An exploration of modern domain specific software architectures

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ulvi Shakikhanli

Supervisor:
Vilmos Bilicki, PhD
assistant professor

Doctoral School of Computer Science
Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged

2024

Preface

Everyone faces several important crossroads during their lifetime. One of the most significant for me was deciding to continue my studies in Hungary. Over the years, studying in Hungary has provided me with valuable experience in both the academic and professional fields. And, attending one of Hungary's best universities has enhanced my vision of becoming a researcher in the field of information technology. It has been a long and challenging journey, but with the help of my teachers, friends, and family, I was able to overcome all the difficulties that I encountered. From now on, new challenges and crossroads await me, and I am sure that the experience and knowledge I gained over the years will be my trusted guides.

Ulvi Shakikhanli, 2024

Contents

Introduction ... 4
1.1 Contributions ... 5
Background ... 7
Thesis Group I: Mono and Multi repository structures and Identification process ... 9
3.1 Introduction ... 10
3.2 Related Works ... 10
3.3 The Roles of Software Architecture ... 11
3.4 Software Architecture Styles ... 13
3.4.1 Service-oriented Architecture. ... 13
3.4.2 Object-oriented Architecture. ... 13
3.5 Thesis I/1: Repository Types ... 14
3.5.1 Thesis I/2: Machine Learning Method for Repository Identification ... 15
3.5.2 Training the machine learning model. ... 17
3.6 Thesis I/3: Identification algorithm for Mono and Multi repository projects ... 18
3.6.1 The identification of Mono repository projects ... 18
3.6.2 The mathematical specification of an algorithm for the Identification of Mono repository projects ... 19
3.6.3 The identification of Multi repository projects ... 19
3.6.4 The mathematical specification of an algorithm for the Identification of Multi repository projects ... 20
3.7 The Database for Mono and Multi repository projects ... 21
3.8 Analyses of Mono and Multi repository Structures. ... 23
3.8.1 The development period in Mono and Multi repository projects ... 23
3.8.2 The developer team size in Mono and Multi repository projects ... 24
3.8.3 The popularity trend of Mono and Multi repository projects over the years. ... 25
3.9 Thesis I/4: Multi Repository Management Tools ... 27
3.10 Concluding Remarks ... 29
Thesis Group II: Branching strategies in Mono and Multi repository projects. ... 31
4.1 Introduction ... 31
4.2 Related Work ... 32
4.3 Version Control Systems ... 33
4.3.1 Centralised Version Control Systems ... 33

4.3.2 Distributed Version Control Systems ... 33
4.4 Branching Strategies ... 34
4.4.1 Thesis II/1: Identification of Branching strategy in Open Source Projects ... 36
4.4.2 Mathematical Representation of Branch Identification ... 37
4.5 The popularity ratio of Branching strategies over the years ... 38
4.6 Thesis II/2: Branching strategies in Mono and Multi repository projects ... 40
4.6.1 Mono repository projects ... 40
4.6.2 Multi repository projects ... 41
4.7 Concluding Remarks ... 42
Thesis Group III: The productivity of software development ... 43
5.1 Introduction ... 43
5.2 Related Works ... 44
5.3 The calculation of Productivity ... 45
5.3.1 An algorithm for the calculation ... 45
5.3.2 Mathematical specification of an algorithm ... 48
5.4 Productivity in repository Structure ... 49
5.4.1 Productivity in the Mono repository ... 49
5.4.2 Productivity in the Multi repository ... 50
5.5 Productivity and the branching Strategy ... 51
5.5.1 Three main branching strategies ... 51
5.6 Properties of branching strategies ... 53
5.6.1 Commit Count ... 53
5.6.2 Branch Count ... 55
5.6.3 The developer team size and development period ... 58
5.7 Thesis III/1: The calculation of Productivity using the new ML method ... 59
5.7.1 Feature Extraction and Training ... 60
5.8 Thesis III/2: Prediction of Software Development Period ... 61
5.8.1 Related work ... 62
5.8.2 Feature Extraction and Training ... 62
5.9 Results and Discussion ... 63
5.9.1 repository Structure ... 63
5.9.2 Branching Strategy ... 64
5.9.3 Project properties ... 65
5.10 Concluding Remarks ... 65
Thesis Group IV: Collaboration of Software Developer Team ... 67

6.1 Introduction ... 67
6.2 Related Work ... 68
6.3 The calculation of Collaboration ... 70
6.3.1 Thesis IV/1: Methodology ... 70
6.3.2 Mathematical Specification ... 72
6.4 Repository Structure and Collaboration ... 73
6.5 Productivity and Collaboration ... 74
6.6 Branching Strategies and Collaboration ... 76
6.6.1 A comparison of the branch count and collaboration ratio of projects ... 76
6.6.2 A comparison of commit count and collaboration ratio of projects. ... 79
6.6.3 Developer Team Size versus Collaboration ... 80
6.6.4 Development Period versus Collaboration ... 82
6.7 Thesis IV/2: Predictive Modelling for Developer Team Sizing ... 83
6.7.1 Results of the model ... 84
6.8 Results and Discussion ... 85
6.9 Concluding Remarks ... 86
Conclusions ... 87
7.1 Results ... 87
7.2 Future Work ... 89
Bibliography ... 90
Summary ... 100
Contributions of the thesis ... 102
Összefoglaló ... 104
Publications ... 108
Acknowledgments ... 109

List of Figures

Figure 3. 1 The role of Software architecture in a project. 12
Figure 3. 2 Visual presentation of Object-oriented architecture. 14
Figure 3. 3 Structure of JSON file in database which represents a Mono repository project...... 22
Figure 3. 4 A comparison of the development period of two repository structures in percentage terms. 24
Figure 3. 5 A comparison of the developer team sizes of two repository structures in percentage terms. 25
Figure 3. 6 A comparison of the developer team sizes of two repository structures in percentage terms. 26
Figure 3. 7 Popularity of each feature among the developers. 28

Figure 4. 1 Schematic explanation of GitFlow branching strategy 34
Figure 4. 2 Schematic explanation of GitHub Flow branching strategy..................................... 35
Figure 4. 3 Schematic explanation of Trunk-based branching strategy...................................... 35
Figure 4. 4 Popularity of three main branching strategies over the years.................................. 39
Figure 4. 5 Usage percentage of three major branching strategies in Mono repository projects. 40
Figure 4. 6 Usage percentage of three major branching strategies in Multi repository projects. 41

Figure 5. 1 Burst sequence of random Mono repository project from our database. 47
Figure 5. 2 The percentage share of Mono repository projects according to their productivity level. 50
Figure 5. 3 The percentage share of Multi repository projects according to their productivity level. 50
Figure 5. 4 Usage of branching strategies according to the productivity level (Mono repository). 51
Figure 5. 5 Usage of branching strategies according to the productivity level (Multi repository). 52
Figure 5. 6 The percentage share of commit counts for two main branching strategies in Mono repository projects. 54
Figure 5. 7 The percentage share of commit counts for two main branching strategies in Multi repository projects. 55
Figure 5. 8 GitFlow - High vs Low Productivity Branch count in Mono repository projects.... 56
Figure 5. 9 Github Flow - High vs Low Productivity Branch count in Mono repository projects. 56
Figure 5. 10 A percentage share of branch counts in the Multi repository projects. 57
Figure 5. 11 A scatter Plot of Development period and Team size with different productivity values in Mono repository projects. 58
Figure 5. 12 A scatter Plot of Development period and Team size with different productivity values of Multi repository projects. 59

Figure 6. 1 A percentage comparison of different collaboration levels in Mono and Multi repository structures. 73
Figure 6. 2 The percentage share of High and Low productivity of Mono repository projects based on the collaboration rates. 74
Figure 6. 3 A comparison of the collaboration rate in different productivity levels among Multi repository projects. 75
Figure 6. 4 A comparison of the branch count and collaboration rate of Mono repository projects including productivity levels. 77
Figure 6. 5 A comparison of the branch count and collaboration rate of Multi repository projects including productivity levels. 78
Figure 6. 6 A comparison of the commit count and collaboration rate of Mono and Multi repository projects. 79
Figure 6. 7 A comparison of the team size and collaboration rate of Mono and Multi repository projects. 81
Figure 6. 8 A comparison of the development period and collaboration rate of Mono and Multi repository projects. 82
Figure 6. 9 A heatmap of the average team size for TypeScript projects 84

List of Tables

Table 3. 1 The accuracy score for different algorithms during the training process. 17
Table 3. 2 A list of MRMTs with their key features and signature files 27

Table 5. 1 A accuracy results of the model training process. 60

Chapter 1

Introduction

Over the past three decades, the field of information technology has experienced significant growth. Several aspects of software development have changed significantly as a result of this extensive development, which is why software architectures have become one of the most debated topics among researchers. The term software architecture is defined and explained in a variety of ways. For instance, the term "architecture" has been primarily used as a contrast to "design," encompassing connotations of codification, abstraction, adherence to standards, formalised training for software architects, and the embodiment of a distinctive stylistic approach [1]. The concept of software architecture has been defined as a collection of plans that operate as a guiding framework for the optimal utilisation, construction, modification, and selection of the enterprise information infrastructure in a distinct scholarly work. The objective of this strategic framework is to enable business enterprises to achieve their desired future goals [2].

One of these definitions can be derived from a paper [3]. The software architecture is a system's overarching structure or structures. The constituent software components, their externally discernible attributes, and the intricate interconnections that define their relationships are all included in this comprehensive structure. Numerous additional studies support the popularity of this topic in the field of information technology. However, there is a sparsity of research on the subject of repository structures, their parameters, and the impact they can have on software development processes. Consequently, our research was primarily centred on the examination of repository structures. In this dissertation, I present my findings in the areas of software development productivity, repository structures, and software team collaboration. I investigated a variety of aspects of the topics, and the numerous methods and algorithms presented here may be beneficial to other academics in overcoming the challenges they encounter in their software development work.

In Chapter 2, I introduce three novel algorithms and approaches for the processing of Mono and Multi repository structures. The first of these is the machine learning approach for the identification of frontend and backend repositories. The Github platform contains over 450 million repositories, with millions more available on other online platforms. Our method simplifies the process of distinguishing repositories from one another, which also facilitated our subsequent research. Secondly, I developed an algorithm for the identification and collection of mono- and multi-repository projects, with a primary focus on the Github platform. The absence of data for analysis was one of the primary challenges that we encountered during our research. After the implementation of our two new algorithms, which facilitated the collection of mono- and multi-repository projects, this problem was mostly resolved. A heuristic approach for the identification

of multirepository management tools is presented in the final section. This is also one of the most neglected areas of research, and our methodology permits the effective identification of MRMTs utilised by various projects.

The subject of branching strategies is the primary focus of Chapter 3. Branching strategies are a critical component of project management and one of the primary factors that influences the software development process. Despite the existence of numerous studies in this field, none of them have conducted an individual analysis of branches or proposed any solutions. In this chapter, I introduce a heuristic methodology for the identification of the three most prevalent branching strategies. This makes it possible to identify branching strategies and the analysis of their relationship with various project parameters. The next chapter provides a few fundamental analyses that can throw light on significant questions, such as the correlation between repository structure and branching strategies.

The productivity of the software development process and the impact of various factors, such as repository structure and branching strategy, are the primary focus of Chapter 4. I employed a method that had been previously tested and proved successful for calculating development productivity. We implemented our own machine learning approach to assess the productivity level of the development process. This new model was in accord with the dissertation's prior findings, and it will be validated for one of the numerous applications of the dissertation's findings. Furthermore, a machine learning approach was introduced to estimate the development period. Naturally, this method is extremely beneficial for project managers, scrum masters, and developers, as it provides an estimate of the development period in months.

My findings in the area of software team collaboration and the correlation between the project development and collaboration aspects are outlined in the final chapter. I developed a straightforward mathematical method for calculating software team collaboration, despite the fact that there are numerous methods available for measuring collaboration. This method is much simpler to implement than the majority of the other existing methods, and it evaluates the collaboration in an objective manner, by considering the contributions made by each developer in the team.

1.1 Contributions

Chapter 3. I investigate a variety of critical subjects, such as full stack development, backend development, and frontend development. I subsequently provide detailed descriptions of the two most common software development architectures; namely the Mono and Multi repository architectures. Then, I develop two new algorithms that employ machine learning to identify and collect projects associated with these repository structures, as there are currently no methodologies available for identifying and collecting them. Furthermore, I develop an algorithm for the identification of frontend and backend repositories. This algorithm employs novel concepts, such as file structure, to facilitate identification.

Chapter 4. This chapter examines branching strategies that were previously unexplored. I devise a novel heuristic methodology for the purpose of identifying the primary three branching strategies among Github projects. This novel method employs the names and counts of branches within the project, and it is applicable to any project that has been retrieved from the GitHub platform. In addition, I present general data regarding usage preferences and the correlation between the repository structure of projects and the branching strategy.

Chapter 5. I develop a novel method for determining the productivity level of the software development process. I employ the repository's activity, branching strategies, repository structure, and other development process parameters in this implementation. This novel method allows one to examine the relationship between productivity and other parameters of the software development process. Next, I develop a machine learning model that predicts the development period of projects based on the initial parameters provided, such as the size of the team and the programming language used.

Chapter 6. I develop a novel concept for the describing collaboration of software development teams. With this new understanding, I developed a mathematical model for calculating the developer team collaboration and represented it by a single number. Then, I develop a model that recommends the optimal size of the developer team for a given project based on its initial parameters, such as the programming language and planned development period, with the assistance of the above methodologies and my novel approach for collaboration.

Chapter 2

Background

This chapter presents basic concepts that are necessary for the better understanding of the dissertation.

Repository Structure

In essence, the repository structure is the type of structure that was established for the primary components of the project. This thesis examines two primary structures; namely Mono and Multi repository structures. A Mono repository is a method which the primary components of a project are stored in a single directory. Conversely, in the multi-repository approach, the primary components of the project are maintained in distinct repositories.

Version Control Systems

In general, version control systems are the instruments that assist developers or project owners in the management of their source codes. There are two fundamental methodologies; these being distributed version control systems (DVCSs) and centralised version control systems (CVCSs). The primary distinction between these two methodologies is their approach to source code management. CVCS employs a single central server to store all versioned files, and clients retrieve files from this central location. In distributed version control systems (DVCSs), each user is granted access to the complete source code of the project, which includes its entire history.

Branching strategies

One of the critical components of the software development process is the branching strategy. This concept was initially introduced in central version control systems (CVCSs) and subsequently in distributed version control systems (DVCSs). The structural nature of this concept is the main reason why it is employed in DVCSs. Developers have the ability to generate their own copy of the source code and implement their own modifications, such as the addition of new features or the resolution of bugs. A new version of the source code in the branch is merged into the main source code at the conclusion of the process.

Github Platform

One of the most widely used version control systems is Github. Github is a significant repository platform, with over 480 million repositories, according to some estimates. This vast repository collection is highly beneficial for research. The Github platform uses the DVCS method, which also makes extensive use of branching strategies. This is the reason why this platform has become even more critical to our research.

Github API

One of the most critical tools that the Github platform implements is the Github API, which is designed to assist researchers and developers. This API enables the retrieval of nearly all types of information regarding a repository and its owner. Special queries can typically retrieve this information, and they can be customised to meet specific requirements.

Multi Repository Management Tools

The management of Multi repository projects is significantly more challenging than that of Mono repository projects due to their intricate nature. Updating all components of the project simultaneously can be a significant challenge, as it can require an extensive amount of time for large projects. Multi-repository management tools (MRMTs) have been created to address this issue and numerous other similar ones.

Chapter 3

Thesis Group I: Mono and Multi repository structures and Identification process

I carried out detailed research in the area of repositories and repository structures. My work mainly focused on Mono and Multi repository structures and here I provide a new detailed definition for each of these structure types. These new definitions are based on my own research and findings related to these two repository structures. Besides this, I devised a special algorithm for the identification of frontend and backend repositories. This approach applies Machine Learning and the File Structure of repositories. There is almost no other approach around for such purposes. Then I have also created a new algorithm for the identification and collection of Mono and Multi repository projects. There is no other similar approach available for this, and the efficiency of this method has been proved by measuring the success rate. I used KNN classification and Machine Learning methods for this.

Publications related to this thesis group: [J1], [J2], [C5]

To date, we have clarified the fundamental principles of software architecture and explored the primary architectural types. This chapter offers a thorough examination of the repository structure, which is one of the critical components of software architecture. This chapter also provides a thorough examination of two primary repository structures; namely Mono and Multi repository architectures. And we introduce a machine learning model that is specifically designed to accurately identify frontend and backend repositories, which are essential components of the Multi repository framework.

Following this, we present an algorithm that was carefully developed to identify projects that belong to one of the two usual repository structures, which is described later on.

In the final section we provide a compilation of a set of analyses that focus on the parameters of software development projects and their compatibility with the chosen repository structure.

3.1 Introduction

The field of software development is constantly evolving and improving. Developers and researchers in this field are undeniably benefitting from this ongoing advancement. Nevertheless, certain factors, such as application architecture, can present specific challenges for this dynamic improvement process. It is widely acknowledged that software design and architectural patterns offer generic and reusable solutions for a variety of challenges, including the achievement of high availability and the mitigation of risks within the business context, as well as constraints in computer hardware performance [4].

Special consideration must be given to repository structure, which is a critical component of software architecture. Repository structures have been the subject of ongoing discussions and debates since 2010. The reproduction of version control systems may be partly responsible for this increased interest. Here we will investigate the two primary repository structure types, as well as the distinctive characteristics and implications of the various types of version control systems.

In general, there are three primary repository structures called Monorepo (also known as mono repository), Multirepo (also known as multi repository or polyrepo), and a combination of these two architectures, which is referred to as Metarepo (also known as meta repository) [5].

3.2 Related Works

As luck would have it, the academic community is hampered by a shortage of direct comparative research that concentrates on Mono and Multi repository structures. To fill this gap, we utilise the Multivocal Culture Review (MLR) as a fundamental resource for establishing our comprehension [6]. This review primarily relies on the valuable information obtained from a variety of blogs and articles.

It is necessary to conduct a comprehensive analysis of the existing literature in order to determine the advantages and disadvantages of these two architectural approaches. To streamline this analysis, we will assess these structures according to predetermined dimensions.

1. Accessibility (Visibility): The Mono repository structure has a unique advantage in the form of a meticulously organised hierarchy [7]. In this context, we presuppose that all developers have access to all aspects of the project, despite the possibility of misalignments with company policies. In this context, the insights obtained from Google's developer survey provide penetrating views [8]. The survey emphasises the developers' gratitude for the capacity to investigate and gain visibility into various segments of the codebase, a capability that is also feasible in a Multi repository approach. Developers can develop a comprehensive understanding of the project by examining the interactions and workflows of various components within the repository. Notwithstanding, the survey emphasises that these advantages may occasionally manifest themselves as disadvantages within the Mono repo architecture. In the area of dependency management, a significant challenge is presented here. In mega corporations such as Google, projects that are housed in a single repository can grow to massive proportions and contains billions of lines of code. The modification of dependencies within such colossal projects can present significant challenges, such as the predicament of diamond dependencies [9].

2. Tooling: This aspect is one of the primary constraints associated with the Mono repository structure. In the majority of cases, developers are obliged to use identical programming languages and management tools. In contrast, the Multi repository approach does not impose such stringent requirements. Team members are granted the freedom to utilise a wide range of programming languages and tools that are most compatible with their needs as they collaborate on discrete project segments that are stored in separate repositories. This creates a conducive environment in which each team member operates within their comfort zone.
3. Functionality and Security Testing: the autonomous testing of security and functionality within isolated repositories is a significant advantage inherent to a multi-repository architecture [10]. Nevertheless, this situation also introduces certain problems into the version control process, which often lead to specific constraints. The existing literature review states that the majority of studies that compare these two repository structures tend to concentrate on fundamental aspects, typically focusing on projects developed by the same developer or within a single corporate entity.

Hence, while the scope of academic research that directly compares the Mono and Multi repository structures is restricted, we can derive invaluable insights from alternative sources, particularly the Multivocal Culture Review and other relevant articles. By meticulously evaluating factors like accessibility, tooling, and security/functionality testing, it is possible to gain a better overall understanding of the benefits and drawbacks of each architectural approach.

3.3 The Roles of Software Architecture

The primary function of software architectures is to serve as an intermediary between the project requirements or tasks and the actual application; or, in other words, the source code. This intricate relationship can be effectively represented visually, as shown in Figure 1.1. This graphical representation clearly represents the software architecture in the intermediary zone, thereby illustrating its critical role as a bridging device. It should be added that similar visual representations, albeit with minor variations in nomenclature, have been described in a variety of other research endeavours.

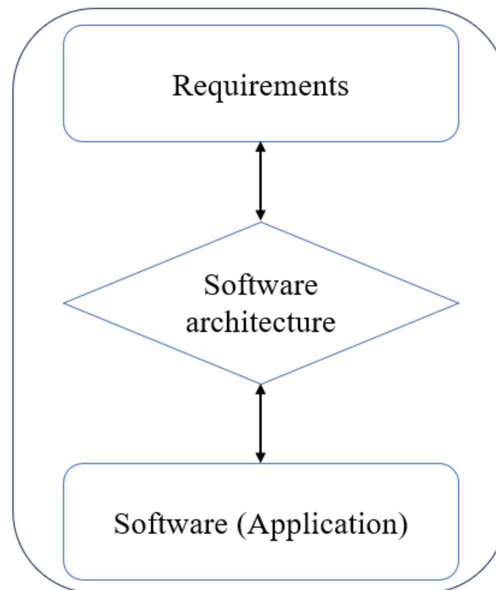

Figure 3. 1 The role of software architecture in a project.

The setting place of software architecture within the software development lifecycle is graphically illustrated in Figure 3.1. Next, it is imperative to acknowledge the distinct functions that software architecture performs during this lifecycle. Despite the fact that there have been numerous discussions and debates about these roles, they are generally defined as follows:

1) Understanding: Software architecture is a method for understanding the overarching workflow of the software and the interconnections among its various elements from a high-level perspective.
2) Reuse: Architectural descriptions enable reuse at multiple levels. Component libraries and frameworks are the primary focus of contemporary efforts in the area of reuse. Domain-specific software architecture types or design patterns may be included in these frameworks [11, 12].
3) Construction: An architectural description provides a fundamental framework for development by outlining the primary components and their interdependencies [13].
4) Evolution: Software architecture offers an insight into the dimensions along which a system is expected to evolve. Maintainers can increase their understanding of the effects of modifications by explicitly defining the "load-bearing walls" of a system. This, in turn, allows for more precise cost estimates of any major changes.
5) Analysis: Architectural descriptions permit different types of analysis, such as checking for system consistency, making sure that the architecture follows the rules of a certain style, ensuring that the architecture meets specific quality standards, performing dependency analyses, and domain-specific analyses that are adapted to architectures built in certain styles [14–16].
6) Management: There is empirical evidence that creating a workable software architecture provides a significant milestone in industrial software development processes, which in turn facilitates the success of projects [17].

7) Communication: An architectural description is frequently employed as a tool for communication between project managers and stakeholders. For example, open architectural design reviews offer stakeholders a platform to express their opinions on the relative prioritisation of quality attributes and features when confronted with architectural trade-offs [18].

3.4 Software Architecture Styles

As previously stated, an architectural style is delimited by a specific set of principles governing system construction and component interaction for data manipulation. These principles encompass various aspects, such as the sequential processing of data, and the hierarchical arrangement of components. We should acknowledge that each architectural style exerts a varying influence on the enhancement or diminishment of certain quality attributes. Consequently, each architectural style has its own unique set of advantages and drawback.

3.4.1 Service-oriented Architecture.

Similar to the multifaceted nature of software architecture, Service-Oriented Architecture (SOA) also boasts several interpretations and definitions:

SOA is a software architectural framework founded upon fundamental concepts encompassing an application front-end, services, a service repository, and a service bus. A service, within this context, comprises a contractual agreement, one or more interfaces, and an implementation [19].

SOA represents a variant of distributed systems architecture, commonly characterised by distinct attributes, including a logic-centric view, message-based communication, emphasis on descriptions, granularity, network orientation, and platform neutrality [20].

Service-Oriented Architecture (SOA) serves as a paradigm for orchestrating and harnessing distributed capabilities that may be dispersed across divergent ownership domains [21].

In overarching terms, the principal objective of SOA may be defined as the establishment of loose coupling among interacting software agents. According to this framework, a service is a discrete piece of work that a service provider undertakes with the intention of achieving particular goals for a service consumer.

3.4.2 Object-oriented Architecture.

Within this architectural case, data representation and operations are encapsulated within the framework of an abstract data type or object. The core components of this paradigm revolve around instances or objects that embody these abstract data types. These objects work together by invoking functions and procedures to facilitate interactions among themselves. The overarching problem is broken down into a set of smaller, more manageable sub-problems, with solutions to these sub-problems shared amongst others via the invocation of procedures. This cooperative procedure invocation culminates in the resolution of the overarching problem. It is worth noting that this

approach can be implemented in either a multithreaded or single-threaded manner, offering flexibility in its application. *Figure 3.2* provides a visual representation of this architecture, to help clarify the process.

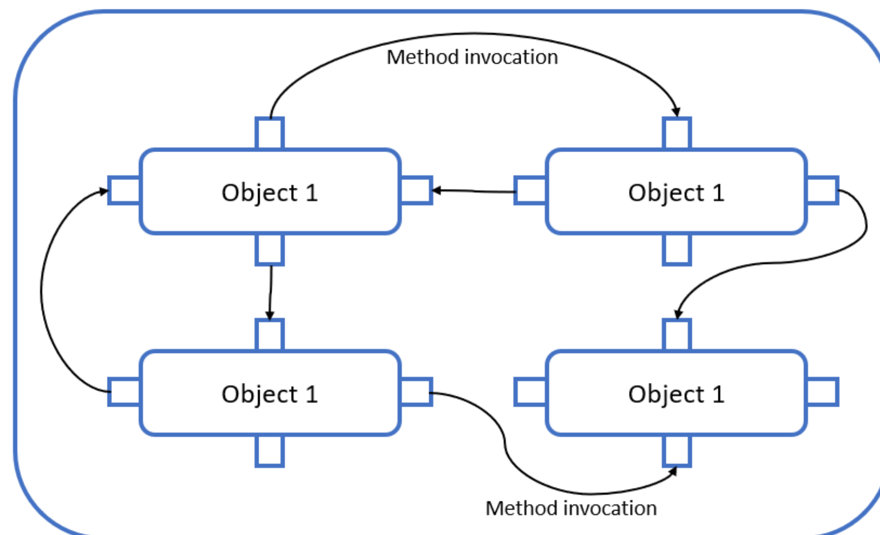

Figure 3. 2 A visual presentation of an object-oriented architecture.

This architectural style provides unique benefits, especially the principle of data hiding, which allows for the modification of an object without affecting its clients, thereby reducing the risk of unauthorised access and related vulnerabilities. Also, this architectural framework enables developers and software designers to systematically divide intricate problems into assemblies of interacting objects, thereby improving modularity and maintainability. Nevertheless, this architectural paradigm is not without its drawbacks, which are primarily related to the interconnectedness of the objects within the system. A change in the type of one object requires corresponding notifications to be propagated to all other objects with which it shares connections, as all objects maintain connections with one another. This interdependence has the potential to introduce complexities and overheads in the management of object relationships and changes.

3.5 Thesis I/1: Repository Types

I introduce a new type of definition for Mono and Multi repository projects. This new definition incorporates both the development and management features of each repository structure.

Publication related to this thesis: [J1]

In the field of information technology, debates regarding repository types are not at all new. Nevertheless, the relevance of these discussions depends upon the particular context the question is posed. It is essential to establish the environmental context in which repositories are being evaluated. Each user project is meticulously organised and preserved within repositories within the GitHub ecosystem. It should be mentioned that these repositories extend an open invitation to other

developers, enabling them to access and, in certain cases, actively perform modifications to the source code, among other things.

The primary focus of the Multi repository environment is on two primary repository categories called frontend and backend repositories. In order to clarify the nature of these repository types, we will use the example of a Web-based project. Frontend repositories are the components of a Web application that users interact with directly through their Web browsers. Typically, these repositories are responsible for tasks such as establishing communication with the application's backend counterpart, processing user inputs, and rendering the user interface.

In contrast, a backend repository is the repository that contains the codebase and backend data that are essential to a Web application. The server-side backend of a Web application is responsible for various critical functions, such as data management, data storage, business logic implementation, and process execution.

It should be remarked that the characterization of a repository can be easily determined by evaluating its contents. In fact, this serves as the foundation for the identification of a variety of repository types. And there may be other repository types besides the above frontend and backend repositories. However, these ancillary repositories frequently serve a less critical function in the overall software development process or project. In numerous cases, these repositories are designated for supplementary functions, including the documentation of project workflows or the storage of third-party elements.

These repository structures can be defined in the following ways, as in my studies:

Mono repository (Monorepo): A Monorepo is a centralised development approach that permits the integration of codebases for a variety of projects, as well as their histories, branches, and dependencies, within a single, unified version control system. This simplifies the management and coordination of projects.

Multirepository (Multirepo): A Multirepo strategy involves the management of the codebase of each project or component in separate, distinct version control repositories, thereby allowing each project or component to operate independently in terms of development, versioning, and release operations.

3.5.1 Thesis I/2: Machine Learning Method for Repository Identification

Almost no algorithm or approach exists for the identification of frontend and backend repositories, as seen in our literature review and research. I developed a novel methodology that employs the repository's file structure to determine its type. The effectiveness of the method was demonstrated through tests, and a machine learning model was trained with the assistance of a specialised database.

Publication related to this thesis: [C5]

The Multi repository structure identification of a variety of repository types is of the utmost importance. Here, we explore the complexities of the identification process by utilising the GitHub API [22], a platform that has been meticulously designed for software hosting. GitHub is a model of utility for both developers and users, providing them with a variety of features and benefits, such

as project access control, bug tracking, software feature requests, task management, and version control. GitHub has an impressive user base of over 100 million developers and hosts a staggering 372 million repositories as of January 2023 [23]. This enormous data repository now means GitHub has the reputation of being a priceless resource for researchers, which has led to a series of academic articles devoted to the subject of GitHub mining [24, 25].

Researchers encounter obstacles when attempting to exploit the capabilities of GitHub, despite the huge research potential it offers. Despite the fact that the GitHub API provides a plethora of repository-related data, it does not inherently have the ability to clearly determine whether a specific repository is classified as either frontend or backend. The developer typically makes this choice and may decide to use additional tags or designate a repository with a special nomenclature, such as "vault-frontend" [26]. However, in numerous cases, developers refrain from incorporating explicit identifying tags or nomenclature, and this gives rise to the use of alternative methodologies.

A machine learning (ML) model was meticulously developed to facilitate the identification of frontend and backend repositories within the provided GitHub repositories. The training of this ML model was contingent upon the availability of a suitable sample dataset to facilitate effective learning.

Several human-based methods for discerning frontend and backend repositories on the GitHub platform were employed:

- Scrutinising the repository name for explicit frontend or backend indications. For instance, repositories bearing names like "frontend" or containing descriptive terms in their titles hinting at frontend involvement [27].
- Analysing the repository's description. Typically, this method complements the initial approach of examining the repository name, with the repository's description providing supplementary clues.
- Inspecting configuration files. This method is mostly applicable for Web-based applications, where discernible indicators may be embedded within the configuration files to ascertain the repository's type.

As luck would have it, these methods are designed for human experts, and they may not be universally applicable to a broader spectrum of repositories. Nevertheless, these methods were meticulously implemented to create an experimental database for the purpose of initial testing.

