INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Análise de Repositórios GIT com Recurso a GraphQL

MIGUEL DE MORAIS TEIXEIRA PINTO
Junho de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

GIT Repository Analysis Exposed via GraphQL

Miguel Pinto

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

Supervisor: Dr. Nuno Bettencourt

Evaluation Committee:
President:
Dr. Ricardo Almeida, Professor, DEI/ISEP

Members:
Dr. Isabel Azevedo, Professor, DEI/ISEP
Dr. Nuno Bettencourt, Professor, DEI/ISEP

Porto, June 30, 2022

Dedicatory

To my family, especially to my parents, without whom all the journey made so far would not have been possible

Abstract

Currently, GIT repositories analysis tools play an important role in all software development teams, as they allow a detailed analysis of the interaction of all team members. It is important for a team leader to understand if the team is in fact being productive, and if not, he can use these tools to understand what can be improved to achieve that goal.

In 2012, facebook launched a query language called GraphQL, which was only released to the general public in 2015. Since then, this language has been gaining more and more popularity, being touted as a possible successor to the REST API. This language has several advantages, namely in terms of application performance and flexibility. Currently this language is already used by large companies such as Facebook, Netflix, GitHub and PayPal.

Given the evolution of the language, the possibility of creating a solution capable of providing a GraphQL API emerged in an academic context. As the use of this tool is often associated with a microservices architecture, this architecture was also chosen for the construction of the solution. The fact that both are recent concepts also contributes to the choice, thus ensuring a greater ease of updating the solution over time.

One of the conclusions of this study is that the use of a GraphQL API makes it easier for the user to obtain the desired answer. It is not possible to say that the use of GraphQL is always advantageous over the use of REST, being necessary to evaluate for any solution which tool is most justified or else the use of both simultaneously, which is also a possibility. Regarding the integrated tools, it was also concluded that the application of MapReduce techniques does not improve the performance of the solution, producing exactly the opposite effect.

Keywords: GraphQL, Microservices, GIT

Resumo

Atualmente as ferramentas de análise de repositórios GIT desempenham um papel importante em todas as equipas de desenvolvimento de software, uma vez que permitem uma análise detalhada da interação de todos os membros da equipa. É importante para um chefe de equipa perceber se a equipa está de facto a ser produtiva, e se não estiver, pode recorrer a estas ferramentas para perceber o que pode ser melhorado para atingir esse objetivo.

Em 2012, o facebook lançou uma linguagem de consulta chamada GraphQL, que apenas foi lançada ao público em geral em 2015. Deste então, esta linguagem tem vindo a ganhar cada vez mais popularidade, sendo apontada como a possível sucessora da API REST. Esta linguagem apresenta diversas vantagens, nomeadamente ao nível do desempenho e flexibilidade das aplicações. Atualmente esta linguagem já é usada por grandes empresas como Facebook, Netflix, GitHub e PayPal.

Dada a evolução da linguagem, surgiu em contexto académico a possibilidade de criar uma solução que fosse capaz de disponibilizar uma API GraphQL. Como o uso desta ferramenta surge muitas vezes associado a uma arquitetura de microserviços, foi também escolhida esta arquitetura para a construção da solução. O facto de serem ambos conceitos recentes também contribui para a escolha, garantindo assim uma maior facilidade de atualização da solução ao longo do tempo.

Uma das conclusões deste estudo é que o uso de uma API GraphQL oferece ao utilizador uma maior facilidade em obter a resposta pretendida. Não é possível afirmar que o uso de GraphQL é sempre vantajoso relativamente ao uso de REST, sendo necessário avaliar para qualquer solução qual a ferramenta que mais se justifica ou então o uso das duas em simultâneo, que também é uma possibilidade. Relativamente às ferramentas integradas, chegou-se também à conclusão de que a aplicação de técnicas de MapReduce não melhora a performance da solução, produzindo exatamante um efeito oposto.

Acknowledgement

To my academic supervisor, Nuno Bettencourt, for all the help throughout the project

To my colleagues, João Fontão and Luís Monteiro, for all the mutual help and good team spirit over the years. Thank you, it was a great pleasure to share this whole journey with you.

To my family, for all the support and encouragement to never give up. Thank you for always being here for me. I will be eternally grateful to you.

Contents

List of Figures

List of Tables

List of Algorithms

List of Source Code

List of Symbols

λmax	relative priority matrix max value
IC	consistency index

List of Acronyms

AHP	Analytic Hierarchy Process.
API	Application Programming Interface.
CI/CD	Continuous Integration Continuous Delivery.
DDD	Domain Driven Design.
DSL	Domain-Specific Language.
FAST	Function Analysis System Technique.
FEI	Front End of Innovation.
GQM	Goal Question Metric.
HTML	HyperText Markup Language.
JEE	Java Enterprise Edition.
JSON	JavaScript Object Notation.
MVC	Model-View-Controller.
NCD	New Concept Development Model.
REST	Representational State Transfer.
SQL	Structured Query Language.
UI	User Interface.
VCS	Version Control Systems.
XML	Extensible Markup Language.

Chapter 1

Introduction

This chapter presents the project developed during the master's degree in Informatics. Engineering, area of specialization in Software Engineering, at Porto School of Engineering (ISEP). Firstly, there is a brief contextualization and explanation of the problem. Then, the primary goals are introduced and used research method. Lastly, it describes the document's organization.

1.1 Context

In the context of software development, the analysis of the performance and productivity of teams has assumed an increasingly important role since all organizations want to know if their teams are producing what is expected according to the company's goals. An individual analysis is also an aspect that must be considered. It allows us to understand if the employee is actively collaborating and generating profitability for the company, thus facilitating the performance evaluation process [3].

With the constant development and technological evolution, the existing solutions may become obsolete in the short term since they use architectures that do not favour maintainability, which can generate an added cost when it is necessary to alter the system. Allied to this evolution, the appearance of GraphQL, introduced by Facebook, presented a viable alternative to using a Representational State Transfer (REST) Application Programming Interface (API), allowing customers to obtain precisely the data they need, unlike what happens with a REST API that can return the user a considerable amount of unwanted data [2, 4].

The monolithic architectures have also been used only in applications of small dimensions that deal with a low amount of data. At the same time, solutions based on microservices are designed to deal with a robust, scalable system capable of handling a considerable amount of data. It also creates independence between all microservices, which makes the system unaffected by a failure in just one or a few modules [5].

1.2 Problem

This project aims at creating a GIT repository analysis service that can produce individual and team performance reports about commits. From a GIT repository statistical point of view, it should provide information about some committed, deleted, altered lines for a given period. Furthermore, it should allow the usage of plugins that might provide different statistics regarding specific languages and code coverage [6].

The system should provide a dashboard to show the GIT repository health and statistics for each individual or team. The system should allow the development of plugins for Bitbucket, GIThub, Sonarqube and other tools that operate on GIT to use the system and display information. Because analysis of GIT repositories is quite demanding, techniques of parallel computing are required.

Another problem is the usage of an architecture capable of dealing with the complexity of analysing large repositories with too many lines of code. It must be used as a technology that favours the communication between other services developed by other students.It must also provide the final users with an easy way to obtain only the data they want .

1.3 Objectives

The main objective of this work is to develop a solution capable of analyzing GIT repositories, also trying to integrate existing tools to provide information to return to the data consumer. This solution must also have concerns about efficiency, performance, horizontal and vertical scalability, maintainability and also in terms costs.

Another objective is the use of recent technologies that make the solution follow the evolution of the technological market.

With the analysis of the presented objectives, it is possible to create the following research question:

- Can the solution integrate different GIT repositories analysis tools?

1.4 Research Method

For the investigation, within the scope of the development of the solution, and to achieve the objectives (section 1.3), was used the *design science research* method. This research method is indicated with the focus of the project being the development of a new solution [7]. This method essentially consists of six phases, which are described in the following topics:

1. Identify the problem, the research challenges and the potential benefits of the solution. This phase of the method is applied on chapter 1.
2. Definition of goals for the solution. This phase of the method is applied on chapter 1.
3. Design and implementation. This phase of the method is applied on chapters 4 and 5.
4. Demonstration of problem solving using the solution developed. This phase of the method is applied on chapter 6.
5. Evaluation of the solution with a comparison between the objectives and the obtained results. This phase of the method is applied on chapter 6.
6. Communication of the problem, the developed solution and how the solution contributes to creating a positive impact. This phase of the method is applied on chapter 7.

The section 1.6 includes the description of all chapters described on section 1.4.

1.5 Hypotheses

The hypotheses that aim to corroborate the research question defined on section 1.4 are:

H1 - The solution to be developed will present more features than other existing solutions

H2 - The solution to be developed will be useful in different contexts, such as business and academic

H3 - The solution to be developed will have a good performance and will be able to handle a large amount of data

1.6 Document Structure

This document consists of the following chapters:

Introduction (Chapter 1): It describes an initial context of the project to be developed, the problem and its objectives, and finally, the research method used.

State of the Art (Chapter 2): It describes a literature review that includes an introduction to GIT and team performance, some studies between monolithic and microservices architecture, a comparison between REST and GraphQL and all used tools and methods on other chapters.

Analysis (Chapter 3): It includes the value analysis for the product and also the requirements engineering analysis, that includes the domain model, functional and non-functional requirements.

Design (Chapter 2): This chapter includes the design of the solution in different levels and some design alternative approaches.

Implementation (Chapter 2): It describes all the implementation of the solution, giving answer to the functional and non-functional requirements defined on analysis (chapter 3).

Evaluation and Experimentation (Chapter 2): This chapter presents all tests for the solution and it is presented how this tests are going to be validated.

Conclusion (Chapter 2): The last chapter presents some conclusions about the described work on the document, namely whether the objectives were met and about the future work that could be done.

Chapter 2

State of the Art

This chapter presents some essential knowledge about Git repositories and what can be done to analyze all of the code presented on that. After that, it is described some existing solutions to answer this problem and some different approaches and technologies that can be used to this effect, as different architectures, communication standards and different query languages to use. The chapter has a sequence for the presentation of information, and this chapter describes all the technologies and solutions that were not developed within the scope of this project.

2.1 GIT

Git has introduced in 2005 and since its introduction has been the strong meaning when the subject matter is Version Control Systems (VCS). The introduction of this very useful tool was a success due to the opportunity to use new patterns that can give the teams a way to work more efficiently. It is really important to understand the impact that Git caused on team performance and their consequent analysis of results [8].

The main advantages that git can offer to the teams are:

- **Easy and cheap creation of branches** - the majority of the teams want to use a parallel development and avoid the usage of a single branch. Git offers developers the chance to create branches to ensure work isolation. And for that, they can use local branches too, a feature that was not present in others VCS
- **Allowing to commit and revert locally** - another feature that contributes to isolation. It allows creating restore points combined with multiple branches
- **Merge operations at a fine level of granularity** - provides a really helpful ability to analyze individual parts of a commit, rather than entire commits. It is helpful to teams because of your capacity for early and speedy detection of possible defects

However, not everything was in favour, and one of the disadvantages of using Git is the artefact store, due to millions of code changes during all development history. If the main goal is to analyze the code velocity of a team, it is a concern to take care of because the definition of branches on Git is very different to other VCS. On the following image is possible to see a git dag, which is an acyclic graph that is constructed with all of the types used on Git. This graph saves a registry of all relations between commits and children from other branches. It is possible to see the commits on the master branch, their fast-forward relations and merges necessary to synchronize the branches with master branch. The final result shows a simple process on git, so the complexity created in a large and complex project is predictable [9].

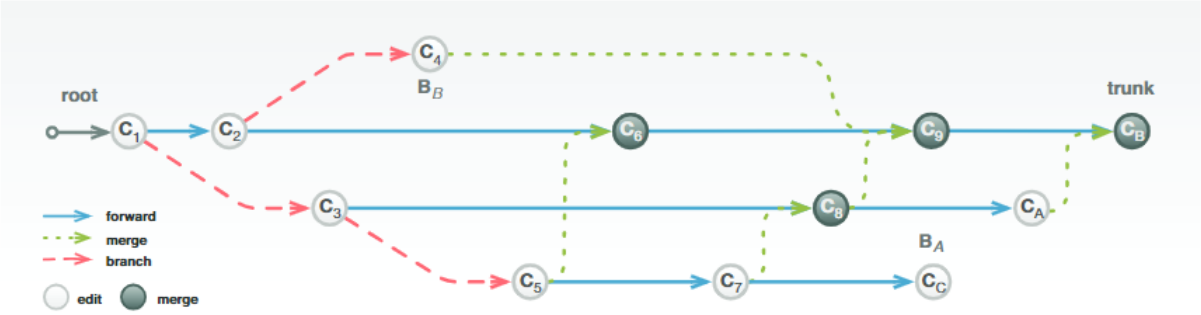


FIGURE 2.1: GIT DAG Example from [8]

2.2 Team Performance Analysis

In the software engineering world, the majority of courses involve team-based tools to develop their work. In these courses, it's relevant to understand what is the contribution of each one of the members to the work developed, and this can be used to generate reports based on predefined metrics to use on the analysis [10].

The metrics used on analysis can be a lot of them, so had to select which ones are more suitable to use, depending on the context of the project. Some examples of these metrics are:

- Number of commits
- Number of merge pull requests
- Number of files
- Total lines of code
- Time spent each day

As said, to look at these metrics, it´s necessary to know what are the different object types that exist on git and what is the different existing files of one repository that can be considered on analysis for team and individual performance [8]:

1. BLOB - contains the content of a single file
2. TREE - it is composed of a directory tree and has a series of entries, which can be a BLOB or another TREE object
3. COMMIT - represents a snapshot in content history. A commit creates a reference to a TREE object, all parent commits,information about the author and the committer
4. TAG - is a point of reference for any GIT object

Each of these object types has a unique ID represented by a SHA1 hash of the object. The header contains the type of object followed by its size in bytes and `\0`. On the following image, it is possible to see all of these objects, and on a repository, they are placed all on the same folder, identified by the ID that is also the filename. The image can give an idea that all of the complexity that can be generated on a project with many commits and the caution that has to be taken to choose an adequated approach to retract and analyze all the artefacts present on a repository [11]. However, the usage of git and his consequent analysis brings back in majority only benefits, since the academical context, that can provide an inclusion of in real-life situations to the curricular units [12], until professional teams that gives an powerful to increase their work productivity.

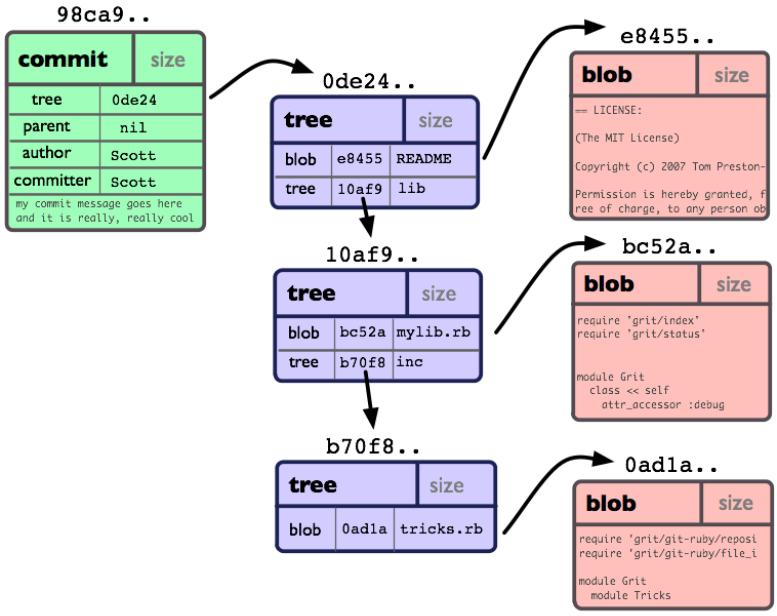


FIGURE 2.2: GIT Objects from [13]

Considering all that has been said before, it is necessary to know the different git providers to be aware of different implementations. These implementations should be a concern because the analysis should be independent of any provider and more abstract as possible [14].

2.3 Git Providers

When the subject is git, it is necessary to talk about those most famous providers that have gained so many users in the last years, like Bitbucket, GitHub and GitLab.

2.3.1 BitBucket

Bitbucket is a git provider developed by Atlassian, and as the company say "is more than just Git code management.Gives teams one place to plan projects, collaborate on code, test, and deploy". A simple overview of this product, provides easy integration with Jira, a product also developed by Atlassian that focuses on agile project management. It provides many features beyond the common git usage, like an implementation of a Continuous Integration Continuous Delivery (CI/CD) tool and a security tool that helps teams to avoid worrying about the weaknesses on their repository and code [15].

2.3.2 GitLab

"GitLab is an open source end-to-end software development platform with built-in version control, issue tracking, code review, CI/CD, and more. Self-host GitLab on your own servers, in a container, or on a cloud provider" [16]. This tool is more oriented to DevOps operations but has all the features that other VCS have too. Some negative points of GitLab are that it has a lot of bugs and a small community, even though the recent new users in the last years. Despite this, it is one of the providers with more flexibility and a lower price to use all the version control features and all the other processes, like deploy or code maintenance. However, there is a free version of this provider with a limited features and resources. [17].

