UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Technical Debt Analysis and Project Architecturization of a Jenkins Platform based on Groovy

by Sergio Preciado Orozco

with the collaboration of:

opentrends

Bachelor Thesis

Specialization in Software Engineering

Thesis supervisor: Jonattan Nieto Sánchez

Thesis examiner: Xavier Burgués Illa

GEP Tutor: Paola Lorenza Pinto

23rd June 2022

Abstract

Currently, Technical Debt (TD) is a latent problem in the vast majority of software projects. Due to the rapid growth of the market, its business vision is focusing on reducing the time-to-market of the product, leaving aside the internal quality of its code. As a result, the global annual cost of maintaining such poor quality code comes to approximately $85 billion.

The thesis focuses on deeply analyzing a corporate platform with a heavy TD and defining a new architecture for it based on its requirements, prioritizing the quality of the product while reducing its technical debt. To achieve this, I will use refactoring techniques, implementation of new functionalities and the definition of internal protocols for the team.

In the thesis, the steps to follow to analyze and re-architect a project with similar characteristics are documented. In addition, strong awareness is raised regarding the technical debt and its problems, an issue that directly affects the code and indirectly impacts the mental health of its developers.

Resumen

Actualmente, la Deuda Técnica (DT) es un problema latente en la gran mayoría de proyectos software. Debido al rápido crecimiento del mercado, su visión empresarial está cada vez más enfocada a reducir el time-to-market del producto, dejando de lado la calidad interna de su código. Por ello, el coste global anual de mantener dicho código de mala calidad, asciende aproximadamente a 81.000 € millones.

La tesis se centra en analizar profundamente una plataforma corporativa con mucha DT y definir una nueva arquitectura para ella, teniendo en cuenta sus requerimientos y priorizando la calidad del producto mientras se reduce su deuda técnica. Para ellos se emplearán técnicas de refactorización, implementación de nuevas funcionalidades y definición de protocolos internos para el equipo.

En la tesis quedan documentados los pasos a seguir para analizar y rearquitecturizar un proyecto con unas características similares. Además, se crea una fuerte conciencia sobre la deuda técnica y sus problemas, una cuestión que afecta directamente al código e indirectamente a la salud mental de sus desarrolladores.

Resum

Actualment, el Deute Tècnic (DT) és un problema latent a la gran majoria de projectes software. A causa del ràpid creixement del mercat, la visió empresarial està cada cop més enfocada a reduir el time-to-market del producte, deixant de banda la qualitat interna del seu codi. Per això, el cost global anual de mantenir aquest codi de mala qualitat, puja aproximadament a 81.000 € milions.

La tesi se centra a analitzar profundament una plataforma corporativa amb molt DT i definir-ne una nova arquitectura, tenint en compte els seus requeriments i prioritzant la qualitat del producte mentre es redueix el seu deute tècnic. Per aconseguir això, es faran servir tècniques de refactorització, implementació de noves funcionalitats i la definició de protocols interns per a l'equip.

A la tesi queden documentats els passos a seguir per analitzar i rearquitecturitzar un projecte amb unes característiques similars. A més, es crea una forta consciència sobre el deute tècnic i els seus problemes, una qüestió que afecta directament el codi i indirectament la salut mental dels seus desenvolupadors.

Acknowledgement

I would like to express my deepest gratitude to **Jonattan Nieto Sánchez**, for his teachings, professionalism, and guidance. This thesis would not have been possible without **Xavier Burgués Illa**, for his good criteria, advice and corrections; and **Paola Lorenza Pinto**, for her excellent suggestions in the GEP phase. Additionally, I would like to extend my sincere thanks to **the thesis defense committee**, for their time, expertise and interest.

Many thanks to **Opentrends** and especially to the very professional members of the **MALM team** for allowing me to work with them on this exciting project.

Special thanks to **my parents**, for trusting and believing in me from the first day of college. Finally, I'd like to thank **Karolina Levanaite**, for being by my side during these four hard months of thesis, for your encouragement and your kindness.

Thank you.

Table of Contents

List of Tables

List of Code Blocks

List of Figures

1 Context and Scope

1.1 Introduction

Every hour wasted on code maintenance is an hour in which a developer could have done wonderful things.

All software developers will, at some point, come across **bad code**. Sometimes driven by tight deadlines or unawareness about code quality and best coding practices, programmers are **somewhat forced to code without thinking about its future repercussions**. This situation can happen on the same project several times, until it becomes a **real problem**.

As Isaac Lyman says in its article, *Code quality: a concern for businesses, bottom lines, and empathetic programmers (2021)* : "Tech companies with a poor understanding of code quality can launch quickly and see success in the short term. But in doing so, they incur an invisible debt that grows every time the code is altered. Once the product exceeds a very low threshold of complexity, the debt comes due, gradually consuming the productivity of their development team and the usability of their software.", as shown on Figure 1.

These types of dangers are called **Technical Debt.**

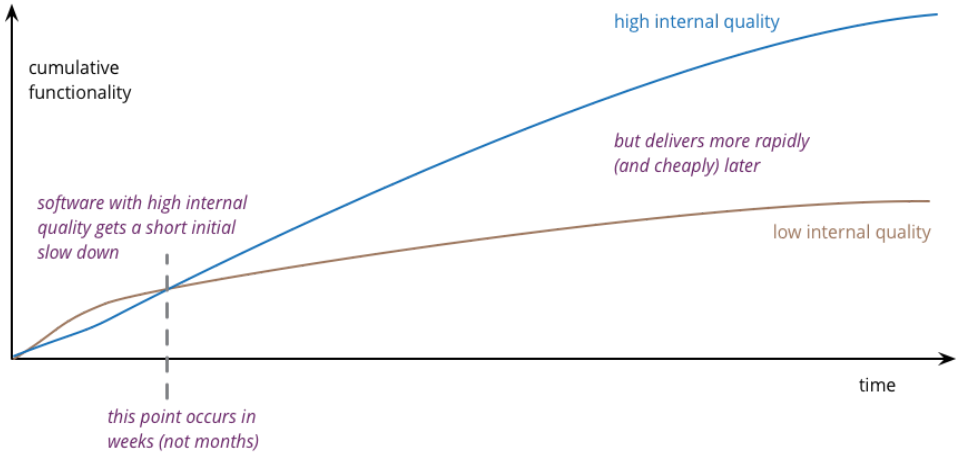


Figure 1 – Software cumulative functionality over time (Fowler, 2019).

This thesis is aimed at **understanding the impact that the current state of a project has on the final product and on the productivity of its development team.** I start from the premise that the project has not scaled as it should over time. Thus, it is necessary to analyze the current situation in order to take actions that have an immediate impact on its overall productivity and performance.

The purpose of this study is **not to be judgmental** about the current state of the project; on the contrary, it is intended to perform a diagnosis in an effort to achieve immediate improvements that may have an impact on the service offered. I am very grateful for the help of the teams that collaborated with me through this project, and of those who provided me with relevant information of any kind.

1.1.1 Project contextualization

This project, *Technical Debt Analysis and Project Architecturization of a Jenkins Platform based on Groovy*, is a bachelor thesis of the Informatics Engineering degree, for the Software Engineering specialization, done in the Barcelona School of Informatics (FIB) of the Polytechnic University of Catalonia (UPC). It is done with the collaboration of **Opentrends**[1], a pioneer consulting and engineering company based at Barcelona, with more than 400 employees and with offices in Madrid (Spain), Silicon Valley (California, USA) and Kerala (India).

The thesis is **directed and supervised by Jonattan Nieto Sánchez**, Senior Software Engineering Manager at Opentrends, and **examined by the speaker and professor Xavier Burgés Illa**, of the UPC's ESSI department.

Furthermore, Opentrends is attached to the United Nations Global Compact, and it is a firm committed to the environment and sustainability that has recently obtained ISO 1400-1 (Opentrends, 2021). With that in mind, this thesis has been developed following the same guidelines.

1.1.2 Concepts and definitions

It is important to know the definition of some basic concepts that will be present throughout the thesis, in order to enhance the reading and comprehension for the reader.

a) CI/CD

The "CI" refers to **Continuous Integration**, which is an automation process for developers. Successful CI means new code changes to an app are regularly built, tested, and merged to a shared repository. It's a solution to the problem of having too many branches of an app in development at once that might conflict with each other. (Red Hat, 2018)

The "CD" refers to **Continuous Delivery** and/or **Continuous Deployment**, which are related concepts that sometimes get used interchangeably. Continuous Delivery usually means a developer's changes to an application are automatically bug tested and uploaded to a repository, where they can then be deployed to a live production environment by the operations team. The purpose of continuous delivery is to ensure that it takes minimal effort to deploy new code. Continuous Deployment can refer to automatically releasing a developer's changes from the repository to production, where it is usable by customers (Red Hat, 2018).

[1] Opentrends – https://www.opentrends.net/en

b) Jenkins

It is the leading, self-contained, open source automation server which can be **used to automate all sorts of tasks related to building, testing, and delivering or deploying software**. (Jenkins, 2020) Jenkins is written in Java and provides hundreds of plugins to improve and extend its functionalities, making it one of the best tools for Continuous Integration and Continuous Delivery at the present time.

c) Pipeline

A Pipeline is a user-defined model of a CD pipeline. A Pipeline's code **defines your entire build process**, which typically includes stages for building an application, testing it and then delivering it. (Jenkins, 2020)

d) Node

A node is a machine which is part of the Jenkins environment and is capable of executing a Pipeline. (Jenkins, 2020)

e) Stage

A stage block defines a **conceptually distinct subset of tasks** (or steps) performed through the entire Pipeline (e.g., "Build", "Test" and "Deploy" stages), which is used by many plugins to visualize or present Jenkins Pipeline status/progress. (Jenkins, 2020)

f) Step

A **single task**. Fundamentally, a step tells Jenkins *what* to do at a particular point in time (or "step" in the process). For example, to execute the shell command `make` use the sh step: `sh 'make'`. (Jenkins, 2020)

g) ALM

Application Lifecycle Management (ALM) is the people, tools, and processes that manage the life cycle of an application from conception to end of life. ALM supports agile and DevOps development approaches by integrating together a list of disciplines: project management, requirements management, software development, testing and quality assurance, deployment, and maintenance; and enabling teams to collaborate more effectively in the organization. (Red Hat, 2018)

h) MALM (Mobile ALM)

Opentrends' MALM team provides the implementation of an ALM to support the lifecycle of mobility applications of its customers through its principal product: **Mobile ALM.**

Just to give a first idea of the size of the MALM project, it has only four Jenkins CI pipelines, but with a total of **more than 40,000 lines of code.** These pipelines are currently being **used by more than a 1,000 users** and with **approximately 100 concurrent daily users,** according to Opentrends internal statistics.

Figure 2 shows a simplified diagram of the components of the current MALM project.

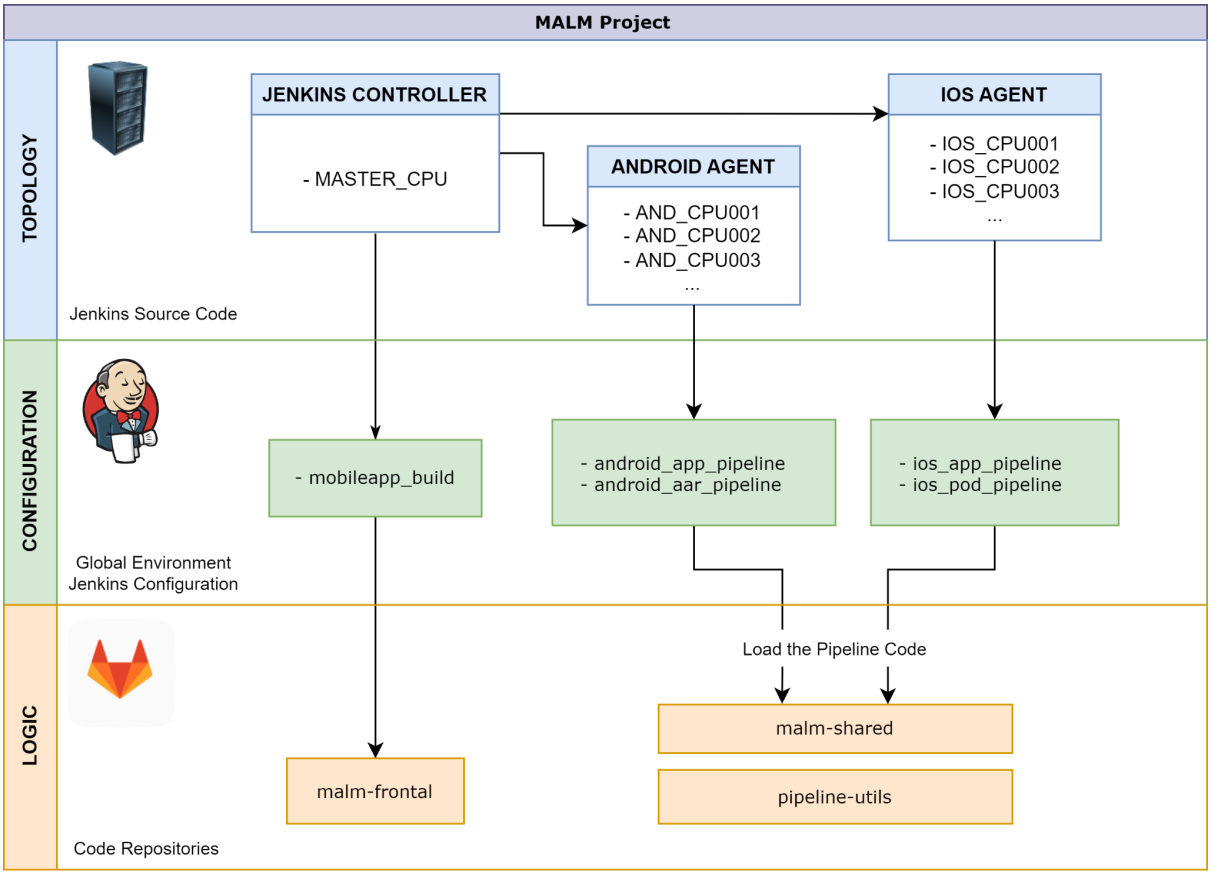


Figure 2 – Simplified Component Diagram of the current MALM project.

1.1.3 Problem to be solved

At the current time, MALM project is fully functional, has been granting its service since 2017 and its changing and evolving teams have continued to expand its functionalities until now. Of course, some people may think, “Why fix something that is not broken?”, but that point comes into question when the cost of maintaining the project’s code is rather high. In that case, **rethinking the project architecture** and its design is something that should be taken into consideration.

Although, in this case, it is not just “bad code” that is causing the increase in maintenance cost. After a superficial analysis of the project's code, we can identify some of the main problems that will be addressed in this thesis.

First, it is a project that appears very early in time, with no community support to develop this type of complex automation products. It is focused on iOS and Android app generations in parallel, but with different requirements. In addition, it has been evolving over time, lacking a well-defined architecture that needs to be focused on easing the programmer's work and allowing an efficient growth of the platform and its services. The software does not respect the

classical design patterns (SOLID: Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion). Second, the source code of the project lacks a suitable versioning, which complicates its distribution. Also, the source code is divided into two or more code repositories, which separates the Utils module (auxiliary functions) from the rest of the program's source code, clearly complicating the development of new functionalities, product management and bug identification.

Third, Unit Testing is not implemented, nor are basic regression tests correctly defined. Fourth, the pipeline code is unorganized and may need a complete redesign of its Stages and Steps, making it modular, reusable and divided into semantically similar Stages. Fifth and last, there are some less significant concerns, such as code refactoring and cleanup of deprecated code, there is no Groovydoc-style code documentation, and some recommended best practices for Groovy and Jenkins are not being followed.

1.1.4 Stakeholders

We can identify two main groups of stakeholders, divided according to their interest. The first one, the stakeholders directly involved with the thesis and its development; The second one, stakeholders who are not involved with the project itself, but in its benefits.

Directly involved Stakeholders are the thesis supervisor and Senior Software Engineering Manager at Opentrends, Jonattan Nieto Sánchez, the thesis examiner, Xavier Burgués Illa and myself as the researcher, Sergio Preciado Orozco.

On the other hand, the stakeholders who will benefit from the project's resulting product are: The MALM team, as the main beneficiary of the project due to the fact that the thesis focuses on enhancing their product source code, greatly decreasing maintenance and testing costs and facilitating future functionality developments; Opentrends, as the company that owns the final product and is, of course, interested in that its employees invest more time on I&D, rather than maintaining “bad code” and launching time-consuming tests; Users of MALM, as the users of a potentially much more robust platform where they can deploy their APPs for their customers to use; Potential clients of the MALM service, as the possible new clients of a platform that will be ready for the long-term growth, they will be interested in having the best service they can find.

In a larger scope, we can also consider the Users of the Apps deployed by MALM, as the millions of users that will be using the APPs deployed by the service, they will want to use an application that is correctly tested and keeps updating with new features on a regular basis.

1.2 Justification

1.2.1 Previous studies

In the last decade, studies on technical debt and the relevance of quality code have been decisive in bringing value and visibility to best practices and to the importance of a well-designed architecture.

The Developer Coefficient study (Stripe and Harris Poll, 2018) found out that **the average developer spends 17.3 hours of their 41 weekly working hours (42%) dealing with technical debt and maintenance issues,** of which 3.8 hours are spent just on debugging "bad code", or poor quality code that's difficult to maintain. The opportunity **cost of bad code comes to $85 billion annually,** resources that could otherwise be used to build better software. (Avery, 2018)

The impact caused by the consequences of poorly designed architecture are even more significant and detrimental, which makes it necessary to coin a new term to differentiate this TD from the rest: the **Architectural Technical Debt** (ATD).

Another study on the *Impact of Architectural Technical Debt on Daily Software Development Work* (Besker et al., 2017), in which data from 258 participants from different projects were gathered, states that the negative consequences of Technical Debt is an area of increasing interest, and more specifically the Architectural aspects of it. Besides the negative effects of Architectural Technical Debt on the overall software product quality in terms of hindering evolution and causing high maintenance costs, Architectural Technical Debt also has a significant negative impact on software developers' daily work.

In addition to all the studies and articles referenced in the other sections of this thesis, this article describes some of the needs and concerns that we will address, *Code quality: a concern for businesses, bottom lines, and empathetic programmers*[2] from the StackOverflow Blog team.

1.2.2 Justification of the approach

Based on these previous studies, I decided to design a completely new architecture for the project, keeping the current paradigm of its Jenkins approach: a CI/CD platform based on jenkins pipeline scripts. It provides **an essential service for day-to-day application development** and any change must be carefully reviewed, mitigating risks as much as possible. It is important not to change its core functionalities with the redesign, because it is **an already deployed product in production**, with **more than 1000 users per day** and a **high resilience to change** (any modification will have a significant impact on its code). Therefore, I will define the roadmap in order to achieve the objectives listed in the following section.

[2] Link to the article here.

1.3 Scope of the thesis

1.3.1 Objective and sub-objectives

The main objective of this thesis is to **analyze in great depth the MALM project** and then **propose an architecture** for its outdated software (with high technical debt) that allows: easy maintenance; scalability (functionally extendable); reliability (end to end automatic testing, reliability of changes without the need for manual testing, etc.). Then, the MALM team will follow the roadmap to implement the necessary improvements to increase the overall quality of the final product. By doing so, it is expected to considerably **reduce the cost of code maintenance** and **provide the project with a more flexible architecture for its scalability**.

The resulting study of this thesis is based on the evidence and improvements of this specific product, but is extrapolable to any software project with similar characteristics.

As stated at section 1.1.3 Problem to be solved, with the first project analysis we identified a list of needs that should be addressed, some of them crucial to the thesis development and the accomplishment of the main objective. Of course, there are many improvements that can be applied to a project of this magnitude, that is why it is essential to narrow down the main objective into more specific sub-objectives. Then, prioritize those that can provide the most benefits in the long-term.

Required

⇒ Architecturization of the software

It can be divided into more specific sub-objective such as: **Define a proper file hierarchy** for the project; **Pattern the code** based on its multiple use cases, taking into account the limitations of Jenkins; **Define a Best Practices** guidelines; **Unify the Workspace, IDE and its extension** between all team members. The definition of a good workspace and its tools can increase productivity. Also synchronizing the team's development environment, making it easy to exchange knowledge and various tips and tricks.

⇒ Use Gradle to create a new repository for the project.

Convert the existing project repository into a real **Groovy Project**. At the current time, the MALM code is just groovy scripts and does not allow test execution, class indexing nor dependency injection. With this Groovy style project, we can implement an intuitive versioning and release strategy for the project's code.

⇒ Unify the scripts repositories.

The Utils functions module of the project is hosted on a corporate git repository called *pipeline-utils*. The rest of the source code is hosted on another repository called *malm-shared*. This first repository, **pipeline-utils, has to be unified into malm-shared** to simplify the architecture.

⇒ Redesign the pipelines and its Stages and Steps.

We need to **define more specific and modular Stages** based on their purpose, divide its computational work into atomic Steps, and correctly manage exceptions and warnings of such steps.

⇒ Implement Unit Testing.

Design and **implement tests for the pipelines Stages and Steps**. Testing the pipelines and its code before its deployment with the use of sample data (mock values) will greatly increase the productivity of the team, as stated earlier at section 1.2.1 Previous studies.

Significant

⇒ Redesign the Jenkins global variables.

Analyze and identify the environment variables file and move into the code the ones that are not acting as a global variable, or need to be changed on the fly.

⇒ Document the code using Groovydoc.

For every Class, function, or method of the Jenkins Controller project, document it and generate a Groovydoc file.

⇒ Upgrade the technological stack of the code and the software platform.

Analyze the project to validate if an update of the software, plugins, and libraries can be possible on the existing corporate platform.

Nice-to-have

⇒ Cleanup of deprecated code.

Delete old comments, deprecated functions and unused files and classes.

⇒ Refactor the name of code variables to provide them with useful semantic information.

Identify those variables that have no useful semantic information and rename them.

⇒ Modify the code so that it complies with Groovy and Jenkins best practices.

Next steps

⇒ Convert repository code into a real Shared Library.

Analyze the project to validate if its conversion to a Shared Library is beneficial for the product and, in that case, implements the required changes and conversion.

⇒ Redesign the Jenkins pipeline parameters. UX improvement.

Modify the parameters of the Jenkins pipelines to improve user experience and easily determine what is exactly executing.

In the course of this thesis, I will be addressing the Required and Significant objectives. Nice-to-have objectives will be indirectly dealt by the MALM team after an analysis commented on the thesis. Next steps objectives will be proposed as improvements in the future.

1.3.2 Requirements

As stated in earlier sections, this thesis is not focused on developing new functionalities of a product, but rather on redesigning and architecturizing its project, maintaining all of its previous features.

The definition of functional and non-functional requirements of the thesis may be unclear due to its theoretical nature, but we can state some of the requirements that the product now meets and should continue to meet after the implementation of the new architecture:

Functional requirements

- ⇒ The software tests the APP's unit tests.
- ⇒ The software builds and deploys the APP on iOS and Android architectures.
- ⇒ The software generates versioned libraries/modules developed for an application.
- ⇒ The software notifies the users after a valid or an invalid execution.
- ⇒ Pipeline parameters are configurable from the frontend.

As for the thesis itself:

- ⇒ The software meets the same requirements as before its redesign.

Non-functional requirements

- ⇒ Response time between failure does not exceed 24 hours.
- ⇒ Users can use the service throughout the week at any time during the day.
- ⇒ All users can see all the public pipeline executions and logs.
- ⇒ Only the users with the role "admin" can manage and configure the pipelines.
- ⇒ The service interface has to be user-friendly and easy to use.

As for the thesis itself:

- ⇒ The software is easily scalable, assuring less than 4 hours to implement a new basic feature.
- ⇒ Use of best practices and a good readability on the project's code.

1.3.3 Potential obstacles and risks

Throughout the course of the thesis, there are some potential obstacles and risks that we should pay attention to in order to prevent them from affecting the project.

> Unclear documentation

Due to the current multiple paradigms of Jenkins, its documentation can be ambiguous, freely interpreted and without a consensus of the community. This can complicate the correct development of the thesis, so it is important to invest a good percentage of the project time in research.

> Complex Project in Production

It is a very complex product already deployed in production, with many active users using its service. It is normal that we may encounter strong resistance to change. In order to mitigate the risk, we can invest a percentage of the thesis' development time in the analysis of the actual project.

> Tight deadlines

Considering the short time frame in which the thesis must be developed (four months), it is important that we do not lose sight of this risk. Sometimes developers have tight deadlines and, in some cases, the team is unable to meet these expectations. We can mitigate this risk by creating a thorough project plan that allow us to set realistic deadlines

> Hardware and Infrastructure problems

In projects where the infrastructure is managed by another team, or depends directly on third parties, it is usual that from time to time the infrastructure may be unavailable or have occasional errors. Hiring a solid infrastructure with a good technical service is essential to mitigate this risk.

> External risks

There are also some rare, unpredictable risks that can be encountered during the development. They can include natural disasters, changes in laws and economic shifts. At first glance it may seem extreme, but in 2018 this same project suffered a crash of its MacStadium[3] machines due to Hurricane Michael as it passed through Atlanta Data Centers (National Weather Service, US Dept of Commerce, 2018). It can be challenging to avoid external risks, but there are actions we can take to mitigate them, like maintaining backups or segregating the infrastructure servers.

[3] MacStadium – https://www.macstadium.com/

1.4 Methodology and rigor

1.4.1 Work methodology

Defining a suitable methodology for the thesis development is a key aspect to assure its profitable outcome. As for now, the MALM team has been using Agile methodologies for a long time, and currently they are working under the SCRUM framework guidelines.

Agile methodologies are a perfect fit for this kind of project where rapidly changing requirements, incremental delivery of functionalities and a correct response to user feedback is needed. As stated at the CHAOS Manifesto, *"Software applications developed through the **agile process have three times the success rate of the traditional waterfall method** and a much lower percentage of time and cost overruns"* (Standish Group International, Incorporated, 2010). Having that in mind, the question now is: "why choose Scrum over other agile methodologies like Kanban?".

A study on *Quantitative Analysis of Agile Methods Study* (J. Reifer & Hastie, 2017) found out that Scrum is the most popular agile framework by a great amount over small and medium projects, being the best approach for those small projects (one agile team) in which the agile methodology is suitable.