The subsequent stage involves the aggregation of the file structure within these repositories, following the compilation of all repositories. Unfortunately, the Github API does not provide a query type that is specifically designed to acquire this information. However, an alternative method is implemented in this context, which capitalises on an existing request that permits the retrieval of a list of files that are nested within a designated directory. The repository archive's comprehensive "file structure" is constructed upon this roster. It should be noted that this initiative necessitates the submission of numerous requests to Github for each repository. A specialised authentication key [28] is judiciously employed to streamline the process in the light of the stipulated limitation on the volume of requests permissible on Github. The Github API allows for the dispatch of a maximum of 5,000 requests per hour. One may choose to monitor the remaining query capacity or calculate the number of folders present in the repository to evaluate the utilisation of query allowances. The latter approach is prioritised here as each query exclusively discloses files and folders that are located within a specific directory. As a result, the script is obligated to

generate a new query each time it encounters a novel folder. The format of a prototypical query string, which is intended to retrieve repository contents, may be like the following:

https://api.github.com/repos/{repository owner}/{repository}/contents/{folder path}

3.5.2 Training the machine learning model.

The development of a database for training purposes is merely the initial phase. While the authors of [29] provide a comprehensive explanation of the process of creating a database and training model, we will look at the most critical components. Another step involves selecting a training model. During this training period, the majority of the models are tested. The accuracy score for a few of them is given in Table 2.1.

	Algorithm	Accuracy Score
	SGD Classifier	0.514648
	Logistic Regression	0.484489
	Multinomial Naïve Bayes	0.514648
	Ridge Classifier	0.484489
	K Nearest Neighbour	0.8778543
	Decision Tree	0.9032744

Table 3. 1 The accuracy scores for different algorithms applied during the training process.

Table 3.1 provides a comprehensive summary of the overall results that were obtained from a variety of algorithms. The archive file structure is interpreted as text that resembles natural language, as previously stated. As a result, it was anticipated that the multinomial Naïve Bayes algorithm would produce superior results compared to other algorithms. Nevertheless, it was found that its performance was inferior to that of the majority of other algorithms and methods. In contrast, the decision tree algorithm yielded the best, most useful results. It is important to note that certain algorithms, including multinomial Naïve Bayes, may incorporate supplementary parameters to improve their accuracy. However, it is unlikely that these changes will have any big effect on the numerical results.

3.6 Thesis I/3: The identification algorithm used Mono and Multi repository projects

Additionally, I developed a novel algorithm for the identification and collection of Mono and Multi repository projects. The success rate of this method demonstrates its outstanding effectiveness. For this, I implemented KNN classification and machine learning methodologies.

Publication related to this thesis: [J1], [J3]

It is not sufficient to distinguish Mono and Multi repository projects on GitHub or other platforms by defining frontend and backend repositories. There are numerous online GitHub archives and databases that store the details of GitHub repositories [30, 31]. However, none of these databases separate repositories according to the Mono or Multi repository projects, and there is no indication of this. Consequently, it become necessary in our research to construct a special database that contains the parameters of Mono and Multi repository projects. An algorithm was used to identify and collect both Mono and Multi repository projects for this database [32].

3.6.1 The identification of Mono repository projects

Given the immense number of repositories on Github, which exceeds 372 million [23], the identification of repositories for inclusion in our database may be a daunting task. Additional parameters were incorporated into the URLs to expedite the acquisition of desired results and streamline our search. To serve as an illustration, consider the following process for constructing a query URL:

https://api.github.com/search/repositories?q=fullstack+language:java+created:2023-01-01..2023-06-30

By utilising these URLs, we can obtain a list of repositories that were created between January 1, 2023, and June 30, 2023, and they contain the term "full stack" in either their name or description. The repositories must be authored in Java. This methodology facilitates the identification of potential Mono repository projects. The file structure of Mono repository projects is the determining factor in their identification. These projects consolidate all essential components within a single folder, necessitating the identification of only a few key folder names within the file structure. Examples of such home directory names include *Frontend, Backend, UI, API, Client, Server, UI, Front,* and *Back*. The process of identifying and collating Mono repository projects may be delineated by may following set of tasks:

1. Compile a list of potential Mono repository projects and their respective users.
2. Analyse all repositories that belong to the identified Github users.
3. If the analysed projects meet the stipulated criteria for validity, append their names to the temporary database.

4. Retrieve all requisite project data in JSON format and archive it within the database.

3.6.2 The mathematical specification of an algorithm for the Identification of Mono repository projects

As previously discussed, the Mono repository framework is a software development approach that involves consolidating all the source code of a project within a mono repository. To formally characterise the structure of the Mono framework, we can apply set theory and formal logic. Let R denote the Mono repository, and S represent the set of all source files contained within the repository. We can define R as a collection of subfolders, each containing a subset of S. Formally, we can represent R as:

$$R = \{D_1, D_2, \ldots, D_n\}$$

where each D_i is a subdirectory of R and is defined as:

$$D_i = \{ f_i \mid f_i \in S \wedge f_i \text{ is in the directory } D_i \}$$

This notation means that each D_i is a subset of S, containing only the files that are located within that subdirectory. We can use set operations to specify relationships between directories, such as union ($\cup$) and intersection ($\cap$). For example, we can specify that a subdirectory Dk is a subset of another subdirectory Dj by stat:

$$D_k \subseteq D_j$$

We can also define relationships among the files within the repository using logical operators. For instance, we can specify that a file f_j is dependent on another file f_i by stating:

$$f_j \text{ depends on } f_i$$

This notation means that the code in a file f_j relies on the code in a file f_i to function properly. Overall, the formal specification of a Mono repository structure involves defining the repository as a collection of subdirectories and specifying relationships between directories and files using set theory and logical operators.

3.6.3 The identification of Multi repository projects

The methodology for identifying and aggregating multi-repository projects is similar to that of Mono repositories. At the beginning, a catalogue of potential Multi repository projects is compiled, mimicking the process for Mono repositories. However, the difference apparent in the way which Multi repository projects are identified. The user repositories are classified into three distinct categories called frontend, backend, and others, as mentioned in [29]. This is achieved via a machine learning methodology. This method utilises the file structure of frontend and backend

repositories to facilitate training. Once these categories have been created, the matching process begins, provided that there is at least one repository in both the frontend and backend clusters.

The K-Nearest Neighbours (KNN) algorithm was implemented to evaluate the influence of a variety of feature combinations on the success rate of matching. KNN is a supervised learning technique that is frequently used for classification tasks. It assigns a class label to a query instance based on its nearest neighbours within the feature space. The success rates were recorded relative to standard benchmarks after testing multiple feature combinations. The success rate represents the fraction of repositories that are correctly matched relative to the total number of instances. Here, the following feature combinations were examined:

All settings: repository name, programming language, framework, database type, developer list, file structure, and a "36% success rate" readme file (success rate: undisclosed).

- All settings except the Readme file: repository name, programming language, framework, database type, developer list, and file structure (success rate: 58%).
- All settings except the file structure: repository name, programming language, framework, database type, developer list, and a readme file (success rate: 61%).
- repository name and readme file: Just the repository name and readme file (success rate: 85%).

The experimental results indicate that feature combinations have a significant effect on the success rate of matching frontend and backend repositories. The "readme file" and "repository name" combination achieved the highest success rate, which is not disclosed in the text. This underscores their effectiveness in accurately identifying matches between repositories. This finding highlights the importance of textual data, which includes project names and accompanying readme files, in the efficient identification of related repositories within a project.

The final two steps closely resemble those observed in the Mono repository scenario upon the conclusion of the matching process. As a result, a comprehensive process of assembling multi-repository projects can be summarised as follows:

- Compile a roster of projects exhibiting the potential for multiple repositories along with their respective users.
- Classify the unearthed GitHub user repositories into three designated groups.
- Execute repository matching within the Frontend and Backend clusters to pinpoint multi-repository projects.
- If the identified projects satisfy the criteria for validity, incorporate their names into the temporary database.

3.6.4 The mathematical specification of an algorithm for the Identification of Multi repository projects

To formalise a Multi repository structure, we can apply set theory and formal logic. Let R_1, R_2, ..., R_n represent the individual repositories comprising the Multi repository structure, and let S denote the set encompassing all source files spanning across all repositories. Each repository Ri can be defined as an assembly of source files:

$$R_i = \{ f_i \mid f_i \in S \land f_i \text{ is in the directory } R_i \}$$

This notation explicitly conveys the message that each repository R_i constitutes a subset of S, housing exclusively those files located within its confines. Set operations can be applied to delineate relationships between repositories, employing operators like union (∪) and intersection (∩). For instance, it is possible to declare that a repository R_k is encompassed within another repository R_j by expressing:

$$R_k \subseteq R_j$$

Furthermore, we can delineate connections between the files situated within repositories using logical operators. For instance, we can stipulate that a file f_j within repository R_j relies on a file f_i within repository R_i by asserting:

$$f_j \text{ depends on } f_i$$

This statement indicates that the code within file f_j is contingent upon the code within file f_i for proper functionality. Overall, the formal specification of a Multi repository structure involves defining each repository as a collection of source files and articulating relationships between repositories and files through the utilisation of set theory and logical operators.

3.7 Database of Mono and Multi repository projects

Up until now, we have utilised a machine learning model to identify frontend and backend repositories, and an algorithm to identify Mono and Multi repository projects. In order to create our database for further analysis and exploration, we gathered over 50,000 projects from the GitHub platform. In order to prevent the acquisition of biassed data, we collected projects from a variety of backgrounds, locations, and development years.

The database is a comprehensive collection of 50,552 repositories, consisting of a combined total of 8,479 multi repository projects and 33,594 mono repository projects. In the context of multi-repository projects, each project consists of both a frontend and backend repository. Each project in the database was stored in JSON format to facilitate efficient management and systematic organisation.

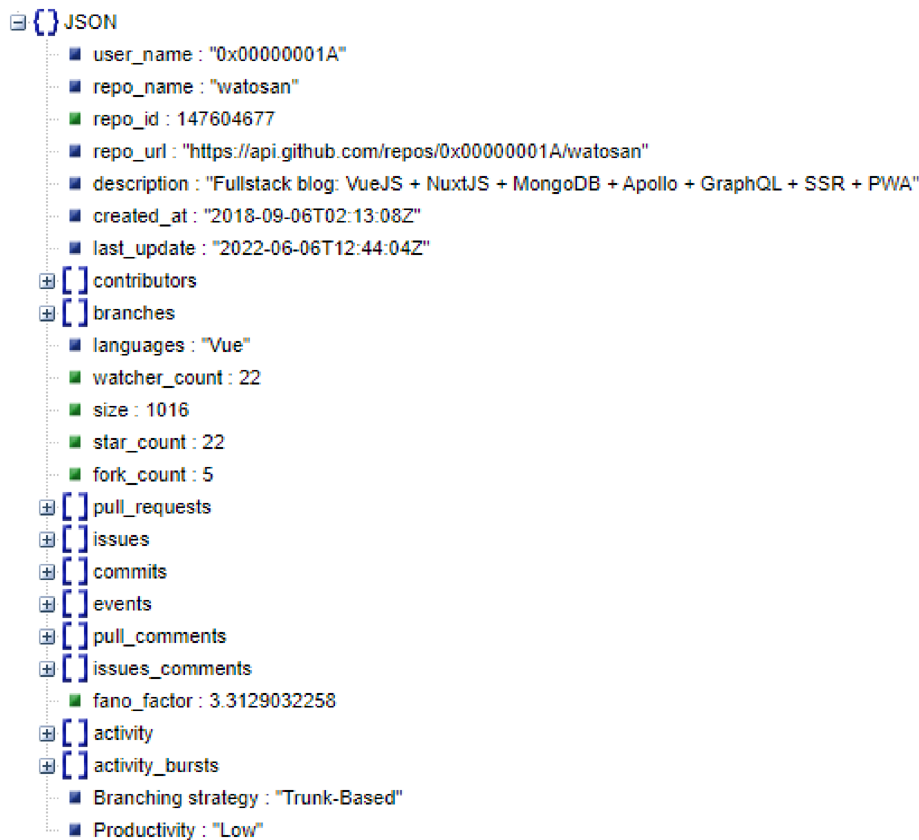

Figure 3. 3 Structure of JSON file in database which represents a Mono repository project.

Each JSON file in the database contains a compilation of essential project configurations, attributes, and functionalities, as shown in *Figure 3.3*. The Github API is employed to execute the data compilation process. The JSON file contains twelve distinct fields, containing the project's nomenclature, Github user handle, Github-assigned repository ID, repository URL, project description, creation date, last update date, and the most frequently used programming language. The first seven fields provide fundamental project details.

Metrics associated with project engagement, including the number of followers, file size in kilobytes (KB), count of stars, and number of branches, are included in the complementary attributes of the JSON file. Utilising the appropriate URL structure, these critical data metrics are converted into the JSON format via API requests, as illustrated below:

https://api.github.com/repos/{user name}/{repository name}

The "Contributors" section provides a comprehensive list of developers who are currently involved in a project, as well as a summary of their contributions in terms of commit counts. This simplifies the process of identifying critical team members. An additional request must be submitted via the Github API to obtain the contributor list. A typical URL for this type of request is as follows:

https://api.github.com/repos/{username}/{repository name}/contributor

The Branches section contains a list of branches within the project, along with the corresponding commits for each branch. This enables us to meticulously examine the lifecycle of each branch. The deployment of a specific URL format within the Github API is required for the retrieval of the branch list. Nevertheless, this request does not by itself furnish a list of commits for specific branches. An additional request must be submitted for each specific branch in order to obtain the commits for that branch. For instance, the URLs for these types of requests are specified below:

Branches:
https://api.github.com/repos/{username}/{repository_name}/branches

A list of commits for a given branch:
https://api.github.com/repos/{username}/{repository_name}/commits?sha={branch_name}

In addition to these sections, there exist six supplementary sections, each showcasing diverse project activity configurations. The creation of each of these sections is achieved by executing distinct URL requests. The structure of a Multi repository project closely resembles that of a Mono repository, with the sole difference being the incorporation of two additional sections at the outset. There are front repository and back repository, corresponding to their respective repository types.

3.8 Analyses of Mono and Multi repository Structures.

The examination of Mono and Multi repository structures and their implications for various aspects of the software development process has not been explored much so far, as was pointed out at the beginning of this chapter. Although some researchers have addressed these points, they frequently confine themselves to either disregarding structural disparities or surveying the experiences of a restricted number of developers who have experience working with Mono and Multi repository methodologies.

Several critical components of software development, such as the prevalence of both Mono and Multi repository paradigms, development duration, and team size, are thoroughly examined below. Our objective is to provide a more comprehensive view of the relationship between these repository structures and the broader landscape of software development practices by exploring these aspects.

3.8.1 Development period in Mono and Multi repository projects

In the preceding section, we emphasised the importance of two repository parameters, namely "created_at" and "last_update" as outlined in the database creation process. The entire development period of software projects can be calculated using these timestamps. This temporal insight allows us to perform a comparative analysis of this duration across various repository structures. In

essence, these timestamps can be used as critical metrics to evaluate project timelines in the Mono and Multi repository structures.

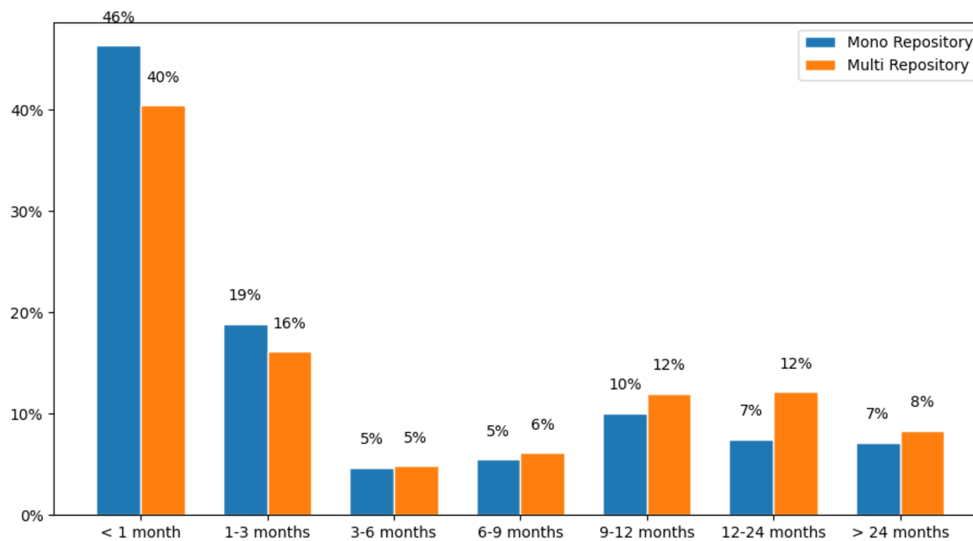

Figure 3. 4 A comparison of development period of two repository structures in percentage terms.

The chart provided provides a visual representation of the development duration distribution for both Mono and Multi repository projects. Here, the chart makes clean that the majority of projects in both categories had a relatively short development period. To be precise less than one month. To be precise, 40% of Multi repository projects and 46% of Mono repository projects were included in this temporal frame. Moving on to the subsequent category, which includes projects with a development period of 1 to 3 months, we observe a similar pattern between Mono and Multi repository projects, namely 19% and 16%, respectively. This observation highlights the fact that the development duration disparity between the two repository structures is negligible for moderately large projects. In the 12–24-month range, the proportion of Multi repository projects slightly exceeds that of Mono repository projects for more extended development durations (12% vs. 7%, respectively). This implies that the use of Multi repository models may be more prevalent in longer-term project scenarios. Nevertheless, there is no significant distinction between the two repository structures for projects with development durations that exceed 24 months.

3.8.2 Developer team size in Mono and Multi repository projects

Similar to the analysis of development periods, databases also allow us to compare the team sizes between both repository structures. This can be achieved by examining the count of developers listed in the "contributors" section of each JSON file for each project.

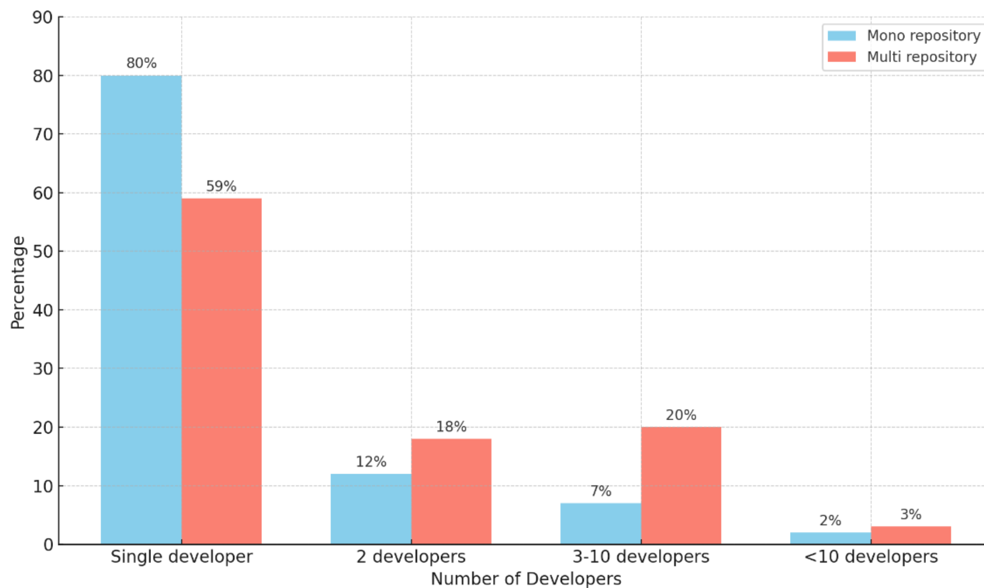

Figure 3. 5 A comparison of the developer team sizes of two repository structures in percentage terms.

The majority of Mono repository projects (80%) were developed by a single developer, as shown in *Figure 3.5*. This suggests that a substantial number of software development projects are conducted by individuals, whether they are in small organisations or a small team. Pair programming or code reviews may be less prevalent in Mono repository projects, as only 9% of projects involved two developers. Only 7% of Mono repository projects have between three and ten developers, which means that large teams are less common. Furthermore, the fact that only 1% of Mono repository projects have more than 10 developers suggests that such projects are relatively rare.

In contrast, the distribution of team sizes is more evenly distributed in multi-repository projects. Although 59% of projects continue to have a single developer, this tells us that the majority of projects have multiple developers, which is higher than the percentage of Mono repository projects. In particular, two developers are involved in 18% of multi-repository projects, which implies that these projects may have more collaboration or code reviews. 20% of Multi repository projects involve between three and ten developers, which suggests that larger teams are more prevalent in this context. Lastly, the percentage of Multi repository projects with more than 10 developers is only 3%, which is slightly higher than the percentage of Mono repository projects.

3.8.3 Popularity trend of Mono and Multi repository projects over the years.

After highlighting some fundamental differences between Mono and Multi repository projects, it is also worthing seeing how the usage percentage of these project structures has evolved over the years. These findings can provide insights into how the demand for these repository structures has shifted over time.

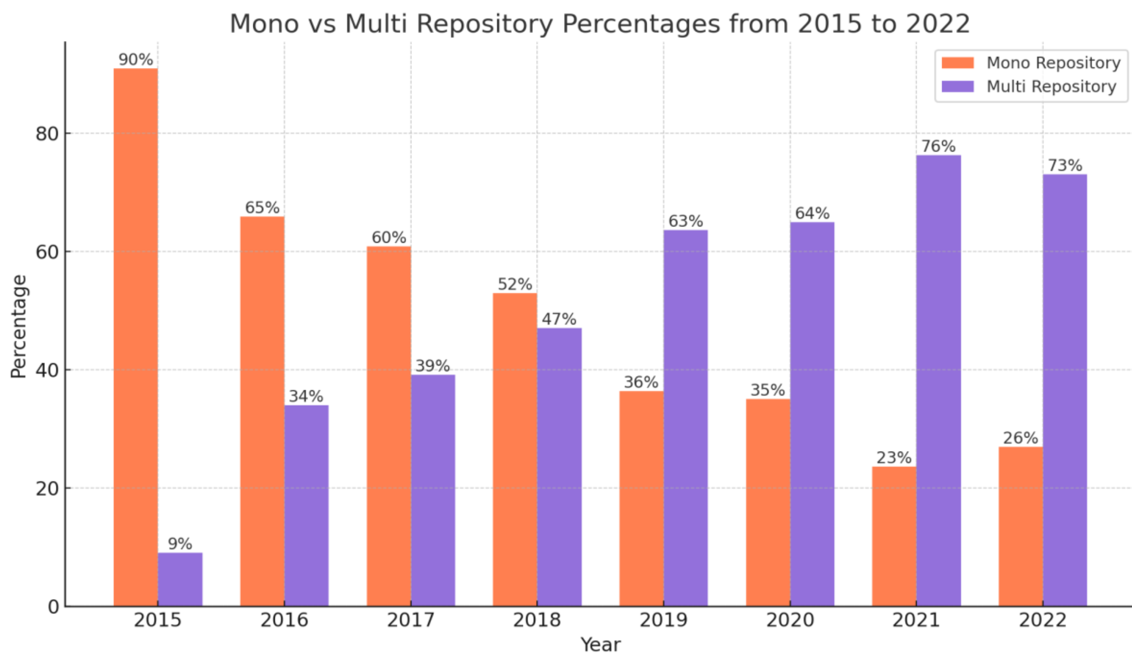

Figure 3. 6 A comparison of developer team sizes of two repository structures in percentage terms.

The Multi repository approach has grown substantially over the years. In 2015, it was nearly non-existent; however, by 2018, it began to establish itself as a competitor to the Mono repository approach. The Multi repository approach experienced a substantial increase in use from 2019 onward, whereas the Mono repository approach remained relatively stagnant in comparison. This trend suggests that there is a growing preference for the Multi repository approach over the mono-repository approach, a preference that appears to be increasing annually.

This change is especially noteworthy in light of the difficulties associated with identifying multi-repository projects. In spite of these challenges, the number of projects with multiple repositories continues to exceed projects with a single repository. This fact reinforces our view regarding the increasing prevalence of the Multi repository approach. The reasons for this change in preference for repository structures over the years are multifaceted and intricate. The subsequent chapters and sections of this dissertation will delve deeper into these factors and provide a more detailed examination of the evolving landscape of repository structure choices and their implications.

3.9 Thesis I/4: Multi Repository Management Tools

A novel heuristic methodology was developed to identify various multi-repository management tools. This approach relies on the identification of these tools based on their configuration files, also referred to as signatures.

Publication related to this thesis: [J2]

One of the most significant distinctions between multi-repository projects is that they occasionally require the use of supplementary tools to coordinate the management of all the project components. The authors of [33] provide a comprehensive description of these tools, which are referred to as Multi Repository Management Tools (MRMT). However, here we will describe some of the main aspects of these tools and the environment in which they are employed. Initially, the primary obstacle was the identification of these tools. In the majority of cases, it is straightforward to identify the most frequently used MRMTs; however, some of these tools are not as easily identifiable, despite their widespread use by developers. The majority of these tools were created by small businesses or individuals. It is essential to analyse these tools to ascertain the trends in this area. Major corporations like Github frequently adopted a new feature that small MRMTs had been promoting. Here, we prevent a method for identifying these tools that is based on their signature files. We found that the majority of the MRMTs have their own configuration file, which may be located in the project folder where they were utilised, during our analysis. A small table was generated by compiling this list of files, where each signature file denotes a single tool.

Name	Feature	Signature
Mepo	*Update, compare, compere, save*	components.yaml
MR	*Update, list, offline*	.mrconfig
Pull	*Update (all forks)*	pull.yml
West	*Combine all repos, update, list*	west.yml
Mention-bot	*Improve review and pull requests*	.mention-bot
Zappr	*Improve review and pull requests*	.zappr.yaml
Mrgit	*Manage project build process*	mrgit.json
Talan CLI	*Manages third party elements*	.tln.conf
DevOps & Swarm-mode	*Manage project build process*	docker-compose.swarm.yml
Lerna	*Monitor development process*	lerna.json

Table 3. 2 A list of MRMTs with their key features and signature files

It is evident that some of the tools mentioned have similar objectives when they are examined. This enables us to identify their key features and illuminate Github's limitations. The following are a few critical components:

1. *Repository Updates*: Despite the fact that they may employ varying methodologies, numerous tools are designed to streamline the process of updating multiple repositories simultaneously. This can significantly improve the overall development speed and save time for developers and project owners. However, the management of updates in extensive projects can be a challenge, and these tools provide a variety of solutions to address these challenges.
2. *Development Oversight*: Although Github provides a comprehensive overview of individual repositories, it is unable to monitor multiple repositories simultaneously. The creation of specific tools that provide insights into contributors, job roles, commitments, and other pertinent information has satisfied this requirement. This information is essential for the efficient management of multi-repository projects.
3. *Management of Third-Party Components*: Although not entirely addressed in multi-repository project management, this aspect is of great importance in this context. Managing a variety of dependencies and packages can be a complex task in multi-repository projects, and Github does not offer any built-in tools or commands to facilitate this process. Nevertheless, the aforementioned tools effectively manage these factors, thereby saving developers a significant amount of time.
4. *Enhanced Review and Pull Request Handling*: While this functionality isn't commonplace in multi-repository management tools (MRMT), some tools excel in this regard. This feature proves especially valuable in large projects with numerous contributors, where managing pull requests can become challenging. These tools aim to simplify and streamline the review and pull request process, ensuring improved collaboration and code integration.
5. *Project Construction Process Management*: This aspect is in accordance with Github's automation capabilities, but it is focused on the simultaneous creation of multiple repositories to guarantee project compatibility. The objective of these tools is to improve efficiency and simplify the project creation process.

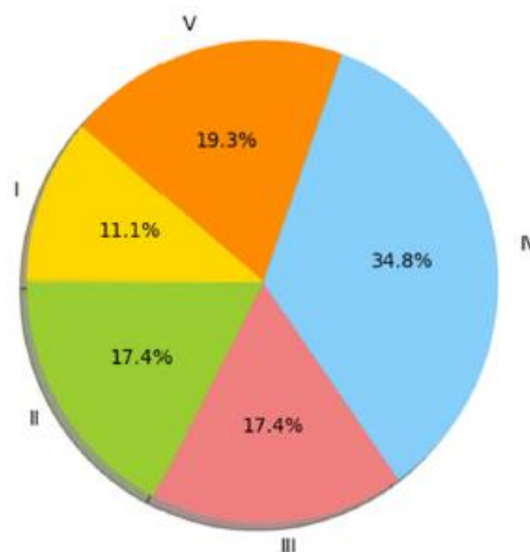

Figure 3. 7 Popularity of each feature among the developers.

The popularity of these features among users is shown in *Figure 3.7*, which considers the user count of each tool. It should be remarked that specific tools may incorporate one or more features from the above list. In such instances, the tool user share is spread across all relevant features to ensure a fair distribution.

Figure 3.7 shows the user count for each tool, demonstrating the popularity of these features among users. It is quite clear that specific tools may include one or more of the features listed above. In such cases, the tool user base is proportionally spread out among all the relevant features, thereby maintaining equity. The ordinal numbers assigned to each feature in the preceding section serve as their identification.

Figure 3.7 offers valuable insights into the primary features of Multi repository management tools (MRMTs). The significant demand for these features and the potential revelation of a gap in the Github platform are underscored by the fact that approximately 35% of users prefer tools with advanced review and pull request capabilities. This demand is influenced by a variety of factors, with the sophisticated of the connection system being a significant factor. In Multi repository projects with numerous branches, these capabilities may also present challenges, especially in the review and pull request processes, which may be difficult for project managers to manage. Nevertheless, they are essential.

Although they are not as significant as the advanced review and pull request functions, the remaining three features are almost of equal important to users. These capabilities encompass the facilitation of project builds, the management of third-party elements, and the monitoring of development and collaboration. However, these features do not have the same priority as the fourth feature. It is quite surprising that users do not prioritise the simultaneous updating of all repositories, despite the potential benefits.

3.10 Concluding Remarks

Within the software development domain, Mono and Multi repository structures are acknowledged as two of the most prevalent and primary repository configurations. Although they are widely used, there is a significant lack of scholarly research on their significance, distinctions, and similarities. Here, we offer is a comprehensive review of the conceptualization and impact of both repository structures on the software development lifecycle.

This chapter introduce two critical algorithms for the identification and consolidation of front-end and back-end repositories, as well as Mono and Multi repository projects. Each algorithm works well, and both have provided a significant contribution to the development of my study here. The classification of repositories is determined by the initial algorithm, which employs the file architecture. The robustness of our methodology is demonstrated by the precision rate of our model, which is approximately 90%. Later, another algorithm was developed with the specific aim of identifying Mono and Multi repository projects on the GitHub platform. Our innovative approach significantly broadened the scope of our database in the absence of pre-existing automated procedures for this task and enriched our knowledge and understanding of repository types.

The application of these algorithms facilitated the compilation of a substantial dataset, which allowed us to observe trends within the two types of repository structures. It was noted that the

Multi repository framework has been gaining popularity among developers since 2018. It appears that the Multi repository model is preferred by larger development teams over the Mono approach. And projects with extended development timelines are inclined to utilise the multi repository structure, which suggests that it is the preferred choice for development teams that are involved in complex projects and have a larger scope. In contrast, the Mono repository framework is more commonly used by smaller teams for projects that are simpler and more manageable.

The author of this thesis is responsible for the following contributions presented in this chapter:

I / 1. My first contribution is the creation of an algorithm that uses a machine learning model to identify and collect frontend and backend repositories from the Github platform. The high accuracy rate proves the efficacy of this approach.

I / 2. I also devised another algorithm for the identification and collection of Mono and Multi repository projects on the Github platform. With this, we were able to collect a huge number of projects in a very short time, which was not the case with other research studies like this one.

