *Synopsys*, a survey was conducted in order to get the worker's thoughts on the problem and the proposed solution. A total of eight questions were asked - two questions to help identify the audience that participated in the survey, four questions regarding the problem and the solution proposed, and two more final questions where participants could express their major problems with the current process of running regression tests, as well as suggestions for other improvements. In total, 22 people answered the survey.

Audience Questions

These questions help perceive the experience of and the area in which the respondents work in.

Question 1 - How long have you been working as a digital design / verification engineer?
Available Answers - 0-2 years, 2-5 years, 5-8 years, 8-12 years, 12+ years.

Question 2 - Which of these matches your area more closely?
Available Answers - Design, Verification.

The design team is in charge of creating and releasing the labels to be implemented in the chips, while the verification team is responsible for testing said labels to ensure quality. This information helps understand if both teams give certain aspects of the solution more value than to others. Furthermore, the question regarding years of experience also helps understand the average experience of the audience - If an user with more years of experience responds to the next set of questions with positive answers, that would mean that the problem under study applies to them and they would appreciate a solution.

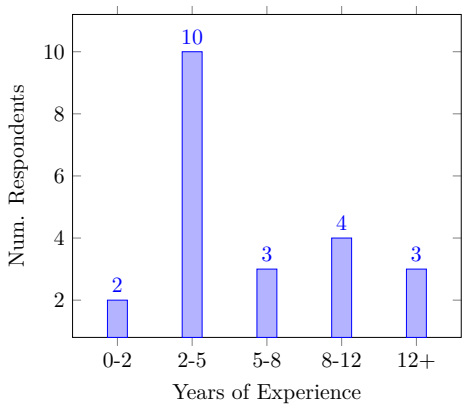


FIGURE 3.1:
Responses to
Survey Question 1

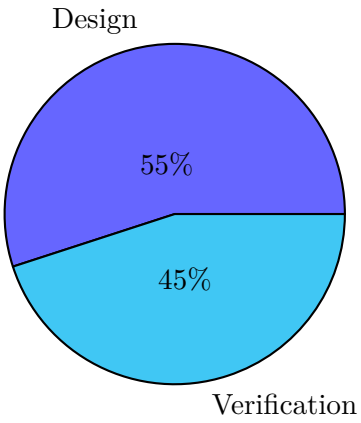


FIGURE 3.2:
Responses to
Survey Question 2

As seen in Figure 3.2, out of the 22 respondents, 12 work in design, while 10 are in charge of verification. This will be taken into account for each question in the following subsection in order to see which team values the solution more and present possible reasons as to why. Furthermore, in Figure 3.1 also possible to see that a vast majority of the respondents have 2 to 5 years of experience, with many others having over that amount, meaning that the group surveyed has an average experience of well over 2 years - showing that the respondents have been in this line of work for a while now.

Problem / Solution Questions

These questions actually pose statements that help evaluate the value of the solution itself, in the opinion of the respondents. The answers available range from "Completely Disagree" to "Completely Agree", on a scale of 1 to 5. The average value for each question will be calculated in order to evaluate the importance of the matter being asked about, with 1 being the absolute worse and 5 being the absolute best. Question 3 and 4 encompass the initial scope of the project, while question 5 and 6 refer to additional features implemented.

The opinions will also be separated into the two teams that submitted them (design and verification), and their average score will be calculated. However, only the overview of all the scores will be presented in a graph in this section, to avoid saturating this section with too many graphs. The excess graphs can be seen in Appendix A.

Question 3 - Would it be useful to have a system in which configurable regression tests are performed on labels as they're released, automatically?

Question 4 - Would it be useful to have all the results of the tests run with this system available on an online dashboard for users to visualize more easily, featuring tracking history through which users could also see the results of past tests?

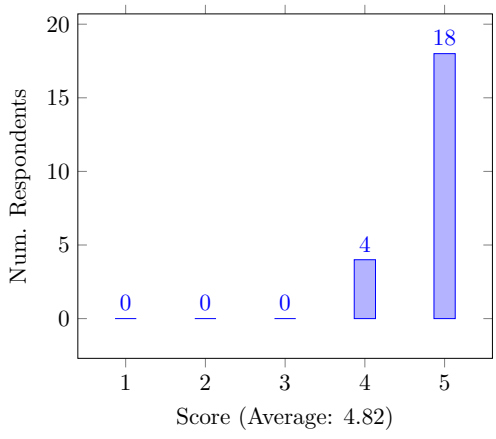


FIGURE 3.3: Responses to Survey Question 3

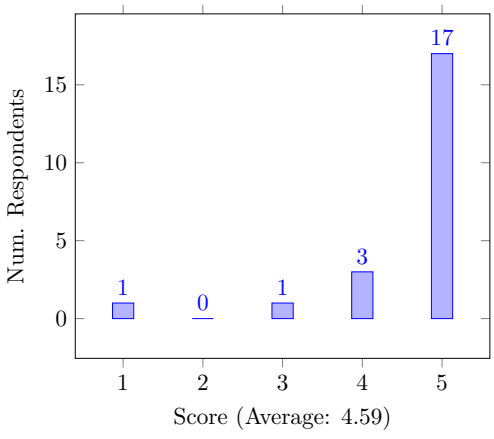


FIGURE 3.4: Responses to Survey Question 4

Figure 3.3 shows that the initial goal of automating regression testing after a new release is published is something that would be appreciated by the group of respondents, seeing as no individual answer got a score lower than 4, and the average is

near the maximum score of 5. The design team scored an average of 4.67 while the verification team got a perfect average score of 5 - both teams are very interested in this aspect.

Whilst not possessing the highest of scores, Figure 3.4 demonstrates that the tracking of results on an online dashboard is also appreciated, as it facilitates keeping track of and verifying the progress on the verification of a release. Due to this being the responsibility of the verification team, it explains the 4.9 average rating given by it when compared to the design team's rating of 4.33.

Question 5 - Would it be useful to write down the information for a regression on a table and have it run automatically, be it a repetitive schedule (such as every Wednesday), or a single run for a specific date (for example, in 3 days from now)?

Question 6 - Would it be useful to have an alternative system in which the user can use the information on the table in order to run a regression immediately with a single click?

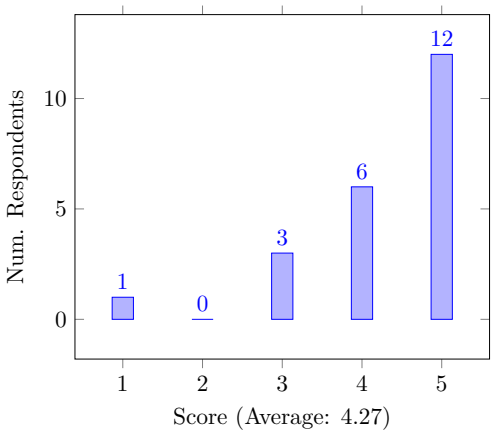


FIGURE 3.5:
Responses to
Survey Question 5

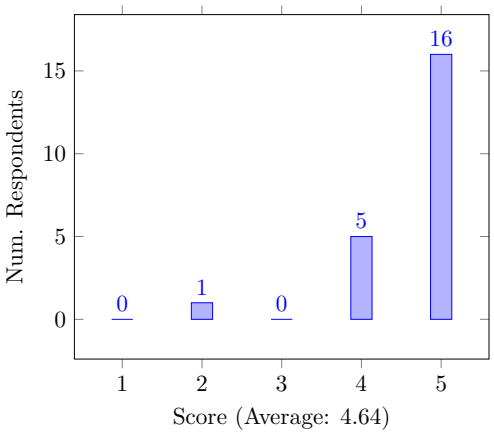


FIGURE 3.6:
Responses to
Survey Question 6

This question has the biggest difference in averages provided by the teams, with 4.7 for the verification team and 3.92 for the design team. This can also be seen through the differences in the responses in Figure 3.5 The reason for this is that the design team likely isn't interested in scheduling tests, only performing them before releasing a label, whilst the verification team's goal is to extensively test the release and ensure its quality, making it more valuable for them.

This last question is the only one in which the design team got a higher average score than the verification team, obtaining an average of 4.67 when compared to the verification team, who got 4.6. The difference may be small, but it is still present. Figure 3.6 shows that this feature would be highly desirable as well.

From the scores obtained, it is possible to see that the solution provides immense value to the company, seeing as no average score of each question was below 4.2. It is also possible to see that the verification team gives this solution more value than the design team, yet the system is applicable for both teams.

Problems and Improvements

These optional questions help identify and understand other problems caused by the manual running of regression tests whilst also confirming other issues that have already been mentioned. Furthermore, they also help identify even more opportunities and improvements for the solution.

Question 7 - When running regression tests manually, what is / are the biggest problems you encounter?

For this question, the following answers were given:

- Time wasted in setting up workspace and disk space management.
- Test configuration.
- The (current) server used to check the results is slow and has a bad UI.

The solution developed aims to fix all of these problems and more, as it sets up areas and manages disk space automatically, allows easier configuration of tests and regression arguments via configuration files or a visual interface such as a table of information, and displays the results on an easily accessible, reliable platform (Confluence), in an organized manner.

Question 8 - Do you have any recommendations to improve or expand a system like the one explained above?

Despite the question asking for suggestions, some users also gave their opinion on the features that were already presented. For this question, the following suggestions were given:

- "Run a simple test on a set of labels when a new change is submitted."
- "Check the current status via link."

For the solution presented, the second suggestion was addressed by providing users with a table of information that serves as an overview of the results, allowing users to click on a link to read the full details of the tests run. The first suggestion, as it was given fairly late in development, it was considered future work.

Other feedback received includes:

- "A graphical interface for running regressions sounds useful."
- "A dashboard with the results of all the regressions run would improve visibility."

All of these topics were addressed in the questions presented as well as the brief presentation of the project given in the survey. The feedback received shows that users would make good use of the solution, making it a great opportunity to explore.

3.2.3 Idea Generation and Enrichment

After analysing and reflecting over the problem at hand and the different tools available for the development of the automated system that solves it, three ideas were considered. All of them have the common aspects of fetching information from emails and uploading it to an online dashboard, utilizing the same tools for those purposes, with the big difference being the CI / CD platform to use in order

to automate the process, this component representing the majority of the project. This being said, the following tools, mentioned in Section 2.3, were studied:

- Jenkins.
- Gitlab.
- Bamboo.

3.2.4 Idea Selection

For the solution under study, in order to select the idea to pursue for the development of said solution, the AHP technique was applied.

Analytical Hierarchy Process

The first step to use this method, explained in Section 2.6.6, is to define the different options available as well as different criteria against which we should evaluate each option [26]. The different options available are Jenkins, Gitlab, and Bamboo. The criteria picked in order to evaluate these platforms were:

- **Performance** - The overall performance of the platform, including speed and robustness when using the platform.
- **Maintainability** - Evaluates how easy it is to maintain the platform when it is used by more and more users.
- **Expandability** - Represents the potential of expansion of the platform, to support more users or implement more features.
- **Cost** - The monetary costs associated with the platform.

Table 3.1 presents how important the criteria on the left are over the criteria at the top. This table is called a pairwise comparison matrix.

Table 3.1: Pairwise Comparison Matrix

Criteria	Performance	Maintainability	Expandability	Cost
Performance	1	$\frac{1}{2}$	1	3
Maintainability	2	1	2	6
Expandability	1	$\frac{1}{2}$	1	3
Cost	$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{3}$	1
Sum	4.333	2.17	4.333	13

The normalized matrix, as well as all of the relative priorities, are present in Table 3.2:

Table 3.2: Pairwise Comparison Matrix (Normalized) and Overall Relative Priorities

Criteria	Performance	Maintainability	Expandability	Cost	Priorities
Performance	0.231	0.231	0.231	0.231	0.231
Maintainability	0.462	0.462	0.462	0.462	0.462
Expandability	0.231	0.231	0.231	0.231	0.231
Cost	0.077	0.077	0.077	0.077	0.077
Sum	1	1	1	1	1

From the table presented, it is possible to conclude that the most important criteria is Maintainability, proven by its relative priority of 0.449 when compared to the other criteria.

It is now necessary to validate the consistency of the values obtained through the calculation of the Consistency Ratio (CR). As mentioned in section 2.6.6, this value can be calculated through equations 3.1 and 3.2.

$$CR = \frac{CI}{RI} \tag{3.1}$$

$$CI = \frac{\lambda_{max} - n}{n - 1} \tag{3.2}$$

Where n represents the number of criteria (4) and λ_{max} represents the principal eigenvalue. To calculate it, it is required to first multiply the non-normalized pairwise comparison matrix by the criteria priority vector. like so.

$$\begin{bmatrix} 1 & \frac{1}{2} & 1 & 3 \\ 2 & 1 & 2 & 6 \\ 1 & \frac{1}{2} & 1 & 3 \\ \frac{1}{3} & \frac{1}{6} & \frac{1}{3} & 1 \end{bmatrix} \cdot \begin{bmatrix} 0.231 \\ 0.462 \\ 0.231 \\ 0.077 \end{bmatrix} = \begin{bmatrix} 0.924 \\ 1.848 \\ 0.924 \\ 0.309 \end{bmatrix}$$

By dividing each cell value of the obtained vector by its corresponding criteria priority vector cell, and calculating the average of the resulting values, the principal eigenvalue is obtained.

$$\lambda_{max} = \frac{\frac{0.924}{0.231} + \frac{1.848}{0.462} + \frac{0.924}{0.231} + \frac{0.309}{0.077}}{4} = 4.003$$

$$CI = \frac{\lambda_{max} - n}{n - 1} = \frac{4.003 - 4}{4 - 1} = \frac{0.003}{3} = 0.001$$

The Random Consistency Index can be obtained from Table 2.1, with n standing for the number of criteria. For this scenario, the value to use for the Random Consistency Index is 0.9.

$$CR = \frac{CI}{RI} = \frac{0.001}{0.90} \approx 0.001$$

Since the value obtained is lower than the limit of 0.1, the values are considered consistent.

The following evaluations, present in Tables 3.3, 3.4, 3.5, and 3.6 were performed based on past experiences, reviews, and research.

Table 3.3: Performance Priority Matrix

Performance	**Jenkins**	**Gitlab**	**Bamboo**	**Priorities**
Jenkins	1	$\frac{1}{2}$	$\frac{1}{2}$	0.200
Gitlab	2	1	1	0.400
Bamboo	2	1	1	0.400

Table 3.4: Maintainability Priority Matrix

Maintainability	**Jenkins**	**Gitlab**	**Bamboo**	**Priorities**
Jenkins	1	2	1	0.400
Gitlab	$\frac{1}{2}$	1	$\frac{1}{2}$	0.200
Bamboo	1	2	1	0.400

Table 3.5: Expandability Priority Matrix

Expandability	**Jenkins**	**Gitlab**	**Bamboo**	**Priorities**
Jenkins	1	6	3	0.667
Gitlab	$\frac{1}{6}$	1	$\frac{1}{2}$	0.111
Bamboo	$\frac{1}{3}$	2	1	0.222

Table 3.6: Cost Priority Matrix

Cost	**Jenkins**	**Gitlab**	**Bamboo**	**Priorities**
Jenkins	1	6	3	0.667
Gitlab	$\frac{1}{6}$	1	$\frac{1}{2}$	0.111
Bamboo	$\frac{1}{3}$	2	1	0.222

In Table 3.7, P represents the criteria priority.

Table 3.7: Global Priorities Matrix

Criteria / Platform	**Performance P = 0.231**	**Maintainability P = 0.462**	**Expandability P = 0.231**	**Cost P = 0.077**
Jenkins	0.200	0.400	0.677	0.677
Gitlab	0.400	0.200	0.111	0.111
Bamboo	0.400	0.400	0.222	0.222

$$\begin{bmatrix} 0.200 & 0.400 & 0.677 & 0.677 \\ 0.400 & 0.200 & 0.111 & 0.111 \\ 0.400 & 0.400 & 0.222 & 0.222 \end{bmatrix} \cdot \begin{bmatrix} 0.231 \\ 0.462 \\ 0.231 \\ 0.077 \end{bmatrix} = \begin{bmatrix} \mathbf{0.436} \\ 0.219 \\ 0.345 \end{bmatrix}$$

From this study, it is possible to conclude that Jenkins is the best tool to use for the development of the solution.

3.2.5 Concept Definition

The final objective is to develop and implement a system capable of automatically setting up areas and running regression tests automatically, according to configurations provided by users. Furthermore, a system for configuring a regression and run it immediately or on a schedule is also a feature that would be used, and thus is also part of the objective.

3.3 Value Proposition

Taking into consideration all of the information presented in Section 2.7, the following value proposition canvas, present in Figure 3.7, was filled according to the problem presented in this thesis along with the solution produced. It is worth noting that, as like in nearly any project, the initial plan for the solution was changed during development. This resulted in additional services, gains, and more aspects beyond the ones already fulfilled by the initial proposal.