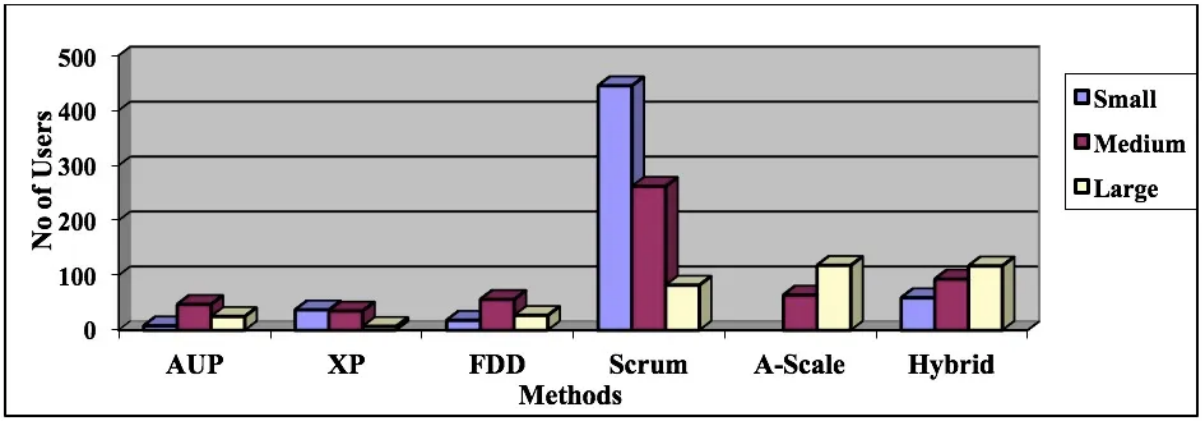


Figure 3 – Framework Usage by No. of Organizations by Size of Project[4].

It is not a secret that Scrum, as an Agile framework, does not adapt to any type of software development project. In fact, it is a common mistake for companies to risk implementing a Scrum on a large project with aggressive challenges and tight deadlines. However, after analyzing the scope of the project, we can state that the SCRUM framework fits the needs of the project. Therefore, it is the right decision to continue using it as the team has been doing so far, thus choosing it as the framework for the development of the thesis.

[4] Notes and Legend of Figure 3 on the List of Figures section.

1.4.2 Monitoring tools and validation

Following the Scrum framework guidelines, our team is composed by: a Proxy Product Owner[5], as the middleman between the Product Owner (in our case, an external corporate owner) and the people developing the product. This same person also takes the role of a scrum master, as the person who guides and instructs the team to comply with the rules and processes of the framework; the rest of the development team, as the professionals with the necessary technical knowledge who develop the project.

The Proxy PO, in consensus with the developers (me included), prioritizes and sets the tasks to be completed in the first ***Sprint*** (the basic unit of work for a Scrum team) at a meeting called the ***Sprint Planning***. During the sprint we will be conducting a ***Daily Scrum*** (sometimes known as Daily, in order to abbreviate) which redundantly consist of a brief daily meeting where each member of the team has to answer to three simple questions: “What did I do yesterday?”, “What am I going to do today?” and “What help do I need?”. (Digité, Incorporated, 2021)

After every sprint, but before the release of its features, the team will meet for a ***Sprint Review*** where each member of the team will review and explain their changes to the rest of the team. Additionally, when a complete iteration of the Sprint is done, the team will meet again for a ***Sprint Retrospective*** in order to identify those tasks that went somewhat wrong, and could be improved or avoided on future sprints.

This will help the team to organize their tasks and create a plan for the next 24 hours, also allowing them to detect any blockers or scheduling changes in the development.

All these tasks and information will be registered on a monitoring tool called Rally Software[6]. It is a card-based, enterprise-class platform specifically created to increase agile development practices where you can plan tasks, organize them and monitor them. The MALM team has been using it for the past year so, after checking that the tool fulfills all the needs, I decided to use the same tool in order to unify the monitoring technologies that the team will use.

[5] Proxy Product Owner – https://www.scrum.org/resources/blog/what-proxy-product-owner-why-it-found-so-often
[6] Rally Software – https://aptasolutions.com/rally-software-ca-agile-central/

1.5 State of the Art

Although the thesis focuses on the re-architecture and refactoring of many of the components of a Jenkins platform, I think that it is interesting to mention the current state-of-the-art of Jenkins and find out how it is used by the community nowadays.

Currently, in the field of CI/CD, specifically in Jenkins, there are 3 components that are essential to simplify and optimize the process of Continuous Integration. These components are:

- ⇒ **Jenkins.**
  - » It is an open-source automation platform widely used by the community, written in Java and with a large catalog of plugins to adapt CI to the user's needs. It is used to build and test the software in a continuous way, facilitating the integration of code changes for developers.
  - » There are two different syntaxes for developing Jenkins pipelines, Scripted and Declarative. Currently, the community is divided as each has its benefits and weaknesses, although there is a growing trend towards Declarative Syntax.
- ⇒ **GitHub.** (or GitLab)
  - » GitHub is a collaborative Source Control Management tool that allows to host and version the source code of any software project. It has many possible integrations with other tools in addition to its functionalities, which makes it an essential tool for programmers.
- ⇒ **Docker.**
  - » It is an Operating System virtualization tool in the form of Containers. It is used to simplify the process of build, execution and distribution of software products.

The idea is to unite these three components in order to be able to launch CI builds with **Jenkins** directly from a MR (Merge Request) of any branch of the **GitLab** repository; To be able to see the results of that CI from the MR comments, before executing the Merge; To use **Docker** to separate the CI build from the Jenkins environment, and thus optimize time and resources.

In this way, it is achieved:

- ⇒ Having centralized Jenkins configuration, in addition to being able to define one for each Job.
- ⇒ Being able to launch CI/CD Jobs from specific branches of the GitHub repositories.
- ⇒ To have the CI result available in the comment of the MR of GitHub.
- ⇒ Prevent the branches from breaking the Build before the merge.
- ⇒ Relieve Jenkins server load by running jobs that require more resources on Docker.
- ⇒ Reduce overall CI execution times.
- ⇒ Make it easier for the user to use (UX).

2 Temporal Planning

The development of the thesis is planned to **start on February 21, 2022**, and to be **completed on June 27, 2022**, this last date being the same as the defense date of the thesis. It has a total workload of 18 academic credits, at 30 hours per credit, that is ~540 hours of expected work.

Between February 21 and June 27 there is a total of 4 months and 6 days (126 days). Of those 126 days, 90 are working days, which means that for a smooth and correct development of the project, 6 hours a day of work are required.

2.1 Description of tasks

The project's tasks, whose information will be used to generate the Gantt diagram in section 2.2 Gantt diagram, are listed below. I will be using a format I like to call Task Table Format, where the information of every task is defined following this table:

ID: EX1	**Name**: Example task for the thesis	**Type**: Documentation
Description: In the example, the fields of the task and their possible values must be defined.		
Resources: Personal workspace.	**Dependency**: D1	**Estimation**: 1 h

Where:

- **ID**: It is the task identifier. It must be unique among all the tasks, in uppercase and no longer than 3 alphanumeric characters. (e.g. EX1, PM3, T1)
- **Name**: The name of the task. It should be descriptive enough to easily recognize the task and no longer than 8 words.
- **Type**: It is the typology of the task. It is a purely informational value to provide more context to the person in charge of executing the tasks. It can be: Documentation, Analysis, Research, Development, or Management.
- **Description**: A brief description of the task in order to give the developer a more precise vision of the task's objective. No longer than 120 words.
- **Resources**: A list of the specific resources needed for the task. It can include anything from human resources to hardware and software. In order to simplify the definition of resources, the speaker has been included in the role of the Director. List and identification of used resources defined in section 2.1.2 Resources.
- **Dependency**: A list of task IDs that must be completed before starting this same task. It must not contain loops between tasks.
- **Estimation**: The number of hours needed to complete the task. It must be a natural number (no decimals) and it is strongly recommended not to exceed 50 hours.

2.1.1 Task definition and Estimation

Given the high number of tasks, I have decided to organize some of them in four major blocks according to their objective and temporal order within the thesis (do not confuse with the typology of the task). These blocks are: Project Documentation, Project Management, Current Architecture Analysis, New Architecture Definition and Roadmap Definition and Conclusions.

As a **justification for the hours**, I would like to point out that all the tasks have been estimated taking into account the estimations of other projects I have developed, with very similar characteristics.

Project Documentation

ID: D1	**Name**: Documentation after the First Sprint	**Type**: Documentation
Description: Revision and unification of the first sprint documentation.		
Resources: R, DC, L, GD	**Dependency**: PM9	**Estimation**: 10 h

ID: D2	**Name**: Documentation after the Second Sprint	**Type**: Documentation
Description: Revision and unification of the second sprint documentation.		
Resources: R, DC, L, GD	**Dependency**: AA5	**Estimation**: 10 h

ID: D3	**Name**: Documentation after the Third Sprint	**Type**: Documentation
Description: Revision and unification of the third sprint documentation.		
Resources: R, DC, L, GD	**Dependency**: AD3	**Estimation**: 10 h

ID: D4	**Name**: Documentation after the Fourth Sprint	**Type**: Documentation
Description: Revision and unification of the fourth sprint documentation.		
Resources: R, DC, L, GD	**Dependency**: AD6	**Estimation**: 10 h

ID: D5	**Name**: Documentation after the Fifth Sprint	**Type**: Documentation
Description: Revision and unification of the fifth sprint documentation.		
Resources: R, DC, L, GD	**Dependency**: RD2	**Estimation**: 10 h

ID: D6	**Name**: Final thesis documentation	**Type**: Documentation
Description: Revision and unification of the entire thesis documentation.		
Resources: R, DC, L, GD	**Dependency**: D5	**Estimation**: 20 h

Project Management

ID: PM0	Name: Meetings	Type: Management
Description: Meetings with the director and supervisor of the thesis.		
Resources: R, D, L	Dependency: –	Estimation: 30 h

ID: PM1	Name: Context definition	Type: Documentation
Description: Documentation of the context for the thesis. Explanation of the main concepts, definition of the problems to be solved and the stakeholders of the project.		
Resources: R, DC, L	Dependency: –	Estimation: 10 h

ID: PM2	Name: Justification of the approach	Type: Documentation
Description: Documentation of the thesis justification. Statement of the previous studies on the matter and justification of the chosen approach.		
Resources: R, DC, L	Dependency: PM1	Estimation: 10 h

ID: PM3	Name: Scope definition	Type: Documentation
Description: Documentation of the scope for the thesis. Definition of the objectives and sub objectives, the list of requirements and the potential obstacles and risks of the project.		
Resources: R, DC, L	Dependency: PM2	Estimation: 15 h

ID: PM4	Name: Methodology	Type: Documentation
Description: Documentation of the methodology for the thesis. Definition of the work methodology and the monitoring tools for the project.		
Resources: R, DC, L	Dependency: PM3	Estimation: 5 h

ID: PM5	Name: Description of tasks	Type: Documentation
Description: Documentation and estimation of the thesis tasks.		
Resources: R, DC, L	Dependency: PM4	Estimation: 15 h

ID: PM6	Name: Gantt diagram	Type: Documentation
Description: Creation of the Gantt diagram for the tasks.		
Resources: R, DC, L, MSP	Dependency: PM5	Estimation: 5 h

ID: PM7	Name: Risk management	Type: Documentation
Description: Documentation of the risks mitigation and alternative plans for these obstacles.		
Resources: R, DC, L	Dependency: PM6	Estimation: 5 h

ID: PM8	Name: Budget	Type: Documentation
Description: Documentation of the budget for the project execution.		
Resources: R, DC, L	Dependency: PM7	Estimation: 20 h

ID: PM9	Name: Sustainability	Type: Documentation
Description: Documentation of the sustainability impact of the project.		
Resources: R, DC, L	Dependency: PM8	Estimation: 5 h

Current Architecture Analysis

ID: AA1	Name: Analysis of the Tech Stack	Type: Analysis
Description: Analysis of the current project's technology, software and Workspace, etc.		
Resources: R, DC, L, GK, INT	Dependency: D1	Estimation: 10 h

ID: AA2	Name: Analysis of the Architecture	Type: Analysis
Description: Analysis of the current architecture, repository, file hierarchy, patterns, dependencies, etc.		
Resources: R, DC, L, GK, INT	Dependency: AA1	Estimation: 40 h

ID: AA3	Name: Analysis of the Pipelines	Type: Analysis
Description: Analysis of the project's pipelines, its stages and its steps.		
Resources: R, DC, L, GK, INT	Dependency: AA2	Estimation: 30 h

ID: AA4	Name: Analysis of the Code	Type: Analysis
Description: Analysis of the project's code, code quality, smells, deprecated code, exceptions throw, etc.		
Resources: R, DC, L, GK, INT	Dependency: AA3	Estimation: 20 h

ID: AA5	Name: Analysis of the Jenkins Environment	Type: Analysis
Description: Analysis of the Jenkins global variables, parameters, and its environment.		
Resources: R, DC, L, GK, INT	Dependency: AA4	Estimation: 10 h

New Architecture Definition

ID: AD1	Name: Definition of the Workspace	Type: Research
Description: Definition of the IDE, plugins and some other tools to facilitate the developer's work.		
Resources: R, DC, L, GK, INT	Dependency: D2	Estimation: 20 h

ID: AD2	Name: Definition of the new Project	Type: Research
Description: Definition of the new project's repository with Gradle and definition of the new versioning methodology for the code.		
Resources: R, DC, L, GK, INT	Dependency: AD1	Estimation: 30 h

ID: AD3	Name: Definition of the new Code Architecture	Type: Research
Description: Definition of patterns, organization of the code in new classes, dependency inclusion, and definition of best practices.		
Resources: R, DC, L, GK, INT	Dependency: AD2	Estimation: 30 h

ID: AD4	Name: Definition of the new Pipelines	Type: Research
Description: Definition of the new pipeline's Stages, Steps and its exceptions.		
Resources: R, DC, L, GK, INT	Dependency: D3	Estimation: 40 h

ID: AD5	Name: Definition of the new Jenkins configuration	Type: Research
Description: Definition of the new Jenkins pipeline parameters and its global variables.		
Resources: R, DC, L, GK, INT	Dependency: AD4	Estimation: 15 h

ID: AD6	Name: Definition of Unit Testing	Type: Research
Description: Definition of the entire Unit Testing module, tests definitions and testing protocol.		
Resources: R, DC, L, GK, INT	Dependency: AD5	Estimation: 40 h

Roadmap Definition and Conclusions

ID: RD1	Name: Definition of the roadmap	Type: Development
Description: Definition of the roadmap for the architecturization of the project. Formalize the steps that the developers have to follow for the correct execution of the roadmap.		
Resources: R, DC, L, GK, INT.	Dependency: D4	Estimation: 50 h

ID: RD2	Name: Conclusions	Type: Documentation
Description: Documentation of the conclusions of the thesis and its next steps.		
Resources: R, DC, L, GK, INT	Dependency: RD1	Estimation: 40 h

ID: TDP	Name: Thesis Defense Preparation	Type: Documentation
Description: Preparation of the presentation for the defense of the thesis.		
Resources: R, DC, L, GK, INT.	Dependency: D6	Estimation: 10 h

The total estimate for the thesis is **575 hours**, as shown in the total amount of work in the summary table of the next page.

It is important to note, of course, **that this planning is only a first approximation** (carefully planned and close to the final one), **which will be adapted according to the needs of the Scrum** framework.

2.1.2 Resources

Due to the theoretical nature of the thesis, I am not going to use many resources nor very specific material other than my personal computer and my laptop, from which I will be doing all of my research. We can identify other types of resources that we will need, such as:

- ⇒ **Human Resources**, like the Researcher (R) and the thesis Director (D).
- ⇒ **Material Resources**, like the Desktop Computer (DC) and the Laptop (L), internet access and the software (including its licenses) used for the development of the thesis. MS Project PRO (MSP), Google Docs (GD), Gitkraken (GK) and IntelliJ (INT) for the moment.

2.1.3 Summary table

In the following page we can find a summary table of the tasks, organized by sprints of similar workload. It includes the total time estimation of the thesis and the time estimation of each section.

ID	Task Name	Work	Dependency	Resources
	Thesis Planning	**575 hrs**		
PM0	Meetings	30 hrs		R, D, L
	Sprint 1 - Project Management	100 hrs		
	Context and Scope	40 hrs		
PM1	Context definition	10 hrs		R, DC, L
PM2	Justification of the approach	10 hrs	PM1	R, DC, L
PM3	Scope definition	15 hrs	PM2	R, DC, L
PM4	Methodology	5 hrs	PM3	R, DC, L
	Temporal Planning	25 hrs		
PM5	Description of tasks	15 hrs	PM4	R, DC, L
PM6	Gantt diagram	5 hrs	PM5	R, DC, L, MSP
PM7	Risk management	5 hrs	PM6	R, DC, L
	Budget	25 hrs		
PM8	Budget	20 hrs	PM7	R, DC, L
PM9	Sustainability	5 hrs	PM8	R, DC, L
	Sprint Documentation 1	10 hrs		
D1	Documentation after the First Sprint	10 hrs	PM9	R, DC, L, GD
	Sprint 2 - Current Architecture Analysis	**120 hrs**		
	Current Architecture Analysis	110 hrs		
AA1	Analysis of the Tech Stack	10 hrs	D1	R, DC, L, GK, INT
AA2	Analysis of the Architecture	40 hrs	AA1	R, DC, L, GK, INT
AA3	Analysis of the Pipelines	30 hrs	AA2	R, DC, L, GK, INT
AA4	Analysis of the Code	20 hrs	AA3	R, DC, L, GK, INT
AA5	Analysis of the Jenkins Environment	10 hrs	AA4	R, DC, L, GK, INT
	Sprint Documentation 2	10 hrs		
D2	Documentation after the Second Sprint	10 hrs	AA5	R, DC, L, GD
	Sprint 3 - New Architecture Definition (Phase I)	**90 hrs**		
	New Architecture Definition	80 hrs		
AD1	Definition of the Workspace	20 hrs	D2	R, DC, L, GK, INT
AD2	Definition of the new Project	30 hrs	AD1	R, DC, L, GK, INT
AD3	Definition of the new Code Architecture	30 hrs	AD2	R, DC, L, GK, INT
	Sprint Documentation 3	10 hrs		
D3	Documentation after the Third Sprint	10 hrs	AD3	R, DC, L, GD
	Sprint 4 - New Architecture Definition (Phase II)	**105 hrs**		
	New Architecture Definition	95 hrs		
AD4	Definition of the new Pipelines	40 hrs	D3	R, DC, L, GK, INT
AD5	Definition of the new Jenkins configuration	15 hrs	AD4	R, DC, L, GK, INT
AD6	Definition of Unit Testing	40 hrs	AD5	R, DC, L, GK, INT
	Sprint Documentation 4	10 hrs		
D4	Documentation after the Fourth Sprint	10 hrs	AD6	R, DC, L, GD
	Sprint 5 - Roadmap Definition and Conclusions	**100 hrs**		
	Roadmap Definition and Conclusions	90 hrs		
RD1	Definition of the roadmap	50 hrs	D4	R, DC, L
RD2	Conclusions	40 hrs	RD1	R, DC, L
	Sprint Documentation 5	10 hrs		
D5	Documentation after the Fifth Sprint	10 hrs	RD2	R, DC, L, GD
D6	Final thesis documentation	20 hrs	D5	R, DC, L, GD
TDP	Thesis Defense Preparation	10 hrs	D6	R, DC, L

Table 1 – Summary Table and Estimation of the thesis tasks.

2.2 Gantt diagram

I have decided to **implement the task schedule sequentially,** since it is a theoretical thesis, performed by a single researcher. The Gantt diagram has been made using MS Project PRO[7]. In the Gantt summary table, only Staff resources are shown in order to simplify the information. On this same page, we can find the task table used to generate the diagram of the next page. To make it easier to read, it has been printed in A3 format and added in landscape layout into the document.

	ID	Task Name	Work	Duration	Start	Finish	Predecessors	Resources
1		**Thesis Planning**	**575 hrs**	**90,5 days**	**Mon 21/02/22**	**Mon 27/06/22**		
2		Milestones	0 hrs	70 days	Mon 21/03/22	Mon 27/06/22		
3		First delivery	0 hrs	0 days	Mon 21/03/22	Mon 21/03/22		
4		Second delivery	0 hrs	0 days	Fri 27/05/22	Fri 27/05/22		
5		Final delivery	0 hrs	0 days	Mon 20/06/22	Mon 20/06/22		
6		Thesis Defense	0 hrs	0 days	Mon 27/06/22	Mon 27/06/22		
7	**PM0**	**Meetings**	**30 hrs**	**89,04 days**	**Mon 21/02/22**	**Fri 24/06/22**		R, D
98		**Sprint 1 - Project Management**	100 hrs	16,5 days	Mon 21/02/22	Tue 15/03/22		
99		**Context and Scope**	**40 hrs**	**6,5 days**	**Mon 21/02/22**	**Tue 01/03/22**		
100	PM1	Context definition	10 hrs	10 hrs	Mon 21/02/22	Tue 22/02/22		R
101	PM2	Justification of the approach	10 hrs	10 hrs	Tue 22/02/22	Thu 24/02/22	PM1	R
102	PM3	Scope definition	15 hrs	15 hrs	Thu 24/02/22	Mon 28/02/22	PM2	R
103	PM4	Methodology	5 hrs	5 hrs	Mon 28/02/22	Tue 01/03/22	PM3	R
104		**Temporal Planning**	**25 hrs**	**4 days**	**Tue 01/03/22**	**Mon 07/03/22**		
105	PM5	Description of tasks	15 hrs	15 hrs	Tue 01/03/22	Fri 04/03/22	PM4	R
106	PM6	Gantt diagram	5 hrs	5 hrs	Fri 04/03/22	Fri 04/03/22	PM5	R
107	PM7	Risk management	5 hrs	5 hrs	Mon 07/03/22	Mon 07/03/22	PM6	R
108		**Budget**	**25 hrs**	**4,13 days**	**Mon 07/03/22**	**Fri 11/03/22**		
109	PM8	Budget	20 hrs	20 hrs	Mon 07/03/22	Fri 11/03/22	PM7	R
110	PM9	Sustainability	5 hrs	5 hrs	Fri 11/03/22	Fri 11/03/22	PM8	R
111		**Sprint Documentation 1**	**10 hrs**	**1,5 days**	**Mon 14/03/22**	**Tue 15/03/22**		
112	D1	Documentation after the First Sprint	10 hrs	10 hrs	Mon 14/03/22	Tue 15/03/22	PM9	R
113		**Sprint 2 - Current Architecture Analysis**	**120 hrs**	**20 days**	**Tue 15/03/22**	**Tue 12/04/22**		
114		**Current Architecture Analysis**	**110 hrs**	**18,13 days**	**Tue 15/03/22**	**Fri 08/04/22**		
115	AA1	Analysis of the Tech Stack	10 hrs	1,25 days	Tue 15/03/22	Thu 17/03/22	D1	R
116	AA2	Analysis of the Architecture	40 hrs	5 days	Thu 17/03/22	Fri 25/03/22	AA1	R
117	AA3	Analysis of the Pipelines	30 hrs	3,75 days	Mon 28/03/22	Fri 01/04/22	AA2	R
118	AA4	Analysis of the Code	20 hrs	2,5 days	Mon 04/04/22	Thu 07/04/22	AA3	R
119	AA5	Analysis of the Jenkins Environment	10 hrs	1,25 days	Thu 07/04/22	Fri 08/04/22	AA4	R
120		**Sprint Documentation 2**	**10 hrs**	**1,5 days**	**Mon 11/04/22**	**Tue 12/04/22**		
121	D2	Documentation after the Second Sprint	10 hrs	1,25 days	Mon 11/04/22	Tue 12/04/22	AA5	R
122		**Sprint 3 - New Architecture Definition (Phase I)**	**90 hrs**	**15 days**	**Tue 12/04/22**	**Tue 03/05/22**		
123		**New Architecture Definition**	**80 hrs**	**13,13 days**	**Tue 12/04/22**	**Fri 29/04/22**		
124	AD1	Definition of the Workspace	20 hrs	2,5 days	Tue 12/04/22	Fri 15/04/22	D2	R
125	AD2	Definition of the new Project	30 hrs	3,75 days	Mon 18/04/22	Fri 22/04/22	AD1	R
126	AD3	Definition of the new Code Architecture	30 hrs	3,75 days	Mon 25/04/22	Fri 29/04/22	AD2	R
127		**Sprint Documentation 3**	**10 hrs**	**1,5 days**	**Mon 02/05/22**	**Tue 03/05/22**		
128	D3	Documentation after the Third Sprint	10 hrs	1,25 days	Mon 02/05/22	Tue 03/05/22	AD3	R
129		**Sprint 4 - New Architecture Definition (Phase II)**	**105 hrs**	**17,63 days**	**Tue 03/05/22**	**Fri 27/05/22**		
130		**New Architecture Definition**	**95 hrs**	**15,88 days**	**Tue 03/05/22**	**Wed 25/05/22**		
131	AD4	Definition of the new Pipelines	40 hrs	5 days	Tue 03/05/22	Thu 12/05/22	D3	R
132	AD5	Definition of the new Jenkins parameters and constants	15 hrs	1,88 days	Thu 12/05/22	Mon 16/05/22	AD4	R
133	AD6	Definition of Unit Testing	40 hrs	5 days	Mon 16/05/22	Wed 25/05/22	AD5	R
134		**Sprint Documentation 4**	**10 hrs**	**1,75 days**	**Wed 25/05/22**	**Fri 27/05/22**		
135	D4	Documentation after the Fourth Sprint	10 hrs	1,25 days	Wed 25/05/22	Fri 27/05/22	AD6	R
136		**Sprint 5 - Roadmap Definition and Conclusions**	**100 hrs**	**16,38 days**	**Fri 27/05/22**	**Mon 20/06/22**		
137		**Roadmap Definition and Conclusions**	**90 hrs**	**15 days**	**Fri 27/05/22**	**Fri 17/06/22**		
138	RD1	Definition of the roadmap	50 hrs	6,25 days	Fri 27/05/22	Wed 08/06/22	D4	R
139	RD2	Conclusions	40 hrs	5 days	Wed 08/06/22	Fri 17/06/22	RD1	R
140		**Sprint Documentation 5**	**10 hrs**	**1,38 days**	**Fri 17/06/22**	**Mon 20/06/22**		
141	D5	Documentation after the Fifth Sprint	10 hrs	1,25 days	Fri 17/06/22	Mon 20/06/22	RD2	R
142	D6	Final thesis documentation	20 hrs	2,5 days	Mon 20/06/22	Fri 24/06/22	D5	R
143	TDP	Thesis Defense Preparation	10 hrs	1,25 days	Fri 24/06/22	Mon 27/06/22	D6	R

Figure 4 – Gantt Table used to generate the Gantt diagram.

