Computer Assisted Methods in Engineering and Science
31(1): 81–111, 2024, doi: 10.24423/cames.2024.1372

Optimizing Branching Strategies in Mono- and Multi-Repository Environments: A Comprehensive Analysis

Ulvi SHAKIKHANLI*, Vilmos BILICKI

Doctoral School of Computer Science, Faculty of Science and Informatics, University of Szeged, H-6720, Szeged, Hungary; e-mail: bilickiv@inf.u-szeged.hu

* *Corresponding Author e-mail: ulvi@inf.u-szeged.hu*

There have been several studies on mono- and multi-repository structures and branching strategies. However, most of those studies focused on the basics of repository structures and used a small number of project samples. This paper uses data from more than 50 000 repositories collected from GitHub. The results indicate that: 1) mono-repository projects generally involve smaller teams, with the majority being handled by one or two developers, 2) multi-repository projects often require larger teams, typically consisting of three or more developers, 3) mono-repository projects are favored for shorter durations, with over half of the projects completed within six months, 4) multi-repository projects, on the other hand, have higher usage percentages in longer development periods, suggesting their suitability for more time-consuming endeavors. Examining branching strategies reveals that: 1) the trunk-based approach is commonly used in both mono- and multi-repository projects, 2) GitHub Flow has much wider usage in multi-repository projects rather than mono-repository.

These findings offer valuable insights for developers and project managers in selecting the appropriate repository structure and branching strategy based on project requirements. Understanding team dynamics, project complexity, and desired development periods aids in optimizing collaboration and achieving successful outcomes.

Keywords: mono-repository structure, multi-repository structure, branching strategy, Git Flow, GitHub Flow, trunk-based.

1. Introduction

The development process of projects has been a subject of discussion for decades. There are several important aspects of it with the repository structure being the most important one. We could not find any definition for "repository structure", but it can be broadly understood as a layout of different components within a project (for example: front-end and back-end repositories). In light of

this definition and usage experience, there are two main repository structures: mono- and multi-repository structures. Each structure possesses unique characteristics, advantages, and disadvantages. This paper is going to analyze such structures by discussing a usage of a unique database with more than 50 000 projects. The results presented in this paper provide an understanding of how repository structures can impact team size and which development stacks are used more commonly by each of these two structures.

Ever since modern software development practices became more popular, several challenges in need of solving have emerged. Among them are collaborations between large development teams and simultaneous task execution without any conflicts in the main source code. This became more problematic since the advent of distributed version control (DVS) systems widely used among developers and different companies. To solve it, DVC systems improved a solution called branching [1]. The adoption of branches helped teams to divide their development processes into stages, and combining or merging them only when specific tasks were completed. In this way, the development process becomes more flexible, and this method gained significant popularity in agile development systems. Almost all modern DVCs support the usage of branching. With the rising popularity of using branching, a new term has emerged: branching strategy. A branching strategy is a strategy used by developer teams to handle their various branches and their respective purposes. Overall, the use of branches can have different purposes, focusing on implementing new features, fixing bugs, preparing release version, and more. While there are several branching strategies, only three of them are widely used in Git [2] and this paper specifically concentrates on them.

In this paper, we aim to provide answers to several questions based on real-world data. The insights derived from these answers can assist software developers in project planning and choosing a suitable repository structure and branching strategy based on team size, preferred programming language and planned development period.

RQ 1. What is the relationship between team size and repository structure?

To the best of our knowledge, there is no detailed investigation on this topic. In this paper, we obtain some insights from the analysis of our database, revealing interesting results that can be crucial for team managers and lead developers during the planning stage of software development. Since there is no similar research in this field, these results become even more important.

RQ 2. What is the connection between development period and repository structure?

While the development period is often researched together with software development productivity [3] or agility and quality of the development process [4],

this paper introduces a totally novel approach and analyzes development pace from the perspective of repository structure. This adds an additional aspect to take in account during the software development planning process and helps in choosing the right repository structure.

RQ 3. Which branching strategy is preferred according to the repository structure?

As it will be discussed in the next section, branching strategies are commonly researched as features of version control systems (VCSs). Yet, there is a lack of academic research on branching strategies and their impact on software development. This paper provides a clear understanding of their usage rate over the years and popularity according to the repository structure by analyzing over 50 000 repositories from various projects. It is one of the main steps in understanding branching strategies and their real significance in software development.

RQ 4. Which branching strategy is preferred according to team size, programming language and development period?

This question intends to address other impacts of branching strategy on software development, such as team size and development period. Since it is highly important to choose the right branching strategy at the beginning of the development process, the answer to this question can a valuable guide during the decision-making process.

2. Related work

Unfortunately, we did not find any academic papers on the mono- and multi-repository structures. Thus, we will use the multivocal literature review (MLR) [5]. A mono-repository structure is a method of organizing source code where engineers have widespread access to the code, use a shared set of tools, and depend on a single set of common dependencies. This arrangement allows for standardization and easy access by employing a single, shared repository that contains the source code for all projects within an organization [6]. On the contrary, the multi-repository approach is quite different. Here, the project is stored in different repositories. This division can be determined by the development policy of the company or developer group, but according to common experience, the project is mostly divided into two main repositories: front-end and back-end repositories. Obviously, there can be several other types of repositories such as documentation repository, data repository, library/module repository and so on. The portion of a web application that users directly interact when using their web browser is called the front end. Its main tasks include presenting the user interface, handling user input, and communicating with the back end of the application. Conversely, a back-end repository is a type of code repository that

holds the codebase and data for the back end of a web application. The back end, which operates on a server, is responsible for managing and storing data, as well as executing logic and business processes.

Each of these structure types has its own advantages and disadvantages. The paper [7] compares both repository structures from different aspects such as visibility, security and functionality, showing the behavior of each structure through academic results. As shown in that paper, there are several trade-offs in choosing the repository structure and sometimes even large companies can change their policy and repository structure for their project back and forth [8]. The famous company Uber is a perfect example of this [9].