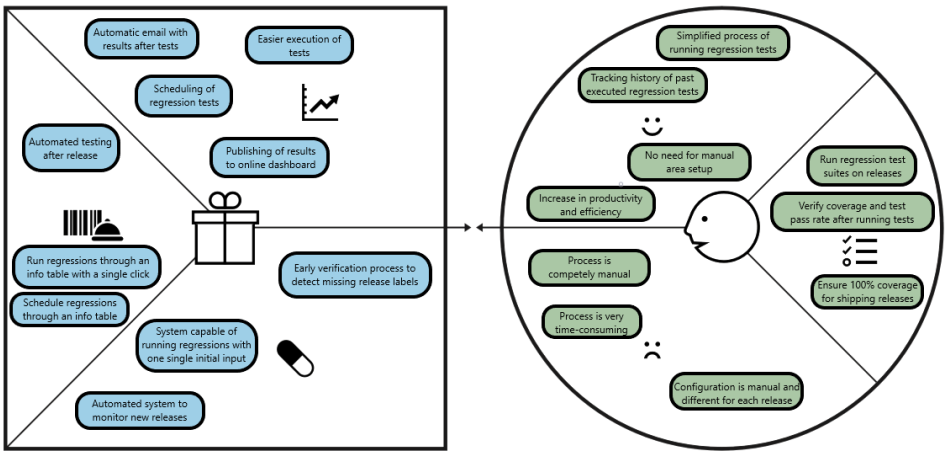


FIGURE 3.7: Filled Value Proposition Canvas

As seen from Figure 3.7, the services provided by the solution are the automated tests, the capability of running regressions through a configurable table with a single click, and use the same table to schedule regressions. These services provide the benefits of executing tests more easily and automatically, also allowing the scheduling of regressions and sending automatic emails with results as well as publishing said results on a dashboard. These benefits provide a simpler way of running tests while also automating them, increasing the productivity and efficiency, removing the need

for manual setup of areas for testing, and also providing tracking of run regression tests.

The services provided also help relieve the users' pains of a completely manual and time-consuming process, as well as the problem of manual configuration that is different for each of the releases. The services provide automated monitoring of new releases, making it faster, detecting missing labels to avoid running with invalid data, and instead warning the user about it, and its capability of running regressions start to finish with a simple configuration helps streamline the testing process and make it more efficient.

3.4 Requirements

The goal of this section is to identify and classify the different requirements presented for the development of a solution that resolves the problem under study. The different requirements have been organized and classified into different categories:

- **Functionality** - Requirements in this category represent the base functionality of the solution - actions it must perform and features that must be available that contribute toward the goal of resolving the problem.
- **Adaptability** - These requirements focus on making the solution capable of conforming to the user's needs, allowing the user to configure it and use it the way they want.
- **Usability** - Lastly, this category presents requirements that assist the user in using the application, be it by supplying the user with useful and relevant information, or by prohibiting the misuse of the solution.

Within each of these categories, requirements are then be placed into a factors that describes the role they play in the system developed. This helps viewers identify the general purpose of a set of requirements from a quick glance. The reason for the requirements being split up in this manner is due to the fact that, in Chapter 6, the solution is evaluated according to the QEF, which assigns a certain score to each requirement and, based on their completion percentage, calculates an overall grade for the system being evaluated. This framework is explained in further detail in said chapter.

With all of this in mind, tables 3.8, 3.9, and 3.10 present all of the requirements for the solution, both on an initial phase, as well as additional requirements that appeared during the development of the solution, organized into categories and subcategories (or when evaluated using QEF, dimensions and factors).

Table 3.8: Functionality Requirements

Category	Sub-Cat.	Requirement
Functionality	**Information Collection and Validation**	System must collect email data automatically
		System must create file with email data
		System must validate data provided in email
	Space Management	System must confirm that the available disk space is enough
		System must clean up older files to free disk space
	Test Setup and Running	System must compile given configurations to run regression
		System must set up test area without human input
		System must run tests in area created
	Tracking and Result Publishing	System must track regression ID run
		System must tag files used to allow users to replicate the area in future (tracking / debug)
		System must save file with regression details to SCM platform
		System must send email stating test status
		System must upload results and information to online dashboard
	User Functionalities	System must allow users to schedule regressions with Confluence table
		System must allow users to run regressions immediately using Confluence table
	System Requirements	System must verify that all necessary information is introduced
		System must reuse the system developed for the automatic email flow
		System must run pipeline on a node owned by the user (nodes are used to run jobs)

Table 3.9: Adaptability Requirements

Category	Sub-Cat.	Requirement
Adaptability	**Configuration**	System must allow configuration of path to run tests in
		System must allow configuration of timeouts for pipelines (avoid stuck tests)
		System must allow configuration of email formats and content
		System must allow configuration of file view to synchronize
		System must allow configuration of client names to use for file synchronization
		System must allow global configurations for regression tests
		System must allow specific configurations for regression tests based on project
		System must allow users to configure the disk, path, and node to use
		System must give priority to specific configurations, completing with global
		System must allow users to configure the schedule for the regression
		System must allow configuration of more than one disk to use and support different grid locations
		System must allow configuration of all parameters required for regression
		System must allow configuration of simulation type and verification environment
		Configurations provided in table should override project specific and global configurations
	Expandability	System must be expandable and modular in order to allow future integration of other environments / projects

Table 3.10: Usability Requirements

Category	Sub-Cat.	Requirement
Usability	**Inform User**	System must inform admins if any part of the pipeline fails
		System must inform users of emails detected
		System must inform users of disk space used and remaining
		System must inform if space is running too low
		System must inform users of missing data in email
		System must inform users when their regression is processed/started
	Prevent Problems and Misuse	System must not read the same email twice
		System must not attempt to run tests with missing information
		System must stop if space is not enough
		System must cancel tests based on timeout (avoid stuck tests)
	Setup Process	Setting up a regression on the table must be simple and intuitive
		A template for setting up a regression must be provided
		A set of instructions on how to set up the system must be provided

3.5 Summary

In summary, this chapter presents an analysis of the project at hand, evaluating its value as well as the many benefits introduced by it, yet not forgetting to point out the downsides to using it. Furthermore, a questionnaire was conducted in order to gather the opinion of several potential users of the solution, the results further proving the value of the solution. Lastly, the requirements were presented and classified according to different metrics for later use in Chapter 6, where the resulting project is evaluated.

Chapter 4

Design

The goal of this section is to explain the design that was constructed in the planning phase of the project. As this project mostly consists of Jenkins jobs, the structure of said jobs is presented along with their respective responsibilities in the overall flow of the project.

It should be noted that designs are always prone to changes upon moving to the development phase, especially in the current context, in which the Action Research methodology is utilized, as the appearance of new requirements may imply changes, be it modifications, additions, or removals of certain components of the design.

When constructing the design, something to take into consideration is the fact that, due to additional requirements introduced during the development, the flow must be easily adaptable in order to fulfil these requirements. This is represented in the Analytical Hierarchy Process as both the Expandability and Maintainability factors (maintain the current flow while expanding to support additional requirements). In order to provide a flow that can easily be built upon and added onto, the process was split into three phases:

- **Starting Phase** - In this phase, the information used for the regression testing is obtained and collected. Based on this information, the area in the disk (directory) for each regression must be created, with cleanups being performed automatically should more space be required.
- **Configuration Phase** - After the starting phase, configurations are loaded. These configurations are defined in configuration files present on the SCM platform. Furthermore, this phase also allows for input from the previous phase, should customized configurations be used for a specific regression.
- **Test Run Phase** - Finally, in this phase, the regression is now executed by compiling all of the information gathered, be it from the start phase or the configuration phase, into a command that is used to run the script responsible for the regression testing. The phase will stay running until the regression is complete, after which it relays the results to the team and records the results on an online dashboard as well as the SCM platform.

Each of these phases consists of a separate pipeline job, which represents a set of instructions given to Jenkins for it to perform. The only exception to this rule is the starting phase, consisting of two pipelines instead - One for gathering the information required, and the other for cleaning up older regressions. This allows users to use the latter alone to clean up their area. Taking all of this into consideration, the final design is shown in Figure 4.1.

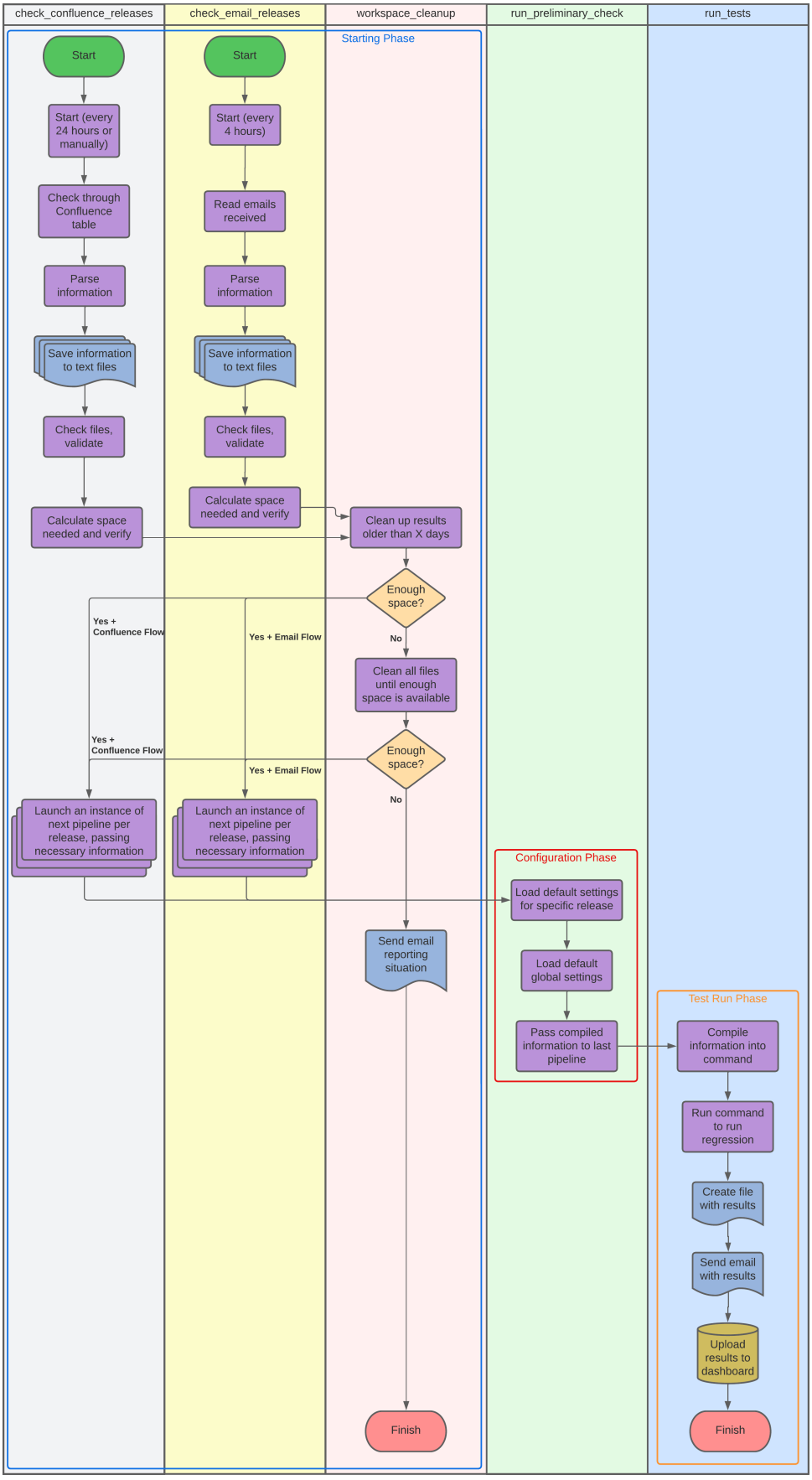


FIGURE 4.1: Design Workflowchart

Visible in Figure 4.1 are all the pipelines used in this project. It should be noted that check_email_releases and confluence_regression_check consist of two different starting pipelines, with the remaining pipelines making up the rest of the flow. This results in the pipeline flow demonstrated in Figure 4.2.

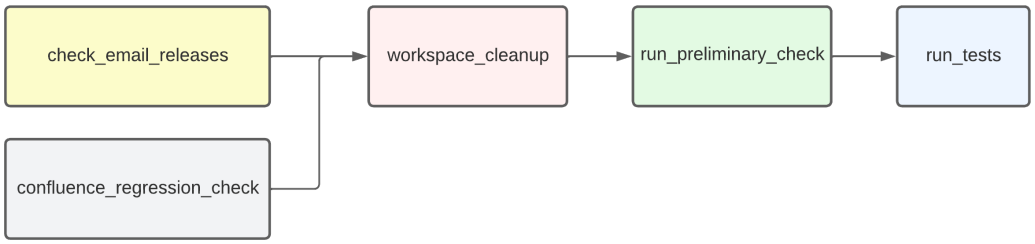


FIGURE 4.2: Pipeline Workflow

It should be noted that, for the goal of simplicity, the interaction between the first pipeline of the sequence and workspace_cleanup in which the former awaits the successful run of the latter is not represented.

4.1 Email Workflow

As seen in Figure 4.2, the flow that makes use of the release emails starts with a pipeline titled check_email_releases. This pipeline is programmed to run every four hours. A Python script is called by the pipeline in order to collect the emails received in the past 12 hours (configurable) by accessing an RDF Site Summary (RSS) feed.

RSS feeds, usually used by news and blog websites, are a means of describing web content that is available for distribution, from a certain publisher to their users. In other words, a website that pretends to publish some of its content is able to create a description of the content and indicate the location of the content on the website in the form of an RSS document. This document is then registered and published. Users are then able to view and read new posts or news from these RSS feeds [31]. While RSS feeds are mostly used for news and blog posts, it does not mean that they cannot be used for other purposes, such as new software releases, akin to what was mentioned regarding Synopsys's method of sharing new releases among the different teams. Furthermore, besides humans, scripts are also capable of accessing information from RSS feeds and use it to their own needs.

After obtaining the emails from the RSS feed, their content is parsed in order to create temporary files that contain the information required for each release to be tested. These files are then checked for any inconsistencies, and if a certain file is considered invalid, it will not be taken into consideration for the subsequent pipelines.

After all files are verified, the total space required to run the regressions is estimated based on configurable parameters. This estimation is done by assigning a "weight" to the configurations that may cause a regression to occupy more or less space in the disk. Based on the configurations, and a default minimum for each regression, the space needed for all the regressions found is therefore calculated. This value is then

passed into another pipeline, named workspace_cleanup, and pauses the execution of the first pipeline to assure that there is enough room for the regressions to run.

The workspace_cleanup starts off by cleaning regression results and files that have not been accessed in over a configurable number of days. It should also be noted that this pipeline runs no matter the amount of regressions found by the first pipeline, meaning that there could be no regressions found and it would still run. This is intentional, as this pipeline doubles as a periodic cleanup that gets rid of old results.

After the initial cleanup of old files, a check is performed in order to verify if the current space is enough. If so, the cleanup pipeline ends and the flow returns to the first pipeline. Otherwise, the cleanup pipeline cleans older regressions (but still more recent than the number of days mentioned earlier), starting with the oldest ones. Between every regression area cleaned up, the check is performed again. This step is repeated until there is enough room for the new regressions. If it is not possible to get the space required, then an email will be sent to the team reporting this incident, containing information about the time and location where it happened, the pipeline will end, and so will the check_email_releases pipeline.

If there's enough disk space however, the starting pipeline will then launch an instance of the run_preliminary_check pipeline for each release that was found in the first pipeline, passing the information of the respective release. In this pipeline, the configurations for the regression are loaded. In the Email Workflow, these configurations can be changed by editing or adding configuration files available on the SCM platform. There are two kinds of configuration files - Global and tech-specific. In this context, tech is referred to as a project, so this allows users to set different settings depending on the project release.

There can only be one global configuration file, but there can be one tech-specific configuration for each project. The latter takes priority over the former, meaning that its settings will be loaded first and completed with the global configuration. All of the information from the release and the settings used to run its regression are then passed to an instance of the run_tests pipeline, and the run_preliminary_check pipeline ends.

Finally, the run_tests pipeline takes all of the information it has received, compiles it into the command necessary to run the regression (a separate script, not part of this project, handles the running of regressions), and runs it. During the execution of the regression, the pipeline will continue running, meaning that users are able to check the status of the pipeline to view for how long the regression has been running as long as its status (the script outputs the number of tests remaining). After the regression is complete, the job will then save a file containing information about the regression run, which is structurally similar to the one used in the first pipeline, but with additional information. This file is submitted to the SCM platform for tracking purposes.