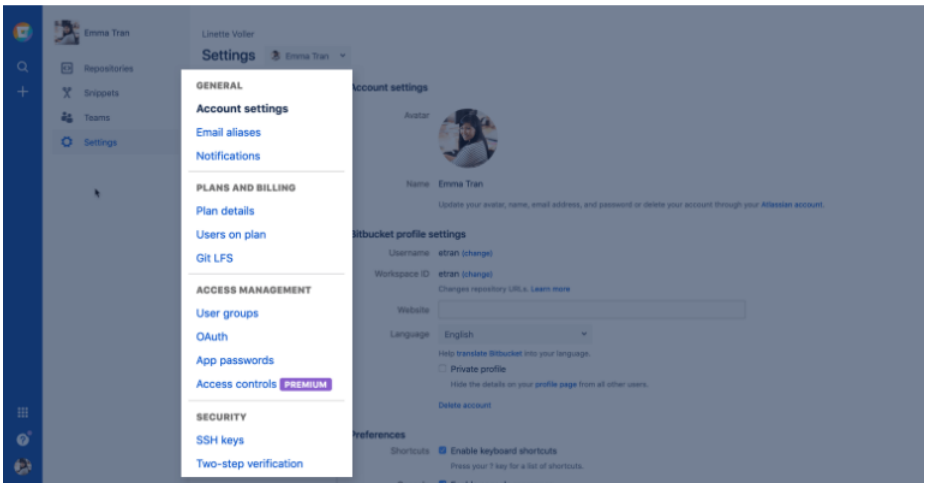


FIGURE 2.3: Bitbucket Interface from [15]

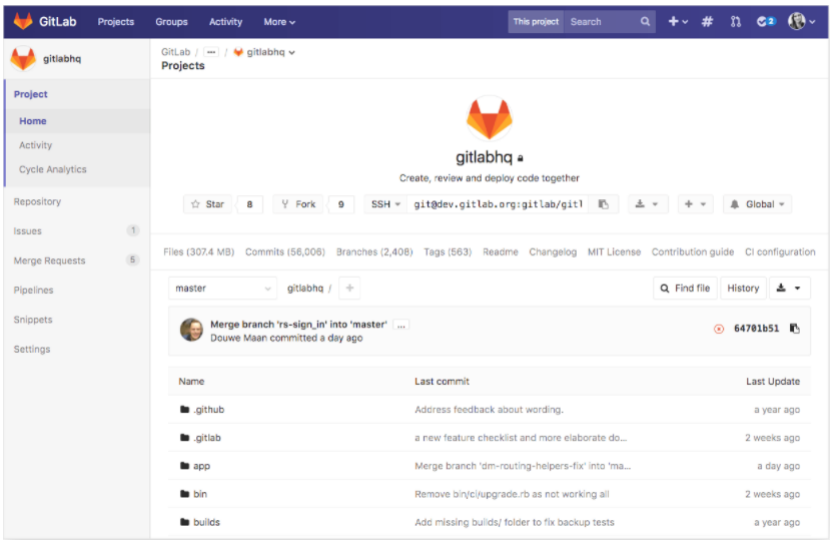


FIGURE 2.4: GitLab Interface from [18]

2.3.3 GitHub

GitHub is another provider of a git implementation and "millions of developers and companies build, ship, and maintain their software on GitHub—the largest and most advanced development platform in the world" [19]. Related to the analysis of repositories, GitHub has a API that can query any project. However, using this API could not be a good solution due to having a limit to the hourly number of requests and becoming a problem when analyzing large projects. Joining this problem, the API can only provide precisely the same information accessed on the GitHub repository interface [20, 21]. Based on GitHub repositories, it was created by the community a tool that can "record the public GitHub timeline, archive it, and make it easily accessible for further analysis". These records are created on JavaScript Object Notation (JSON) files, and the data can be accessed using arbitrary SQL queries [1, 21].

```
SELECT event as issue_status, COUNT(*) as cnt FROM (
  SELECT type, repo.name, actor.login,
    JSON_EXTRACT(payload, '$.action') as event,
  FROM `githubarchive.day.20190101`
  WHERE type = 'IssuesEvent'
)
GROUP by issue_status;
```

Listing 2.1: SQL Query used on Github Archive from [1]

2.3.4 Reporting and Analysis Tools

Nowadays, already some tools that can create reports and activity analysis from software projects that use Git. However, these tools cannot answer all of the problems that this context creates. With the subsequent analysis of some of these tools, it was possible to conclude this.

Gitana

"Gitana is a project inspector that analyzes the support tools used in software projects and imports the information in a relational database, thus providing a central point to perform all kinds of cross-cutting analysis on project data" [22]. This tool was created to try to correct some of the failures that exist on other tools, like the case of GHTorrent, a tool that only analyzes GitHub repositories, [23]. These two tools are GrimoireLab and Kibble, similar to Gitana, but this last one is the only one that can provide interaction with information by using Structured Query Language (SQL) queries. This implementation can be a beneficial feature to the people unfamiliar with Python libraries used on the other tools to explore the project data. The database is the centre of the system because it was what the other systems use to save all data and users of the application to create SQL queries to obtain repositories data. The Gitana architecture is described on image 2.5.

The architecture has systems that can import or update data to the previously created database with the help of a conceptual schema. By the end, it can use exporters to produce graph analysis and activity reports.

2.4 Architecture

After a superficial analysis of git and some existing analysis tools, the architecture used on the approach is another point to discuss and check the advantages and disadvantages between current architectures.

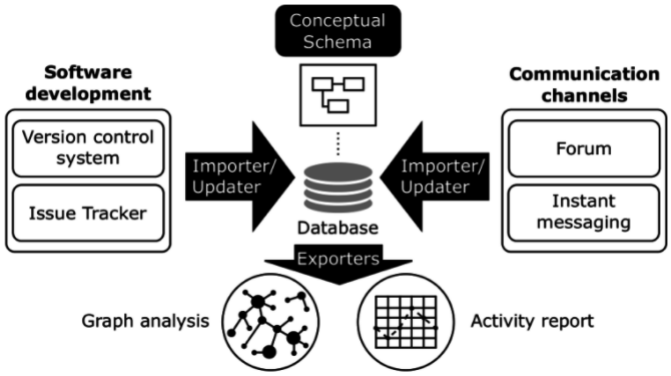


FIGURE 2.5: Gitana Architecture from [22]

2.4.1 Monolithic Architecture

"Monolithic architecture is an application with a single code base that includes multiple services". These services can communicate with possible consumers or external systems in different ways, like Web services, HyperText Markup Language (HTML) pages or a REST API [5]. When the idea is to create an application with a monolithic architecture, it is crucial to be aware that only exists a single codebase that will supply all the different services that were designed using a Model-View-Controller (MVC) web application framework like .NET or Java Enterprise Edition (JEE)[24]. Generally, applications with this architecture have the services encapsulated between them and can not be executed independently, as is possible to see on the image 2.6 [25].

On the image 2.6, it is possible to see the layered architecture inherent to a monolithic architecture, meaning that each layer has a different purpose on the application. The presentation layer can communicate with external systems data that is mapped on the other layers. The business layer is responsible for all business logic that will communicate with the persistence layer that uses the database layer to save all data used on the entire workflow.

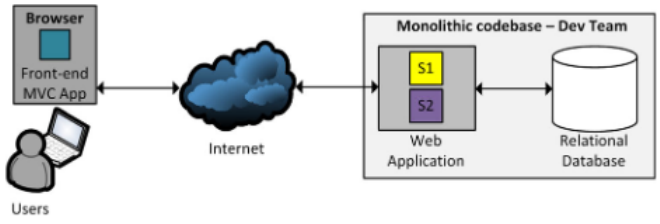


FIGURE 2.6: Monolithic Architecture from [24]

2.4.2 Microservices Architecture

Microservices architecture is a popular solution on software development, as said on [26], and is different to a monolithic architecture, due to organizes all the application in a collection of services. Each microservice have the responsibility to provide a part of the business logic [27].

As the opposite of the monolithic architecture, "microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service". An essential characteristic of this architecture is that each microservice is independent,

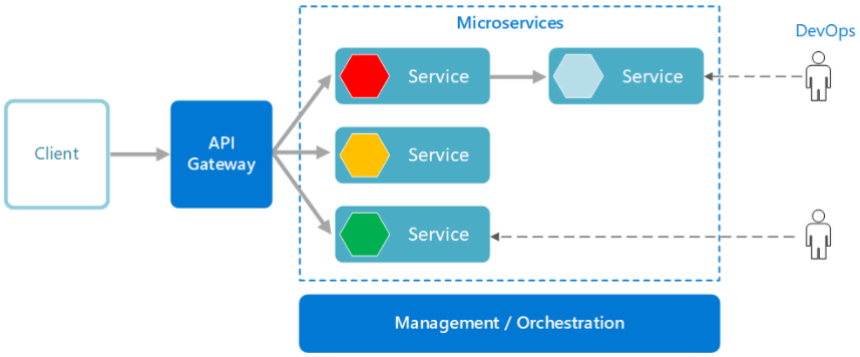


FIGURE 2.7: Microservices Architecture from [28]

avoiding all the problems when having some dependencies on an architecture. The communication between microservices is another crucial characteristic due to its lightweight way and do not need to use any complex communication channel [28].

With this architecture, some design patterns, like API Gateway and Event Sourcing, can be applied and have specific usage depending on the context, and it must be a concern to care about during architecture design. On this section, are analyzed three of this patterns that can be considered when developing a microservices solution.

Database per Microservice

One of the biggest challenges after choosing a microservices architecture is to choose what is the way to persist data how to use databases. On a large-scale system, as the case when using microservices, the usage of a single database can be an anti-pattern, which will cause high coupling with the database layer [29].

Instead of using a single database, a solution that uses a database for each microservice is a better idea due to can provide non-strong-coupling architecture. It can be used in different databases or schemas on the same physical database. This solution generates a challenge in the process to share data among services.

On small-scale applications and in cases that is one team developing all microservices, this can not be a good approach and have to be evaluated the pros and cons to ensure that will not create more problems in the future [30].

Event Sourcing

Another of the approaches with microservices, especially combined with the pattern mentioned on section 2.4.2, is the communication asynchronously with event exchanges to data transactions. In Event Sourcing, the business entities are not directly stored on the database, thought that are the stage-changing events and all types of events that are stored instead of entities. The state of one of these entities is obtained by an immutable series of events that can reply to many events to query the appropriate state [30].

The usage of this pattern is appropriate on systems that also use an event-oriented architecture with relational or non-relational databases and with a highly scalable transactional system.

API Gateway

This approach tries to avoid the problem that exists when a User Interface (UI) have to connect with multiple microservices. With all the security and layers that each microservice provides, it is difficult and complex to communicate directly between the front-end and back-end. So, API Gateway has an extra layer that functions like a

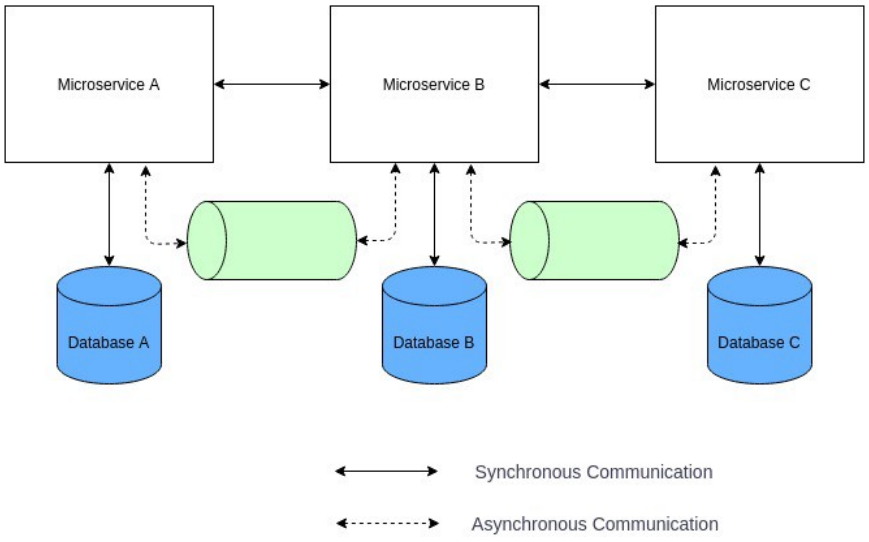


FIGURE 2.8: Database per Microservice Architecture from [30]

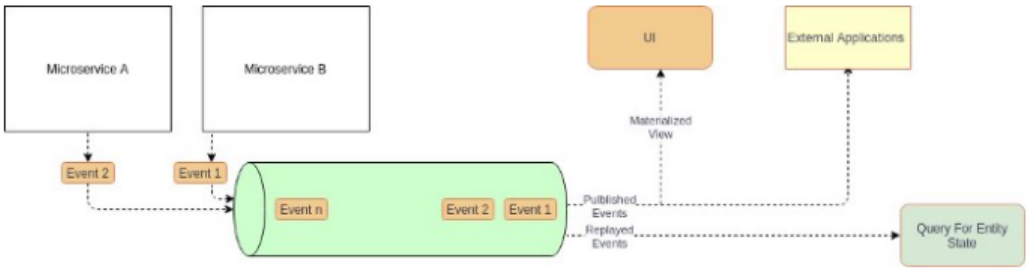


FIGURE 2.9: Event Sourcing Architecture from [30]

facade between the client-side application and the system's back-end. Depending on the request, the API gateway will forward to the appropriate microservice to address the request and answer with the requested data. The cross-cutting concerns are compatible with this pattern and easier to implement because the requests are sent to the same endpoint [30].

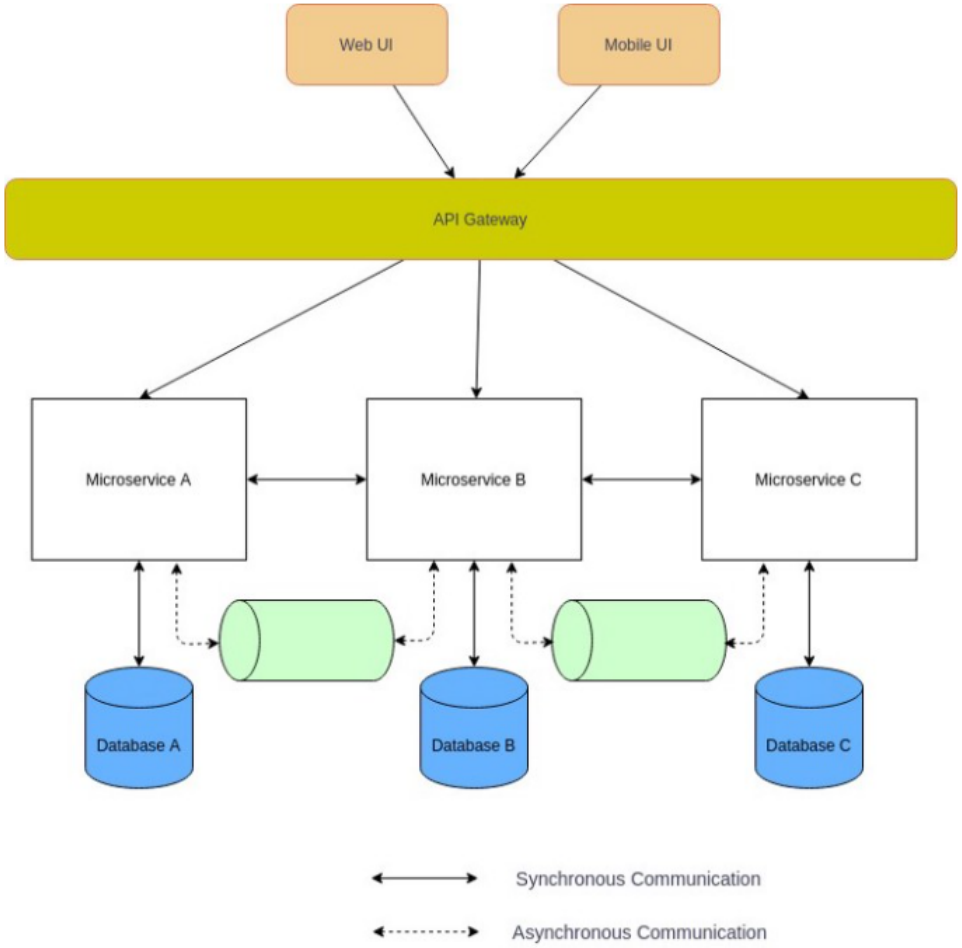


FIGURE 2.10: API Gateway Architecture from [30]

2.4.3 Monolithic and Microservices Architecture

Different advantages prove that using a microservices architecture is more adequate than a monolithic. Still, some situations don't justify the usage of microservices on a small project with lower scalability. On different studies developed by [27] and [5], are analyzed some advantages and disadvantages for the different architectures. For the monolithic architecture, the advantages are the following:

- Easy to develop
- Simple to deploy

For this architecture, the pros are few compared to the cons that are presented on the following topic:

- Complex maintenance (high coupling cause that)

- Reliability (not have a single point of failure and a simple failure can bring down all the system)
- Availability (need to deploy all the applications when an update is done)
- Hard to scale (a single module with all services compacted can turn this a hard task)

At the opposite of the monolithic architecture, microservices offer more pros than cons. Some advantages are:

- Easy maintenance (one module to a specific feature)
- Reliability (a fault on a microservice doesn't bring down all the system)
- Availability (an update needs a lower time down to the microservice)
- Easy to scale (low coupling)

Related to disadvantages, are not properly a negative point, are consequences due to the complexity and strength of the system:

- More complex deployment (each microservice needs a different configuration)
- Autonomy (represents only a big challenge to achieve, not a concrete disadvantage)

The two images from the two studies prove the conclusions presented above. The first compares the number of requests per service to milliseconds that these architectures delay processing. At the same time, it is possible to see that there is no significant difference between the two architectures.