In addition to the repository structure, another important aspect of software development is VCS. This concept was introduced in 1972. Since then, there have been several papers written on this topic. Some, such as [10–12], discuss basic concepts such as what VCSs are and in which cases we can use them. Mainly, there are two types of VCSs: centralized (CVCSs) and distributed (DVCSs) VCSs. CVCSs being a much earlier approach, were extensively studied in papers such as [13–15]. Primarily, CVCSs exhibit centralization since they possess a singular, authoritative source repository. Every developer operates in relation to this repository by obtaining a checkout, which essentially captures a specific point-in-time snapshot from the repository [16]. These systems operate by storing all project files and versions on a central computer or server. Users can access specific files or entire repositories for their work. Upon completion, users must "push" their changes along with a commit message. Once these changes are pushed, other users need to "update" their files to reflect the new version of the repository. It is important to note that CVCSs only store the most recent version of a project, requiring other users to stay informed about source code changes. During the development phase, users can use branches to implement new features or functionalities. Although branches may occasionally encounter issues, they will not disrupt the overall project workflow. After thorough testing and implementation, branches can be merged into the source code. Further details about branches will be discussed later.

CVCSs have two main disadvantages, namely, single-point risk and low speed. As mentioned earlier, CVCS contains only one central server for storing repositories. If it goes down, the whole project becomes inaccessible, halting the development process. In addition, using one server creates another problem during the development phase. Users must communicate with the central server for each command (create a branch, push, merge, and so on), and this creates a vast traffic overload for the server, and often induces a slow response from the server.

DVCSs do not require a central server. They have the advantage of storing the entire repository on each user's local computer. This characteristic makes them particularly well-suited for large projects involving numerous independent

developers [17]. DVCS also offers faster performance compared to CVCS, as most commands are executed locally without the need for a network connection. Many of the terminologies used in CVCS are also applicable to DVCS. However, one potential drawback of DVCS is higher memory consumption since it stores the entire project locally. To mitigate this, DVCS employs compression techniques to reduce the repository size. Despite this concern, the DVCS approach is faster, more flexible, and reliable. If one repository becomes inaccessible, another user can quickly upload their own version. Due to these advantages, DVCS has gained significant popularity in the development industry. Well-known platforms such as GitHub and Mercury provide extensive DVCS services.

Starting from 2010, there have been several papers published addressing branching strategies. Papers, such as [1, 18, 19], mostly analyze branching as an advantage of DVCSs, explaining only some characteristics and how they work. The paper [20] can be considered as one of the most successful research projects, highlighting the usage of branches and the connection between different project parameters and branching strategy. However, researchers did not present a specific branching strategy or how to define it. In [20], the authors used nearly 3000 repositories, but only 200 of their branches were analyzed. Obviously, such a limited sample size cannot give any objective results, as it is a significantly small subset for drawing conclusions.

Another research paper, published in 2014, focuses on the role of branching strategies in developer team collaboration during the development process [21]. The paper presents interview results and gives information on how a branching strategy can affect the development process and how it is seen from the developer's perspective. Unfortunately, the authors do not focus on some specific branching strategies, and all of their findings are mainly focused on the idea of branching strategy without a comparison of different branching strategies.

As evident from the mentioned research, in most cases, researchers did not analyze specific branching strategies and their impact on the development process. Instead, they only explained it as one of the side factors in the development process. However, research presented below shows that choosing the correct branching strategy can be as important as selecting the right repository structure.

3. Branching strategies

The paper [22] provides a brief overview of branching strategies and their purposes, while this paper aims to explain these strategies in a more detailed way.

Git Flow – this strategy can be considered as one of the most complex branching strategies. It contains the master, develop and various feature and hotfix branches. The master branch is a branch where the main source code of a project

is stored and, in most cases, all branches eventually have to be merged into it. According to some regulations in Git, in recent years, the name "master" has started to be replaced with "main" in most GitHub repositories [23]. The develop branch, also known as release, is where developers prepare for the release of new production. Feature branches are created from master branch and mainly focus on implementing new features to the project and their testing phase. It is important to mention that there can be also additional branches for different purposes, such as bug fixing and adding documentation. For clarity, all of them will be grouped under feature branches in this paper. Given its diverse range of branch types, Git Flow is a complex branching strategy preferred typically by large and professional developer teams. Its biggest advantage is that it enables several developers to work in parallel while protecting the production source code. While the abundance of branches can be problematic, effective management can help each developer focus on her/his tasks, reducing conflicts during the merge process. On the other hand, this complex structure can cause problems during testing and other phases of development if it is not properly managed by team leaders or senior developers.

GitHub Flow – it can be considered as a simpler and less sophisticated version of Git Flow. This branching strategy does not have any release or development branches and consists solely of the master branch and different feature branches. This approach is quite popular among most developer teams and according to the collected database it has been used in 60% of mono-repository projects. This strategy is mostly used for quick and short development phases and gives clear control over the development process.

Trunk-based – it is the simplest branching strategy and is used by a large portion of developers on GitHub. In this strategy, there is only one master branch, and it is always ready for deployment. All development is conducted on this main branch and developers do not need to create additional branches for feature adding or bug fixing. The biggest advantage of this strategy is its simplicity. This is why it is greatly preferred by small developer teams or amateur developers. On the other hand, it also can cause problems since there is limited room for error or mistakes, and the source code must always be ready for deployment.

4. Motivation

The efficient management of source code repositories is a crucial aspect of software development projects. With the increasing popularity of both mono- and multi-repository structures, understanding the impact of branching strategies and their optimization becomes paramount. The motivation behind this paper stems from the need to explore the real-world data obtained from a vast array of projects, encompassing over 50 000 repositories across different types of projects.