I / 3. I also created a heuristic approach for the identification of Multi repository management tools among the projects on the Github platform. This approach uses the configuration file for identification and has a notably good accuracy as well.

I / 4. A new heuristic approach was developed by me for the identification of different multi repository management tools. This process is carried out based on the signature files and it may play a crucial role for researchers and project managers in their respective field of study.

Chapter 4

Thesis Group II: Branching strategies in Mono and Multi repository projects.

I introduce a new heuristic approach to the identification of branching strategies in various projects in this chapter. I employ the names of branches and their counts to ascertain the project's branching strategy. At present, this approach concentrates solely on the identification of three primary branching strategies: Trunk-based, GitFlow, and GitHub Flow.

Publication related to this thesis: [J3]

Here, we concentrate on another critical aspect of the software development process: the branching strategy. We begin by exploring the subject of Version Control Systems (VCS), introducing their definitions, various types, and applications in the software development environment. We then examine three primary branching strategies that are frequently encountered in the development sector. We present a methodological framework that was developed to identify these branching strategies within the context of open-source projects. Our investigation of the relationship between the underlying repository structure of software projects and branching strategy is the focus of the remaining sections of this chapter. Our analysis demonstrates the critical relationship between the specific branching strategies that were implemented during the developmental phases and the diverse project parameters used.

4.1 Introduction

The necessity of addressing collaboration and concurrent work within large development teams has become of paramount importance in response to the evolving landscape of software development practices. Distributed Version Control Systems (DVCs) have emerged as a resilient solution to address this exigency, with branching being one of their central features [34]. Branching enables development teams to organise their work into distinct streams and subsequently merge these streams upon the completion of specific tasks. This method is especially well suited for agile development systems, as it allows for a more flexible development process.

The concept of a branching strategy was introduced based on the realization that the majority of DVCs provide support for branching. A development team's approach to managing their branches, each of which provides distinct objectives such as the implementation of new features, bug fixes, and the preparation of the final version, is referred to as a branching strategy. Although there are numerous additional branching strategies, here we concentrate on three of the most frequently implemented strategies in the context of Git [35]. The strategic implementation of branching strategies enables development teams to effectively manage their codebase, optimise collaborative endeavours, and establish efficient workflows during the development process.

4.2 Related Work

Branching strategies have been the subject of analysis since the beginning of 2010. The branching strategies were primarily examined from the perspective of version control systems, with the majority of researchers treating them as one of the parameters [34–37]. This method may be quite accurate; however, it obscures the full significance of the branching strategies, which is why there has been a lack of significant research on them. In this chapter, we demonstrate that branching strategies may be essential during the development process of a project, and they may be associated with various project parameters, such as the size of the team, repository structure, and development.

In their research work, certain researchers attempted to clarify these connections. For instance, in an effort to ascertain the correlation between the productivity of the software development process and the branching strategy, the author of [38] conducted an analysis of nearly 3000 projects. Although this study is interesting, it has certain drawbacks. First of all, researchers examined the branching strategies of about 200 projects. Secondly, the productivity measurement of a specific study was a straightforward process, which we will elaborate on in the next chapter. It soon become clear that that this type of analysis cannot provide an impartial perspective on the significance and role of branching strategies in the software development industry.

A different approach by the researchers is described in [39], which examines branching strategies from the perspective of software team collaboration. The study is based on the findings of an interview with a variety of developers, and it examines their perspectives on specific branching strategies. Unfortunately, the researchers did not concentrate on any particular branching strategy. They conducted an exhaustive analysis of all branching strategies, making it difficult to see the precise impact of particular strategies on the developer workflow.

For this reason, in the subsequent sections, I have selected three primary branching strategies and I have analysed each one separately to get better understanding of their workflow and the impact these strategies have on the development process.

4.3 Version Control Systems

Version control systems represent a foundational aspect of software development, their inception dating back to 1972. The discourse surrounding this critical subject matter spans numerous articles, encompassing diverse foundational concepts and utility scenarios [40–42]. Broadly speaking, version control systems are categorised into two primary types, namely centralised version control systems (CVCS) and distributed version control systems (DVCS).

4.3.1 Centralized Version Control Systems

In articles such as [43, 45], CVCS, an older approach, was thoroughly examined. The primary source repository in the area of CVCS serves as the control base, with each developer operating relative to this central repository. Developers execute a "pull" operation to acquire a specific snapshot of the repository at a specific temporal condition [46]. All project files and their respective versions are stored on a central server or computer within CVCS. Access to specific files or the entire repository is granted to users for the purpose of their work. After making their modifications, users had to "push" them by sending commit messages with them. Later, other users are assigned the responsibility of "updating" their files to ensure that they are in sync with the new version in the repository. It should me mentioned that CVCS exclusively preserves the most recent iteration of the project, which requires a continuous awareness of the modifications that have been made to the source code. Consequently, the repository exclusively retains the most recent version of the code. The utilisation of branches is authorised during the development phase to accelerate the development of new features or functionalities. During the workflow, these branches must coexist with the main project and be copies of the repository. After the testing phase, modifications implemented in these branches may be integrated into the primary source code.

The presence of a single point of failure and diminished performance are the two primary drawbacks of CVCS. Any server malfunction, which causes the cessation of development activities because CVCS depends on a central server, causes the entire project to become inaccessible. Furthermore, each command (such as branching, pushing, and merging) must interact with the server due to CVCS's server-centric nature. This frequently leads to server responses that are significantly slower as a consequence of the increased network traffic.

4.3.2 Distributed Version Control Systems

DVCS operates without a primary central server, in contrast to the previous system. The repository's entirety is replicated on the local computer of each user in the DVCS approach. DVCS is particularly well-suited for extensive projects that are characterised by a multitude of independent developers due to this distinguishing feature [47]. DVCS also distinguishes itself by providing a good performance. And CVCS eliminates the necessity for a network connection by executing the majority of commands locally. Although many terminologies that are pertinent to CVCS are also applicable to DVCS, the latter may have a higher memory consumption as a result of the local

storage of the entire project. DVCS employs compression techniques in moderation to reduce the size of the repository. Despite this limitation, the DVCS paradigm is exceptional in terms of reliability, flexibility, and alacrity.

In summary, DVCS is a significantly more appealing option than CVCS due to its enhanced reliability, performance efficiency, and flexibility.

4.4 Branching Strategies

GitFlow:

- *Complexity*: GitFlow is renowned for its intricacy, and it is often regarded as one of the most convoluted branching strategies in use.
- *Branch Structure*: Its architecture encompasses a "Master" branch, which, in recent years, has transitioned to "Main," serving as the repository's primary location for housing the core source code. In tandem, there exists a "Development" or "Release" branch, instrumental in orchestrating the preparation of new product releases.
- *Feature Branches*: The strategy relies heavily on feature branches, sprouting forth from the Master/Main branch, each dedicated to the implementation of distinct features, subject to rigorous testing.
- *Supplementary Branches*: GitFlow is not devoid of supplementary branches. These ancillary branches come to the fore for tasks such as bug fixes, documentation, and assorted purposes, often gathered beneath the umbrella of feature branches.
- *Advantages*: GitFlow fosters a conducive environment for the simultaneous endeavours of multiple developers, diligently safeguarding the production source code. It aids the seamless resolution of conflicts during merging by allowing developers to concentrate on their respective tasks.
- *Challenges*: Nevertheless, the intricate branch structure of GitFlow can, if not carefully managed, introduce challenges in the testing and other developmental phases.

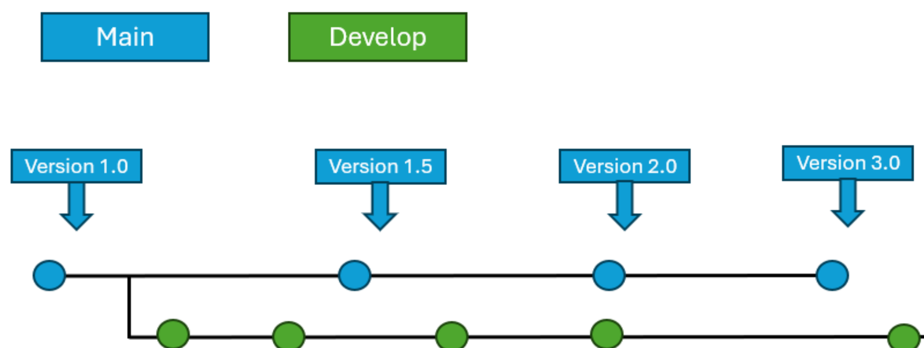

Figure 4. 1 Schematic explanation of GitFlow branching strategy

GitHub Flow:

- *Simplicity*: In stark contrast, GitHub Flow adopts a more streamlined and uncomplicated approach, positioning itself as a simplified variant of GitFlow.

- *Branch Arrangement*: GitHub Flow dispenses with dedicated release or development branches. Instead, it places its reliance chiefly on main branches and feature branches.
- *Utilization*: This strategy has garnered favour within developer circles and enjoys widespread usage, particularly within a significant cohort of Mono repository projects.
- *Objectives*: GitHub Flow excels in scenarios marked by concise, expeditious development phases, furnishing clear perspectives throughout the developmental trajectory.

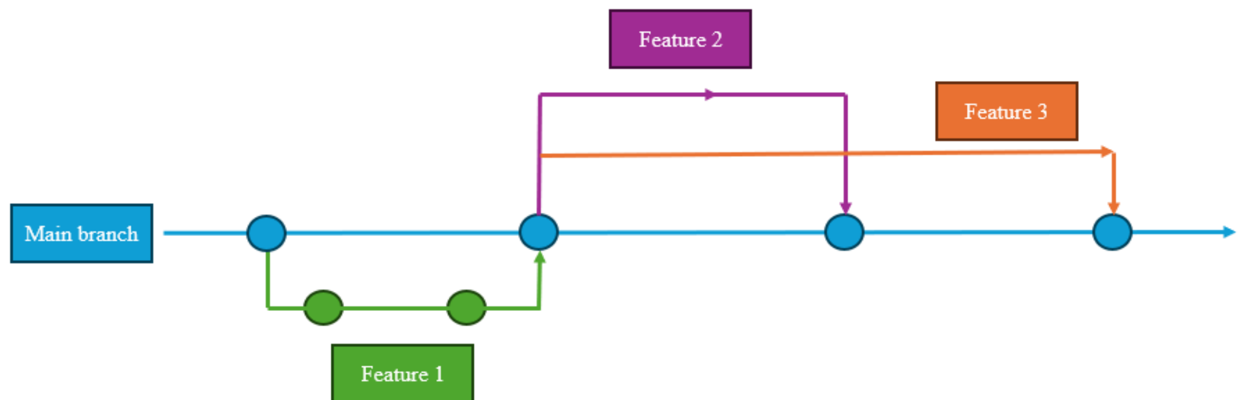

Figure 4. 2 Schematic explanation of GitHub Flow branching strategy

Trunk-based:

- *Simplicity*: Trunk-based represents the epitome of simplicity in the realm of branching strategies.
- *Branch Composition*: In particular, it prescribes the utilisation of a solitary primary branch, one that is invariably poised for deployment.
- *Development Modality*: Under the Trunk-based paradigm, all developmental activities, inclusive of feature integration and bug rectification, unfold directly upon the master branch, without the creation of additional branches.
- *Merits*: The straightforwardness inherent in Trunk-based methodology renders it an appealing choice, especially for lean development teams or novices in the field of software development.
- *Challenges*: Notwithstanding, this simplicity is a double-edged sword, introducing potential challenges, as it allows for a minimal margin of error or oversight. The source code must consistently maintain a state of readiness for deployment.

Figure 4. 3 A schematic description of the Trunk-based branching strategy

These three branching strategies, each with its own distinctive attributes, cater to diverse developmental contexts and preferences, and offer agility and control in software development workflows.

4.4.1 Thesis II/1: Identification of Branching strategy in Open Source Projects

I propose a novel heuristic method for determining branching strategies in various projects. I utilise the names of branches and their respective counts to establish the project's branching strategy. Currently, this method just concentrates on identifying three primary branching strategies, namely GitHub Flow, GitFlow, and Trunk-based.

Publication related to this thesis: [J3]

In the previous chapters, we described three main methods for branching. One key characteristic that sets apart each of these strategies is the range of branches they include. Within this context, determining the branching strategy involves carefully examining the project's collection of branches, taking into account both their names and statistics [44]. As shown in Figure 2.1, the database maintains a complete record of all branches, and identifying them only requires examining their names. Below is a step-by-step algorithm that outlines the process of identifying:

Input: Set of branches in the project
Output: Branching strategy of the project

Function identifyBranchingStrategy(branches):
1. Filter out branches created by automated processes (bots), which will result in a set of non-bot branches.

2. If the number of non-bot branches is 1:

- Check to see whether the branch is a master branch by examining its name.
- If the branch name is "master" or "main", return "Trunk-based" as the branching strategy.
- Otherwise, return "Unknown" as the branching strategy does not match any known strategy.

3. If the number of non-bot branches is greater than 1:

- Check to see whether there is a master branch by examining its name.
- If there is no master branch, return "Unknown" as the branching strategy.
- Check to see whether there is a development branch by examining its name.
- If the development branch name is "dev" or "development", return "GitFlow" as the branching strategy.
- Otherwise, check if there are feature or bug fix branches by examining their names.
- If any of the branch names contain "feature", "bug fix", "bug", or "hotfix", return "GitHub Flow" as the branching strategy.
- Otherwise, return "Unknown" as the branching strategy does not match any known branching strategy.

4.4.2 Mathematical Representation of Branch Identification

GitFlow Branching strategy:

Let M be the mainline branch of the Git repository, and let F be the feature branches, R be the release branches, H be the hotfix branches, and S be the support branches. We can define each branch type as follows:

$F = \{ f_1, f_2, \dots, f_n \}$

$R = \{ r_1, r_2, \dots, r_m \}$

$H = \{ h_1, h_2, \dots, h_p \}$

$S = \{ s_1, s_2, \dots, s_q \}$

where each f_i, r_i, h_i, and s_i is a branch in its respective set.

The GitFlow branching strategy involves creating new branches from M and merging them back into M when they are ready. Specifically, we can define the branching and merging process as follows:

1. Feature branches are created from M:
   $f_i \subseteq M$
2. Changes are made to the feature branch, and the branch is merged back into M when the feature is complete:
   $f_i \rightarrow M$
3. Release branches are created from M:
   $r_i \subseteq M$
4. Changes are made to the release branch, and the branch is merged into M and tagged with a version number when the release is complete:
   $r_i \rightarrow M$
5. Hotfix branches are created from M:
   $h_i \subseteq M$
6. Changes are made to the hotfix branch, and the branch is merged back into M and the release branch that it fixes:
   $h_i \rightarrow M$ and $h_i \rightarrow r_i$
7. Support branches are created from tagged release branches:
   $s_i \subseteq r_i$
8. Changes are made to the support branch, and the branch is merged back into the release branch and M when necessary:
   $s_i \rightarrow r_i$ and $s_i \rightarrow M$

Overall, the GitFlow branching strategy involves creating branches from M and merging them back into M and other branches as needed. The branching and merging process can be formally specified using set theory and logical operators.

GitHub Flow Branching strategy:

Let M be the mainline branch of the Git repository and let F be the feature branches. We can define each branch type as follows:

$F = \{ f_1, f_2, \dots, f_n \}$
where each f_i is a branch in the set F.

The Github Flow branching strategy involves creating new branches from M and merging them back into M when they are ready. Specifically, we can define the branching and merging process as follows:

1. Feature branches are created from M:
   $f_i \subseteq M$
2. Changes are made to the feature branch, and the branch is merged back into M when the feature is complete:
   $f_i \rightarrow M$

Overall, the Github Flow branching strategy is simpler than the GitFlow branching strategy, as it only involves creating feature branches from M and merging them back into M. The branching and merging process can be formally specified using set theory and logical operators.

Trunk-based Branching strategy:

Let M be the mainline branch of the Git repository and let S be the set of all source files. We can define M as follows:

$$M = \{ f_i | f_i \in S \}$$

This notation states that the main branch M is a subset of S, containing all source files in the repository.

The Trunk-based branching strategy involves making changes directly to M and avoiding the creation of feature branches. Specifically, we can define the branching and merging process as follows:

1. Changes are made directly to M:
   $f_i \subseteq M$
2. Changes are committed and pushed to the remote repository:
   $f_i \rightarrow M$
3. Continuous integration (CI) and automated testing are used to verify changes:
   CI $(f_1, f_2, \dots, f_n)$
4. Code reviews and manual testing may be used to verify changes:
   Review $(f_1, f_2, \dots, f_n)$

Overall, the Trunk-based branching strategy involves making changes directly to M and using CI and automated testing to verify these changes. Manual testing and code reviews may also be used to maintain the quality of the code. The branching and merging process may be formally specified using set theory and logical operators.

4.5 Popularity ratio of Branching strategies over the years

Defining branching strategies and checking their workflow is not enough to provide good estimates about their usage ratio. It is vital for developers to have a comprehensive understanding of

the evolving paradigms in branching strategies. This acumen enables them to synchronize their methodologies with prevailing industry norms and to execute judicious decisions.

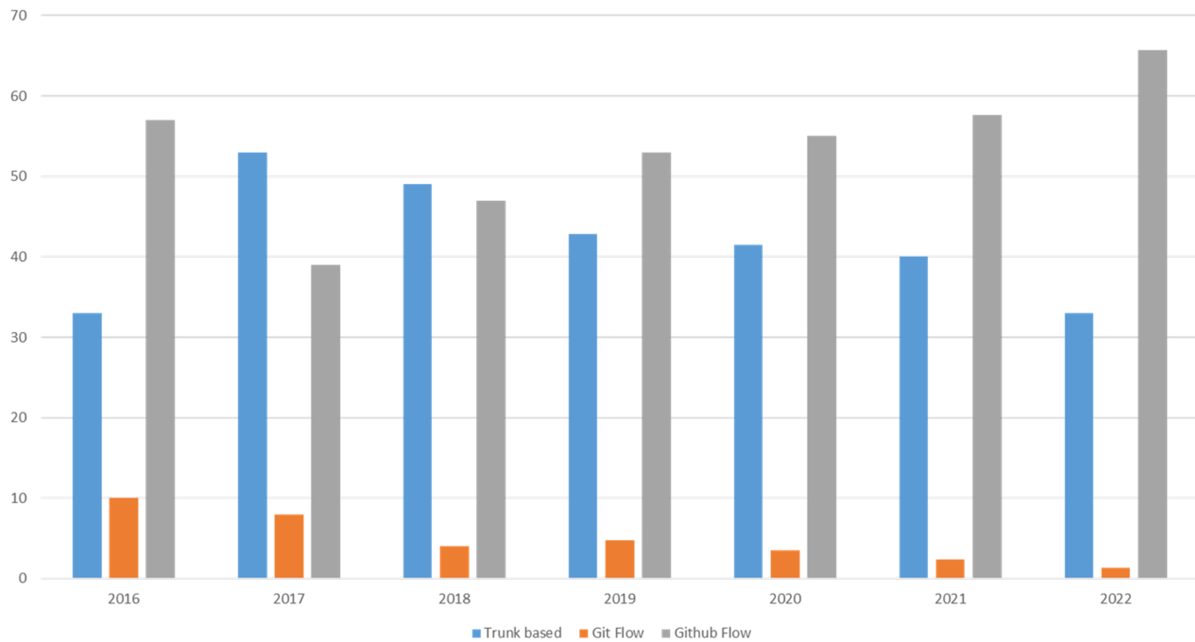

Figure 4. 4 Popularity of three main branching strategies over the years.

In recent years, the area of branching strategies in software development has experienced significant changes. An examination of the percentage trends from 2016 to 2022 reveal noticeable changes in the dominance of different branching methodologies, including trunk-based development, Git Flow, and GitHub Flow. In 2016, GitHub Flow was the prevailing approach, with a popularity of 58%, surpassing trunk-based development at 32% and Git Flow at 10%. Although the main trend persisted, there were distinct surges of favourability towards these branching strategies. In 2017, trunk-based development gained significant popularity, reaching 52.5%. In contrast, GitHub Flow decreased to 39.5%, and Git Flow remained stagnant at 8%. Over the following years, trunk-based development and GitHub Flow continued to be dominant, albeit with some minor changes. The former exhibited a prevalence ranging from 40% to 49%, while the latter showed fluctuations between 47% and 58%. Despite this, the popularity of Git Flow declined significantly, reaching a low point of 1% by the year 2022.

These statistical trends highlight a growing preference for more flexible and less complicated branching strategies. Trunk-Based Development, known for its support of continuous integration, and GitHub Flow, recognised for its efficient and simple workflow, have gradually gained popularity. Conversely, Git Flow, due to its intricate and hierarchical framework, has experienced a decline in user numbers.

4.6 Thesis II/2: Branching strategies in Mono and Multi repository projects

I conducted several analyses in the domain of branching strategies and their correlation with Mono and Multi repository structures. The findings are crucial for gaining an in-depth understanding of the relationship between branching strategies and repository structures.

Publication related to this thesis: [J3]

Once the main aspects of the three branching strategies have been introduced, it is necessary to determine their correlation with the repository structure and other project parameters. This section presents various measurements regarding the utilisation percentage of branching strategies in Mono and Multi repository projects. We plan to assess the utilisation rates of each of the three branching strategies in divergent repository structures and compare them accordingly. This analysis will provide significant insights into the cultural aspects of development practices within the respective repository frameworks, as a branching strategy reflects how developers interact with their projects.

4.6.1 Mono repository projects

The following pie chart shows the usage percentage of three main branching strategies in practice.

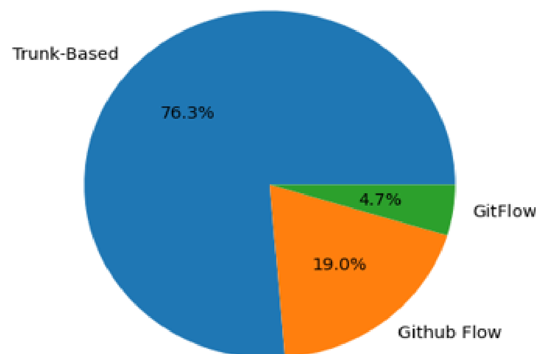

Figure 4. 5 The usage percentage of three major branching strategies in Mono repository projects.

Based on the data presented in *Figure 4.5*, it is clear that a large majority of Mono repository projects use the trunk-based branching strategy, representing 76.3% of usage. This strategy involves developers directly committing their modifications to the central trunk, or main branch, of the codebase. Although it is typically preferred for smaller projects with a smaller team, its main benefit is that it allows for quick iterations and faster feedback loops. The subsequent branching strategy of significance in the context of M-Ono repositories is GitHub Flow, which has been adopted by 19% of users. This approach involves creating separate branches for each new

feature or defect fix and then merging them into the main branch once they are completed and verified. This approach improves team collaboration and produces a clearer chronological record of changes in the codebase. Conversely, the GitFlow branching strategy is utilised in only a small portion of Mono repository projects, making up 4.7%. GitFlow is distinguished by a hierarchical branching structure that includes a main branch, a development branch, and separate feature branches for each distinct development effort. Although this methodology encourages a systematic and controlled approach to development workflow, it is frequently criticised for being complex and requiring extra time.

4.6.2 Multi repository projects

After analysing the data on Mono repository projects, it is crucial to examine similar outcomes for Multi repository projects. An investigation will uncover differences between the two repository structures in terms of branching strategies. In order to examine these outcomes in more detail, the analysis has been divided into two parts, focusing separately on the front-end and back-end components of Multi repository projects.

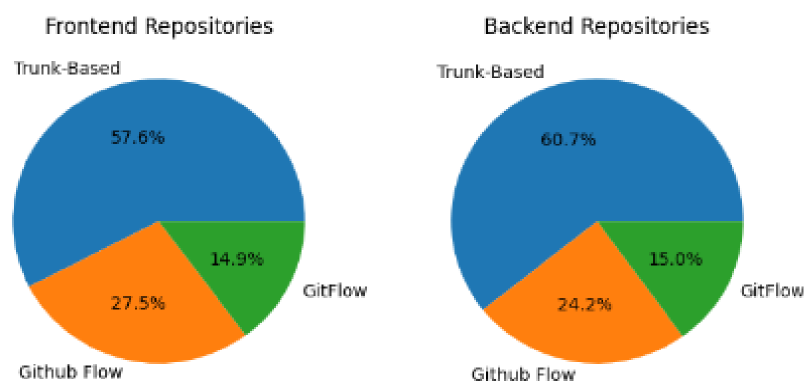

Figure 4. 6 Usage percentage of three major branching strategies in Multi repository projects.

Based on the data shown in *Figure 4.6*, it can be observed that the majority (57.6%) of Multi repository projects in the frontend segment employ the Trunk-based branching strategy. GitHub Flow, with a usage rate of 27.5%, and GitFlow, accounting for 14.9%, are two popular methodologies in this field. Similarly, an examination of the backend sector of Multi repository projects shows that a trunk-based branching strategy is the most prevalent, accounting for 60.7% of cases. The usage rates of GitHub Flow and GitFlow in the back-end domain mirror those in the front-end. GitHub Flow is the second most commonly used strategy, with a utilisation frequency of 24.2%, while GitFlow is the least utilised, with a frequency of 15.0%. The Trunk-based branching strategy is more prevalent in the backend than in the frontend. This may be attributed to the intricate nature of the backend, which necessitates an organised approach to its development. The combined utilisation rates of the three branching strategies in multirepository projects indicate a preference for larger and more complex undertakings compared to Mono repository projects, which involve multiple teams working on different project segments.

4.7 Concluding Remarks

Version control systems are essential during the development phase of software projects. These systems can be found in both centralised and distributed forms, and accurately assessing their actual effect on the operational dynamics of a project is difficult because of the numerous factors that influence them.

In order to clarify the connection between repository architecture and the development workflow, we have conducted an analysis specifically on branching strategies. Three primary branching strategies, namely Trunk-based, GitHub Flow, and GitFlow, have been chosen for an in-depth examination and will be discussed in detail in upcoming chapters as well. Each strategy possesses unique attributes and workflow patterns that not only characterise their individual functioning but also represent the overall workflow of the project. Therefore, the examination of these different strategies is crucial within the framework of this thesis.

This research demonstrates the relationship between the complexity of the repository structure and the selected branching strategy. The chosen methodological framework entails documenting the names and frequencies of branches in a project in order to infer the branching strategies utilised. Using this method, I have gathered important data that traces the changes in preference for the three main branching strategies over time. This provides insight into the changing patterns of the development environment over the past six to seven years. From 2015 to 2017, developers showed a preference for the Trunk-based branching strategy. However, in 2018, this trend started to change, suggesting that workflows became more complex as multi-repository architectures became more popular compared to Mono repositories.

The findings indicate a tendency for simpler branching strategies in Mono repository structures, which are naturally less complex than their multi-repository counterparts. Additional research provides further evidence in support of this hypothesis. My research clearly shows that the Trunk-based strategy is significantly preferred in Mono repository projects compared to multi-repository projects. Furthermore, the most complex of the strategies, GitFlow, is significantly more common in Multi repository projects compared to Mono repositories.

The author of this thesis is accountable for the subsequent contributions outlined in this chapter:

II / 1. My first contribution in this chapter involves developing a heuristic method to determine the branching strategy for any open-source project on Github. This approach utilises the names and quantities of branches to determine the strategy, as explained in greater depth in the preceding paragraphs.

II / 2. I conducted multiple analyses, which allowed me to establish a clear correlation between the branching strategy employed by projects and the structure of their repositories. It demonstrates that the choice of a repository structure can have a significant impact on the overall approach to development.

Chapter 5

Thesis Group III: Productivity of software development

By considering various factors like branching strategy, development time, and developer effort, I developed a novel methodology for evaluating the effectiveness of projects. There are three pre-established levels of productivity, namely high, low, and none, which I also introduced. In addition to this novel approach, I developed a new machine learning model to forecast the duration of the project development phase. This machine learning model utilises various project parameters, such as the number of developers, the intensity of the development process, the productivity level, branching strategies, and other relevant factors. Extensive testing has shown that the recently developed algorithm accurately predicts the length of the development process in terms of months, with a minimal average margin of error of just a few months.

Publication related to this thesis: [J4]

Here, we focus on the efficiency and effectiveness of software development. This section critically examines an established approach and applies it to the methodology of this thesis. The main goal of the productivity calculation is to assess it from various angles, including branching strategy, development duration, and team size. Firstly, we perform the calculation of productivity and its division into three primary categories. These classifications will be used to demonstrate the effect of different parameters on productivity with the highest level of objectivity.

Productivity was measured in three main branching strategies separately, allowing for the identification of a connection between productivity and the branching strategy in software development. In order to get a better understanding, additional variables were also analysed, which allows us to identify correlations between these parameters and productivity in the software development process.

5.1 Introduction

Over the course of many decades, there has been a significant amount of research and discussion surrounding the productivity of software development. This has involved exploring different definitions and metrics to accurately measure productivity. Productivity serves as a measure of the amount of work that is successfully accomplished within a specific period of time.

This research project goes beyond simply studying productivity as a concept. It utilises a branching strategy, repository structure, and other relevant parameters to assess and measure productivity from various perspectives.

Furthermore, this study is supported by a unique and extensive database containing more than 50,000 repositories, which guarantees a more meticulous and unbiased examination of the subject. Utilising this database enables us to provide many more impartial results compared to prior investigations. Firstly, we analysed the structures of the repositories and their relationship with productivity. By assessing the proportion of projects in different productivity categories across both repository structures, we can estimate the dominant patterns for each type of repository structure. These percentages on their own are not sufficient to draw any definite conclusion. Therefore, we will use different correlation methods to examine the connection between productivity and repository structure. In the next chapter, we will analyse different methods used to calculate productivity and provide an overview of the three main branching strategies that have received significant attention and have been applied in various situations. We seek to offer valuable insights into the influence of branching strategies on productivity in the software development field, examining both the frequency and effectiveness of these strategies. By conducting these analyses, we will determine which branching strategy is more commonly observed in projects with high and low productivity. This will aid our understanding of the workflow of projects relative to their levels of productivity, and in doing so improve our overall grasp of productivity. Also, other aspects of the development process, such as the size of the team, the duration of development, and other factors, will be closely examined. This approach will provide a clearer perspective on the characteristics of highly productive projects.

5.2 Related Works

Project productivity has been a significant focus of research for many years, and it remains a topic of intense discussion in academic and business communities. Every academic pursuit concerning this subject seeks to define 'project productivity' with a unique explanation and evaluate it using various project criteria or attributes. For example, in document [48], productivity is defined as the ratio of project effort to product size:

$$Productivity = Size / Effort \qquad (1)$$

However, it is obvious that not all types of projects will benefit from using this approach, especially when there is no correlation between the project size and the amount of development work required. These situations are quite common in projects that are based on the internet. Moreover, this method of enhancing productivity may unintentionally encourage developers to create excessive amounts of projects that have little to no value. By implementing the methodology described in document [49], this approach is improved, and a new formula for determining productivity has been created:

$$Productivity = AdjustedSize / Effort \quad (2)$$

Only the size metrics that display a strong correlation with effort are included in the calculation as adjusted size. This methodology, which is rather like to its previous version, calculates productivity based on the amount of work put in. However, it also contains specific exclusions, such as maintenance tasks and other jobs that fall outside the scope of this notion. Introduced in 2004, there have been significant advancements in software and Web development ever since. Therefore, it is no longer practical to treat this method as completely precise.

There are alternative methodologies available, as described in document [50], where the authors assess software development productivity by considering two main factors, those of quantity and quality.

Quantity: The authors developed a model that determines the average level of effort put in by developers and calculates the time gap between their commits. Essentially, this entails analysing the frequency and extent of code modifications made by developers in their commits.