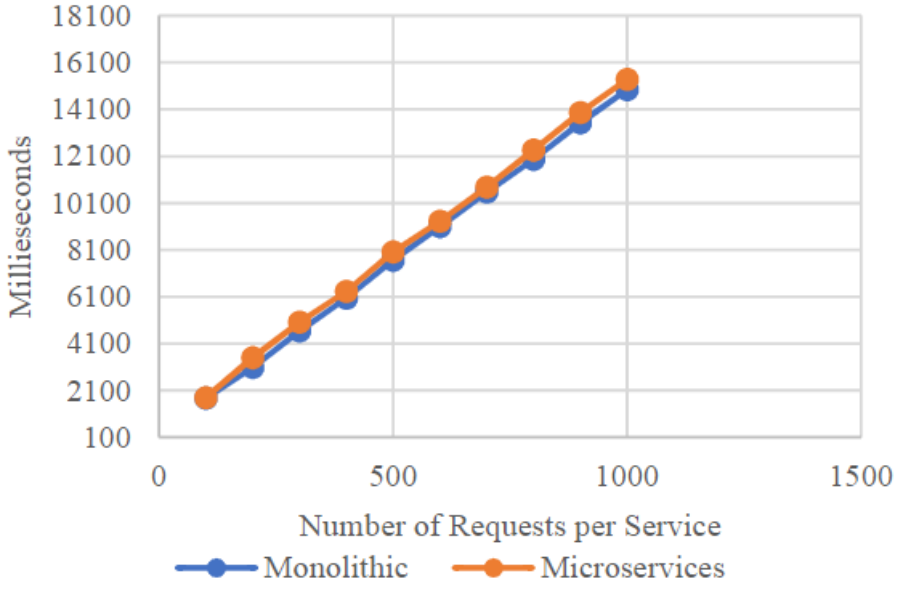


FIGURE 2.11: HTTP Requests Results from [5]

For the second, the same number of requests are realised on the architectures and compared the performance. On this graphic, the advantage of using a microservices architecture is more perceptible due to the lower time on execution.

2.5 REST API and GraphQL

When the subject is to create a solution, the query language to use over HTTP is quite a demand question. The discussion in the last years, with GraphQL growth, is to

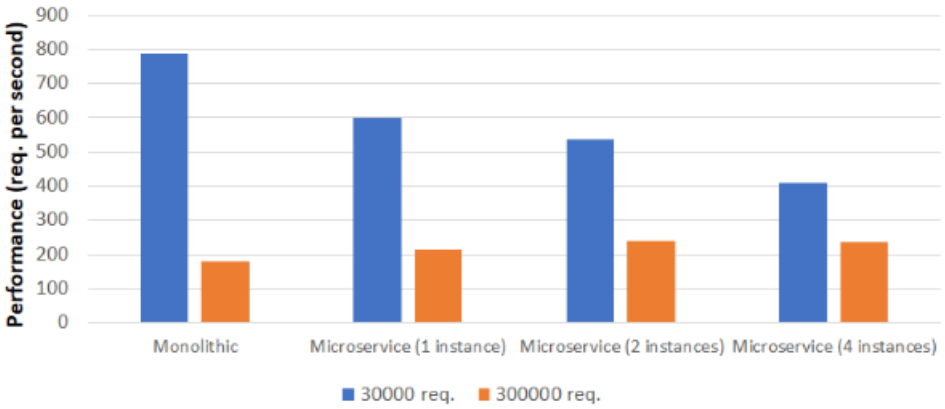


FIGURE 2.12: Requests per second from [27]

understand the best approach and, if possible, to use both together. This analysis is a little out of thesis context, so only some characteristics of both technologies and some different types of usage are presented.

REST was the standard web service architectural style for API creation in the last years. But recently, it lost some space with GraphQL appearance, an open-source query language for APIs introduced in 2012 by Facebook. Since then, many other companies have also adopted this language like GitHub, Twitter, Airbnb and others. As was mentioned before, it is possible to use only GraphQL but also can be used with REST [4]. One of the main goals of GraphQL development is to fix problems that are created on solutions that use REST [2]. One of these problems is that "users often require multiple related resources at a time, resulting in multiple round trip requests to fetch each resource" [4]. GraphQL can offer a different approach to deal with these issues, being that users can query all data they need in a single request.

As was mentioned before, GraphQL can provide a lot of improvements compared to REST. GraphQL recent existence does not allow clear conclusions about the efficiency and feasibility of its application on different types of projects.

As for differences between both, it is noteworthy how the data is exposed on GraphQL, using a graph represented by a schema and by contrast, server applications with REST implement a list of endpoints. With the Domain-Specific Language (DSL) used on GraphQL, users can specify what the fields is they demand from serves; on the contrary of REST, queries are defined using endpoints. Each endpoint always returns the same attributes for all requests, which represents a significant advantage for GraphQL usage [2].

The following image shows an example of a REST request to a specific endpoint with a parameter, and it is possible to see the limitations that this type of API can offer.

```
GET /search/repositories?q=stars:>100
```

LISTING 2.2: Example REST Request from [2]

The image below shows the same request presented on the previous image, but now with an implementation on GraphQL.

```
query searchRepos {
  search (query:"stars:>100", first:100, type:REPOSITORY){
      nodes{
      ... on Repository{ nameWithOwner }
      }
    }
  }
```

Listing 2.3: Example GraphQL Request from [2]

As mentioned before, the last request is handled by a schema previously defined with the GraphQL DSL, so the following image shows the schema created by [2] to do his experiment with two differents APIs.

```
  interface Node {
    id: ID!
  }

  type Repository implements Node {
    nameWithOwner: String!
    primaryLanguage: Language!
    ...
  }

  type Language implements Node {
    id: ID!
    name: String!
    color: String
```

Listing 2.4: GraphQL Schema created by [2]

2.6 Business and Innovation Process

This section describe all used methods for the business and innovation process

2.6.1 New Concept Development Model (NCD)

The NCD was introduced by Peter Koen, and it grants a common language and a holistic view of the front end. As is possible to see in the figure 2.13, this model has three principal components that can allow defining the Front End of Innovation (FEI) of a project or product [31]:

- **Engine:** Center of the model which accounts for the vision, strategy and culture which drives the FEI
- **Five activity elements -** the opportunity identification, opportunity analysis, idea generation, and enrichment, idea selection and concept definition
- **External Environment factors:** variables that can influence the activity elements

2.6.2 Analytic Hierarchy Process (AHP)

"AHP is a theory of measurement through pairwise comparisons and relies on the judgements of experts to derive priority scales. It is these scales that measure intangibles in relative terms" [32]. The method divides the decision into the following steps:

1. Define the problem and what is knowledge to be achieved
2. Structure the decision hierarchy from the top with the primary goal of the decision, then the objectives from a broad perspective, through the intermediate levels, based on criteria that influence the subsequent elements, to the lowest level that has a set of alternatives

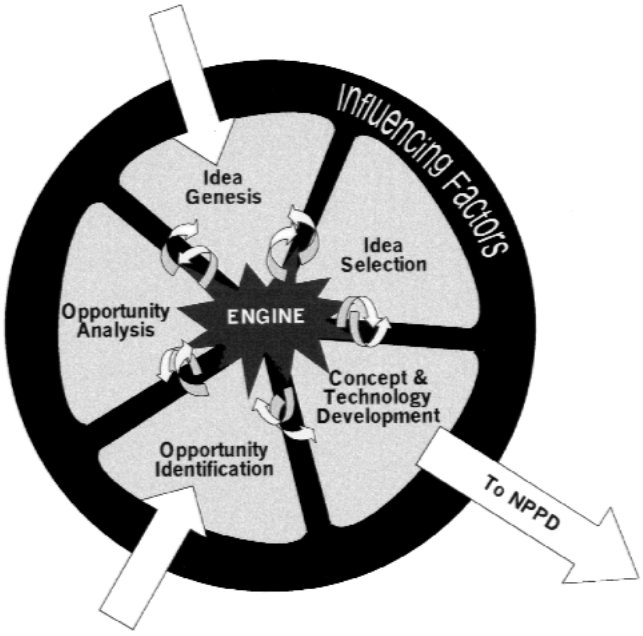


FIGURE 2.13: NCD model from [27]

3. Construct a set of pairwise comparison matrices. Each element is used to compare the elements on the below level
4. Use the priorities obtained from the last step to weigh the priorities in the level immediately below. For each element is obtained overall or global priority, and this process continues up until the last level

2.6.3 Function Analysis System Technique (FAST)

FAST is a technique that allows a graphical representation of functions of a project and tries to answer to three main questions:[1]

- How can achieve this function?
- Why do you do this function?
- When you do this function, what other functions must you do?

The development of this diagram can be beneficial to teams due to can identify some missing functions and create a simple and organized view of the product. The image 2.14 presents a resume than it is possible to represent on this type of diagram.

2.6.4 Business Model Canvas

The Business Model Canvas is a useful model that must be used to create a value proposition for a product [34]. This model tries to answer the following questions:

- Who are the clients?
- What do they are interested in?
- How can we reach them?
- What skills do we need?

[1] [33]

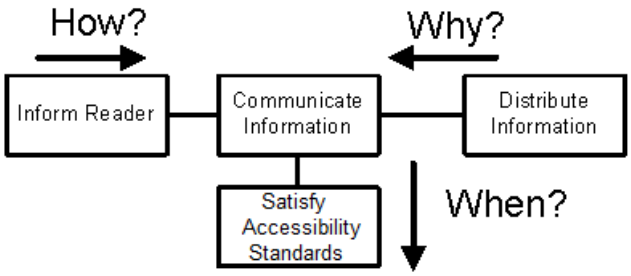


FIGURE 2.14: FAST diagram [33]

This model is divided into the following areas:

- **Key partners:** Definition of who are the key patterns and what are their motivations
- **Key activities:** Activities that have more importance to the value proposition and to the others segments of this model
- **Value proposition:** Definition of the value that will be created and delivered to the customer
- **Customer relationship:** Customer relationship expectation to the business
- **Customer segment:** Definition of the customers and their interests
- **Key resource:** Most important resources for different areas
- **Distribution channel:** Channels that will be used to reach customers
- **Cost structure:** Definition of main costs for the business
- **Revenue stream:** Definition of the way that the customers will pay for the solution and how this payments contributed to the overall revenues

2.7 Technological Stack

On this subchapter will be explained the two technologies integrated in the solution and the main language used to develop the prototype.

2.7.1 Java

According to Oracle documentation, Java is the number one programming language and development platform [35]. For this reason, this language provides a lot of accessible features to develop different solutions. Beyond this, Java can also be integrated with external plugins or dependencies through the build automation tool that is used. For this project, Spring and Spring Boot, was two of this external sources that was very helpful. It provides an easy way to create stand-alone and production-grade Spring based applications. Spring Boot also provides embedded Tomcat, Jetty or Undertow directly, without the need to deploy WAR files and an automatic configuration of Spring and their third party libraries, whenever possible.

On the image 2.15, it is possible to see different modules of spring boot, and one of this layers are tools that can be used to facilitate the integration with other services or development tools. For this project, the most useful tool to develop a Spring based application with GraphQL API is Java GraphQL. On image A.1 it is possible to see a distribution of microservices that are registered on a discovery service, like Netflix.

Each one of this microservices provides data on a GraphQL API that can be consumed by external clients.

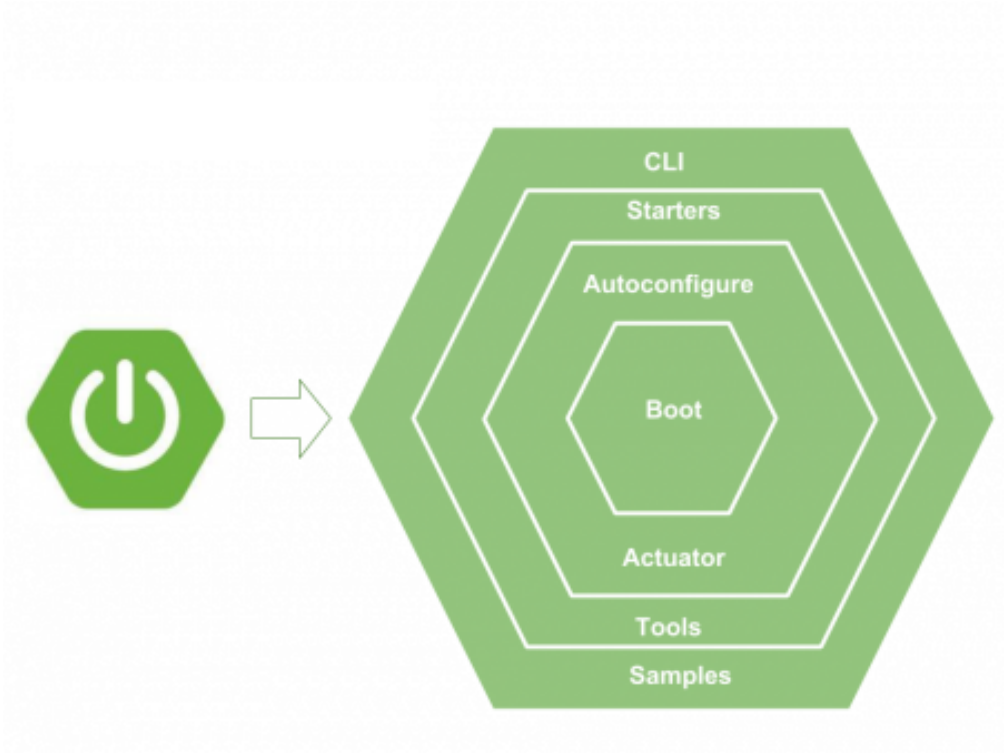


FIGURE 2.15: Spring Boot Modules [36]

2.7.2 GitInspector

GitInspector is an open-source tool that a provides features to generate statistical reports for Git repositories. The default analysis can produce reports with statistics for all authors of the repository, but with the appliance of some filters is possible to reduce the data generated. Can be applied filters per author, date, file extensions and others. This tool was originally written to help fetch repository statistics from student projects in the course Object-oriented Programming Project (TDA367/DIT211) at Chalmers University of Technology and Gothenburg University [38].

The generated reports can be in four different formats: HTML, JSON, Extensible Markup Language (XML) or plain text output. The image 2.17 shows a HTML report generated by GitInspector. It is possible to see different authors, statistical values for some metrics, like commits, insertions, deletions and also the extension files that was found on the repository.