By examining this extensive dataset, the paper aims to reveal valuable insights into team size variations, branching strategy preferences, and other pertinent factors that can significantly impact the development process. The primary goal of this paper is to conduct a comprehensive analysis of branching strategies in both mono- and multi-repository environments. By leveraging the large-scale dataset derived from real-world projects, the study aims to achieve the following objectives:

- Understand team size differences: Analyze the relationship between team size and the choice of repository structure (mono or multi). Investigate whether team size influences the preference for a specific branching strategy. Identify any notable patterns or trends related to team size and branching strategy.
- Evaluate branching strategy preferences: Examine the distribution of branching strategies (such as trunk-based, Git Flow, GitHub Flow, etc.) across different types of projects, within both mono- and multi-repository setups. Identify the most adopted strategies and uncover any variations or preferences specific to project types.
- Optimize branching strategies: Based on the insights gained from the analysis, propose recommendations and best practices for optimizing branching strategies in both mono- and multi-repository environments. Highlight the benefits, drawbacks, and considerations associated with different strategies, considering their impact on team collaboration, feature delivery, and bug-fixing efficiency.
- Provide practical insights: Offer practical insights and actionable recommendations for software development teams and organizations regarding the selection and implementation of branching strategies in their respective repository setups. Discuss the potential benefits of aligning branching strategies with project characteristics, team dynamics, and development goals.

By achieving these goals, the paper aims to contribute to the existing body of knowledge on optimizing branching strategies in software development, providing evidence-based insights that can aid practitioners and organizations in making informed decisions about their repository structures and branching approaches.

5. Methodology

5.1. Database

As it has been mentioned earlier, there are nearly 50 000 repositories in our database including 8479 multi-repository and 33 594 mono-repository projects.

Since each multi-repository project contains two repositories: one front-end and one back-end, the whole database consists of 50 552 repositories. Each project in the database is stored in JSON format for better processing and organizing.

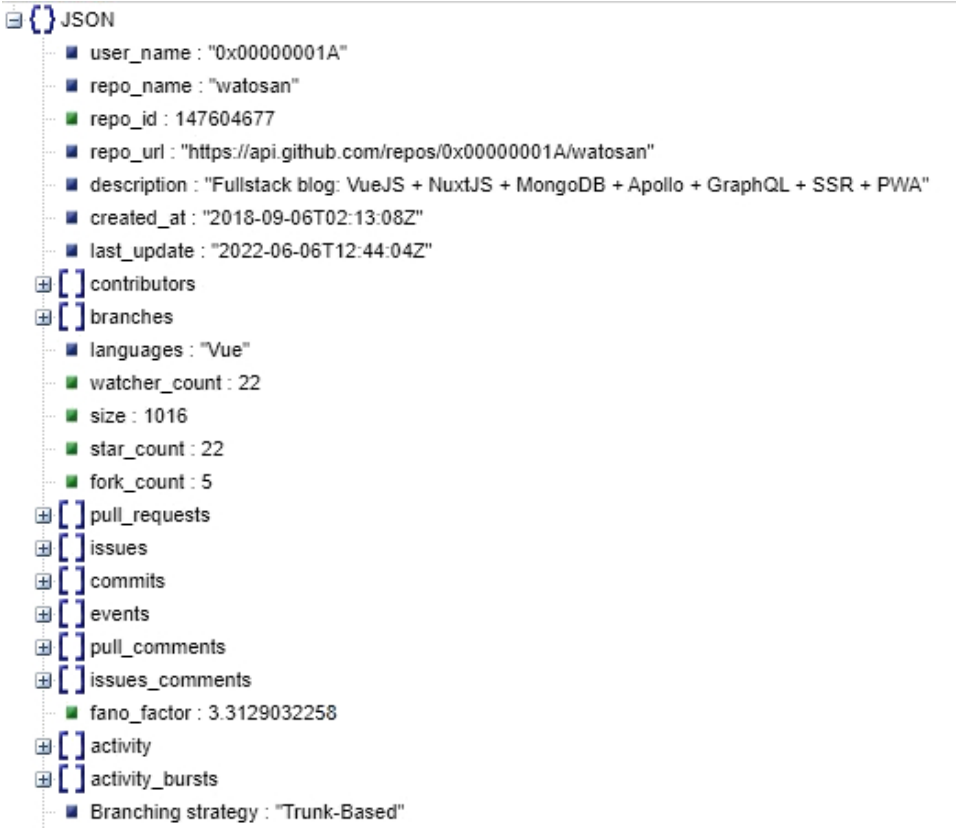

Fig. 1. Example of JSON for mono-repository project.

As illustrated in Fig. 1, each JSON file stores both basic parameters and features and also activities. GitHub API is used for collecting all of this data. The first seven fields and the following five fields contain basic information about the project, such as its name, name of the GitHub user, id of the repository which is assigned by GitHub itself, URL of repository, description, creation and last update dates, most used language in project, count of watchers, size in Kbs, star and fork counts. All of this data can be obtained in JSON format from API requests using the URL shown below:

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

The contributors section contains a list of all developers working on a project and the number of commits they have contributed to the project. This informa-

tion helps in the easy identification of main developers within the team. In order to get the list of all contributors we have to send another request to the GitHub API. An example of URL for this request is as follows:

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

The branches section contains a list of branches together with the list of commits for each branch. This allows to observe the life cycle of each branch and conduct an in-depth analysis. There is also a special URL type in the GitHub API for obtaining the list of branches, but unfortunately, the result does not contain a list of commits for each branch. Therefore, additional requests have to be sent for each branch. Examples of the URL are as follows:

- Branches: https://api.github.com/repos/{user name}/{repository name}/branches.
- Commit list for a certain branch: https://api.github.com/repos/{user name}/{repository name}/commits?sha={branch name}.

There are also six more sections that represent different types of activity parameters of the project. Each of these sections is created by specific URL requests (see Appendix).

Most of these fields are similar to the ones in response from the GitHub API. Fields such as *watcher_ count*, *start_ count* and *fork_ count* are added in order to rate the popularity of the given repository. Also, there are additional fields such as *fano_ factor*, *activity* and *activity_ bursts*, which are related to the productivity and activity of the given repository. Since they are the topic of future research, it is sufficient for now to say that these three field represents the amount of work done during a given time period on this repository. As mentioned above, the other six activity parameters: *pull_ requests*, *issues*, *commits*, *events*, *pull_ comments*, and *issues_ comments* are all related to the activity of the repository development process. All of these six fields contain a list of dictionaries about the given activity.

The structure of the multi-repository project is almost the same. The only difference lies in two different sections at the top: front repository and back repository, each containing the respective type of repository. So, all of these mentioned fields are collected separately for both parts of multi-repository projects.