Assessment: The LGTM [51] system is used to evaluate the quality of the code. Although this method is considered more reliable than others because it utilises machine learning, its main drawback is its dependence on commitments as the sole measurement metric, which is recognised. It is worth mentioning that most of the studies examined so far have chosen to evaluate productivity using the measure of commits. For instance, articles [52], [53], and [54] employ the total count of commits as a measure of the project's productivity.

The author of [39] present another study to illustrate the effect of branching strategies on productivity. A total of nearly 3000 projects were subjected to analysis, but none of the branching strategies have been considered. Instead, they analysed the overall characteristics of the branching strategy.

5.3 Calculation of Productivity

As mentioned earlier in the previous sections, calculating productivity is a crucial task. There are multiple methodologies available for this purpose, as the effectiveness of software development has been clarified by applying different approaches. Now, we present a verified method for calculating productivity. Firstly, this approach will begin by providing a clear explanation of its fundamental concept and algorithm. Following this, comprehensive mathematical requirements will be presented to facilitate a better understanding of it all.

5.3.1 Algorithm for the calculation

In the previous chapter, we conducted an examination of various methods for quantifying productivity. Most of these methodologies primarily base their productivity estimates on a narrow range of project parameters. In our study, we employed the methodology outlined in reference [55]. There were numerous compelling reasons for doing this. The main ones are:

- When juxtaposed with alternative methodologies, the selected approach provides a discernible superiority in accuracy and dependability, owing to its incorporation of an extensive set of specific projects.
- The researchers in [55] mainly employed open-source projects from the GitHub platform, and this aligns well with the attributes of our database.
- The authors of [55] carried out a comprehensive examination of diverse approaches and substantiated the effectiveness of their methodology via comparative assessments.

The term "burstiness" in the context of statistical analysis refers to the periodic variations in activity or frequency that an event exhibits. After a thorough analysis, the data regarding project activity was identified as exhibiting bursty behaviour. This requires the identification of periods with increased, decreased, or no activity. In order to identify active bursts, the authors in [55] employ three separate models called Maximum Sum Segments [56], Kleinberg Burst Detection [57], and the Hidden Markov Model [58].

In order to make a comprehensive comparison of all three approaches, the authors included supplementary evaluation metrics prior to validating their method. The goal was to compare the bursts found by the models mentioned above to a lexically coherent segmentation of the project timeline using Beeferman et al.'s [59] P_k as a standard measure. As explained in [55], a different approach identifies consecutive days with similar conversation topics as 'lexically coherent bursts', by analysing the interactions between project developers. These bursts, which reflect the identified lexical segments, demonstrate coordinated efforts towards similar goals, illustrating increased levels of activity and thus forming a cohesive body of work. We utilise the widely acknowledged TextTiling text segmentation technique as the foundation for our analysis of lexical cohesion [60].

TextTiling is a method used to divide texts into distinct segments that conform with the underlying structure of different subtopics. This model assumes that changes in the arrangement of words in the text suggest a shift in the underlying subject. The method entails employing a movable frame across a textual representation, such as a vector space. The similarity score is computed at each window position by calculating the cosine similarity between the upper and lower halves of the window. In order to realise their goal, they collected a wide range of textual information, including issue comment text, issue titles, pull request comments, commit messages, and commit comments. Afterwards, these texts were combined for each day, and they went through a series of procedures, which involved removing unnecessary parts, reducing words to their basic forms using stemming, and converting them into word vectors. As a result, there was a separate word vector for each day of the project timeline.

The average segment lengths generated by the above three methods differ. After comparing the values using these methods and considering the length of the segments, it was discovered that the results were similar. The Hidden Markov Model (HMM)'s predicted work units have higher lexical coherence than the Max-Sum method's forecasts. In contrast to the parameter-free Max-Sum model, the Hidden Markov Model (HMM) offers improved interpretability. Examining the distribution of average parameter values within the states of a trained Hidden Markov Model (HMM) can provide valuable insights. This provides a more comprehensive depiction of the characteristics of the data stream. Based on these factors, it was found that the Hidden Markov Model (HMM) was the optimal method for segmenting their dataset. Our analysis of the database corroborates the fact that the third alternative—the Hidden Markov Model—aligns optimally with

our requirements, mirroring the selection and application in the aforementioned study. Now, we shall use the subsequent burst stream of projects as an illustrative example:

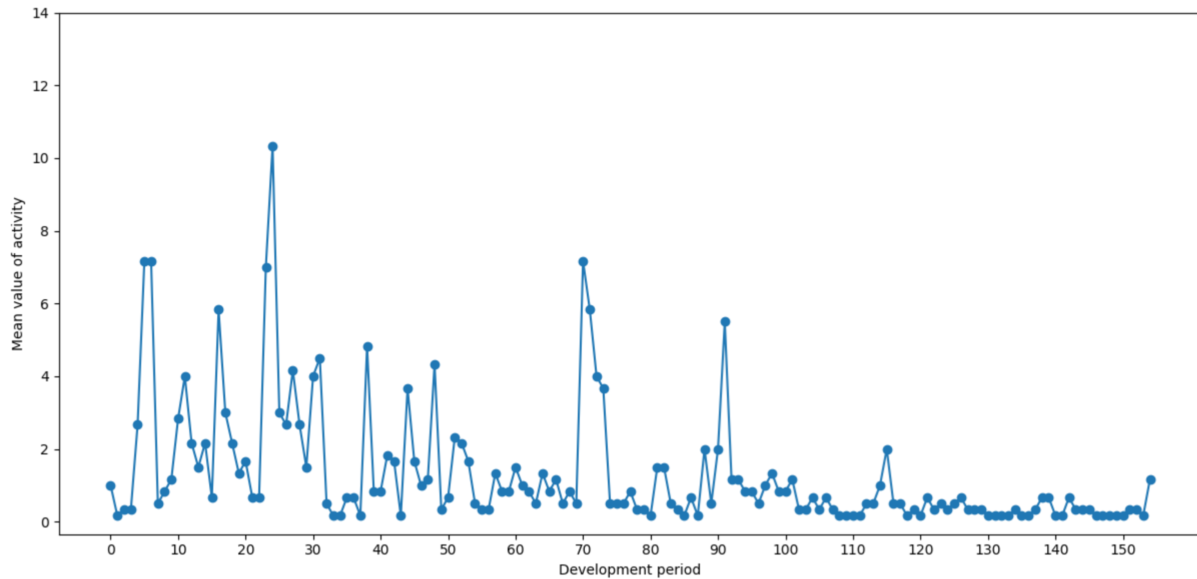

Figure 5. 1 A burst sequence of random Mono repository project taken from our database.

States with activity values below a specific threshold were categorised as 'low active,' while those exceeding the threshold were labelled as 'active.' The occurrences when each project was identified as being in the 'active' state were documented. Consequently, a series of these 'active days' were combined into a burst, with a maximum gap of three days between each active day. This methodology for generating bursts divides the project timeline into specific periods that are defined by bursts of activity. This allows for more meaningful comparisons with other burst segmentations. In our study, we employed burst duration as a metric to evaluate "productivity," which is quantified by the number of days it spans. The collaborative dynamics of the open-source ecosystem have an impact on this metric and serve as a success indicator. We suggest that individuals work efficiently during a productive burst, which results in the prompt completion of tasks. This is at the heart of our core hypothesis. References [61] and [62] provide evidence that this concept is consistent with earlier congruence studies.

Projects in this chapter are categorised into three distinct states to facilitate clarity. Bursts that surpass the average duration are classified as "high productive." In contrast, bursts that are shorter than the average are considered "low productive". And the project is classified as "non-productive" in instances where burst length is noticeably absent. This approach provides a more thorough evaluation of the influence of a variety of parameters on productivity. This classification into three categories facilitates a comprehensive analysis of the variations in these parameters, thereby facilitating the development of more precise hypotheses regarding their respective effects. 'High productivity' is the most exclusive category in this context, encompassing only the most productive projects. Projects that marginally exceed the threshold but fail to achieve the necessary level of productivity to be classified as high productivity projects are referred to as "low productivity." In contrast, "non-productive" projects, which display activity levels that are below the threshold, make up the lowest category in our analysis. This categorization serves to elucidate

the interplay between branching strategies and other project parameters, thereby enhancing our comprehension of their influence on productivity.

5.3.2 Mathematical specification of an algorithm

A critical aspect of project management and evaluation is the accurate computation and quantification of project productivity. It is essential to establish a formal mathematical specification that delineates the computation process in order to achieve a comprehensive and impartial evaluation. Here, we provide a comprehensive framework for formalising the mathematical calculation of project productivity. By offering a systematic and unequivocal methodology, this specification guarantees the consistency, reproducibility, and comparability of productivity metrics in a variety of contexts and projects.

The project ensemble $P = \{ p_1, p_2, \dots, p_n\}$ encompasses six principal activity parameters for each project, explicitly: commits (p_i), issues (p_i), issue_comments (p_i), pull_comments (p_i), pull_requests (p_i), and events (p_i).

To calculate bursts within each project, we shall define the following mathematical entities and operations:

- Let $D(p_i)$ be the set of days in the development period of project p_i.
- Let $A(p_i, d)$ represent the total activity count on day d for project p_i.
- Let $T(p_i)$ be the predefined threshold value for distinguishing "low active" and "active" states in project activities.
- Let BurstSegments (p_i) be the list of burst segments for project p_i.
- Let BurstStart (b_s) and BurstEnd (b_s) represent the start and end days of burst segment bs, respectively.
- Let Length (b_s) denote the length of burst segment b_s, calculated as the difference between BurstEnd (b_s) and BurstStart (b_s) plus one.

The calculation of bursts and determination of productivity states follow these steps:

Step 1: Initialization

- Initialize BurstSegments (p_i) as an empty list for each project p_i in P.
- Initialize BurstStart (b_s) and BurstEnd (b_s) as null variables for each burst segment b_s.

Step 2: Identifying Active Days

- For each day d in $D(p_i)$ for project p_i in P :
  - If $A(p_i, d) >= T(p_i),$ mark day d as an "active day" for project p_i.

Step 3: Creating Bursts

- For each active day d in project p_i:
  - If BurstStart (b_s) is null:
    - Set BurstStart (b_s) = d.
    - Set BurstEnd (b_s) = d.
  - Else:
    - If d is within 3 days of BurstEnd (b_s):
      - Set BurstEnd (b_s) = d.
    - Else:

- Create a new burst segment with BurstStart (b_s) and BurstEnd (b_s) as the start and end days, respectively.
- Add bs to BurstSegments (p_i).
- Set BurstStart (b_s) = d.
- Set BurstEnd (b_s) = d.

Step 4: Calculating Average Burst Length

- Compute the average burst length AvgBurstLength (p_i) for project pi as: AvgBurstLength (p_i) = (sum of Length (b_s) for all bs in BurstSegments (p_i)) / (number of bursts in BurstSegments (p_i)).

Step 5: Determining Productivity States

- For each burst segment bs in BurstSegments (p_i):
  - If Length (b_s) > AvgBurstLength (p_i), label bs as "high productive."
  - If Length (b_s) < AvgBurstLength (p_i), label bs as "low productive."
  - If Length (b_s) is nearly zero, label bs as "non-productive."

5.4 Productivity in repository Structure

The relationship between repository structure and software development productivity has been the subject of limited investigation, despite the fact that productivity has been examined from a variety of perspectives in the past. Below, we will analyse the percentage distribution of projects across various productivity categories with the aim of identifying the most prevalent productivity category for Mono and Multi repository projects. Although this analysis does not offer any definitive conclusions and it requires the consideration of additional factors, it nevertheless provides an initial understanding of the relationship between productivity and repository structure.

5.4.1 Productivity in Mono repository

At the beginning, we assessed the outcomes of projects that were classified into one of the three productivity tiers. The distribution of projects across the three productivity categories and the percentage of projects that utilise the Mono repository structure are depicted in Figure 5.2. The productivity dispersion of the Mono repository projects was as follows: only 11.93% were classified as highly productive, 18.12% were deemed lowly productive, and the majority, 69.95%, were classified as non-productive. The open-source nature of the projects in our database is the primary reason for the high percentage of non-productive projects. A significant number of these belong to the non-productive category due to the fact that they were developed by individuals without commercial motivations. We anticipate that projects employing the Multi repository structure will exhibit a comparable pattern. These results suggest that the Mono repository approach is not frequently chosen by the majority of highly productive projects. Later, we shall examine the fundamental causes of this trend.

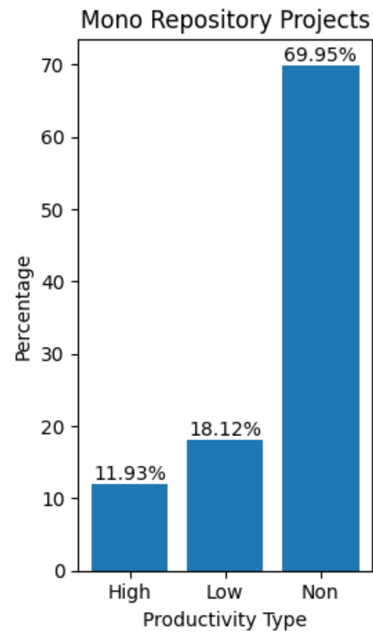

Figure 5. 2 The percentage share of Mono repository projects based on their productivity level.

5.4.2 Productivity in Multi repository

Comparable analyses were also conducted for Multi repository projects. *Figure 4.3* shows the results of an identical analysis for these projects, where distinct differences are evident at first glance.

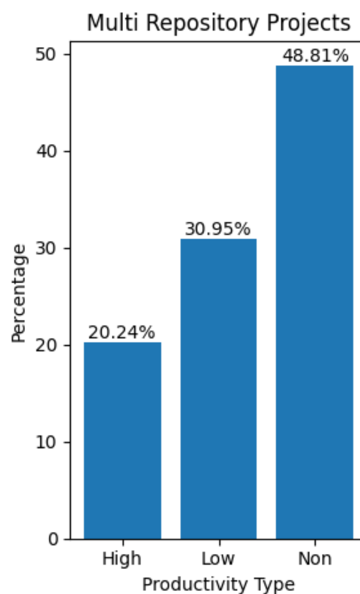

Figure 5. 3 The percentage share of Multi repository projects according to their productivity level.

The results of our investigation indicate that 20.24% of the Multi repository projects were classified as high productive, 30.91% as low productive, and the remaining 48.81% as non-

productive. It is evident from these figures that the proportion of highly productive projects in multi repository projects is significantly higher than that in mono repository projects. Although there are numerous potential explanations for this, it generally implies that Multi repository projects are more productive than their Mono repository counterparts.

The operational mechanisms of these repository structures and our methodology for calculating productivity may be the primary factors that influence this trend. This implies that Multi repository projects exhibit substantially more activity than their counterparts, as our productivity calculation is based on project activity and activity bursts. In the subsequent sections of this chapter, we will investigate the supplementary parameters of each repository structure to gain a more comprehensive view of the observed patterns.

5.5 Productivity and Branching Strategy

The results from our examination of various repository structures suggest that highly productive projects prefer the multi-repository structure over the mono-repository approach. A more in-depth examination of the development process is necessary because a variety of factors may affect this trend. Branching strategies are an effective method for scrutinising the development workflow, as previously mentioned. We divided our analysis into two distinct groups in order to improve clarity, evaluating Mono and Multi repository projects separately. This division permitted an evaluation of the most preferred branching strategies within each repository structure. It will offer a clearer understanding of the development workflow across various productivity levels and provide insights into the methods that developers prefer.

5.5.1 Three main branching strategies

Mono repository case:

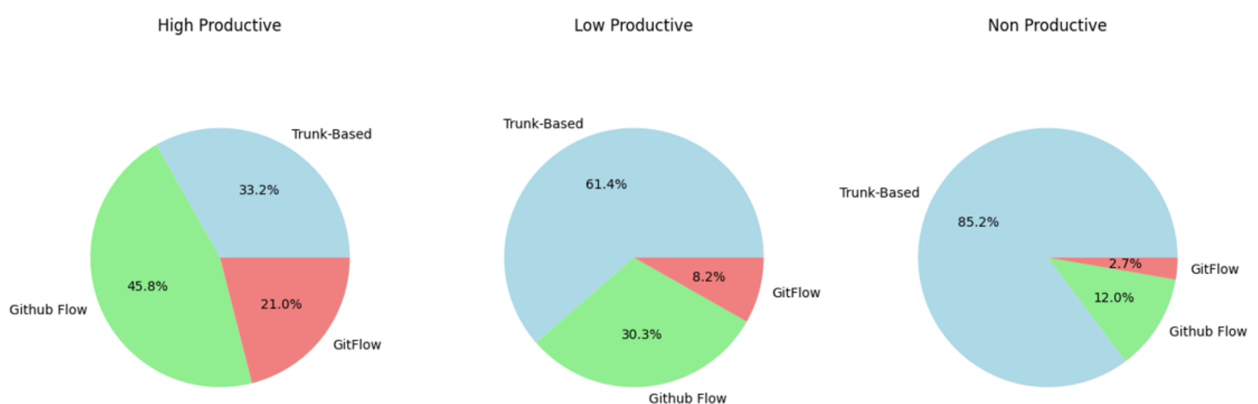

Figure 5. 4 The usage of branching strategies based on the productivity level (Mono repository).

First of all, let us check each productivity level separately for a better understanding:

High Productive:

The following is a breakdown of branching strategies in high-productive Mono repository projects (see *Figure 5.4*). A significant 32.3% of these projects used the Trunk-based approach, indicating a preference for a continuous and effective development process. In the interim, 45.8% of the projects implemented the Github Flow strategy, which permitted a collaborative and iterative development process. 21% of the exceptionally productive projects used the GitFlow approach, which stood out for its feature-centric and organised workflow.

Low productive:

A distinctive pattern is evident in the distribution of branching strategies within the low productive category of Mono repository projects, as shown in *Figure 5.4*. A sizable 61.4% of these projects used the Trunk-based strategy, indicating a preference for a quicker and more streamlined development process. The Github Flow approach was implemented by 30.3% of the projects, which facilitated rapid deployment and ease of collaboration. In contrast, the GitFlow strategy, which is renowned for its emphasis on a more structured and regulated approach to development, was implemented by only 8.2% of the low-productive projects.

Non-Productive:

Furthermore, *Figure 5.4* explains why the implementation of branching strategies in Mono repository projects are not productive. The Trunk-based approach was the preferred method for a significant portion of these unproductive projects, accounting for 85.2%. This suggests a preference for a straightforward and agile development methodology. The Github Flow strategy's utilisation decreased to 12.0%, underscoring the difficulties associated with achieving effective deployment and collaboration within unproductive projects. Only 2.7% of the non-productive projects used the GitFlow strategy, indicating a low adoption rate for feature-focused workflows in this group.

Multi repository case:

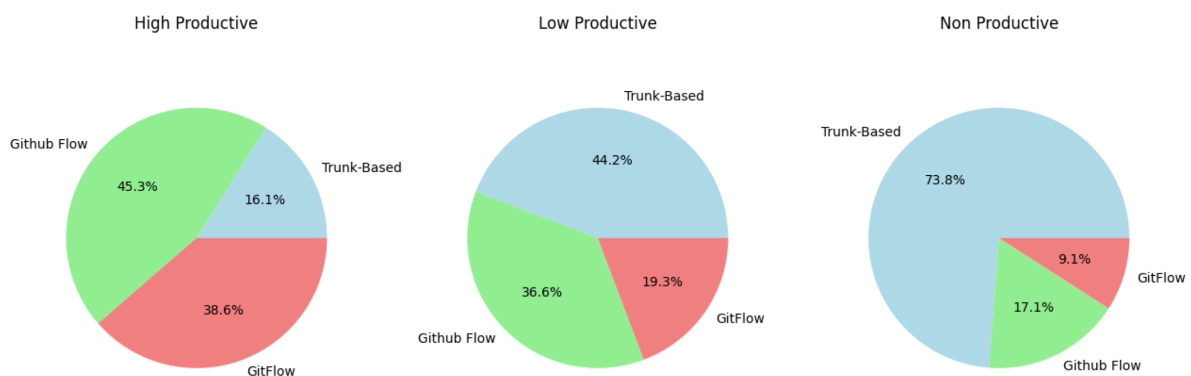

Figure 5. 5 The usage of branching strategies based on the productivity level (Multi repository).

As it was done in Mono repository case here, we will again explain the different cases one at a time:

High Productive:

In the section of high productive frontend Multi repository projects, 16.1% embraced the trunk-based approach, which helped a more efficient development process. The Github Flow strategy,

known for fostering collaborative and iterative development, was employed by 45.3% of these projects. A significant 38.6% of the projects utilised the GitFlow approach, which is characterised by its emphasis on a feature-driven and organized workflow.

Low Productive:

The Trunk-based approach was preferred by 44.2% of frontend Multi repository projects in the low productive category, highlighting its simplicity and effectiveness. The Github Flow approach was implemented in 36.6% of these projects, which facilitated rapid deployment and ease of collaboration. Furthermore, 19.3% of the projects implemented the GitFlow strategy, suggesting a preference for a more structured and regulated development process.

Non-Productive:

The Trunk-based strategy was noticeably dominant among non-productive frontend Multi repository projects, with 73.8% of them employing it. This underscores its relevance for simple and agile development methodologies. The adoption of the Github Flow strategy was significantly reduced in these projects, with a rate of just 17.1%. This indicates that there are obstacles to effective deployment and cooperation in unproductive contexts. The GitFlow strategy had the lowest adoption rates in non-productive projects, with only 9.1% usage. This suggests that feature-driven workflows are less common in this category.

After carrying out a comprehensive examination of branching strategies, it is quite obvious that highly productive project developers favour more intricate branching strategies, such as Github Flow or GitFlow, while simple strategies, such as Trunk-based, are not as prevalent. However, it is clear that a Trunk-based strategy is implemented in nearly 33% of the highly productive projects within the Mono repository. This implies that there may be a correlation between the repository structure and the developer team's workflow.

5.6 Properties of branching strategies

Next, we shall investigate the potential correlation between the productivity rate associated with a branching strategy and a variety of branching strategy characteristics. The primary emphasis is on the branch and commit counts of a variety of branching strategies in Mono and Multi repository setups. We will perform numerous correlation analyses, which will be represented by a set of graphics plots and charts.

In a manner similar to previous instances, the results for Mono and Multi repository projects, as well as for various branching strategies, are presented in distinct graphical representations.

5.6.1 Commit Count

Commit counts may be interpreted as an indicator of the workload undertaken during the development process. Each commit encompasses a compilation of altered files and the specific lines that have been modified, added, or removed. In numerous instances, commits are utilised to dissect the entirety of the development process, as demonstrated in studies such as [63 - 66], which analysed commits to comprehend the workflow of development. In addition, other studies have

employed commit counts as a metric for gauging a project's productivity. These instances highlight the significant role of commits in research, providing valuable insights into a project and its developmental trajectory. Consequently, in our study, we laid particular emphasis on the analysis of project commits. This section will detail our examination of commit counts in various contexts.

The data presented in *Figure 5.6* provides an insight into the level of commitment associated with the two types of productivity in Mono repository projects. This visual representation helps us to identify which commit counts are more common in projects with either high or low yields.

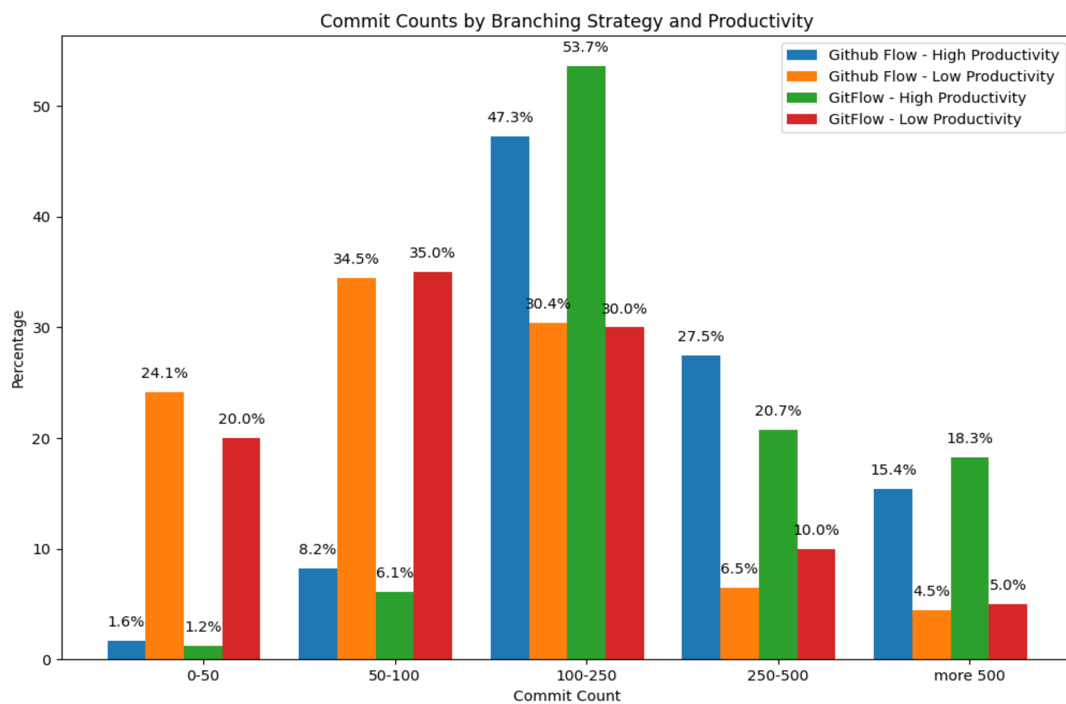

Figure 5. 6 The percentage share of commit counts for two main branching strategies in Mono repository projects.

Figure 5.6 reveals a stark contrast in commit counts between projects with high and low productivity. It is intriguing that the results are relatively consistent across both Github Flow and GitFlow branches, with the primary variation being between high and low productivity levels. It is worth noting that 58.6% of low productivity projects that utilise Github Flow and 55% of those that use GitFlow have commit counts below 100. In contrast, projects with high productivity tend to have lower commit counts. The third column of the chart indicates a clear divergence: nearly half of the high productivity projects in both branching strategies have commit counts between 100 and 250, which could be regarded as an optimal range. The percentage of high productivity projects decreases in the subsequent columns, but it remains three times more prevalent than low productivity projects. These results tell us that the majority of high productivity projects have a commit count in the 100–250 range, with higher commit numbers being more prominent in high productivity projects than in low productivity projects.

Figure 5.7 shows the same results, but for Multi repository projects. Some clear similarities can be seen from a first glance at the chart.

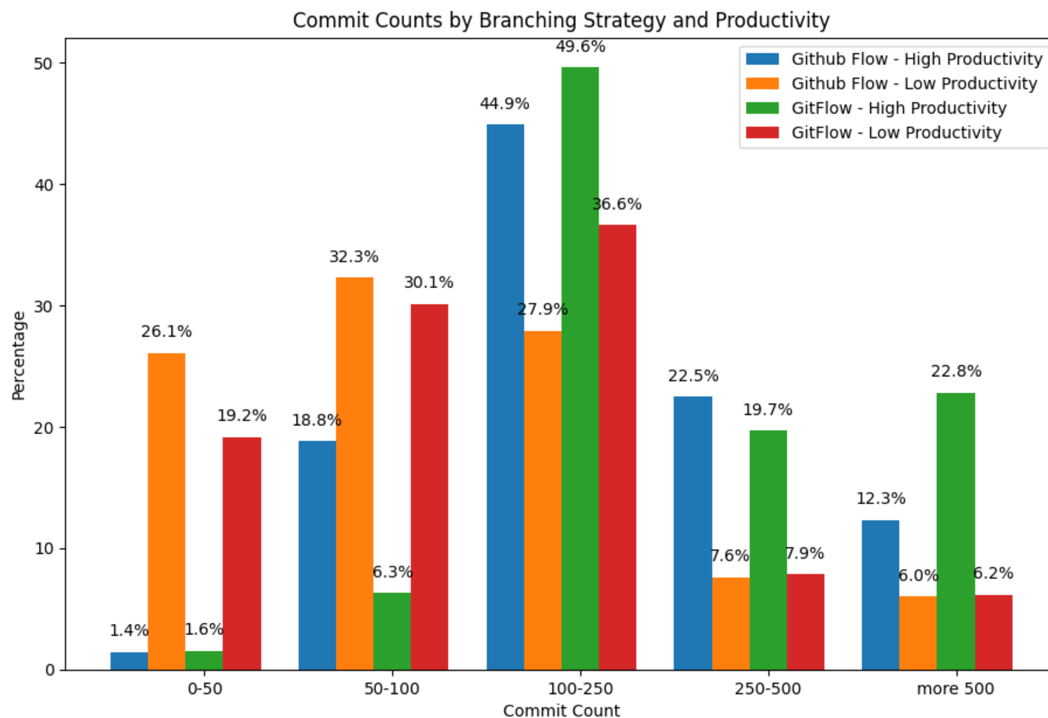

Figure 5. 7 The percentage share of commit counts for two main branching strategies in Multi repository projects.

The overall database displays a negligible presence of those with a branch count between 0 and 50 and high productivity rates in the context of multi repository projects. The first two categories, which correspond to projects with 0–50 and 50–100 commit counts, however, demonstrate a preponderance of low productivity projects in both branching strategies. These percentages are broadly comparable to those observed in the Mono repository scenario. Approximately 50% of the middle section of the chart is occupied by high productivity projects from both branching strategies when the commit count falls between 100 and 250. The remaining portion of the histogram plot reflects the pattern observed in the Mono repository case.

Upon examination of *figures 5.6* and *5.7*, it is evident that the correlation between commit count and branching strategy is consistent in both Mono and Multi repository contexts.

5.6.2 Branch Count

The preceding chapter concentrated on the critical role of branches and branching strategies within this framework. In certain cases, branches have attracted more attention than commits in research studies. Since the early adoption of distributed version control systems, branches have been the subject of numerous academic studies. In order to explain the complexities of the development workflow and other critical aspects of software development, research papers, such as [67–71], investigate a variety of branches, including their descriptions, life cycles, and other parameters, in a manner similar to the studies on commits. Branches are of great importance in this research, as they are essential indicators of the development process. The branch counts of both Mono and Multi repository projects will be analysed in a manner similar to the analysis conducted for commit counts.

First, the results for Mono repository projects will be presented, as in the previous cases.

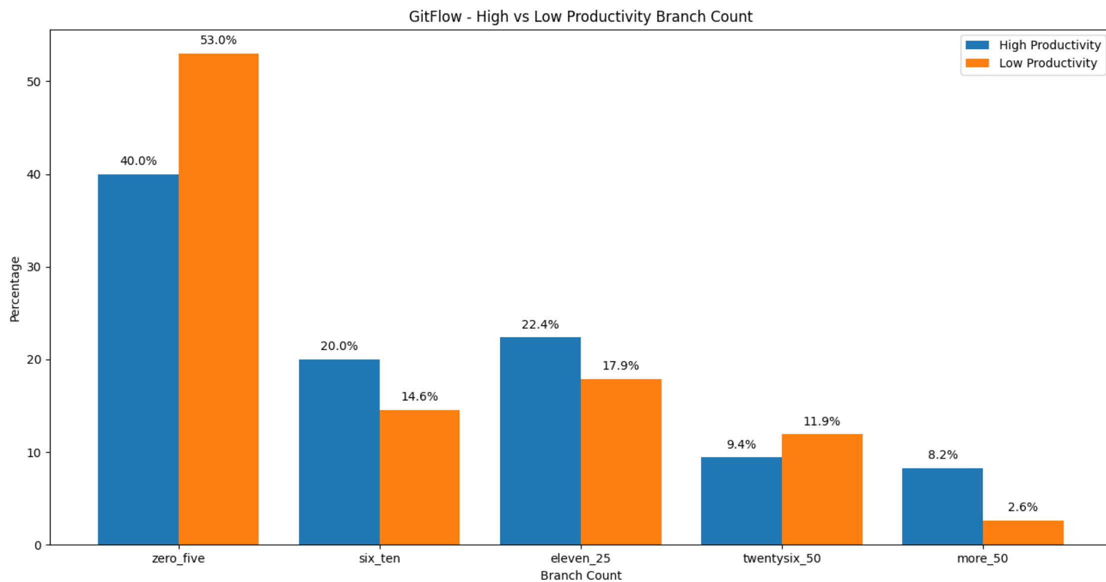

Figure 5. 8 GitFlow - High vs Low Productivity Branch count in Mono repository projects.