[7] MS Project PRO – https://www.microsoft.com/en-us/microsoft-365/project/project-management-software

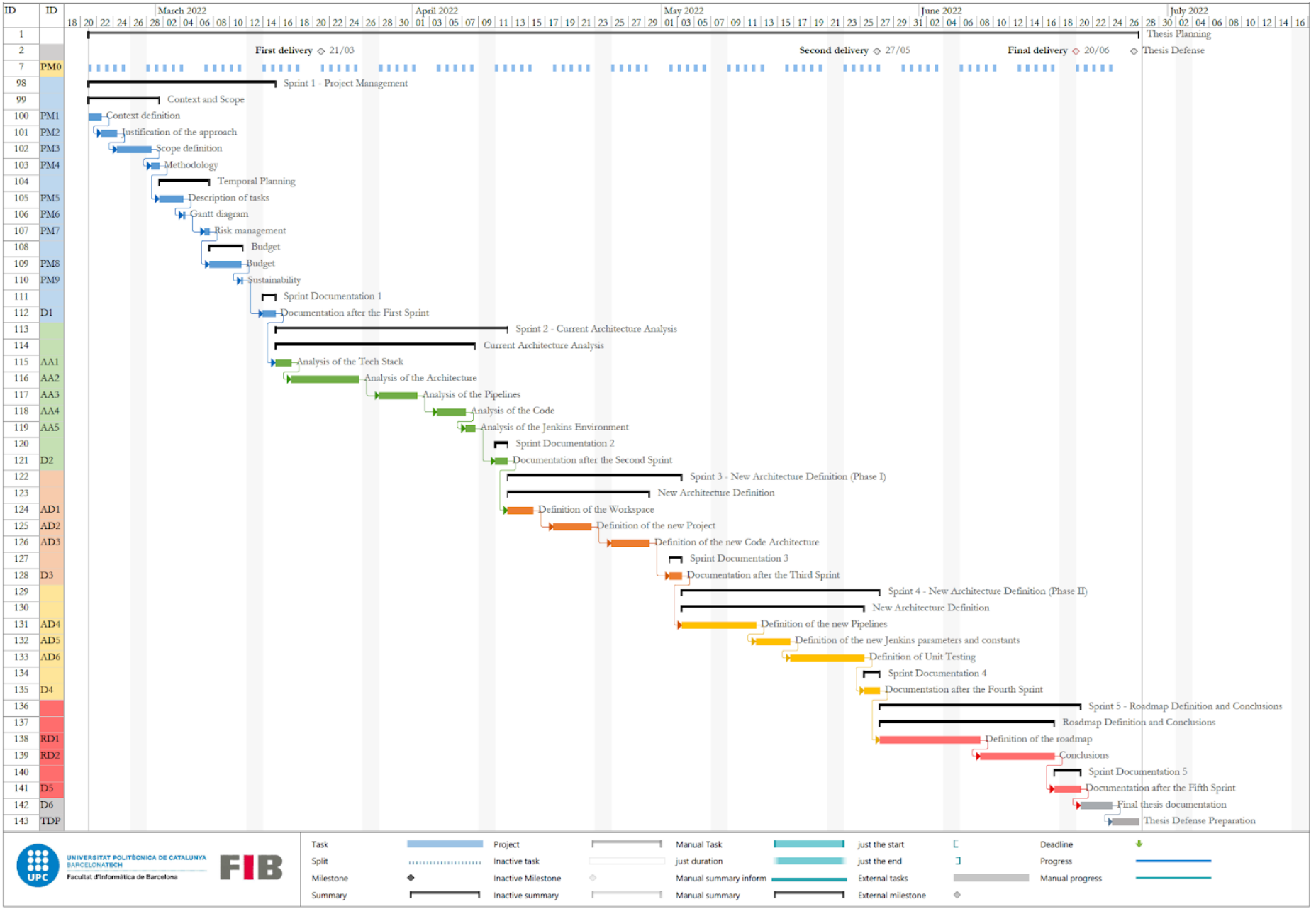


Figure 5 – Gantt diagram of the thesis.

2.3 Risk management

2.3.1 Alternative plans and obstacles

In the previous section 1.3.3 Potential obstacles and risks we detected some possible threats that could affect the development of the project. Therefore, in this section we are going to present a mitigation plan for those that are more likely to be encountered. We will try to foresee how they can impact the task schedule so that, if necessary, we can reschedule the project.

Unclear documentation

If we encounter ambiguous and poorly defined documentation, this risk will directly impact the *New Architecture Definition* phase. In the event of such a risk, there can be 3 incremental scenarios:

⇒ Low impact: (5-10 hours of deviation) The tasks have been estimated with a 10% overestimate in order to overcome delays with little impact of this kind.
⇒ Medium impact: (10-20 hours of deviation) Task AD5 must be discarded, and the 15 hours of the task will be used to correct the deviation.
⇒ High impact: (20-40 hours of deviation) A meeting with the director must be scheduled within a maximum of 3 days from the detection of the risk. At this meeting, expert assistance will be requested, and it must be decided whether the project requires a change of scope in order to be re-planned.

Complex Project in Production

Due to the high complexity of the current project code, the *Current Architecture Analysis* phase may be affected. Again, there can be 3 incremental scenarios:

⇒ Low impact: (5-10 hours of deviation) The tasks have been estimated with a 10% overestimate in order to overcome delays with little impact of this kind.
⇒ Medium impact: (10-20 hours of deviation) Task AA5 and AD5 must be discarded, and the 25 hours of the tasks will be used to correct the deviation.
⇒ High impact: (20-40 hours of deviation) A meeting with the director must be scheduled within a maximum of 3 days from the detection of the risk. At this meeting, expert assistance will be requested, and it must be decided whether the project requires a change of scope in order to be re-planned.

Tight deadlines

Thanks to the work already completed of organizing the thesis into tasks and creating a Gantt diagram with them, we have been able to mitigate this risk to a great extent, so we can consider it covered.

3 Budget

In the following section we will identify the agents that produce an economic expense in the development of the thesis, calculate the project budget and describe what we can do to control any possible deviations from the budget during its course. A proper budget estimation is essential to accurately anticipate the economic cost of the project and to ensure that our proposal does not generate large losses.

3.1 Identification of costs

In order to calculate the budget, I have separated the costs according to whether they are staff costs, material costs, generic costs, and any other costs that we should take into account. We will identify them, and then calculate its estimated cost in the next section 3.2 Cost estimates.

3.1.1 Staff cost

I opted to bring this budget as close as possible to the reality of the thesis. To do so, I have decided to calculate it taking into account only the figure of the Director (Jonattan, as Senior Software Project Manager) and the Researcher (me, as Software Architect) as the staff costs, without assuming any other role.

To determine the hourly rate for the two roles, we will use the Glassdoor[8] website, which allows us to identify the average salary (including Social Security) of all employees with the same positions. In addition, to calculate the hourly salary, we will assume a working day of 8 and a half hours, approximately 180 hours per month, with 14 payments and no bonuses.

ID	Role	Annual salary (€)	Hourly wage (€)
R	Software Architect	48,618	19.30
D	Senior Software Project Manager	70,456	27.96

Table 2 – Annual salary and Hourly wage table.

3.1.2 Material cost

Equipment Amortization cost

I will only need my personal computer and the company's laptop, from where I will do my research. No other specific hardware components are required for this project. To calculate the amortization cost of the equipment, we will use the straight-line depreciation formula (Corporate Finance Institute, 2016):

$$Annual\ Depreciation\ Expense = \frac{(Initial\ Cost\ of\ the\ Asset - Actual\ Salvage\ Value)}{Estimate\ Useful\ Life\ of\ the\ Asset\ in\ years} \quad (1)$$

[8] Glassdoor – https://www.glassdoor.com/

Given that the project has a duration of 4 months, we will multiply the annual amortization cost by 0.33 (4 months of use between the 12 months of a year).

$$Equipment\ Depreciation\ Expense = \frac{(Initial\ Cost\ of\ the\ Asset - Actual\ Salvage\ Value)}{Estimate\ Useful\ Life\ of\ the\ Asset} \times \frac{4}{12} \quad (2)$$

This next table summarizes the amortization cost of the assets and equipment using the Equipment Deprecation Expense formula (2).

ID	Equipment	Initial Cost of the Asset (€)	Actual Salvage Value (€)	Estimate Useful Life of the Asset (years)	Amortization cost (€)
DC	Mountain Desktop Computer	1,974.99	980	8	41.46
L	Lenovo ThinkPad Laptop	1,240.89	500	5	49.40

Table 3 – Equipment Amortization table.

Software Amortization cost

We must calculate as well those costs generated by the software licenses that will be used. Those that are open source and free of charge will not be included. To calculate the amortization cost of the software licenses, we will use the same straight-line depreciation formula from before but without the Salvage Value, as stated in *Accounting for Computer Software Costs* (Wehner, 2020).

Again, given that the project has a duration of 4 months, we will multiply the annual amortization cost by 0.33 (4 months of use between the 12 months of a year).

$$Software\ Depreciation\ Expense = \frac{(Total\ \ Cost\ of\ the\ License)}{Estimate\ Useful\ Life\ of\ the\ License} \times \frac{4}{12} \quad (3)$$

This next table summarizes the amortization cost of the software licenses using the Software Deprecation Expense formula (3).

ID	Software	Monthly Cost (€)	Total Cost (€)	Estimate Useful Life of the License (years)	Amortization cost (€)
MSP	MS Project 2019	29.77	119.08	1	39.69
GK	Gitkraken PRO	4.95	19.8	1	6.60
INT	IntelliJ	180.29	721.16	1	240.39

Table 4 – Software Amortization table.

3.1.3 Generic costs

Costs pertaining to electricity consumed and internet connection should also be taken into account. In our case, we will ignore the estimated office rent and travel expenses, due to Opentrends' current policy of permanent telework.

Electricity cost

We will take as a reference for the calculations the data of the last electricity bill of my house (in other words, my office). The average energy cost is 0.194326 €/kWh and the average power consumption of a desktop computer is 200Wh and 75Wh for a laptop (Energuide, 2020).

Out of the 575 hours of the project, we will assume that 60% are spent on the laptop and the remaining 40% on the desktop computer. Thus, we can use the following equations:

$$Desktop\ Electricity\ Cost = (Consumption \times Hours\ of\ use \times Energy\ cost) \times 0.4 \quad (4)$$

$$Laptop\ Electricity\ Cost = (Consumption \times Hours\ of\ use \times Energy\ cost) \times 0.6 \quad (5)$$

This next table summarizes the electricity cost using the previous formulas (4) (5).

Equipment	Consumption (kWh)	Energy cost (€/kWh)	Hours of use (h)	Electricity cost (€)
Mountain Desktop Computer	0.200	0.194326	230	8.94
Lenovo ThinkPad Laptop	0.075	0.194326	345	3.02

Table 5 – Electricity cost table.

Internet cost

We will take as a reference for the calculations the last internet bill of my house.

$$Internet\ Cost = (Internet\ monthly\ cost \times \frac{1\ month}{30\ days} \times \frac{1\ day}{24\ hours}) \times Hours\ of\ use \quad (6)$$

This next table summarizes the Internet cost using the Internet Cost formula (6).

Internet monthly cost (€)	Hours of use (h)	Internet cost (€)
29.95	575	23.92

Table 6 – Internet cost table.

3.1.4 Other costs

Contingency cost

To be able to correct errors due to incomplete information or oversights, we must add an extra contingency cost to the budget, based on a contingency rate, which in this case is set at 15% of all expenses incurred so far. This includes staff costs, material costs and the generic costs.

$$Contingency\ Cost = (Staff\ cost + Material\ costs + Generic\ costs) \times 0.15 \quad (7)$$

The total contingency expense is calculated in section 3.2 Cost estimates, following the previous formula.

Incidental cost

In addition to the tasks in the Gantt chart, budgets often include actions that are part of alternative plans designed to manage incidental events. These incidentals are not calculated at 100%, but according to a percentage that is equal to the estimated probability of the risk occurrence.

We will assume that the initial risk of "Tight deadlines" has already been neglected thanks to Gantt's planning, so we will ignore its cost in the calculation.

$$Estimated\ Risk\ Cost = Risk\ Max\ hours\ deviation \times Staff\ hourly\ wage \quad (8)$$

$$Incidental\ Cost = Estimated\ Risk\ Cost \times \frac{Probability\ of\ ocurrence}{100} \quad (9)$$

Risk	Probability of occurrence (%)	Estimated risk cost (€)	Incidental cost (€)
Unclear documentation	40	772	308.8
Complex Project in Production	30	772	231.6

Table 7 – Incidental cost table.

3.2 Cost estimates

In the following page we can find a summary table of the total budget for the thesis, including the staff costs, the material costs, the generic costs, the contingency costs and the incidental costs.

Activity	Cost (€)	Resources	Estimation (h)
PM0 - Meetings	579.00	R, L	30
PM0 - Meetings	838.80	D, L	30
PM1 - Context definition	193.00	R, DC, L	10
PM2 - Justification of the approach	193.00	R, DC, L	10
PM3 - Scope definition	289.50	R, DC, L	15
PM4 - Methodology	96.50	R, DC, L	5
PM5 - Description of tasks	289.50	R, DC, L	15
PM6 - Gantt diagram	96.50	R, DC, L, MSP	5
PM7 - Risk management	96.50	R, DC, L	5
PM8 - Budget	386.00	R, DC, L	20
PM9 - Sustainability	96.50	R, DC, L	5
D1 - Documentation after the First Sprint	193.00	R, DC, L	10
AA1 - Analysis of the Tech Stack	193.00	R, DC, L, GK, INT	10
AA2 - Analysis of the Architecture	772.00	R, DC, L, GK, INT	40
AA3 - Analysis of the Pipelines	579.00	R, DC, L, GK, INT	30
AA4 - Analysis of the Code	386.00	R, DC, L, GK, INT	20
AA5 - Analysis of the Jenkins Environment	193.00	R, DC, L, GK, INT	10
D2 - Documentation after the Second Sprint	193.00	R, DC, L	10
AD1 - Definition of the Workspace	386.00	R, DC, L, GK, INT	20
AD2 - Definition of the new Project	579.00	R, DC, L, GK, INT	30
AD3 - Definition of the new Code Architecture	579.00	R, DC, L, GK, INT	30
D3 - Documentation after the Third Sprint	193.00	R, DC, L	10
AD4 - Definition of the new Pipelines	772.00	R, DC, L, GK, INT	40
AD5 - Definition of the new Jenkins configuration	289.50	R, DC, L, GK, INT	15
AD6 - Definition of Unit Testing	772.00	R, DC, L, GK, INT	40
D4 - Documentation after the Fourth Sprint	193.00	R, DC, L	10
RD1 - Definition of the roadmap	965.00	R, DC, L	50
RD2 - Conclusions	772.00	R, DC, L	40
D5 - Documentation after the Fifth Sprint	193.00	R, DC, L	10
D6 - Final thesis documentation	386.00	R, DC, L	20
TDP - Thesis Defense Preparation	193.00	R, DC, L	10
Total DC (Direct Costs)	11,936.30		575
DC - Mountain Desktop Computer	41.46		
L - Lenovo ThinkPad Laptop	49.40		
MSP - MS Project 2019	39.69		
GK - Gitkraken PRO	6.60		
INT - IntelliJ	240.39		
Electricity Mountain Desktop Computer	8.94		
Electricity Lenovo ThinkPad Laptop	3.02		
Internet cost	23.92		
Total IC (Indirect Costs)	413.42		
Total DC + IC Cost	12,349.72		
Contingency	1,852.46		
Total DC + IC + Contingency Cost	14,202.18		
Unclear documentation	308.80		
Complex Project in Production	231.60		
Total Incidental Cost	540.40		
TOTAL Budget:	**14,742.58 €**		

Table 8 – Budget Summary Table of the thesis.

3.3 Management control

In this section, I will define the protocol that will be used to control the budget during the course of the thesis. I will also list the indicators that will help us identify any deviations in the budget, to ensure that we are not generating financial losses.

To ensure proper budgetary control, we will use the Sprint Review meetings to calculate the possible deviations, both positive (benefits) and negative (losses) of each sprint. This will help us to detect deviations in time and thus be able to approach the following sprints differently, if needed.

The protocol states that, after every Sprint Review:

1) Calculate the exact hours spent for each Sprint task.
2) Aggregate such hours and use them to calculate the staff cost deviation with the following formula:

$$H\ Spent\ Sprint\ = \sum_{i=0}^{n} Hours\ spent\ on\ task_i$$

$$Staff\ Cost\ Dev.\ = (H\ Estimated\ Sprint\ -\ H\ Spent\ Sprint) \times Role\ Hourly\ Wage$$

   a) If the Staff Cost Deviation value is positive, it means that there is no negative deviation, and, therefore, there are benefits.
   b) If the Staff Cost Deviation value is negative, it means that there have been losses of that same value in the project.

3) In both cases, if the value of the deviation is considerably large, action should be taken to solve or detect the problem of the deviation, and modify the budget estimate to bring it closer towards a more realistic number.

This same protocol can be extrapolated, with very few changes, for the rest of the project costs, although in the context of our thesis, these will not be as determinant as staff costs.

4 Sustainability

Nowadays, conducting a sustainability analysis is crucial in any project, as it is something that must not be overlooked. I strongly believe that it is our duty as engineers to work towards sustainability.

In this section we will make a brief introspection to discuss sustainability in the field of computer engineering and, to conclude, answer some questions related to economic, environmental and social dimensions of the subject.

4.1 Sustainability report

4.1.1 Self-assessment

Sustainability is a concept that everyone is more or less familiar with and, although one may think that their actions on a personal and professional level are being sustainable, they are not really aware of everything it involves. This is, for instance, my case. After the EDINSOST2-ODS survey and a short introspection on the subject, I have been able not only to understand those values and indicators of sustainability that I did not know, but also to realize how insufficiently educated we are on the subject.

I have always thought that the job of an engineer is to make people's lives easier, and to work for a better future. In this (often optimistic) approach to the professional environment of a Software Engineer, I have sometimes wondered what we can do in our day-to-day work to make our profession more sustainable.

To my pleasant surprise: Many.

Thanks to the writing and discussion of this section, I have been able to discover that it is in our hands the duty to critically evaluate if the economic viability of a project is compatible with the environmental and social aspects of sustainability. To be capable of suggesting sustainable projects or to bring new ideas and solutions to make the projects more sustainable, taking into account the environmental, economic and social aspects. And to exercise my profession in accordance with the ethical principles that support the values of sustainability, and to actively participate in responsible action in the entities where I work.

We must not forget that we are in a situation of critical Climate Emergency and, going back to the thesis main purpose, we have to change as soon as possible our mindset and methodologies when it comes to developing. Minimizing technical debt is of vital importance for a sustainable future.

4.1.2 Economic dimension

> Regarding PPP: Reflection on the cost you have estimated for the completion of the project.

The reflection on the estimated cost of the project can be found in Section 3.1 Identification of costs. We have also measured contingencies and incidental costs, and their impacts on the budget, in order to obtain a more accurate estimation and avoid an unnecessary use of resources. From my point of view, a good improvement for the project would have been to use open source software, but due to the requirements on the side of the company, I have not been able to use it in its totality.

> Regarding Useful Life: How are currently solved economic issues (costs...) related to the problem that you want to address (state of the art)?

Although it is currently a popular and studied topic, it is only put into practice in a minority of companies. Technical debt is an area closely linked to the economics of each organization. If this debt is not addressed early enough, the economic costs of maintaining an application or any code will skyrocket. Therefore, it is critical to resolve technical debt for a more sustainable economic future of the company.

> How will your solution improve economic issues (costs ...) with respect to other existing solutions?

In section 1.1 Introduction, I attached a graph that answers this question in a visual and explanatory way. The initial investment required by this solution turns out to be much smaller compared to the large amount of benefits obtained in the short term. Thanks to it, the project can evolve faster and with much higher quality, requiring smaller investments. If corrective measures are not implemented in time, there is a risk that the project will require new features at a cost of implementation that is too high due to its technical debt.

4.1.3 Environmental dimension

> Regarding PPP: Have you estimated the environmental impact of the project?

It has not been possible to make a direct estimate of the impact of the project, but the thesis is related to the removal of the technical debt, which is directly related to several sustainability criteria. On the one hand, we highly reduce the time invested on maintainability, reducing electricity costs, resources involved in the project and consequently reducing its environmental impact. On the other hand, we can contribute to justifying and raising awareness of a problem that needs to be addressed on a social level, in order to improve the common welfare.

> Regarding PPP: Did you plan to minimize its impact, for example, by reusing resources?

The project aims to reuse certain modules of the project's code, but due to its theoretical completeness, it does not require reusing many resources. One of the main objectives of the thesis is to raise awareness of the existing issue, insisting that understanding the proposed methodology from the outset would help to reduce costs of a project that has reached its inflection point in terms of technical debt.

> Regarding Useful Life: How is currently solved the problem that you want to address (state of the art)?

The current concern is that we, as a society, are not prioritizing technical debt, which leads to other environmental, socio-ecological and economic debts. What is happening is that time after time, exponentially more technical debt continues to be generated so, in the long-term, the only solution that would be implemented in this project is to redo it or to consume a lot of resources in order to maintain it.

> How will your solution improve the environment with respect to other existing solutions?

My solution aims to greatly reduce all the development and maintenance cost of a software project, thereby reducing its environmental footprint. My approach is specific to this project, but can be extrapolated to others with similar characteristics. It seeks to establish guidelines that can be followed in an understandable way in order to materialize this new methodology efficiently. The goal is to extend the life cycle of an application or software platform much longer.

4.1.4 Social dimension

> Regarding PPP: What do you think you will achieve -in terms of personal growth- from doing this project?

On a professional scale, I have been given a great deal of responsibility within the team. This opportunity has allowed me to discover a new area of technology and the importance of a good architecture before diving into programming. Furthermore, Opentrends has asked me to hold a conference for the whole company when I finish the project to make all the employees aware of the risks of technical debt and what is the best approach to avoid it.

A great lesson I am gaining is the methodological and conscientious organization of all the tasks and resources involved, which is exactly what I find essential to face for any project in the future.

> How will your solution improve the quality of life (social dimension) with respect to other existing solutions?

My proposal would significantly reduce the work of project developers, facilitating the implementation of new features. It would also greatly reduce the time spent in testing. Beyond the economic and temporary gains, the welfare of the employees will increase, they could invest more time and resources in research and investigation than in maintenance or code test execution. All this would lead to a general improvement of the company's performance in the long term.

> Regarding Useful Life: Is there a real need for the project?

On a society level, I strongly believe that a developer who is dedicated to researching and creating new things is happier and more productive than a developer who is hardly maintaining code and running tests. If this methodological vision is extended to the company in the short term, both the company and its employees will gain motivation and a new useful mindset. The project's product seeks to be an CI tool so, with an easy growth and without technical debt, future functionalities will be developed faster and deployed with a more consistent pace.

5 Current Architecture Analysis

Analyzing the current state of the project will greatly help us to define the necessary steps to follow in order to improve those aspects that may generate technical debt. It is not enough just to detect these weaknesses, but also to understand how they have been reached and what methodologies or situations to avoid in future projects.

To simplify this process, I have structured the analysis into four sections, each corresponding to the fundamental components of the project in question: The Technology used in the project, its Architecture (at a global scale), the definition of the Pipelines to be executed, and the quality of the Source Code.

Currently, as of March 16th, 2022, the MALM product is replicated on two different platforms, one of them physical, called CPD and the other one Cloud, much more up to date, called ICP. The reason these two platforms exist is that the service is in the middle of a migration process to the new infrastructure (from CPD to ICP). Considering that the migration is expected to be completed by mid-June, and that the old infrastructure will be deprecated, I have decided to discard the analysis of the current infrastructure and base my analysis on the new one.

5.1 Analysis of the Tech Stack

→ Definition

To provide more value to the analysis and to make further improvements, this first section includes a revision of the **software used by the MALM service** and a check of the **developer team's Workspace** (IDE, tools, etc).

It is important to keep in mind that this platform has been published and under constant development since 2017. By then, the bases for the development of Scripted Pipelines had not yet been established by the community, so it would not be surprising that some core technologies may now be deprecated or at risk of deprecation.

5.1.1 Workspace

After reviewing the project documentation and after a series of meetings with the development team, I have noticed that there is **no consensual development environment** among the whole team. **Each member uses a specific IDE and tools**. Among these IDEs, the most common are Visual Studio Code, Sublime, Atom and IntelliJ Community. Finally, those who share IDEs do not share versions of the same IDE nor use the same plugins.

When it comes to deciding whether or not to unify the work environment, the developer community is somewhat divided. To help me decide if unifying Workspaces is a good idea, I have gathered information from different forums and summarized it in the following lists:

It is a good idea to unify the IDEs when:

- ⇒ If peer reviews/coding is done on a regular basis. Like pair programming.
- ⇒ It is an Agile team. The development team might benefit from a common tool set.
- ⇒ Simplification is pursued by standardizing the tools of the team.
- ⇒ Team members are rapidly changing.
- ⇒ New inexperienced developers are expected to join the team.
- ⇒ The team works with simple text editors, or less complex IDEs.
- ⇒ The code needs a specific or custom plugin, created specifically for that IDE.

It is a bad idea to unify the IDEs when:

- ⇒ The team is composed of senior developers, each with expertise in using a specific IDE.
- ⇒ The project requires using different IDEs to improve performance. For example, using Eclipse for most of the work, but instead using NetBeans for its good GUI builder for the Swing components.
- ⇒ It can cause a decrease in the morale of the team.

It is important to note that the above points are only recommendations to be followed. In any case, unless there is a compelling reason to strongly unify all IDEs, always allow for flexibility, and treat the unification standard as a recommendation for the team.

5.1.2 Tech Stack

Three basic software components are required to develop and run Scripted Pipelines in Jenkins: A **Jenkins client** to run the pipelines, a **Groovy SDK** to develop and run the code, and a **Java JDK** to run the Groovy code inside Jenkins.

The Jenkins version can be extracted from the current Jenkins client page. We can find the version number in the bottom right corner of every page.

Jenkins version: **2.277.3** Release date: 2021-04-20

We can use the *Jenkins Script Console*[9] to launch the following script that retrieves the version of Groovy that is currently being used by the Jenkins nodes.

```
public String getGroovyVersion() {
    try {
        return org.codehaus.groovy.runtime.InvokerHelper.version
    }
    catch (Throwable ignore) { }
    return GroovySystem.version
}

println getGroovyVersion()
```

Code 1 – Script to retrieve Groovy version from Jenkins Script Console.