5.2. Database creation

This process has already been described in [5], but here we will talk about it again for clarification. Overall, the whole process of database creation can be divided into two major parts: collecting mono-repository projects and collecting multi-repository projects.

5.2.1. Collecting mono-repository projects. There are over 372 million repositories on GitHub, so identifying those that can be added to our database can be a considerable challenge. That is why we added additional parameters to the URL in order to narrow down our search pool and obtain results much faster. An example of such a request URL is as follows:

https://api.github.com/search/repositories?q=fullstack+language:javascript +created:2022-01-01..2022-01-15.

This URL provides a list of all repositories containing the term "fullstack" in their name or description, written in JavaScript and created between January 1 and 15, 2022. This way we can find potential mono-repository projects. Different keywords and varied time ranges can be used in order to increase the effectiveness of the search process. In most of the cases, this is dependent on the researchers' objectives. For example, in the case of this research, it was more important to find projects with a structure similar to the "fullstack" technology.

Identification of mono-repository projects is conducted according to their file structure. Since this type of project combines all essential parts under one folder, it is sufficient to search for some key folder names in the file structure. These key folder names include: Frontend, Backend, UI, API, Client, Server, UI, Front and Back. The collection of file structure for mono-repository projects follows the algorithm given in [24]. To elaborate briefly, the GitHub API, unfortunately, does not provide a direct approach to collect the file structure of a repository. By file structure we simply mean the names of all folders and files in the repository. However, these names must be constructed in the same way as in repository itself. For example, if there is a "logo.png" file inside a folder named "images" and that folder is situated inside a folder named "assets", then file structure of this piece will be as follows:

"Assests / Images / logo.png".

This way it is possible to obtain the names of all folders and files in a repository. But, for the identification of mono-repository projects, there is no need to go too deep until individual file names are obtained. In this approach, the focus is on obtaining the names of the main folders and comparing those names with the ones mentioned before. Overall, the process for the identification and collection of mono-repository projects can be explained with the following steps:

Step 1. Collect a list of possible mono-repository projects and the users to which they belong.

Step 2. Analyze all repositories of the identified GitHub user.

Step 3. If the analyzed projects are valid, add their names to a temporary database.

Step 4. Retrieve all necessary data for the project in JSON and save it into the database.

5.2.2. Collecting multi-repository projects. The process of identifying and collecting multi-repository projects shares some similar steps with mono-repository ones. The first step – collecting a potential list of multi-repository projects, is the same as it was in the mono-repository case. However, the identification of multi-repository projects is different. Firstly, all repositories of users are separated into three groups: front-end, back-end and others. This grouping of repositories into main groups is achieved through a machine learning (ML) method described in [24]. Simply, the ML model is trained based on the file structures of front-end and back-end repositories. After the identification of these two repository groups, if there is at least one repository in each group, we start the matching process.

To investigate the impact of different feature combinations on matching success rates, the K-nearest neighbor (KNN) algorithm was employed. KNN is a supervised learning method widely used for classification tasks. The algorithm assigns a class label to a query instance by considering its nearest neighbors in the feature space. The experiments were conducted using various feature combinations, and the success rates of repository matching were recorded. The success rate represents the proportion of correctly matched repositories out of the total number of instances. The following feature combinations were evaluated:

- All parameters: repository name, programming language, framework, database type, list of developers, file structure, and readme file (36% success rate).
- All parameters except readme file: repository name, programming language, framework, database type, list of developers, and file structure (32% success rate).
- All parameters except file structure: repository name, programming language, framework, database type, list of developers, and readme file (48% success rate).
- Repository name and readme file: repository name and readme file only (89% success rate).

The experimental results reveal that the choice of feature combination significantly impacts the success rate of front-end and back-end repository matching. The combination of "repository name" and "readme file" achieved the highest success rate of 89%, indicating their strong discriminative power in determining repository matches. This finding suggests that textual information, such as project names and accompanying readme files, plays a crucial role in identifying related repositories within a project.

After completing this matching process, the last two steps are nearly identical to the previous mono-repository case. So, the overall steps for collecting multi-repository projects are as follows:

Step 1. Collect a list of possible multi-repository projects and the users to which they belong.

Step 2. Group the repositories of the identified GitHub user into three groups.

Step 3. Match repositories in the front-end and back-end groups and thus identify multi-repository projects.

Step 4. If the identified projects are valid, add their name to a temporary database.

Step 5. Retrieve all needed data of the project in JSON and save it into the database.

5.3. Branching strategy identification

In the preceding chapters, we have presented three main branching strategies. Each branching strategy is distinguished by the list of branches it includes. In this case, it is sufficient to check the name and count of branches in the project and this way we can identify its branching strategy. As shown in Fig. 1, the database contains a list of all branches and only checking their names is sufficient for identification. Here is an algorithmic step for this:

- Input: Set of branches in the project,
- Output: Branching strategy of the project,
- *identifyBranchingStrategy*: Set of branches → Branching strategy,
- *identifyBranchingStrategy(branches)*:

Filter out branches created by bots, resulting in a set of non-bot branches.
If the number of non-bot branches is 1:

a. Check if the branch is a master branch by examining its name.
b. If the branch name is “master” or “main”, return “trunk-based” as the branching strategy.
c. Otherwise, return “unknown” as the branching strategy since it does not match any known strategies.

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

a. Check if there is a master branch by examining its name.
b. If there is no master branch, return “unknown” as the branching strategy.
c. Check if there is a development branch by examining its name.
d. If the development branch name is “dev” or “development”, return “Git Flow” as the branching strategy.
e. Otherwise, check if there are feature or bug fix branches by examining their names.
f. If any of the branch names contain “feature”, “bug fix”, “bug”, or “hotfix”, return “GitHub Flow” as the branching strategy.

g. Otherwise, return "unknown" as the branching strategy since it does not match any known strategies.

If the given branching strategy is identified as unknown it will be saved in an additional database for further inspection. This is a useful approach because, sometimes, algorithm can make some mistakes, most often caused by human factor. In some cases, developers just name branches by either using a very specific approach or in a completely wrong way. That is why it is important to save all the projects with the "unknown" branching strategy. This way we are able not only to prevent the loss of important piece of data, but also get more experience about the naming and other practices of developers from different backgrounds.