As in the previous cases, the results of non-productive projects have been excluded from this segment, as they do not have any significant value. The introduction section delineated the fundamental attributes of each branching strategy. In the context of GitFlow, the branch count of nearly half of the low-productive projects falls within the range of 0-5, while this figure is precisely 40% for high-productive projects. In the subsequent portions of the chart, which represent "6–10 branches" and "11–25 branches," the incidence of high-productive projects is over 6% higher than that of low-productive ones. It is worth noting that the final section of the chart is the most intriguing, as it indicates that 8% of highly productive projects have over 50 branches. This statistic suggests that a significant number of branches are present in nearly one out of every ten high productivity projects. The types of these branches and their implications will be further explored in the following sections.

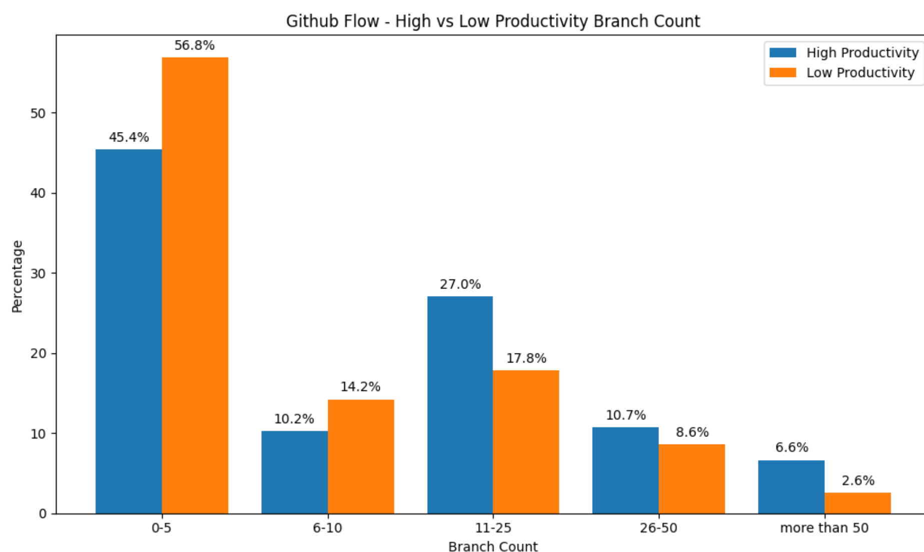

Figure 5. 9 Github Flow - High vs Low Productivity Branch count in Mono repository projects.

Figure 5.9 indicates that the number of branches in over half of the low productive projects and 45% of the high productive projects is less than five, which represents a 5% deviation from the scenario previously discussed. This discrepancy is to be expected, given the nature of the majority of these projects. Another noteworthy observation from the third portion of the chart is that projects with a branch count between 11 and 25 account for more than 25% of all high-productive projects that utilise the Github Flow branching strategy. The figures in the remaining portions are comparable to those depicted in *Figure 5.8.*

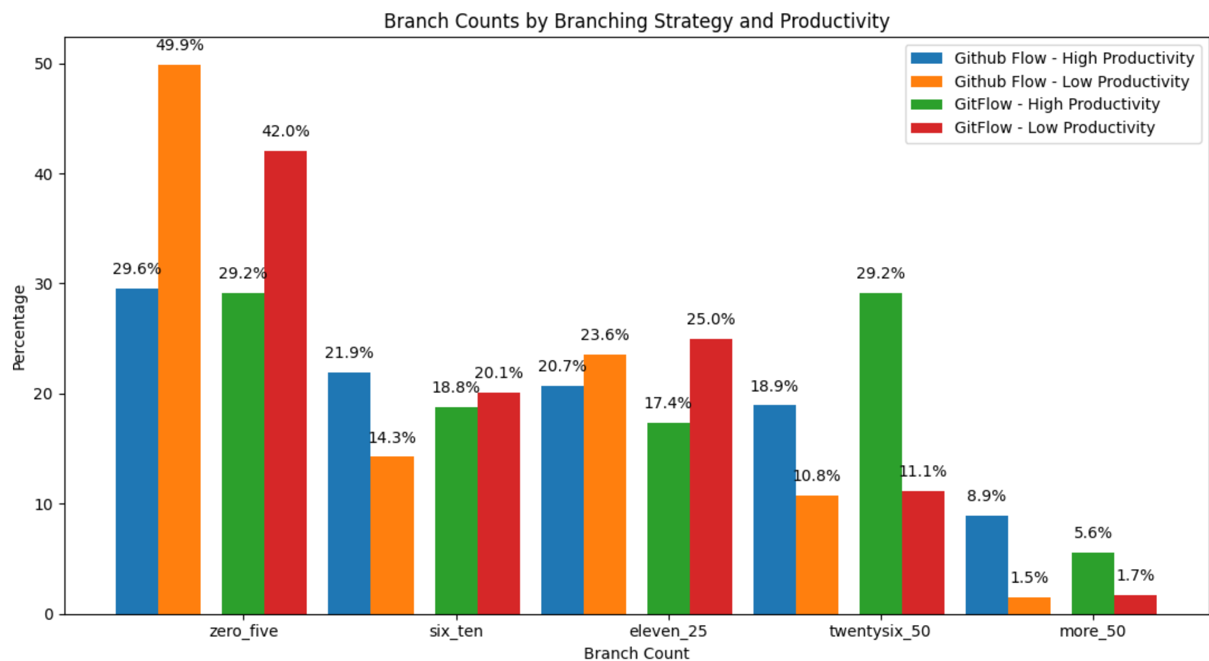

Figure 5. 10 Percentage share of branch counts in the Multi repository projects.

Figure 5.10 demonstrates a noticeable difference in the number of branches in multi-repository projects. We will commence our investigation by looking at the GitFlow statistics for Multi repository projects, similar to the approach taken for the analysis of mono-repository projects. Upon comparing the data from *figures 5.8* and *5.10*, it is clear that having a low number of branches is not preferred in the frontend segment of Multi repository projects that use GitFlow. GitFlow has the highest proportion of low productive projects, with approximately 42% of them using fewer than 5 branches. Portions "6–10" and "11–25" also demonstrate a higher frequency of these branch counts in projects with low productivity compared to those with high productivity. An interesting finding can be seen in the "26–50" category, where approximately 30% of highly productive projects prefer a branch count within this range, indicating its popularity among GitFlow projects with high productivity. The final portion of the chart closely corresponds to the findings displayed in *Figure 5.8.*

Github Flow: The results for low productive projects in *Figure 5.10* reveal a difference of only 6% compared to *Figure 5.9*. However, there is a 15% difference in the proportion of high productive projects. This suggests that a lower number of branches is less frequent in the frontend of high productive projects. The portions labelled "6–10" and "25–50" exhibit comparable preferences, indicating that these ranges are equally favoured in the initial stages of projects. The last segment of the chart displays a distinct result that was not observed in *figures 5.8 or 5.9* or the

GitFlow section of this chart. Almost 9% of the high productive projects use over 50 branches, a notably significant proportion compared to other results.

5.6.3 Developer team size and development period

When analysing commit counts to measure workload and productivity levels across different branching strategies, it is important to consider two key factors: the duration of the development period and the size of the team. These elements are crucial during the planning phase of software development, as they have the potential to significantly reduce the time and effort required and improve productivity. It is vital to mention that when discussing "developer team size," it just refers to the number of developers actively engaged in the coding stage of the project. In addition, although the duration of project development is initially measured in days, it will also be expressed in months in later analyses and sections to provide a clearer analysis.

Following previous methodologies, I have categorised the outcomes for both Mono and Multi repository structures with the intention of providing a more distinct and targeted analysis.
The Mono repository case:

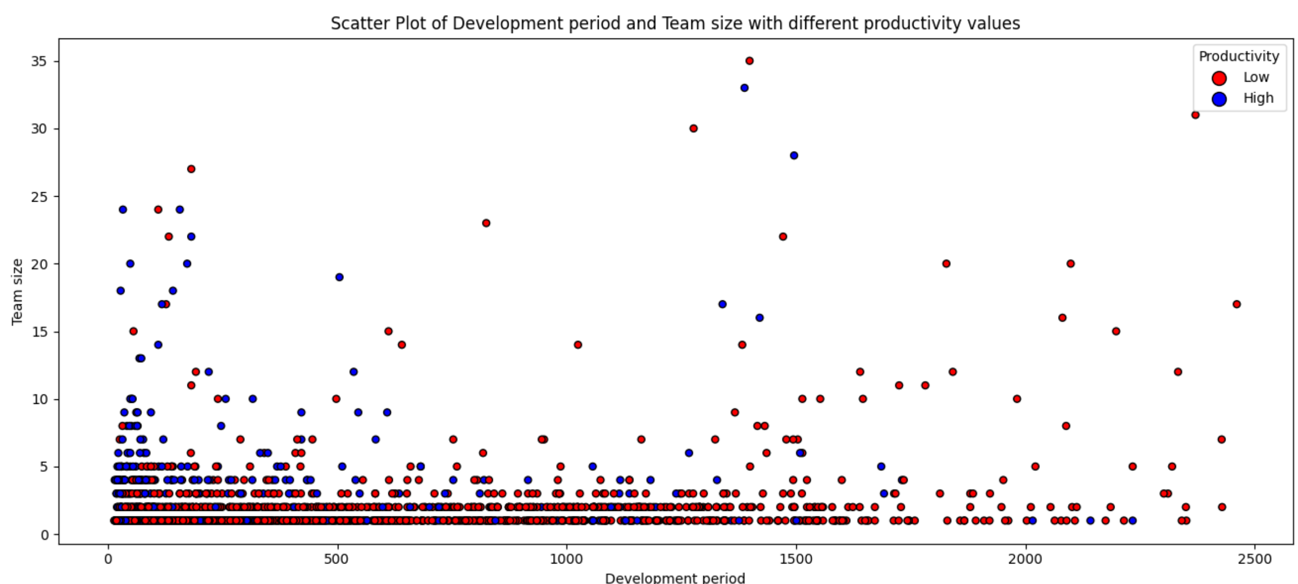

Figure 5. 11 A scatter Plot of Development period and Team size with different productivity values in Mono repository projects.

According to *Figure 5.11*, the highest productive projects are completed in less than 1000 days, which is roughly equivalent to 3 years. The typical team size for these projects ranges from 3 to 20 members. It should be added that numerous factors, including some that may not be measurable in this context, influence both the size of a team and the time required for development. Nevertheless, this empirical data remains valuable as it offers significant and previously unexplored insights. For this and future analyses, the results are not categorised based on branching strategies, as the main emphasis is on team size and the duration of development.
The Multi repository case:

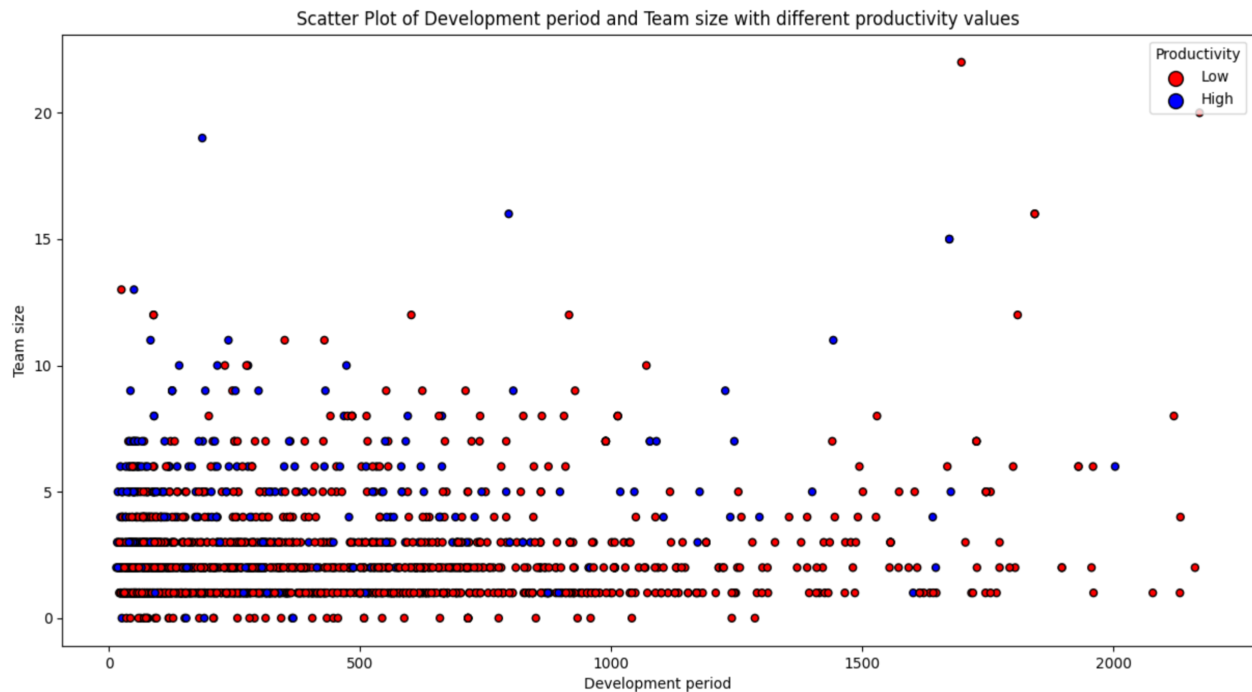

Figure 5. 12 A scatter Plot of Development period and Team size with different productivity values of Multi repository projects.

Figure 5.12 indicates that Multi repository projects exhibit comparable patterns. In the same way, projects that are high productive typically have shorter development periods than those that are low productive. The team size metrics show a significant difference between *Figures 5.11 and 5.12*. The corresponding findings are also apparent in *Figure 3.3*, where the analyses suggest that Multi repository projects have larger development teams than Mono repository projects. This observation implies that the productivity of software development processes is likely to increase in Multi repository projects with appropriately scaled team sizes. It should be mentioned that just increasing the size of the team does not guarantee increased productivity rates; however, it remains a substantial factor for projects that implement the Multi repository approach.

5.7 Thesis III/1: Calculation of Productivity with new ML method

I developed a new approach for the identification of productivity of projects using several parameters like branching strategy, development period and the work intensity of developers. There are three predefined productivity levels which were also introduced by myself these being: High, Low and None.

Publication related to this thesis: [J4]

Based on the findings stated in the preceding chapter, it is evident that productivity within the software development process exhibits a notable correlation with various facets of the development process and the project's inherent characteristics. A pivotal aspect of this thesis is the development of a Machine Learning model, designed by myself, which seeks to, that aims to prognosticate the productivity levels of the development process that utilise these identified

parameters. To optimize the efficiency of this model, multiple experimental trials were conducted to find the most effect amalgamation of features.

5.7.1 Feature Extraction and Training

As explained earlier there are several features for each project in our database. Over 35 of them can be used for the Model Training. There are four main types of features in our database. These are:

- *String format*: repository name, repository types, branching strategy, and so on.
- *Text format*: Description of repository, commit comments, and so on.
- *Integer format*: Commit count, branch count, developer count, and so on.
- *Date format*: Creation date of repository, commit date, and so on.

Not all of these features are suitable for our model. For example, there are features like “contributors”, “branches”, “pull requests”, “issues”, “issue comments”, “pull request comments”, “events” and “commits”. All of these features contain a list of dictionaries with several values inside, like the content of features, the data of added content, and so on. In our model, adding these features can create noisy data, and this in itself can greatly decrease the accuracy of our model [72]. In most cases, only the count of these features was used. This approach was chosen in order to simplify the feature generation process. Additionally, features like “languages” and “branching strategies” were one-hot encoded for the models.

After modifying the features for the model training process, the next phase was to choose the most suitable model for the training process. Based on the structure of the database and the features, four main models were chosen. In the following table, the results of the training and testing process are presented:

Model	Accuracy	Precision	Recall	F1 Score
Logistic Regression	0.9003	0.8125	0.9019	0.8533
Decision Tree	0.6802	0.4952	0.6705	0.5765
Random Forest	0.9344	0.8410	0.9274	0.8948
Support Vector Machine	0.6397	0.4732	0.6221	0.5255

Table 5. 1 Accuracy results of model training process.

The four categories containing results presented can be explained by the following:

- *Accuracy* is a useful metric when the target classes are well balanced. It measures the proportion of correctly predicted instances.
- *Precision* provides a measure of correctness achieved in positive prediction. It is crucial when the cost of false positives is high.

- *Recall (Sensitivity)* indicates how many actual positive instances are captured by the positive predictions and it is essential when the cost of false negatives is high.
- *F1 Score* combines precision and recall into a single metric, providing a balance between them and being especially useful when dealing with imbalanced datasets.

According to the results from Table 5.1 it can clearly be seen that Random Forest has the highest value for accuracy and other categories. There can be several reasons why this model performs much better than the others.

The algorithm's suitability is the primary factor, as previously mentioned. Random Forest is an ensemble method that builds multiple decision trees and merges their results. In comparison with a single decision tree, this method typically results in increased robustness and accuracy. Random Forest is capable of modelling intricate interactions between features, and it is proficient in managing a combination of numerical and categorical features. Second, Random Forest can do a good job by using the strong predictors and reducing the noise from the weaker ones. This is because the data includes both strong and weak predictors, like different counts, repository characteristics, and encoded categorical variables.

There are several advantages to using this model. These can be listed as follows:

- The features of this model can easily be collected from GitHub using the Github API hence they are useful databases for testing and training purposes.
- After the collection and training of the model productivity, each project can be detected much more easily. Instead of performing mathematical calculations and other operations, we can obtain the desired results much more easily.
- During the creation of the model different projects from various backgrounds and properties were used. This model can be used for every type of project to determine the productivity level.

5.8 Thesis III/2: Prediction of Software Development Period

In addition, I created a novel machine learning model designed to forecast the duration of project development. This machine learning model utilises various project parameters, such as the number of developers, the intensity of the development process, the productivity level, branching strategies, and other relevant factors. The newly developed algorithm accurately estimates the duration of the development period in terms of months, with an average error ratio of only a few months, as confirmed by the tests.

Publication related to this thesis: [J4]

Forecasting the duration of software development projects remains a critically debated issue in Information Technology. This chapter highlights the significance of such predictions, especially in the planning phase of software development. An overview of prevalent methodologies, along with their limitations, is provided. Subsequently, I introduce a novel approach to meet this challenge. Similar to my analysis of software development productivity, this approach also

exploits the relationship between various project parameters and the development timeline and offers a new perspective on project duration estimation.

5.8.1 Related work

Prediction of the software development process was one of the major topics of Information Technology systems. There are several important topics of research in this area. These are:

1) Expert Judgment - Within this methodology, the estimation of the development period and other facets of the software development process are forecasted by a special group of experts. This method primarily relies on expert opinion as the foundational basis for prediction [73]. However, a significant limitation of this approach is the inherent bias, as the outcomes depend heavily on the experts' knowledge and experience, potentially leading to subjective and skewed results.
2) Analogy-Based Estimation. This methodology has been widely used not only for the prediction of the development period but also for the prediction of effort, cost, and several other aspects of the development process. Overall, it focuses on examining similar projects and comparing their parameters with a given one. Despite its simplicity, there has been a huge amount of academic research conducted both on the potential and characteristics of Analogy-based estimation methods. Several papers, like [74 - 77] describe this.
3) Agile Estimating and Planning. In agile methodologies, estimation is often done using story points, ideal days, or hours. Agile planning includes methods like velocity-based planning, where future performance is estimated based on historical velocity (the rate at which teams complete work). As was in previous cases, there is also several studies on this field like [78 - 81].

Needless to say, there are many other methodologies and approaches used for the calculation of development period but in most cases they lack objectivity, which is crucial in this topic. The Machine Learning approach that was proposed by myself can solve both issues of being biased and it can be applied to all sorts of projects.

5.8.2 Feature Extraction and Training

As in previous case on the prediction of productivity, in this approach it is also necessary to choose the right set of features for the best result. Some of the features that were mentioned in previous cases have a much more complex structure than needed for our model. This is why these features were modified. The dataset provided information on several aspects of the GitHub projects. The features used for the model were selected based on their potential relevance to the development period. These features included:

- Number of Commits
- Number of Contributors
- Number of Branches
- Number of Pull Requests
- Number of Issues
- Languages

- Productivity (High, Low, None)
- Branching Strategy (Trunk-Based, GitFlow, Github Flow)

The machine learning model employed for this task was the Random Forest Regressor. This model was chosen for its robustness to overfitting and its ability to handle a wide range of data types and complex relationships within the data. The model was used with default parameters, as provided by the scikit-learn [82] library.

The target variable, initially provided in days, was converted to represent the development period in months. This conversion was done to align the predictions more closely with typical project planning timelines, which are often measured in months. Furthermore, to simplify the interpretation of the results, the development periods were rounded to whole months for the final predictions.

The model's performance was evaluated using three key metrics: Mean Absolute Error (MAE), Mean Squared Error (MSE), and the coefficient of determination (R-squared). The results were as follows:

- Mean Absolute Error (MAE): 3.40 months.
- Mean Squared Error (MSE): 52.42 months^2
- R-squared (R^2): 0.441

These results indicated a low level of error, with the model predicting the development period, on average, with an error of approximately 3.40 months. The R^2 value was medium, indicating that the model significantly explains the variability in the development periods of the projects.

5.9 Results and Discussion

Next, we explore the relationship between the productivity of software development processes and the interplay between repository structure, branching strategy, and other critical project parameters. The findings presented here provide a clear perspective due to the fact that they are either based on findings from a relatively small number of projects or they have not undergone any extensive analysis by other researchers. The results are unbiased and as objective as possible due to the diverse nature of the projects analysed, which vary in creation times, technology stacks, and other variables. These factors lend credibility to our findings. The following section of this chapter is dedicated to a comprehensive examination of these findings and their implications.

5.9.1 repository Structure

The literature review section of this paper stresses that the analysis of repository structure was largely disregarded in previous productivity research. Here, we conduct a comprehensive analysis of this topic in order to address this gap, providing real-world data for a more in-depth analysis. The percentage distribution of high, low, and non-productive projects within our database is presented in *figures 5.2* and *5.3*, with separate analyses for Mono repository and Multi repository

projects. This method helps us to better understand the subject matter. In each instance, the proportion of non-productive projects is significantly higher than that of other categories, which is consistent with our expectations. The database primarily comprises open-source projects taken from the GitHub platform, a significant number of which were created by non-professional developers or for non-commercial purposes. As a result, the majority of these projects are classified as non-productive due to their minimal activity.

The proportion of non-productive projects in Mono repository projects is approximately 70%, which is approximately 20% higher than in Multi repository projects, as shown in *Figure 5.2*. In contrast, the rates of high- and low-productive projects are approximately 12% and 19%, respectively. It should be added that the frontend and backend components of Multi repository projects exhibit a percentage that is approximately 2-3 times higher.

Although the productivity level of a project cannot be solely determined by the structure of the repository, these real-world results, which encompass a variety of projects taken from a variety of backgrounds, offer valuable insights and improve our comprehension of the topic.

5.9.2 Branching Strategy

Initially, *Figure 4.4* shown the prevalence of a variety of branching strategies in Mono repository projects. Over 85% of non-productive projects use trunk-based branching as their primary approach. This is a notable trend. This real-world data indicates that the Trunk-based approach is widely used by non-productive projects, despite its popularity among some large corporations such as Google. This may be attributed to the Trunk-based method's structure, which involves a single branch, which makes it difficult to implement new features and resolve bugs.

The Trunk-based approach is less prevalent in low and high productive projects, with an approximate 30% adoption rate. Github Flow and GitFlow, however have a 46% and 21% rate respectively. This suggests that the Trunk-based approach remains relevant in high-productivity projects, but it is not as ubiquitous as the Github Flow, which is mostly popular in both low (30%) and high (46%) productivity projects. This implies that Github Flow is more popular and preferred among high-productivity projects. GitFlow, despite its lower prevalence than the Trunk-based approach, exhibits a distinctive distribution, with projects that are either high productive or low in productivity. This reflects its suitability for more structured and mission-critical projects.

Figure 4.5, which depicts comparable trends in Multi repository projects, further reinforces these findings. The Trunk-based strategy is the most prevalent in non-productive projects; however, it is less often implemented in low and high-productive projects within the Multi repository framework than in Mono repository projects.

High productivity projects exhibit a strong preference for the Github Flow strategy in the multi repository setting. Nevertheless, the Trunk-based approach is only utilised in approximately 15% of high-productive Multi repository projects, suggesting a more restricted preference for this branching strategy in the Multi repository context. This analysis offers a sophisticated comprehension of the adoption of various branching strategies in terms of the type of repository and the productivity of the project.

5.9.3 Project properties

The investigation of the relationship between productivity and team size, programming language, and repository structures have proved to be relatively limited. Agile development methods and team motivation are frequently the primary focus of existing research on team productivity, which frequently neglects the specifics of team size and the development period. The objective of this study is to examine the interaction between these factors and their influence on projects that are both high and low in productivity.

The Mono repository:

A scatter plot chart is depicted in *Figure 5.11*, which demonstrates the correlation between the development period and the size of the team in Mono repository projects. The results for both the Github Flow and GitFlow methods are combined to enhance clarity. Key insights from *Figure 5.11* regarding high-productive projects reveal that the majority of them, regardless of the branching strategy, have a development period of less than two years. Furthermore, the majority of teams are composed of fewer than fifteen members. This pattern is also consistent with projects that are not highly productive. The occurrence of smaller team sizes is understandable in light of the fact that the given projects are open source. Lower productivity is frequently observed in projects with development periods that exceed two years, which is likely the result of a slowdown or halt in project development.

The Multi repository:

Our examination of Multi repository projects, as shown in *Figure 5.12*, reveals significant differences from Mono repository instances. There is a significant disparity during the development period. Compared to mono repository projects, multi repository projects typically have longer development periods. The scatter plot in *Figure 5.12* has a more extensive distribution of data points, which are indicative of high-productive projects in multi-repository scenarios. This implies that, despite the fact that team sizes may be comparable, high-productive projects in a multi-repository context are distinguished by significantly longer development periods than their Mono repository counterparts.

5.10 Concluding Remarks

In this chapter, we addressed the fact that there is a substantial gap in current research by conducting a thorough analysis of repository structures and their relationship to software development productivity. Providing new insights into these areas, we presented in-depth data and analyses for Mono repository projects and Multi repository projects.

The first significant discovery is the increased prevalence of non-productive projects in Mono repositories, which account for approximately 60% of these projects. In contrast, multi-repository projects exhibit a lower percentage of non-productive projects, with a significant difference of approximately 20%. Furthermore, the rates of high and low productive projects in Mono repositories are approximately 12% and 19%, respectively; and these percentages increase significantly in the frontend and backend components of Multi repository projects.

Yet another critical aspect of this chapter is its examination of branching strategies. It tells us that Trunk-based branching is the primary approach employed by nearly 90% of non-productive projects, despite its inability to effectively manage new features and bug fixes. In contrast, productive projects are more likely to implement the Github Flow and GitFlow strategies. The Github Flow, in particular, is the most preferred strategy in high-productive projects, which suggests its effectiveness in real-world applications. While the GitFlow strategy is less prevalent, it is influential in projects that necessitate a more intricate, structured approach.

We also investigated the branch and commit counts of high-productive Mono and Multi repository projects. It was found that the branch count of half of these projects is typically between two and five, regardless of the productivity level. This is especially the case in open-source projects, where smaller branch counts are the norm. The utilisation of over ten branches is common in high-productive projects, suggesting a preference for more intricate repository structures in these types of projects.

The relationship between the development period and team size in high and low productive projects was presented. Smaller teams and shorter development periods of less than two years are frequently linked to high productivity in Mono repository projects. In contrast, multi-repository projects typically have extended development periods. This suggests the necessity of further research into the dynamics of project productivity in various repository structures, as distinct patterns that affect the development period and team size are evident.

The author of this thesis is responsible for the following contributions presented in this chapter:

III / 1. A machine learning model was created in order to identify the productivity level of the project development process. Here there are three levels of productivity: High, Low and None. The model uses several main activity characteristics of the project development period.

III / 2. Machine Learning algorithms are now used to calculate software development periods, utilising various project activity parameters such as issue count, event count, and commit count. This approach, noted for its general impartiality, is thought to be superior to existing methods, and it provides a more effective solution for duration estimation in software projects.

Chapter 6

Thesis Group IV: Collaboration of Software Developer Team

In this section, I introduce a new mathematical approach for the computation of the collaboration rate among developer teams. One of the primary benefits of this approach is that it represents collaboration by a single number. The new system for advising us on the optimal number of developers for projects is based on our previous research and that described in this chapter. Utilising the productivity level and numerous other project parameters, this novel approach was implemented. Furthermore, in order to greatly enhance the precision of this methodology, the findings of my previous study were explored.

Publication related to this thesis: [J5]

We examine the collaboration rate within software development teams and establish a correlation between this rate and other project parameters, such as productivity and branching strategy. At the beginning, a mathematical approach will be applied to determine the developer team's collaboration rate. This calculation primarily employs data from the GitHub platform, specifically a metric called "contribution". Then, the correlation between repository structure and collaboration is investigated. In a manner similar to the analysis of productivity, we determine the repository structure that best encourages a collaborative work environment based on the preferences of developer teams.

Next, we investigate the correlation between the rate of collaboration and a variety of branching strategies. Project managers can develop a model for determining the optimal team size by determining which branching strategy best supports team collaboration. We introduce a model that suggests the optimal number of developers for a project by considering factors such as the anticipated development period, programming language, and repository structure. The aim of this model is to optimise project management strategies and facilitate the formation of effective teams.

6.1 Introduction

The exploration of team collaboration among software developers and its influence on software development productivity is a long-standing topic in the field of information technology. This subject has been the focus of extensive academic research, with some foundational studies published as far back as half a century ago, exemplified by seminal works like [83].

The foundation for comprehending the complex relationship between productivity and team dynamics in software development was established by these initial investigations.

In this chapter, we intend to conduct a comprehensive examination of these and other academic studies that are relevant. This analysis will not be restricted to the above studies; it will also include a variety of scholarly efforts that have attempted to establish a connection between the structural aspects of software development and team collaboration. One of the primary objectives of this study is to highlight the existing research gaps in this area, especially the absence of studies that concentrate on specific aspects of team collaboration and productivity in software development.

In this chapter, we will explore the ways in which the collaboration of software developers varies across various repository structures and how these variables affect productivity levels. We shall attempt to improve our comprehension of effective collaboration strategies in the field of software development by investigating these dynamics, provide a deeper understanding of the impact of structural decisions on team efficiency and output.

6.2 Related Work

Software development processes are distinguished by their collaborative nature, which requires the coordination of numerous software engineers to develop intricate software systems. The development of a shared understanding among team members is a critical component of this collaborative endeavour. This understanding is centred on a variety of approaches, each of which represents its own model throughout the development process. One of the objectives stated in [84] is to manage dependencies among activities and organisations, which may be accepted as the primary objective of team collaboration during software development. This includes a wide range of collaborative activities, including critical management responsibilities such as the decomposition of work into individual tasks, the establishment of their sequence, and the subsequent monitoring, evaluation, and maintenance of control over the plan of activities [85]. The significance of a developer teams' collaboration in software development has increased, especially in light of the widespread adoption of version control systems. Version control systems are classified into two broad categories: namely distributed and centralised.