Furthermore, an email containing the results of the regression is sent to the team. Most of the information contained in the email is generated by the regression script, save for some information that the Jenkins script must extract from the log files. Lastly, the information contained in the email is also uploaded to an online dashboard on Confluence, allowing users to view all of this information, be it from new or old regressions, more easily.

In order to upload the information to Confluence, its REST API is used once more, this time to create new pages with information regarding the results of the regressions. This process is better explained in Chapter 5.

4.2 Confluence Workflow

In the previous flow, the confluence_regression_check pipeline is not mentioned as it is not used in the Email Workflow, instead being used in this flow titled as the Confluence Workflow. This pipeline replaces the very first job on the previous flow (check_email_releases), as the method for obtaining information is very different. Instead of accessing an RSS feed and extracting the information from emails, it instead accesses a table on a configured Confluence page in order to find the information needed to run regressions. The Confluence page also allows users to set their own configurations for the release without the use of a configuration file, thus making it more versatile for running specific test cases. Much like posting information to Confluence, this access is done by using the Confluence REST API.

REST APIs make use of endpoints to allow users access to data on websites. These endpoints must be configured, following a directory structure with different Universal Resource Identifier (URI). In order to obtain or modify data on web applications, REST API makes use of the Create, Read, Update, Delete (CRUD) operations. These operations are comparable to operations used by Structured Query Language (SQL) [32]:

- **Create (POST)** - Used to create new information, similar to the SQL operation INSERT.
- **Read (GET)** - Used to obtain information, similar to the SQL operation SELECT.
- **Update (PUT)** - Used to create update information, similar to the SQL operation UPDATE.
- **Delete (DELETE)** - Used to create delete information, similar to the SQL operation DELETE.

To be able to use the Confluence REST API, the users must first authenticate. In the event that a script makes use of the API, it too needs to authenticate in order to successfully perform CRUD operations on the web application. There are a few ways to do this, one of the most popular being through the Open Authorization (OAuth) system. This works by requesting a token that will allow the script to authenticate itself as an user. This means that every separate system running the script will require a token for authentication before being able to use the API [33].

After the authentication process, the script or user is then able to perform CRUD operations on the API. They are able to create pages and spaces, access and modify their contents, and delete them, among many other things. For the project developed, these operations are the most important. The page information is returned to the script or user in a JavaScript Object Notation (JSON) format, with one of the parameter being the page contents, in a HyperText Markup Language (HTML) format [33].

This process is important, as every user who wants to make use of the Confluence Workflow must get an OAuth token to authenticate themselves. However, since the Email Workflow is managed by a service account, only that account needs this token in order to access Confluence through the REST API.

This pipeline is triggered in two different ways - It can be run immediately and manually via a link that is constructed for each line of the table depending on the information specified in it, allowing users to set a regression to run and start it with one click, and additionally the pipeline is scheduled to run automatically once every day. The table includes a configuration for the days of the week or specific dates in which the specified regression should run. The automatic run of the pipeline will comb through the table to find the regressions it needs to run, and extract the information. After this, the confluence_regression_check pipeline also runs the workspace_cleanup pipeline in order to clean up the area of the regressions.

Despite the initial difference when it comes to collecting information, the rest of the flow is mostly identical between the two flows, seen as the information for each release is saved in identical temporary files. The only major difference in the rest of the flow between these two use cases is the fact that Confluence users are able to specify the settings that they want to use for the regression on the table itself, while the Email Workflow makes use of only the configuration files present in the SCM platform. The settings defined in Confluence take priority over the other two.

4.3 Overview of Resources Used

The diagrams presented at the start of this chapter focus on the Jenkins aspect of the flows, but there are more components involved in the flow than just the pipelines created in Jenkins. As such, an additional diagram is presented in Figure 4.3, showing how all the components interact with each other.

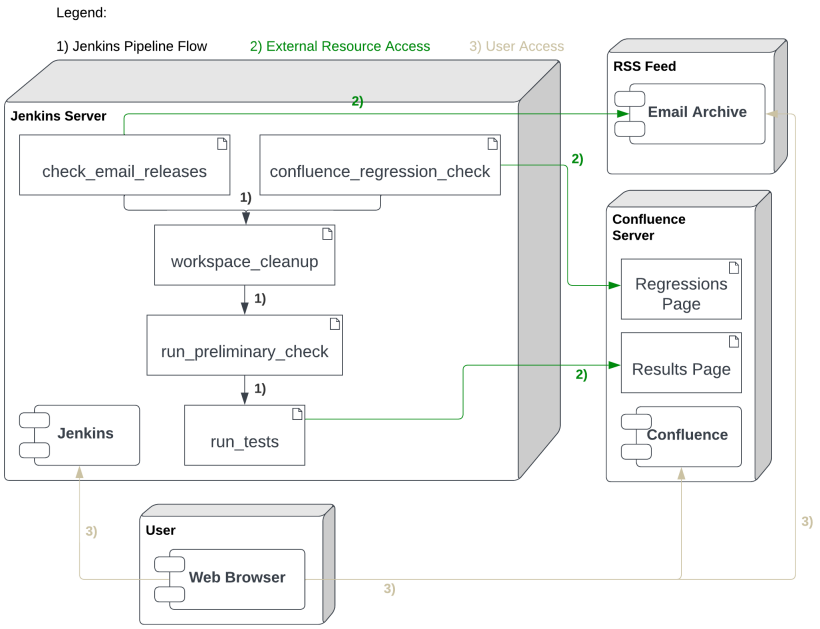


FIGURE 4.3: Deployment Overview (Design)

The components shown consists of the Jenkins jobs, the RSS feed, and the Confluence pages involved. Connections associated to the number 1, in black, show how the Jenkins system flows from pipeline to pipeline. Connections with the number 2, in green, represent accesses to external systems (RSS feed and Confluence). Finally, the connections marked with number 3, in beige, show how the user connects to each of the represented platforms. An advanced version of this diagram is presented in Chapter 5, taking into account the implementation done.

4.4 Summary

In summary, the design was developed with expandability and maintainability in mind. This is due to the ever-evolving nature of the project, as new requisites and possibilities appear around every corner. As such, in order to implement these new features, the project needs to be as flexible as it can be in order to allow for a smoother implementation. This is why the project presents three distinct phases, with configurations that can be changed at any time, and supporting scripts to establish the bridges needed between Jenkins and other resources, such as Confluence and the RSS feed.

Chapter 5

Implementation

This section will describe and detail all of the procedures and decisions taken in order to implement the solution according to the requirements provided by Synopsys, referred in Section 3.4.

Due to the existence of different flows of use for the developed project, this section will be separated into two major sections that describe the respective flows, with subsections explaining into further detail as to how they were implemented. The flows developed, as described per the requirements and the design presented, are the following:

- **Email Workflow** - The main flow of the solution. Upon the reception of an email of a new release, the solution should automatically launch a regression test making use of the information contained in the email.
- **Confluence Workflow** - A secondary flow introduced during development that allows users to specify their own sets of regression tests to run either immediately or on a customized schedule.

It should also be noted that, due to confidentiality reasons, most images will have confidential information blanked out, or replaced by placeholders.

5.1 Email Workflow Implementation

According to the design in Chapter 4, this flow is separated into three major phases - The Starting Phase, Configuration Phase, and Test Run Phase. In this section, the pipelines used to build up these phases will be described in detail. It should also be noted that the other flows reuse this sequence of pipelines to a certain extent, so in their respective sections, only the main differences when compared to the Email Workflow will be mentioned to avoid the repetition of information.

For easier interpretation of the process explained in the following subsections, the diagram present in Figure 5.1 represents an overview of how each of the Jenkins pipelines interact with each other, as well as the information transmitted between them and the method used to obtain it.

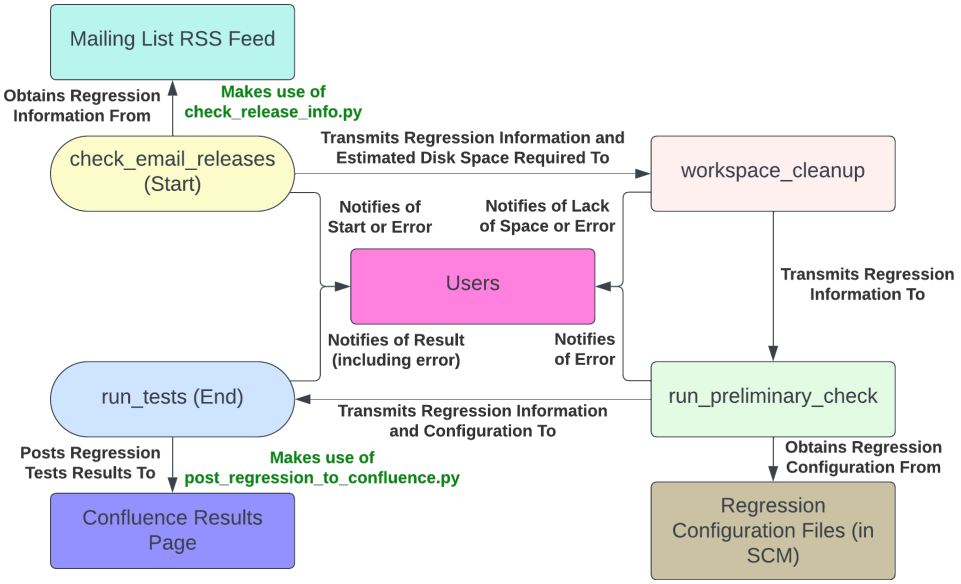


FIGURE 5.1: Information Flow Diagram for Email Workflow

5.1.1 Check Email Releases

The goal of the first pipeline of this respective flow is to gather the information provided by the emails sent when there is a new release that requires testing. Before doing this, however, seeing as the solution should be as customizable as possible, it was necessary to find a way to load a set of configurations to be used during the execution of the pipeline. Furthermore, these configurations should be available for the users to change, should a change in the flow be necessary. This is also useful during development, as changes can be made to the flow by changing the configurations and not the script that makes use of them.

The solution found was to use the P4 Jenkins plugin. P4 stands for Perforce, which is the SCM platform used by the company. Like many other plugins that establish a connection between Jenkins and other SCM platforms, this plugin allows the pipeline to automatically synchronize the files it uses with the files available on the platform. This is an oftenly used technique to automatically compile, build, and test projects available on these platforms by making Jenkins fetch the files from the platform upon the submission of new modifications, with posterior operations to ensure the correct functioning of the project. The P4 plugin allows users to specify which files they want to obtain from the SCM platform at the start of the pipeline.

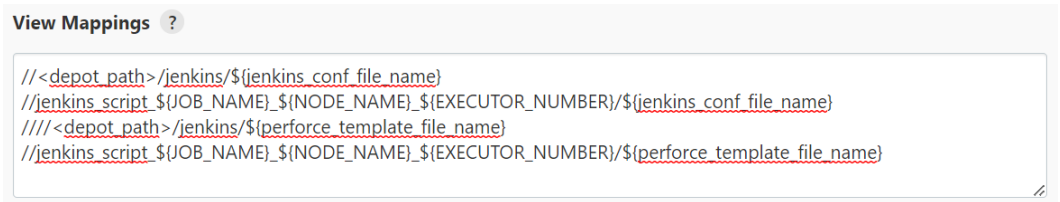


FIGURE 5.2: SCM Checkout Configuration

As shown in Figure 5.2, this plugin is used to synchronize two files from the repository that are of utmost importance to the pipeline - The Jenkins configuration file, and

the Perforce view file. The name of the file makes use of a Jenkins variable, so that it can be changed if the file is changed later on, and doing it this way ensures that the name of the file is changed for all pipelines by editing only one, as the file name variable is passed from pipeline to pipeline. The Perforce view file consists of a file that defines the names of the clients to be used and the view to use to fetch files with. A client consists of a new "agent" of sorts that is used to perform operations with the SCM platform, while the view consists of a mapping that indicates where the files will end up upon being synced. This is important not only to specify which files to sync and where they should be positioned, but also to keep the pipeline from fetching too many files and thus slowing the execution of the pipeline - With this being one of the problems with the previous, manual flow.

It should be noted that the SCM checkout performed at the start in order to obtain the Jenkins configuration file and the Perforce view template could also be used to obtain all of the other files necessary right at the start. However, should a change be required in the files that need to be synced (either more or less files are necessary), that would imply a change to the job's configuration directly, and so, for the sake of easier customization, the Perforce view file is utilized to define which files are required.

The Jenkins configuration file is a text file that consists of multiple different parameters used by the different pipelines on Jenkins and other scripts that said pipelines rely on. Depending on the pipeline in question, different settings are extracted. Listing 5.1 shows an example of some of the settings in the file, relevant to this pipeline. To edit a setting, it is simply required to change the content after its respective colon character. Comments detailing each setting are included. All of the available configurations and their purposes can be seen listed in Appendix B.

```
// Limit of emails read when using script
mailing_group_linklist_responses_number: 200

// Limit of hours ago from when emails should be read
mailing_group_linklist_responses_hours_number: 12

// Filters to look for to read an email
filter_list: "Design Labels", "Verification"

// Filters to exclude emails from being read
exclusion_list: "RE: ", "FW: "
```

Listing 5.1: Jenkins Configuration File

After synchronizing the necessary files and extracting the respective settings from the Jenkins configuration file, the pipeline then needs to find and parse the relevant emails. These emails are sent by the design team, and follow a template available to the different teams. Knowing this, it is possible to use the template as a basis in order to understand how to parse the required information. Furthermore, the emails are sent to mailing lists, which allows users to subscribe to said lists in order to receive the emails. The emails sent to the mailing list are viewable online through a private Synopsys website, which also features an RSS feed, which is crucial for the extraction of information via the use of scripts.

Upon starting the development of the solution, there was already an incomplete Python script developed by the company with the goal of extracting and parsing the information from the RSS feed of the mailing list mentioned. With this in mind,

to avoid scrapping something that was proven to be functional, this Python script was improved, making use of the logic that was already implemented. In order to obtain the information from the feed, three Python libraries are used:

- **Feedparser** - Used to read the RSS feed and obtain the latest emails and the links to their information on the archive.
- **Requests** - Used to send Hypertext Transfer Protocol (HTTP) requests, in order to access the links and obtain the HTML from those pages.
- **BeautifulSoup** - Used to parse through the HTML content and extract only what is necessary.

Figure 5.3 shows how these libraries interact with the each other and the RSS feed:

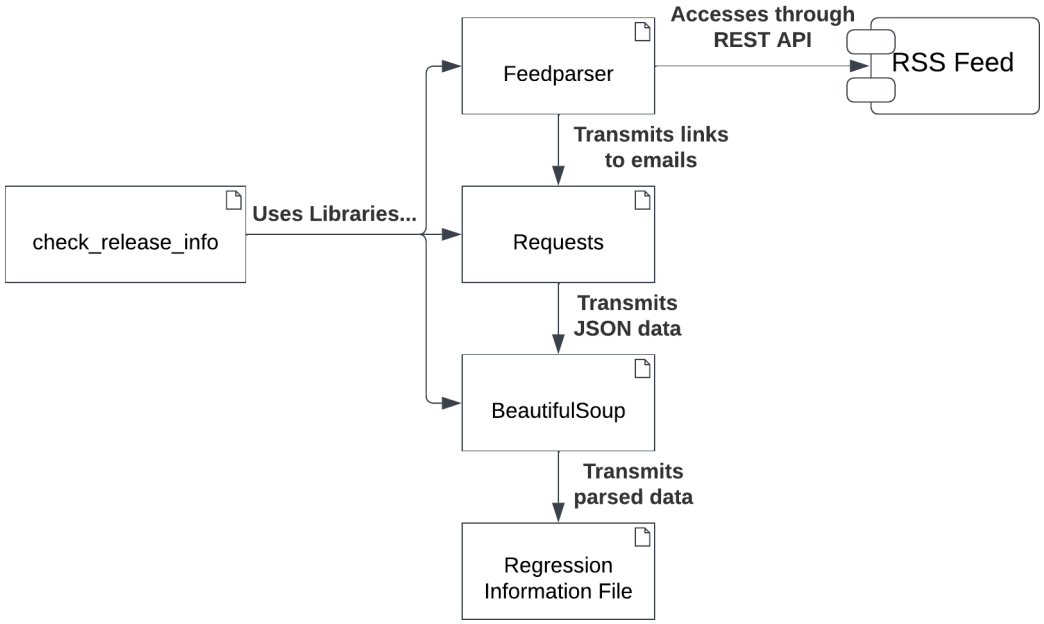


Figure 5.3: Interaction Between Python Libraries

Thus, in this step, the pipeline makes a call to execute the Python script, appropriately titled check_release_info, which starts by accessing the feed by using the Feedparser library and finding the links to the emails archived in the past twelve hours, this value being customizable in the configuration (shown in Listing 5.1). The reason for this default time period is that there is always a chance for the pipeline to fail, be it because it gets stuck running due to system overload and automatically terminates (as it features a timeout setting), or because the SCM is having issues. These have both occurred and are usually not caused by Jenkins itself, though there have been instances in which the Jenkins servers were faulty and thus caused errors. The extended time period to search through makes it so, if a certain email is not used in the flow, it will be used in the next execution of the pipeline. A JSON file tracks the used emails, thus avoiding using the same email twice. Furthermore, the script also allows the specification of filters that it must include on the title, and filters that must be excluded (also shown in Listing 5.1). This setting can be changed in the configuration, and is used to identify emails that follow a specific

naming convention, while avoiding emails in response or forwarding the same email, thus avoiding duplicate information and incorrect parsing.