Groovy version: **2.4.12** Release date: 2017-06-24

To identify the JDK it uses, we can check the System Information[10] page of the Jenkins Client itself. In this page we can find information related to system properties, environment variables and the list of installed plugins.

Java JDK version: **1.8.0_252** Release date: 2020-04-15

The following table summarizes the versions of the core technologies currently used by the MALM service. Just for comparison, the latest released version is included.

Technology	Version	Release Date	Latest Version	Latest Release Date
Jenkins	2.277.3	2021-04-20	2.332.3	2022-05-04
Groovy SDK	2.4.12	2017-06-24	4.0.2	2022-04-23
Java JDK	1.8.0	2014-03-18	1.18.0.1.1	2022-03-22

Table 9 – Technology Analysis Summary.

[9] Jenkins Script Console – https://www.jenkins.io/doc/book/managing/script-console/
[10] Jenkins System Information – https://www.jenkins.io/doc/book/managing/system-info/

5.2 Analysis of the Architecture

→ Definition

In this section I will analyze the project architecture from a rather global point of view, addressing the use of software architecture patterns, the use of design patterns in the code and how the files are organized within the project.

First, to remind ourselves of the great importance of architecture design, we will review the benefits of having a good architecture and why it is necessary to invest the necessary time in this preliminary step before starting to code.

Apiumhub[11], a Barcelona based Tech company, has a great article about the benefits of a good Software Architecture. It does not go into very technical details, but it is very easy to understand for profiles that do not have a technological background. That is why I find it interesting, because it is also aimed at a business vision and takes into account the Technical Debt. Many of the benefits listed below are very basic, and not very specific, but they make companies hear, and more important, listen.

According to the article *15 benefits of software architecture you should know.* (Novoseltseva, 2021), having a good Architecture:

- ⇒ Creates a **solid foundation** for the software project.
- ⇒ Makes your platform **scalable**.
- ⇒ Increases **performance** of the platform.
- ⇒ **Reduces costs**, avoids code duplicity.
- ⇒ Implementing a **vision**. Looking at the architecture is an effective way to view the overall state of IT and to develop a vision of where the organization needs to or wants to go with its IT structure.
- ⇒ Identifies areas for potential **cost savings**.
- ⇒ Better **code maintainability**.
- ⇒ Enables **quicker changes**.
- ⇒ **Increases the quality** of the platform.
- ⇒ Helps **manage complexity**.
- ⇒ Higher **adaptability** on achieving new features.
- ⇒ Helps in **risk management**.
- ⇒ Reduces its **time to market**.
- ⇒ **Prioritize** conflicting Goals. It facilitates communication with stakeholders, contributing to a system that better fulfills their needs.
- ⇒ Reduces overall **technical debt**.

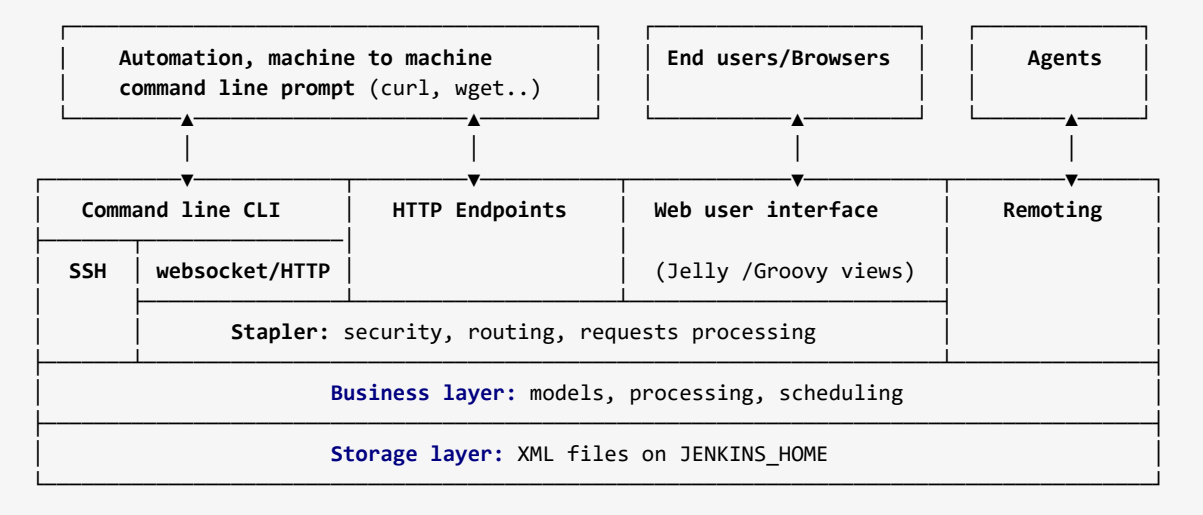
After this brief introduction, let's start with the analysis.

[11] Apiumhub website – https://apiumhub.com/

5.2.1 Software Architecture Patterns

The MALM product is in essence a **Jenkins Scripted Pipelines Controller**, programmed **in the form of a Jenkins Shared Library**, so it is not an application in its whole, but a module within the service that provides the functionality to the system.

At the Architecture level, the Jenkins service, in its entirety, follows the next model[12]:



Code 2 – High level view of Jenkins application.

Our MALM product would be in charge of implementing the business layer and managing the storage layer. With this information, we could say that the general Jenkins service does **use a Layered Pattern model** with certain modifications. Therefore, we would have to review the code to see if some of these patterns are met within the product.

For this first analysis of the Architecture, we will go through the components or "layers" on Figure 6 below, extracted from the Layered Pattern (Richards, 2015) and validate if the code complies with these principles.

⇒ Presentation Layer:
  » Is responsible for handling the user interface. It is provided by the Jenkins client service. Although it can be managed by the team, it is not developed by them.

⇒ Business Layer:
  » The first layer of the MALM Controller. It is responsible for receiving the Request from Jenkins and executing the Pipelie. This first layer could be translated to our service as the main Pipeline groovy file (Jenkinsfile) that is in charge of calling the Services to execute the Stages and Steps of the pipeline. The Business Layer is connected to the Services Layer and the Persistence Layer via calls.

[12] Jenkins Architecture Model – https://www.jenkins.io/doc/developer/architecture/model/

⇒ Services Layer (Added Layer):

» Serves as a link between the Business Layer and the Persistence Layer. In the MALM product, it is a set of classes that are responsible for forwarding the calls of these Steps to other groovy classes that contain the implementation and logic of the step. It does not seem to have a real utility within the project, considering that it does not really generate an isolation layer. It is one of the layers that we should review before the rest to confirm its removal or transformation.

⇒ Persistence Layer:

» This layer should be the only one that has access to the Database information and returns that data to the Business Layer (or the Services Layer). Here, the Persistence Layer works as a group of additional Implementation classes, which has a fairly high amount of logic in it. Jenkins does not use a Database directly (like SQL or other relational Databases) although they can be used through Plugins. Jenkins stores the data in plain text files or in Java property files on the JENKINS_HOME directory. Both of these methods (SQL and FileSystem) are used in the MALM service. This layer seems to be combined with the Business and Services Layer, breaking completely with the rules of the Layered Pattern model and the single-responsibility principle.

⇒ Database/DataSource Layer:

» It can be a Database, the File System or even an API. Gets the data, and returns this information to the upper layer. Although it can be managed by the team, it is not developed by them.

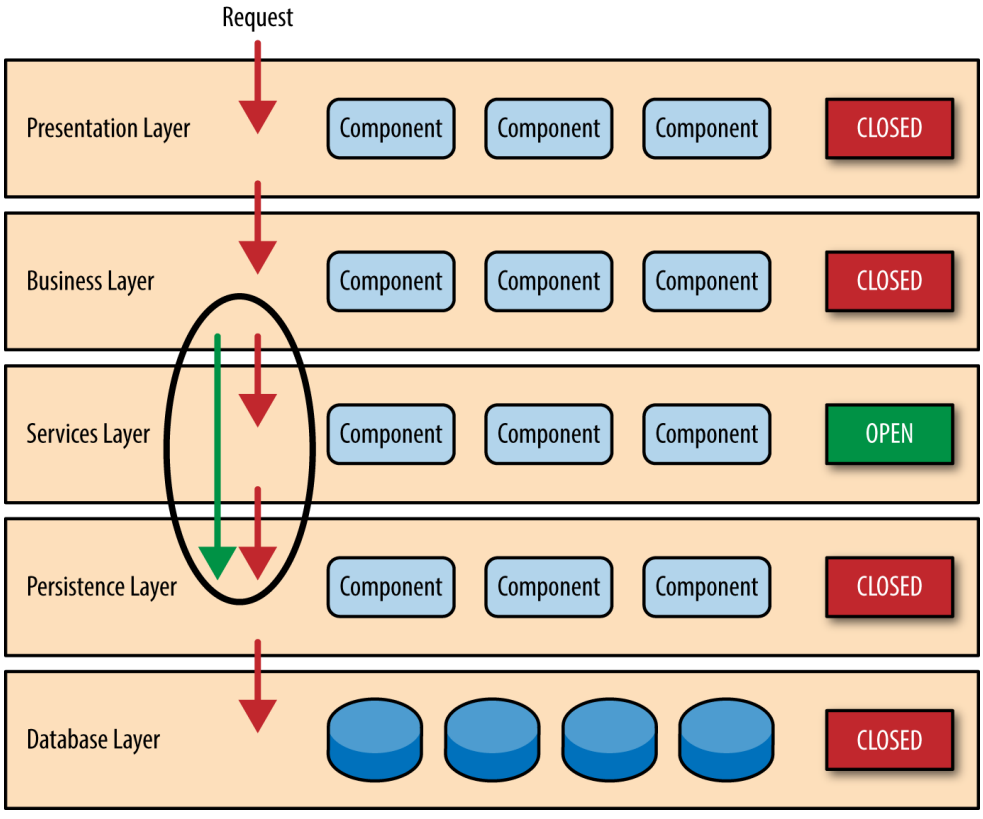


Figure 6 – Open layers and request flow on a Layered Pattern model.

5.2.2 Software Design Patterns

In order to analyze which design patterns are used on the project, it is important to know what these patterns look like and how to apply them correctly in our code. By doing so, we will be able to detect them within the code and determine if they make sense in the context of the project, if they are correctly applied or if, on the other hand, they need to be redesigned.

To refresh such knowledge on Design Patterns and to simplify this section, a summary of the design patterns (brief explanation and list of scenarios where it is recommended to use them) is attached in the section 9.1 Design Patterns of this thesis. All the information has been extracted from the book Dive Into Design Patterns by Alexander Shvets (2018).

To be able to draw useful conclusions, we must first take into account the context and nature of the service we are analyzing. The architecture and design of an application for selling clothes will not be the same as that of a service for managing patient information in a hospital. By the same logic, we must analyze the project according to its intended functionality:

⇒ Runs >4 **different pipelines, for two different technologies**: iOS_app, iOS_aar, and_app and and_aar; which may share (or not) components and classes from the MALM project.
  » The code has to be easily reusable to avoid code repetition, and isolated enough so that an error in the Android code impacts the iOS Jobs (for example).

⇒ It can run as many Jobs as the number of nodes it has configured, but each node only runs **a single pipeline at a time**.
  » There is no concurrency within a pipeline execution. A single thread of execution with a start, some defined steps, and an end, so it will rarely be necessary to create more than one instance of the classes. (This statement is only applicable in our case).

⇒ Within the Jenkins execution context, **pipelines share environment and global variables**.
  » During the execution, code can access environment variables and Jenkins parameters by calling the `env.varName`[13] variable from anywhere in the code, so it may be necessary to review this aspect carefully in order to improve it.

⇒ The MALM service must **ensure that a Pipeline can recover**[14] **if the server crashes** or experiences another kind of problem.
  » All Jenkins classes have to implement the Serializable interface in order to ensure that, after a system crash, the Job can be resumed without issues.

[13] Use of env variable on Jenkins – https://www.jenkins.io/doc/book/pipeline/jenkinsfile/#using-environment-variables
[14] More information on serializable data – https://www.jenkins.io/doc/book/pipeline/syntax/#differences-from-plain-groovy

I find it interesting to state that the MALM product, despite being highly complex in its implementation, content, and integrations with other services, does not appear to be a service that requires much complexity in terms of design patterns. It is often not necessary to implement exactly the Design patterns, but rather to adapt them to the context of the service and avoid making more complex a code that does not require an extra layer of abstraction.

Back to the subject of the analysis, I can state that, it has been difficult to analyze the use of design patterns due, ironically, to the almost non-existent use of them. Sometimes, I have given a lot of thought to certain classes, thinking about how they had been implemented in terms of design. For example, what seemed to be a Decorator pattern, turned out to be simply a class that was extended with its respective subclasses.

The only design pattern that is implemented in the code is the Singleton Pattern.

The use of it can be seen reflected in several classes of the MALM project. However, the need for it seems questionable, since it does not appear to be of any real use in the context of the execution. First, let's look at some examples:

```
1  package com.openrends.utils
2
3  class GsaServices implements Serializable {
4
5      def context
6      static def gsaServices = null
7      // Some more variable definitions
8
9      private GsaServices(context) {
10         this.context = context
11     }
12
13     public static getInstance(context){
14         if (gsaServices == null) {
15             gsaServices = new GsaServices(context)
16         }
17         return gsaServices
18     }
19
20     // Some more functions and methods
21 }
```

Code 3 – Structure of the Singleton Pattern in the GsaServices class.

The same singleton pattern structure is used for other classes, such as `MalmBuildStateHolder.groovy` or `GitServices.groovy`.

```groovy
import com.opentrends.utils.CommonUtils
import com.opentrends.utils.GsaServices
import groovy.transform.Field

@Field static def parametersMap = null

@Field static GsaServices gsaService = null

static def call(context, CommonUtils jkCommonUtils) {
    try {
        GsaServices gsaService
        gsaService = GsaServices.getInstance()
        parametersMap = gsaService.getParametersMap()
        context.println parametersMap
        // [...]
}
```

Code 4 – Use of the GsaServices Singleton Pattern on the gsa.groovy file.

The structure of the pattern is correct at first glance, it does not implement a thread validation mechanism, but since the MALM project does not use multithreading, there would be no problem other than having to implement such validation in the future (if necessary).

The instances of these classes are only used once during the execution of a Jenkins Job, and in the case of `MalmBuildStateHolder.groovy`, it is simply used as Holder for the `def buildState` variable. The latter could easily be refactored into a static global variable.

Now, putting into question the rest of the classes, these only use the `getInstance()` method once. While this ensures that there is only one instance of the class, by not calling `getInstance()` a second time, it is like there were no use of the benefits offered by the pattern, such as the creation of a single Database model object, shared by the entire application.

I have detected that the MALM project requires a class that encapsulates all the environment variables used by the pipeline and, even if it does not currently exist, it would still be debatable whether it is necessary to implement it with a singleton pattern or rather create a static data model object, which is shared within the execution of the job.

The lack of design patterns made me think about whether a project like this, a Jenkins Controller in Shared Library format, really requires a high presence of design patterns. As I said before, sometimes it is better to adapt the code to the needs of such a project, without forcing patterns or adding layers of abstraction that do not provide value to the product.

For now, to help us decide which solution is better and have further context of the system to study, let's continue with the analysis of the file hierarchy.

5.2.3 Project File Hierarchy

File Hierarchy matters.

As mentioned in the article, *File Structure: Broad Institute of MIT and Harvard. MIT Communication Lab.* (Chien, 2020), "Knowing where files are, when to use certain code for certain operations, and how to find associated results, data, and figures can not only streamline productivity, but also allow for consistency (even across multiple projects) and shareability."

There are many benefits of having the project's code well organized and structured, among them, we can highlight those that help other developers and promote healthy methodologies for the team:

- Time saved in learning the project structure.
- Employees are able to collaborate more easily.
- Quickly learning how the analysis was performed.
- Quickly reproduce the code, which adds confidence to the collaborators.

Once we know its benefits, let's see the state of the file structure of our project. The MALM project is designed for its code to be used as a Jenkins Shared Library, so its file hierarchy should be organized in a similar way.

Base File Structure for a Jenkins Shared Library[15]

```
.                           # root
├── src                     # Groovy source files
│   └── org
│       └── foo
│           └── Bar.groovy  # for org.foo.Bar class
├── vars
│   ├── foo.groovy          # for global 'foo' function file
│   └── foo.txt             # help/documentation for 'foo' file
└── resources               # resource files (external libraries only)
    └── org
        └── foo
            └── bar.json    # static helper data for org.foo.Bar
```

Code 5 – Jenkins Shared Library File Hierarchy.

MALM uses another Shared Library, which is also developed and maintained by the team, called **Pipeline-Utils**. It contains some of the repository's "Utils" modules and other auxiliary functions. The need for this Shared Library will be discussed later in section 5.3 Analysis of the Pipelines, for now we will simply highlight and discuss its file structure.

Let's now compare the usual base file hierarchy on Code 5 with the ones in our project:

[15] Shared Library Directory Structure – https://www.jenkins.io/doc/book/pipeline/shared-libraries/#directory-structure

MALM's current File Structure

```
.                                  # (root) malm-shared
├── src                            # Groovy source files
│   └── com.opentrends
│       ├── malm
│       │   ├── constants
│       │   │   ├── MalmConstants.groovy
│       │   │   └── AndroidMalmConstants.groovy
│       │   ├── exceptions
│       │   │   └── CustomBuildException.groovy
│       │   ├── Malm.groovy
│       │   ├── MalmBuildHolder.groovy
│       │   ├── MalmGitRepo.groovy
│       │   ├── MalmMergeRequest.groovy
│       │   └── MalmStageSate.groovy
│       ├── holders
│       │   ├── MalmBuildContext.groovy
│       │   ├── AndroidContext.groovy
│       │   ├── AuthLogsHolder.groovy
│       │   ├── BuildKeyParmeters.groovy
│       │   ├── CocoapodsLogsHolder.groovy
│       │   └── IOSLogsHolder.groovy
│       ├── services
│       │   ├── appcenter
│       │   │   ├── AppcenterLogsHolder.groovy
│       │   │   └── AppcenterServices.groovy
│       │   ├── auth
│       │   │   └── AuthServices.groovy
│       │   ├── buildcode
│       │   │   └── BuildCodeServices.groovy
│       │   ├── checkmarx
│       │   │   └── CheckmarxServices.groovy
│       │   ├── clean
│       │   │   └── IOSCleanServices.groovy
│       │   ├── config
│       │   │   └── MalmConfigServices.groovy
│       │   ├── credentials
│       │   │   └── CredServices.groovy
│       │   ├── email
│       │   │   └── EmailServices.groovy
│       │   ├── gradle
│       │   │   ├── GradleLogsHolder.groovy
│       │   │   ├── GradleServices.groovy
│       │   │   └── GradleUtils.groovy
│       │   ├── images
│       │   ├── node
│       │   │   └── NodeServices.groovy
│       │   ├── notification
│       │   │   └── NotificationServices.groovy
│       │   ├── prepare
│       │   │   ├── IOSPrepareServices.groovy
│       │   │   └── PrepareServices.groovy
│       │   ├── proxy
│       │   │   ├── GitLabProxyService.groovy
│       │   │   ├── ProxyIOSService.groovy
│       │   │   ├── ProxyServices.groovy
│       │   │   └── ProxyServicesLogsHolder.groovy
│       │   │
```

```
58  │   │           │   │
59  │   │           │   ├── sonar
60  │   │           │   │   ├── SonarAndroidServices.groovy
61  │   │           │   │   ├── SonarIosServices.groovy
62  │   │           │   │   └── SonarLogsHolder.groovy
63  │   │           │   ├── test
64  │   │           │   │   ├── IOSTestServices.groovy
65  │   │           │   │   └── TestReportsServices.groovy
66  │   │           │   └── xcode
67  │   │           │       └── XcodeServices.groovy
68  │   │           └── utils
69  │   │               ├── AndroidUtils.groovy
70  │   │               ├── AssembleFileServicesUtils.groovy
71  │   │               ├── GitServicesUtils.groovy
72  │   │               ├── GsaServicesUtils.groovy
73  │   │               ├── ImagesDockerControllerUtils.groovy
74  │   │               ├── IosPlistUtils.groovy
75  │   │               ├── MalmUtils.groovy
76  │   │               └── TimeUtils.groovy
77  │   │
78  │   ├── vars
79  │   │   ├── MalmStage.groovy
80  │   │   ├── MalmStageSummary.groovy
81  │   │   ├── almConfig.groovy
82  │   │   ├── and_aar_pipeline.groovy              # main CPD pipeline file
83  │   │   ├── and_app_pipeline.groovy              # main CPD pipeline file
84  │   │   ├── android_aar_pipeline.groovy          # main ICP pipeline file
85  │   │   ├── android_app_pipeline.groovy          # main ICP pipeline file
86  │   │   ├── appcenter.groovy
87  │   │   ├── appName.groovy
88  │   │   ├── appParent.groovy
89  │   │   ├── auth.groovy
90  │   │   ├── buildCodeVersion.groovy
91  │   │   ├── checkmarx.groovy
92  │   │   ├── checkValidBuildParameters.groovy
93  │   │   ├── commonUtils.groovy
94  │   │   ├── configureSignConfig.groovy
95  │   │   ├── credentials.groovy
96  │   │   ├── email.groovy
97  │   │   ├── email_notification_pipeline.groovy
98  │   │   ├── garName.groovy
99  │   │   ├── gitlabEnv.groovy
100 │   │   ├── gitUtils.groovy
101 │   │   ├── gradleUnix.groovy
102 │   │   ├── group.groovy
103 │   │   ├── gsa.groovy
104 │   │   ├── gsa_nightly_json_processing.groovy
105 │   │   ├── imagesDockerController.groovy
106 │   │   ├── InitMalmGitRepo.groovy
107 │   │   ├── InitMalmMergeRequest.groovy
108 │   │   ├── initBuildParameters.groovy
109 │   │   ├── initIosBuildParameters.groovy
110 │   │   ├── initProxy.groovy
111 │   │   ├── ios_app_pipeline.groovy              # main CPD pipeline file
112 │   │   ├── ios_distribute_tesflight_pipeline.groovy
113 │   │   ├── ios_pod_pipeline.groovy              # main CPD pipeline file
114 │   │   ├── ios_podRepoPush.groovy
115 │   │   ├── isAarq.groovy
```

```
116 │   │   ├── isAndroid.groovy
117 │   │   ├── isAppParent.groovy
118 │   │   ├── isBlf.groovy
119 │   │   ├── isCommons.groovy
120 │   │   ├── isFwk.groovy
121 │   │   ├── isIOS.groovy
122 │   │   ├── isMCA.groovy
123 │   │   ├── isPRO.groovy
124 │   │   ├── isTST.groovy
125 │   │   ├── loadBuildPaths.groovy
126 │   │   ├── mergeRequestComment.groovy
127 │   │   ├── mergeRequestId.groovy
128 │   │   ├── mergeRequestSource.groovy
129 │   │   ├── mergeRequestTarget.groovy
130 │   │   ├── mobileapp_build_dispatcher.groovy
131 │   │   ├── mobileapp_pipeline.groovy
132 │   │   ├── notification.groovy
133 │   │   ├── printChanges.groovy
134 │   │   ├── repoPath.groovy
135 │   │   ├── rubyRun.groovy
136 │   │   ├── saucelabs_nighty_pipeline.groovy
137 │   │   ├── saucelabs_test.groovy
138 │   │   ├── sonar.groovy
139 │   │   ├── soanrIOS.groovy
140 │   │   ├── sonarProject.groovy
141 │   │   ├── Stagex.groovy
142 │   │   ├── testAnalyzeReports.groovy
143 │   │   └── uploadBlf.groovy
144 │   │
145 │   ├── resources                    # resource files
146 │   │   ├── credentials
147 │   │   │   ├── pre.json
148 │   │   │   └── pro.json
149 │   │   ├── maven
150 │   │   │   └── settings.xml
151 │   │   ├── saucelabs
152 │   │   │   └── generate_saucelabs_ipas.sh
153 │   │   ├── sonar
154 │   │   │   ├── configtest.gradle
155 │   │   │   ├── jacoco-flavored.gradle
156 │   │   │   ├── jacoco-flavored-gradle6.gradle
157 │   │   │   └── sonar.gradle
158 │   │   ├── xcode
159 │   │   │   ├── delete_duplicate_simulators.rb
160 │   │   │   ├── xcode_build_paths.sh
161 │   │   │   └── xcode_derived_path.sh
162 │   │   └── new_jobs.txt
163 │   │
164 │   ├── .gitignore
165 │   ├── CHANGELOG.md
166 │   └── README.md
```

Code 6 – MALM's project File Hierarchy.

I opted to list the entire file hierarchy and not summarize any directory, in order to have a realistic and detailed view of the repository status. Due to a corporate policy, I had to change almost all the file names.

Pipeline-Utils Shared Library File Structure

```
.                                # (root) pipeline-utils
├── auth
│   ├── malm_pre.cer
│   ├── malm_pre.cnf
│   ├── malm_pre.csr
│   ├── malm_pre.key
│   ├── malm_pre.pem
│   ├── malm_pro.cer
│   ├── malm_pro.cnf
│   ├── malm_pro.csr
│   ├── malm_pro.key
│   └── malm_pro.pem
├── dependencies
│   ├── android.whitelist
│   ├── dependencyCheck.groovy
│   ├── ios.whitelist
│   └── malm-dependency-check.sh
├── img
│   └── logo_malm3.png
├── ios
│   └── testFlight.groovy
├── ios-sonar
│   ├── .gitkeep
│   ├── ios-sonar.groovy
│   └── run-sonar-swift.sh
├── nexus
│   ├── maven.app.gradle
│   └── maven.gradle
├── services
│   ├── and
│   │   ├── pre
│   │   │   └── service.json
│   │   ├── pro
│   │   │   └── service.json
│   │   ├── authorized_external_hosts_pre.json
│   │   └── authorized_external_hosts_pro.json
│   ├── internal
│   │   ├── and
│   │   │   ├── pre
│   │   │   │   ├── .gitkeep
│   │   │   │   └── service.json
│   │   │   ├── pro
│   │   │   │   ├── .gitkeep
│   │   │   │   └── service.json
│   │   │   └── .gitkeep
│   │   └── .gitkeep
│   └── ios
│       ├── pre
│       │   └── service.json
│       ├── pro
│       │   └── service.json
│       ├── authorized_external_hosts_pre.json
│       └── authorized_external_hosts_pro.json
├── xiflo
│   ├── xiflo.plist
│   └── xiflo.properties
│
```

```
58  │
59  ├── .gitignore
60  ├── appcenter.groovy
61  ├── buildcode.groovy
62  ├── CHANGELOG.md
63  ├── ci_parameters.groovy
64  ├── coverage.groovy
65  ├── dependecies_builder.groovy
66  ├── email.groovy
67  ├── gitlab_api.groovy
68  ├── http.groovy
69  ├── junit_analyzer.groovy
70  ├── login.groovy
71  ├── Playstore.groovy
72  ├── project_analyzer.groovy
73  ├── README.md
74  ├── RunUtils.groovy
75  ├── saucelabs.groovy
76  ├── send_email.groovy
77  ├── sh.groovy
78  ├── sign-config-ios.groovy
79  ├── metrics.groovy
80  ├── sqlite.groovy
81  ├── utils.groovy
82  ├── utils_rules.groovy
83  ├── xcode_bot.groovy
84  └── xml.groovy
```

Code 7 – Pipeline-Utils project File Hierarchy.