6. Formal specifications

To give a clear understanding about the used terms it is beneficial to present them in a mathematical way. While definitions of mono- and multi-repository structures have been given in previous chapters, mathematical formulation for these terms were not provided. Starting with the versioning system, those definitions can be given as follows.

6.1. Mono-repository

As it was mentioned earlier, a mono-repository structure is a software development approach that involves storing all of a project's source codes in a single repository. To formally specify a mono-repository structure, we can use set theory and formal logic. Let R be the mono-repository, and let S be the set of all source files in the repository. We can define R as a collection of subdirectories, each of which contains a subset of S. Formally, we can represent R as follows:

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

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

$$D_i = \{f_i | f_i \in S \wedge f_i \quad \text{is in the directory} \quad D_i\}.$$

This notation states 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 D_k is a subset of another subdirectory D_j by stating:

$$D_k \subseteq D_j.$$

We can also define relationships between 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:

$$F_j \quad \text{depends on} \quad f_i.$$

This notation indicates that the code in file f_j relies on the code in 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.

6.2. Multi-repository

To formally specify a multi-repository structure, we can use set theory and formal logic. Let $R_1, R_2, ..., R_n$ be the individual repositories that make up the multi-repository structure, and let S be the set of all source files across all repositories. We can define each repository R_i as a collection of source files:

$$R_i = \{f_i | f_i \in S \wedge f_i \quad \text{is in the repository} \quad R_i\}.$$

This notation states that each repository R_i is a subset of S, containing only the files that are located within that repository. We can use set operations to specify relationships between repositories, such as union ($\cup$) and intersection ($\cap$). For example, we can specify that a repository R_k is a subset of another repository R_j by stating:

$$R_k \subseteq R_j.$$

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

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

This notation indicates that the code in file f_j relies on the code in file f_i to function properly. Overall, the formal specification of a multi-repository structure involves defining each repository as a collection of source files and specifying relationships between repositories and files using set theory and logical operators.

6.3. Mono-repository identification

Input:

- A set P of possible mono-repository projects, where each project $p \in P$ is associated with a user.

Output:

- A database D containing the identified mono-repository projects and their associated data.

Let U be the set of GitHub users:
$identifyMonoRepositories$: $P \times U \rightarrow P'$
$collectMonoRepositories$: $P* \rightarrow D$
$identifyMonoRepositories(P, u) = \{p' \in P | analyseRepositories(u)\}$
$collectMonoRepositories(P) = \{(p, jsonData(p)) | p \in P\}$
where:

- $analyseRepositories(u)$ is a function that analyses the repositories of user u and returns true if they meet the criteria for a mono-repository, and false otherwise,
- $jsonData(p)$ is a function that retrieves the needed data of project p in JSON format.

Note: The functions $analyseRepositories(u)$ and $jsonData(p)$ are implementation-specific and can vary based on the criteria and data you require for identifying and collecting mono-repository projects.

The algorithm can be summarized as follows:

1. Iterate through each project $p \in P$ and its associated user u.
2. Use the *identifyMonoRepositories* function to check if the repositories of user u meet the criteria for a mono-repository.
3. Collect the identified projects in the set P'.
4. Use the *collectMonoRepositories* function to retrieve the needed data for each identified project $p' \in P'$ and store it in the database D.

The resulting database D will contain the identified mono-repository projects and their associated data.

6.4. Multi-repository identification

Input:

- A set P of possible multi-repository projects, where each project $p \in P$ is associated with a user.

Output:

- A database D containing the identified multi-repository projects and their associated data.

Let U be the set of GitHub users:
$identifyMultiRepositories$: $P \times U \rightarrow P'$
$collectMultiRepositories$: $P* \rightarrow D$
$identifyMultiRepositories(P, u) = \{p' \in P | groupRepositories(u)\}$
$collectMultiRepositories(P) = \{(p, jsonData(p)) | p \in P\}$

where:

1. *groupRepositories*(u) is a function that groups the repositories of user u into three groups: front-end, back-end, and others.
2. *jsonData*(p) is a function that retrieves the needed data of project p in JSON format.

Note: The functions *groupRepositories*(u) and *jsonData*(p) are implementation-specific and can vary based on the criteria and data you require for identifying and collecting multi-repository projects.

The algorithm can be summarized as follows:

1. Iterate through each project $p \in P$ and its associated user u.
2. Use the *identifyMultiRepositories* function to group the repositories of user u into three groups: front-end, back-end, and others.
3. Match repositories in the front-end and back-end groups to identify multi-repository projects.
4. Collect the identified projects in the set P'.
5. Use the *collectMultiRepositories* function to retrieve the needed data for each identified project $p' \in P'$ and store it in the database D.

The resulting database D will contain the identified multi-repository projects and their associated data.

7. Measurements

In this chapter, we present the results of a study that analyzed different statistics related to mono-/multi-repository structures, branching strategies, and other related topics. The data used in this analyses were collected by the algorithm mentioned in the previous chapter. We analyzed the data using various statistical techniques, and we present our findings in this chapter. This study sheds light on different aspects of repository and branching strategies. Our goal is to provide developers with a comprehensive overview of the current state of practice in this area and to help them make informed decisions about their own development processes.