Knowing the base URI used for the RSS feed, along with the settings loaded from the Jenkins configuration file, the script will start by using the Feedparser library to obtain all of the archive entries of the relevant emails, filtering through the time passed since it was sent and the filters specified in the configuration. All these entries are then combed through and checked if their title and respective information are present in the JSON file used for tracking. If so, the email will not be used in the pipeline, and is thus ignored, in order not to run twice.

After collecting the entries to the relevant emails, the script makes use of the Requests library to obtain the HTML contents of the page accessible through the entry's Universal Resource Locator (URL) field. This is required to obtain the body of the email, as Feedparser is only able to obtain the link to the email on the archive and its title. The Requests library makes use of a REST API to access the information required.

In this case, the operation used is GET, in order to obtain all of the content of the page located at the link obtained through the RSS feed. Based on the structure of the template used in this emails, the BeautifulSoup library is used to navigate through the content, easily identifying the necessary information and saving it to a dictionary of variables. Upon obtaining all of the information of a specific release, the script saves it to an array, and moves on to the next link, repeating the process. Once all of the links have been viewed and extracted from, all of the obtained information is saved to separate files, to be used by the Jenkins script. An example of such a file can be seen in Listing 5.2. It should be noted that all confidential information has been changed, as this is only for demonstration purposes, including the names of a few variables, namely the ones with "label" in the name.

```
id: <ID_of_regression>
pubDate: 25/11/22 16:55:49
hash: 3a22e2fd8955f1f424c402b43c1cd2c6
email: <email title>
link: <link_to_email_in_archive>
lane: <lanes>
proc: <project>
protocol: <protocol>
release: <release>
release_type: <type>
jira: N/A
label1: <label1>
label2: <label2>
label3: <label3>
label4: <label4>
label5: <label5>
proc_area_2: <project>
label1_version: <version>
label2 version: <version>
label3_version: <version>
description_of_change: <description>
comments: <comments>
release_date: <release_date>
owner: <owner>
```

Listing 5.2: Example of a Regression Information File

Once the Python script ends execution, the Jenkins script goes through all the information in the files created, validates the integrity of the information, and estimates

the space required for all of the regressions to run. The validation is done after the files are written and not during it, with the goal of later issuing an email to a customizable mailing list reporting which regressions were launched and which were not due to faulty information – something that would be more complicated if done through the Python script, as the information may also be valid but the regression may not be launched due to lack of space.

The estimation of space needed is done based on a setting available on the configuration file, where users can assign weights, in other words, blocks of needed space, to certain parameters of the release. For example, for a certain variable field, different weights can be assigning depending on its value. After the total is calculated, the overall value is sent to the workspace_cleanup for the cleanup to be executed. For an easier understanding of what this pipeline does, the cleanup pipeline will be explained in its own section.

The check_email_releases pipeline waits for the cleanup to finish. If no errors have occurred and there is enough disk space for the regressions, the flow will continue. Otherwise, the flow will end, and the cleanup pipeline sends an email to a customizable mailing list reporting the error, or the lack of space in the disk being used. Should there be enough room, the check_email_releases will go through each of the files produced by the Python script to load the information present in each of them and launch an instance of the next pipeline in the flow, run_preliminary_check, with the information provided, deleting the file containing the necessary information afterwards as it will not be necessary anymore. The first pipeline thus ends execution after all instances of the following pipeline have been launched, and an email is sent to a mailing list that users can subscribe to, reporting the releases used. This starts the Configuration Phase, composed of the workspace_cleanup pipeline.

5.1.2 Workspace Cleanup

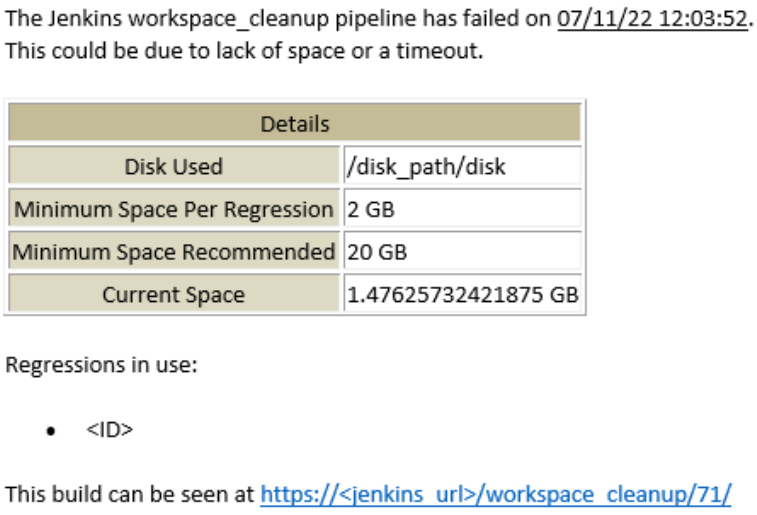
As explained earlier, this pipeline is separate from the check_email_releases pipeline for easier understanding. Much like the pipeline before it, the cleanup pipeline first needs to synchronize the necessary files through the use of the P4 Jenkins plugin. This is a mandatory step in all of the pipelines. After this, the pipeline makes use of the Jenkins configuration file to find the path where regressions are set to be executed in. This path is divided into folders for each of the releases used, following a naming convention. The name of each folder is generated using information from the respective release, so that each one has a separate area for running regressions.

The pipeline first checks the date in which each folder was last accessed, deleting the folder and the Perforce client used to synchronize the files within it in the event that it is older than a specified amount of days. This step is always performed regardless of the space estimated to run the regressions, meaning that there could be no regressions to run and this step would be performed - This is intentional, as this pipeline doubles as a periodic cleanup to keep from cluttering the disk being used. Disks are shared and used by multiple people, and due to them being common areas, it is appreciated that any unnecessary data is removed to avoid overloading the disk.

Should there be regressions to launch however, and thus a necessary amount of space, the pipeline will then evaluate if the space available is greater, or equal to the estimation. This is done with the use of a command provided by the Synopsys

file system, in which it is possible to obtain the amount of disk space remaining on a specified partition. However, this command is prone to failure, so a backup calculation that makes use of a specified max space value (in the configuration) and the Linux du command to calculate the remaining space is performed as a backup. If there is enough space, execution will end, and check_email_releases will continue its execution. Otherwise, the cleanup pipeline will start going through other old releases and cleaning them up one by one, checking if the available space is enough between cleanups. As soon as there is enough space, cleanup will end and the flow will continue. However, if all of the files are cleaned up and there is still not enough room, the flow will be terminated, and an email reporting the occurrence will be sent. Additionally, the pipeline features the sending of warning emails should the space on the disk be getting below a configurable threshold.

The email sent when there is not enough space available shows the regressions that were set to be launched. It should be noted that if the check_email_releases pipeline is terminated, the files containing the release information will not be deleted, and thus are reused in the subsequent executions of the pipeline. Figure 5.4 shows an example of the email sent in the event the cleanup is not successful, and there is not enough space to run the regressions.



The Jenkins workspace_cleanup pipeline has failed on 07/11/22 12:03:52.
This could be due to lack of space or a timeout.

Details	
Disk Used	/disk_path/disk
Minimum Space Per Regression	2 GB
Minimum Space Recommended	20 GB
Current Space	1.47625732421875 GB

Regressions in use:

- <ID>

This build can be seen at https://<jenkins_url>/workspace_cleanup/71/

FIGURE 5.4: Failed Cleanup Email

5.1.3 Run Preliminary Check

As explained earlier, for each release of a project found in an email, one instance of this pipeline will be launched, with all of the information collected beforehand, and made use of through the Jenkins Parameterized Build plugin. This plugin allows the definition and passing of variables between pipelines, which is essential for the flow developed.

Much like the other pipelines, run_preliminary_check starts off by synchronizing all of the files it requires from the Perforce file system. In this pipeline, there are two additional files required from the system (not counting the Jenkins configuration file and the Perforce view file) – the project-specific configuration file (if it exists), and the global configuration file. These files contain all of the regression parameters that

should be used when launching said regressions, and can be modified at any time through the SCM platform.

These configuration files complement each other – if a setting is not modified by the former, but is changed by the latter, the change will still occur. In the event that both configurations use a setting, the former takes priority. The structure of the regression configuration files is identical to the one presented in Listing 5.1, with the name of the setting followed by a colon and the value of the setting.

The pipeline compiles all of the settings read into a list, which is then submitted to the next pipeline, responsible for running the regression, through the use of the Jenkins Parameterized Build plugin once more, with each setting separated into its own variable, and thus the Configuration Phase ends. Furthermore, this pipeline also saves a file with a format very similar to the temporary files used in the first pipeline, with the addition of containing the loaded settings. This file will later be completed by the last pipeline with the pass rate of the regression, among other information.

Out of all the pipelines, run_preliminary_check does the least work, thus being the fastest to execute out of all of them. However, it is important to instantiate parallel executions of this pipeline in order to load the configurations, as different projects may use different configurations, and loading all the settings in a sequential manner would increase the execution time exponentially.

5.1.4 Run Tests

Finally, the last pipeline is the one with the longest execution time, as it handles the running of the regression with the later publishing of the results on an online dashboard. It receives all of the information gathered by the previous pipelines, and uses it to fetch the files required to run the regression, and also compile the command to run it with the specified settings. As a result, this is also the pipeline that synchronizes the most files from the SCM platform, as a large number of files, as well as other scripts, are necessary to run the regression. This explains why it is so important to have a configuration file for the Perforce view, as most, if not all of the files mapped using it are synchronized here. Should a modification in the regression scripts require the synchronization of additional files, a simple modification to the configuration will cover it, without editing the scripts used for the pipelines.

The pipeline starts by synchronizing all of the files required, including the ones used for the regression, into the folder created for the regression, which once more, follows a naming convention based on a few parameters from the information gathered. After this, the command used to run the regression is compiled using the labels loaded from the email and the settings from the configuration files. When everything is ready, the regression is started by calling the respective script with the command compiled from the information, and the pipeline waits for execution to end. The regression script was developed by the company, and thus is not included in the scope of this document.

It should be noted that, sometimes, regressions get stuck during execution. This may be due to several reasons, outside of the reach of the project developed. All that can be done is prepare the pipeline to handle this kind of situations. As specified by the company, regressions should run for no longer than three days, and so a timeout was

added to verify this. Should the pipeline run for longer than this amount of time, it will automatically stop the regression and the Jenkins pipeline, sending an email that reports the situation, with the regression result showing as “Stuck”.

In the event of a successful regression however, the pipeline will continue execution once the regression is finished running. By accessing a report generated by the script and parsing for some information, the pass rate of the regression is obtained, alongside other pertinent information such as the seed of the regression, which can be used to replicate the regression. This additional information is appended to the file generated by the previous pipeline, and the result is submitted to Perforce with a newly generated label, created by Jenkins. This label follows a naming convention that identifies the project used along with some of its labels, and the description contains its pass rate. Furthermore, the report generated by the regression script is forwarded to a configurable mailing list, along with information extracted from the regression log. Figure 5.5 shows a snippet of the email sent as a result of this pipeline. However, all of the confidential information is blanked out.

Result:
100.0%

Location:
/disk_path/disk/<ID>/<env_type>/cron_regres/May11_111334

Baseline Label:
jenkins_<ID>_230511_182443

Email Info:

Jenkins Info	
Publish Date	11/05/23 16:11:26
Hash	ce6c0f9d007aacc7636d0d75283a9284
Email	[confluence] <proc> - Design Labels Release for Verification - 11-May-2023 - Release <release>
Link	<link>
Jira	N/A
Description of Changes	Changes done to label1 and label2
Comments	Please proceed with verification.
Release Date	23-Aug-2023
Owner	<owner>

Release Info:

Release Info	
ID	<ID>
Protocol	<protocol>
Release	<release>
Simulation Type	<sim_type>

Labels	
Label1	<label1>
Label2	<label2>
Label3	<label3>
Label4	<label4>
Label5	<label5>
Label6	DEFAULT
Label7	N/A
Label8	N/A

Simulation Options	
Lane	<number>
Proc	<proc>
Label1 Version	<label1_version>
Label2 Version	<label2_version>
Label6 Version	DEFAULT

Settings	
Setting1	<setting1>
Setting2	<setting2>
Setting3	<setting3>
Setting4	0
Setting5	1

FIGURE 5.5: Run Tests Result Email

Finally, the report is also uploaded to Confluence, where it can be viewed by the team included in the space created for the regression dashboard. In order to do this, the Confluence REST API must be used, not to obtain information, but to provide it. For this, a Python script titled post_regression_to_confluence was created. A page is created through the POST operation containing the report of the regression, and it is easily accessible through tables that contain more generic information about the regressions that have been run.

As an additional feature, users may run this pipeline manually, skipping the other pipelines, by directly inputting the necessary information and configurations for the regression they wish to run, including the disk path where they wish to run it and the Jenkins node to use for it. This feature was made a bit more pointless however after the implementation of the Confluence Workflow described after this section, as that flow allows more control over the disks and nodes to use, provides more flexibility, and allows for scheduling of regressions. Nevertheless, it's still an useful tool in the event that the pipeline fails due to incorrect configuration on the user's end, or other system failures, as it allows the user to rebuild the pipeline and run it with the same settings, or slight manual changes. Manual runs of this pipeline feature a comprehensive user interface, a snippet of which can be seen in Figure 5.6

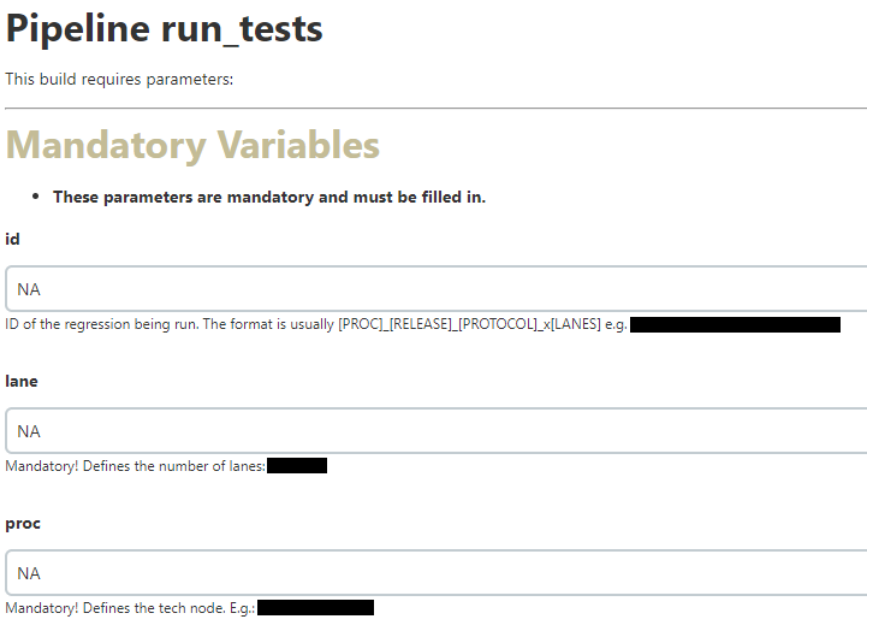


Figure 5.6: Run Tests Manual Run

5.2 Confluence Workflow Implementation

The biggest difference between the Email Workflow and the Confluence Flow is present in the Starting Phase. The reason for this is that the information is gathered from a Confluence page rather than an RSS feed of emails. Furthermore, the Confluence Workflow also allows for two methods of execution. An user may choose to run a regression immediately, which can be done through a single click after configuring the settings to use, or they may choose to schedule it for a different time.

In other words, the Email Workflow is fully automatic upon receiving emails, and performs only a simple test in order to ensure the integrity of the Perforce labels (modules used for testing), whilst the Confluence Workflow is a bit more manual and caters to the different users' needs, allowing them to run their own sets of regressions with less human interference required, and makes it possible to schedule the regressions for another time instead of running them immediately, should the user choose to do so.