In order to draw better conclusions, I needed to perform the 5.4 Analysis of the Code in parallel with the file structure analysis. The following lists expose the observations and aspects that are object of study when it comes to the definition of the new file hierarchy of both repositories:

MALM's project

- ⇒ The main base structure (src, resources, vars) is correct, but the **files inside /src lack a proper organization.** There seems to be an attempt at taxonomic organization, but it has not been followed correctly (e.g., "utils" and "holder" files within the "services" directory). Also, **the taxonomy can be further improved.**
- ⇒ There is **no standard for file naming**. Non-specific names are used and naming conventions like camelCase, PascalCase and snake_case are mixed in elements of the same type.
- ⇒ **Android, iOS and cross files are not easily distinguishable**. In some cases an abbreviation of the technology is included in the name, but these are few cases.
- ⇒ *(After the Code Analysis)* There are **a lot of files that are currently deprecated**, and they have not been removed or cleaned from the repository. It might be interesting to implement a cleaning protocol for future deprecations.
- ⇒ The **README.md and CHANGELOG.md files** exist, but **are heavily outdated**. An update protocol should be implemented to recover these files.

Pretty much the same for the pipeline-utils repository.

Pipeline-Utils

- ⇒ It does not comply with the basic structure of a Shared Library. It lacks a useful hierarchy.
- ⇒ There is no standard for file naming.
- ⇒ Android, iOS and cross files are not easily distinguishable.
- ⇒ *(After the Code Analysis)* There are a lot of files that are currently deprecated.
- ⇒ The README.md and CHANGELOG.md files exist, but are heavily outdated.

5.3 Analysis of the Pipelines

→ Definition

5.3.1 Shared Libraries

As we have already mentioned, Jenkins uses the code of the MALM project as a Shared Library. For this, Jenkins only has to configure its library inclusion[16] on the definition of the pipeline[17] script.

```
1  /* Using just the library name */
2  @Library('my-shared-library') _
3  /* Using a version specifier, such as branch, tag, etc */
4  @Library('my-shared-library@1.0') _
5  /* Accessing multiple libraries with one statement */
6  @Library(['my-shared-library', 'otherlib@abc1234']) _
```

Code 8 – Shared Library inclusion.

Then, call the pipeline file to start the execution, like this:

```
1  @Library('malm-shared@master') _       # 'nameOfTheLibrary'@'branch'
2  android_app_pipeline()                 # pipeline file call to be executed
```

Code 9 – Pipeline Definition Script (Declarative Syntax).

It is a good way to include a Scripted Pipeline code such as MALM. Thanks to the Shared Library nature of the project, we can select the branch of the GIT repository we want to run. Furthermore, this also allows us to set up Test Pipelines, so we can run tests on development branches.

In addition to our own Shared Library, MALM uses two other Shared Libraries: **Pipeline-Utils** and **Core-Utils-Shared-Library**.

Pipeline-Utils is developed and maintained by the same MALM team, which allows us to analyze it and apply the necessary modifications to it, if that implies an improvement to the overall system.

On the other hand, the Core-Utils-Shared-Library library belongs to an internal OpenTrends team unrelated to the MALM team. It is therefore outside the scope of the code analysis, I will just analyze its usage inside the service. MALM uses this library to retrieve corporate information, such as the mailing list of the managers.

[16] Library Inclusion – https://www.jenkins.io/doc/book/pipeline/shared-libraries/#using-libraries
[17] Definition Script field – https://www.jenkins.io/doc/book/pipeline/getting-started/#through-the-classic-ui

After analyzing both libraries, we can determine that:

Pipeline-Utils

⇒ **Most of the files are deprecated.**
⇒ The content of the library does not meet the requirements expected for a Shared Library. Its **methods and functions are specific to the MALM project**, so it could not be reusable by another project.
⇒ According to the development team, the **maintenance of the Shared Library is very time-consuming**, making the development and release of new functionalities difficult.

Core-Utils-Shared-Library

⇒ This library is included directly from the file that uses it, with the same @Library call that we have seen before. This can be confusing, since it is **not easy to detect which third-party libraries are being used** by the service. It could cause problems when determining project dependencies.
⇒ The library include is configured **pointing to the develop branch**, this being one of the biggest concerns, since if the Core Utils team were to modify its development branch, it could cause errors in the execution of all client pipelines.

5.3.2 Pipelines

I will analyze the pipelines following the Best Practices of Jenkins Pipelines. Although there are four pipelines, their condition is very similar in all of them, so I will analyze them as a whole. Before doing so, I will mention a couple of aspects that are not covered in the Best Practices guide, concerning the structure of the pipeline code.

```
node {
    stage('Build') {
        # Perform steps related to "Build" stage
        step_prepareBuild()
        step_executeBuild()
    }
    stage('Test') {
        # Perform steps related to "Test" stage
        step_launchTests()
    }
    stage('Deploy') {
        # Perform steps related to "Deploy" stage
        step_deployToEndpoint1()
        step_deployToEndpoint2()
    }
}
```

Code 10 – Scripted Pipeline structure.

The MALM service uses a Pipeline structure based on Stages[18] and Steps[19], as in most Jenkins projects. A Stage is an element that conceptually groups a set of Steps within the Pipeline execution. Such a block is used by many plugins to visually show the status or progress of the execution. In the same way, a Step is a call that performs a single task. This task has to be as atomic as possible. We can see an example in *Code 7* above.

After analyzing the pipelines structure and the content of its Stages, we can notice that:

⇒ **There is no separation by Steps**, there is only consistency in the structure of Stages. We need to refactor the code inside the stages to divide it correctly by Steps.
⇒ Some **Stages have a permanent SKIP configured**, since they were deprecated and were not eliminated from the code.
⇒ Some **Stages lack a clear and explanatory name**.
⇒ **Stages can be regrouped** conceptually by functionality, to simplify the information displayed to the user.

Listed on the next pages are the progress bars of the base pipelines of the MALM service, the information is shown as the client sees it. The times shown for each Stage should not be taken as a reference, since many of the Stages have been skipped in order to speed up the analysis.

To conclude the analysis of the pipelines, I have reviewed in collaboration with the MALM team all the stages of the pipelines. We have summarized in a table what each stage executes for each pipeline. This will help us enormously to refactor the code and transform all these requirements and functionalities into correctly defined Steps.

[18] Stage – https://www.jenkins.io/doc/book/pipeline/#stage
[19] Step – https://www.jenkins.io/doc/book/pipeline/#step

android_app_pipeline

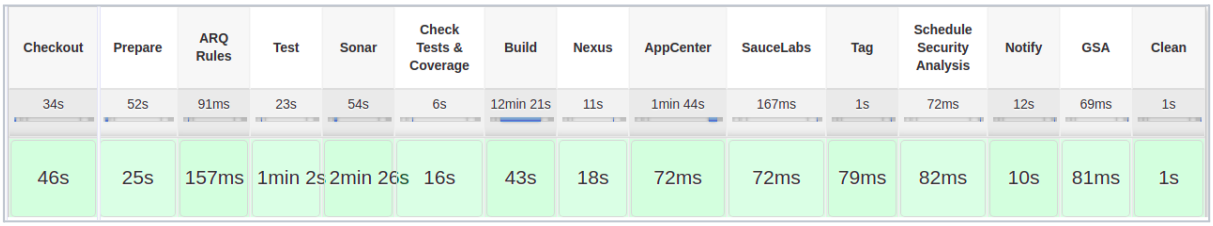


Figure 7 – Progress status bar for the android_app_pipeline execution.

Stage	Description
Checkout	» Download the project repository and the Pipeline-Utils repository. » Initializes Pipeline GitLab status.
Prepare	» The variables of the project to be built are assigned in the gradle.properties file. » If it is a "store", the gradle.properties file is renamed to "gradle_store.properties". » Signature configuration is added. » If specified by the app, the configuration variables from AuthServer are assigned for the use of the Gateway API. » Services.json, gradle.properties and xiflo.properties files are replaced. » AppCenter Validation.
ARQ Rules	*Deprecated.*
Test	Execution of the application's Unit Tests.
Sonar	Sonar analysis execution.
Check Tests & Coverage	Check if the Tests and Sonar analysis are completed without errors.
Build	Assemble Gradle to generate the package.
Nexus	Package upload to Nexus.
AppCenter	APK upload to AppCenter. "Mappings.txt" file is included if it exists.
SauceLabs	» APK upload to Sauce Labs Integration. » Scheduling of the execution of functional tests.
Tag	If it is "release", create a Git tag with the current version.
S.Security Analysis	Asynchronous programming of security analysis. Integration with Checkmarx.
Notify	Sending email with final CI status and notification to GitLab's CI status.
GSA	*Deprecated.*
Clean	» Jenkins workspace cleanup. » Notifies GitLab repository of status end.

Table 10 – android_app_pipeline Stage description Summary.

android_aar_pipeline

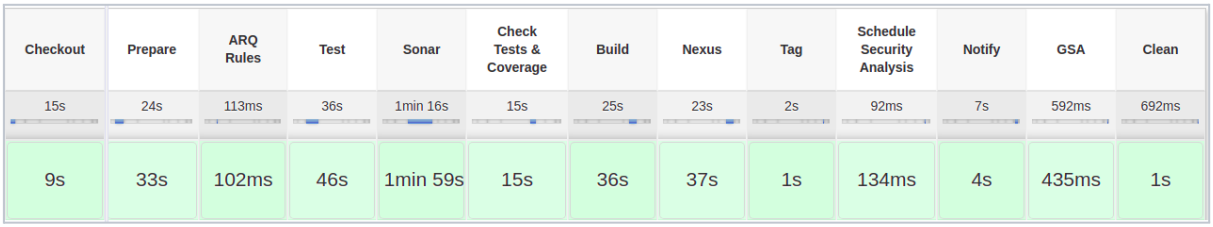


Figure 8 – Progress status bar for the android_aar_pipeline execution.

Stage	Description
Checkout	» Download the project repository. » Download Pipeline-Utils repository. » Initializes Pipeline GitLab status.
Prepare	» Manages the Stages that will be skipped. » The variables of the project to be built are assigned in the gradle.properties file. » Notification mail preparation.
ARQ Rules	*Deprecated.*
Test	Execution of the application's Unit Tests.
Sonar	Sonar analysis execution.
Check Tests & Coverage	Check if the Tests and Sonar analysis are completed without errors. Seems to be deprecated on aar.
Build	Assemble Gradle to generate the package with aar format.
Nexus	aar Package upload to Nexus.
Tag	If it is "release", create a Git tag with the current version.
Schedule Security Analysis	*Not used on aar.*
Notify	Notifies GitLab repository of status change.
GSA	*Deprecated.*
Clean	» Jenkins workspace cleanup. » Notifies GitLab repository of status end.

Table 11 – android_aar_pipeline Stage description Summary.

ios_app_pipeline

Figure 9 – Progress status bar for the ios_app_pipeline execution.

Stage	Description
Clean	Jenkins workspace previous cleanup.
Checkout	» Download the project repository. » Download Pipeline-Utils repository. » Initializes Pipeline GitLab status.
Prepare	» Parameters are adjusted in the Info.plist. » services.json and xiflo.plist files are replaced. » Call to the method for obtaining credentials assertions. » AppCenter Validation. » Run "pod update".
MALM Rules	*Deprecated*
Test	Generates the configuration and log files for Sonar. Execution of the application's Unit Tests.
Sonar	Sonar analysis execution.
Check Tests & Coverage	Check if the Tests and Sonar analysis are completed without errors. This Stage seems to be deprecated, the validation is done on the Test and Sonar Stage.
Build	Generate the IPA and dSym using Fastlane and the ExportOptions.plist (ExportOptions.store.plist for the Store version) of the project.
Nexus	Package upload to Nexus.
AppCenter	IPA upload to AppCenter. dSYM files are included if they exist.
SauceLabs	» IPA upload to Sauce Labs Integration. » Scheduling of the execution of functional tests.
TestFlight	If requested by comment in the Merge Request, schedule an upload to Testflight.
Tag	If it is "release", create a Git tag with the current version.
S.S.A.	Asynchronous programming of security analysis. Integration with Checkmarx.
Notify	Finalizes the Pipeline GitLab status and notifies the project partners via email
GSA	Sending catalog information to GSA

Table 12 – ios_app_pipeline Stage description Summary.

ios_pod_pipeline

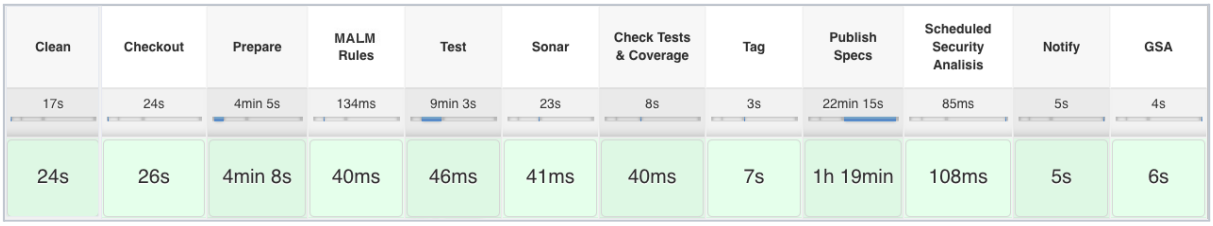


Figure 10 – Progress status bar for the ios_pod_pipeline execution.

Stage	Description
Clean	Jenkins workspace previous cleanup.
Checkout	» Download the project repository. » Download Pipeline-Utils repository. » Initializes Pipeline GitLab status.
Prepare	» Parameters are adjusted in the Info.plist. » Generate the project: pod update
MALM Rules	*Deprecated*
Test	Generates the configuration and log files for Sonar. Execution of the application's Unit Tests.
Sonar	Sonar analysis execution.
Check Tests & Coverage	Check if the Tests and Sonar analysis are completed without errors. This Stage seems to be deprecated, the validation is done on the Test and Sonar Stage.
Tag	If it is "release", create a Git tag with the current version.
Publish Specs	Publish the Pod in the Specs repository.
Schedule Security Analysis	Asynchronous programming of security analysis. Integration with Checkmarx. *Currently, the stage is disabled.*
Notify	Notifies GitLab repository of status change
GSA	Sending catalog information to GSA

Table 13 – ios_pod_pipeline Stage description Summary.

5.4 Analysis of the Code

→ Definition

Nowadays, the importance of code quality is no longer a surprise to anyone. We can find hundreds of articles and papers warning us about it, and there is an increasing number of tools available to measure code quality (SonarQube, Crucible, PVS-Studio, etc.).

> "*Any fool can write code that a computer can understand.*
> *Good programmers write code that humans can understand.*"

States Martin Fowler on its book *Refactoring: Improving the Design of Existing Code* (1999)

Back in 1999, code quality (not only focused on performance, but also on the developer's welfare) was a topic of concern among the experts. Nowadays, it is something that is strongly accepted by the community and should be internalized in developers methodologies. Unfortunately, this decision does not always depend on the developer, an external factor (well known to all of us) prevents things from being done the right way.

The problem now lies in bringing this concept, this need for quality code, to the business-corporate level.

As stated in the recently published paper, Code Red: The Business Impact of Code Quality. A Quantitative Study of 39 Proprietary Production Codebases[20] (Tornhill & Borg, 2022)*, "Code quality remains an abstract concept that fails to get traction at the business level. Consequently, software companies keep trading code quality for time-to-market and new features. The resulting technical debt is estimated to waste up to 42% of developers' time. [...] The business advantage of high quality code should be unmistakably clear."*

From the same study, we can extract the following empirical data:

- ⇒ Low quality code contains **15 times more defects** than high quality code.
- ⇒ Resolving issues in low quality code takes on average **124% more time in developmen**t.
- ⇒ Issue resolutions in low quality code involve higher **uncertainty**, manifested as **9 times longer maximum cycle times**.

Based on these arguments, in this thesis I will not further elaborate on the importance of code quality, but rather rely on its principles to analyze and improve the current code.

[20] Recommended lecture of the paper – https://arxiv.org/pdf/2203.04374.pdf

5.4.1 Groovy Style Guide

MALM is developed with Groovy which, quoting from the language official website[21]: *"Groovy is a powerful, **optionally typed** and **dynamic language**, with **static-typing** and **static compilation** capabilities. [...] It **integrates smoothly with any Java program**, and immediately delivers powerful features, including **scripting capabilities**, functional programming, [...]"*

As stated on the Jenkins website[22], Groovy is the preferred language for developing Scripted Pipelines for Jenkins, but its recommendations on the official Style Guide[23] are somewhat debatable for our necessities.

When a project grows in size and complexity, it is important that its code is indexed so that it can be more easily navigated. A complete indexing, with functions and variable information, is not possible without an IDE. More importantly, it also requires a strong variable typing (defining whether a variable is a String, Boolean, Integer […] on compile time).

```
String name = "Jhon"
Integer age = 25
Boolean isMarried = true
ArrayList<String> friends = ["Mark", "Lucy", "Karol"]
```

Code 11 – Set of basic static-typing variable definition.

```
def name = "Sergio"
def age = 25
def isMarried = false
def friends = ["Mark", "Lucy", "Karol"]

age = "twenty-five"
/* age is now a String */

friends = "${name} loves ${friends[2]}."
/* friends can be either a GString or a String */
```

Code 12 – Set of basic dynamic-typing variable definition.

From the very beginning, the project was developed using the optional (dynamic) typing that Groovy allows, so this indexing is not possible at the moment. This optional typing allows not to specify the type of the variable, using the reserved word `def`, in this way, the type of the variable is determined dynamically, at run time.

The use of this typing offers certain benefits, usually in scenarios where rapid prototyping or dynamic adaptability is important, but this does not seem to be the case for the MALM service.

[21] Apache Groovy Website – https://groovy-lang.org/
[22] Scripted Pipelines Documentation – https://www.jenkins.io/doc/book/pipeline/syntax/#scripted-pipeline
[23] Groovy Style Guide – https://groovy-lang.org/style-guide.html

5.4.2 Jenkins Pipeline Best Practices

To ensure the quality of our code, we must also take into account the Jenkins Pipeline Best Practices. In order to carry out a solid analysis, I will list these practices, extracted from the official documentation[24], and review one by one if they are complied with or if, on the contrary, it would be interesting to implement them.

General Jenkins usage

⇒ Making sure to use Groovy code in Pipelines as glue.
  » It is recommended to reduce the amount of Groovy code running in the pipeline. It is preferable not to use Groovy or other external libraries for functionalities or processes that can be converted, for example, into a more complex sh script. In the case of MALM, this first practice is not fulfilled, since there are several Steps and functionalities that are implemented purely in Groovy.

⇒ Running shell scripts in Jenkins Pipeline.
  » The approach is similar to the one discussed above, but in this case it refers specifically to the use of Shell Scrits instead of logic programmed in Groovy (whenever possible).

  This is the case for a method that takes care of replacing values of .plist files. It uses a specific library for the replacement, but due to the reads and writes that Jenkins has to do, the process usually takes between 20 and 30 minutes (25% of the entire pipeline time). Turning this Groovy method into a shell script could dramatically reduce these execution times.

⇒ Avoiding complex Groovy code in Pipelines.
  There are a couple of native Groovy methods that they recommend not to use due to the high consumption of resources in the Controller. The main idea is the same as before, reduce the workload of the Controller and transform these methods into commands to send the work to the agent. The most common methods are: `JsonSlurper` and `HttpRequest`.

  Use of `JsonSlurper`; 26 finds on the code.
  Use of `HttpRequest`; 15 finds on the code.

  Both methods are used several times by MALM code, therefore they should be converted into commands to comply with these best practices.

⇒ Avoiding calls to Jenkins.getInstance.
  » It is important to avoid severe security and performance issues. In our case, this practice is being followed.

[24] Pipeline Best Practices – https://www.jenkins.io/doc/book/pipeline/pipeline-best-practices/

⇒ Cleaning up old Jenkins builds.
  » Keeping Jenkins Controller clean of old or unwanted jobs promotes an efficient environment with good resource management. Jenkins allows you to configure a job-specific policy for deleting old builds and logs, called logRotator.

    Currently this policy is configured with rather high parameters, allowing Logs to exist in the platform 3 months after being executed. Considering the large number of Jobs that are launched daily, and the size of the logs, the LogRotator configuration could be adjusted to be less permissive.

Using Shared Libraries

⇒ Do not override built-in Pipeline steps.
  » Overwriting Jenkins' own steps like sh or timeout can have really bad results if they are not programmed correctly. It is preferable to create a custom step before overwriting an existing one. After analyzing the code, I have found no step that overwrites an existing one, so we can confirm that this practice is being followed.

⇒ Avoiding large global variable declaration files.
  » If it exists, a global variable file is loaded for each execution of a Job, whether it is used or not, so it can be an unnecessary waste of resources. Currently, no variable file is used, only the Jenkins global variables, so we could state that this practice is followed.

⇒ Avoiding very large shared libraries.
  » Like the variables file, all the files of a Shared Library are downloaded for each execution, so it is necessary for the Shared Library not to be extremely large. In the case of MALM, after several tests and checks, I do not see that it takes too long to download the entire library (~20 seconds), so we can assure that this practice is not currently a concern.

5.4.3 Code Quality

Besides the issues already discussed about the code, there are other more basic aspects that I have been noticing and that will require a review. These issues are:

⇒ There is no proper indentation of the code.

⇒ The variables do not have a defined standard nomenclature.
  » camleCase, snake_case and PascalCase are used for variables of the same kind.

⇒ Languages are mixed in the logs that will be seen by the user.
  » The final Job log shows logs in English, Spanish and Catalan.

⇒ There are many variables that are defined, but not used.

⇒ Library imports are not optimized.
  » Some of them are not used and others are used with their qualified reference in the middle of the code, repeatedly.

⇒ There are many unnecessary `def`, `public`, and semicolons.

⇒ Groovy deprecated methods such as `encode(String)` and `.instance` are used.

⇒ There are many `try-catch` functions that have an empty, unused `catch` block.

⇒ In addition to those mentioned above, there are many warnings in the code (4,529 of the 7,342 total) that require additional review.

5.4.4 Code Documentation

Code documentation is a very important practice to ensure that our code is easily maintainable and understandable for every developer. Besides giving quality to the product, it is also one of the most useful components for the development team.

In A Study of the Documentation Essential to Software Maintenance (Cozzetti B. de Souza et al., 2005), the importance of documentation in software projects is represented in data, and they state that the source code and its documentation are the most relevant components to understand a system to be maintained.

It also adds: *"Although it has always been heralded as an important aid to software development and maintenance, it is notoriously absent or out-dated in many legacy software. Agile methods have shaken a bit the traditional view of software documentation, proposing a development model that rely more on informal communication than on documentation. We explained, however, that this model does not suit software maintenance, which still has great need for documentation."*

Yes, as we have already seen, the MALM team works with an Agile methodology, but that does not mean that certain aspects of the project should not be documented. The Agile methodology does not prohibit documentation, but rather its goal is to find a middle ground between complete documentation and the transmission of information between team members.

Considering this information, after analyzing the code and its documentation, we can state that:

⇒ There is no documentation of classes or functions within the source code. Most of the existing documentation is outdated.
⇒ There is, however, a very extensive and detailed functional documentation, deployed in a corporate tool. This is similar to a user manual, where customers can access to consult all product information, details of new releases and other useful data.

6 New Architecture Definition

In this chapter, I describe the proposals and decisions that should define the future of the MALM project. It is important that these proposals are discussed, questioned and adapted to bring more benefit to the product and, above all, to the developers. Once these proposals are accepted, they should be implemented on the platform in no more than 6 months, to prevent the Technical Debt from further increasing.

I have structured the definition of the new architecture in six different sections, four of them referring to the topics of the Analysis: Definition of the new Tech Stack, Definition of the new Architecture, Definition of the new Pipelines and Definition of the new Code.

Additionally, I have added two other sections that have not been detected by the Analysis, but are necessary to meet the new requirements of the platform. These are: Definition of the new Project and Definition of Unit Tests.

For this chapter, all decisions have been backed by proof-of-concept developments, reviewed and accepted by the MALM team.

I would like to point out that the objective of this thesis is not to point out the flaws of the product, but to understand how such technical debt has been generated, and to solve it in order to bring the platform back to a good performance level in terms of cost-development. In my own experience, the need for a complete refactor or redesign is, unfortunately, quite common in the IT engineering sector, for a platform that is more than 5 years old.

6.1 Definition of the new Tech Stack

← Analysis

Based on the results of the 5.1 Analysis of the Tech Stack section, I have chosen to implement a workspace configuration protocol for the MALM team and to propose an upgrade of the technology stack to the latest possible version (making sure that is supported by the company).

6.1.1 Workspace

The context of the current MALM team's workspace is the following:

- ⇒ The MALM team is formed by:
  - » One Technical Leader.
  - » Three senior developers.
  - » Four developers.
  - » Two inexperienced developers.
- ⇒ Of these three senior profiles, two of them use IntelliJ as their IDE.
- ⇒ It is a team that works with Agile methodology.
- ⇒ They typically use pair-programming and similar practices.
- ⇒ Each week they do a Review of the developments sharing screen and code.
- ⇒ There is currently no standard IDE and the ones currently used are varied (Atom, Visual Studio Code, IntelliJ and Eclipse).

Considering the above list and making use of the results of the analysis, I have chosen to implement a standardized workspace setup guide. The installation of the IDE and its custom configuration is totally optional, but strongly recommended for the whole team.