7.1. Team size in mono-/multi-repository projects

In mono-repository projects (Fig. 2), most projects (86%) are developed by a single developer. This indicates that many software development projects are undertaken by individuals working alone, potentially in smaller organizations or as solo projects. Only 9% of projects involve two developers, which may suggest that pair programming or code reviews are less common in mono-repository projects. Between 3 and 10 developers are involved in 5% of mono-repository

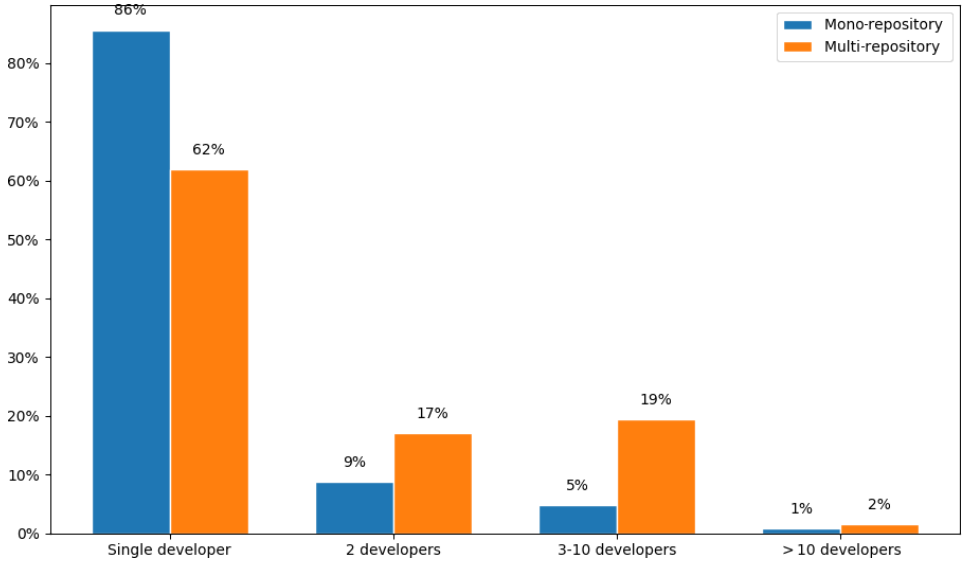

Fig. 2. Team size percentage in mono- and multi-repository projects.

projects, indicating that larger teams are less common. Lastly, only 1% of mono-repository projects have more than 10 developers, indicating that these projects are relatively rare. In contrast, multi-repository projects have a more evenly distributed team size. While 62% of projects have a single developer, which is still the majority, the proportion of projects with multiple developers is higher than in mono-repository projects. Specifically, 17% of projects have two developers, indicating that pairing or code review may be more common in multi-repository projects. Between 3 and 10 developers are involved in 19% of multi-repository projects, indicating that larger teams are more common in these projects. Lastly, only 2% of multi-repository projects have more than 10 developers, which is a slightly higher proportion than in mono-repository projects.

7.2. Programming language in mono-/multi-repository structures

Figure 3 shows that JavaScript is by far the most popular language in mono-repository projects, with 48.9% of projects using it. This is unsurprising given the widespread popularity of JavaScript as a language for web development. TypeScript is the second most popular language in mono-repository projects, with 13.6% of projects using it. This reflects a growing trend towards the use of TypeScript, which is a superset of JavaScript that adds type annotations and other features. HTML-CSS is the third most popular language in mono-repository projects, with 19.1% of projects using it. This reflects the importance of HTML and CSS in web development, as these languages are used to create

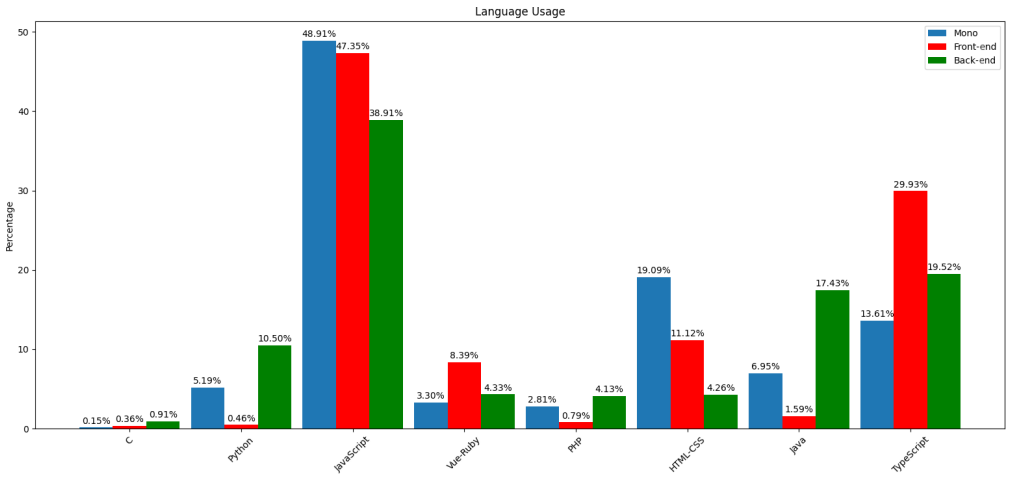

Fig. 3. Usage percentages of languages in mono- and multi-repository projects.

the structure and styling of web pages. Java is the fourth most popular language in mono-repository projects, with 6.9% of projects using it. This is likely due to the widespread use of Java in enterprise applications and other large-scale systems. Python is the fifth most popular language in mono-repository projects, with 5.2% of projects using it. This reflects the growing popularity of Python as a general-purpose programming language, as well as its widespread use in data science and machine learning. PHP is the sixth most popular language in mono-repository projects, with 2.8% of projects using it. This reflects the continued use of PHP in web development, particularly in the creation of server-side scripts and content management systems. Vue-Ruby is the seventh most popular language in mono-repository projects, with 3.3% of projects using it. This is likely due to the use of the Vue.js JavaScript framework and the Ruby programming language in web development projects. Finally, C languages are the least popular languages in mono-repository projects, with only 0.1% of projects using them. This may reflect the fact that C is a low-level language that is typically used for systems programming rather than web development.

In the front-end part of multi-repository projects, the chart shows that JavaScript is still the most popular language, with 47.35% of projects using it. This is followed by TypeScript, which is used in almost 30% of projects. This is not surprising as TypeScript is a strongly typed superset of JavaScript and has become increasingly popular in recent years, particularly in front-end development. Vue-Ruby is the third most popular language in the front-end part of multi-repository projects, with 8.39% of projects using it. This is likely due to the use of the Vue.js JavaScript framework and the Ruby programming language in web development projects. HTML-CSS is the third most popular language in the front-end part

of multi-repository projects, with 11.12% of projects using it. This reflects the importance of these languages in front-end web development.