The excerpt of code 2.5 shows an example of a filter that can be applied to GitInspector execution. In this case, it will be provided information only about author John Smith.

```
gitinspector -x "author:^(?!(John Smith))"
```

LISTING 2.5: GitInspector Run to Generate Report for one author

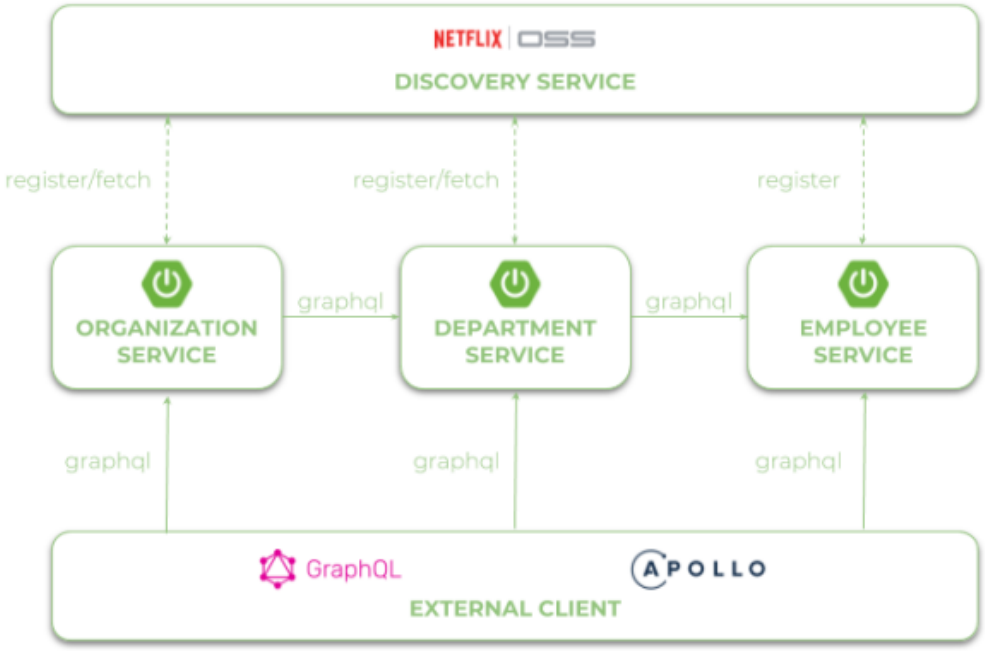


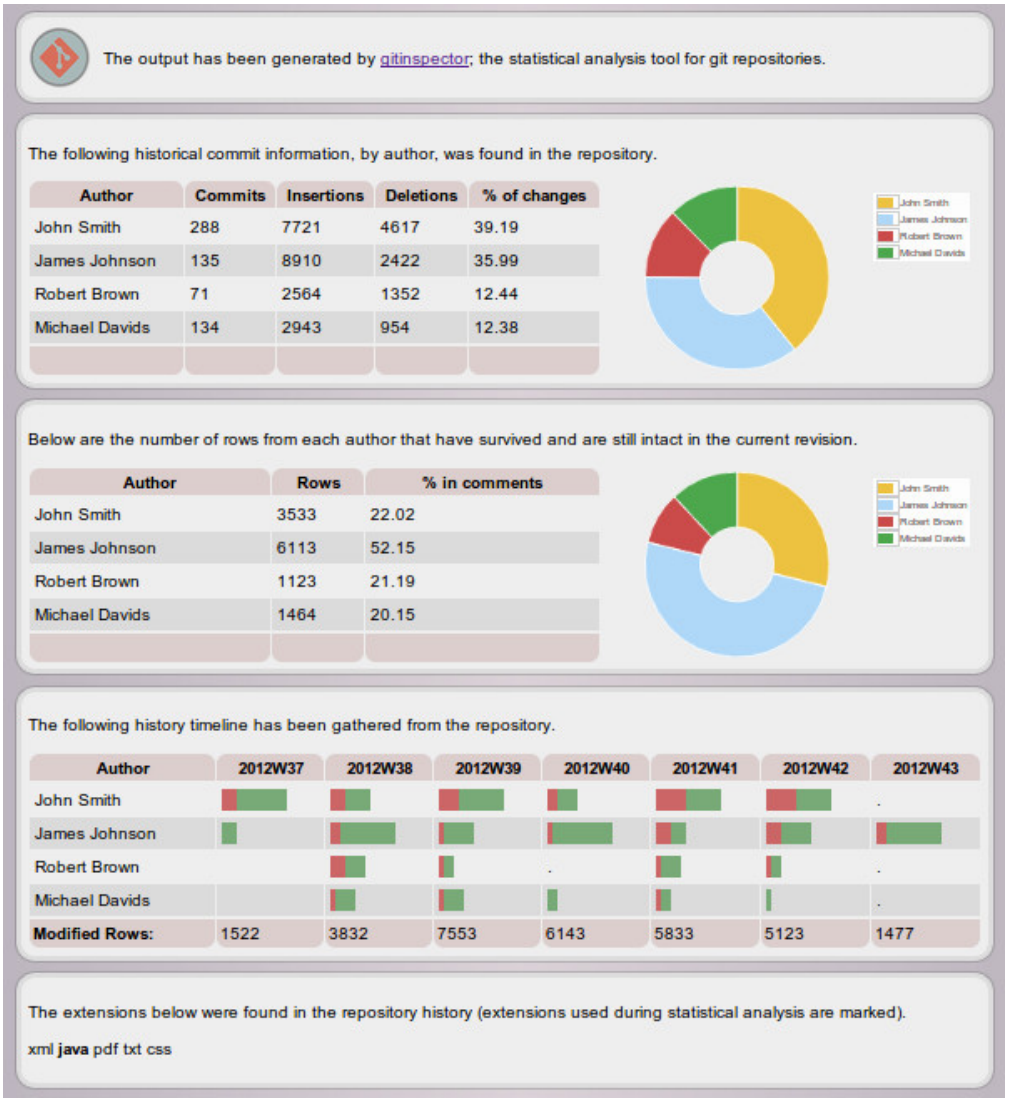
Figure 2.16: GraphQL Usage [37]

2.7.3 SonarQube

SonarQube empowers all developers to write cleaner and safer code [2]. The main goals of Sonarqube are to ensure code quality and security. It also provides support for several languages, such as Java, Kotlin, TypeScript, among others.

During the execution of an analysis of a repository, several metrics are evaluated that allow later to classify the repository, as is the case of the technical debt, which allows evaluating the maintainability of the code. The figure 2.18 shows some of the metrics evaluated by the tool and also the exact location of problems in the code.

[2]https://www.sonarqube.org/



The output has been generated by gitinspector: the statistical analysis tool for git repositories.

The following historical commit information, by author, was found in the repository.

Author	Commits	Insertions	Deletions	% of changes
John Smith	288	7721	4617	39.19
James Johnson	135	8910	2422	35.99
Robert Brown	71	2564	1352	12.44
Michael Davids	134	2943	954	12.38

Below are the number of rows from each author that have survived and are still intact in the current revision.

Author	Rows	% in comments
John Smith	3533	22.02
James Johnson	6113	52.15
Robert Brown	1123	21.19
Michael Davids	1464	20.15

The following history timeline has been gathered from the repository.

Author	2012W37	2012W38	2012W39	2012W40	2012W41	2012W42	2012W43
John Smith							.
James Johnson							
Robert Brown				.			.
Michael Davids							.
Modified Rows:	1522	3832	7553	6143	5833	5123	1477

The extensions below were found in the repository history (extensions used during statistical analysis are marked).

xml **java** pdf txt css

FIGURE 2.17: HTML GitInspector Report [38]

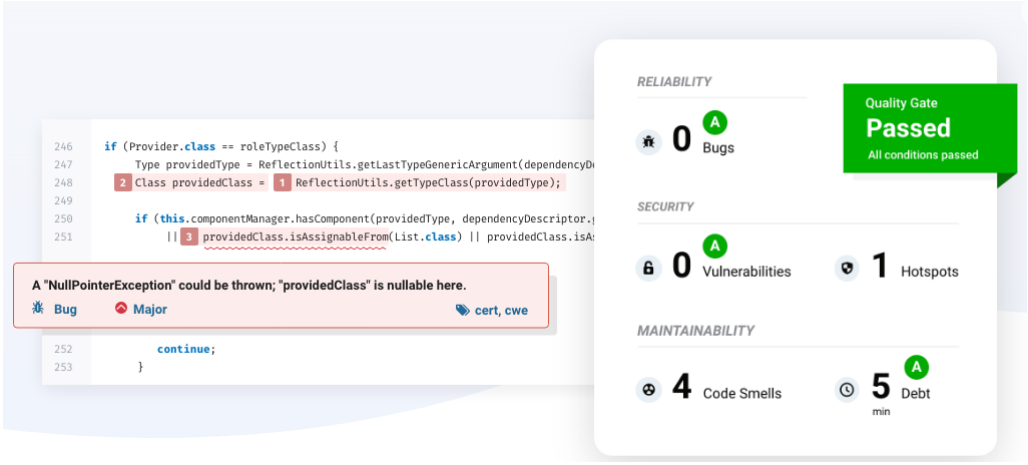


FIGURE 2.18: SonarQube Execution

Chapter 3

Analysis

The value analysis is an approach that have as the main goal analyze and improve the value of a project or product, considering its components and respective costs. The requirements engineering is also part of the analysis process.

This chapter describes the business value for the project using NCD model, and still the value offer and proposition and the canvas business model. The engineering requirements are also described, including the domain model and functional and non-functional requirements.

3.1 Business Value

On this sub chapter, will be used the NCD model, explained on section 2.6.1, to describe the business value for the project.

3.1.1 Business Perspective

Using Peter Koen's model, this project has the following five activity elements:

Opportunity identification

With new solution creation to analyze git repositories and with the possibility to be integrated into existing git providers, like Bitbucket, there is a need to guarantee that the final consumer of data can obtain as much data as possible. The product should have features of different tools, and it has to be flexible to be capable of satisfying every consumer.

One of the approaches that can be used to identify opportunities is to analyze technology trends, verify that used technology has a crescent growth, and guarantee that the product does not become outdated in a short deadline.

Opportunity analysis

For this project, the opportunity analysis occurs in an academic context, with the requirement to create a solution that can be useful to many teams in different contexts, for example, by a college teacher who wants to check each student's contribution. He wants a deep analysis of the repositories that can allow him to analyze data easier or a team leader who wants to know if his team has the expected productivity and effort.

The main questions are:

- How relevant is creating an integration solution for different tools?
- Can this solution be so independent of the technologies used over GIT?
- Can this solution be integrated with all existing GIT providers?
- The integration solution can has a good performance with the integration of some tools?

After the opportunity identification, it is essential to analyze the questions generated over the identification already done before. Furthermore, for this analysis, the methods described for the opportunity identification can be used for this key element to predict the project's outcomes and expectations.

Ideia generation and enrichment

The solution performance can be guaranteed with the usage of a tool that can provide a fast answer to all of the requests that can be made, that can be a REST solution, creating all endpoints for any request, or the GraphQL usage, that allows the consumer to specify only required data. Another question to take care of is the scalability, and for this, the implementation must be analysed using a monolithic or microservices architecture and evaluate the option that can give more business value to the product.

The questions created on the opportunity analysis are a significant help to the idea generation, and enrichment, because can be created options that have to be evaluated on the key element related to ideia selection.

Ideia selection To choose what is the best idea to develop and increase the business value for the project, it was used the AHP method described in chapter 2.6.2. Given the context of the project, the decision to choose this method was based on the approach to make the business value for the product, due to it is a new solution, and the value should be analyzed taking into account the benefits and increases related to the other existing solutions. First of all, have to be different explicit attributes chosen to make the comparison between all alternatives:

- **Performance:** the processing capacity to all requests;
- **Efficiency:** the capacity to take care of all solution's requirements with less resources and errors as possible;
- **Scalability:** how much this solution can improve and be receptive to the change in the future;
- **Costs:** how much cost the implementation of this solution in terms of time, formation, licenses and specialized hardware.

The main question that have to be answered is:

- Which is the best architecture and API to implement a git repositories analysis to guarantee the maximum bussiness value possible comparing to the existing solutions?

During the development of a new solution, many decisions have to be done, and for the implementation of this multi-criteria method, was chosen the decision about the architecture and API, due to be the decision that could impact more the business value for the project.

Based on the data presented on state of art (chapter 2), was created the following hypoteses:

1. Monolithic Architecture with REST
2. Microservices Architecture with REST
3. Microservices Architecture with GraphQL

After that, have to be defined the priorities among each attribute, where it was classified the performance, followed by efficiency, followed by scalability and the costs in the end. The scale of comparison is created by [32], and is presented on table 3.1.

The first step is to assign the respective values for the importance for each attribute defined before, that is described on the table 3.2.

Table 3.1: Scale of absolute numbers, adapted [32]

Importance	**Definition**
1	Equal Importance
3	Moderate importance
5	Strong importance
7	Very strong
9	Extreme importance
2,4,6,8	Intermediate values

Table 3.2: Comparison matrix of ideia selection

	Performance	**Efficiency**	**Scalability**	**Costs**
Performance	1	6	2	4
Efficiency	1/6	1	1/4	2
Scalability	1/2	4	1	2
Costs	1/4	1/2	1/2	1

The next is to normalize the matrix presented on table 3.2 and calculate the respective relative priority of each attribute, that is described on tables 3.3 and 3.4

Table 3.3: Comparison matrix of ideia selection with sum

	Performance	**Efficiency**	**Scalability**	**Costs**
Performance	1	6	2	4
Efficiency	1/6	1	1/4	2
Scalability	1/2	4	1	2
Costs	1/4	1/2	1/2	1
Sum	23/12	23/2	15/4	9

The next step is to evaluate the relative priorities consistency calculated on table 3.4. To obtain consistency value, is necessary to calculate first the largest eigenvalue of the matrix that can be obtained using the equation 3.1. The equation 3.2 presents the application of equation 3.1.

$$Ax = (\lambda max)x \tag{3.1}$$

$$\lambda max = \frac{23}{12} \times (0,4860) + \frac{23}{2} \times (0,1125) + \frac{15}{4} \times (0,2615) + 9 \times 0,1030 = 4,1029 \tag{3.2}$$

The next step is to calculate the consistency index (IC). The calculation is obtained using the equation 3.3 and is presented on equation 3.4.

$$\text{IC} = \frac{\lambda max - n}{n - 1} \tag{3.3}$$

$$\text{IC} = \frac{4,1029 - 4}{4 - 1} = 0,0343 \tag{3.4}$$

Table 3.4: Comparison matrix of idea selection normalized with a relative priority

	Performance	**Efficiency**	**Scalability**	**Costs**	**Relative Priority**
Performance	12/23	4/9	8/15	4/9	0,4860
Efficiency	2/23	2/27	1/15	2/9	0,1125
Scalability	6/23	8/27	4/15	2/9	0,2615
Costs	3/23	1/27	2/15	1/9	0,1030

After getting the value for consistency index (IC), the next step is to calculate the reason of consistency (RC), and it can be obtained using the equation 3.5. The IR value is a random value that is calculated to square matrices with order. In this case, the value n is 4, due to the 4 existing criteria and the corresponding value for IR is 0,90. The equation 3.6 introduces the value calculated for RC.

$$\mathrm{RC} = \frac{IC}{IR} \tag{3.5}$$

$$\mathrm{RC} = \frac{0,0343}{0,90} = 0,0381 < 0,1, \text{ so the values are consistent} \tag{3.6}$$

The next phase is the definition of the comparison matrix for each attribute, with each alternative.

Table 3.5: Comparison matrix of idea selection for performance

	Monolithic REST	**Microservices REST**	**Microservices GraphQL**	**Priority Vector**
Monolithic REST	1	1/2	1/6	0,1066
Microservices REST	2	1	1/4	0,1935
Microservices GraphQL	6	4	1	0,6970

Table 3.6: Comparison matrix of idea selection for efficiency

	Monolithic REST	**Microservices REST**	**Microservices GraphQL**	**Priority Vector**
Monolithic REST	1	2	3	0,5390
Microservices REST	1/2	1	2	0,2973
Microservices GraphQL	1/3	1/2	1	0,1638

The last step is to calculate composite priorities of alternatives, and it can be obtained by multiplying previous values for each attribute by each relative priority for each alternative (equation 3.7).

Table 3.7: Comparison matrix of idea selection for scalability

	Monolithic REST	**Microservices REST**	**Microservices GraphQL**	**Priority Vector**
Monolithic REST	1	1/4	1/6	0,0854
Microservices REST	4	1	1/4	0,2435
Microservices GraphQL	6	4	1	0,6711

Table 3.8: Comparison matrix of idea selection for costs

	Monolithic REST	**Microservices REST**	**Microservices GraphQL**	**Priority Vector**
Monolithic REST	1	1/2	1/3	0,1636
Microservices REST	2	1	1/2	0,2973
Microservices GraphQL	3	2	1	0,5390

The idea selection was to explore GraphQL with a microservices architecture, which allows creating a solution with good performance and scalability.

This crucial element analysis can also be considered the usage of probability methods based on the technical success of used tools and technologies.

$$\begin{pmatrix} 0,1066 & 0,5390 & 0,0854 & 0,1636 \\ 0,1935 & 0,2973 & 0,2435 & 0,2973 \\ 0,6970 & 0,1638 & 0,6711 & 0,5390 \end{pmatrix} \times \begin{pmatrix} 0,4860 \\ 0,1125 \\ 0,2615 \\ 0,1030 \end{pmatrix} = \begin{pmatrix} 0,1516 \\ 0,2218 \\ 0,5882 \end{pmatrix} \tag{3.7}$$

Concept definition This last key element will be found as a solution that uses a microservices architecture based on GraphQL to provide all the communications to the consumers and between components inside the product.

The method that can also be used to analyze this key element can be the analysis of performance capability limit of the technology.

3.1.2 Value Offer and Proposition

For the study of value offer and proposition will be used FAST model (section 2.6.3), due to the value analysis of this product being based on the increased value when compared to existing products, and it not based adequately on software requirements. The image 3.1 presents the FAST diagram that represents all order functions of the product, that on the left is the higher order function and on right the lowest order functions.

After the functional analysis, have to be evaluated the business value for the product. And for this, the business model canvas, described on section 2.6.4, is a good model to describe all the interactions, needs and all the interested parts on the business. The image 3.2 describes the application of the model.