IntelliJ Community[25] is the IDE of choice due to:

- ⇒ Being one of the pioneer IDEs in Java development.
- ⇒ Its multiple plugins and integrations with Jenkins.
- ⇒ Its potential and performance.
- ⇒ Its ease of use, adaptability and the great amount of documentation.
- ⇒ Being an open-source IDE, licensed under Apache 2.0.

This decision has been consulted, discussed and agreed upon by all members of the team.

The installation guide can be found in section 9.2 Workspace Setup Guide in the Annex.

[25] IntelliJ official website – https://www.jetbrains.com/idea/

In addition to the guide, I have also implemented a simple protocol to determine when it is most advisable to install the standardized Workspace for new or existing team members. The protocol criteria are as follows:

> Current Valid IDEs: IntelliJ, Eclipse, Visual Studio Code.
> IDEs should be researched and reviewed before being listed as valid.

⇒ **If** the member has an inexperienced developer or developer profile;
- » **If** the member has no expertise in any IDE;
  Then use IntelliJ following the Installation guide.
- » **If** the member has expertise in any valid IDE;
  Then use the valid IDE with which he/she has expertise.
- » **If** the member has expertise in an invalid IDE;
  Then use IntelliJ following the Installation guide.

⇒ **If** the member has a senior profile;
- » **If** the member has expertise in any valid IDE;
  Then use the valid IDE with which he/she has expertise.
- » **If** the member has expertise in an invalid IDE;
  Then the IDE should be reviewed;
  - » **If** it is defined as valid;
    Then use the valid IDE with which he/she has expertise.
  - » **If** it is defined as invalid;
    Then use IntelliJ following the Installation guide.

6.1.2 Tech Stack

When defining the new Technology Stack, I had to ask for support from the IT team in charge of managing the corporate software we have available at Opentrends. This team is responsible for checking each version and validating that it can be installed on the corresponding machines.

Due to a compatibility issue, the Groovy version cannot be updated yet, so it will be necessary to address this update in the future, when the problem is fixed.

After several meetings with them, an update of the Technology Stack to the following versions has been requested and scheduled for the next month:

Technology	Current Version	Release Date	New Version	Release Date
Jenkins	2.277.3	2021-04-20	2.319.2	2022-01-12
Groovy SDK	2.4.12	2017-06-24	-	-
Java JDK	1.8.0	2014-03-18	11.0.14	2022-01-18

Table 14 – Technology Analysis Update Summary.

Once updated, it is essential to review and fix any incompatibilities that may have appeared.

6.2 Definition of the new Project

One of the main requirements/objectives of the thesis is to convert the MALM code repository into a Gradle project. This provides us with a starting point in which to properly manage dependencies, implement a versioning protocol and allows us to integrate tools that generate Groovydoc-style documentation and execution of Unit Tests.

The need for the MALM project to have a "real project" look and structure is one of the most clear priorities that will serve as a basis for us to start developing further improvements to the system.

6.2.1 Gradle

Given that our code is programmed in Groovy, and it does not need to be compiled, it is quite easy to choose Gradle over Maven (which does require compiled code). Besides, Maven requires plugins to be compatible with Groovy, while Gradle already offers native compatibility with Groovy projects.

Gradle	Maven
Focused on domain-specific language (DSL) projects.	Focused on pure Java language-based software.
It uses a Groovy-based DSL for creating the project build file.	It uses Extensible Markup Language (XML) for creating project build file.
Developing applications by adding new features to them.	Developing applications in a given time limit.
It performs better than maven as it is optimized for tracking only the current running task.	It does not create local temporary files during software creation, and is hence – slower.
It avoids compilation.	It is necessary to compile.
Gradle is a newer tool, which requires users to spend some time to get used to it.	Maven is known to many users, and it is easily available.
Highly customizable as it supports a variety of IDE's.	Not that customizable compared to Gradle.
It supports software development in Java, C, C++, and Groovy.	It supports software development in Java, Scala, C#, and

Table 15 – Comparison Gradle vs Maven (Geeks for Geeks, 2022).

Having compared both tools, following the official documentation[26] and several community comparisons[27], I have selected **Gradle 7.1** as the build tool for the project.

Also, I have implemented a PoC (Proof of Concept) to validate that the use of Gradle is compatible with our project and its dependencies. Finally, after the validation of the PoC, I have created a configuration guide for the malm-shared project with Gradle 7.1 so that the development team can configure their environments.

[26] Gradle vs Maven comparison – https://gradle.org/maven-vs-gradle/

[27] Gradle vs Maven community comparison – https://www.geeksforgeeks.org/difference-between-gradle-and-maven/

Before the team members follow the steps in the setup guide, it is necessary to merge the `feature/POC_gradle_project` branch to the `origin/develop` branch of the project, so all the source code from the Gradle build tool can be integrated.

The Gradle setup guide can be found in section 9.3 Gradle 7.1 Setup Guide in the Annex.

Additionally, I have implemented the base schema of the project's build.gradle file, shown in the code below.

```
// plugins block that add 'tasks' to the build.gradle file
plugins {
    id 'maven-publish'
    id 'groovy'
}

group 'com.opentrends.malm-shared'
version '1.0-SNAPSHOT'
description 'Gradle project for Jenkins Pipeline Controller'

// Repositories block for resolving the dependencies
repositories {
    mavenCentral()
    google()
    maven { url 'https://repo.jenkins-ci.org/public/' }
    maven { url 'https://repo.jenkins-ci.org/releases/' }
    maven { url 'https://repo.spring.io/plugins-release/' }
    maven { url 'https://plugins.gradle.org/m2/' }
}

// SourceSets to comply with the Jenkins Shared Library file structure
sourceSets {
    main {
        groovy {
            srcDirs = ['src', 'vars']
        }
        resources {
            srcDirs = ['resources']
        }
    }
}

// Dependencies block for the libraries and functions on the code
dependencies {
    implementation 'org.codehaus.groovy:groovy-all:2.4.12'
    implementation 'org.jenkins-ci.main:jenkins-core:2.337'
    implementation 'com.cloudbees:groovy-cps:1.31'
    implementation 'com.cloudbees.jenkins.plugins:cloudbees-credentials:3.3'
    // [...]
    implementation 'org.connectbot.jbcrypt:jbcrypt:1.0.0'
    implementation 'io.provis:provisio-jenkins-runtime:0.1.40'
    implementation 'org.jenkins-ci.plugins:matrix-auth:1.7'
    implementation 'org.jenkins-ci.plugins.workflow:workflow-cps:2.41'
}
```

Code 13 – Base structure for the gradle.build file.

6.2.2 Versioning

Another requirement that was raised at the beginning of the thesis is to provide the project with a version control management tool. Having the project already configured with Gradle, allows us to make use of several of its tools to implement such versioning.

Although the MALM product is currently an internal corporate project used by a single team, it is expected to be eventually converted into a more general Shared Library, used by other teams within Opentrends. The versioning protocol will be more useful when this conversion is done, but for now, I think it will be interesting to implement it and to familiarize the team with it.

We will use the Semver[28] versioning scheme for our versioning protocol. Extracted from the official documentation:

X.Y.Z-pre-release+build

e.g., 2.0.4-*alpha*+001

Given a version number **MAJOR.MINOR.PATCH-pre-release+build** increment the:

⇒ 🔴 **MAJOR** version when you make incompatible API changes.
⇒ 🔵 **MINOR** version when you add functionality in backwards compatible manner.
⇒ 🟢 **PATCH** version when you make backwards compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

⇒ 🟠 **pre-release** version indicates that the version is unstable and might not satisfy the intended compatibility requirements. (optional)
⇒ 🟣 **build** version indicates the build identifier generated on the build. It can be an iterator or a timestamp. (optional)

Furthermore, I propose the implementation of a simple tagging system via Gradle for creating on-demand TAGs in the GIT repository.

```
1  group 'com.opentrends.malm-shared'
2  version '1.0-SNAPSHOT'
3  description 'Gradle project for Jenkins Pipeline Scripts'
4
5  task createReleaseTag() {
6      def tagName = "release/${version}"
7      ("git tag $tagName").execute()
8      ("git push --tags").execute()
9  }
```

Code 14 – Git tag creation task.

[28] Semver Official Website – https://semver.org/

6.3 Definition of the new Architecture

← Analysis

6.3.1 Software Architecture Patterns

After reviewing the results of the analysis, we can see that:

⇒ The use of a Layered pattern is suitable.
⇒ The layers of the pattern are unstructured and mixed.
⇒ Requires some adaptation to meet the product needs.
⇒ The aim is to simplify the code, not to make it unnecessarily complex.

With these previous points in mind, I propose to continue using a Layered Pattern model, very appropriate for the needs of MALM. In our case, we can only manage those layers that our Jenkins Controller project can handle, so the Presentation Layer and the DataSource Layer are out of the scope.

Of the 3 remaining layers, we have found that the Services Layer does not add value or benefit to the project, but rather adds an unnecessary layer of abstraction that can be easily removed.

This leaves us with a project that is composed of two layers:

⇒ Business Layer.
⇒ Persistence Layer.

The main idea is to adapt this pattern to the needs of MALM. Basically, the product receives an initial call to start one of the pipelines, then the pipeline is in charge of calling the Step files that are responsible for executing logic, launching scripts, modifying files, etc. These Steps use classes and methods specific to the project, located in the `/src/com/opentrends/impl/` directory. In the need to access DB or API requests (we will treat these two as components of the Database Layer), the Steps will have to call the “repository classes”, located at `/src/com/opentrends/repository/`, that forms the Persistence Layer.

This would imply:

⇒ A refactoring of the project code to ensure the Single Responsibility Principle of the Business Layer (pipeline files, steps, implementation, and utils) and the Persistence Layers (DB repository files and API requests).
⇒ A restructuring of the project file hierarchy to organize the files according to the needs of the new adapted pattern.
⇒ A change in the approach when developing new functionalities, which would have to be adapted to this way of organizing the code and the flow of an execution.

Although the basis of the Architecture Pattern remains the same, the above changes bring great benefits to the project by reducing the complexity of MALM, allowing a clearer execution flow and, in general, making it easier to develop new functionalities.

6.3.2 Software Design Patterns

As we have commented in the analysis, there is no use of design patterns in the project because **it does not need them**. The project in terms of Design Patterns is very simple, since its code executes Steps that are in charge of doing a single action already delimited and defined. There are no functions or classes that require a Strategy pattern or a Visitor pattern, for example.

In fact, the current use of the Singleton pattern is not appropriate, since there is no benefit in using it. So it would be interesting to avoid it and turn this pattern into a simple instantiable class like the rest of the project.

The classes are simple, there are no `interfaces`, `implements` or complex `extends` that may require the adaptation of patterns. The only implements that exist is the one that must be added to every class, the `implements Serializabe`, by own limitation and technical requirement of Jenkins. From my point of view, I believe that it should not be treated as a pattern.

Considering the previous discussion and the results of the analysis, my main proposal for this issue is to make the team aware that **the use of Design Patterns, whenever possible and beneficial, should be the priority** when developing future functionalities.

6.3.3 Project File Hierarchy

My proposed changes to improve the aspects identified by the analysis are to:

⇒ **Define a naming standard** for file names.
⇒ Detect and **clean the project of deprecated files**.
⇒ **Define a new file hierarchy**, based on the architecture changes shown in section 6.3.1 Software Architecture Patterns.
⇒ Implement a **protocol to update README.md & CHANGELOG.md** after the releases.

File naming standard

I decided to use the Java naming Convention[29], since Groovy is a language totally focused on Java platforms and, therefore, it seems logical to use the same naming standard. Besides, it is a choice strongly supported by the Groovy community.

The naming convention for Java is as follows:

Type	Naming Rules	Examples

Class	- It should start with the uppercase letter.
	- It should be a noun such as Color, Button, Thread, etc.
	- Use appropriate words, instead of acronyms.
		`public class Employee {`
		`//code snippet`
		`}`

Method	- It should start with a lowercase letter.
	- It should be a verb such as `main()`, `print()`, `println()`.
	- If the name contains multiple words, use camelCase.
		`void draw() {`
		`//code snippet`
		`}`

Interface	- It should start with the uppercase letter.
	- It should be an adjective such as Runnable, Remote.
	- If the name contains multiple words, use PascalCase.
		`interface Printable {`
		`//code snippet`
		`}`

Variable	- It should start with a lowercase letter, such as id, name.
	- It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
	- If the name contains multiple words, use camelCase.
	- Avoid using one-character variables such as x, y, z.
		`int id;`
		
		`String nameEmployee;`

Package	- It should be a lowercase letter, such as java, lang. - If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.	`package com.javatpoint;`
Constant	- It should be in uppercase letters such as RED, YELLOW. - If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY. - It may contain digits, but not as the first letter.	`static final int MIN_AGE = 7;`

Table 16 – Java Naming Convention. (JavaTPoint, 2011-2021)

[29] Java Naming Conventions – https://www.javatpoint.com/java-naming-conventions

However, I will add a small change in this convention to adapt it to the needs of the Jenkins project. This change is directly focused on the problem of having mixed Android, iOS and Cross files (these being indistinguishable at a glance), and the Jenkins design limitation of not being able to create directories inside the `/vars` folder.

This limitation forces us to implement a visual separation by filename for the files inside `/vars`.

All Steps files must follow this format:

tecnology_STEP_***descriptiveNameOfTheStep***.groovy

⇒ The *technology* field can be:
- » and - for Android
- » ios - for iOS
- » cross - for Android and iOS

Also, **the filename of the Pipelines must remain unchanged** for the moment.

By adding these **exceptions** to the Java Naming Convention, and adapting it a bit to comply with Groovy syntax (e.g., deleting the semicolons), we now have our **MALM Naming Convention** defined.

Deprecated files

It was necessary to deeply analyze the code to detect which files were deprecated and target them for removal.

The results of the code deprecation analysis are that, of the 228 files that compose the malm-shared project, at least **84 of them are deprecated** and could be deleted without impacting the service. This means that almost **37% of the files** in the project were of no use to the development team and had a negative impact on the team's productivity.

Before creating the final list of files to be deleted, it is important that the MALM team reviews the analysis and evaluates whether to delete the selected files or if more should be added to the list. Currently, the deprecated files are marked in the code with a comment tag `@Deprecated` at the start of the file.

New File Hierarchy

In the following page we can find the proposal for the new file hierarchy. In it, the files are divided by Layered Pattern responsibility layer, class logic taxonomy and technology while maintaining the base structure of a Jenkins Shared Library.

In addition, the Unit Testing directories defined in section 6.6 Definition of Unit Testing are included.

```
.                                   # (root) Malm-Shared
├── src                             # Groovy source files
    ├── com.opentrends
        ├── impl                    # Implementation files (Business Layer)
            ├── ios
                ├── sonar
                    └── SonarIosImpl.groovy
              [...]
                └── proxy
                    └── ProxyIosImpl.groovy
            ├── android
                ├── sonar
                    └── SonarAndroidImpl.groovy
              [...]
                └── proxy
                    └── ProxyAndroidImpl.groovy
            └── cross
                ├── notification
                    └── NotificationImpl.groovy
              [...]
                └── saucelabs
                    └── SaucelabsImpl.groovy

        ├── repository              # Repository files (Persistence Layer)
            ├── ios
                ├── sonar
                    └── SonarIosRepository.groovy
              [...]
                └── proxy
                    └── ProxyIosRepository.groovy
            ├── android
                ├── sonar
                    └── SonarAndroidRepository.groovy
              [...]
                └── proxy
                    └── ProxyAndroidRepository.groovy
            └── cross
                ├── notification
                    └── NotificationRepository.groovy
              [...]
                └── saucelabs
                    └── SaucelabsRepository.groovy
      [...]
            └── utils
                ├── AndroidUtils.groovy
                ├── AssembleFileUtils.groovy
             [...]
                ├── MalmUtils.groovy
                └── TimeUtils.groovy
        └── test                    # test directory files for Unit Testing
            ├── groovy
                ├── AppcenterLogsHolder.groovy
              [...]
                └── XcodeServices.groovy
            └── resources
                ├── AndroidUtils.groovy
              [...]
                └── TimeUtils.groovy

```

```
59  │
60  ├── vars                        # Jenkins steps and pipeline files
61  │     ├── android_aar_pipeline.groovy              # PIPELINES files
62  │     ├── android_app_pipeline.groovy
63  │     ├── ios_app_pipeline.groovy
64  │     ├── ios_pod_pipeline.groovy
65  │     │
66  │     ├── and_STEP_CheckCoverage.groovy            # STEPS files
67  │     ├── and_STEP_ExecuteSaucelabAction.groovy
68  │   [...]
69  │     ├── ios_STEP_CheckCoverage.groovy
70  │     ├── ios_STEP_ExecuteSaucelabAction.groovy
71  │   [...]
72  │     │
73  │     └── ios_STEP_uploadPackageToNexus.groovy
74  │
75  ├── resources                   # resource files
76  │     ├── exampledir
77  │     │     ├── ruby_script.rb
78  │     │   [...]
79  │     │     └── shellscript.sh
80  │   [...]
81  │     └── textfile.txt
82  │
83  ├── .gitignore
84  ├── build.gradle                # Gradle files
85  ├── gradle.properties
86  ├── gradlew
87  ├── gradlew.bat
88  ├── settings.gradle
89  ├── CHANGELOG.md
90  └── README.md
```

Code 15 – MALM's project new File Hierarchy.

Changelog.md and Readme.md updating protocol

The proposal is to apply a small change in the internal methodology of the project. Currently, the team maintains a very good functional documentation and a changelog properly updated to date, so it will not require much effort to apply the new methodology.

The idea is that, after each Release of the product, besides updating the documentation and the changelog in the external corporate tool, this information will also be added to the CHANGELOG.md files and in README.md.

6.4 Definition of the new Pipelines

← Analysis

6.4.1 Shared Libraries Unification

Considering the results of the analysis and knowing that:

⇒ Pipeline-Utils does not have a valid Shared Library structure and usage.
⇒ Pipeline-Utils complicates the development and maintenance of the code.
⇒ Part of Pipeline-Utils code is deprecated.
⇒ Pipeline-Utils is not reusable as a library for other projects.

I suggest that the two libraries maintained by the MALM team, **malm-shared and pipeline-utils, should be unified into the new malm-shared project**.

Such unification requires quite a lot of refactoring work. Currently, each groovy class in the `/vars` directory of Pipeline-Utils is used in malm-shared as a Utils class by downloading the code using GIT commands and loading the class into a variable.

```
1  // Load Utils config
2  dir("MALMUtils") {
3      def repo_url = "https://git.ot.com/malm-alm/pipelines-utils"
4
5      sh "git clone --depth 1 --branch master ${repo_url} ."
6      sh "git --no-pager log --pretty=raw"
7
8      // Load each file from the library
9      CI_XCODE_BOT = load("xcode_bot.groovy")
10     CI_API_GITLAB = load("gitlab_api.groovy")
11     CI_APPCENTER = load("appcenter.groovy")
12
13     // more pipeline-utils loads [...]
14
15     CI_BUILDCODE = load("buildcode.groovy")
16     CI_JUNIT = load("junit_analyser.groovy")
17     CI_SQLITE = load("sqlite.groovy")
18     malmBuildContext.CI_PARAMS = load("ci_parameters.groovy")
19     malmBuildContext.CI_MAIL = load("email.groovy")
20     malmBuildContext.CI_DEPENDENCIES = load("dependencies_builder.groovy")
21 }
```

Code 16 – Initialization of Utils variables with the Pipeline-Utils library.

One of the first changes the team should apply in order to start the unification is to add the files from the `/vars` directory of Pipeline-Utils to the `/src/com/opentrends/utils` directory of the Malm-Shared project. Then, in combination with an initial refactor, start to class-format those files and replace the `loads()` with class instances, as in the following example:

```groovy
// Load Utils config
private def loadUtils() {
    malmBuildContext.jkPipelineUtils = new PipelineUtils()
    malmBuildContext.jkGitlabApi = new GitlabApi()
    malmBuildContext.jkAppcenter = new Appcenter()

    // more class instances [...]

    malmBuildContext.jkBuildcode = new Buildcode()
    malmBuildContext.jkJunit = new Junit()
    malmBuildContext.jkSqlite = new Sqlite()
    malmBuildContext.jkParameters = new PipelineParameters()
    malmBuildContext.jkEmail = new Email()
    malmBuildContext.jkDependencies = new Dependencies()

}
```

Code 17 – Initialization of Utils variables with unified Malm-Shared.

In a separate branch, I have implemented a unification PoC of the Pipeline-Utils `/vars/utils.groovy` file and successfully migrated and refactored it to Malm-Shared's `/src/com/opentrends/utils/PipelineUtils.groovy` class. So the rest of the unification should not cause too many unexpected issues besides the possible bugs that may appear.

Once the unification is completed, all regression tests currently defined must be launched manually and reviewed one by one for possible bugs.

6.4.2 New Pipelines Steps

As seen in the Analysis, if we intend to reduce the costs of maintainability of the project and improve the development time of new features, one of the most important issues to address is the division by Steps of the existing Stages. For this, I propose a mid-long term refactoring (due to the high workload involved) that consists of converting all the Stages code to Steps inside groovy files in the /vars directory.

In conjunction with the results of the analysis of the Pipelines, the help of the MALM team and the existing documentation of the current project, we have been able to define the list of actions or Steps that are executed for each Stage.

Therefore, I will list, for each of the four pipelines, the Stages and Steps that form them. The purpose of this is to facilitate the developers' work when it comes to refactoring the project's code and atomizing the Stages.

To simplify the section, I will only show the list of Steps of one of the pipelines.
The rest of them can be found in section 9.4 New Pipeline Steps list of the Annex.

android_app_pipeline

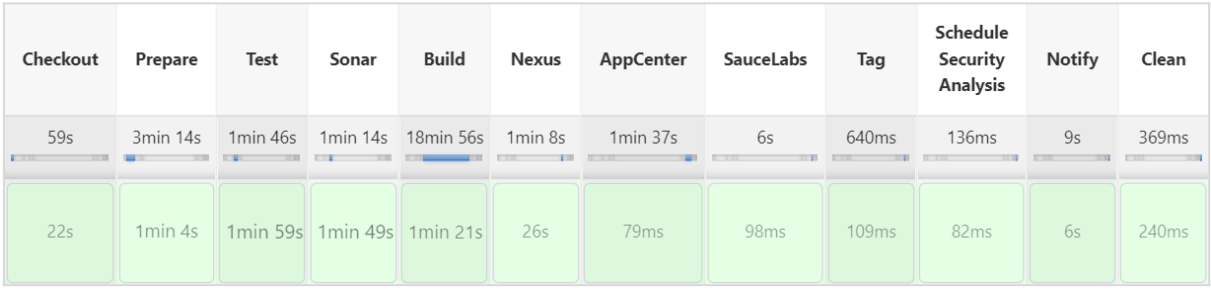


Figure 11 – New progress status bar for the android_app_pipeline execution.

Stage	Step	Description
Checkout		
	1	Project checkout and download.
	2	Loading of libraries.
	3	Initialization of the loaded libraries.
	4	Clone of the parent repository.
	5	Configure Job parameters.
	6	Signature configuration.
	7	Loading and initialization of the project configuration.
	8	AppCenter initialization.
	9	Notification of the start of the job execution in the MALM Jenkins.
Prepare		
	1	Obtaining the configuration file from the gradle.properties file.
	2	Configuration and generation of the properties of the gradle.properties file.
	3	Obtaining the dependencies implemented in the project to include them in the notification mail.
	4	Configuring the AppParent signature.
	5	Obtaining the names of the PRO nodes of the Client servers.
	6	Manage and replace the services.json file for service consumption.
	7	Replacement of Actors values for API/Gateway.
	8	Validation of dependency layout with aarqMALM.
	9	Sonar configuration.
	10	Obtaining application ID.
	11	AppCenter validation.
ARQ Rules		STAGE TO DELETE ON THE FIRST REFACTOR
Test		
	1	Test execution and report generation.
	2	Validation of test reports.
	3	Inclusion of test results in the notification email.

Sonar		
	1	Access to the Sonar profile.
	2	Checking the build variant of the project.
	3	Configuring the properties of the gradle.properties file.
	4	Sonar execution.
	5	Inclusion of the Sonar URL of the project in the notification email.
	6	Obtaining and validating metrics from Sonar.
	7	Print metrics in log.
	8	Retrieval of the coverage value from Sonar.
	9	Checking the % coverage of the project in Sonar.
	10	Inclusion of the project's coverage data and test results in the notification email.
C.T & Coverage		STAGE TO DELETE ON THE FIRST REFACTOR
Build		
	1	Checking the type of assembly.
	2	Compilation of the project.
	3	Upload Bundle to PlayStore.
	4	Upload mapping file to Splunk.
Nexus		
	1	Verification of parameters.
	2	Access to Nexus corporate for project uploading.
	3	Modification of the Gradle file.
	4	Generation of the resources file.
	5	Generation of the pom file.
	6	Preparation of the package (apk/aab) and upload to Nexus.
	7	Inclusion of the Nexus URL in the notification email.
AppCenter		
	1	Getting data from the assemble.properties file.
	2	Checking the distribution group.
	3	Obtaining the remaining parameters.
	4	Checking the mapping file.
	5	Uploading the application.
	6	Distribute the application to the distribution groups.
	7	Checking the Store version.
	8	Inclusion of AppCenter URL.
	9	Sending additional information to the GitLab component about the successful completion of the process.

SauceLabs		
	1	Checking the SauceLabs activation parameters
	2	Load SauceLabs configuration from the CI
	3	Initializing SauceLabs execution
	4	Upload and run SauceLabs
	5	Inclusion of SauceLabs run information in the notification email
Tag		
	1	Obtaining the version to tag.
	2	Checking project parameters.
	3	Access to GitLab.
	4	Creating the tag.
S. Security Analysis		
	1	Calculation of a random value for use as a delay.
	2	Initialization of Checkmarx service values.
	3	Execution of the Checkmarx analysis.
Notify		
	1	Obtaining the status of the build
	2	Obtaining the required parameters
	3	Adding missing information to the notification email
	4	Configure email recipients
	5	Attach files
	6	Send the notification email
GSA		STAGE TO DELETE ON THE FIRST REFACTOR
Clean		
	1	Deleting project data and project folders

Table 17 – New Steps summary for the android_app_pipeline.

6.5 Definition of the new Code

← Analysis

6.5.1 Best Practices

We will follow the Jenkins Best Practices, but not all the Groovy ones. The MALM team is interested in using Groovy for its code, but I think it is important not to lose the benefits of static typing of variables and objects (additional information, indexing, early bug detection, etc.).