Authors of [86] refer to a study that examines the influence of these version control systems on the software development process and team collaboration. The researchers conducted surveys among developer teams and discovered that commits in DVCSs, especially in Git, are smaller and more isolated than those in other systems. This may be due to the high level of collaboration among developers, who share the workload by dividing it into smaller tasks. The concept of new product development (NPD) has been introduced as a result of the rapid development of new tools and innovations in the field of software development, in addition to version control systems. Businesses employ New Product Development (NPD) as a method to conceptualise, design, develop, and introduce new products to the market. It entails a sequence of procedures that help a company turn concepts into marketable products or services [87]. This concept posits that there are numerous instruments available to enhance software team collaboration during the development process. Basecamp [88], Asana [89], Teamwork [90], and others are among the most widely used tools. These tools and others have gained huge popularity among the developers for

the past couple of years and several research papers have been published to check the impact of these tools on the collaboration of a software team [91, 92]. But almost none of them provide a measurement for the collaboration during the development process.

There are numerous methods for evaluating collaboration within a software team. For instance, numerous researchers examined collaboration in the context of Agile systems, as evidenced by papers [93–97]. In these studies, various aspects of team collaboration are examined, including the psychological objectives of team members, the tools employed for collaboration, and collaboration among various developer groups. Although these papers offer valuable insights into the subject matter, none of them specifically examine team collaboration from a productivity perspective. Furthermore, those that were presented also include a variety of methods for evaluating the influence of collaboration on the development process. Papers such as [98, 99] examine the collaboration of software teams by studying the communication among team members.

In essence, the researchers in both papers evaluated the level of collaboration among the software team and the comments regarding issues and code changes. This method may be beneficial for the analysis of a single, large project in which all team members' communication has been stored or recorded. Nevertheless, the development of this type of communication scheme can be time-consuming, requiring hours or even days, as indicated in [98]. Consequently, this process proves impractical when working on a large number of projects, as is the case in our situation. One article [100] is a prime example of a successful analysis of productivity and collaboration, as it employed "pull requests" as its primary source of data. Pull requests (PR) are proposals for new code or modifications to the existing code that are made during the development process. These PRs may be accepted or rejected by the primary developer or team leader. This article has a significant advantage in that it generates graphs in a manner similar to that of previous studies, but it exclusively employs PRs. Consequently, it avoids the time-consuming tasks that were present in previous papers. In this way, they categorised software teams into five categories and evaluated their collaboration accordingly. However, they exclusively employed PRs in the productivity calculation, as was the case in the collaboration case, which may result in some inaccuracies. This is why we opted for a more sophisticated methodology for productivity calculation.

Another article [101] presents an additional comparable methodology. Researchers used logs to quantify the workloads of developers. By ascertaining the quantity of work that each developer had completed, they were able to ascertain the centralization of work. This method is much more straightforward than the previous ones and it can be implemented in nearly every repository on the Github platform. Our paper will also employ a comparable methodology, albeit in a more straightforward and efficient manner.

Conducting surveys is also one of the main approaches to measuring the collaboration rate and studying its main characteristics. Papers like [102, 103] show the different levels of collaboration and the main parts of the development phase where the collaborators do the most. For example, an article [102] shows that almost 50% of the job during maintenance tasks involves the collaboration of several developers. Professional developers with years of experience working on the GitHub platform have provided the answers that have driven these insights.

The straightforward productivity calculation approach is explained in numerous publications, including [104]. The collaboration in this study is solely based on the size of the

developer team, and productivity is determined by the number of commits and changes made to the source code. Our investigation into the relationship between repository structure and team collaboration did not result in any academic paper, despite an extensive research effort that spanned nearly four decades. This emphasises the fact that there are still aspects of software development that have not been fully examined in research on team collaboration.

6.3 Calculation of Collaboration

In software development, team collaboration is a crucial element that helps bring together a group of individuals with a variety of professions and areas of expertise. This collaborative endeavour is indispensable, as it encompasses the entire spectrum of software development, maintenance, and improvement. Development teams can generate software that is of superior quality and effectively addresses user requirements by fostering effective collaboration. And this collaborative approach ensures that software projects are completed within the designated timelines and permits more efficient management throughout the software lifecycle.

Extensive research has been conducted on different aspects of development team collaboration, as previously mentioned. Various forms of collaboration in software development are the focus of certain researchers. Alternative methods are employed by others to improve these types of collaborations. Furthermore, the majority of research in this area is conducted through interviews with company employees or developer groups. The objective of these studies is to shed light on the various facets of collaborative work in the software development context, including team interactions and developer expertise. Its importance in achieving successful project outcomes is underscored by using a comprehensive approach to understand better how collaboration works in software development.

6.3.1 Thesis IV/1: Methodology

I also devised a new mathematical method for calculating the degree of rate of developer team collaboration. This method uses parameters provided by the GitHub platform. One of the main advantages of this method is that collaboration can be characterised by a single number.

Publication related to this thesis: [J5]

As mentioned in the previous chapters, collaboration of the developer team has been analysed from many different aspects so far. Some researchers check the types of collaboration as described in a paper [105] or simply improve it by using different methods like those outlined in [106]. Several other methods are also described in second section of this paper, and it appears that in most of the cases the collaboration rate of software teams was measured using three main methods. These are:

- Surveying the different team members and developers [83, 84, 87].

- Checking the communication logs during the development process. This requires checking the different comments and messages of team members when making changes on the source code [98, 99, 100].
- Measuring the workload of team members and checking to see if it is distributed evenly [101].

Following an examination of each of these methodologies, it soon becomes apparent that the most appropriate method for our objective is to assess the workload of the team members. First of all, it was significantly less time-consuming and simpler than the first two possibilities. Secondly, the approach is not really appropriate for our research work because not all repositories contain communication logs between developers. It is evident that a perfect collaboration rate cannot be achieved solely by measuring the workload share of developers. However, in our particular situation, this method is definitely more convenient, and as previously mentioned, there are comparable methods available that demonstrate the efficacy of this approach.

In order to measure the workload of team members in most of the cases researchers measure the amount of their PRs, accepted commits, etc, but in our case all of this information can be retrieved directly. Since we are collecting our projects from the GitHub platform and API request like below will let us learn the exact amount of contribution a developer made to the project [107]. GitHub API provides a list of contributors of project and their data after sending the following request:

https://api.github.com/repos/{username}/{reposiotry_name}/contributors

The result is sent in json format which display different parameters like *ID, Login, User profile, Repositories* and *Contributions*. The value of the *Contributions* parameter is represented by a number, and a sum of following activities of user in project:

- Commits: This includes creating new commits, as well as merging commits via pull requests.
- Pull Requests: The number of pull requests created and the number merged are typically displayed.
- Issues: This includes both issues created by the developer and issues they participate in, such as commenting or providing solutions.
- Reviews: The number of code reviews a developer participates in may also be displayed.
- Comments: This includes both creating new comments and replying to existing ones.

To determine whether a team has a relatively equal work share among its members, you can calculate the coefficient of variation (CV) [108] for the work distribution. The coefficient of variation measures the relative variability of data points compared to their mean (average). So, the algorithm for calculating the coefficient of variation of the project is the following:

- Calculate the total sum of contributions.
- Calculate the mean value of contributions.
- Calculate the standard deviation of the contribution values.
- Calculate the coefficient of variation

This way we can calculate the workload share of the developer team and express it in mathematical terms. In other words, doing it this way it is possible to represent a collaboration rate of the whole developer team by a single figure.

6.3.2 Mathematical Specification

In order to make our algorithm more understandable we will also provide a formal mathematical specification for it:

Input:

- A dataset *X* with *n* data points: $X = \{x_1, x_2, x_3, \ldots, x_n\}$

Output:

- The Coefficient of Variation (CV) for the dataset *X*.

Algorithm:

1. Calculate the Mean (*Mean*) value of dataset *X*:

$$Mean = \frac{1}{n}\sum_{i=1}^{n} x_i$$

   - The mean is the arithmetic mean (average) of the data points in *X*.
   - *n* is a number of data points in *X*.
   - x_i represents each individual data point in the dataset.

2. Calculate the Standard Deviation (*StandardDev*) of the dataset *X*:

$$StandardDev = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - Mean)^2}$$

   - $StandardDev$ is the standard deviation of the data points in *X*.
   - *n* is a number of data points in *X*.
   - $Mean$ is the calculated mean from step 1.
   - x_i represents each individual data point in the dataset.

3. Calculate the Coefficient of Variation (CV):

$$CV = \frac{StandardDev}{Mean} \times 100$$

   - CV is the coefficient of variation, expressed as a percentage.
   - $StandardDev$ is the calculated standard deviation from step 2.
   - $Mean$ is the calculated mean from step 1.

4. Return calculated CV as result.

A higher coefficient of variation means a greater variability in work distribution among team members. Perfect collaboration would typically result in a lower coefficient of variation, closer to 0. This means that the best collaboration rate is “0”. So, if the rate value is 0 then the workload of the development process has been shared equally among all developers and this

allows us to say that the team has a high collaboration rate overall. But the 0 value is more or less a perfect condition and only a few percent of the projects have this rate of collaboration. Based on our calculations the ideal collaboration rate is “100”.

6.4 Repository Structure and Collaboration

The repository structure is once again prominent, as it was in previous instances. It is crucial to determine the optimal repository structure to optimise the collaboration rates, as it is a critical component of the project planning and development process. In this part I will, demonstrate the preference for repository structure measures among the respective collaboration levels. It should be added that the collaboration rate in this research work is based on a reverse value concept before further undertaking the analysis.

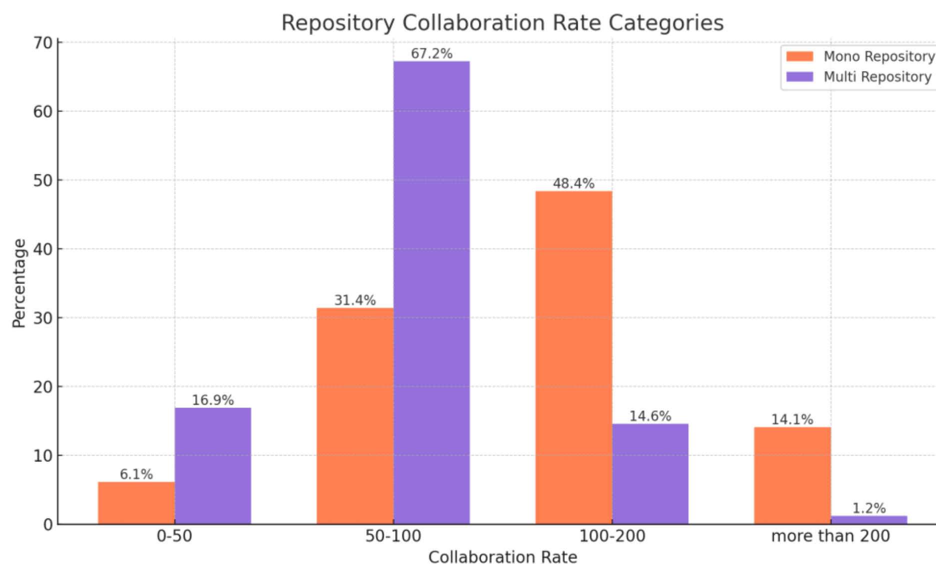

Figure 6. 1 A percentage comparison of different collaboration levels in Mono and Multi repository structures.

The distribution of Mono and Multi repository projects across four major collaboration rate categories is shown in *Figure 6.1*. These categories were identified via a comprehensive analysis and they are regarded as the most suitable for describing the collaboration rate of development teams in projects. The data figures suggest that Multi repository projects typically have a higher rate of collaboration than Mono repository projects. Multi repository projects exceed mono repository projects by approximately 10% in the category of highest collaboration. In contrast, Mono repository projects account for a smaller portion of the top collaboration category, with a value of approximately 6%. The contrast is more pronounced in the second category, which is defined as a moderate yet more common collaboration rate ("50–100"). This range encompasses 67% of Multi repository projects, as opposed to 31% of Mono repository projects. Although this implies a reasonable level of collaboration among Mono repository teams, the rate is significantly lower than that of Multi repository projects. Mono repository projects have a significantly higher percentage share than multi repository projects in the remaining two categories, where the collaboration rate decreases. For example, their percentage share is respectively nearly three and

ten times that of multi repository projects. This pattern tells us about that projects in the Mono repository are more likely to be classified as having a lower collaboration rate. These observations lead to the inference that teams using the Multi repository approach are more inclined towards a collaborative development environment. The reasons for this might vary, including the inherent structure of the Mono repository approach, typically smaller team sizes in Mono repositories compared to Multi repositories, and differences in development culture. A further exploration of these potential factors will be undertaken in the remaining sections of this chapter, with the aim of providing a deeper understanding of the dynamics that influence collaboration in different repository structures.

6.5 Productivity and Collaboration

The methodology section of this thesis gives an explanation of the productivity calculation. To ensure clarity, productivity has been categorised into three categories: high productivity, low productivity, and none. The first two categories indicate the level of productivity, while the third category shows the project with virtually no productivity. The purpose of this section is to determine the correlation between productivity and collaboration. The third category has been omitted from the calculations in this section due to its negligible value; however, it will be provided in the later sections. The results of two repository structures will be analysed in separate graphs to determine the relationship between productivity and collaboration.

The Mono repository.

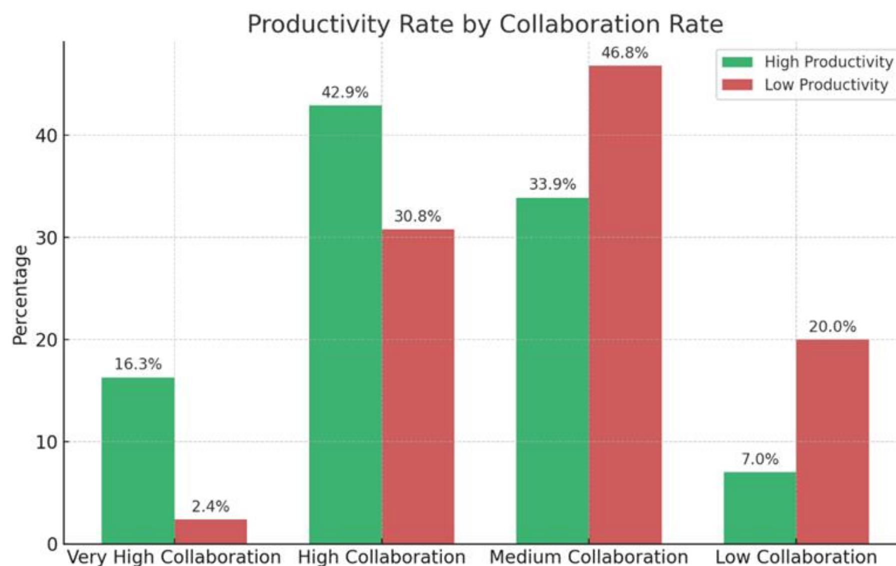

Figure 6. 2 The percentage share of High and Low productivity of Mono repository projects based on the collaboration rates.

The structure of *Figure 6.2* is not unlike to that of the preceding figure. The percentage share of Mono repository projects with high and low productivity is given in this section. As was the case in the preceding section, we will commence by examining the initial category of very high

collaboration. Here, a clear distinction between the percentage share of projects with high and low productivity is evident. The percentage of projects that are high productive is nearly 16%, while those that are lowly productive are only around 2%. This trend is more or less altered in the second category, which has a high rate of collaboration. Here, projects that are high productive account for nearly 43% of the total, while those that are low productive account for only 31%. This discrepancy, when contrasted with the values from the first category, allows us to state that approximately 60% of the high productive projects have a high or very high collaboration rate, whereas this value is only around 33% for low productive projects. In the third category, the trend of the disparity between high and low productive projects is inverted. In this instance, high productivity projects account for nearly 34% of the total, while low productivity projects dominate with approximately 47% of the total. This type of differentiation has also been observed in prior charts; however, it is not as large as it was in the third category of *Figure 4.1*. The fourth and final category demonstrates that low productive projects have a threefold greater percentage share than high productive ones. In this context, projects that are high productive account for 7% of the total, while those that are low productive account for 20%.
The Multi repository.

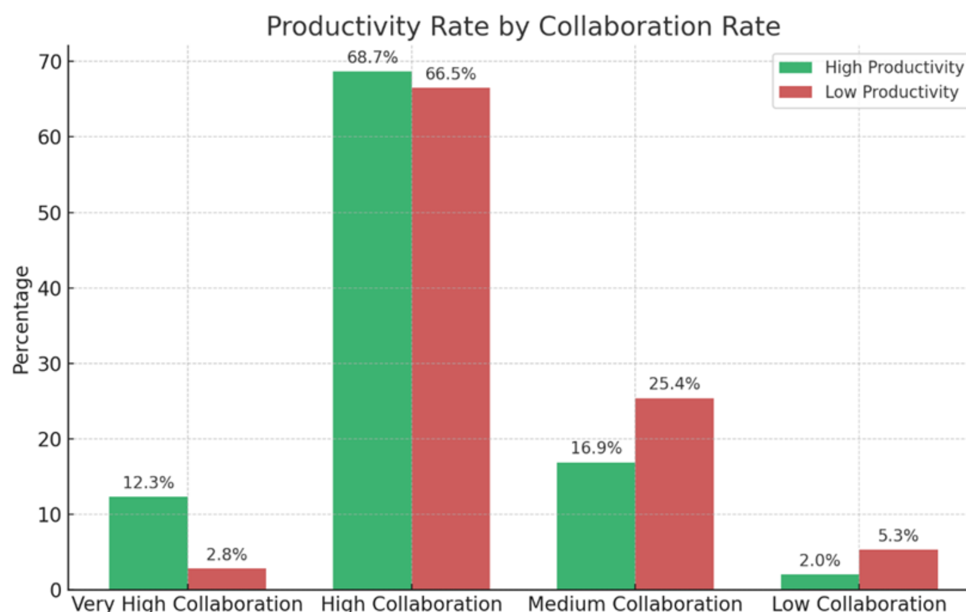

Figure 6. 3 A comparison of the collaboration rate in different productivity levels among Multi repository projects.

The results of a comparable analysis, excepts for the Multi repository projects, are shown in *Figure 6.3*. In this case, some similarities and differences are apparent. High productivity projects account for nearly 12% of the first category, which have an exceptionally high collaboration rate, while low productivity projects account for 3%. This is a trend that is roughly comparable to the one depicted in the previous *Figure 6.2*, in where high productive projects had a higher percentage value than low productive projects. It tells us that projects with a high productivity level in both Mono and Multi repository structures have a significantly higher percentage value than those with a low productivity level at a very high collaboration rate. The

results are quite different from any trend that has been previously investigated in the second category of the collaboration rate. In this case, the percentage value of both high and low productive projects on the Multi repository side is nearly identical. It means that the productivity level of Multi repository projects is not significantly affected by the high collaboration rate. The two previous charts did not have this type of relationship. The general nature of Multi repository projects is one potential explanation for this. As shown in *Figure 6.1*, Multi repository projects have a significantly higher collaboration rate than mono-repository projects. This may account for the low percentage value in the medium and low collaboration categories in *Figure 6.2*. For instance, in the fourth category, the aggregate sum of all Multi repository projects is nearly 7%; however, this figure was 27% for Mono repository projects. This serves as further evidence that projects with multiple repositories have a significantly higher rate of collaboration than those with a single repository. The collaboration rate also influences the productivity level in the Multi repository project, although the impact is much smaller than in the Mono repository case.

6.6 Branching Strategies and Collaboration

Software development necessitates the implementation of branching and collaboration strategies. It became apparent during our investigation that there are correlations between the collaboration rate of projects and branching strategies. This can be attributed to a variety of factors; however, this section here serves as an introduction to our discoveries in this domain. The results have been divided into two sections to get a more detailed comprehensive view of the data.

6.6.1 A comparison of the branch count and collaboration ratio of projects

VCSs are essential in a variety of project development environments, including large corporations, small developer teams, and freelancers. One of the significant benefits of these systems is their capacity to save various versions of projects. This feature enables developers to revert to previous code versions, thereby facilitating more effective code management and collaboration. And VCSs allow developers to communicate and collaborate in a seamless manner throughout the software development process.

Although there are numerous VCS alternatives, this chapter just concentrates on Git. The primary justification for this decision is the vast GitHub platform, which is home to millions of developers and projects. This extensive user base significantly simplifies the process of identifying a variety of developer teams and projects for analysis and study.

Developers generally divide their projects into branches, which can be subsequently merged into the primary source code or project. These branches are designed to ensure that the development process is both secure and clean, enabling each developer to work on their designated section of code without affecting others. There are numerous branching strategies that are determined by the number of branches and their intended functions within the project. The following strategies will be the primary focus here: Trunk-based, Github Flow, and GitFlow. The research findings will be divided into two Mono and (Multi repository projects) for a comprehensive analysis, as in the previous case.

The Mono repository

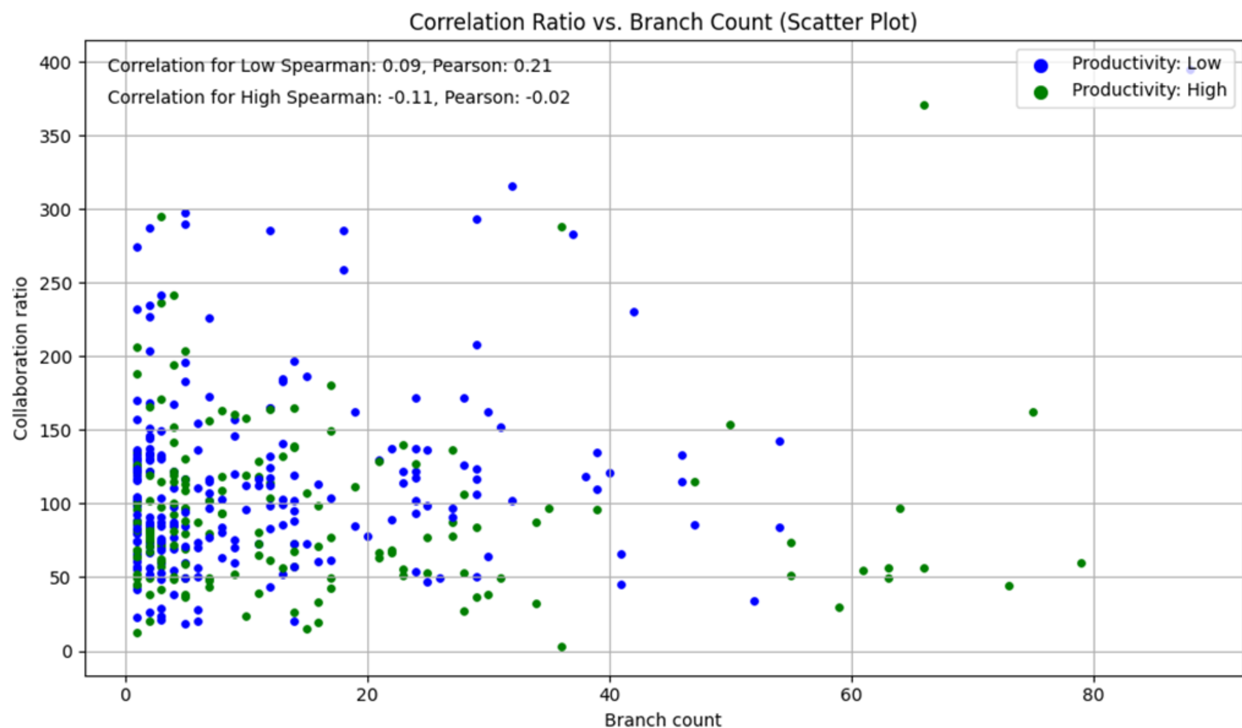

Figure 6. 4 A comparison of the branch count and collaboration rate of Mono repository projects including productivity levels.

The correlation rate, productivity, and branch count of Mono repository projects are shown in *Figure 6.4*. First of all, it is necessary to clarify a few critical components of this chart before we proceed. It is evident that the correlation has been demonstrated via numerical data rather than any category or percentage. The calculation of correlation is described in *Section 6.3* as a distribution of workload among the developers. The standard deviation was computed to determine the extent of the variance between the workloads of the developers. If the value is nearly zero, it means that the workload is distributed equally; and then, we consider it to be a high collaboration rate. The software team collaborates significantly less as a result of the higher deviation value (referred to here as the collaboration rate).

The purpose of selecting this representation is to show the correlation between the branch count and the collaboration rate of the projects. Before proceeding with the analysis, it is necessary to state that Pearson and Spearman correlations were implemented for this and all subsequent charts.

Upon initial inspection, it is clear that the correlation values are relatively low for both high and low productive projects. The branch counts more or less demonstrate the branching strategy itself, which is why it is obvious that the collaboration of project development has almost no connection with the branching strategy used in the given projects. Also, this chart contains supplementary data regarding the utilisation of branch count in projects with high and low productivity. It seems that projects that are high productive have a higher branch count than those that are low productive. This may be due to a variety of factors. Firstly, the branch count is a representation of the developers' work output. Therefore, it is common for projects with a higher branch count to appear to be high productive, as work output has a significant impact on our productivity measurement.

The Multi repository

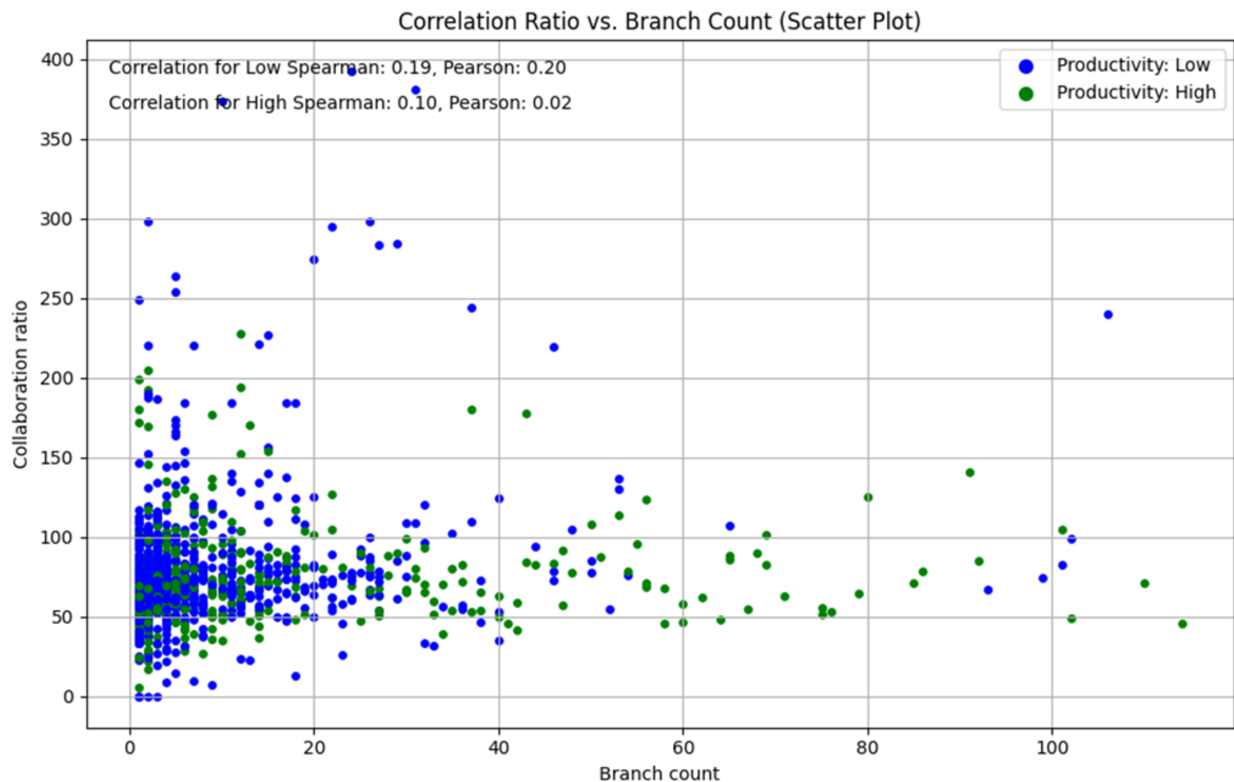

Figure 6. 5 A comparison of the branch count and collaboration rate of Multi repository projects including productivity levels.

The same analysis results are depicted in *Figure 6.5* as for the Mono repository projects. Once more, it must be stated that collaboration has not been represented as a category or percentage this time. The collaboration rate values on the y-axis represent the standard deviation value from *Section 6.3*. If the value is closer to 0, it means that the project has a high collaboration, while the opposite means a low collaboration in the software team.

As in the previous case, the Spearman and Pearson correlation values for high and low productive Multi repository projects are both quite low. Once again, this implies that the collaboration rate of projects is not significantly correlated with the number of branches or the branching strategy. Although low productive projects do have a correlation value of approximately 0.20, this is insufficient to generate any statistically significant effect. However, this chart can also provide an alternative outcome regarding the repository structure and branching strategy. In contrast to Mono repository projects, multi repository projects typically have a significantly higher branch count, irrespective of the collaboration rate. This implies that branching strategies such as Github Flow and GitFlow are significantly more prevalent among Multi repository projects. And the chart's points again demonstrate that Multi repository projects have significantly higher levels of collaboration than Mono repository projects do.

The results of *figures 6.4 and 6.5* indicate that there is no significant correlation between the collaboration of software teams and the branching count or strategy in either Mono or Multi repository projects. Moreover, it is clear that the branching count or strategy of the projects are significantly correlated with the repository structure.

6.6.2 A comparison of commit count and collaboration ratio of projects.

We explored the subject of branches and their diverse types in the preceding section. Now, it is necessary to discuss the concept of commits. The act of saving edited files, where each commit records changes made to one or more files within a branch, can be succinctly described as commits. In order to capture critical information, such as modified files, specific lines of code changes, the individuals responsible for the modifications, and the timestamps of these alterations, Git assigns a unique identifier, known as a SHA or hash, to each commit [109]. Essentially, commits function as a comprehensive representation of the development workflow and they can be used as a metric to assess the amount of work that has been completed during the software development process.

In this section, we will examine the correlation between the collaboration rate and the commit count of developer teams in both Mono and Multi repository projects. We seek to gain an understanding of the relationship between the quantity of commits and the level of collaboration within a development team by analysing this correlation.