Identically to Section 5.1, to better explain the connection between all the components of the project, Figure 5.7 presents a diagram that consists of the different Jenkins pipelines for this flow and their interactions, as well as the methods used to collect information.

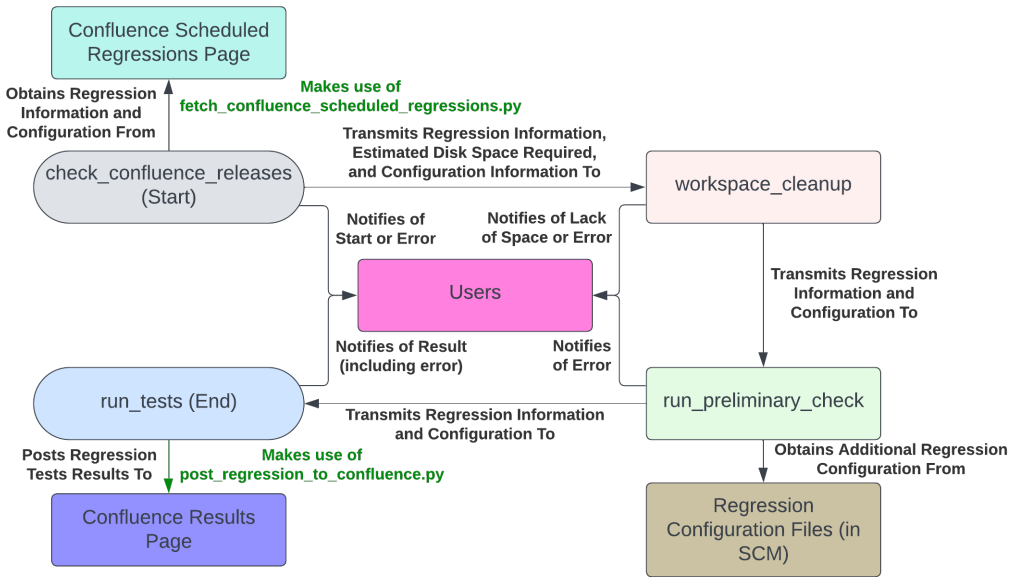


FIGURE 5.7: Information Flow Diagram for Confluence Workflow

Regarding the structure, the remainder of the flow, Starting Phase aside, stays very similar to the one presented for the Email Workflow, with only a few differences for some of the configuration, mainly the one provided through Confluence. However, a major difference between the two figures is that, in the Confluence Workflow, the regression settings (not to be confused with Jenkins settings) are already present in the first pipeline, and thus have to be transmitted across the chain of pipelines. This can be seen in the connection between the first two pipelines in Figure 5.7. The settings are then completed with additional information obtained in run_preliminary_check.

5.2.1 Confluence Tables

Before explaining how the flow starts, it is first necessary to demonstrate the tables used in Confluence to configure a regression and schedule it, if desired. To make the flow functional, two tables are used.

In one of them, the one that should be configured first, the user sets information regarding their username, the Jenkins node to use, the disk that will be used, the path within that disk where the regressions will be run, and the space that should be considered available for the regressions. All of these settings have a configurable

identity (hereby referred to as ID) that is used in the other table in order to identify which settings to use. All of the settings could make part of the same table, but this architecture was chosen over that option, as the same user will most likely use only one or two different configurations of these settings, and having to input them all over again will prove to be tedious and unnecessary.

As such, the first table, named User Configuration table, can be seen in Figure 5.8. Only one line is presented and it consists of an example of a set of information that can be used - Real information would be considered confidential.

ID	Username	Jenkins Node Name	Disk to Use	Disk Space (GB)	Regression Path
example_id	example_user	example_jenkins_node	/disk_path/disk	300	/jenkins_regressions/<ID>/<ENV_TYPE>

FIGURE 5.8: Confluence User Configuration Table

The ID and ENV_TYPE variables, seen under Regression Path, are filled in by Jenkins, depending on the ID of the regression and the environment type used. The ID of the regression is built based on four parameters of the regression - The project (sometimes referred to as proc), the release, the protocol used, and the number of lanes. This allows the user to utilize one single directory and automatically partition it for every release ID and environment used.

The second table to be configured is the one that contains the remaining information required to run the regression. As such, users have to fill the table in with a status, which consists of a basic description of what that regression is used for, date, if the user intends to schedule the regression, the ID of the configuration used in the first table, the project areas, the protocol, the release, the number of lanes used, the type of simulation, the environment used for verification, the release labels (a Perforce label on each line), and finally, a vast number of arguments separated through lines, the same ones used in the configuration files, both global and project-specific, mentioned in Section 5.1.3, that are used within the regression script, developed by other users and thus outside of the scope of this document.

This table, located in a page named Scheduled Regressions, can be seen in Figure 5.9 and Figure 5.10 (the table was split into two images in this document due to its size). Much like in Listing 5.2, all confidential information has been changed. This is only an example of how the table is configured.

Row#	Status	Date	ID of Configuration	Project Area(s)	Protocol	Release	Lane
1	Example	Wed, 01/02/2023	example_id	<proc>	<protocol>	<release>	<lanes>

FIGURE 5.9: Confluence Scheduled Regressions Table, First Half

Simulation Type	Environment Type	Release Labels	Arguments to Run With	Regression Trigger
<sim_type>	<env_type>	Label1: <label1> Label2: <label2> Label3: <label3> Label4: <label4> Label5: <label5> Label6: <label6> Label7: <label7> Label8: <label8>	setting1: <value> setting2: <value> setting3: <value>	Click here to run this regression

FIGURE 5.10: Confluence Scheduled Regressions Table, Second Half

The Regression Trigger column is generated automatically via the use of the Confluence Table Transform macro. With the use of SQL-like code, it parses each line to verify if all the necessary information is present, and if true, it generates a link, visible in Figure 5.10, that can be used to launch the confluence_regression_check pipeline, explained in the following section.

5.2.2 Confluence Regression Check

The first pipeline of this flow, called confluence_regression_check, is rather similar to the first pipeline of the Email Workflow. The regression configuration files, or in other words, the files containing all of the information needed for the regression, minus additional arguments for the regression script, are the same. However, the inputs, or the sources of the information, are different. While check_email_releases runs every four hours to collect new emails, this pipeline can run in one of two ways. It runs every day at 9:00 PM in Coordinated Universal Time (UTC), checking the date column in Figure 5.9 for any set of information whose date matches the day of the week, the full date (day, month, and year), or configured to be daily, and extracts the respective information from the table, saving it into files to later be used in the rest of the pipeline, much like check_email_releases does.

On the other hand, this pipeline can be launched by an user through the links generated using the Confluence Table Transform macro. The way this is done is that the table generates a link that can be used to trigger the job with certain variables. With the use of the job configuration present in Figure 5.11, the pipeline can then be launched with the use of an URL.

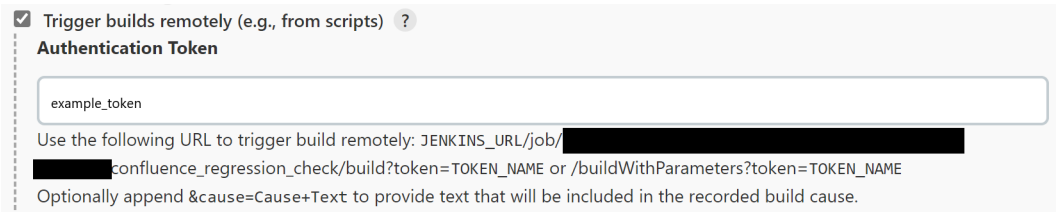


FIGURE 5.11: Jenkins Remote Trigger Setting

The URL consists of of the URL to the location of the job on Jenkins, with the token appended to the URL as a job parameter, and with additional parameters passed

through with the use of the & character, followed by the parameter and its value. This technique is used to input three parameters besides the mandatory token - The Jenkins node (so that it is used and the main Jenkins node is not necessary, and thus is left unoccupied), the line of the set of information used, and a checksum, that consists of an agglomeration of other data in the same line. This checksum is used in order to verify that the line number provided matches the checksum, as during the small amount of time between the user launching the pipeline and the table being read by the script, the table information may be edited, and the information may be lost. If the line does not match the checksum, a sweep will be performed to find a set of information that does. If no matches are found, the pipeline stops running. However, this step is only performed in the pipeline after a few other initial steps.

The pipeline starts by performing the exact same actions as check_email_releases. The SCM platform is checked for updates on any of the relevant scripts or configurations, and after the Jenkins configuration and Perforce view template are loaded, the remaining necessary files will be synced. One of these files is a Python script, named fetch_confluence_scheduled_regressions. Much like check_email_releases, this pipeline makes use of a Python script in order to obtain the information required to run the regressions. In this script, the Confluence REST API is used in order to retrieve the information from the tables presented earlier.

A company script allows users to make use of a function called make_request, which allows users to perform CRUD operations on a Confluence space or page. This function is therefore used in the fetch_confluence_scheduled_regressions Python script in order to perform a GET operation on the pages that contain the tables mentioned in Section 5.2.1. The output comes in the form of a JSON, which can then be parsed through to obtain the HTML content of the body of the page, where the tables are present. After obtaining the HTML content, it can be parsed through with the use of the aforementioned BeautifulSoup Python library.

To easily find the table, this library is used to find all tbody fields, and make use of the last one found. The reason for this is that the Scheduled Regressions page contains two tables instead of one, with the first one serving as an example for new users. The following example in Listing 5.3 shows what the general structure of the HTML used for a table in a Confluence page looks like. It should be noted that this is only an example, as the real tables are more complex and contain sensitive information, but it helps understand how the parsing of information is done. Rows are identified by searching all tr tags within the tbody using the BeautifulSoup library, with columns being identified within the tr tags via the td and th tags, the latter of which represents a header cell.

```
<h1>
  <span style="color: rgb(255,0,0);">
    <strong>Example Table - This text should not affect script</strong>
  </span>
</h1>
<table class="fixed-table wrapped">
  <tbody>
    <tr>
      <th>Status</th>
      <th>Date</th>
      <th>ID of Configuration</th>
      <th>Project Areas</th>
    </tr>
    <tr>
      <td>Example</td>
      <td>Wed, 01/02/2023</td>
      <td>user_id_1</td>
      <td>
        <p>Area1: area_id_1</p>
        <p>Area2: area_id_2</p>
      </td>
    </tr>
  </tbody>
</table>
```

Listing 5.3: Table HTML Example

After obtaining the HTML code of the page, depending on how the pipeline initiated, the following step is different. In the event that the pipeline triggered automatically due to its daily run, it will search through the entire Scheduled Regressions table, extracting the information from the rows that contain a date that matches the day it is running on, be it by matching the day of the week, the exact day, or the word "Daily". For this, the table is searched through with the use of two for structures - The first one for rows, and the second for columns. If the date matches the situation, all columns are checked and the information is extracted and added into a dictionary before moving to the next row, as intended. This dictionary is then put into an array. Listing 5.4 shows a basic version of the code that is used to go through the table and find all of the information whose date matches the day the pipeline is automatically running on.

```
def read_confluence_table(self):
    soup = BeautifulSoup(self.content, "html.parser")
    table = soup.find_all('tbody')[-1]
    rows = table.find_all('tr')
    conf_data_array = []

    day_of_week = datetime.utcnow().strftime("%a")
    day = datetime.utcnow().strftime("%d/%m/%y")
    for i in rows:
        row_data = i.find_all(['td', 'th'])
        day_data_array = row_data[1].text.split(",") # Date is second column,
    multiple dates are supported
        date_match = False
        for date in day_data_array: # If correct day, save info
            if day_of_week.strip().lower() == date.strip().lower() or day.
    strip() == date.strip() or date.strip().lower() == "daily":
                date_match = True
        if date_match:
            jenkins_data_conf = self.parse_row(row_data) # Method used to
    extract line information
            conf_data_array.append(jenkins_data_conf)
    return conf_data_array
```

Listing 5.4: Code for Table Sweep

Should the pipeline be initiated by an user, the number of the line where the regression information is located will be used as an input, given through the Jenkins pipeline script. The Python script will find all the rows through the tr tags, and check the index given. The checksum of the line will be built in the same way the one provided to Jenkins was built, and the two resulting strings will be compared. If there is no mismatch, the information of the line will be parsed and saved to a dictionary, then put into an array. Otherwise, if the strings do not match, the script will use the same methodology as the one presented for the daily run, but instead of comparing the date field, it will generate the checksum for each row until a matching string is found. If a match is encountered, the information present on that line will be used, thus covering the scenario in which some user may accidentally change the line number by adding or removing lines before it. However, if no match is found, the script will end execution without generating new files. This scenario assumes that the information was removed from the table, and without it, the script may not be run.

The following image, in Listing 5.5, shows a basic version of the code used in order to fetch the line located at the index provided, as well as the procedure taken in case the checksums obtained from the line and provided through Jenkins do not match each other. Some adaptations were made to make the code more simple to understand. The parse_row function is not shown in either of the code examples, but it makes use of the same structure to parse through the columns, extracting the information present and storing it into different variables depending on the index of the column (the table's structure does not change regarding the column order and total, meaning that the behaviour can be fixed within the script).

```
def read_confluence_table(self):
    soup = BeautifulSoup(self.content, "html.parser")
    table = soup.find_all('tbody')[-1]
    rows = table.find_all('tr')
    conf_data_array = []
    line_number_fail = False

    if self.confluence_table_line > len(rows) - 1 or self.
    confluence_table_line < 0: # Line exceeds lines on table
        line_number_fail = True
    else: # Line number is within the number of lines on the table
        row_data = rows[self.confluence_table_line].find_all(['td', 'th']) #
    Obtain array of rows
        if self.confluence_checksum is not "" and self.confluence_checksum is
    not None: # If checksum argument is passed
            checksum = ""
            counter = 0
            for column in row_data: # Use information in all columns except
    the first two to create line checksum
                if counter > 1:
                    for string in column.stripped_strings:
                        checksum = checksum + string
                counter = counter + 1
            if self.confluence_checksum != checksum: # Only runs if the
    checksum obtained and the checksum provided by Jenkins do not match
                line_number_fail = True
                for i in rows: # Look through the entire table to find
    matching checksum
                    row_data = i.find_all(['td', 'th'])
                    checksum = ""
                    counter = 0
                    for column in row_data:
                        if counter > 1:
                            for string in column.stripped_strings:
                                checksum = checksum + string
                        counter = counter + 1
                    if self.confluence_checksum == checksum: # When matching
    checksum is found, use that line
                        line_number_fail = False
                        break

        if not line_number_fail:
            jenkins_data_conf = self.parse_row(row_data) # Method used to
    extract line information
            conf_data_array.append(jenkins_data_conf)
        else:
            self.errors = self.errors + f"LINE_ERROR; {self.
    confluence_table_line}"
    return conf_data_array
```

Listing 5.5: Code to Find Given Line

In both scenarios, the dictionaries stored in the array contain all of the pertinent information obtained from the Scheduled Regressions table, one of which is the ID of the configuration present in the User Configuration. In order for Jenkins to understand what disk and Jenkins nodes it should use for each of the regressions collected, an additional file is produced, containing all the necessary information for that purpose. The array of dictionaries is combed through, and all the different user IDs are gathered. Each of these IDs are then located on the User Configuration table, and their respective information is stored in this file. This means that, when Jenkins checks each of the regression configuration files generated, it then checks this additional file to obtain the remainder of the information. An example of the content of this file is present below, in Listing 5.6.

```
example_id; example_user; example_jenkins_node; /example/disk; 300; /
    jenkins_regressions/<ID>/<ENV_TYPE>
example_id_2; example_user; example_jenkins_node_2; /example/disk; 100; /
    jenkins_regressions_2/<ID>/<ENV_TYPE>
```

Listing 5.6: User Configuration File

The reason why the information is printed to a separate file and not included in the regression configuration file is to keep its format consistent with the files produced by the Email Workflow. The methodology used to extract this information is very similar to the one present in Listing 5.4, replacing the date variable with the ID of the configuration. The information present in the row found is merged into a string, separated by a semicolon, and put into an array. This array is then written into the file that is printed from this.

The rest of the flow remains mainly the same, with a few exceptions, most notably an increased number of arguments that gets passed from pipeline to pipeline. In the Email Workflow, the disk path where the regressions are run is always within the same folder, whilst in this flow, the path changes from user configuration to user configuration, so the path to be used must be kept in mind. Additionally, the Jenkins node used will be one that the user owns, instead of a service account node, like the one used for the Email Workflow, and the space limit of the disk used must also be taken into account. Finally, since the Scheduled Regressions table allows the full configuration of the regressions, including settings that would be obtained from the configuration files in run_preliminary_check, the list of settings must also be passed. With all these differences, the following list summarizes the modifications done to the other pipelines in order to support the Confluence Workflow.