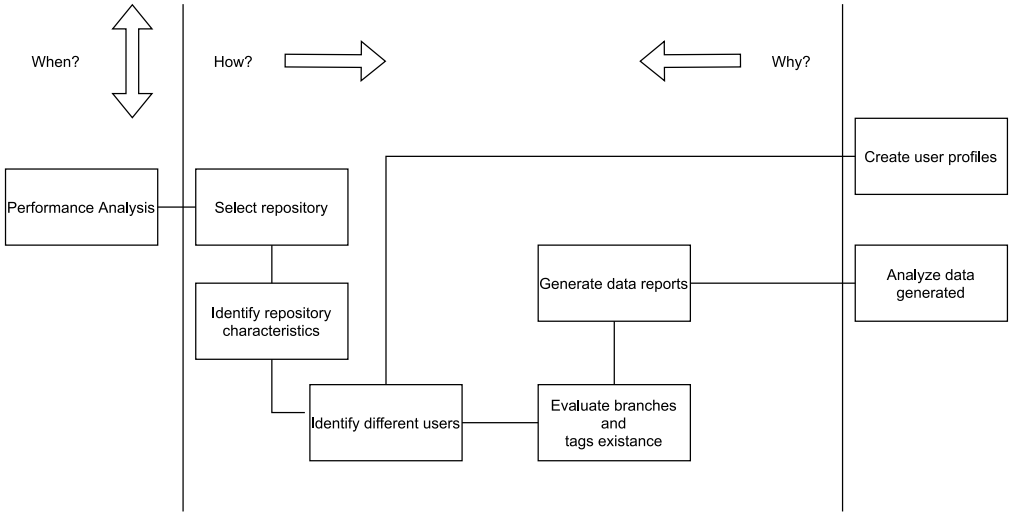


FIGURE 3.1: FAST model

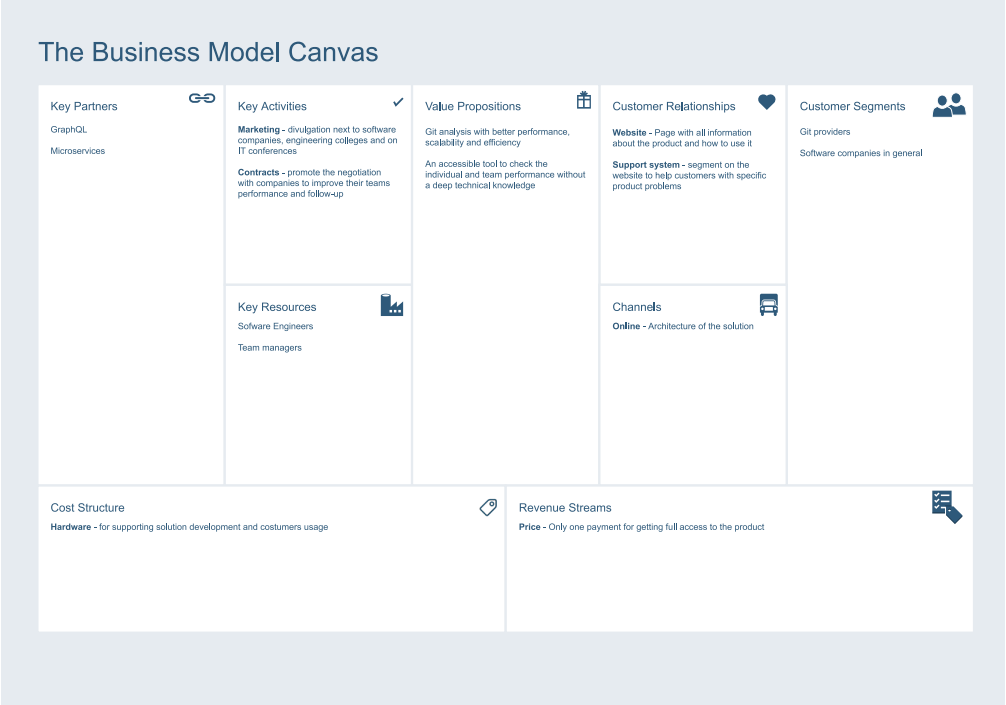


FIGURE 3.2: Business Model Canvas

The project developed has two customer segments: git providers and software companies in general.

The customers expected that the documentation of this solution would be on a website, and it could answer all of their doubts. Besides that, customers expected to have a support system that can solve all the technical problems that could appear with the usage of the product.

Some marketing should be made in software companies, engineering colleges, and IT conferences to reach the customers.

The key partners are GraphQL and microservices because these two concepts are responsible for the increased value for this product compared to the existing ones. The solution's features also add some value to the product, but communication and architecture are the essential concepts that have to be considered when the subject is value analysis for this product, like was described in section 3.1.1.

The main cost structure is that the hardware needs to develop and provided customers to use.

This product can be provided with full access by one payment.

3.2 Requirements Engineering

"Requirements engineering is the process of eliciting stakeholder needs and desires and developing them into an agreed-upon set of detailed requirements that can serve as a basis for all subsequent development activities" [39]. For this project, the approach chosen to develop requirements engineering is to understand the concepts that can represent the implementation of a domain class. After these identified concepts, the functional and non-functional requirements that are the specification of stakeholders and the needs and interactions they want to do with the system are described.

3.2.1 Functional Requirements

Before defining funcional requirements, it is essential to understand what is the stakeholders and users that have necessities and requirements that must be present on the system.

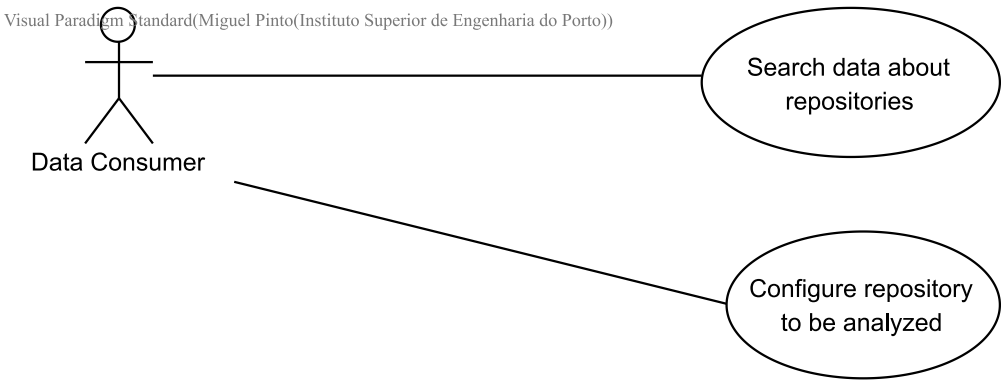


Figure 3.3: Solution Use Case

On the image 3.3, it is possible to see the only user that wants to use the system to analyze data about his repositories. The requirements that the user wants to be on the system are described on table 3.9.

Table 3.9: Functional Requirements

Identification	Description
FR01	The user shall be able to search data about repositories
FR01.1	The user shall be able to search data about each author on a repository
FR01.2	The user shall be able to analyze all contributions to a repository in a timeline
FR01.3	The user shall be able to see how many files exist on a repository and all changes that all authors did
FR01.4	The user should be able to see what is the code coverage in a day
FR01.5	The user should be able to know the evolution of the project if it improved or got worse during a specific time interval
FR01.6	The user should be able to see all the existing issues on the repository
FR01.7	The user should be able to select the metrics to evaluate the repository
FR01.8	The user should be able to know the existing metrics that can be used to evaluate the repository

3.2.2 Non-Functional Requirements

In addition to functional requirements, other requirements can emerge during the solution's development. For now, it is possible to define some non-functional requirements related to data specification, interfaces and quality attributes.

As a data specification requirement, it is possible to define that data access as easy and personalized for each user.

As interfaces requirements, data is requested via GraphQL queries.

The quality attributes end up defining non-functional requirements that can not be included in functional requirements, and it is possible to define the following ones:

- **Performance:** the developed solution shall answer each request with less time as possible, given the possibility of a large amount of data to be analyzed.
- **Usability:** the solution must have an easy interface to use, given the possible lack of technical knowledge of users
- **Interoperability:** the solution shall be able to be consumed by other systems via an API provided by GraphQL

Table 3.10: Non-Functional Requirements

Identification	Title	Description
NFR01	Performance	The developed solution shall answer each request with less time as possible, given the possibility of a large amount of data to be analyzed
NFR02	Usability	The solution must have an easy interface to use, given the possible lack of technical knowledge of users
NFR03	Interoperability	The solution shall be able to be consumed by other systems via an API provided by GraphQL

3.2.3 Domain Model

After defining the functional and non-functional requirements, the domain model is the next phase of the analysis process. The image 3.4 presents all the concepts that try to pass the business logic to the implementation of the product.

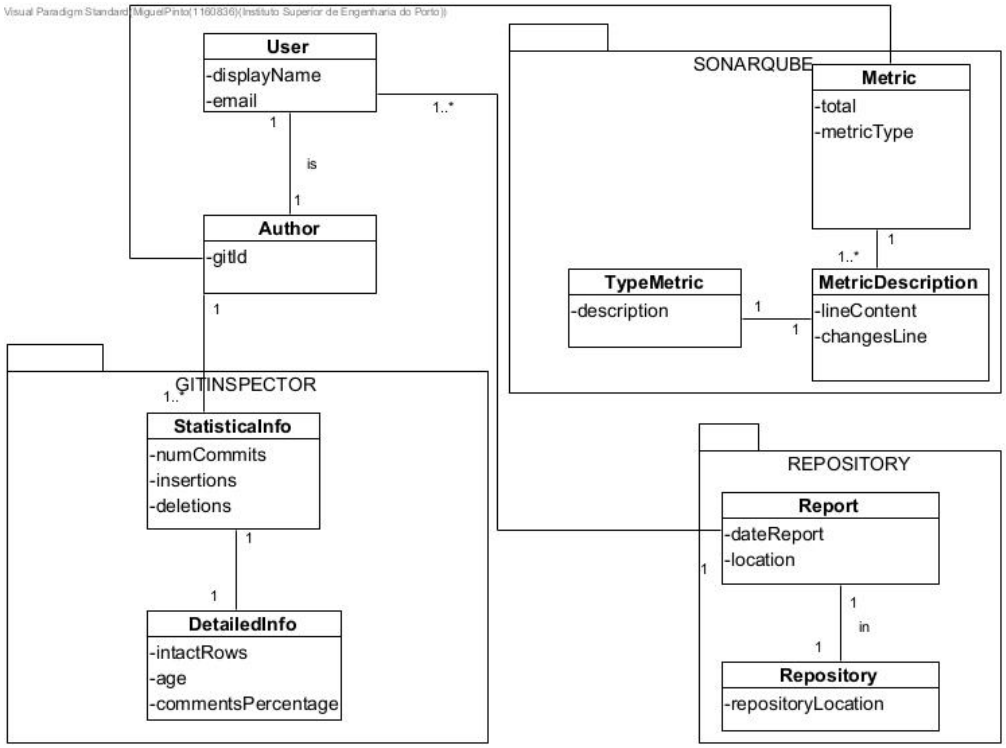


FIGURE 3.4: Domain Model

For the development of this model, was chosen an approach of Domain Driven Design (DDD), which is a popular model-driven methodology for capturing domain knowledge to help on design and analysis creation to a solution. Once the result of idea selection realized with AHP method help, on section 3.1.1, was a microservices solution with GraphQL, it is a significant advantage using this approach with microservices, due to DDD can provide resources to decomposite domain concepts into contexts, on what context aggregates coherent concepts . Each one of these contexts provides a distinct business capability. The attributes defined on a conceptual model may not match to the attributes created on the implementation model.

The first step is to define existing bounded contexts [1]. There are some different bounded contexts:

- **SonarQube:** responsible for processing and saving from SonarQube service, like metric, metric type, metric description and some information about the author
- **GitInspector:** responsible for processing all data from GitInspector service, like number of commits, insertions done by an author and a percentage of insertions which are in comments
- **Repository:** responsible for processing and saving all information about the repository, like the git provider host, location and report date

[1] limit inside the domain where a particular domain model and business logic is applied

Each of these bounded contexts will take a different microservice that will have a specific implementation and be independent of other microservices on the solution's architecture.

3.3 Summary

This chapter presents the value analysis for the product, using the NCD model that belongs to FEI, the FAST model and Canvas business model to the value proposition.

The domain model for the solution is presented for the requirements analysis and engineering requirements, followed by functional and non-functional requirements definitions. The quality attributes are defined together with the non-functional requirements.

Chapter 4

Design

In this chapter, it will be presented all documentation about the design of the solution, that includes architecture diagrams with different view levels and diagrams that explains functional requirements, like sequence and class diagrams.

4.1 Architecture

For the architecture design, it will be presented different views of the system, based on model view $4 + 1$, that was originally introduced by Philippe Kruchten in 1995 as a unique method that assembles the five sorts of views to address the problems of software architecture [40].

4.1.1 Logical View

Logical view is concerned with the functionality of the system as it relates to end users. One of the diagrams that is able to present this view is the class diagram, that is which can be found in the annex A. This diagram demonstrates the application of the domain model described in section 3.2.3. The business concepts created in the domain model are often translated into classes during implementation, which does not always happen due to some adjustments that have to be made during the implementation. Some of these adjustments may be motivated by language limitations or even code optimization issues.

4.1.2 Process View

The process view is focused on the behavior of the system during execution. In this view it should be possible to observe the interaction between different classes and also some specifics of the system. One of the diagrams capable of describing a process view is the sequence diagram. The sequence diagram presented on image 4.1 shows some of the requirements defined in section 3.2.1, such as creating a GraphQL API and also integrating external services.

It is also possible to observe the application of some non-functional requirements described in the section 3.2.2, as is the case of the usability that is achieved through the GraphQL API, since it allows the customization of the service responses according to the needs of the user.

4.1.3 Development View

The development view is intended to demonstrate the point of view of the programmer and a system administration view. This view is very focused on showing the system components, so the best diagram to show this view is the component diagram.

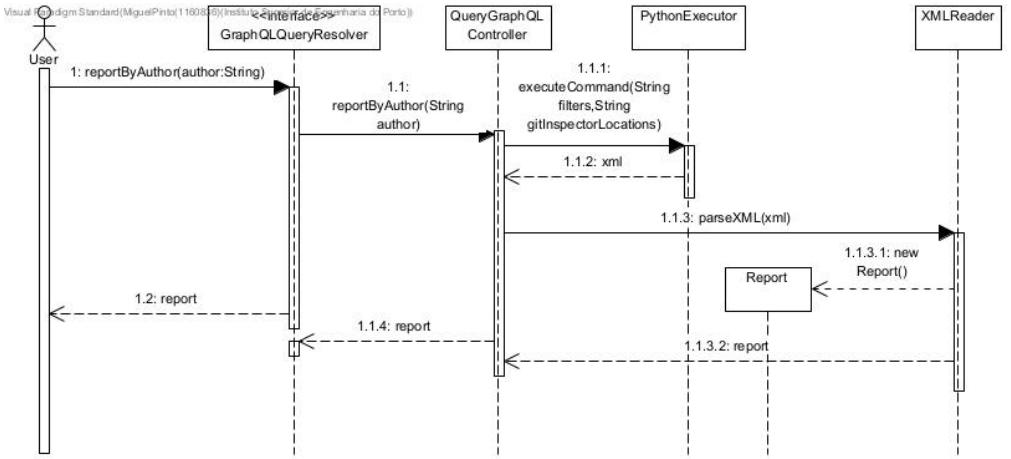


FIGURE 4.1: Architecture Process View

The image 4.2 presents the different components in the system, and allows the validation of one more of the requirements defined in the section 3.2.1, which is the creation of a solution based on a microservices-oriented architecture.

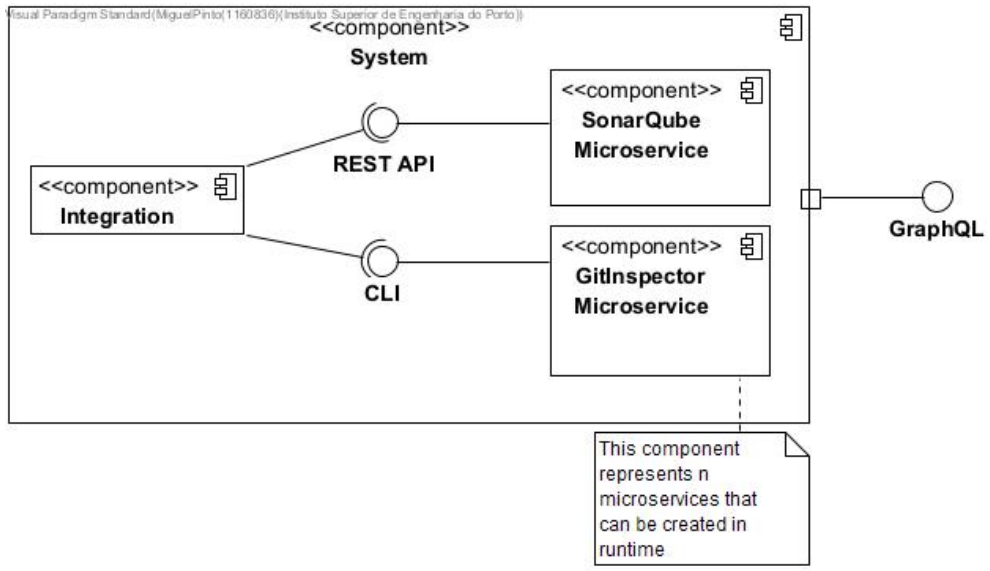


FIGURE 4.2: Architecture Logical View

4.1.4 Physical View

Regarding the physical view, it was not possible to create any diagram showing this view, since none of the existing components had any kind of deployment to an external server. All implementation and tests were performed on a local machine, which does not justify the creation of a deployment diagram.

4.2 Design Alternatives

The image 4.3 represents an alternative to the design presented on image 4.1. The difference between the two diagrams is the API chosen for developing the solution. The approach using GraphQL was chosen for this thesis, but the alternative proposed using REST was one of the hypotheses evaluated in the section 3.1.1. It was concluded

that the solution with GraphQL was the one with the most interest and the best value proposition for the business.