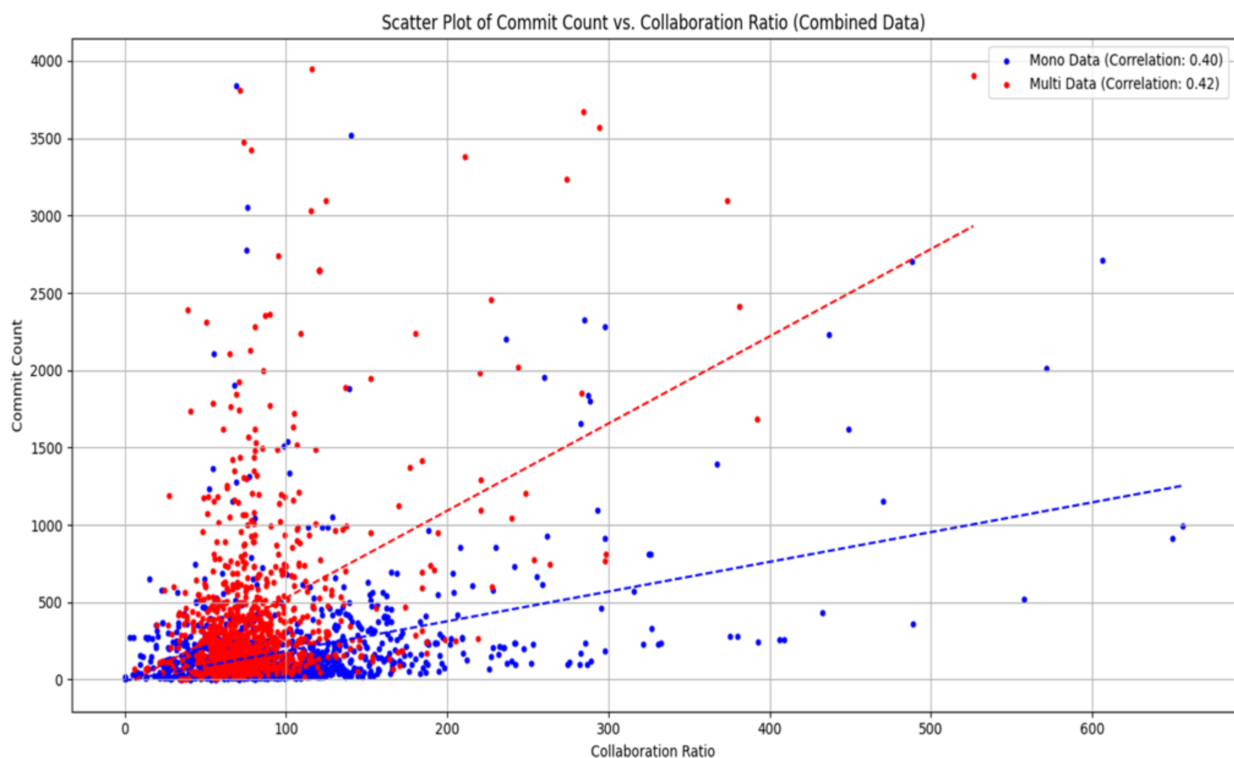

Figure 6. 6 A comparison of the commit count and collaboration rate of Mono and Multi repository projects.

The relationship between the commit count and collaboration rate in both Mono and Multi repository projects is shown in *Figure 6.6*. According to the chart, the correlation coefficient is approximately 0.40 for both Mono and Multi repository projects. Furthermore, we shall mention that the Pearson method was employed to calculate the correlation, and again, the standard deviation values of collaboration were employed instead of any percentage or category.

This positive correlation between commit count and collaboration ratios has to be

understood in a negative way. The results are consistent for both Mono and Multi repository projects, as a lower collaboration rate (standard deviation) means a greater collaboration. There is a medium level correlation between the standard deviation and the commit count in both cases, indicating that the software teams' collaboration decreases as the commit count increases. There are numerous potential explanations for this, the workload being one of the most obvious. As stated in Section 5.3, the distribution of workload among the developers was used to calculate the collaboration. A substantial quantity of commits implies a substantial workload, which may ultimately lead to an unequal distribution of the workload among the developers. The activity of contributors diminishes as the project advances, as indicated in a paper [101]. This can also be said about the project's size and commit count in this instance. However, it should be recalled that this correlation value is a medium value, and there may be other, more intricate reasons for this. Furthermore, the database we employed exclusively includes open-source projects from the Github platform, and this perspective may differ greatly in commercial professional projects. Overall, it is clear that there is a moderate correlation between the collaboration rate and commit count for both Mono and Multi repository projects.

6.6.3 Developer Team Size vs Collaboration

The size of a developer team is the number of developers that are involved in a project, excluding team managers and non-source code contributors. Although other team members are crucial to the software development process, their participation does not directly influence the development or writing of code. Scrum masters, project managers, project owners, and other individuals in comparable positions may comprise this group. This part will now focus on the developers who are actively involved in the modification of the project's source code, with the exception of the above non-coding personnel.
Two factors serve as the strategy's main drivers:

- The database we created based on GitHub projects and GitHub API only stores the actions of developers and no other team members since it is the main focus here.
- There are already similar research findings about the effect of project managers and other team members on the development process, but here we just focus on developers [110 - 114].

In line with previous approaches, the findings of both Mono and Multi repository projects will be presented separately for the sake of clarity and to facilitate a comprehensive understanding of the respective scenarios. Furthermore, projects with fewer than three developers have been excluded from our analysis. This decision is based on the rationale that discussing collaboration within developer teams comprising only one or two members would not yield any meaningful insights or contribute significantly to the overall research objectives. By focusing on projects with three or more developers, the dynamics of collaboration can be better explored and evaluated within larger developer teams.

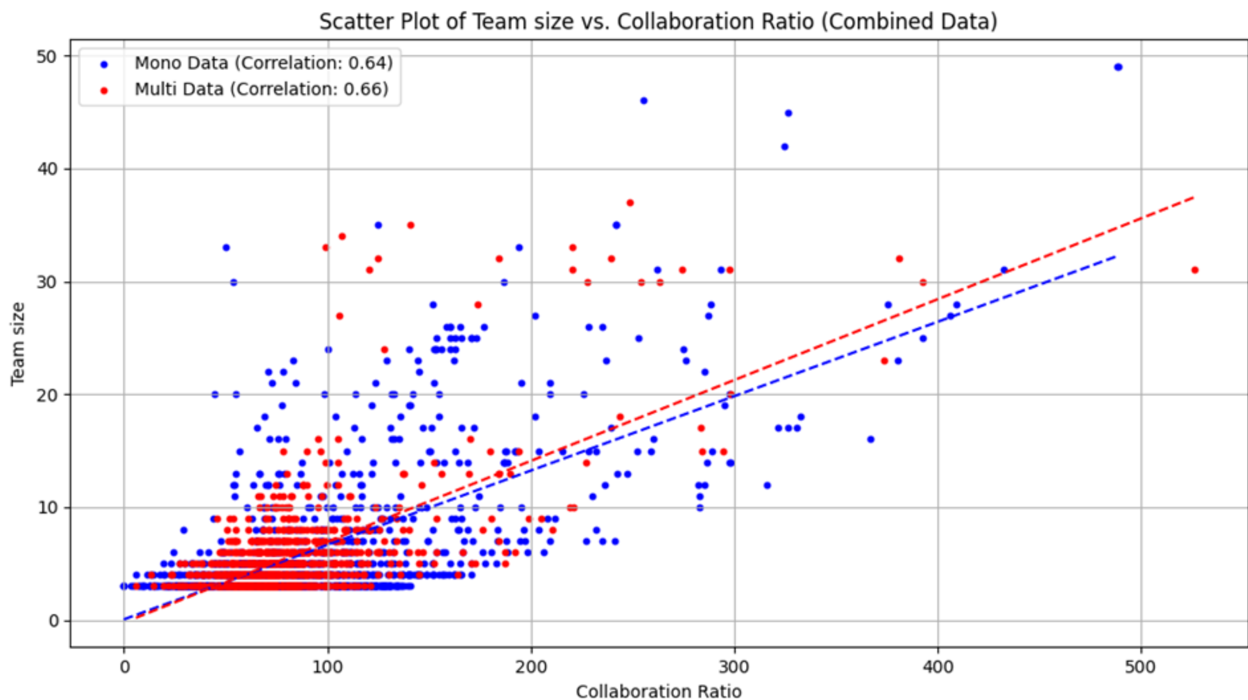

Figure 6. 7 A comparison of the team size and collaboration rate of Mono and Multi repository projects.

There are some differences between *Figure 6.7* and the previous charts. The Pearson method was employed to calculate the correlation between the team size and collaboration in this case, and the collaboration ratio is depicted on the x-axis of the chart. The negative correlation between team size and collaboration rate is not unexpected, as the higher value in the x-axis indicates less collaboration. The correlation is greater than 0.60 in both instances, which is not exceptionally high, but this exceeds the medium value. This implies that the collaboration of a small software team is significantly greater than that of a large team in both Mono and Multi repository structures. What is more, this chart serves to substantiate certain previously stated assertions regarding the rate of collaboration in Mono and Multi repository structures. From the figure, it is clear that projects with multiple repositories have significantly higher levels of collaboration than those with a single repository.

Furthermore, this chart furnishes us with data regarding the sizes of teams in both Mono and Multi repository projects. The majority of the dots from both repository structures are grouped below the y axis, where the team size is 10. This means that the majority of the projects in the database have fewer than 10 developers in their teams. Once more, it should be stressed that these findings are exclusively derived from the database, and they may differ slightly certain specific research work or significantly more complex projects.

6.6.4 Development Period vs Collaboration

The development period, in the context of this study, refers to the time duration between the first and last commit recorded in a GitHub repository. The calculation process for determining this period is quite similar for both Mono and Multi repository structures.

In the case of a Mono repository, the calculation involves subtracting the value of "created_at" from the value of "updated_last" in the database. This computation yields a time period, which can then be converted into various date formats such as hours, weeks, days or months. For the purpose of simplicity and accuracy, we will utilise the day format.

Conversely, the Multi repository scenario presents a slightly more complex calculation. Given the presence of two repositories (frontend and backend), there are corresponding "created_at" and "updated_last" values for each. Therefore, prior to performing the subtraction, the smallest "created_at" value and the largest "updated_last" value are identified.

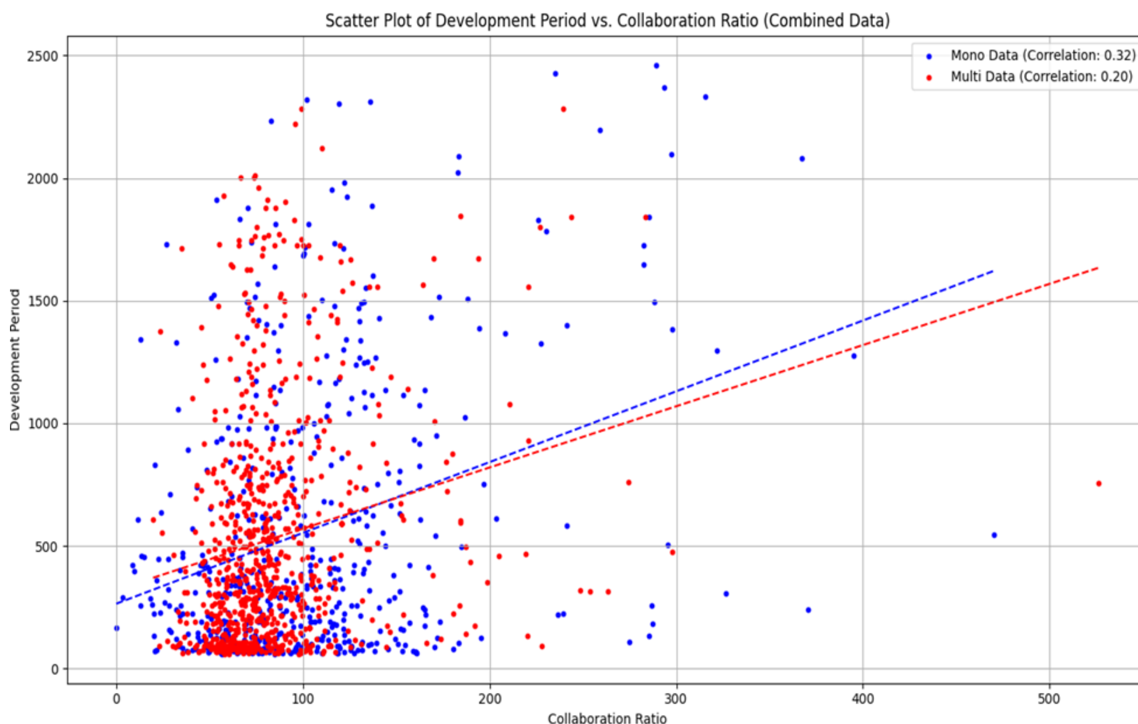

Figure 6. 8 A comparison of the development period and collaboration rate of Mono and Multi repository projects.

In order to make some meaningful comparisons, the same chart was used to present projects that employ both Mono and Multi repository approaches. This will lead to a better analysis and evaluation of the two approaches being compared. The objective of this study is to offer a clear and concise overview of the development periods that are associated with both Mono and Multi repository structures through the use of calculations and visual representations.

Figure 6.8 gives us a comparative analysis of the development period and collaboration rate within both Mono and Multi repository structures. The correlation values depicted in this chart are not that significant as it was in the previous case. For the Mono repository structure, the

correlation ratio is 0.32, while for the Multi repository structure it is 0.20. These correlations were calculated based on the Pearson method as in the previous charts. The development period does not significantly affect the collaboration of projects, as revealed by the weak correlation between the two repository structures. And the chart contains a significant observation. Namely, the majority of projects that employ the Multi repository structure have significantly longer development periods, which is consistent with the typical characteristics of this particular structure. In contrast, the Mono repository structure has a negative correlation between the two variables, as a slight decrease in the team collaboration rate is accompanied by an increase in the development period.

These finding offer some clarity of the complex interplay among repository structures, the development period, and collaboration rate. The distinctive challenges and considerations associated with each structure are made clear by the disparities observed between the Mono and Multi repository approaches. Software development practitioners can make informed decisions about the repository structure they choose by recognising these patterns, considering the potential impact on team collaboration and the duration of the development process.

6.7 Thesis IV/2: Predictive Modelling for Developer Team Sizing

I introduced an advice system that provides developers with recommendations for the optimal number of developers based on the project's parameters. Utilising the productivity level and numerous other project parameters, this novel approach is implemented. Furthermore, in order to greatly enhance the precision of this methodology, I implemented the findings of my prior research.

Publication related to this thesis: [J5]

Determining the optimal team size for a software development project is crucial for its success. This analysis examines various metrics taken from a repository of GitHub projects, looking for patterns and insights that can guide the assembly of development teams. This report presents the findings in a detailed, visual manner to facilitate understanding and utility.

Our methodology employed a robust quantitative analysis, exploring an extensive dataset of GitHub projects. Each project in this dataset was encapsulated in a JSON file, rich with various metrics indicative of the project's health and activity. Of key value amongst these metrics were the 'star_count' and 'fork_count', serving as proxies for the project's popularity and community engagement, respectively. And the 'number of contributors' was a critical metric, providing a direct insight into the team size.

The initial phase of the analysis involved preprocessing the data to sort out the projects that met our criteria for 'high performing'. This categorisation was multi-faceted; projects not only had to have a high count of stars and forks (top 25th percentile), but they also had to be tagged with

'High' and 'Very High' productivity. This subset of projects was presumed to exemplify optimal operational and team dynamics, thereby serving as a good foundation for our subsequent analyses [107].

A grouping mechanism was implemented after the filtration process, which segmented the projects based on the programming language used, the branching strategy implemented, and the project's lifecycle duration. The duration was further divided into three distinct categories: short-term (0–3 months), medium-term (4-6 months), and long-term (7+ months). This stratification allowed for a more sophisticated analysis, which accounted for the variation in operational dynamics among various project types. The primary objective of the analysis was to determine the average number of contributors per project group. This indicator was our primary metric for determining the optimal team size. It is based on the premise that high-performing projects, identified via our criteria, are likely to have optimised their team sizes for efficiency and productivity, and hence they should provide a reliable benchmark for other projects.

6.7.1 Results of Model

As was stated in the previous section, several parameters can be used for this modelling and therefore it greatly depends on the need of developers and what type of parameters they can add. For the testing of our approach, we will use three parameters called *Programming language, Branching Strategy* and *Development period.*

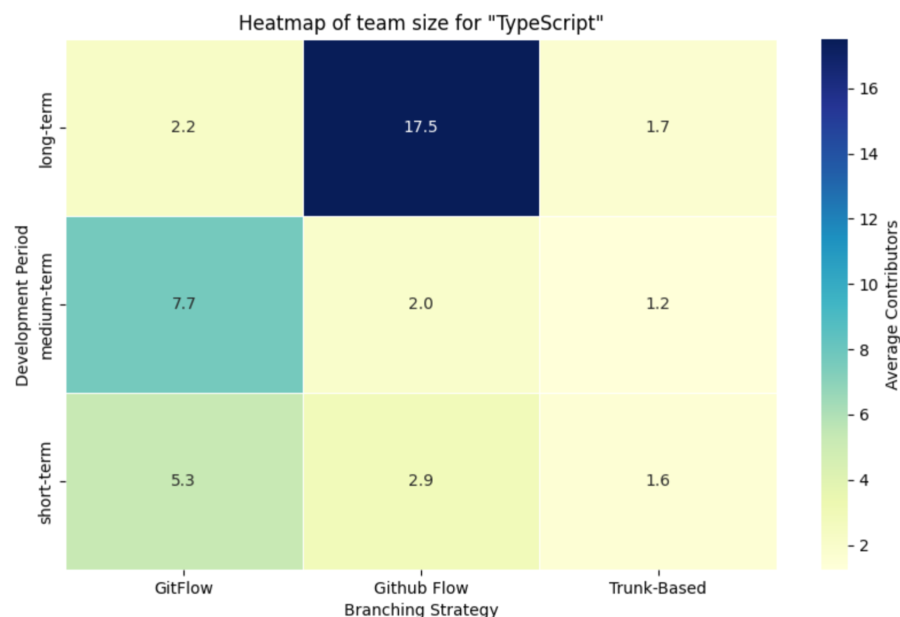

Figure 6. 9 A heatmap of the average team size for TypeScript projects.

The results shown in *Figure 6.9* is only one of the cases that can be driven from our database. It seems from here as well that projects which use the Github Flow branching strategy and have a long-term development period usually have 17 developers in their team. Once more, as was shown in our previous analyses, projects using Trunk based approach have a much smaller team size than the others. These results can be improved by adding additional parameters like

repository structure, more precise development period and so on. But nevertheless, these results show the power of our database and how it can be implemented in order to create advice systems for developers and project managers. One of the main advantages of this approach is that users can add their own parameters or remove the ones they don't need. For example, they can add parameters like "commit count" or "project size" and remove ones like "programming language". This way the project owners who value the project size more than programming language can create their own advice systems and get an idea about the optimal team size.

6.8 Results and Discussions

Some direct conclusions can be drawn after a thorough examination of the percentage values of both repository structures in the above-mentioned collaboration rates. The high percentage value of Multi repository values in both very high and high collaboration categories made it clear that these types of projects tend to have significantly higher collaboration levels than Mono repository ones. The percentage values of multi-repository projects are nearly three times higher in the first collaboration category and twice as higher in the second collaboration category. Furthermore, it is evident that nearly 86% of all Multi repository projects in the database have a high or very high collaboration rate after summarising this percentage value. And the Mono repository projects account for only approximately 43% of the total sum. This implies that the collaboration rates are either very high or high for less than half of the Mono repository projects. When we also consider that all the projects were randomly collected from the Github platform and each has a unique background, parameter, and technology stack, it can be stated that software teams using a multi-repository structure have a significantly higher collaboration rate than those using a mono-repository structure.

Regarding the examination, first, it is possible to identify a specific trend. The results of the comparison of productivity and collaboration for Mono repository projects are presented in Figure 6.2. There are two categories of productivity and four categories of collaboration. Projects with an exceptionally high level of collaboration are classified as the initial category of collaboration. This category encompasses nearly 16% of Mono repository projects with high productivity, while only 2.4% of low productive ones do. This means that only a small fraction of low productivity projects has a very high collaboration rate. The second category of the chart demonstrates that, once again, high productive projects have a significantly higher percentage share than low-productive ones in the high collaboration rate. It is apparent that over half of the high productive projects have either a very high or high collaboration rate after combining these two categories. Although this value is only 33% for low-productive projects, an examination of these two categories and the percentage values reveals a tangible correlation between productivity and collaboration in Mono repository projects. The is also supported by the percentage values of high and low productive projects in the following two categories of collaboration. Low-productive projects have significantly higher percentage rates than high productive ones. In the fourth category, which denotes the lowest level of collaboration, low productive projects possess three times the percentage of high productive projects.

Figure 6.3 provides the same sort of results, but for Multi repository projects. Here, some of the results were different from the previous Mono repository case. First of all, again, the first category of collaboration, which represents the highest level, had a high productivity. For Multi repository projects, the values of high and low productive projects are nearly identical in the second category, which represents high collaboration rates. The primary reason for this may be the preceding inquiry. It was found that the majority of Multi repository projects have high levels of collaboration. Consequently, the productivity of these projects is not significantly correlated with collaboration, as nearly all of them have high levels of collaboration. However, this does not mean that a relationship between productivity and collaboration does not exist. The third and fourth categories of the chart demonstrate that the percentage share of low productive projects increases concurrently with a decrease in the collaboration rate.

Overall, it is evident that the correlation between the productivity of Multi repository projects and their collaboration rate is not as robust as in the Mono repository projects. However, it may still contribute to the project's productivity.

6.9 Concluding Remarks

Here, we concentrated on to pies like the optimal team size for highly productive projects and the collaboration of software teams. First, we introduced a novel method for evaluating the collaboration rate of software teams. This method differs from others by its mathematical nature and simplicity; however, it also prioritises the workload of developers in its calculations. One of the primary benefits of this method is that it calculates the collaboration at a significantly faster rate than the majority of methods and requires only one argument from GitHub repositories to accomplish this.

I presented a collection of measurements that demonstrate collaboration among various project parameters, including the repository structure, productivity level, and collaboration ratio, following the introduction of a collaboration calculation approach. By utilising all of these things, it is possible to provide a technical definition for software team collaboration and to propose an advice system for the data provided concerning the optimal team size based on the project's parameters.

The author of this thesis is responsible for the following contributions presented in this chapter:

IV / 1. I formulated mathematical method for the calculation of the collaboration rate of developer team collaboration. This method uses parameters provided by the GitHub platform. One of the main advantages of this method is that one can represent collaboration by a single numerical value.

IV / 2. I presented an advice system in order to provide developer suggestions about an optimal number of developers based on the project parameters. This new method uses the productivity level and several other parameters of the project. In addition to these, I implemented the results of my previous research to make the work of this method much more accurate.

Chapter 7

Conclusions

7.1 Results

The results are presented in four categories in my dissertation. The initial group concentrates on respiratory structures, with a particular emphasis on the two most prevalent types: Mono and Multi repository structures. The research conducted on both of these repository types is of significant value to this field, as they are very popular among developers. The primary challenge during this phase was the identification of an impartial database of Mono and Multi repository projects that could be employed. It became apparent after conducting a thorough investigation that, despite the existence of numerous online databases for various types of projects, there weren't any far specific types of projects. This is why we developed our own algorithms and models for the identification and collection of these types of projects. First of all, we developed our own methodology for identifying the two primary components of Multi repository projects; that is the frontend and backend repositories. The file structure of these repositories serves as the foundation of our methodology. A random forest classifier was employed to train a machine learning model for the identification process. The model demonstrated exceptional success, achieving an accuracy score of nearly 90%. It was extensively employed in the subsequent phases of our research work. We developed one of our primary algorithms for the later stage of our research. This algorithm offers a novel approach to the identification and collection of Mono and Multi repository projects. This algorithm is different and represents one of our most significant accomplishments. It simplifies the process of identifying and collecting projects, and it can be customised to meet the specific requirements of a project. When an extensive number of projects are gathered, it is still feasible to analyse them using Mono and Multi repository projects. These analyses allowed us to provide new technical definitions for each type of repository structure. Ultimately, we developed a heuristic approach for the identification of Multi repository management tools (MRMTs). This method is also novel, and it can facilitate the analysis of MRMTs done by other researchers and developers, thereby helping them to make critical project management decisions.

The second thesis group focused on branching strategies, which are one of the primary components of project management. Although some research in this area has been done, none of addressed the characteristics of this topic or analysed the extent to which they are related to other aspects of the project development process. Our primary contribution was once again to adapt a heuristic approach to the identification of branching strategies. This method, which is based on the structure type of branching strategies, has allowed us to identify the three primary branching strategies. Those strategies were selected due to their widespread popularity. It is also possible to customise this novel method to meet the specific requirements of the research or analysis. Our

database was enhanced with branching strategies, and we conducted an analysis of the relationship between repository structures and branching strategies.

The third thesis group explores the subject of software development productivity. Productivity analysis was a critical component of our investigation into the more extensive relationship between repository structure and software development. We established a baseline for our research by employing one of the most widely recognised and effective productivity measurement methods. This methodology enabled the execution of numerous analyses on parameters such as the size of the software team, the duration of the development cycle, the number of commits, the structure of the repository, the branching strategy, and so forth. These analyses provided us with significant findings and allowed us to develop our own ML model for evaluating the productivity of the software development process. This productivity measurement method produced a 85% accuracy rate and numerous advantages over the current methods. Initially, it encompasses all facets of the software development process and the project's properties, and it provides significantly more objective results than other methods. And this method, similar to our other methods, is flexible and can be implemented in a variety of scenarios. Furthermore, we employ an alternative methodology for the estimation of software development processes in conjunction with this novel methodology. There are numerous methods available for this purpose, but only a few of them provide precise results or figures. The majority of these methods are based on expert opinion; however, our machine learning model uses our previously acquired data to accurately predict the development period with a mere 3.4-month error margin.

The fourth thesis point examines software team collaboration, which is an additional critical component of the software development process. Firstly, we conducted an analysis of numerous existing methodologies and approaches for the calculation of the collaboration rate. In the majority of instances, researchers employed either straightforward methods, such as calculating the commit count of each developer, or intricate ones, such as analysing hundreds of thousands of commit messages and issues. We sought to develop a method that was both efficient and straightforward, and as a result, we devised our own mathematical approach for the collaboration rate calculator. This method is definitely applicable to projects that have been gathered on the Github platform. We calculated the software team collaboration rate by utilising the unique "contribution" parameter of each developer on the project. This parameter is determined by Github, which considers all forms of developer contributions. The primary foundation of our approach is the workshare of developers in relation to the entire team. One of the primary benefits of our methodology is that we got a mathematical value for the collaboration rate, which can be subsequently employed to quantify the rate. Overall, a unique advice system was developed by integrating the majority of the findings obtained in this thesis. The primary objective of this system is to provide users with the optimal value for the size of their team. These decisions are made by this system using the projects with the highest productivity and collaboration rate.

7.2 Future Work

Our study significantly advanced the area of two primary repository structures and their effect on the software development process, including productivity and collaboration. In order to conduct their own analyses and generate databases for Mono/Multi repository projects, other researchers may wish to apply our methodology. Three primary directions for future work are given below:

First of all, we plan to extend our algorithm and methods to identify an even bigger number of Mono and Multi repository projects. By doing this, we can expedite the expansion of our database and develop more effective tools for the collection of additional data. This work will also be beneficial to other researchers, as they will be able to utilise our database in the future to conduct their own analyses and calculations.

Secondly, we plan to enhance our productivity calculation methodology. It is a well known that the productivity calculation is a complex and challenging endeavour. Our objective is to develop a more comprehensive methodology that will incorporate an even greater number of parameters related to the software development process and allow a more precise calculation of productivity. It should also help in pursuing research in software architectures and productivity.

Lastly, we would like to emphasise the human element of repository structures. There has been some minor research conducted here; however, it would be far simpler for us to provide a comprehensive overview of this area with our newly developed calculation methodologies and the data that has been collected so far.

Bibliography

[1] Perry, Dewayne E., and Alexander L. Wolf. "Foundations for the study of software architecture." ACM SIGSOFT Software engineering notes 17.4 (1992): 40-52.

[2] Valipour, Mohammad Hadi, et al. "A brief survey of software architecture concepts and service oriented architecture." 2009 2nd IEEE International Conference on Computer Science and Information Technology. IEEE, 2009.

[3] L. Bass, P. Clements and R. Kazman. Software Architecture in Practice. Addison Wesley, 1999, ISBN 0-201-19930-0.

[4] Comparison of version-control software. Wikipedia. 2019. [Online] Available: https://en.wikipedia.org/wiki/Comparison_of_version-control_software

[5] Monorepo, Manyrepo, Metarepo. Burke Libbey 2019. [Online] Available: https://notes.burke.libbey.me/metarepo/ (Last accessed 6 August 2023)

[6] Rodney T. Ogawa and Betty Malen. "Towards rigor in reviews of multivocal literatures: Applying the exploratory case study method. Review of Educational Research", 61(3):265–286, 1991.

[7] Ciera Jaspan, Matthew Jorde, Andrea Knight, Caitlin Sadowski, Edward K. Smith, Collin Winter, Emerson Murphy-Hill Advantages and Disadvantages of a Monolithic repository: A Case Study at Google, 2018 ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice, May 27-June 3, 2018, Sweden.

[8] Mark Florisson and Alan Mycroft. Towards a Theory of Packages. University of Cambridge. 2015

[9] Terraform Mono Repo vs. Multi Repo: The Great Debate. Tracy Holmes. HashiCorp. Jan 28, 2021. [Online] Available: https://www.hashicorp.com/blog/terraform-mono-repo-vs-multi-repo-the-great-debate

[10] F. Kuhl, R. Weatherly, J. Dahmann. Creating Computer Simulation Systems: An Introduction to High Level Architecture. Prentice Hall, 2000.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal. Pattern Oriented Software Architecture: A System of Patterns. John Wiley & Sons, 1996.

[12] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. Software Architecture Documentation in Practice, Addison Wesley Longman, 2001.

[13] R. Allen and D. Garlan, "Formalizing architectural connection," in ICSE '94: Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 1994, pp. 71–80.

[14] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In Proceedings of SIGSOFT'93: Foundations of Software Engineering. ACM Press, December 1993.

[15] J. A. Stafford, D. J. Richardson, and A. L. Wolf, "Aladdin: A tool for architecture-level dependence analysis of software," University of Colorado at Boulder, Technical Report CU-CS-858-98, April 1998.

[16] B. Boehm, P. Bose, E. Horowitz and M. J. Lee. Software requirements negotiation and renegotiation aids: A theory-W based spiral approach. In Proc of the 17th International Conference on Software Engineering, 1994.

[17] P. Clements, L. Bass, R. Kazman and G. Abowd. Predicting software quality by architecture-level evaluation. In Proceedings of the Fifth International Conference on Software Quality, Austin, Texas, Oct, 1995.

[18] D. Krafzig, K. Banke, D. Slama, Enterprise SOA: Service-oriented architecture best practices, in: Enterprise SOA: Service-Oriented Architecture Best Practices, Prentice Hall PTR, 2005.

[19] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard, Web services architecture w3c working group note 11, 2004.

[20] C.M. MacKenzie, K. Laskey, P.F. Brown, R. Metz, Reference model for service oriented architecture 1.0. OASIS open, 2006.

[21] Kumar, Ashish. "Software architecture styles: A survey." International Journal of Computer Applications 87.9 (2014).

[22] Github Rest API Documentation, Github 2022, [Online] Available: https://docs.github.com/en/rest?apiVersion=2022-11-28

[23] Github, Wikipedia 2023. [Online] Available: https://en.wikipedia.org/wiki/GitHub#cite_note-12

[24] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, Daniela Damian, "The promises and perils of mining GitHub", MSR 2014: Proceedings

of the 11th Working Conference on Mining Software Repositories, May 2014 Pages 92–101, https://doi.org/10.1145/2597073.2597074

[25] Valerio Cosentino, Javier L. Cánovas Izquierdo, Jordi Cabot, “A Systematic Mapping Study of Software Development With GitHub”, IEEE, Volume 5, https://ieeexplore.ieee.org/abstract/document/7887704.

[26] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter, and Emerson Murphy- Hill, “Advantages and disadvantages of a Monolithic repository: A case study at Google,” in Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, Sweden, May 27-June 3, 2018, pp. 225–234

[27] Guardian/frontend. 2024. [Online] Available at: https://github.com/guardian/frontend

[28] Github 2022, [Online] Available: https://docs.github.com/en/authentication/keeping-your- account-and-datasecure/creating-a-personal-access-token.