- **All Pipelines**
  - The disk, path, Jenkins node, and disk space varies from user configuration to user configuration.
  - Additional variables are passed - Disk, path, Jenkins node, disk space, string of regression settings.
- **workspace_cleanup**
  - The threshold for how old regressions need to be to be removed is different, and no other regressions besides old ones are removed, as the user may need them. The Email Workflow is intended to be a simple test to ensure the integrity of the release, and thus removing older files in the Email Workflow will have no impact on any user. The checking for space will still happen, and the pipeline will also stop if not enough space is found.
  - The space specified by the user in the User Configuration table will be used if an error occurs while trying to determine the amount of space required, as a backup measure.
- **run_preliminary_check**
  - Receives string of regression settings, splits it into the different necessary variables.

  - The regression settings provided through Confluence override those that are present in the tech-specific and global regression configuration files.

- **run_tests** - Makes use of the disk and path provided to run the regression in the location specified.

5.3 Online Dashboard

As mentioned before, both of the flows upload the results of the regressions they run into an online dashboard, located under a separate Confluence page, titled Regression Results. This step is performed at the end of the run_tests pipeline, and the page created to hold the results contains the same information as the report automatically generated by the regression script created by the company. The Regression Results page is considered the overview page, and under it, several other sub pages are created depending on regression parameters, thus establishing a hierarchy that separates the results from project to project and more. The hierarchy utilized is as follows:

- Regression Results
  - <project> - Overview of regressions for the project.
    - <protocol>_<release>_<lanes> - Overview for regression ID.
      - <env_type>_<sim_type>_<datetime> - Page of result report.

If two regressions have the exact same ID, they will be located under the same page at the third level of the hierarchy (under <protocol>_<release>_<lanes>). If only their project is in common, then they will share the second level of the hierarchy (<project>). If none are in common, then they will be separated into different project pages. The datetime argument in the last level of the hierarchy ensures that a new page is created for each regression. A Python script handles the creation of the page, once again making use of the Confluence REST API. The creation of a page is done with the use of the POST operation, providing the HTML body to be used for the page. The script also maintains a configurable limit of pages under each regression ID (level three), meaning that if a new page is created and the regression ID goes over the limit of pages, the oldest page is deleted. The oldest page is identified via comparisons of the datetime argument present in the name of each page. Figure 5.12 shows a snippet of what a result page for a given regression looks like when uploaded to the dashboard.

<env_type>_<sim_type>_23-05-15_12:19:28

Created by [redacted]

Simulation type:

- <sim_type>

Location:

- /disk_path/disk/<ID>/<env_type>/cron_regres/May12_022336/

Regression launched in:

- 12 May 2023

Regression time:

- 84.0 hours / 5039.985 minutes

Description:

- <sim_type> regression report for <proc>

Options

Perforce labels	Simulation options		Tools version	
Label1: label1	Test list	basic_test	Tool1	2020.6
Label2: label2				
Label3: label3	Proc	<proc>	Tool2	2021.8
Label4: label4				
Label5: label5	Lanes	<number>		
Label6: DEFAULT				
Label7: label7				

Metrics

Daily testbench Metrics			RTL Metrics	
Regression pass rate	99.4082840237 %		Line coverage	93.60 %
Daily functional coverage	97.78 %		Conditional coverage	95.10 %

FIGURE 5.12: Regression Result Page

Additionally, each of the pages in the hierarchy except the last contain a table that presents a brief overview of each regression that has been run and fits under that page. This table is also maintained by the script, not surpassing a configured limit of lines. This setting, as well as the number of pages under each regression ID, are configurable in the Jenkins Configuration file. Figure 5.13 shows an example of what one of the tables present in the higher hierarchy levels look like. Not all of the information present on the table is shown here. Again, confidential information has been replaced with placeholders.

Run Type	Run Date	Pass Rate	Proc	Release	Protocol	Lanes	Simulation Type	Environment Type	Release Labels
Confluence	06/05/23 11:36:38	99.2452830189%	<proc>	<release>	<protocol>	<number>	<sim_type>	<env_type>	Label1: label1 Label2: label2 Label3: label3 Label4: label4 Label5: label5 Label6: DEFAULT Label7: label7

FIGURE 5.13: Overall Regression Results Table

The example shown displays the top level table, meaning that all regressions run using this system are listed here, up to a set maximum. Whilst the project (proc) and release values are hidden due to confidentiality, it should be noted that both of these values, along with the pass rate, include hyperlinks that allow the user to more easily navigate the space. The project hyperlink takes the user to the second hierarchy level in which the regression result is present, the one in the release column

takes the user to the third hierarchy level, and the pass rate redirects the user to the corresponding regression report directly. When a new regression is run, the respective tables are updated through the REST API, with the use of the PUT operation (update), after editing the old HTML contents of the page in order to include the new information.

5.4 Testing Performed

In order to test the implementation to verify if all the requirements are fulfilled, many manual tests were performed. The source of information used to test the system varies, being provided by different team members. By using information provided by other users, the testing is not biased and thus not influenced by the creator. All aspects and features of the system were tested, to ensure that it functions correctly in of itself. The result of this testing is presented in Chapter 6.

5.5 Summary

In summary, the implementation of the project has been done according to the specifications given by the company, as well as the new requirements that were added during development. In order to implement the system, Python scripts were used in order to make use of REST APIs to obtain or upload information. Jenkins was used in order to automate all of the process, and when faced with the task of implementing a separate flow that makes use of information provided through Confluence, the situation was thoroughly analyzed in order to allow for the most seamless implementation possible, with most of the already existing flow implemented for the Email Workflow being re-utilized, with minor modifications to the pipelines to allow for additional features.

In the end, a total of five Jenkins pipelines were developed. For REST API operations, three Python scripts were created. Finally, three main Confluence pages are used, two of them being used for running regression tests, and the other for a dashboard of results, containing a sub-page hierarchy for organized reports.

Lastly, should an adjustment be required in the future, to avoid modifying the scripts developed, two major configuration files are used to control most of the features without direct editing of the scripts - The Jenkins configuration file, and the Perforce view template file.

With all of this taken into consideration, the following diagram in Figure 5.14 is presented, as an advanced version of Figure 4.3. All the connections between each of the components is shown, and detailed by the legend.

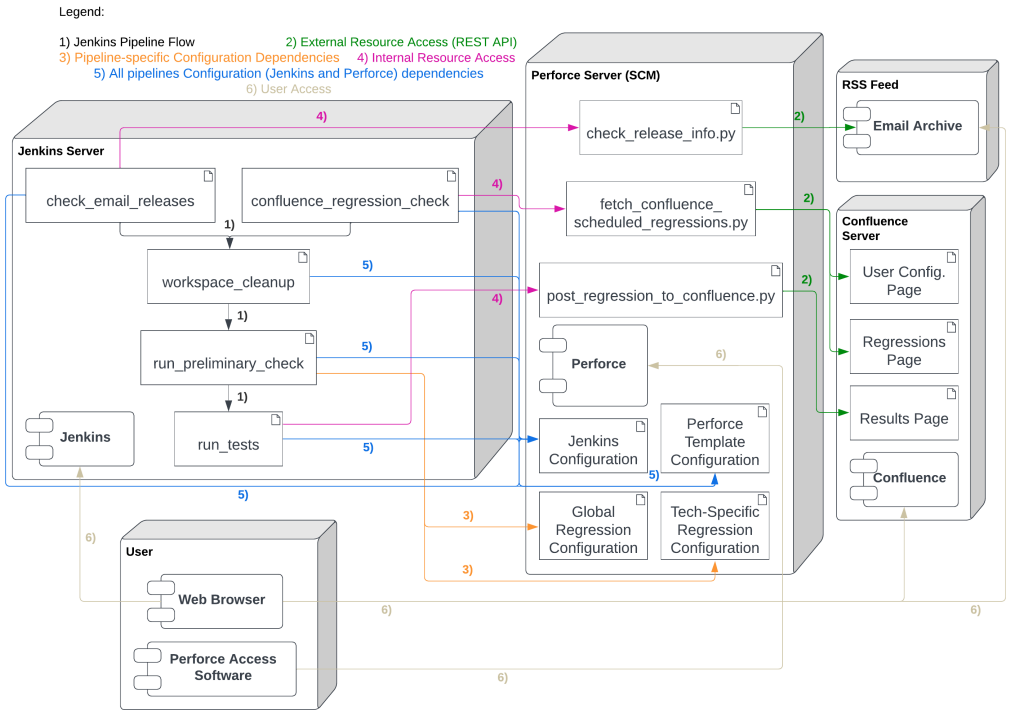


FIGURE 5.14: Deployment Overview (Implementation)

Seen in Figure 5.14, the three types of connection presented in Figure 4.3 are also shown here, along with three new types. The list is as follows:

1. **Black** - Flow between Jenkins pipelines.
2. **Green** - External connections (uses REST API).
3. **Orange** - Pipeline-specific dependencies to configurations.
4. **Magenta** - Internal connections (direct call).
5. **Blue** - Jenkins and Perforce template configurations, which all pipelines depend on.
6. **Beige** - User access to different platforms.

Chapter 6

Evaluation of the Solution

In this chapter, the solution produced will be tested and evaluated to measure how it resolves the presented problem, and completes the goals that were set. Through this evaluation, done through the QEF model, presented in section 2.8, and a comparison between the old flow utilized to run regressions and the new implemented system, it will also be possible to answer the research questions (RQs) that were introduced in section 1.3.

6.1 Quantitative Evaluation Framework Applied to the Project

Taking into consideration everything that has been mentioned in section 2.8, for the evaluation of the project, three criteria were chosen, which will be used as the three dimensions:

- **Functionality** - The features that the solution must possess as well as system characteristics (Technical Domain).
- **Adaptability** - The system's capability of adapting to different users' situations and needs (Technical Domain).
- **Usability** - The interactions between the user and the system (Ergonomic Domain).

As seen under section 3.4, all of the requirements for the application were classified with one of the dimensions presented above, and then separated into factors that present their role in the project. Afterwards, all of the different requisites were given a relative importance, and from these values, the relative importance of each of the factors used within their respective dimension was calculated. The values for the factor and requirement weights, or relative importance, are present in tables 6.1, 6.2, and 6.3.

TABLE 6.1: Functionality Factors, Weight, Requirements, and Requirement Weights

Wij (Factor Weight j in Dim	**Factor**	rwjk (requirement weight k in Factor j) {2, 4, 6, 8, 10}	**Requirement**
0.16	Information Collection and Validation	10.00	FIC01 - System must collect email data automatically
		8.00	FIC02 - System must create file with email data
		8.00	FIC03 - System must validate data provided in email
0.11	Space Management	10.00	FSM01 - System must confirm that the available disk space is enough
		8.00	FSM02 - System must clean up older files to free disk space
0.18	Test Setup and Running	10.00	FTS01 - System must compile given configurations to run regression
		10.00	FTS02 - System must set up test area without human input
		10.00	FTS03 - System must run tests in area created
0.27	Tracking and Result Publishing	8.00	FTR01 - System must track regression ID run
		10.00	FTR02 - System must tag files used to allow users to replicate the area in future (tracking and debug)
		6.00	FTR03 - System must save file with regression details to SCM platform
		10.00	FTR04 - System must send email stating test status
		10.00	FTR05 - System must upload results and information to online dashboard
0.12	Confluence User Functionalities	10.00	FCU01 - System must allow users to schedule regressions with Confluence table
		10.00	FCU02 - System must allow users to run regressions immediately using Confluence table
0.16	Confluence System Requirements	8.00	FCS01 - System must verify that all necessary information is introduced
		8.00	FCS02 - System must reuse the system developed for the automatic email flow
		10.00	FCS03 - System must run pipeline on a node owned by the user (nodes are used to run jobs)

TABLE 6.2: Adaptability Factors, Weight, Requirements, and Requirement Weights

Wij (Factor Weight j in Dim	**Factor**	rwjk (requirement weight k in Factor j) {2, 4, 6, 8, 10}	**Requirement**
0.93	Configuration	10.00	AC01 - System must allow configuration of path to run tests in
		6.00	AC02 - System must allow configuration of timeouts for pipelines (avoid stuck tests)
		8.00	AC03 - System must allow configuration of email formats and content
		8.00	AC04 - System must allow configuration of file view to synchronize
		8.00	AC05 - System must allow configuration of client names to use for file synchronization
		10.00	AC06 - System must allow global configurations for regression tests
		10.00	AC07 - System must allow specific configurations for regression tests based on project
		8.00	AC08 - System must give priority to specific configurations, completing with global
		10.00	AC09 - System must allow users to configure the disk, path, and node to use
		10.00	AC10 - System must allow users to configure the schedule for the regression
		8.00	AC11 - System must allow configuration of more than one disk to use and support different grid locations
		10.00	AC12 - System must allow configuration of all parameters required for regression
		10.00	AC13 - System must allow configuration of simulation type and verification environment
		8.00	AC14 - Configurations provided in table should override project specific and global configurations
0.07	Expandability	10.00	AE01 - System must be expandable and modular in order to allow future integration of other environments / projects

Table 6.3: Usability Factors, Weight, Requirements, and Requirement Weights

Wij (Factor Weight j in Dim	Factor	rwjk (requirement weight k in Factor j) {2, 4, 6, 8, 10}	Requirement
0.48	Inform User	10.00	UI01 - System must inform admins if any part of the pipeline fails
		10.00	UI02 - System must inform users of emails detected
		8.00	UI03 - System must inform users of disk space used and remaining
		10.00	UI04 - System must inform if space is running too low
		10.00	UI05 - System must inform users of missing data in email
		10.00	UI06 - System must inform users when their regression is processed/started
0.30	Prevent Problems and Misuse	10.00	UP01 - System must not read the same email twice
		8.00	UP02 - System must not attempt to run tests with missing information
		10.00	UP03 - System must stop if space is not enough
		8.00	UP04 - System must cancel tests based on timeout (avoid stuck tests)
0.22	Setup Process	8.00	US01 - Setting up a regression on the table must be simple and intuitive
		8.00	US02 - A template for setting up a regression must be provided
		10.00	US03 - A set of instructions on how to set up the system must be provided

After the development of the project, the QEF model was revisited, and no major changes were made to the requirements that were already present. As such, the tables 6.1, 6.2, and 6.3 are final, regarding the criteria and requirements. After the rating of the completion of each requirement, some areas that require improvement were detected, but no rating lower than 75% was given. It should also be noted that ratings were done in increments of 25%, with a minimum of 0% and maximum of 100%. To avoid repetition of topics already presented in Chapter 5, only the areas that require improvement and areas that were not expressed in full detail already will be discussed. With that in mind, tables 6.4, 6.5, and 6.6 show the ratings given for each requirement in the QEF.

Table 6.4: Functionality Requirement Ratings

Factor	rwjk (requirement weight k in Factor j) {2, 4, 6, 8, 10}	Requirement	wfk % requirement fulfillment k) [0,100]
Information Collection and Validation	10.00	FIC01 - System must collect email data automatically	100
	8.00	FIC02 - System must create file with email data	100
	8.00	FIC03 - System must validate data provided in email	75
Space Management	10.00	FSM01 - System must confirm that the available disk space is enough	75
	8.00	FSM02 - System must clean up older files to free disk space	100
Test Setup and Running	10.00	FTS01 - System must compile given configurations to run regression	100
	10.00	FTS02 - System must set up test area without human input	100
	10.00	FTS03 - System must run tests in area created	100
Tracking and Result Publishing	8.00	FTR01 - System must track regression ID run	75
	10.00	FTR02 - System must tag files used to allow users to replicate the area in future (tracking and debug)	100
	6.00	FTR03 - System must save file with regression details to SCM platform	75
	10.00	FTR04 - System must send email stating test status	100
	10.00	FTR05 - System must upload results and information to online dashboard	100
Confluence User Functionalities	10.00	FCU01 - System must allow users to schedule regressions with Confluence table	100
	10.00	FCU02 - System must allow users to run regressions immediately using Confluence table	100
Confluence System Requirements	8.00	FCS01 - System must verify that all necessary information is introduced	100
	8.00	FCS02 - System must reuse the system developed for the automatic email flow	100
	10.00	FCS03 - System must run pipeline on a node owned by the user (nodes are used to run jobs)	100

Regarding the Functionality dimension, seen in Table 6.4, only a few problems arise.

For requirement FIC03, the issue is that the data validation is not 100% accurate. While the pipeline does check for missing information, it is unable to recognize if a given Perforce label is correct, and thus, it will continue operation, eventually failing during the testing phase due to being unable to synchronize the necessary files due to the wrong label.