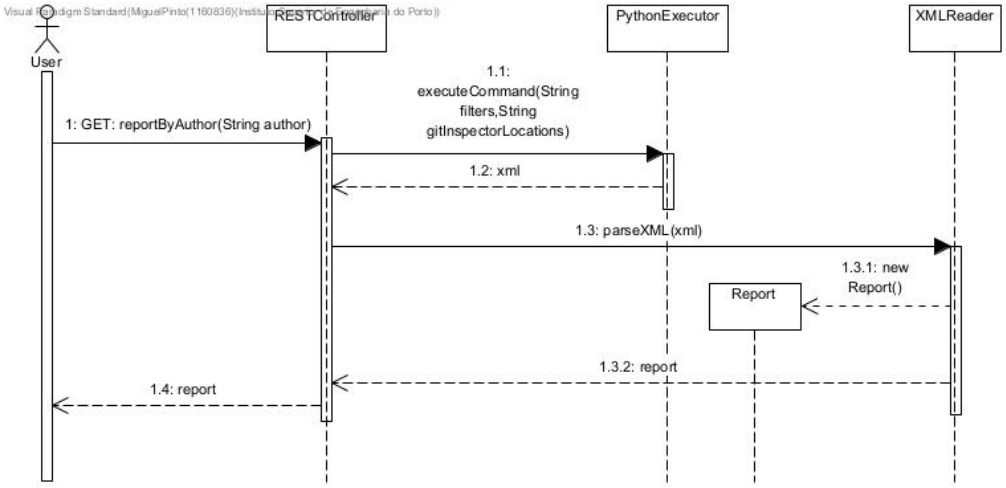


FIGURE 4.3: Design Alternative

4.3 Summary

This chapter presents the entire design created to document the created solution, always trying to associate the respective functional and non-functional requirements that it intends to respond to.

Chapter 5

Implementation

At this chapter, it explains some details about the implementation of the solution described on chapter 4. The development of this prototype was focused on the back-end, specifically on the way how to create and provide data from repositories to other systems. The data reports was created with the integration of two different tools, SonarQube and GitInspector. The main focus of development was to create an GraphQL API to provide data from tools integrated in the system. This API can be consumed by external systems.

All of the code developed for this project is available on a public repository [1].

5.1 General Implementation

The approach to create an integration with GitInspector starts with the doubt how to run commands on a service that doesn't provides an API to consume. On this service, the execution stops with the generation of a report, so for each request made by the solution developed, have to be launched a new instance execution of GitInspector.

For GitInspector execution, the repository location has to be defined and be passed as one of the command line parameters. To allow this configuration on an easy way, was developed a query on GraphQL that allows to set the runtime value of a property (named gitInspectorLocations), that represents the path for the repository.

```
type Query {
    configureRepository(location:String):String
    (...)
}
```

Listing 5.1: Query for Repository Location

Also for SonarQube execution, the system needs to know where is the service is running. The property sonarQubeLocaions represents the location of this service and where the Web API is provided. There is a difference between two services, that is SonarQube provides a very well documented API, and represents a better way to obtain all data from analysis.

Due to dependency of GraphQL Java, it was possible to implements the interface GraphQLQueryResolver, that allows to map the queries defined on GraphQL Schema to java methods declared inside a Spring Component [2]. For the query configureRepository exists a Java method with exactly the same name, and have to be done at this way to avoid errors on GraphQL schema parser. The excerpt of code 5.2 shows the method created to check if the variable of repository location is updated correctly during runtime. Since Spring boot is being used, have to be updated all variables of repositoryLocation

[1] https://github.com/1160836/tools-git-integration

[2] Component is a class-level annotation. It is used to denote a class as a Component. can be used @Component across the application to mark the beans as Spring's managed components [38].

for all beans [3], that are always loaded when the application starts. This method only works if the repository already exist locally. If the repository is not available locally, it was created a method that allows to clone a remote repository, only available at the moment for BitBucket repositories. This method is presented on excerpt of code 5.3.

```
public String configureLocalRepository(String location){
    setRepositoryLocation(location);
    PythonExecutor pythonExec = (PythonExecutor)appContext.getBean("
pythonExecutor");
    pythonExec.setRepositoryLocation(location);
    if(!pythonExec.getRepositoryLocation().equals(location) || !
repositoryLocation.equals(location)) return "Error configuring
repository location";
    return "Repository configured with success!";
}
```

Listing 5.2: Java Method Repository

```
public String getRepositoryBitBucket(String username,String
appPassword,String repository,String projectName){
    String path = null;
    try{
        path = "git clone " + "https://" + username + ":" +
appPassword + "@" + repository;
        Runtime.getRuntime().exec(path);
    } catch (IOException e) {
        return e.getMessage();
    }
    return configureLocalRepository(projectName);
}
```

Listing 5.3: Configuration Remote Repository

5.2 GitInspector Integration

GitInspector was developed in Python, so for execute the service Python has to be installed and declared as an environment variable of the system. The excerpt of code 5.4 shows a method developed to execute commands against the GitInspector. The parameter filters is to define what is the filters that will be applied to generated results, and parameter gitInspectorLocation indicates where is the executable of the service. This method returns a String that contains the response in XML from GitInspector execution. For this information can be made available in GraphQL, it was necessary to develop a mapping to a domain classes, that is described also with domain model on subchapter 3.4.

[3] A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container [38]

```
    public String executeCommand(String filters, String
    gitInspectorLocation) {
        if(repositoryLocation == null) return "Repository not configured";
        String line = "";
        StringBuilder xml = new StringBuilder();
        try{
        Process p = Runtime.getRuntime().exec("python " +
    gitInspectorLocation + " " + filters + " " + repositoryLocation + " --
    format=xml");
         BufferedReader is =
                        new BufferedReader(new InputStreamReader(p.
    getInputStream()));
            while ((line = is.readLine()) != null)
            xml.append(line).append("\n");
        } catch(IOException e){
            System.out.println("Error reading file!");
        }

        return xml.toString();
    }
}
```

Listing 5.4: Python Executor

The method developed is on appendix B and it returns a list of reports, that are represents a domain class named Report. For GitInspector data, was created the input types on GraphQL schema described on excerpt code 5.5.

```
type AdditionalInfo{
    intactRows:Int
    age:Float
    stability:Float
    commentsPercentage:Float
}


type Author {
    authorName:String
    numCommits:Int
    insertions:Int
    deletions:Int
    changesPercentage:Float
    addInfo:AdditionalInfo
}


type Report{
    dateReport: String
    repository:String
    author:Author
}
```

Listing 5.5: Types for GitInspector Data

The type Report can provide some information about the report in general, like the date of generated report, the repository analyzed and creates a reference to the type Author. This type has some attributes that have the following specifications:

authorName: describes the author name of a user that interacts with the repository

numCommits: describes the number of commits of the author described on field authorName

insertions: describes all the insertions made by the author on the repository

deletions: describes all the deletions made by the author on the repository

changesPercentage: describes in percentage the changes made by the author on the current revision

The type Author has also a reference to another type, AdditionalInfo, that has the following specifications:

intactRows: describes the number of rows without any change since last revision

age: describes the average age in months of all the rows that the author has written

stability: describes the percentage of all the code written by the author who survived until now

commentsPercentage: describes the number of the rows that can be found on comments

The type Report is the return type of some queries developed to interact with GitInspector service. These queries are presented on excerpt code 5.6.

```
type Query{
  configureLocalRepository(location:String):String
    getRepositoryBitBucket(username:String!,appPassword:String!,repository
    :String!,repositoryName:String!):String
    reportByAuthor(author:String):Report
    reportAllAuthors:[Report]
    reportAllAuthorsMultipleInstances(authors:[String]):[Report]
    reportByAuthorTimeInterval(author:String,startDate:String,endDate:
    String):Report
    reportAllAuthorTimeInterval(startDate:String,endDate:String):[Report]
  (...)
}
```

Listing 5.6: Queries for GitInspector Service

All of the queries returns the same object Report, the only difference is that can be a single object or multiple objects. There are queries that refers to only one author that returns a single object, with data related to the author activity in the repository. The other queries are related to all authors and returns a list of objects that contains information about all authors with activity on the repository.

These queries have a similar implementation, due to the response of the service has always the same format. As it is possible to observe on code 5.6, there are some differences on the name of the queries. The query reportByAuthor only applies a filter to obtain a report from one author. While the query reportByAuthorTimeInterval have a parameter to filter by one author, but also has parameters to filter the result by a time interval.

As described throughout the document, the architecture used to develop this solution was a microservices architecture. The application of microservices to this project can be explained with the usage of different external services that interacts directly with the system. For the queries explained above, only two microservices are used, one instance of GitInspector and one instance of SonarQube. During the project, was evaluated the chance to throw different instances of these services in runtime. On the chapter 6 will be evaluated the usage of this Map-Reduce [4] feature comparing to the usage of a single instance.

The method created to throw multiple instances of the services can receive different of authors. For each author, it is created a different thread that throws a new instance of the services to process the information about each author. At the end of the method, all threads have to be joined to show the information of all authors at the same time.

[4] Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results

In other words, a synchronous mechanism had to be created to avoid the end of the GraphQL request without all information.

5.3 SonarQube Integration

SonarQube provides a REST API to interact with the system. To ensure the correct mapping between the response objects and the solution developed, there has to be Java classes that can represents all of these objects. As a GraphQL API is being developed, the return of GraphQL queries will be always a GraphQL type. Therefore, the types described on 5.9 were created:

```
type Measure{
    component:Component
    metrics:[MetricDescription]
}
type MeasureDescription{
    metric:String
    value:String
    period:Period
}
type Period{
    value:String
    bestValue:Boolean
}
type Metric{
    total:Int
    p:Int
    ps:Int
    metrics:[MetricDescription]
}
type Issue{
    paging:Paging
    issues:[IssueDescription]
    components:[Component]
    rules:[Rule]
    users:[User]
}
type Paging{
    pageIndex:Int
    pageSize:Int
    total:Int
}
type User{
    login:String
    name:String
    active:Boolean
    avatar:String
}
type Rule {
    key:String
    name:String
    status:String
    lang:String
    langName:String
}
type Component{
    key:String
    enabled:Boolean
    qualifier:String
    name:String
    longName:String
    path:String
    measures:[MeasureDescription]
}
```

Listing 5.7: Types for SonarQube Integration

The type Issue is the return type for method or query getIssuesSonar that has the specification described on code 5.8.

```
type Query {
  getIssuesSonar(author:[String]):Issue
}
```

Listing 5.8: Method for Issues SonarQube

This method can receive a list of authors, that allows to filter the issues only for the authors passed on arguments. On the return type Issue is obtained some information about the paging, represented by the type Paging, that contains information about the page index, the page size and the total of pages. It also has a more detailed type IssueDescription, wich is also part of the GraphQL types, as it is possible to observe on excerpt 5.9.

The attributes of type IssueDescription are described below:

key: unique identifier of the issue

component: can be a portfolio, project, module, directory or file

project: project name

rule: coding rule key with format <repository>:<rule>

status: status of the issue, for example CONFIRMED

resolution: value for resolution, if it present, for example FIXED

severity: severity of the issue, for example CRITICAL

message: hint to fix the issue

line: line code of the issue

hash: hash value of the issue

author: last author that have influence on the code with issue

effort: estimated time needed to fix the issue

creationDate: date of issue creation

updateDate: date of issue update

tags: keywords to help on the search

type: type of the issue, for example CODE_SMELL

comments: comments attached to the issue

```
type IssueDescription{
    key:String
    component:String
    project:String
    rule:String
    status:String
    resolution:String
    severity:String
    message:String
    line:String
    hash:String
    author:String
    effort:String
    creationDate:String
    updateDate:String
    tags:[String]
    type:String
    comments:[Comment]
}
type Comment{
    key:String
    login:String
    htmlText:String
    markdown:String
    updatable:String
    createdAt:String
}
type TypeMetric {
    types:[String]
}
type MetricDescription{
    id:String
    key:String
    name:String
    description:String
    domain:String
    type:String
    direction:String
    qualitative:Boolean
    hidden:Boolean
    custom:Boolean
    higherValuesAreBetter:Boolean
}
```

Listing 5.9: More Types for SonarQube Integration

In addition to this method, it was also developed a method to obtain the existing measures. To obtain the measures, it is mandatory to specify on parameters what is the metrics that will help to calculate all measures. On next description, there are some examples of these metrics:

Condition Coverage on New Code: Condition coverage of new or changed code

Lines of Code Per Language: Non Commenting Lines of Code Distributed By Language

Profiles: Details of quality profiles used during analysis

Security Hotspots Reviewed on New Code: Percentage of Security Hotspots Reviewed on New Code

Quality Gate Status: The project status with regard to its quality gate

Finally, for all of these methods was developed an auxiliar class that allows an easy configuration to create an authentication interceptor. This class extends a super class that provides the possibility to create a HTTP context with a basic authentication with username and password. This class is presented on excerpt 5.10.

```
public class HttpComponentsClientHttpRequestFactoryBasicAuth
        extends HttpComponentsClientHttpRequestFactory {

    HttpHost host;

    public HttpComponentsClientHttpRequestFactoryBasicAuth(HttpHost host)
    {
        super();
        this.host = host;
    }

    protected HttpContext createHttpContext(HttpMethod httpMethod, URI uri
    ) {
        return createHttpContext();
    }

    private HttpContext createHttpContext() {
        AuthCache authCache = new BasicAuthCache();

        BasicScheme basicAuth = new BasicScheme();
        authCache.put(host, basicAuth);

        BasicHttpContext localcontext = new BasicHttpContext();
        localcontext.setAttribute(HttpClientContext.AUTH_CACHE, authCache)
    ;
        return localcontext;
    }
}
```

Listing 5.10: HTTP Authentication

To facilitate user interaction with the solution, it was also created a query that allows users to create a SonarQube access token. This token can replace HTTP parameter in a basic authentication, without the need of a password. This method is presented on excerpt 5.11.

```
    public String createAccessTokenSonar(String nameToken) {
        Token token = null;
        try{
            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);

            MultiValueMap<String, String> map= new LinkedMultiValueMap<>()
    ;
            map.add("name", nameToken);

            RestTemplate restTemplate = new RestTemplate();
            restTemplate.getInterceptors().add(new
    BasicAuthenticationInterceptor(userSonar,passSonar));
            String resourceUrl = sonarQubeLocation + "/api/user_tokens/
    generate";
            HttpEntity<MultiValueMap<String, String>> request = new
    HttpEntity<>(map, headers);
            token = restTemplate.postForObject(resourceUrl, request, Token
    .class);
        }catch (HttpClientErrorException e){
            return e.getResponseBodyAsString();
        }
        return token.getToken();
    }
```

Listing 5.11: Token Creation for SonarQube

5.4 Summary

This chapter presents the implementation for the project. Initially, it is described the general implementation for the project, that includes the description of some methods that are common for both services.

The GitInspector Integration describes all the adjustments to integrate a tool that generates reports in a specific format, that is different of the main purpose of this project. And once that main purpose is to develop a GraphQL API, it also be consumed the REST API of SonarQube to be converted on a GraphQL API.

Finally, this is a solution that be capable of providing data from two different services, GitInspector and SonarQube.

Chapter 6

Evaluation and Experimentation

In this chapter, the developed solution will be evaluated, and for this, the Goal Question Metric (GQM) method will be used to evaluate the solution in terms of quality factors and metrics, thus allowing us to assess whether the solution is in accordance with the expected objectives.

The GQM method is based on a first definition of a goal that have to be reached, and represents conceptual level that includes as objects of measurement the products (specifications, design, programs), processes (designing, testing) and resources (hardware, software).

The second level includes the definition of a question and represents a logical level that explains the way the assessment is going to be performed.

The third level, quantitative level, allows to define a set of data for each question in order to answer it in a quantitative way [41].

In addition to the GQM method, test methods will be used to assess the technical quality of the entire developed solution.

The hypothesis formulated for this project was defined after all solution were developed and after the concepts of microservices and GraphQL were well established. The hypothesis is the following:

- A microservices architecture with GraphQL provides a git analysis solution with good performance, maintainability, and lower time to answer all requests.
- The solution meets all the requirements defined for the technical quality assessment metrics

This evaluation will help the people who want to choose a tool to analyze their repositories. They will have consistent test results about this solution to understand if it is appropriate to their needs and objectives.

6.1 Methodology

The process of evaluation will be divided into two phases: technical evaluation and quality assurance.

6.1.1 Technical Evaluation

The technical evaluation will included five different types of tests: unit and integration tests (JUnit), code coverage (JaCoCo), mutation tests and loading tests (Apache JMeter).

Unit Tests

Unit tests allow you to test the unit, that is, the smallest piece of code that is isolated from the system. The metric is that all tests must pass, and all units in the system must be tested, like was described on table 6.1. The tool that will be used JUnit, an open-source framework that allows the development of automatized tests.

Table 6.1: GQM for Unit Testing

Goal	Question	Metrics
Evaluate the funcionality of each unit by using unit tests	Is each unit of the solution totally functional as expected?	All considered tests have to be with success status

Integration Tests

Integration tests allows to test the integration between existing modules of the solution. The metric is that all tests must pass, and functionalities between modules have to be tested, like was described on table 6.2. The tools that will be used is JUnit, also used for unit testing, and mockito, that allows the development of duplicated objects to facilitate testing.