[29] Shakikhanli, Ulvi, and Vilmos Bilicki. "Machine learning model for identification of frontend and backend repositories in Github." Multidisciplinary Science Journal 5 (2023).

[30] GHTorrent Tutorial. 2024. [Online] Available at: https://ghtorrent.github.io/tutorial/

[31] GH Archive 2024. [Online] Available at: https://www.gharchive.org

[32] Shakikhanli, Ulvi and Vilmos Bilicki, Comparison between mono and multi repository structures. Pollack Periodica, vol. 17, no. 3, pp. 7-12, 2022.

[33] Shakikhanli, Ulvi, and Vilmos Bilicki. "Multi repository Management Tools." The Journal of CIEES 2.2 (2022): 13-18.

[34] D. Arve. Branching strategies with distributed version control in agile projects. pages 1–12, 2010.

[35] GitKraken 2024. [Online] Available at: https://www.gitkraken.com/learn/git/best-practices/git-branch-strategy

[36] “Mining File Histories: Should We Consider Branches?”, Vladimir Kovalenko, Fabio Palomba, Alberto Bacchelli. https://fpalomba.github.io/pdf/Conferencs/C35.pdf

[37] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and isolated development with branches. In International Conference on Fundamental Approaches to Software Engineering, pages 316–331. Springer, 2012.

[38] Zou, Weiqin, et al. "Branch use in practice: A large-scale empirical study of 2,923 projects on github." 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2019.

[39] E. Kalliamvakou, D. Damian, L. Singer, and D. M. German. The code-centric collaboration perspective: Evidence from github. Technical Report DCS-352-IR, University of Victoria, February 2014.

[40] Spinellis, Diomidis. "Version control systems." IEEE software 22.5 (2005): 108-109.

[41] Zolkifli, Nazatul Nurlisa, Amir Ngah, and Aziz Deraman. "Version control system: A review." Procedia Computer Science 135 (2018): 408-415.

[42] Otte, Stefan. "Version control systems." Computer systems and telematics (2009): 11-13.

[43] B. Berliner. CVS II: Parallelizing software development. In Proc. USENIX Winter 1990 Technical Conference, pages 341–352, Berkeley, USA, 1990. USENIX Association.

[44] U. Shakikhanli, V. Bilicki, Optimizing Branching Strategies in Mono-and Multi repository Environments: A Comprehensive Analysis. Computer Assisted Methods in Engineering and Science, vol. 31, no. 1, pp. 81-111, 2024.

[45] I. C. Clatworthy. Distributed version control: Why and how. In Proc. Open Source Development Conf. (OSDC), 2007.

[46] De Alwis, Brian, and Jonathan Sillito. "Why are software projects moving from centralized to decentralized version control systems?." 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering. IEEE, 2009.

[47] Koc, Ali, and Abdullah Uz Tansel. "A survey of version control systems." ICEME 2011 (2011).

[48] Renaming the default branch from master, Available at: https://github.com/github/renaming

[49] A. MacCormack, C. Kemerer, M. Cusumano, and B. Crandall, “Trade-Offs between Productivity and Quality in Selecting Software Development Practices,” IEEE Software, pp. 78-79, Sept./Oct. 2003.

[50] Kitchenham, Barbara, and Emilia Mendes. "Software productivity measurement using multiple size measures." IEEE Transactions on Software Engineering 30.12 (2004): 1023-1035.

[51] LooksGTM, https://github.blog/2022-08-15-the-next-step-for-lgtm-com-github-code-scanning/, Available 25/08/2023.

[52] Hélie, Jean, Ian Wright, and Albert Ziegler. "Measuring software development productivity: A machine learning approach." Proceedings of the Conference on Machine Learning for Programming Workshop, Affiliated with FLoC, Oxford, UK. 2018.

[53] J. Gamalielsson and B.Lundell. Long-term sustainability of open source software communities beyond a fork: A case study of libreoffice. In IFIP International Conference on Open Source Systems, pages 29–47. Springer, 2012.

[54] B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov. Gender and tenure diversity in github teams. In Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, pages 3789–3798. ACM, 2015.

[55] Choudhary, Samridhi Shree, et al. "Modeling coordination and productivity in open-source GitHub projects." School of Computer Science, Carnegie Mellon University (2018).

[56] Walter L Ruzzo and Martin Tompa. A linear time algorithm for finding all maximal scoring subsequences. In ISMB, volume 99, pages 234–241, 1999.

[57] Jon Kleinberg. Bursty and hierarchical structure in streams. Data Mining and Knowledge Discovery, 7(4):373–397, 2003.

[58] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE, 77(2):257–286, 1989.

[59] Doug Beeferman, Adam Berger, and John Lafferty. Statistical models for text segmentation. Machine learning, 34(1):177–210, 1999.

[60] Marti A Hearst. Multi paragraph segmentation of expository text. In Proceedings of the 32nd annual meeting of the Association for Computational Linguistics, pages 9–16. Association for Computational Linguistics, 1994.

[61] Marcelo Cataldo and James D Herbsleb. Coordination breakdowns and their impact on development productivity and software failures. IEEE Transactions on Software Engineering, 39(3):343–360, 2013.

[62] Marcelo Cataldo, Patrick A Wagstrom, James D Herbsleb, and Kathleen M Carley. Identification of coordination requirements: implications for the design of collaboration

and awareness tools. In Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work, pages 353–362. ACM, 2006.

[63] Kalliamvakou, Eirini, et al. "An in-depth study of the promises and perils of mining GitHub." *Empirical Software Engineering* 21 (2016): 2035-2071.

[64] Coelho, Jailton, et al. "Identifying unmaintained projects in github." Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. 2018.

[65] Guzman, Emitza, David Azócar, and Yang Li. "Sentiment analysis of commit comments in GitHub: an empirical study." Proceedings of the 11th working conference on mining software repositories. 2014.

[66] Barnett, Jacob G., et al. "The relationship between commit message detail and defect proneness in java projects on github." Proceedings of the 13th International Conference on Mining Software Repositories. 2016.

[67] Michaud, Heather M., et al. "Recovering commit branch of origin from github repositories." *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016.

[68] Montalvillo, Leticia, and Oscar Díaz. "Tuning GitHub for SPL development: branching models & repository operations for product engineers." *Proceedings of the 19th International Conference on Software Product Line*. 2015.

[69] Ertel, Chris. "Developing with Git and Github." *Introduction to Scientific and Technical Computing* (2016): 53-68.

[70] Pipinellis, Achilleas. *GitHub essentials*. Vol. 2. Packt Publishing, 2015.

[71] Rashid, Ekbal, and Mohan Prakash. "An empirical analysis of inferences from commit, fork, and branch rates of top GitHub projects." *International Journal of Open Source Software and Processes (IJOSSP)* 13.1 (2022): 1-16.

[72] García, Salvador, et al. "Dealing with noisy data." Data preprocessing in data mining (2015): 107-145.

[73] Jørgensen, Magne. "A review of studies on expert estimation of software development effort." Journal of Systems and Software 70.1-2 (2004): 37-60.

[74] Idri, Ali, Fatima azzahra Amazal, and Alain Abran. "Analogy-based software development effort estimation: A systematic mapping and review." Information and Software Technology 58 (2015): 206-230.

[75] Shepperd, Martin, and Chris Schofield. "Estimating software project effort using analogies." IEEE Transactions on software engineering 23.11 (1997): 736-743.

[76] Khoshgoftaar, Taghi M., and Naeem Seliya. "Analogy-based practical classification rules for software quality estimation." Empirical Software Engineering 8 (2003): 325-350.

[77] Amazal, Fatima Azzahra, Ali Idri, and Alain Abran. "Software development effort estimation using classical and fuzzy analogy: a cross-validation comparative study." International Journal of Computational Intelligence and Applications 13.03 (2014): 1450013.

[78] Usman, Muhammad, et al. "Effort estimation in agile software development: a systematic literature review." Proceedings of the 10th international conference on predictive models in software engineering. 2014.

[79] Ziauddin, Shahid Kamal Tipu, and Shahrukh Zia. "An effort estimation model for agile software development." Advances in computer science and its applications (ACSA) 2.1 (2012): 314-324.

[80] Usman, Muhammad, Emilia Mendes, and Jürgen Börstler. "Effort estimation in agile software development: a survey on the state of the practice." Proceedings of the 19th international conference on Evaluation and Assessment in Software Engineering. 2015.

[81] Fernández-Diego, Marta, et al. "An update on effort estimation in agile software development: A systematic literature review." IEEE Access 8 (2020): 166768-166800.

[82] "Random Forest Classifier", Scikit-Learn, Online 2023 https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[83] Curran, Connie L., and Carolyn A. Smeltzer. "Collaboration not competition: a model for nursing continuing education." Journal of Nursing Education 20.6 (1981): 24-29.

[84] Whitehead, Jim. "Collaboration in software engineering: A roadmap." Future of Software Engineering (FOSE'07). IEEE, 2007.

[85] T. W. Malone and K. Crowston, "The Interdisciplinary Study of Coordination," in ACM Computing Surveys (CSUR), vol 26, no 1, pp. 87-119, 1994.

[86] Brindescu, Caius, et al. "How do centralized and distributed version control systems impact software changes?" *Proceedings of the 36th international conference on Software Engineering*. 2014.

[87] Marion, Tucker J., and Sebastian K. Fixson. "The transformation of the innovation process: How digital tools are changing work, collaboration, and organizations in new product development." Journal of Product Innovation Management 38.1 (2021): 192-215.

[88] *Basecamp*, Available at: https://basecamp.com, Accessed February 05, 2024.

[89] *Asana*, Available at: https://asana.com, Accessed February 05, 2024.

[90] *Teamwork*, Available at: https://www.teamwork.com, Accessed February 05, 2024.

[91] Marion, Tucker, and Sebastian Fixson. "The influence of collaborative information technology tool usage on npd." Proceedings of the Design Society: International Conference on Engineering Design. Vol. 1. No. 1. Cambridge University Press, 2019.

[92] Marques, Joana Ferreira, and Jorge Bernardino. "Evaluation of Asana, Odoo, and ProjectLibre Project Management Tools using the OSSpal Methodology." KEOD. 2019.

[93] Brown, Judith M., Gitte Lindgaard, and Robert Biddle. "Collaborative events and shared artifacts: Agile interaction designers and developers working toward common aims." *2011 Agile Conference*. IEEE, 2011.

[94] Calefato, Fabio, et al. "A case study on tool support for collaboration in agile development." *Proceedings of the 15th International Conference on Global Software Engineering*. 2020.

[95] Jones, Alexander, and Volker Thomas. "Determinants for successful agile collaboration between UX designers and software developers in a complex organization." *International Journal of Human–Computer Interaction* 35.20 (2019): 1914-1935.

[96] Hoda, Rashina, James Noble, and Stuart Marshall. "The impact of inadequate customer collaboration on self-organizing Agile teams." *Information and software technology* 53.5 (2011): 521-534.

[97] Al-Saqqa, Samar, Samer Sawalha, and Hiba AbdelNabi. "Agile software development: Methodologies and trends." International Journal of İnteractive Mobile Technologies 14.11 (2020).

[98] Wolf, Timo, et al. "Mining task-based social networks to explore collaboration in software teams." IEEE software 26.1 (2008): 58-66.

[99] Bettenburg, Nicolas. "Mining development repositories to study the impact of collaboration on software systems." Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011.

[100] Zöller, Nicolas, Jonathan H. Morgan, and Tobias Schröder. "A topology of groups: What GitHub can tell us about online collaboration." Technological Forecasting and Social Change 161 (2020): 120291.

[101] Saadat, Samaneh, et al. "Analyzing the productivity of GitHub teams based on formation phase activity." 2020 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT). IEEE, 2020.

[102] Constantino, Kattiana, et al. "Perceptions of open-source software developers on collaborations: An interview and survey study." Journal of Software: Evolution and Process 35.5 (2023): e2393.

[103] Constantino, Kattiana, et al. "Understanding collaborative software development: An interview study." Proceedings of the 15th international conference on global software engineering. 2020.

[104] Scholtes, Ingo, Pavlin Mavrodiev, and Frank Schweitzer. "From Aristotle to Ringelmann: a large-scale analysis of team productivity and coordination in Open-Source Software projects." *Empirical Software Engineering* 21.2 (2016): 642-683.

[105] Robillard, Pierre N., and Martin P. Robillard. "Types of collaborative work in software engineering." *Journal of Systems and Software* 53.3 (2000): 219-224.

[106] Kusumasari, Tien Fabrianti, et al. "Collaboration model of software development." *Proceedings of the 2011 International Conference on Electrical Engineering and Informatics*. IEEE, 2011.

[107] Ulvi Shakikhanli, Vilmos Bilicki, Repository Structures: Impact on Collaboration and Productivity. Pollack Periodica, 2024.

[108] Abdi, Hervé. "Coefficient of variation." Encyclopedia of research design 1.5 (2010).

[109] *Github commits,* Available at: https://docs.github.com/en/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/about-commits, Accessed November 15, 2023

[110] Thesing, Theo, Carsten Feldmann, and Martin Burchardt. "Agile versus waterfall project management: decision model for selecting the appropriate approach to a project." Procedia Computer Science 181 (2021): 746-756.

[111] Ciric Lalic, Danijela, et al. "How does a project management approach impact project success? From traditional to agile." International Journal of Managing Projects in Business 15.3 (2022): 494-521.

[112] Tam, Carlos, et al. "The factors influencing the success of on-going agile software development projects." International Journal of Project Management 38.3 (2020): 165-176.

[113] A. MacCormack, C. Kemerer, M. Cusumano, and B. Crandall, "Trade-Offs between Productivity and Quality in Selecting Software Development Practices," IEEE Software, pp. 78-79, Sept./Oct. 2003.

[114] J. Gamalielsson and Blundell. Long-term sustainability of open source software communities beyond a fork: A case study of libreoffice. In IFIP International Conference on Open Source Systems, pages 29–47. Springer, 2012.

Summary

This PhD thesis concentrates on the relationship between two main repository structure types and project parameters, that include development productivity, software team collaboration, development period, and the developer team size. In order to get the maximum number of general results, a specialised database was constructed, which includes over 50,000 mono- and multi-repository projects. The primary objective of this thesis is to demonstrate the existence of a correlation between the structure of the repository and the project development process and, where such a correlation exists, to demonstrate how it can influence the overall development process. We developed our own special algorithms for the identification and collection of Mono and Multi repository projects from the Github platform, as it is a narrowly analysed field of software development. And a novel machine learning methodology was developed to estimate development productivity and devise a mathematical method for calculating software team collaboration.

Mono and Multi repository structures

Chapter 2 of the thesis focuses on the first phase of our research on Mono and Multi repository structures. Until now, there have been several research and academic articles that analysed the properties of Mono and Multi repository structures. However, most of these analyses were conducted either on a local scale, hence lost objectivity, or done from a narrow perspective, that did not allow any general ideas or understanding. In this study, I solved both issues by choosing my research project from Github, which is the biggest project repository of all time. According to the current data, Github now hosts more than 420 million repositories, according to current estimates. Due to the huge number of repositories available, we chose them as a source for our projects. The Github platform provides a Github API, which may be useful for collecting information about repositories, but also, this API doesn't provide any information about the structure or type of repository. Because of this, we decide to develop our own algorithm and approach for the identification and collection of Mono and Multi repository projects on the Github platform.

Branching strategies in Mono and Multi repository projects

This dissertation examined three primary branching strategies; namely GitFlow, Github Flow, and Trunk-based. The main justification for choosing these three is their widespread use among developers, which our own analyses of thousands of projects confirmed. The identification work method we developed is based on the structure of these branching strategies. In other words, we employ a number of branches and their attributes. For a better explanation of our approach, the following steps are stated:

1. Remove all the branches created automatically by bots or MRMTs.
2. Record the total count of the branches and their names for identification.
3. If a project has only one branch which shares the names like *main, master* or *product*, then this project uses a Trunk-based approach.
4. If a project has more than one branch then and some of them have labels like *dev* and *development* etc, then this project most probably uses the GitFlow approach.
5. If a project has more than one branch but it doesn't have any development branches and it has several bug fix or error fix branches then this project most probably uses the Github Flow branching strategy.

It should be added that this approach assumes that all the branches have been named correctly and developers used their branches based on the main rules of the branching strategy.

Productivity of software development

This chapter focuses on our findings concerning software development productivity. The primary concern was the calculation of productivity, and we examined a variety of methods for this. We developed our own methodology that evaluated the project's productivity by utilising a variety of parameters. We note that this productivity value is correlated with other project parameters, such as the development period, repository structure, and branching strategy. This connection led us to propose a novel machine learning approach for productivity calculation that did not require any mathematical calculations, as the above-mentioned approach does. The model was trained using the following features of repositories: *the number of commits, the size of the developer team, the size of the project, the number of issues, the number of events,* and *the number of pull requests*.

Next, we developed novel methods for estimating the development period. Similar to the preceding methodology, this novel methodology employed a machine learning algorithm to compute the development period in months. At present, our method has an average error margin of 3.4 months, which is among the most impressive results in this area.

Collaboration of Software Developer Team

The final chapter of this thesis concentrated on the collaboration of the software developer team during the development process and the relationship between this collaboration and the project's above-mentioned parameters, including productivity, repository structure, and branching strategy.

The novel method of calculating software team collaboration using a mathematical formulation is one of the primary themes here. We carried out an analysis of numerous methods for the calculation. It is clear that the majority of the methodologies are compute of the aggregate workload of each developer. This procedure has been implemented by computing the quantity of pull requests, commits, and other relevant metrics. Based on this, we developed a mathematically rigorous and objective methodology that is comparable.

A contribution is a value that is assigned to each contributor to the Github repository. This value is represented by an integer value. This number is essential for determining the developer's workload during the development process and represents their overall contribution. We did some mathematical calculations to find the percentage share of each developer's work, which resulted in the classification of our projects into the following categories: very high collaboration, high collaboration, medium collaboration, and low collaboration.

Contributions of the thesis

In the first thesis group, the contributions are related to the publications “Comparison between mono and multi repository structures”, “Machine learning model for identification of frontend and backend repositories in Github” and “Multi repository Management tools”. Detailed discussion can be found in Chapter 3.

I/1. A new definition for Mono and Multi repository projects was proposed which encompasses both their characteristics and structures.

I/2. Creating a new ML method for the identification of frontend and backend repositories on the Github platform. This method is especially useful for quick identification of both types of repositories.

I/3. A new algorithm for the identification of Mono and multi repository projects was proposed. This unique approach for the identification and collection of projects belonging to both repository structures can be used for all types of projects, and it is possible to adopt it to other projects as well.

I/4. A heuristic approach for the identification of different multi repository management tools was applied. Developers and researchers can use this approach to assist their work and research.

In the second thesis group, the contributions are related to the publication “Optimizing Branching Strategies in Mono and Multi repository Environments: A Comprehensive Analysis”. Detailed discussion can be found in Chapter 4.

II/1. We proposed a heuristic approach for the identification of branching strategies used by the project during the development phase. Branching strategies are essential parts of the project management process and this way they can be identified much faster than any other approach.

II/2. Conducting research and an analysis into the connection between branching strategies, repository structure and productivity.

In the third thesis group, the contributions are related to the publication "Analyzing Branching Strategies for Project Productivity: Identifying the Preferred Approach". Detailed discussion can be found in Chapter 5.

III/1. A new machine learning method was proposed for the assessment of productivity of the software development process. This approach is based on the correlation between productivity and several parameters of the project and development process.

III/2. A machine learning method was proposed for the estimation of the development period.

In the fourth thesis group, the contributions are related to the publication "Repository Structures: Impact on Collaboration and Productivity". Detailed discussion can be found in Chapter 6.

IV/1. Use of a mathematical method for the calculation of the software team collaboration rate.

IV/2. A special advice system was created for the estimating the number of developers required for a project based on the given parameters.

Összefoglaló

A doktori értekezés témája a két fő repository-struktúra típus, illetve ezek kapcsolatának vizsgálata a szoftverfejlesztési projektek olyan paramétereivel, mint a fejlesztői produktivitás, a szoftverfejlesztői csapaton belüli kollaboráció mértéke, a fejlesztési időszak időtartama, a fejlesztőcsapat mérete stb. Az eredmények kellő generalizációs mértékének elérése érdekében egy speciális projekt adatbázis jött létre, amely több mint 50 000 mono és multi repository projektet tartalmaz. A disszertáció fő célja annak bemutatása, hogy milyen kapcsolat van a repository struktúra és a fejlesztés folyamata között, és ha léteznek ilyen kapcsolatok, hogyan befolyásolhatják a fejlesztési folyamat egészét. Mivel a szoftverfejlesztés szűken kevésbé területéről van szó, saját egyedi algoritmusokat hoztunk létre a Mono és Multi repository projektek azonosítására és begyűjtésére a GitHub platformról. Ezen kívül egy új gépi tanulási megközelítést dolgoztam ki a fejlesztési folyamat produktivitásának becslésére, valamint egy matematikai módszert a fejlesztői csapatok együttműködési mértékének kiszámítására.

Mono és Multi repository struktúrák

A disszertáció 2. fejezete a Mono és Multi repository struktúrák területén végzett kutatásom első szakaszára fókuszál. Korábban már számos kutatói és tudományos munka elemezte a Mono/Multi repository struktúrák tulajdonságait. Az elemzések többsége azonban vagy limitált adatforrásokból dolgozott, elveszítve ezáltal az objektivitás és generalizálhatóság lehetőségét, vagy olyan korlátozott fókusz mentén készült, amely nem adott teret általános elméletek tesztelésére és analízisekre. Kutatásom során a két problémakört egységesen úgy kíséreltem megoldani, hogy a vizsgálatokhoz felhasznált projekteket Githubról gyűjtöttem be, mely az egyik jelenlegi legnagyobb publikus adatbázis forráskódelemzéshez. A jelenlegi adatok szerint a Github több mint 420 millió repositoryt tárol. A Github platform biztosít egy standardizált API-t is, amely hasznos lehet ugyan a repositorykkal kapcsolatos információk kigyűjtésében, de sajnos a repository szerkezeti típusáról már nem ad információt. Emiatt fejlesztettük ki saját algoritmusunkat és megközelítésünket a Mono/Multi repository projektek azonosítására és összegyűjtésére a Githubról.

Branch kezelési stratégiák Mono és Multi repository projektekben

Három fő branch kezelési stratégiát elemeztem a disszertáció következő részében, a GitFlow-t, a Github Flow-t és a Trunk alapút. A választásuk elsődleges oka a fejlesztők körében elterjedt kimagasló népszerűségük, melyet gigavolt a több ezer projekten végzett saját elemzésünk

is. A repository kategorizálásra fejlesztett módszerünk alapját lényegében ezeknek a branch kezelési stratégiáknak a felépítését képezték – a branchek számát és jellemzőit használtuk az elemzések során. Módszerünk tömören az alábbi lépésekből áll:

1. Eltávolítjuk a botok, illetve MRMT-k által automatikusan létrehozott összes branchet.
2. Feljegyezzük az ágak számát és nevét a későbbi követhetőség érdekében.
3. Ha egy projektnek csak 1 branche van, olyan névvel, mint például main, master, product stb., akkor ez a projekt Trunk alapú stratégiát használ.
4. Ha egy projektnek több branche van, és ezek közül néhányat devnek, developmentnek stb. neveznek, akkor ez a projekt valószínűleg a GitFlow stratégiát használja.
5. Ha egy projektnek több branche van, de ezek között nincs egyértelmű fejlesztési, ellenben több hibajavításra használt ága van, akkor az a projekt valószínűleg Github Flow elágazási stratégiát használ.

Fontos, hogy ez a módszer feltételezi, hogy az összes branchet helyesen nevezték el, és a fejlesztők az adott stratégia elveinek megfelelően használták őket.

Szoftverfejlesztés produktivitása

A következő fejezet a szoftverfejlesztés produktivitásával kapcsolatos megállapításainkra összpontosít. A fő kérdés maga a produktivitás kiszámítása volt, melynek érdekében többféle létező megközelítést elemeztünk, mielőtt megkezdtük a kigyűjtött repositoryk feldolgozását. Saját elvünk kidolgozása során megfigyeltünk különböző összefüggéseket a produktivitás értéke és a projektek olyan paraméterei között, mint például a repository struktúra, az branching stratégia, a fejlesztési időtartam, stb.. A megfigyelt korrelációknak köszönhetően egy új gépi tanulási megközelítést javasoltunk a produktivitás kiszámítására, anélkül, hogy már létező módszertanok által előírt matematikai számításokat végeznének. A modellt a repositoryk következő jellemzői alapján tanítottuk: Commitok száma, fejlesztői csapat mérete, projekt mérete, issuek száma, események száma, pull requestek száma stb.

Emellett új megközelítéseket is alkottunk a fejlesztési időszak becslésére. A fentihez hasonlóan ez a módszertan is gépi tanulási algoritmust használ, és a hónapok alapján számítja ki a fejlesztési időszakot. Jelenleg a módszerünk átlagos hibahatára 3,4 hónap, mely a szakirodalommal összevetve az egyik legjobb eredménynek bizonyult ezen a területen.

Szoftverfejlesztői csapat együttműködése

Ez az utolsó fejezet a szoftverfejlesztő csapat együttműködésének vizsgálatára összpontosít a fejlesztési folyamat során, és arra, hogy mi a kapcsolat ezen együttműködés és a projekt korábban említett paraméterei között, mint például a repository szerkezet, az branch kezelési stratégia és a produktivitás.

A fejezet fókusza a szoftverfejlesztői csapatok együttműködési mértékének kiszámítása egy új, matematikai megközelítés szerint. A számítás kidolgozásához szintén több, már létező módszertant elemeztünk. Ezek alapján nyilvánvalónak tűnt, hogy a legtöbb megközelítés az egyes

fejlesztők teljes munkaterhelésének kiszámítására összpontosított, a fejlesztők által végzett branch- és repository műveletek, mint commitok, pull requestek száma alapján. Ezt figyelembe véve egy hasonló, ám sokkal objektívebb és matematikaibb eljárást dolgoztunk ki.

Egy Github repository minden közreműködőjé rendelkezik egy értékkel, amelyet contributionnek, azaz hozzájárulásnak neveznek, és ezt az értéket egész számmal reprezentálja a rendszer. Ez a szám a fejlesztő teljes hozzájárulását jelöli az adott projekthez, és kulcsfontosságú a fejlesztési folyamat során felmerülő munkaterhelés kiszámításához. Matematikai számításokat alkalmaztunk az egyes fejlesztők munkájának százalékos meghatározásához, és így projektjeinket a következő együttműködési, avagy kollaborációs kategóriákba soroltuk: Nagyon magas fokú együttműködés, Magas fokú együttműködés, Közepes fokú együttműködés és Alacsony fokú együttműködés.

Az értekezés eredményei

Az első téziscsoportban közölt eredményeim a “Comparison between mono and multi repository structures”, “Machine learning model for identification of frontend and backend repositories in Github” és “Multi repository Management tools” című publikációimban lettek leközölve. Részletes kifejtésük a disszertáció 2. fejezetében található.

I/1. Egy új definíciót javasoltam a Mono és Multi repository verziókezelésű projektek kategorizálására, amely figyelembe veszi azok főbb jellemzőit és felépítését is.

I/2. Új ML metódust dolgoztam ki a frontend és a backend azonosítására adattárak a Github platformon. Ez a módszer különösen fontos a repository típusok hatékony meghatározásához.

I/3. Egy új algoritmust mutattam be a Mono és a Multi repositoryt használó projektek azonosítására. A módszer segítségével lehetőség van a két típushoz tartozó repositoryk egyértelmű kategorizálására és begyűjtésére, valamint kiterjeszthető más kategorizálási elvekre is.

I/4. Heurisztikus megközelítést mutattam be a különböző Multi repositoryk azonosítására, illetve az ehhez szükséges, általam fejlesztett eszközöket. A fejlesztők és kutatók ezt mind a szoftverfejlesztés, mind a kutatások során használhatják.

A második téziscsoportban bemutatott eredményeim az „Optimizing Branching Strategies in Mono and Multi repository Environments: A Comprehensive Analysis” című publikációban jelentek meg. Részletes kifejtésük a 3. fejezetben található.

II/1. Heurisztikus megközelítést vázoltam fel a projekt kezelési során alkalmazott branch kezelési stratégiák kategorizálására. Ezek a stratégiák elengedhetetlen részei a projektmenedzsment folyamatnak, a módszerem segítségével pedig sokkal hatékonyabban azonosíthatóak, mint bármely korábbi megközelítéssel.

II/2. Kutatásokat, elemzéseket készítettem a branch kezelési stratégiák a repository típusok és a produktivitás kapcsolatáról.

A harmadik téziscsoportban bemutatott eredményeim az "Analyzing Branching Strategies for Project Productivity: Identifying the Preferred Approach" című publikációban kerültek közlésre. Részletes kifejtésük a 4. fejezetben található.

III/1. Egy új gépi tanulási módszert dolgoztam ki a szoftverfejlesztési folyamat produktivitásának becsléséhez. Ez a módszer a a produktivitás, a projekt és a fejlesztési folyamat számos paramétere közötti korrelációkon alapul.

III/2. Bemutattam egy gépi tanulási módszert a várható fejlesztési idő becslésére.

A negyedik téziscsoportban szereplő eredményeim a szoftvercsapaton belüli együttműködés területéhez kapcsolódnak. Részletes kifejtésük az 5. fejezetben található.

IV/1. Matematikai módszert dolgoztam ki a szoftverfejlesztői csapat együttműködési mértékének kiszámításához.

IV/2. A megfelelő becsléshez speciális tanácsadó rendszert fejlesztettem ki, mely a megadott paraméterek alapján felméri a hatékony fejlesztéshez szükséges szoftverfejlesztők számát.

Publications

[J1] **U. Shakikhanli**, V. Bilicki, Comparison between mono and multi repository structures. *Pollack Periodica*, vol. 17, no. 3, pp. 7-12, 2022.

[J2] **U. Shakikhanli**, V. Bilicki, Multi repository Management tools. *The journal of CIEES*, vol. 2, no. 2, pp. 13-18, 2022.

[J3] **U. Shakikhanli**, V. Bilicki, Optimizing Branching Strategies in Mono- and Multi repository Environments: A Comprehensive Analysis. *Computer Assisted Methods in Engineering and Science*, vol. 31, no. 1, pp. 81-111, 2024.

[J4] **U. Shakikhanli**, V. Bilicki, Analyzing Branching Strategies for Project Productivity: Identifying the Preferred Approach. *Journal of Electrical Systems*, 2024.

[J5] **U. Shakikhanli**, V. Bilicki, Repository Structures: Impact on Collaboration and Productivity. *Pollack Periodica*, 2024.

Full papers in conference proceedings

[C6] **U. Shakikhanli**, V. Bilicki, Machine learning model for identification of frontend and backend repositories in Github. *Multidisciplinary Science Journal*, vol. 5, 2023.

Acknowledgments

I would like to take this opportunity to thank everyone who helped me during my PhD studies. It has been a long, hard, nevertheless rewarding journey for me.

First of all, I wish to thank my supervisor, Professor Vilmos Bilicki. He not only helped me with my scholarly goals and to publish papers, but he also helped me become a real academic. With his guidance, I have improved over the years, and his good advice and wisdom will remain with me for the rest of my life.

Secondly, I would like to express my gratitude to the professors and academic staff of the University of Szeged. It has been an honour to study at this university, and their influence was decisive in my development as an academic researcher.

Last but not least, I want to thank my family and friends. They have supported me over the years, and I never felt alone whatever challenges came my way.

Finally, I would like to thank the light of my life. Her love and support have illuminated my darkest days. This journey would have been much harder, and maybe even impossible, without her.