This is not an uncommon case. As stated on the official Groovy Style Guide, on section 21.Optional typing advice[30]: *"… whenever the code you are writing is going to be used by others as a public API, you should always favor the use of strong typing, it avoids possible passed arguments type mistakes, gives better documentation, and also helps the IDE with code completion. Whenever the code is for your use only, like private methods, or when the IDE can easily infer the type, then you are more free to decide when to type or not."*

Considering the above, I propose that typing all variables should be required.

Regarding the Jenkins best practices, I will now define an action plan, for the MALM team to follow, for each practice that is not being complied with as detected in the analysis section.

⇒ Making sure to use Groovy code in Pipelines as glue.
  » It is necessary to perform a code review of the whole project, detect those functional blocks that can be optimized and refactor those blocks trying to reduce the logic programmed and executed in Groovy.

⇒ Running shell scripts in Jenkins Pipeline.
  » The approach is similar to the one discussed above. It is necessary to apply the same refactor, but for functional blocks that can be migrated to a shellscript. From my point of view, solving this should be the priority, as it could reduce the execution time by 20-25% of the total time.

⇒ Cleaning up old Jenkins builds.
  » Taking into account the amount of logs that are generated, I propose to configure the LogRotator so that the logs of each Job are deleted after 30 days of its execution, or if more than 300 jobs of the same pipeline have been executed (FIFO).

    This cleaning policy should be reviewed and adapted in the future to meet the growth of the platform and its needs.

[30] Groovy Style Guide – https://groovy-lang.org/style-guide.html

⇒ Avoiding complex Groovy code in Pipelines.

» Both of the methods mentioned in the analysis can be covered in simple commands. Quoting from the official documentation:

- `JsonSlurper:` Instead of using JsonSlurper, use a shell step and return the standard out. This shell script would look something like this:

```
String JsonReturn = sh(label: '', returnStdout: true,
    script: 'echo "$LOCAL_FILE"| jq "$PARSING_QUERY"')
```

Code 18 – Example of shell script equivalent to JsonSlurper.

- `HttpRequest:` Use a shell step to perform the HTTP request from the agent, for example using a tool like `curl` or `wget`, as appropriate.

```
String url = "http://localhost:8080/api/v3/path/json"
String response = sh(script: "curl -u userName -s $url",
                     returnStdout: true).trim()
```

Code 19 – Example of shell script equivalent to HttpRequest.

6.5.2 Code Quality

The list of code quality improvements detected in the analysis has to be reviewed and corrected. These changes do not require a high technical profile, but they may take some time to complete.

⇒ No proper indentation of the code.
⇒ Variables and Methods do not have a defined standard nomenclature.
⇒ Mixed languages in the logs.
⇒ Many variables that are defined, but not used.
⇒ Library imports are not optimized.
⇒ Unnecessary `def`, `public`, and semicolons.
⇒ Groovy deprecated methods.
⇒ `try-catch` functions that have an empty `catch` block.
⇒ Many warnings in the code (4,529 of the 7,342 total) that require additional review.

Once these refactors are completed, it will be necessary to raise awareness among MALM team members about the best practices and style guides. I also propose to implement a Code Analysis check before each Code Review in order to detect possible improvements in code quality.

6.5.3 Documentation

In the analysis, we have seen that the MALM project has a very good and extensive functional documentation, but it lacks specific documentation for programmers within the code. Therefore, I propose to document all the classes and functions of the project. Additionally, I will implement a Gradle integration so that the project documentation can be generated as a Javadoc-style HTML page.

Here are some examples how to document the methods using the Javadoc preferred syntax:

```
/**
 * Registers the global variables and Build Parameters
 * of the Job Execution
 *
 * @param context  the execution context class that called the function
 *                 to have access to the LOG print functionality.
 * @param env  the Jenkins Environment global access variable to
 *             get all the information.
 */
 static void setEnvironmentVars(context, env) {
     CONTEXT = context
     REPO_NAME = env.repo_name
     BRANCH_RESOURCE = env.branch_resource
     EXTRA_PARAMS = env.extra_params
     // [...]
}
```

Code 20 – Javadoc comment block on a setter function.

```
/**
 * Get the proxy that should be used to reach GitLab.
 * It is the same proxy for any communication with the Client network.
 *
 * @param isProxyRequired  indicates if the current Build needs a
 *                         proxy to communicate with the network or not
 * @return the proxy URL as a String
 */
String gitlabProxyAsString(Boolean isProxyRequired) {
    def https_proxy = ""
    // [...]
    return https_proxy
}
```

Code 21 – Javadoc comment block on a custom getter function.

All the information regarding the Javadoc tags and syntax guide can be found on the official Javadoc website[31].

[31] Javadoc Official Documentation – https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

In order to provide our project with a module that allows us to automatically generate HTML API documentation for the MALM project, I propose to use the Gradle **Groovydoc** task.

We will add to the build.gradle file the `groovydoc` task to execute the generation of the documentation.

```
// Customize groovydoc task that is
// added by the Groovy plugin.
groovydoc {
    // Set document title.
    docTitle = "Malm-Shared GroovyDoc"

    // Set window title.
    windowTitle = "PoC for GroovyDoc"

    // The directory to generate the documentation into.
    destinationDir = "src/groovydoc"
    // Set custom header.
    header = '''\
        <h2>Malm-Shared</h2>
        '''.stripIndent()

    // Set custom footer for generated documentation.
    footer = """\
        <div class="custom-footer">
            Generated on: ${new Date().format('yyyy-MM-dd HH:mm:ss')}
        </div>""".stripIndent()
}
```

Code 22 – Groovydoc additions to the build.gradle file for the Malm-Shared project.

The documentation for the `groovydoc` execution task can be found on the official Gradle web site[32].

[32] Groovydoc Gradle Documentation – https://docs.gradle.org/current/dsl/org.gradle.api.tasks.javadoc.Groovydoc.html

6.6 Definition of Unit Testing

6.6.1 Unit Testing

In order to provide our project with a module that allows us to launch unit tests, I propose to use the **Gradle JUnit tool**, in conjunction with the **JenkinsPipelineUnit** module that allows us to develop such tests.

We will need to add to the build.gradle file the dependencies and `test` tasks for the test module, and also set up a new `sourceSet` for the test directory:

```
plugins {
    id 'maven-publish'
    id 'groovy'
    id 'java' // adds 'test' task
}

sourceSets {
    main {
        groovy {
            srcDirs = ['src', 'vars']
        }
        resources {
            srcDirs = ['resources']
        }
    }
    // add to the existing sourceSets the tests directory
    test {
        groovy {
            srcDirs = ['src/test/groovy']
        }
    }
}

// add the Gradle task to launch the tests from the test directory
test {
    // test launch configuration options

    // Fail the 'test' task on the first test failure
    failFast = true

    // explicitly include or exclude tests
    include 'src/test/**'
    exclude 'src/test/groovy/excludedTests.groovy'
}

dependencies {
    implementation 'org.codehaus.groovy:groovy-all:2.4.12'
    // [...]
    // add the tests module dependencies
    implementation 'junit:junit:4.13.2'
    implementation 'com.lesfurets:jenkins-pipeline-unit:1.9'
}
```

Code 23 – Unit Testing additions to the build.gradle file for the Malm-Shared project.

The documentation for the test execution task can be found on the official Gradle web site[33].

My approach for such tests is to develop and launch only STEPS unit tests, as it facilitates the setup and creation of mockup variables. Launching a Test of the whole Pipeline would involve creating very large data models to be able to mockup all the variables and, in addition, this execution would be equivalent to running a job in pre-production environments, so we would not really get much benefit.

Tests will be created as Groovy class files in the `/src/test/groovy` directory. The resources for the tests (files, images or data models) will be in the `/src/test/resources` directory. I have implemented (following the guidelines of **JenkinsPipelineUnit**[34] framework) a base schema for all the tests that will help the MALM team to develop the unit tests, shown below on Code X.

```groovy
import com.lesfurets.jenkins.unit.BasePipelineTest
import org.junit.Before
import org.junit.Test

class TestExampleJob extends BasePipelineTest {

    @Before
    void setUp() throws Exception {
        super.setUp()
        // Here, all the mockup variables for the STEP (global
        // variables, parameters, etc) have to be initializaed.

        binding.setVariable('FIRST_PARAMETER_VARIABLE', 'true')
        binding.setVariable('SECOND_PARAMETER_VARIABLE', 42)
        // Defines the previous execution status
        binding.getVariable('currentBuild').previousBuild = ['UNSTABLE']
    }

    @Test
    void launchSpecificStepTest() throws Exception {
        // Load the STEP groovy file to execute
        def script = loadScript("./vars/and_STEP_stepExample.groovy")
        script.call()
        printCallStack()
    }
}
```

Code 24 – Example of a Unit test Structure for the Malm-Shared project.

The result of the tests can be checked in the Run console of IntelliJ (or any of the valid IDE).

To maintain a robust and aligned unit test module, it is necessary that for every existing Step and for every new Step development, a unit test must be implemented. Without correctly executed unit tests, the User Story of the development should not be accepted.

[33] JUnit Gradle Documentation – https://docs.gradle.org/current/dsl/org.gradle.api.tasks.testing.Test.html
[34] JenkinsPipelineUnit Documentation – https://github.com/jenkinsci/JenkinsPipelineUnit

7 Roadmap Definition

In this chapter, I will present the proposed Roadmap for the implementation of the changes to help organize the work of the MALM team for the next 9 months.

The timings of the roadmap tasks have been calculated taking into account that:

- ⇒ There are only going to be 2 programmers implementing the changes at the same time.
- ⇒ The effort of these programmers is 50%, since the other half must be dedicated to developing the petitions that the customers require.
- ⇒ Some unforeseen events may arise, so they have been valued with an extra 10%.
- ⇒ Between developments, there are a couple of days to do code reviews with the team.

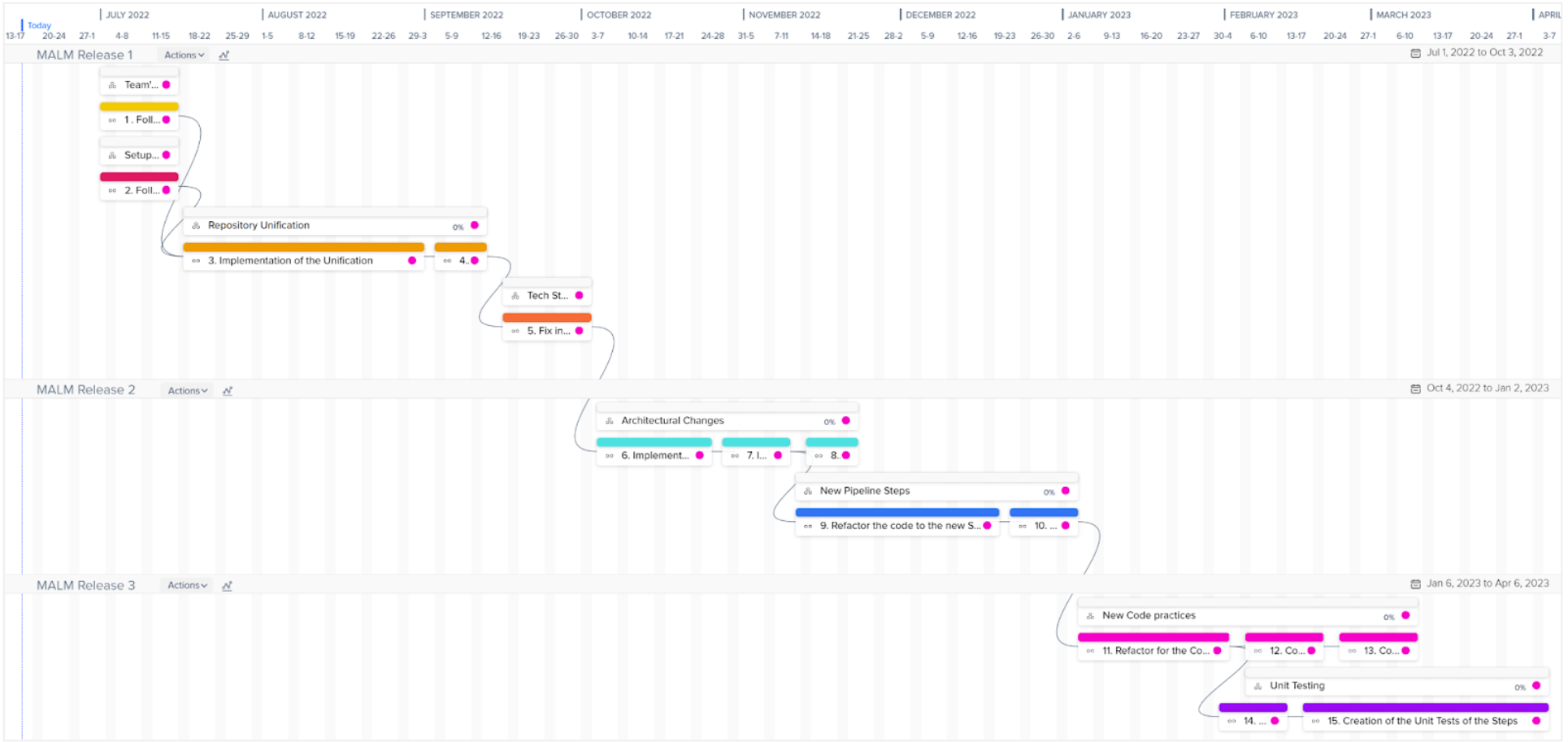
I have divided all the changes in releases of three months and scheduled them according to its technological dependence and, as a second factor, the criticity level.

Group	Task name	Release	Crit.	Est.
Team's Workspace setup	1. Follow the Workspace Setup Guide	1	Med	2w
Setup of the new Project	2. Follow the Gradle Setup Guide	1	High	2w
Repository Unification	3. Implementation of the Unification	1	High	6w 3d
Repository Unification	4. Regression tests after the Unification	1	High	1w 4d
Tech Stack Upgrade	5. Fix incompatibilities after the Upgrade	1	High	2w 2d
Architectural Changes	6. Implementation of the Architecture code Refactor	2	High	3w
Architectural Changes	7. Implementation of the new File Hierarchy	2	High	2w 2d
Architectural Changes	8. Implementation of the Filename Changes	2	Med	1w 4d
New Pipeline Steps	9. Refactor the code to the new Steps	2	High	5w 3d
New Pipeline Steps	10. Regression tests after the Steps Refactor	2	High	2w
New Code practices	11. Refactor for the Code Best Practices	3	Low	4w
New Code practices	12. Code Cleaning and Warnings Review	3	Low	2w
New Code practices	13. Configuration of the Groovydoc module	3	Low	2w
Unit Testing	14. Configuration of the new Unit Testing module	3	Low	2w 2d
Unit Testing	15. Creation of the Unit Tests of the Steps	3	Low	6w 4d

Table 18 – Roadmap tasks estimation table.

In Figure 12, on the next page, we can see the Roadmap diagram generated using the Craft.io[35] tool, with the task information in Table 18.

[35] Craft.io Official Website – https://craft.io/



1. Follow the Workspace Setup Guide.
2. Follow the Gradle Setup Guide.
3. Implementation of the Unification.
4. Regression tests after the Unification
5. Fix incompatibilities after the Upgrade
6. Implementation of the Architecture code Refactor
7. Implementation of the new File Hierarchy
8. Implementation of the Filename Changes
9. Refactor the code to the new Steps
10. Regression tests after the Steps Refactor
11. Refactor for the Code Best Practices
12. Code Cleaning and Warnings Review
13. Configuration of the Groovydoc module
14. Configuration of the new Unit Testing module
15. Creation of the Unit Tests of the Steps

Figure 12 – Roadmap diagram of the MALM's re-architecturization.

8 Conclusions

In this thesis, we have witnessed first hand a real case of **Technical Debt** that has taken its toll on a platform that is very resilient to change. This same concept has gone from a theoretical framework to a physical framework in the form of a document, thanks to the execution of the Analysis and Definition of this project.

With a cost of 14,742.58 €[36] (cost of the thesis), plus the cost of all the hours of the developers who will invest in implementing the refactoring of the code, we can ensure that the technical debt is a latent problem and, moreover, **very expensive**.

Focusing on the results of the thesis, for each of the main objectives, I will justify to what extent they have been achieved:

Required

⇒ Architecturization of the software. **(100%)**

- ✓ The architecture has been analyzed and re-architected according to its needs.
- ✓ A new file hierarchy has been defined.
- ✓ A naming convention has been specified.
- ✓ A Best Practices guidelines has been defined.
- ✓ A standardization protocol for the Work Environment has been implemented.

All the sub-objectives have been met, although I find that the first point (the one regarding architecture patterns) could have been much more significant on other platforms with clearer patterns.

⇒ Use Gradle to create a new repository for the project. **(100%)**

- ✓ The malm-shared repository has been converted into a Gradle project.
- ✓ Proper indexation of the code has been implemented.
- ✓ Dependencies have been centralized on the build.gradle file.
- ✓ A versioning protocol has been implemented for the repository source code.

All the sub-objectives have been met.

⇒ Unify the scripts repositories. **(100%)**

- ✓ The malm-shared and pipeline-utils repositories have been unified.

All the sub-objectives have been met.

[36] Extracted from section 3.2 Cost estimates.

⇒ Redesign the pipelines and its Stages and Steps. (66%)

- ✓ New, atomic pipeline steps have been defined.
- ✓ The Stages have been redefined.
- ✗ The management of the Exceptions have not been defined.

Almost all the sub-objectives have been met. The management of the Exceptions thrown by the Steps have not yet been defined, because I found it quite complex to define the exceptions of the steps without them being developed, at least, at a low level. I believe that it will be more useful to address this exception management in a phase 2 of the refactoring.

⇒ Implement Unit Testing. (50%)

- ✓ A Unit Testing module has been implemented.
- ✗ The specific tests have not been defined.

Almost all the sub-objectives have been met. The tests have not been implemented due to the high workload of this development, but a PoC and a fully functional module have been provided for developing and executing them.

Significant

⇒ Redesign the Jenkins global variables. (100%)

- ✓ The global variables have been refactored as part of the code cleaning section.

All the sub-objectives have been met.

⇒ Document the code using Groovydoc. (50%)

- ✓ An automatic documentation module has been defined and implemented.
- ✗ The specific documentation has not been defined.

Almost all the sub-objectives have been met. The documentation has not been implemented due to the high workload of this development, but a PoC and a fully functional module have been provided for its generation.

⇒ Upgrade the technological stack of the code and the software platform. (100%)

- ✓ The update of the Technology Stack has been defined and requested.

All the sub-objectives have been met.

Nice-to-have

⇒ Cleanup of deprecated code. (100%)

- ✓ The deprecated code has been analyzed and tagged for cleanup.

All the sub-objectives have been met.

⇒ Refactor the variable names to provide them with useful semantic information. (100%)

✓ The variable names have been refactored as part of the code cleaning section.

All the sub-objectives have been met.

⇒ Modify the code so that it complies with Groovy and Jenkins best practices. (100%)

✓ The code has been refactored as part of the code cleaning section.
✓ Project Best Practices have been defined.

All the sub-objectives have been met.

Next steps

⇒ Convert repository code into a real Shared Library. (0%)

✗ The specific analysis for the Shared Library conversion (the use of more global steps for all corporate projects) has not been done.

Due to time limitations, the objective has been left for a phase 2 refactoring.

⇒ Redesign the Jenkins pipeline parameters. UX improvement. (0%)

✗ The Jenkins pipeline parameters have not been analyzed nor defined.

Due to time limitations, the objective has been left for a phase 2 refactoring.

I consider that the results have been quite positive. In addition to the benefits brought to the MALM team and Opentreds, I feel that this thesis has allowed me to grow professionally as a developer and, most importantly, as a future Software Architect.

8.1 Limitations

Of course, there have been some limitations that have hindered the development of the project. The first limitation has been **not being able to show corporate or compromising information** for the company or the main customer that the MALM product is focused on.

It has been a challenge to approach the project with this requirement, since it implies masking data, changing file/tool names, undoing sections and eliminating diagrams for being too compromising. And more importantly, to achieve all of these aspects without the project losing content and value.

Another limitation has been to **remove the Analysis and Definition of Jenkins Parameters section from the scope**. This elimination is due to the fact that, being such a large platform and with so many users, changing the parameters of the Jenkins Jobs (which are informed by the users) requires much more planning and attention.

In addition, changing the parameters is a very specific requirement for the MALM application, which cannot be extrapolated to other systems. Since this is a requirement that I initially categorized as Next-Steps, foreseeing that it could be left out of scope, the decision seems appropriate to me.

8.2 Technical skills

⇒ **CES1.1:** Develop, maintain and evaluate complex and/or critical software systems and services. [In depth]

» The technical skill has been successfully achieved. A new architecture for a complex project has been analyzed, refactored and defined. PoCs have been implemented.

⇒ **CES1.2:** Provide solutions to integration problems based on available strategies, standards, and technologies. [Fairly]

» The technical skill has been successfully achieved. I have used several integrations with Gradle tools to add value and features to the Jenkins project.

⇒ **CES1.3:** Identify, evaluate and manage potential risks associated with building software that may arise. [Fairly]

» The technical skill has been achieved. Since this is a technical debt related project, I was able to analyze a project that was already affected by the risks, evaluate them, and solve them by defining a new architecture.

⇒ **CES1.7:** Control quality and design tests in software production. [In depth]

» The technical skill has been successfully achieved. I have implemented improvements and defined protocols focused on the quality of the project. I have also implemented a Unit Testing module.

⇒ **CES2.1:** Define and manage the requirements of a software system. [Fairly]

» The technical skill has been achieved. I have evaluated the platform requirements and defined new ones, as well as implemented and documented them.

8.2 Further work

In the short term, the work to be done is to implement this re-architecturing and refactoring of the code by the MALM team. It would be ideal to collect metrics (execution times, development time, number of implemented features per release, etc.) of the entire current platform, in order to be able to compare it with the platform once the changes are implemented. In this way, we can show these benefits in numbers, and not only in the theoretical framework of the project, although they may seem ominous.

Since Technical Debt is such a common aspect of the programmer's day-to-day life, it is difficult to expect that after the MALM refactor, such technical debt will cease to generate. Therefore, while gradually working on changing the corporate vision of fast time-to-market, I will allow myself to make one last proposal that I hope will help to detect TD in the future.

My proposal is to **set up a new TAG[37], similar to the existing @TODO or @FIXME**, to help record the TD. The idea is that, the developer himself will be able to detect when a development is forced to be done fast or see that it could be improved with more time, and add a tag @TEDE (TEchnical DEbt) right at the beginning of the method.

```
1  @TEDE                              // New Technical Debt tag
2  void methodToBeImpoved() {
3      // rest of the code
4  }
```

Code 25 – Use of the new @TEDE tag.

The work to be done in the long term is much more difficult, even utopian. **The business vision of forcing workers to deliver functionalities in very short times, and meeting very tight deadlines, must change** for the good of the community. Apart from the economic cost of TD and the waste of resources, it must be emphasized that **the mental health and well-being of workers is at stake**. Workers who spend hours and hours developing in environments that are difficult to maintain, difficult to scale and, in essence, environments that have been developed to meet a deadline, and not functional requirements.

[37] How to configure a @TODO tag, IntelliJ – https://www.jetbrains.com/help/idea/using-todo.html#add_pattern_filter_todo

9 Annex

9.1 Design Patterns

All the information has been extracted from the book Dive Into Design Patterns by Alexander Shvets (2018). Patterns can be categorized by three main groups: Creational Patterns, Structural Patterns and Behavioral Patterns. For each category, we can list the designs it includes, and add a brief description and an explanation of when it makes sense to use them.

Creational Patterns

They offer object creation mechanisms that increase flexibility and reuse of existing code.

⇒ Factory Method: Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
  » Use the Factory Method when you do not know beforehand the exact types and dependencies of the objects your code should work with.
  » Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components.
  » Use the Factory Method when you want to save system resources by reusing existing objects instead of rebuilding them each time.

⇒ Abstract Factory: Lets you produce families of related objects without specifying their concrete classes.
  » Use the Abstract Factory when your code needs to work with various families of related products, but you don't want it to depend on the concrete classes of those products.

⇒ Builder: Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
  » Use the Builder pattern to get rid of a very long constructor with a lot of optional parameters.
  » Use the Builder pattern when you want your code to be able to create different representations of some product.

⇒ Prototype: Lets you copy existing objects without making your code dependent on their classes.
  » Use the Prototype pattern when your code should not depend on the concrete classes of objects that you need to copy.
  » Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects.

⇒ Singleton: Lets you ensure that a class has only one instance, while providing a global access point to this instance.
  » Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.
  » Use the Singleton pattern when you need stricter control over global variables.

Structural Patterns

They explain how to organize objects and classes into larger structures, while keeping these structures flexible and efficient

⇒ Adapter: Allows objects with incompatible interfaces to collaborate.
  » Use the Adapter class when you want to use some existing class, but its interface is not compatible with the rest of your code.
  » Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can not be added to the superclass.

⇒ Bridge: Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.
  » Use the Bridge pattern when you want to divide and organize a monolithic class that has many variants of some functionality (for example, if the class can work with various database servers).
  » Use the pattern when you need to extend a class in many orthogonal (independent) dimensions.
  » Use the Bridge if you need to be able to switch implementations at runtime.

⇒ Composite: Lets you compose objects into tree structures and then work with these structures as if they were individual objects.
  » Use the Composite pattern when you have to implement a tree-like object structure.
  » Use the pattern when you want the client code to treat both simple and complex elements uniformly.

⇒ Decorator: Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.
  » Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.
  » Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.

⇒ Facade: Provides a simplified interface to a library, a framework, or any other complex set of classes.
  » Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.
  » Use the Facade when you want to structure a subsystem into layers.

⇒ Flyweight: Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all the data in each object.
  » Use the Flyweight pattern only when your program must support a huge number of objects which barely fit into available RAM.

⇒ Proxy: Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.
  » Lazy initialization (virtual proxy). This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.
  » Access control (protection proxy). This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various launched applications (including malicious ones).
  » Local execution of a remote service (remote proxy). This is when the service object is located on a remote server.
  » Logging requests (logging proxy). This is when you want to keep a history of requests to the service object.
  » Caching request results (caching proxy). This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.
  » Smart reference. This is when you need to be able to dismiss a heavyweight object once there are no clients that use it.

Behavioral Patterns

Behavioral design patterns are concerned with algorithms and the assignment of responsibilities between objects

⇒ Chain of Responsibility: Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.
  » Use the Chain of Responsibility pattern when your program is expected to process different kinds of requests in various ways, but the exact types of requests and their sequences are unknown beforehand.

» Use the pattern when it's essential to execute several handlers in a particular order.
» Use the CoR pattern when the set of handlers and their order are supposed to change at runtime.