Table 6.2: GQM for Integration Testing

Goal	Question	Metrics
Evaluate the funcionality of all modules by using integration tests	Is all modules and integration between them totally functional as expected?	All considered tests have to be with success status

Code Coverage

Code coverage is a metric that allows to measure the developed code that is covered by the existing tests. The metric defined for this evaluation is that the coverage have to be greater than 90%, like was described on table 6.3. The tool that will be used is JaCoCo, that is totally configurable to analyze developed code and produce coverage reports.

Table 6.3: GQM for Code Coverage

Goal	Question	Metrics
Evaluate the coverage for all developed code	Is all developed code covered by existing unit and integration tests?	Code coverage have to be greater than 90%

Mutation Tests

Mutation Tests is a system to create new software tests and evaluate the quality of the developed unit and integration tests. For this, it is created some changes on the existing solution to verify tests quality. The metric for this evaluation is that the mutation percentage have to be greater than 90%, like was described on table 6.4.

Loading Tests

Loading Tests are intended to test the load capacity of computer systems. For this, are made many requests to the system to evaluate the capacity to deal with a consequent

Table 6.4: GQM for Mutation Tests

Goal	Question	Metrics
Evaluate mutation for the existing tests	Is all developed tests with expected quality?	Mutation percentage have to be greater than 90%

requests. The tool that will be used is Apache JMeter, that was developed by Apache to develop loading tests. The GQM for this tests is presented on table 6.5.

Table 6.5: GQM for Loading Tests

Goal	Question	Metrics
Evaluate the load capacity of the solution	Is the solution capable to deal with many consequent requests?	The solution have to answer correctly to all requests

6.1.2 Quality Assurance

The quality assurance includes the evaluation of some metrics that contributes to the system quality, like maintainability and performance.

Performance

Performance is one of metrics that have to be evaluated to check the solution quality. For this, the tool that will be used is the same of loading tests tool, Apache JMeter. This tool that allows the creation of a lot of requests to test the capacity load of the solution. Furthermore, is is possible to analyze the response time for each request, which will allow the performance evaluation. The GQM for this metric is presented on table 6.6.

Table 6.6: GQM for Performance

Goal	Question	Metrics
Evaluate the performance of the solution	Is the solution capable to do all functionalities with the best performance as possible?	The solution must have a very low lead time

Maintainability

Maintainability is the capacity of a system to provides an easy way to maintenance its functions. For this, there are many metrics that can be evaluated, like technical debt ratio, that is the time that have to spent to fix all issues on a system. The tool that will be used is SonarQube, that provides a collection of tools to analyze software systems, and have a maintainability rating that reflects the technical debt ratio. The GPM for this metric is presented on table 6.7.

6.2 Assessment

On this section, it will be described all of the steps created to evaluate the solution. Each metric has a different subsection.

Table 6.7: GQM for Maintainability

Goal	Question	Metrics
Evaluate the maintainability of the solution	Is the solution easy to mantain and change during the time?	The solution must have at least grade B for maintainability rating

6.2.1 Methodology Changes

The main goal of the project can not be properly tested with resource of common tests. Initially, it was planned to use unit tests, integration tests, code coverage and also mutation tests. As the project progressed, it came to the conclusion that it would not be the best way to evaluate a solution that aims to integrate other existing services. These services are solutions that already exist on the market and are very usable, and therefore it is considered that the services will always produce the same response to same request. On the section 6.1, all tests initially thought will continue to be described, since they are part of the planning process and ended up not justifying their application during the project. From section 6.1.2, only Loading tests will be appplied.

6.2.2 Loading Tests

The main purpose of loading tests is to verify the responsiveness of the solution to a high number of requests. To evaluate this, it was used Apache JMeter. For each test was used 50 different requests, that simulates the interaction of 50 different users. This configuration is possible to see on image 6.1.

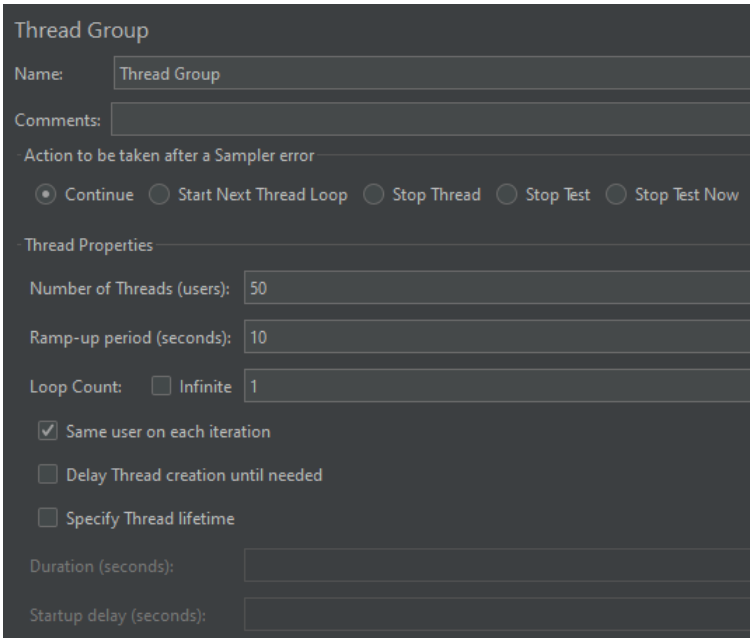


Figure 6.1: JMeter Configuration

To evaluate the metric created for these tests, load tests were performed on one method of each service. The image 6.2 shows request configuration for running loading tests against a GitInspector method.

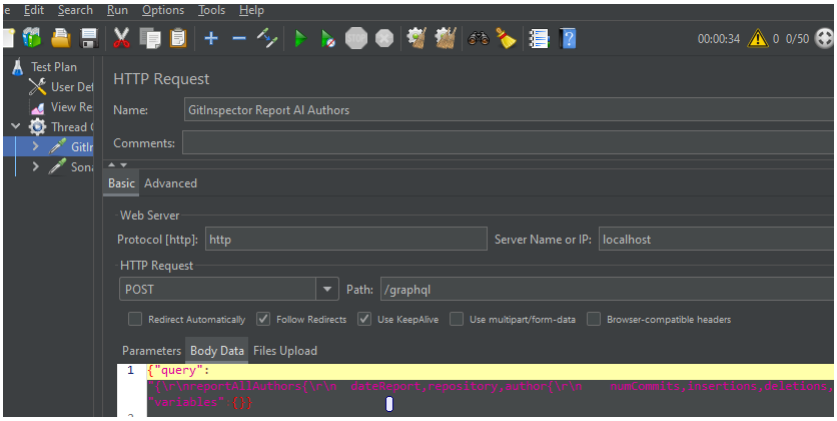


FIGURE 6.2: JMeter Configuration for method ReportAllAuthors

After executing 50 requests, it is possible to see in the image 6.3 that all requests were executed successfully. It is also possible to check another details about the execution, like the time to answer to request. This metric will be evaluated within the scope of performance.

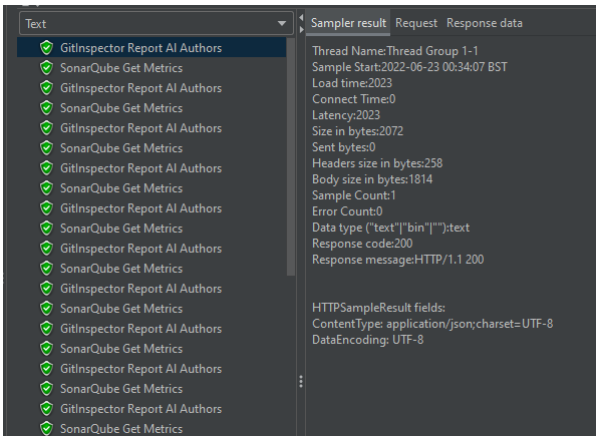


FIGURE 6.3: Results for JMeter Execution

The time to answer to the first request was 2023 milliseconds, while the response time for the last request was 29641 milliseconds. This can be explained with the launch of a new GitInspector instance for each request. To launch a new instance, the computer used to develop the solution has to use a new processor thread to compute in parallel all the requests. The gradual increase of response time can be explained with the limited number of threads of the processor.

For the metric of loading tests, it is possible to say that was fulfilled with sucess, although the response time increases considerably. This reponse time will be evaluated on section 6.2.4.

6.2.3 Performance

The main purpose of performance evaluation is the ability of the solution to respond quickly to requests. To evaluate this, it was used Apache JMeter. To evaluate this metric, have to be considered the time to answer to a request. Initially, will be evaluated the time to answer of a normal request, which only uses one thread to execute code.

Then it will be done a performance comparison between a single thread method and a multi thread method (uses MapReduce to share the processing).

The test result for the method reportAllAuthors is presented on image 6.4.

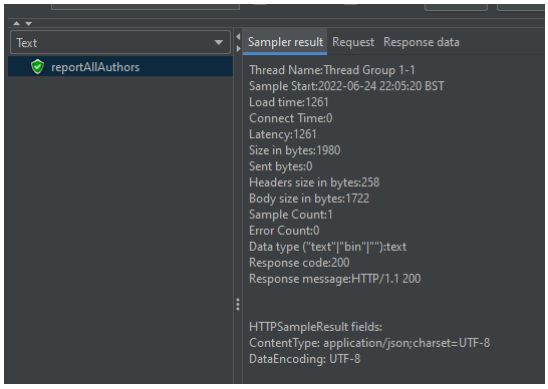


FIGURE 6.4: Execution reportAllAuthors

How it is possible to see on image 6.4, the response time for the method is 1261 ms. This time includes the time that GitInspector service takes to generate the report and the time that solution take to read and mapping the generated report. It can be said that the response time is low, since it is not possible to control the response time of GitInspector service.

The next test is to compare response time of last method to the response time of a similar method, but with the usage of multiple threads for a single execution. The method reportAllAuthors generate reports for all existing authors on the repository. The second method to be compared is reportAllAuthorsMultipleInstances, which was called for exactly the same authors. The result of the execution is possible to see on image 6.5.

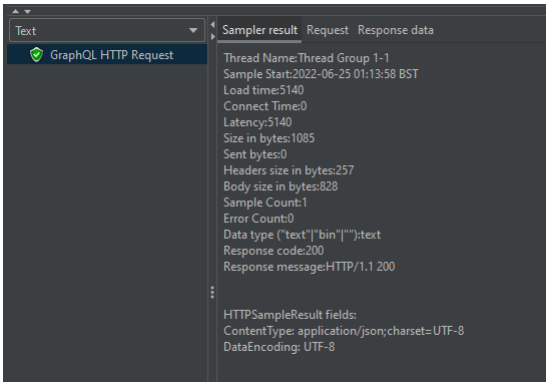


FIGURE 6.5: Execution reportAllAuthorsMultipleInstances

As it is possible to observe, the response time was 5140 milliseconds. Comparing the values obtained for the execution of two different methods, it is possible to conclude that the usage of MapReduce and multi-threading programming on GitInspector service, it is not efficient. This can be explained by the need for the main thread to have to wait for all other threads to execute, since the report must be presented to the user with all the data. For this reason, the method must necessarily be synchronous.

About the created metric, it is possible to conclude that the solution developed has a low response time, since most of the response time is caused by the execution of external services, such as the GitInspector and SonarQube services.

6.2.4 Maintainability

The main purpose of maintainability evaluation is to evaluate the capacity of the solution to provides an easy way to maintenance its functions. To evaluate this, it was used SonarQube. With metric maintainability ratio, available on SonarQube, it is possible to assign a value to maintainability, taking into account the technical debt ratio, that is the time that must be used to fix all the code smells existing on the code. The best value for maintainability is grade A and the image 6.6 shows the results obtained on SonarQube for this project.

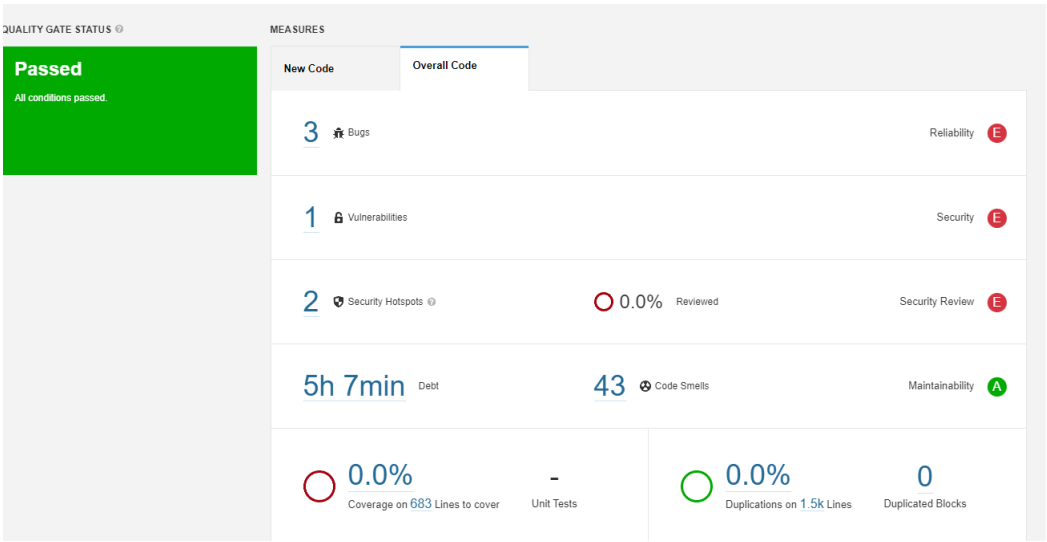


FIGURE 6.6: SonarQube Maintainability

It is possible to conclude that the solution meets the requirements created for this metric, also presenting a low value of technical debt.

6.3 Summary

On this chapter, were presented the results for all metrics and it is possible to conclude that all the objectives defined for the metrics have been met. Regarding loading tests, all tests were performed successfully, regarding performance evaluation, came to the conclusion that the solution has a low response time and the usage of MapReduce techniques of this solution does not improve response time. Finally, a good maintainability value was also obtained.

Chapter 7

Conclusion

This section summarizes what it was done, difficulties found, but also future work and contributions. All of the code developed for this project is available on a public repository [1], as already mentioned in chapter 5. All conclusions are based on the code available in the repository and also on the tests and evaluation carried out in the chapter 6.

7.1 Achievements

This document presents all the steps made in order to find a solution that was able to integrate some git repositories analysis tools. Firstly, it was analyzed what is the state of art for the git analysis tools. Some git providers were also analyzed, and it came to the conclusion that none of the existing technologies used GraphQL.

Then, a process to find a solution was followed, which is described on the subsection 3.1. There were three possible solutions on the table, and the decision was to take a step into GraphQL with a microservices architecture. This technology was unknown until then, what caused some time spent on the learning. Due to the limit time to present the thesis, the knowledge of GraphQL was not very deep but it was not an impediment to the creation of the presented solution.

Graphl was also one of the topics covered in the state of the art. The comparison between REST and GraphQL allowed us to compare some of the advantages and disadvantages of the two technologies. Some patterns that can be applied to a microservices-oriented architecture were also discussed, since one of the objectives was to create a GraphQL-based and microservices-oriented solution.

One of the main difficulties experienced during the development of this thesis was the lack of scientific documents on repositories analysis. It is not a very explored topic, and for this reason, gray documentation had to be used as a complement to the more reliable documentation that already exists.

Despite the difficulties presented above, it was possible to create a state of the art with enough information to build some design alternatives and also some hypotheses to be validated during the realization of the project.

The research question presented on chapter 1.3 was:

- Can the solution integrate different git repositories analysis tools?

Although performance was not mentioned in the research question, it was one of the aspects to be taken into account during the development of the thesis. The application of MapReduce techniques was an alternative solution created just to evaluate if it improved the performance of the solution or not. This evaluation was carried out in the chapter6.

[1] https://github.com/1160836/tools-git-integration

Regarding the question itself, it is possible to answer in the affirmative, since it was achieved the integration of two different tools, Gitinspector and Sonarqube, as described in the chapter 5. The chapter 6 also allowed to conclude that the MapReduce techniques do not improve the performance of the two tools integrated in the solution, since the analysis of the repositories is always done in full. For this reason, the attempt to distribute processing causes repeated execution of code that has already been executed, which makes the response time higher. On the other hand, GraphQL offers many advantages, such as the use of a single endpoint and the fact that users can filter the attributes they want the response to contain, unlike what happens with a REST API.

In conclusion, it is possible to integrate different analysis tools of git repositories. The data generated by these services are processed and presented based on a GraphQL API, thus facilitating the interaction of the solution with possible consumers of the developed API.

7.2 Future Work

There are several tools for analyzing git repositories. Related to the thesis, the created solution should be able to integrate more analysis tools. When integrating a new tool, an evaluation must be carried out to conclude which API is best to use. The existence of a solution capable of providing different APIs is a possibility and should be a hypothesis if justified.

As described in the topic 7.1, GraphQL was a new technology to learn, which may have provoked a superficial analysis of the potential of this API. Therefore, future work should include a more detailed process of learning GraphQL applied on a microservices architecture and also an analysis of existing large-scale applications that make use of the tool. This will allow observing how the tool is used in a business context and how the solution can be updated according to good practices and observed standards.

7.3 Contributions

The context of this thesis was applied to ISEP. It is an academic context project that can provide some knowledge about the implementation of a GraphQL API along with a microservices oriented architecture. All of the code developed for this project is available on a public repository [2], as already mentioned in chapter 5.