Turning to the back-end part of multi-repository projects, the chart shows that JavaScript is still the most popular language, but its usage is lower compared to the front-end part, with 38.91% of projects using it. This is followed by TypeScript with over 19% and Java with over 17% of projects. This is not surprising given the widespread use of Java in enterprise applications and other large-scale systems. Python is the third most popular language in the back-end part of multi-repository projects, with 10.50% of projects using it. This reflects the growing popularity of Python as a general-purpose programming language, as well as its widespread use in data science and machine learning. TypeScript is the second most popular language in the back-end part of multi-repository projects, with 19.52% of projects using it. This suggests that TypeScript is also gaining traction in back-end development.

7.3. Development period in mono-/multi-repository structures

Figure 4 shows that the majority of both mono- and multi-repository projects have a development period of less than a month, with 46% of mono-repository projects and 40% of multi-repository projects falling into this category. In the next category, projects with a development period between 1 and 3 months, there is a similar distribution between mono- and multi-repository projects, with 19% and 16%, respectively. This suggests that for slightly larger projects, there is little difference in development period between the two repository structures. For longer development periods, there are slightly more multi-repository

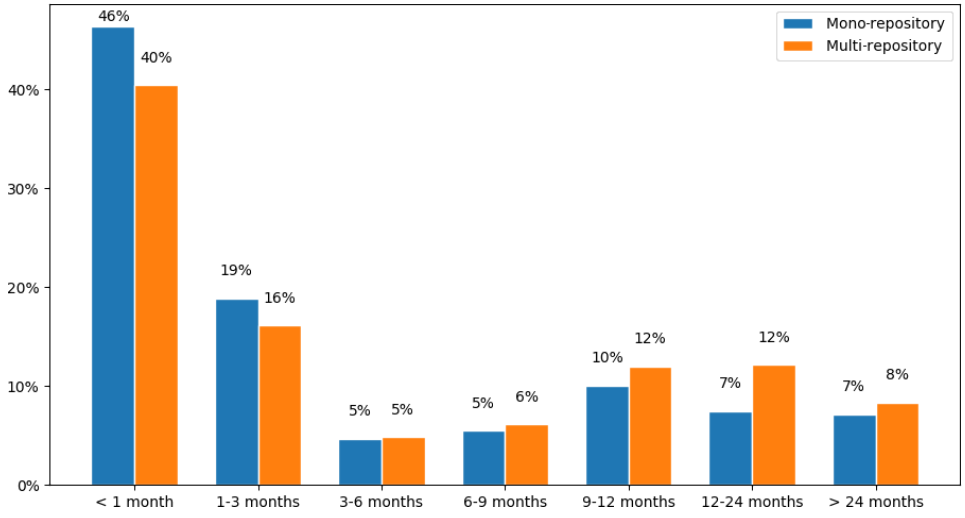

Fig. 4. Percentage share of projects in both structures according to their development period.

projects with a development period between 12 and 24 months (12% vs. 7% for mono-repository projects), suggesting that larger-scale projects may be more commonly organized as multi-repository projects. However, for projects with development periods of more than 24 months, there is not a significant difference between the two repository structures.

7.4. Popularity of branching strategies over the years

The popularity of branching strategies in software development has evolved over the years. Analyzing the percentage trends from 2016 to 2022, it is evident that trunk-based, Git Flow, and GitHub Flow have undergone shifts in popularity. In 2016, GitHub Flow was the preferred choice, capturing 58% of the popularity, followed by trunk-based at 32% and Git Flow at 10%. However, by 2017, trunk-based experienced a significant increase, rising to 52.5%, while GitHub Flow dropped to 39.5% and Git Flow remained at 8%. In subsequent years, trunk-based and GitHub Flow maintained their prominence, with minor fluctuations. Between 2018 and 2021, trunk-based ranged from 40% to 49%, while GitHub Flow hovered between 47% and 58%. Git Flow saw a decline, reaching a low of 1% by 2022. These trends indicate a preference for more flexible and streamlined branching strategies. Trunk-based, known for continuous integration, and GitHub Flow, emphasizing lightweight workflows, have gained popularity over the years. Meanwhile, the structured approach of Git Flow has witnessed a decline in usage.

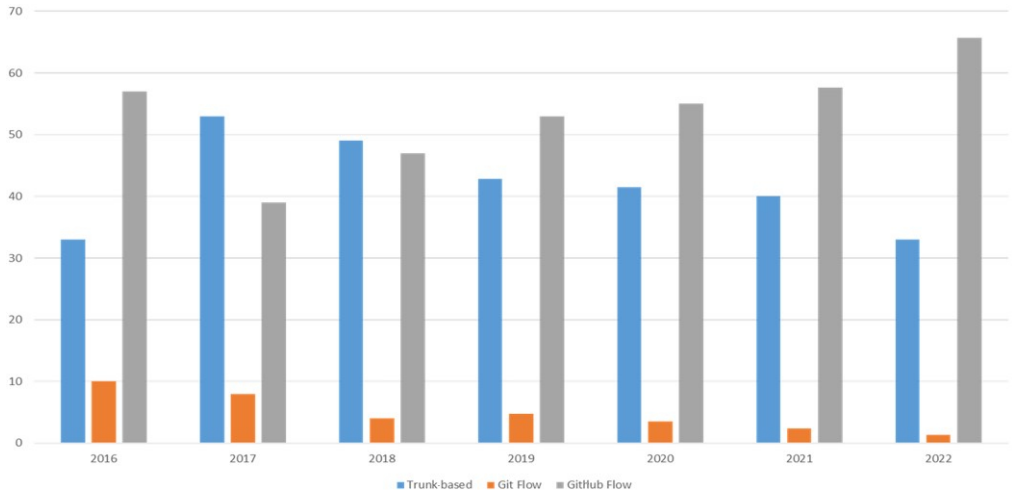

Fig. 5. Popularity of three major branching strategies over the years.

Understanding the evolving popularity of branching strategies is essential for developers, enabling them to align their practices with industry trends and make

informed decisions. It highlights the industry's pursuit of efficient and collaborative development workflows.