⇒ Command: Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method argument, delay or queue a request's execution, and support undoable operations.
» Use the Command pattern when you want to parametrize objects with operations.
» Use the Command pattern when you want to queue operations, schedule their execution, or execute them remotely.
» Use the Command pattern when you want to implement reversible operations.

⇒ Iterator: Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).
» Use the Iterator pattern when your collection has a complex data structure under the hood, but you want to hide its complexity from clients (either for convenience or security reasons).
» Use the pattern to reduce duplication of the traversal code across your app.
» Use the Iterator when you want your code to be able to traverse different data structures or when types of these structures are unknown beforehand.

⇒ Mediator: Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.
» Use the Mediator pattern when it's hard to change some of the classes because they are tightly coupled to a bunch of other classes.
» Use the pattern when you can't reuse a component in a different program because it's too dependent on other components.
» Use the Mediator when you find yourself creating tons of component subclasses just to reuse some basic behavior in various contexts.

⇒ Memento: Lets you save and restore the previous state of an object without revealing the details of its implementation.
» Use the Memento pattern when you want to produce snapshots of the object's state to be able to restore a previous state of the object.
» Use the pattern when direct access to the object's fields/getters/setters violates its encapsulation.

⇒ Observer: Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.
  » Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.
  » Use the pattern when some objects in your app must observe others, but only for a limited time or in specific cases.

⇒ State: Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
  » Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently.
  » Use the pattern when you have a class polluted with massive conditionals that alter how the class behaves according to the current values of the class's fields.

⇒ Strategy: Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.
  » Use the Strategy pattern when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
  » Use the Strategy when you have a lot of similar classes that only differ in the way they execute some behavior.
  » Use the pattern to isolate the business logic of a class from the implementation details of algorithms that may not be as important in the context of that logic.
  » Use the pattern when your class has a massive conditional operator that switches between different variants of the same algorithm.

⇒ Template Method: Defines the skeleton of an algorithm in the superclass, but lets subclasses override specific steps of the algorithm without changing its structure.
  » Use the Template Method pattern when you want to let clients extend only particular steps of an algorithm, but not the whole algorithm or its structure.
  » Use the pattern when you have several classes that contain almost identical algorithms with some minor differences. As a result, you might need to modify all classes when the algorithm changes.

⇒ Visitor: Lets you separate algorithms from the objects on which they operate.
  » Use the Visitor when you need to perform an operation on all elements of a complex object structure (for example, an object tree).
  » Use the Visitor to clean up the business logic of auxiliary behaviors.
  » Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.

9.2 Workspace Setup Guide

1. IDE setup:

IntelliJ Community. → Website: https://www.jetbrains.com/idea/

Download for Windows, Mac, Linux

Currently, as of 05/24/2022, we now have the version:

Version: 2021.3.3 **Build:** 213.7172.25 - 17 March 2022

There should be no problem upgrading *minor* versions of the IDE.

For *major* version upgrades, the team should meet and decide whether to upgrade the IDE to keep an aligned workspace among the members.

If the *major* version is upgraded, it is required to update this documentation.

2. Plugins setup:

Next, on the IDE, we must access File > Settings > Plugins and then search and install the plugins from the following list:

Name	Version	Necessity	Comments
CodeNarc	4.4.0	Required	
CodeNarc Updated	4.2.1	Required	
Gruvbox Theme	0.5.4	Recommended	File > Settings > Appearance > Appearance , then set the theme to Gruvbox Dark Hard
Rainbow Brackets	6.22	Recommended	File > Settings > Rainbow Brackets
Statistic	4.1.10	Recommended	
TestMe	5.1.0	Required	

Table 19 – Plugins to configure on IntelliJ.

The plugins with the "Recommended" requirement, although not mandatory, are strongly recommended.

After installing the plugins, IntelliJ will require a restart to finish the configuration. New plugins can be installed at any time and can be shared with the team to improve the Workspace.

If the list of plugins is updated, this documentation needs to be updated too.

3. Custom Settings file Import

a) Download the settings_INTELLIJ.zip file from the Team's shared File System.

b) We navigate to File > Manage IDE Settings > Import Settings and select the file we just downloaded.

9.3 Gradle 7.1 Setup Guide

1. Download Groovy SDK:

Groovy SDK: 2.4.12

- Download apache-groovy-sdk-2.4.12.zip from the distribution list.

Currently, as of 05/27/2022, we have version 2.4.12 configured in the Jenkins Controller environments, so we will configure the same for the project.

For version updates, the team should meet and update the SDK version together to maintain an aligned workspace.

Once the version is updated, this documentation needs to be updated.

2. Setting up the Groovy SDK:

a) Access the project properties by Right-Clicking on the root directory of the Project, “malm-shared”, and navigate to Open Module Settings > Libraries.

b) Click on the + icon to add a new Java library. Then select the library of the SDK that we have downloaded in step 1 and click “Accept”.

c) Click on “Apply” and restart the IDE.

3. Setting up the Java JDK:

a) Access the project properties with Right-Click on the root directory of the Project, “malm-shared”, and navigate to Open Module Settings > Project.

b) Add the following settings:

Name:	malm-shared
Project SDK:	14AdoptOpenJDK (HotSpot) version 14.0.2
Language level:	SDK default (14 - Switch expressions)
Compiler Output:	*leave empty*

c) Click on “Apply” and restart the IDE.

4. Configuring the Gradle Project modules:

a) Access to the project properties with Right-Click on the root directory of the Project, "malm-shared", and navigate to Open Module Settings > Modules

b) Add the corresponding Content Root and configure the modules following the images below:

malm-shared module

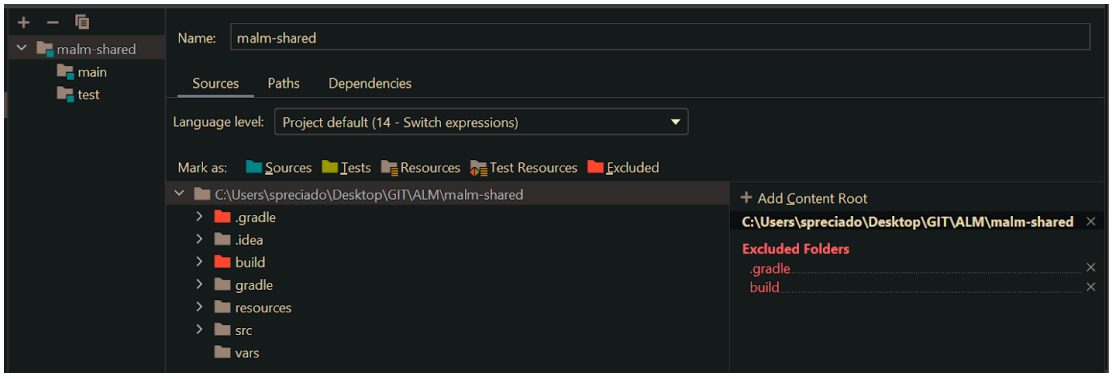


Figure 13 - malm-shared base project module configuration.

main module (Content Root /resources)

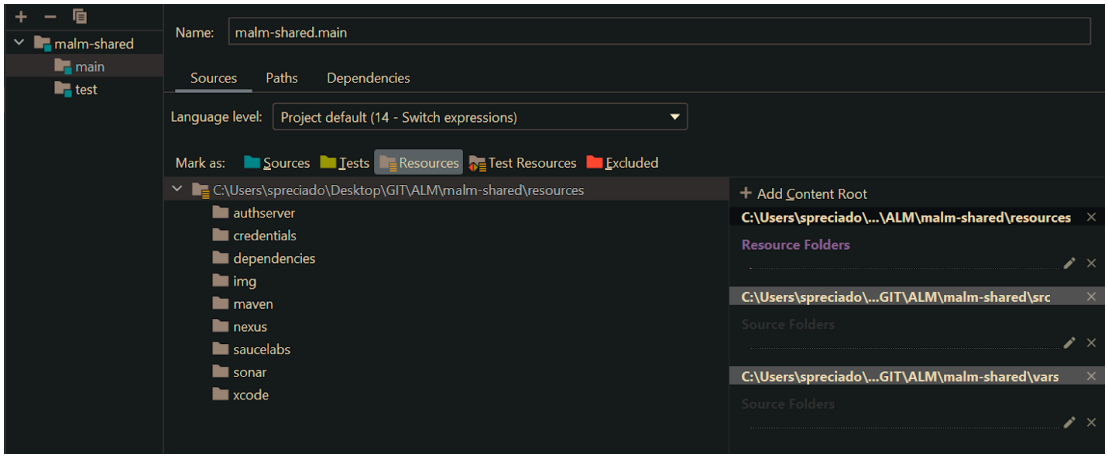


Figure 14 - main resources module configuration.

main module (Content Root /src)

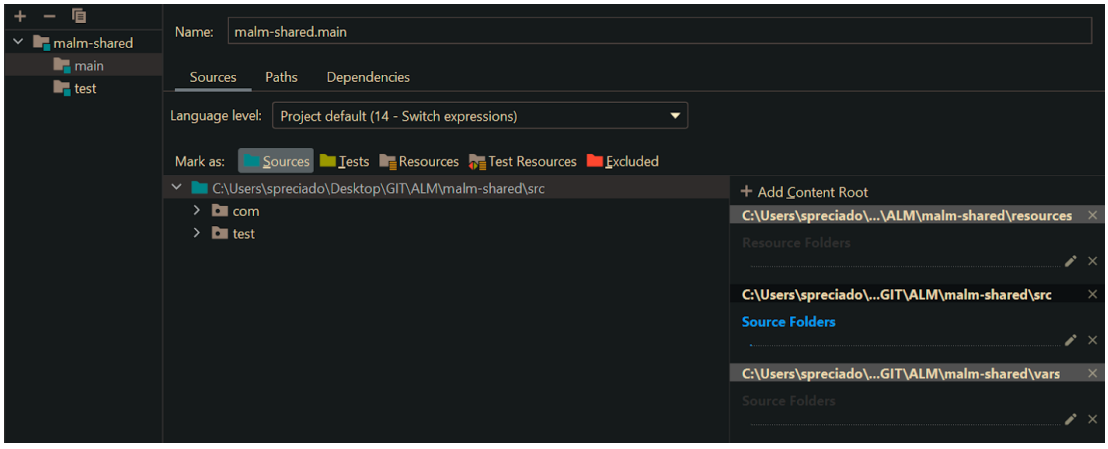


Figure 15 - main src module configuration.

main module (Content Root /vars)

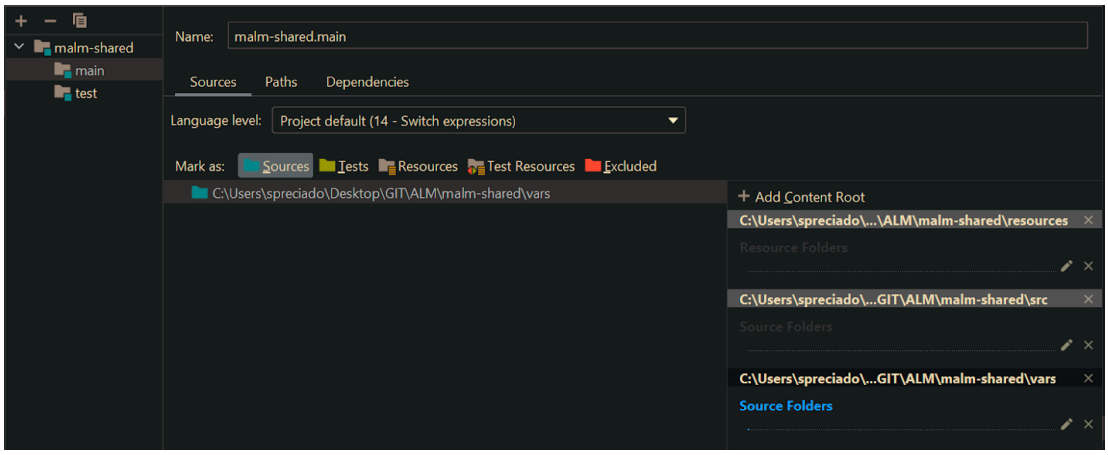


Figure 16 - main vars module configuration.

test module

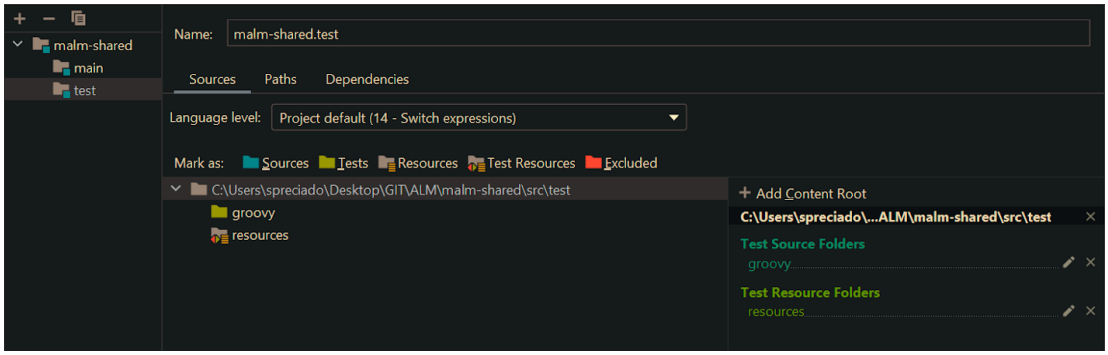


Figure 17 - test module configuration.

c) Click on “Apply” and restart the IDE.

9.4 New Pipeline Steps list

android_aar_pipeline

Stage	Step	Description
Checkout		
	1	Project checkout and download.
	2	Loading of libraries.
	3	Initialization of the loaded libraries.
	4	Notification of the start of the job execution in the MALM Jenkins.
Prepare		
	1	Specifying Nexus URLs
	2	Managing MALM parameters
	3	Obtaining data to build the notification mail
	4	Generating the properties file
	5	Sonar Configuration
	6	Obtaining the project dependencies
ARQ Rules		STAGE TO DELETE ON THE FIRST REFACTOR
Test		
	1	Test execution and report generation.
	2	Validation of test reports.
	3	Inclusion of test results in the notification email.
Sonar		
	1	Access to the Sonar profile.
	2	Configuring the properties of the gradle.properties file.
	3	Sonar execution.
	4	Inclusion of the Sonar URL of the project in the notification email.
	5	Obtaining and validating metrics from Sonar.
	6	Print metrics in log.
	7	Retrieval of the coverage value from Sonar.
	8	Checking the % coverage of the project in Sonar.
	9	Inclusion of the project's coverage data and test results in the notification email.
C.T & Coverage		STAGE TO DELETE ON THE FIRST REFACTOR
Build		
	1	Compilation of the project.
Nexus		
	1	Creation of the .zip file with the project resources
	2	Access to corporate Nexus to upload the project

	3	Modification of the Gradle file
	4	Generation of the pom file.
	5	Preparation of the package .aar and upload to Nexus.
	6	Inclusion of the Nexus URL in the notification email.
Tag		
	1	Access to GitLab.
	2	Creating the tag.
S. Security Analysis		STAGE TO DELETE ON android_aar_pipeline ON THE FIRST REFACTOR
Notify		
	1	Obtaining the status of the build
	2	Adding missing information to the notification email
	3	Configure email recipients
	4	Attach files
	5	Send the notification email
GSA		STAGE TO DELETE ON THE FIRST REFACTOR
Clean		
	1	Deleting project data and project folders

Table 20 – New Steps summary for the android_aar_pipeline.

ios_app_pipeline

Stage	Step	Description
Clean		
	1	Stopping running simulators.
	2	Deleting duplicate simulators.
	3	Execution of `deleteDir()` method.
	4	Deleting the contents of compilation folders.
	5	Resetting the simulators to their original state.
Checkout		
	1	Clone of the project.
	2	Selecting the gemset to use.
	3	Obtaining data to build the notification email.
	4	Selecting the target platform.
	5	Presentation of the environment variables.
	6	Notification of the start of MALM execution.

Prepare		
	1	Managing which Stages will be skipped.
	2	Viewing the compilation parameters.
	3	Obtaining the PRO nodes from Client servers.
	4	Managing the services.json file.
	5	Replacing Actors values for API/Gateway.
	6	Execute pod update.
	7	Generation of sources.zip file for Nexus.
MALM Rules		STAGE TO DELETE ON THE FIRST REFACTOR
Test		
	1	Creation of Sonar Properties.
	2	Creation of the Sonar Files.
	3	Execution of the project tests.
	4	Validation of test reports.
	5	Collection of coverage data for the notification email.
Sonar		
	1	Test coverage report generation with Slather
	2	Code quality review with SwiftLint
	3	Code complexity analysis and Xcode report generation with Lizard
	4	Coverage data scanning with SonarQube
	5	Obtaining and validating metrics from Sonar.
	6	Printing metrics to log.
	7	Recovery of the coverage value from Sonar.
	8	Checking the % coverage of the project in Sonar.
	9	Inclusion of the project's test results in the notification email.
C.T & Coverage		STAGE TO DELETE ON THE FIRST REFACTOR
Build		
	1	Copying files to destination path.
	2	Encryption of JSON files.
	3	Archive of the project via Fastlane.
Nexus		
	1	Determine the build type.
	2	Publishing the IPA in the corporate Nexus.
AppCenter		
	1	Checking the existence of the distribution group.
	2	Checking and limiting the number of Release Notes file characters.
	3	Uploading the application to the AppCenter platform.

	4	Search for the release once uploaded.
	5	Redistribution of the app to the Distribution Groups.
SauceLabs		
	1	Check for the existence of the IPA to be tested.
	2	Obtain the execution environment in SauceLabs.
	3	Get the type of execution to be performed in SauceLabs.
	4	Run the IPA upload to SauceLabs.
TestFlight		
	1	Validation of the generated IPA in the Stage Build.
	2	Publication of the IPA in Testflight.
Tag		
	1	Apply the tag to the branch with the new version.
	2	Update origin with the new tag.
S. Security Analysis		
	1	Calculation of a random value to be used as delay.
	2	Executing an HTTP POST request with the application data.
Notify		
	1	Compile the mail from the data collected in other stages.
	2	Send the mail to the teams involved in the development of the project.
	3	Notify via GitLab in the Merge Request discussion of the job status.
GSA		
	1	Obtaining the instance.
	2	Obtaining the platform.
	3	Obtaining the project parameters.
	4	Process the files.
	5	Process the build information.
	6	Sending the information.

Table 21 – New Steps summary for the ios_app_pipeline.

ios_pod_pipeline

Stage	Step	Description
Clean		
	1	Stopping running simulators.
	2	Deleting duplicate simulators.

	3	Execution of `deleteDir()` method.
	4	Deleting the contents of compilation folders.
	5	Resetting the simulators to their original state.
Checkout		
	1	Clone of the project.
	2	Selecting the gemset to use.
	3	Obtaining data to build the notification email.
	4	Selecting the target platform.
	5	Presentation of the environment variables.
	6	Notification of the start of MALM execution.
Prepare		
	1	Managing which Stages will be skipped.
	2	Checking the existence of the TAG to be published.
	3	Execute pod update.
MALM Rules		STAGE TO DELETE ON THE FIRST REFACTOR
Test		
	1	Creation of Sonar Properties.
	2	Creation of the Sonar Files.
	3	Execution of the project tests.
	4	Validation of test reports.
	5	Collection of coverage data for the notification email.
Sonar		
	1	Test coverage report generation with Slather
	2	Code quality review with SwiftLint
	3	Code complexity analysis and Xcode report generation with Lizard
	4	Coverage data scanning with SonarQube
	5	Obtaining and validating metrics from Sonar.
	6	Printing metrics to log.
	7	Recovery of the coverage value from Sonar.
	8	Checking the % coverage of the project in Sonar.
	9	Inclusion of the project's test results in the notification email.
C.T & Coverage		STAGE TO DELETE ON THE FIRST REFACTOR
Tag		
	1	Get the current tag list.
	2	Apply the tag to the branch with the new version.
	3	Update origin with the new tag.
Publish Specs		
	1	Publication of the podspec.

Notify		
	1	Compile the mail from the data collected in other stages.
	2	Send the mail to the teams involved in the development of the project.
	3	Notify via GitLab in the Merge Request discussion of the job status.
GSA		
	1	Obtaining the instance.
	2	Obtaining the platform.
	3	Obtaining the project parameters.
	4	Process the files.
	5	Process the build information.
	6	Sending the information.

Table 22 – New Steps summary for the ios_pod_pipeline.

References

Avery, L. (2018, December 30). The $85 Billion Cost of Bad Code | PullRequest Blog. Code Review as a Service. Retrieved March 1, 2022, from https://www.pullrequest.com/blog/cost-of-bad-code/

Besker, T., Martini, A., & Bosch, J. (2017, August 30). Impact of Architectural Technical Debt on Daily Software Development Work. Euromicro Conference on Software Engineering and Advanced Applications. Retrieved March 2, 2022, from https://ieeexplore.ieee.org/document/8051360

Camargo, J., & Martín-Sosa, S. (2019, September 30). DE LUCHA CONTRA. Economía Solidaria. Retrieved March 15, 2022, from https://www.economiasolidaria.org/wp-content/uploads/2020/08/manual-de-lucha-contra-el-cambio-climatico.pdf

Chien, D. (2020, May 03). File Structure : Broad Institute of MIT and Harvard. MIT Communication Lab. Retrieved March 26, 2022, from https://mitcommlab.mit.edu/broad/commkit/file-structure/

Corporate Finance Institute. (2016, Februrary 13). Straight Line Depreciation - Formula & Guide to Calculate Depreciation. Corporate Finance Institute. Retrieved March 12, 2022, from https://corporatefinanceinstitute.com/resources/knowledge/accounting/straight-line-depreciation/

Cozzetti B. de Souza, S., Anquetil, N., & M. de Oliveira, K. (2005, January 23). A Study of the Documentation Essential to SoftwareMaintenance. Researchgate. Retrieved 05 15, 2022, from https://www.researchgate.net/publication/200040518_A_Study_of_the_Documentation_Essential_to_Software_Maintenance

Digité, Incorporated. (2021, December 21). What Is Scrum Methodology? & Scrum Project Management. Digite. Retrieved February 28, 2022, from https://www.digite.com/agile/scrum-methodology/

Energuide. (2020, October 28). How much power does a computer use? And how much CO2 does that represent? Energuide. Retrieved March 12, 2022, from https://www.energuide.be/en/questions-answers/how-much-power-does-a-computer-use-and-how-much-co2-does-that-represent/54/

Fowler, M. (2019, -). Refactoring: Improving the Design of Existing Code. Addison-Wesley. https://martinfowler.com/books/refactoring.html

Fowler, M. (2019, May 29). Is High Quality Software Worth the Cost? Martin Fowler. Retrieved March 1, 2022, from https://martinfowler.com/articles/is-quality-worth-cost.html

Geeks for Geeks. (2022, February 23). Difference between Gradle and Maven. GeeksforGeeks. Retrieved June 12, 2022, from https://www.geeksforgeeks.org/difference-between-gradle-and-maven/

Indeed Editorial Team. (2021, May 27). 12 Risks in Software Development. Indeed. Retrieved March 1, 2022, from https://www.indeed.com/career-advice/career-development/risks-in-software-development

JavaTpoint. (2011-2021, - -). Java Naming Conventions. Javatpoint. Retrieved June 15, 2022, from https://www.javatpoint.com/java-naming-conventions

Jenkins. (2020, April 21). Jenkins User Documentation. Jenkins. Retrieved February 26, 2022, from https://www.jenkins.io/doc/

J. Reifer, D., & Hastie, S. (2017, August 10). Quantitative Analysis of Agile Methods Study (2017): Twelve Major Findings. InfoQ. Retrieved February 28, 2022, from https://www.infoq.com/articles/reifer-agile-study-2017/

Lyman, I., Donovan, R., & Pureur, P. (2021, October 18). Code quality: a concern for businesses, bottom lines, and empathetic programmers. Stack Overflow Blog. Retrieved March 1, 2022, from https://stackoverflow.blog/2021/10/18/code-quality-a-concern-for-businesses-bottom-lines-and-empathetic-programmers/

National Weather Service, US Dept of Commerce. (2018, 10 10). Hurricane Michael Hits Georgia. National Weather Service. Retrieved March 1, 2022, from https://www.weather.gov/ffc/2018_hurricane_michael

Novoseltseva, E. (2021, January 2). 15 benefits of software architecture you should know. Apiumhub. Retrieved April 24, 2022, from https://apiumhub.com/tech-blog-barcelona/benefits-of-software-architecture/

Opentrends. (2021, September 27). Opentrends demuestra su implicación con la sostenibilidad y el medio ambiente al obtener la ISO 1400-1. Opentrends. Retrieved February 25, 2022, from https://www.opentrends.net/es/opentrends-demuestra-su-implicacion-con-la-sostenibilidad-y-el-medio-ambiente-al-obtener-la-iso

Red Hat. (2018, 9 17). Topics Understanding DevOps What is CI/CD? Red Hat. Retrieved February 26, 2022, from https://www.redhat.com/en/topics/devops/what-is-ci-cd

Richards, M. (2015, -). Software Architecture Patterns: Understanding Common Architecture Patterns and when to Use Them. O'Reilly Media. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

Shvets, A. (2018, -). Dive Into Design Patterns (1st ed.). Refactoring.Guru. https://www.goodreads.com/en/book/show/43125355-dive-into-design-patterns

Standish Group International, Incorporated. (2010, -). CHAOS Manifesto: The Laws of CHAOS and the 100 Best PM Practices. Standish Group International, Incorporated. https://www.immagic.com/eLibrary/ARCHIVES/GENERAL/GENREF/S110415C.pdf

Stripe and Harris Poll. (2018, September 21). The Developer Coefcient. Stripe. Retrieved March 2, 2022, from https://stripe.com/en-es/reports/developer-coefficient-2018

Sultan, S. (2019, October 26). What is Bad Code? How to Write Clean Code? Sunny Sultan. Retrieved March 1, 2022, from https://sunnysultan1640.medium.com/what-is-bad-code-how-to-write-clean-code-a9b7b539ad8

Tornhill, A., & Borg, M. (2022, March 8). [2203.04374] Code Red: The Business Impact of Code Quality -- A Quantitative Study of 39 Proprietary Production Codebases. arXiv. Retrieved April 24, 2022, from https://arxiv.org/abs/2203.04374

Wehner, C. (2020, January 16). Accounting for Computer Software Costs. Gross Mendelsohn. Retrieved March 12, 2022, from https://www.gma-cpa.com/blog/accounting-for-computer-software-costs