The created solution has not been put into production. As it is a prototype, it should only move to the next phase when it is more consolidated, more tested and also with a greater variety of integrated tools. This solution is a good option for students and teachers who want to consult information about their repositories in a more centralized way.

[2] https://github.com/1160836/tools-git-integration

Bibliography

[1] *GitHub Archive*. 2022. URL: https://www.gharchive.org/.

[2] Gleison Brito and Marco Tulio Valente. "REST vs GraphQL: A controlled experiment". In: *Proceedings - IEEE 17th International Conference on Software Architecture, ICSA 2020* (Mar. 2020), pp. 81–91. DOI: 10.1109/ICSA47634.2020.00016.

[3] Yuanyuan Yin, Shengfeng Qin, and Ray Holland. "Development of a project level performance measurement model for improving collaborative design team work". In: *Proceedings of the 2008 12th International Conference on Computer Supported Cooperative Work in Design, CSCWD* 1 (2008), pp. 135–140. DOI: 10.1109/CSCWD.2008.4536970.

[4] Sri Lakshmi Vadlamani et al. "Can GraphQL Replace REST? A Study of Their Efficiency and Viability". In: *Proceedings - 2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice, SER and IP 2021* (June 2021), pp. 10–17. DOI: 10.1109/SER-IP52554.2021.00009.

[5] Omar Al-Debagy and Peter Martinek. "A Comparative Review of Microservices and Monolithic Architectures". In: *18th IEEE International Symposium on Computational Intelligence and Informatics, CINTI 2018 - Proceedings* (Nov. 2018), pp. 149–154. DOI: 10.1109/CINTI.2018.8928192.

[6] Sivana Hamer et al. "Measuring students' contributions in software development projects using Git metrics". In: *Proceedings - 2020 46th Latin American Computing Conference, CLEI 2020* (Oct. 2020), pp. 531–540. DOI: 10.1109/CLEI52000.2020.00068.

[7] *Design science research — a short summary | by Rauno Pello | Medium*. URL: https://medium.com/@pello/design-science-research-a-summary-bb538a40f669.

[8] Sascha Just et al. "Switching to Git: The Good, the Bad, and the Ugly". In: *Proceedings - International Symposium on Software Reliability Engineering, ISSRE* (Dec. 2016), pp. 400–411. ISSN: 10719458. DOI: 10.1109/ISSRE.2016.38.

[9] Caius Brindescu et al. "How do centralized and distributed version control systems impact software changes?" In: *Proceedings - International Conference on Software Engineering* CONFCODENUMBER (May 2014), pp. 322–333. ISSN: 02705257. DOI: 10.1145/2568225.2568322.

[10] Reza M. Parizi, Paola Spoletini, and Amritraj Singh. "Measuring Team Members' Contributions in Software Engineering Projects using Git-driven Technology". In: *Proceedings - Frontiers in Education Conference, FIE* 2018-October (Mar. 2019). ISSN: 15394565. DOI: 10.1109/FIE.2018.8658983.

[11] Jordan Brekke et al. "INSIGHT and ANALYTICS from GIT REPOSITORIES". In: *IEEE International Conference on Electro Information Technology* 2020-July (July 2020), pp. 383–388. ISSN: 21540373. DOI: 10.1109/EIT48999.2020.9208267.

[12] Michael Cochez et al. "How Do Computer Science Students Use Distributed Version Control Systems?" In: *Communications in Computer and Information Science* 412 CCIS (2013), pp. 210–228. ISSN: 18650929. DOI: 10.1007/978-3-319-03998-5{_}11.

[13] *Git Book - The Git Object Model*. URL: http://shafiul.github.io/gitbook/1_the_git_object_model.html.

[14] Susi Eva Maria Purba et al. "Measuring the Individual Performance of A Software Development Team". In: *Proceedings of the 8th International Conference on*

Computer and Communication Engineering, ICCCE 2021 (June 2021), pp. 78–81. DOI: 10.1109/ICCCE50029.2021.9467198.

[15] Atlassian. *Bitbucket Overview | Bitbucket*. 2022. URL: https://bitbucket.org/product/guides/getting-started/overview#a-brief-overview-of-bitbucket.

[16] GitLab. *GitLab.org / GitLab · GitLab*. 2022. URL: https://gitlab.com/gitlab-org/gitlab.

[17] Rohan Vats. *GitHub vs GitLab: Difference Between GitHub and GitLab | upGrad blog*. 2020. URL: https://www.upgrad.com/blog/github-vs-gitlab-difference-between-github-and-gitlab/.

[18] Taurie Davis. *Redesigning GitLab's navigation | GitLab*. 2017. URL: https://about.gitlab.com/blog/2017/07/17/redesigning-gitlabs-navigation/.

[19] GitHub. *GitHub: Where the world builds software · GitHub*. 2022. URL: https://github.com/.

[20] GitHub. *API de REST do GitHub - GitHub Docs*. 2022. URL: https://docs.github.com/pt/rest.

[21] Jordi Cabot. *20+ tools to help you mine and analyze GitHub and Git data*. 2017. URL: https://livablesoftware.com/tools-mine-analyze-github-git-software-data/.

[22] Valerio Cosentino et al. "Gitana: A software project inspector". In: *Science of Computer Programming* 153 (2018), pp. 30–33. DOI: 10.1016/j.scico.2017.12.002. URL: www.elsevier.com/locate/scico.

[23] Georgios Gousios. "The GHTorent dataset and tool suite". In: *IEEE International Working Conference on Mining Software Repositories* (2013), pp. 233–236. ISSN: 21601852. DOI: 10.1109/MSR.2013.6624034.

[24] Mario Villamizar et al. "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures". In: *Service Oriented Computing and Applications* 11.2 (June 2017), pp. 233–247. ISSN: 18632394. DOI: 10.1007/S11761-017-0208-Y.

[25] Francisco Ponce, Gaston Marquez, and Hernan Astudillo. "Migrating from monolithic architecture to microservices: A Rapid Review". In: *Proceedings - International Conference of the Chilean Computer Science Society, SCCC* 2019-November (Nov. 2019). ISSN: 15224902. DOI: 10.1109/SCCC49216.2019.8966423.

[26] Google. *Microservices Popularity*. 2022. URL: https://trends.google.pl/trends/explore?date=today%205-y&q=microservices.

[27] Konrad Gos and Wojciech Zabierowski. "The Comparison of Microservice and Monolithic Architecture". In: *International Conference on Perspective Technologies and Methods in MEMS Design* (2020), pp. 150–153. ISSN: 25735373. DOI: 10.1109/MEMSTECH49584.2020.9109514.

[28] Microsoft. *Microservice architecture style - Azure Application Architecture Guide | Microsoft Docs*. June 2022. URL: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.

[29] S. Suresh Kumar and P. M. Mallikarjuna Shastry. "Database-per-service for e-learning system with micro-service architecture". In: *Proceedings of the 2017 International Conference On Smart Technology for Smart Nation, SmartTechCon 2017* (May 2018), pp. 705–708. DOI: 10.1109/SMARTTECHCON.2017.8358462.

[30] Md Kamaruzzaman. *Microservice Architecture and its 10 Most Important Design Patterns | by Md Kamaruzzaman | Towards Data Science*. 2020. URL: https://towardsdatascience.com/microservice-architecture-and-its-10-most-important-design-patterns-824952d7fa41.

[31] Peter Koen. *Front End Innovation - What is the New Concept Development (NCD) model?* 2004. URL: http://frontendinnovation.com/fei/what-is-the-new-concept-development-ncd-model.

[32] Thomas L Saaty. "Decision making with the analytic hierarchy process". In: *Int. J. Services Sciences* 1.1 (2008), pp. 83–98.

[33] *Function Analysis system Technique (FAST) - Canadian Society of Value Analysis*. Feb. 2022. URL: https://www.valueanalysis.ca/fast.php.

[34] Alexander Osterwalder and Yves Pigneur. "Aligning Profit and Purpose Through Business Model Innovation". In: Feb. 2011, pp. 61–76.

[35] *Java Software | Oracle*. URL: https://www.oracle.com/java/.

[36] *Spring Boot*. URL: https://spring.io/projects/spring-boot#overview.

[37] *Spring Boot Interview Questions - HowToDoInJava*. URL: https://howtodoinjava.com/interview-questions/spring-boot-interview-questions/.

[38] *GitHub - ejwa/gitinspector: The statistical analysis tool for git repositories*. URL: https://github.com/ejwa/gitinspector.

[39] Zhi Jin. "Requirements Engineering Methodologies". In: *Environment Modeling-Based Requirements Engineering for Software Intensive Systems* (2018), pp. 13–27. DOI: 10.1016/B978-0-12-801954-2.00002-9.

[40] Philippe B. Kruchten. "The 4+1 View Model of Architecture". In: *IEEE Software* 12.6 (1995), pp. 42–50. ISSN: 07407459. DOI: 10.1109/52.469759.

[41] Victor R Basili, Gianluigi Caldiera, and H Dieter Rombach. "THE GOAL QUESTION METRIC APPROACH". In: ().

Appendix A

Class Diagram

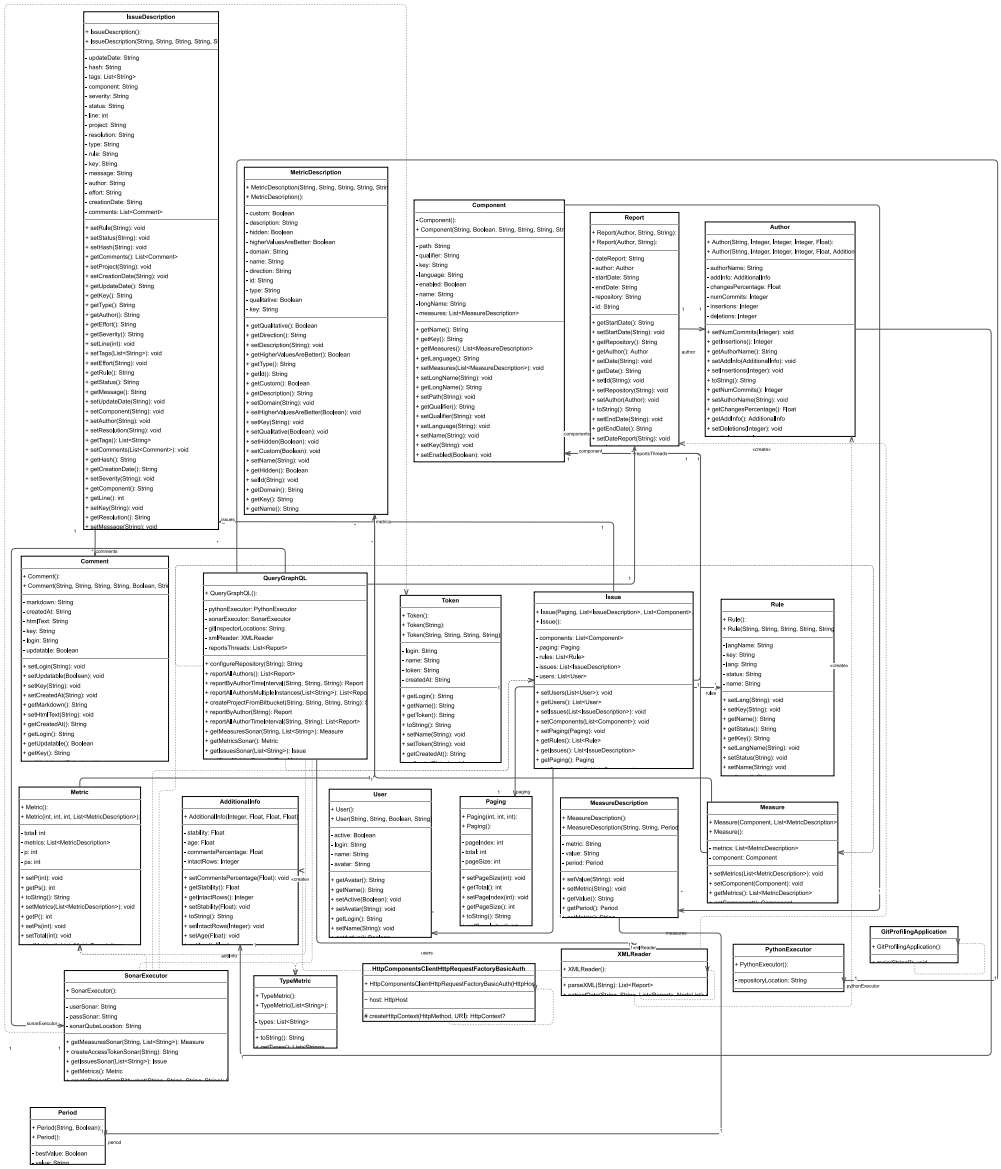


FIGURE A.1: Class Diagram

Appendix B

XML Reader

```
public class XMLReader {

        public List<Report> parseXML(String xml) {

            System.out.println(xml);
            DocumentBuilderFactory dbf = DocumentBuilderFactory.
    newInstance();

            String repository = "";
            String report_date= "";

            List<Report> listReports = new ArrayList<>();

            try {

                dbf.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING,
    true);

                DocumentBuilder db = dbf.newDocumentBuilder();

                File file = new File("report.xml");
                FileWriter writer = new FileWriter("report.xml");
                writer.write(xml);
                writer.close();
                Document doc = db.parse(file);

                doc.getDocumentElement().normalize();

                System.out.println("Root Element :" + doc.
    getDocumentElement().getNodeName());
                System.out.println("------");

                NodeList list = doc.getDocumentElement().getChildNodes();


                for (int temp = 0; temp < list.getLength(); temp++) {

                    Node node = list.item(temp);

                    if (node.getNodeType() == Node.ELEMENT_NODE) {

                        Element element = (Element) node;

                        switch(element.getNodeName()){
                            case "repository":
                                repository=element.getTextContent();
                                break;
                            case "report-date":
                                report_date=element.getTextContent();
                                break;
```

LISTING B.1: XML Reader

```
case "blame":
                                NodeList listNodes1 = element.
    getChildNodes();

                                extractData(repository, report_date,
    listReports, listNodes1);
                                break;

                            case "changes":
                                NodeList listNodes = element.getChildNodes
    ();

                                extractData(repository, report_date,
    listReports, listNodes);
                                break;

                        }
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
            return listReports;
        }

    public void extractData(String repository, String report_date, List<
    Report> listReports, NodeList listNodes1) {
        for(int temp1 = 0; temp1 < listNodes1.getLength(); temp1++) {
            if (listNodes1.item(temp1).getNodeName().equals("authors")) {

                NodeList authors = listNodes1.item(temp1).getChildNodes();

                for (int temp2 = 0; temp2 < authors.getLength(); temp2++)
    {
                    String author_name ="";
                    Integer numCommits = null;
                    Integer insertions = null;
                    Integer deletions = null;
                    Float changesPercentage = null;
                    Integer rows = null;
                    Float stability = null;
                    Float age = null;
                    Float commentsPercentage = null;

                    if(authors.item(temp2).getNodeName().trim().equals("
    author")){
                        NodeList authorAtributes= authors.item(temp2).
    getChildNodes();

                        for (int temp3=0 ; temp3 < authorAtributes.
    getLength(); temp3++){

                            switch (authorAtributes.item(temp3).
    getNodeName()){
                                case "name":
                                    author_name = authorAtributes.item(
    temp3).getTextContent();
                                    break;

                                case "commits":
                                    numCommits = Integer.parseInt(
    authorAtributes.item(temp3).getTextContent());
                                    break;

                                case "insertions":
                                    insertions = Integer.parseInt(
    authorAtributes.item(temp3).getTextContent());
                                    break;
```

Listing B.2: XML Reader

```
case "deletions":
                                    deletions = Integer.parseInt(
    authorAtributes.item(temp3).getTextContent());
                                    break;

                                case "percentage-of-changes":
                                    changesPercentage = Float.parseFloat(
    authorAtributes.item(temp3).getTextContent());
                                    break;

                                case "rows":
                                    rows = Integer.parseInt(
    authorAtributes.item(temp3).getTextContent());
                                    break;

                                case "stability":
                                    stability = Float.parseFloat(
    authorAtributes.item(temp3).getTextContent());
                                    break;

                                case "age":
                                    age = Float.parseFloat(authorAtributes
    .item(temp3).getTextContent());
                                    break;

                                case "percentage-in-comments":
                                    commentsPercentage = Float.parseFloat(
    authorAtributes.item(temp3).getTextContent());
                                    break;

                            }
                        }
                        if(listReports.size()==0)listReports.add(new
    Report(new Author(author_name,numCommits,insertions,deletions,
    changesPercentage),report_date,repository));
                        else {
                            boolean flag = false;
                            for (int i = 0; i < listReports.size(); i++) {
                                if (author_name.equals(listReports.get(i).
    getAuthor().getAuthorName()) && rows != null) {
                                    listReports.get(i).getAuthor().
    setAddInfo(new AdditionalInfo(rows, stability, age, commentsPercentage
    ));
                                    flag = true;
                                }
                            }
                             if (!flag){
                                listReports.add(new Report(new Author(
    author_name, numCommits, insertions, deletions, changesPercentage),
    report_date, repository));
                            }
                        }
```

Listing B.3: XML Reader