For requirement FSM01, the estimation of remaining disk space is not as robust and accurate as it should be. This is mainly due to a problem out of the user's control – a tool developed by the company is used to verify the remaining space with the use of a single command, but the tool is prone to failure. Because of this, a backup system was developed, which makes use of the disk space variable to estimate the remaining space. However, the calculation is not entirely accurate, and while it does work for a great majority of the time, some issues have been verified in the past.

Lastly, both FTR01 and FTR03 share a common problem. In the rare event that the Jenkins pipeline ends unexpectedly due to an error (either on the Jenkins system, or other systems it relies on), an error with the interactions with the SCM platform may occur. The way the files used for tracking in both requirements are handled is that, if the file already exists on the SCM platform, it will be opened for editing, edited, and then submitted, closing the file and making it unavailable for any further editing. In the event that the pipeline ends execution after editing, but before submitting, the file will be left open, which will cause errors in the feature if new changes are made to the file on the SCM platform by another client, thus warranting these requirements a rating of 75%.

Regarding topics not thoroughly explored previously, FTR02 consists of the tagging of the files used in the regression. After run_tests finishes running, a Perforce label is created automatically and applied onto all the files involved in the regression, so that users may refer to it when performing further verification. Tagging labels does not involve the same process mentioned for requirements FTR01 and FTR03, and thus is more resilient in the event of a pipeline failure.

Table 6.5: Adaptability Requirement Ratings

Factor	rwjk (requirement weight k in Factor j) {2, 4, 6, 8, 10}	**Requirement**	wfk % requirement fulfillment k) [0,100]
Configuration	10.00	UC01 - System must allow configuration of path to run tests in	100
	6.00	UC02 - System must allow configuration of timeouts for pipelines (avoid stuck tests)	100
	8.00	UC03 - System must allow configuration of email formats and content	100
	8.00	UC04 - System must allow configuration of file view to synchronize	100
	8.00	UC05 - System must allow configuration of client names to use for file synchronization	100
	10.00	UC06 - System must allow global configurations for regression tests	100
	10.00	UC07 - System must allow specific configurations for regression tests based on project	100
	8.00	UC08 - System must give priority to specific configurations, completing with global	100
	10.00	UC09 - Confluence system must allow users to configure the disk, path, and node to use	100
	10.00	UC10 - Confluence system must allow users to configure the schedule for the regression	100
	8.00	UC11 - Confluence system must allow configuration of more than one disk to use and support different grid locations	75
	10.00	UC12 - Confluence system must allow configuration of all parameters required for regression	100
	10.00	UC13 - Confluence system must allow configuration of simulation type and verification environment	100
	8.00	UC14 - Configurations provided in Confluence table should override project specific and global configurations	100
Expandability	10.00	UE01 - System must be expandable and modular in order to allow future integration of other environments / projects	100

Regarding the Adaptability dimension, visible in Table 6.5, the only problem present

is requirement UC11. Whilst different grid locations are supported and tested intensively, the feature of using multiple disks on one configuration is not thoroughly tested, and thus is prone to error.

All other requirements were discussed in detail throughout Chapter 5, and a full list of the available configurations is included in Appendix B. Regarding UE01, the system has proven to be modular and expandable, as seen by the minor adjustments that were necessary to implement the Confluence Workflow (Email Workflow was fully structured beforehand, and adapted to support the new flow), and the separation of the flow throughout different pipelines makes it more easily modifiable.

Table 6.6: Usability Requirement Ratings

Factor	rwjk (requirement weight k in Factor j) {2, 4, 6, 8, 10}	Requirement	wfk % requirement fulfillment k) [0,100]
Inform User	10.00	UI01 - System must inform admins if any part of the pipeline fails	75
	10.00	UI02 - System must inform users of emails detected	100
	8.00	UI03 - System must inform users of disk space used and remaining	100
	10.00	UI04 - System must inform if space is running too low	100
	10.00	UI05 - System must inform users of missing data in email	100
	10.00	UI06 - System must inform users when their regression is processed/started	100
Prevent Problems and Misuse	10.00	UP01 - System must not read the same email twice	100
	8.00	UP02 - System must not attempt to run tests with missing information	75
	10.00	UP03 - System must stop if space is not enough	100
	8.00	UP04 - System must cancel tests based on timeout (avoid stuck tests)	100
Setup Process	8.00	US01 - Setting up a regression on the table must be simple and intuitive	100
	8.00	US02 - A template for setting up a regression must be provided	100
	10.00	US03 - A set of instructions on how to set up the system must be provided	100

Finally, regarding the dimension of Usability, seen in Table 6.6, two problems arise. Requirement UP02 is very similar to FIC03. Whilst variables are checked to verify if they are not present, the contents of the variables are not checked as thoroughly. In the event an email is sent with wrong information or a wrong format, or an user configures a regression with invalid information, if the information makes it through the check whilst being an invalid label, it will not be detected, as there is no fits all label format used by everyone — the format may change depending on the user, so it is hard to verify.

As for the other problem, requirement UI01 states that the system must inform admins if any part of the pipeline fails. While this is mostly respected, a rare occurrence where the pipeline fails during the initial checkout of the SCM platform, done at the start of every pipeline, will not be detected by the script, and thus an email will not be sent. Whilst this is a very rare and unlikely error, the pipeline should still be more robust.

Overall, the project obtained a grade of 98% according to the QEF. Whilst it is not perfect and should be optimized in a few areas, it fulfils the requirements presented to a very satisfying degree.

6.2 Comparison to Previous Workflow

With all of this taken into consideration, Table 6.7 presents the differences, positive and negative, between the new flows and the old flow, as well as the new features

that resulted from this project.

Table 6.7: Comparison of New and Old Workflows

New Workflows	Old Workflow
Automatic setup of regression area	Manual setup of regression area
Automatic cleanup	Manual cleanup
Scheduling of regressions (Confluence Workflow)	Command runs regression immediately
Results shared through email and dashboard	Results shared through email
Simple and friendly user interface	No user interface
Automatic tracking of regressions	Tracking of regressions is done manually
History of past regressions	User must search through emails
Depends on more tools - less robust	Depends on less tools - more robust
Initial manual setup required	Only initial setup is the area
Entirely New Features Thanks to New Workflows	
Automatic simple testing of new releases (Email Workflow)	
Scheduling of regression tests (Confluence Workflow)	
Automatic cleanup	
Automatic tracking of regressions	
Online dashboard with regression results	
User interface through Confluence / Jenkins	

6.3 Summary

To sum it up, this chapter put the solution developed to the test through the use of the QEF model, with the requirements being divided into different dimensions in order to calculate an accurate rating of the project's performance. With the rating obtained, it is possible to respond to the research questions that were presented back in Chapter 1, under section 1.3.

In the end, the system implemented brings many benefits to the table when used, and if some more optimizations are performed to cover its weaker areas (such as a few cracks in the robustness of the system), it could end up replacing the manual method in most scenarios. The rating of 98% on the QEF also proves that the requirements were fulfilled with minor issues on a few of them.

Chapter 7

Conclusion

This final chapter summarizes the contents of this document, using the information presented to answer to the Research Questions proposed in Chapter 1, and presenting future plans and work for the project developed. Additionally, the limitations present during the development of the project are also explained.

As mentioned previously, the flow of running a regression up until the use of the regression script provided by the company, as well as the later publishing and sharing of the results, is fully manual. The goal of the project was to assist and make this process more agile by automating the running of simple tests on new releases of labels, whilst also facilitating manual runs of new regressions, especially if they are to be scheduled. In terms of the regression execution time, not much has changed as the farm used to run the regression is the same, meaning that the regression times between Jenkins and the old flow fluctuate around the same values. What is more noticeable however, is the time saved for performing other operations automatically. With Jenkins, the user does not need to:

- Set up a regression area;
- Clean up older regressions;
- Wait until a certain time to run the regression (Confluence Workflow, can schedule instead);
- Share the results with the team when the regression finishes running.

Additionally, the user is able to make use of better interfaces when running regressions and evaluating the results of said regressions:

- Better user interface when running regressions (manual run of run_tests or Confluence table is better than command line);
- Results are made available on online dashboard automatically for ease of access;
- User is also notified when regression starts / ends.

And to top it all off, the system also provides tracking of the regressions run:

- Information about latest regression for a specific regression ID is saved to Perforce;
- Online dashboard presents a hierarchy of pages that the user can navigate and easily view the latest regressions for a release / ID;

- Automatic attribution of Perforce label to files used means the user can sync the files used and verify any problems that may have appeared, or it can also be used for debugging purposes.

However, this system also comes with a few downsides:

- The user needs to follow some setup instructions before being able to use the system;
- The system needs to be more robust when faced with errors with dependencies (SCM failure, plugin error) - Send email to admins explaining the error, stop execution without causing further problems;
- Even with utmost robustness, the system is still prone to failure if one of the systems it relies on fails, and in that scenario, the manual method will be required.

7.1 Research Questions

Recalling back to the research questions defined in Section 1.3, after going over all of the information present in the previous chapters, it is now possible to give them a definitive answer.

RQ1 - How useful would it be to have a system capable of automatically running configurable sets of regression tests on labels as they're released?

This question is answered to using the Email Workflow. There is a major upside to this system - As new releases are published, the team will not have to run a regression using the labels provided in the release to ensure the integrity of the release, as Jenkins will perform this task entirely automatically and notify the team of the result, giving users more time to spend on more important tasks. The downside is that this pipeline executes on its own on a fixed schedule, every four hours by default, meaning that there may be a sizeable delay between a release and its testing. To correct this downside, a new workflow is planned and explained in Section- 7.3.

RQ2 - How useful would it be for tracking to automatically obtain the results from the regressions run using this system, and make them available on an online dashboard?

Information is key, and communication is extremely important. In an environment with multiple teams working on the same projects, it is crucial that information gets across as clearly and efficiently as possible. Not only is an email sent to a configurable mailing list, notifying those who subscribe to it regarding new regression results, those same results are also made available in an online dashboard so that other teams, or even members within the same team, may see the latest results. Both the Email Workflow and the Confluence Workflow adhere to this.

RQ3 - How useful would it be to have a system that can be used to schedule certain regressions, and thus automatically run them and obtain their results?

Whilst a new set of labels, or release, is being worked on, it is very likely that multiple regressions will have to be performed to test every aspect of said release. Not only does the Confluence Workflow facilitate the configuration of multiple regressions and allows to run them simultaneously, it allows the team members to set a schedule for

when said regressions should run. This is extremely useful in a scenario where an user may want to run a regression weekly to observe any changes in the results of the release, be those changes positive or negative. Furthermore, the online dashboard facilitates once more the visibility of the evolution of the results obtained.

In the end, all of the questions are answered with positive feedback, as this system provides the verification team with immense improvements over the old flow, whilst keeping it all simple to use. The Email Workflow removes the need for the team to verify if the information provided in new email releases is valid, as the system performs this verification automatically and informs the team about it. On the other hand, the Confluence Workflow makes it easier for users to set up new regressions to run, allows them to schedule them, be it a repeating instance or a one time run, and makes all of the results available on an online dashboard, providing easier communication with other teams as well by automatically sharing the results of the regression this way.

The tracking history of the regressions run also provides immense utility, showing users the evolution in the results of a certain release, and tracking all of the results obtained with customized labels, making the files used in the respective regressions easily accessible for verification. The files saved to the SCM system containing the regression information also bring new possibilities of new flows, as explained in Section 7.3.

7.2 Limitations

Seeing as this project was developed under the guidance of Synopsys, adjusting to their requirements and specific needs, this type of solution would most likely require some significant changes in order to adapt to other situations. However, the solution was developed in a manner that would suit other teams at the same company and not only the one it was developed for, making it so other teams can take benefit from the system developed.

7.3 Future Work

For future work, making the system as robust as possible is the number one priority. The system should be capable of handling any problem it may encounter, and if it is not possible to recover from the problem, the system should notify the administrators about the occurrence, and terminate without causing any additional problems. While these situations are fairly rate, the system should in no case ever cause additional errors, even if an external cause was involved.

After that, a new flow is planned to be implemented. The idea is to monitor the SCM platform, Perforce, for any changes under a specific path in the depot. Then, based on the changes performed and the projects affected, the system should check the SCM platform for the latest regressions performed for said projects with the highest pass rates, and run a simple regression test using the information obtained to ensure the integrity of the project after the change. All of this information can be found in the regression information files saved to Perforce by this very own system at the end of the run_tests pipeline, and by using the Perforce label created by it,

it is possible to sync all the same files used in that regression, whilst keeping the ones introduced or modified by the change that occurred.

Further into the future, this project will be maintained, as it is expected that more and more people make use of this tool, so it will be crucial that any issues that arise are taken care of immediately. As such, the project will likely undergo further modifications and improvements, possibly expanding to support more protocols, projects, and even verification environments, than the ones supported currently, re-using the same release framework.

7.4 Final Remarks

The project described in this paper has been in use for production since the 17th of December of 2022, used by a few team members, with continuous improvements being developed and introduced throughout the following months. Overall, the project developed was successful in fulfilling the company's requirements. This is proven by the benefits brought by it when compared to the old flow of running regressions, described in Section 6.2, as well as the score obtained from the QEF, 98%, as explained in section 6.1. The two flows created, the Email Workflow and the Confluence Workflow, possess different purposes, both of them aiding the verification teams in their daily lives by automating a sizeable portion of the process.

All the features introduced by the system are of great use for the team within the company, and are even being noticed by other teams that wish to replicate the system for their own purposes. This fact alone shows that the development of this project was of major benefit to the company as a whole, and it is planned that the entire team makes some use of the system in the future.

Bibliography

[1] Synopsys. *Synopsys | EDA Tools, Semiconductor IP and Application Security Solutions*. URL: https://www.synopsys.com (visited on 12/20/2022).

[2] Arun J. Kurian. "An ASIC design and test methodology for an undergraduate design and fabrication project". MA thesis. University of Texas, El Paso, Dec. 2012. URL: https://scholarworks.utep.edu/dissertations/AAI1533234.

[3] Mehvish Rashid. "Evaluating the Effectiveness of Regression Testing". MA thesis. Chalmers University of Technology, Goteborg, Dec. 2011. URL: https://odr.chalmers.se/server/api/core/bitstreams/500258c1-91ff-42f3-a360-c701a0647956/content.

[4] John Markoff. "Flaw Undermines Accuracy of Pentium Chips". In: *Business Day* Section D (Nov. 1994), p. 1.

[5] Gaurav Duggal and Bharti Suri. "Understanding Regression Testing Techniques". In: (Jan. 2008).

[6] Amir Ngah, Malcolm Munro, and Mohammad Abdallah. "An Overview of Regression Testing". In: *Journal of Telecommunication, Electronic and Computer Engineering* 9 (Oct. 2017), pp. 45–49.

[7] Heleno de Souza Campos Junior et al. "Test case prioritization: a systematic review and mapping of the literature". In: Sept. 2017, pp. 34–43. DOI: 10.1145/3131151.3131170.

[8] Synopsys. *What Is CI/CD and How Does It Work?* URL: https://www.synopsys.com/glossary/what-is-cicd.html (visited on 01/02/2023).

[9] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* PP (Mar. 2017). DOI: 10.1109/ACCESS.2017.2685629.

[10] Leonardo Ferreira Leite et al. "A Survey of DevOps Concepts and Challenges". In: *ACM Computing Surveys* 52 (Nov. 2019), pp. 1–35. DOI: 10.1145/3359981.

[11] Ritheesh Baradwaj Yellenki. *DevOps RoadMap — How to Get Started!!* URL: https://pub.towardsai.net/devops-roadmap-how-to-get-started-39ce861e15c0 (visited on 12/30/2022).

[12] John Ferguson Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. O'Reilly Media, Inc., 2011.

[13] Atlassian. *Bamboo Continuous Integration and Development Build Server*. URL: https://www.atlassian.com/software/bamboo (visited on 12/30/2022).

[14] Prithwiraj Choudhury et al. "GitLab: work where you want, when you want". In: *Journal of Organization Design* 9 (Nov. 2020), p. 23. DOI: 10.1186/s41469-020-00087-8.

[15] GitLab. *GitLab CI/CD*. URL: https://docs.gitlab.com/ee/ci/ (visited on 12/30/2022).

[16] Dr Majumdar et al. "Source code management using version control system". In: Sept. 2017, pp. 278–281. DOI: 10.1109/ICRITO.2017.8342438.

[17] Scott Chacon and Ben Straub. *Pro Git (Second Edition)*. Springer Nature, 2014. DOI: 10.1007/978-1-4842-0076-6.

[18] Atlassian. *Bitbucket Overview*. URL: https://bitbucket.org/product/guides/getting-started/overview (visited on 12/31/2022).

[19] Matthew B. Doar. *Practical JIRA Administration: Using JIRA Effectively: Beyond the Documentation*. O'Reilly Media, Inc., 2011.

[20] Stefan Kohler. *Atlassian Confluence 5 Essentials*. Packt Publishing, 2013.

[21] Laura Wingerd. *Practical Perforce*. O'Reilly Media, Inc., 2005.

[22] Johan Sjöblom and Caroline Strandberg. "Automatic Regression Testing using Visual GUI Tools". MA thesis. Chalmers University of Technology, Goteborg, Oct. 2014. URL: https://publications.lib.chalmers.se/records/fulltext/215187/215187.pdf.

[23] Raimund Hocke. *RaiMan's SikuliX*. URL: http://sikulix.com/ (visited on 01/02/2023).

[24] Filip Olsson and Philip Ridderheim. "Automated Cooperative API Regression Testing Using Jenkins". MA thesis. Lund University, May 2019. URL: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9031071&fileOId=9031073.

[25] Peter A. Koen et al. "Fuzzy Front End: Effective Methods, Tools, and Techniques". In: *The PDMA ToolBook for New Product Development*. John Wiley & Sons, Inc., 2002, pp. 5–35.

[26] R.W. Saaty. "The analytic hierarchy process—what it is and how it is used". In: *Mathematical Modelling* 9.3 (1987), pp. 161–176. ISSN: 0270-0255. DOI: https://doi.org/10.1016/0270-0255(87)90473-8. URL: https://www.sciencedirect.com/science/article/pii/0270025587904738.

[27] Almoatazbillah Hassan. "The Value Proposition Concept in Marketing: How Customers Perceive the Value Delivered by Firms – A Study of Customer Perspectives on Supermarkets in Southampton in the United Kingdom". In: *International Journal of Marketing Studies* 4 (May 2012). DOI: 10.5539/ijms.v4n3p68.

[28] Alexander Osterwalder et al. *Value Proposition Design: How to Create Products and Services Customers Want*. John Wiley & Sons, Inc., 2014.

[29] Paula Escudeiro and José Bidarra. "Quantitative Evaluation Framework (QEF)". In: *Conselho Editorial/Consejo Editorial* (Jan. 2008), p. 16.

[30] Albert Graf and Peter Maas. "Customer value from a customer perspective: A comprehensive review". In: *Journal für Betriebswirtschaft* 58 (Apr. 2008), pp. 1–20. DOI: 10.1007/s11301-008-0032-8.

[31] Zeki Çelikbaş. "What is RSS and how can it serve libraries?" In: (Jan. 2004).

[32] A Ais Alimuddin et al. "Design and Implementation of REST API for Academic Information System". In: *IOP Conference Series: Materials Science and Engineering* 875 (July 2020). DOI: 10.1088/1757-899X/875/1/012047.

[33] Atlassian. *The Confluence Cloud REST API*. URL: https://developer.atlassian.com/cloud/confluence/rest/v1/intro/ (visited on 01/01/2023).

Appendix A

Company Survey Graphs Divided by Teams

The following Figures A.1, A.2, A.3, and A.4 present the responses given to the survey presented in section 3.2.2, divided between the two teams (design and verification). The left side (yellow) represents the design team, while the right side (blue) consists of the verification team.

Question 3 - Would it be useful to have a system in which configurable regression tests are performed on labels as they're released, automatically?

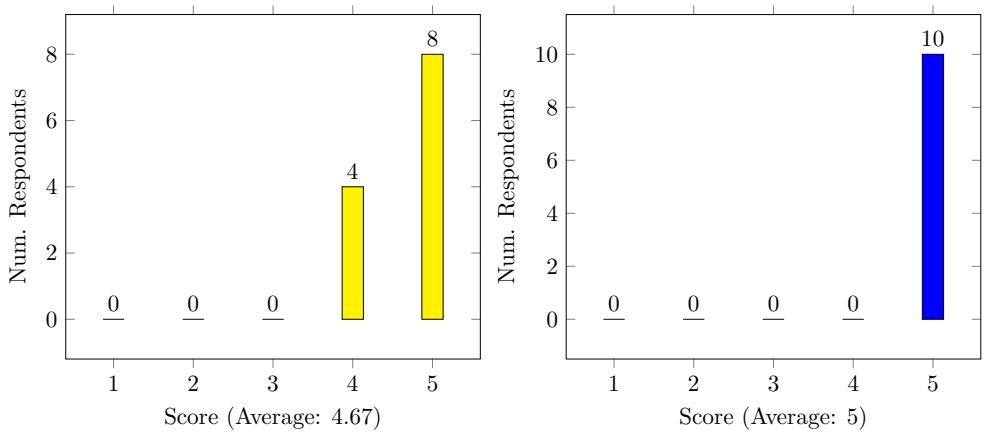


FIGURE A.1: Team Responses to Survey Question 3

Question 4 - Would it be useful to have all the results of the tests run with this system available on an online dashboard for users to visualize more easily, featuring tracking history through which users could also see the results of past tests?

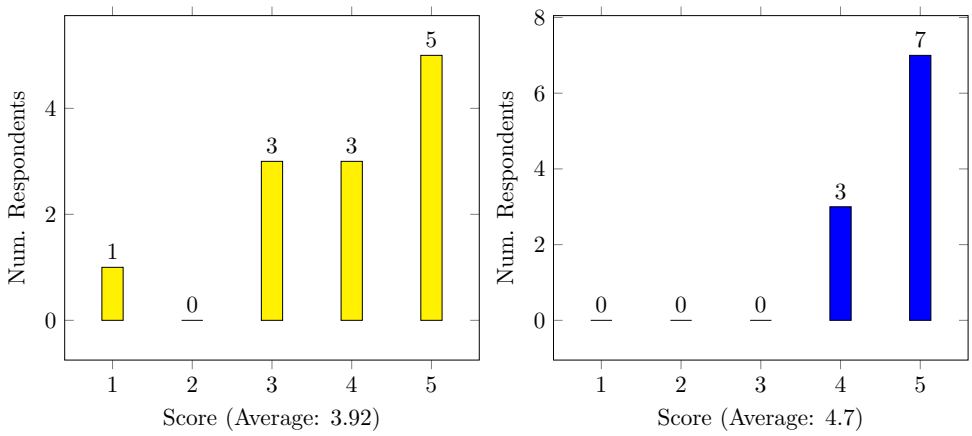


FIGURE A.2: Team Responses to Survey Question 4

Question 5 - Would it be useful to write down the information for a regression on a table and have it run automatically, be it a repetitive schedule (such as every Wednesday), or a single run for a specific date (for example, in 3 days from now)?

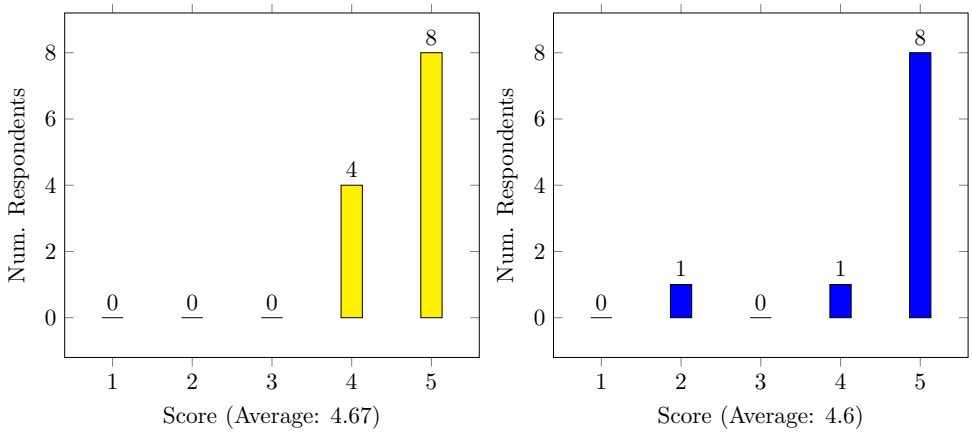


FIGURE A.3: Team Responses to Survey Question 5

Question 6 - Would it be useful to have an alternative system in which the user can use the information on the table in order to run a regression immediately with a single click?

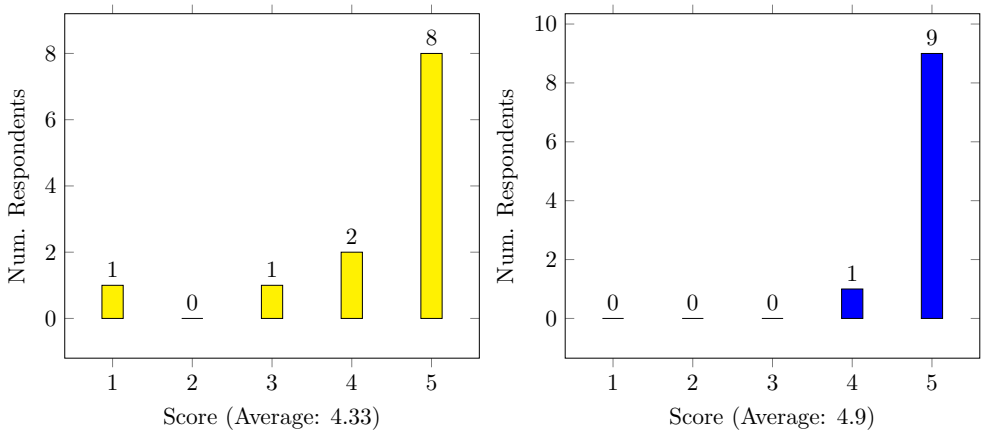


FIGURE A.4: Team Responses to Survey Question 6

Appendix B

List of All Settings Implemented in Jenkins Configuration

The following tables list all of the settings implemented in the Jenkins configuration file, as well as their uses. Due to these tables being too extensive, they were left out of Chapter 5 and included here instead.

B.1 General Settings

The following settings in Table B.1 are used in more than one pipeline, often required in all pipelines.

Table B.1: Jenkins Configuration - General Settings

General Settings	
Setting Name	**Functionality**
target_disk	Default disk to be used. Other paths build off of this.
disk_space	Max space for regressions in the disk specified.
jenkins_email_admin	Mailing list for admin emails (pipeline failures and monitoring).
jenkins_email_results	Mailing list for regression results (notified of start and finish).
protocol_names	Mapping of protocol names and aliases.
protocol_group_1 protocol_group_2 protocol_group_3	Groups for protocols. Depending on the protocol, the sync from the SCM platform may be slightly different.
environment_types_to_run	Default environments to use for the regression.

B.2 Check Email Releases Settings

Table B.2 presents all of the settings used for the check_email_releases pipeline.

Table B.2: Jenkins Configuration - Check Email Releases Settings

check_email_releases Settings	
Setting Name	**Functionality**
check_email_releases_workspace	Path to workspace to be used for pipeline.
check_email_releases_script_dir	Path to Python script used to collect emails.
check_email_releases_script_name	Name of Python script used to collect emails.
json_output_path	Path to where JSON file tracking emails read should be located.
conf_files_output_path	Path to where regression information files should be created.
mailing_group_link	Main link to mailing groups RSS feed.
mailing_group_name	Name of mailing group for emails.
mailing_group_linklist_ responses_number	Max number of emails to collect.
mailing_group_linklist_ responses_hours_number	Max number of hours ago to check for emails.
filter_list	Filters that should be present in email title.
exclusion_list	Filters that should not be present in email title.
regression_tracking_file_path	Path to file that tracks regressions started.
regression_tracking_file_name	Name of file that tracks regressions started.
check_email_releases_ timeout_minutes	Timeout in minutes for pipeline.
prevent_running_if_ no_space_guarantee	1 if regression should not run if not enough space is guaranteed.
mandatory_labels	Variables that, if missing, means that regression is not valid.
check_email_releases_ fail_email_html	Path to HTML file used for email if pipeline fails.
check_email_releases_ launch_email_html	Path to HTML file used for email if pipeline succeeds.

B.3 Run Preliminary Check Settings

Table B.3 presents all of the settings used for the run_preliminary_check pipeline.

TABLE B.3: Jenkins Configuration - Run Preliminary Check Settings

run_preliminary_check Settings	
Setting Name	**Functionality**
run_preliminary_check_name	Name of the pipeline on Jenkins.
run_preliminary_check_workspace	Path to workspace to be used for pipeline.
regression_conf_files_path	Path to where the regression information files will be saved on the Perforce depot.
tech_specific_conf_files_path	Path to where project-specific configuration files should be saved on Perforce.
tech_specific_conf_file_name	Name format of the project-specific configuration files.
global_conf_file_path_group_1 global_conf_file_path_group_2 global_conf_file_path_group_3	Depending on protocol group, different global files may be used.
variables_to_compare	List of variables to compare with previous release. If they all match, stop running. Only used in Email Flow.
run_preliminary_check_ timeout_minutes	Timeout in minutes for pipeline.
preliminary_check_fail_email_html	Path to HTML file used for email if pipeline fails.

B.4 Run Tests Settings

Table B.4 presents all of the settings used for the run_tests pipeline.

Table B.4: Jenkins Configuration - Run Tests Settings

run_tests Settings	
Setting Name	**Functionality**
run_tests_name	Name of the pipeline on Jenkins.
run_tests_workspace	Path to workspace to be used for pipeline.
generate_conf_on_ manual_run	1 if manual runs should generate regression information files to store in Perforce.
run_tests_timeout_ minutes_automatic	Timeout in minutes for pipeline if automatic run (Email Flow).
run_tests_timeout_ minutes_manual	Timeout in minutes for pipeline if manual run (Confluence or manual run of run_tests).
run_tests_result_ email_html	Path to HTML file used for email when pipeline ends (regardless of outcome).

B.5 Workspace Cleanup Settings

Table B.5 presents all of the settings used for the workspace_cleanup pipeline.

Table B.5: Jenkins Configuration - Workspace Cleanup Settings

workspace_cleanup Settings	
Setting Name	**Functionality**
workspace_cleanup_name	Name of the pipeline on Jenkins.
folders_to_delete	Path to folders that should be deleted (regression folders).
time_window_hours	How old regressions need to be to be cleaned up automatically (Email Flow).
time_window_hours_confluence	How old regressions need to be to be cleaned up automatically (Confluence Flow).
minimum_space_ recommended_GB	Warn users when disk has less than this space left.
minimum_space_per_ regression_GB	Bare minimum space required per regression.
workspace_cleanup_workspace	Path to workspace to be used for pipeline.
workspace_cleanup_fail_ email_html	Path to HTML file used for email if pipeline fails.
workspace_cleanup_warn_ email_html	Path to HTML file used for email if space is running low.
regression_argument_weights	Mapping of regression arguments, their values, and the corresponding space requirement associated. (Example: A coverage regression needs more space)

B.6 Confluence Regression Check Settings

Table B.6 presents all of the settings used for the confluence_regression_check pipeline.

Table B.6: Jenkins Configuration - Confluence Regression Check Settings

confluence_regression_check Settings	
Setting Name	**Functionality**
confluence_regression_check_ workspace	Path to workspace to be used for pipeline.
confluence_regression_check_ schedule_space	Confluence space under which the regression schedule page is located.
confluence_regression_check_ schedule_page	Confluence page where the regression schedule table is located.
confluence_regression_check_ user_space	Confluence space under which the user configuration page is located.
confluence_regression_check_ user_page	Confluence page where the user configuration table is located.
confluence_regression_check_ script_dir	Path to Python script used to parse Confluence tables.
confluence_regression_check_ script_name	Name of Python script used to parse Confluence tables.
confluence_regression_check_ launch_email_html	Path to HTML file used for daily run of pipeline.
confluence_regression_check_ launch_email_single_html	Path to HTML file used for user run of pipeline.
confluence_regression_check_ line_error_email_html	Path to HTML file used if there is a line error.
confluence_regression_check_ timeout_minutes	Timeout in minutes for pipeline.

B.7 Post Regression to Confluence Settings

Finally, Table B.7 presents all of the settings related to the Python script responsible for uploading the regression results to the Confluence dashboard.

Table B.7: Jenkins Configuration - Post Regression to Confluence Settings

post_regression_to_confluence Settings	
Setting Name	**Functionality**
confluence_dashboard_space	Confluence space under which the top level dashboard page is located.
confluence_dashboard_page	Confluence page under which the top level dashboard table is located.
confluence_dashboard_max_ lines	Max number of lines one each of the higher hierarchy page tables.
confluence_dashboard_result_ limit_specific	Max number of co-existing result pages at the last level of the hierarchy.
confluence_dashboard_ script_dir	Path to Python script used to upload regression information to Confluence dashboard.
confluence_dashboard_ script_name	Name of Python script used to upload regression information to Confluence dashboard.