7.5. Branching strategies in mono-/multi-repository projects

Based on Fig. 6, it seems that most mono-repository projects use the trunk-based branching strategy (76.3%). This strategy involves all developers committing their changes directly to the main trunk or branch of the codebase. This approach is often used for smaller projects with a small number of developers, as it allows for rapid iteration and faster feedback loops. The next most popular branching strategy in mono-repositories is GitHub Flow (19%). This strategy involves creating a new branch for each feature or bug fix and merging it back into the main branch once it is completed and tested. This approach allows for better collaboration among teams and provides a clearer history of changes made to the codebase. Finally, Git Flow branching strategy is used in a minority of mono-repository projects (4.7%). This strategy involves creating a structured branching model with a main branch, develop branch, and feature branches for each new development task. This approach provides a more organized and controlled development process, but it can be more complicated and time-consuming.

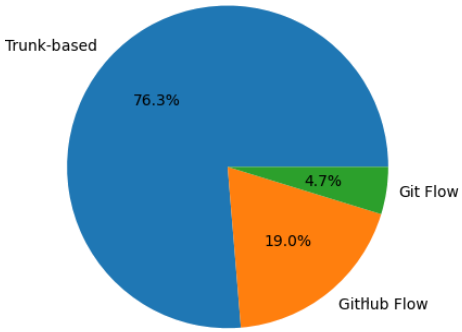

Fig. 6. Usage percentage of three major branching strategies in mono-repository projects.

According to Fig. 7 it appears that in the front-end part of multi-repository projects, most projects use trunk-based branching strategy (57.6%). This is followed by GitHub Flow (27.5%) and Git Flow (14.9%). Similarly, in the back-end part of multi-repository projects, most projects also use trunk-based branching strategy (60.7%). The usage percentages of GitHub Flow and Git Flow in the back-end part are like those in the front-end part, with GitHub Flow being the second most popular strategy (24.2%) and Git Flow being the least popular (15.0%). It is interesting to note that the usage percentages of trunk-based branching strategy are higher in the back-end part of multi-repository projects compared to the front-end part. This could be because back-end development

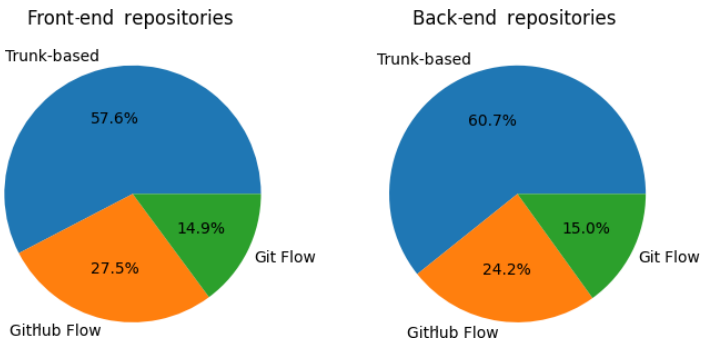

Fig. 7. Usage percentages of three major branching strategies in both parts of multi-repository projects.

often involves more complex code and requires a more structured development process. Overall, the usage percentages of the three branching strategies in multi-repository projects suggest that these projects tend to be larger and more complex than mono-repository projects, with multiple teams working on different parts of the project.

8. Discussion

8.1. RQ 1. What is the connection between team size and repository structure?

Figure 2 showed that mono-repository structure is mostly preferred by a small group of developers. More than 90% of the mono-repository projects are developed by developer teams smaller than 5. It is worth noting that there is a significant number of freelancer or nonprofessional projects in our database; nevertheless, 90% is still a substantial number. To gain a better understanding about this relationship, we also conduct some measurements on the remaining 10% of the projects and calculate the correlation between team size and other parameters of projects.

Again, we must mention that the projects shown in Fig. 8 are the 10% parts of mono-repository projects with a team size of more than 5. According to our calculations, there is a 0.48 correlation between team size and project size in this case. In most cases, this is called a "medium correlation", suggesting a medium chance for mono-repository structure projects to have bigger team size with the increase in project size. On the other hand, a different perspective can be observed in the team size share of multi-repository projects in Fig. 2. It is obvious that larger teams prefer to work with multi-repository projects. Approximately 20% of multi-repository projects have developers with team sizes exceeding 5. The correlation result for project size and team size in these projects shows that

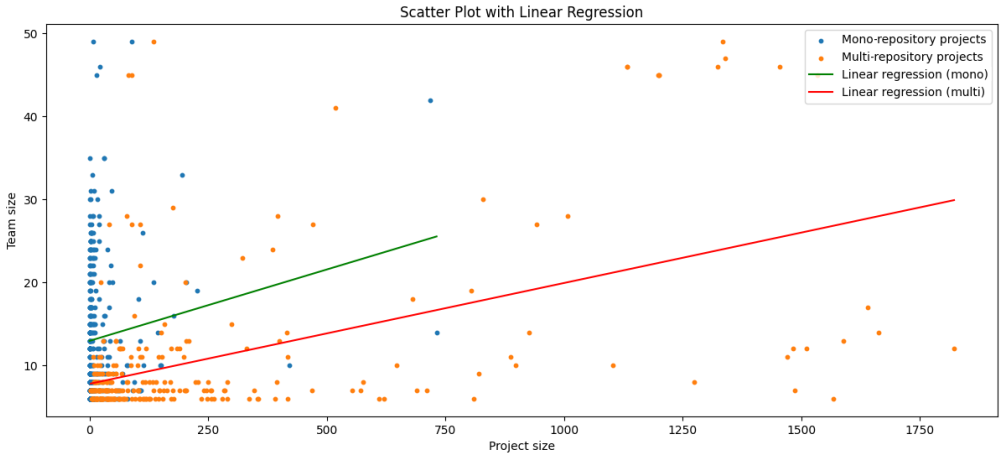

FIG. 8. Relationship between team size and project size in mono- and multi-repository projects.

there is weak correlation of 0.22. This proves that the count of team members has very little relationship with the project size.

8.2. RQ 2. What is the connection between development period and repository structure?

Figure 9 illustrates the relationship between team size and development period in mono-repository structure projects. As depicted, there is no strong correlation between these two parameters with a correlation value is 0.15, indicating a weak correlation. Figure 4 shows that almost 65% of the projects have been developed in under 3 months and only 24% of the projects have a lifespan of more than a year.

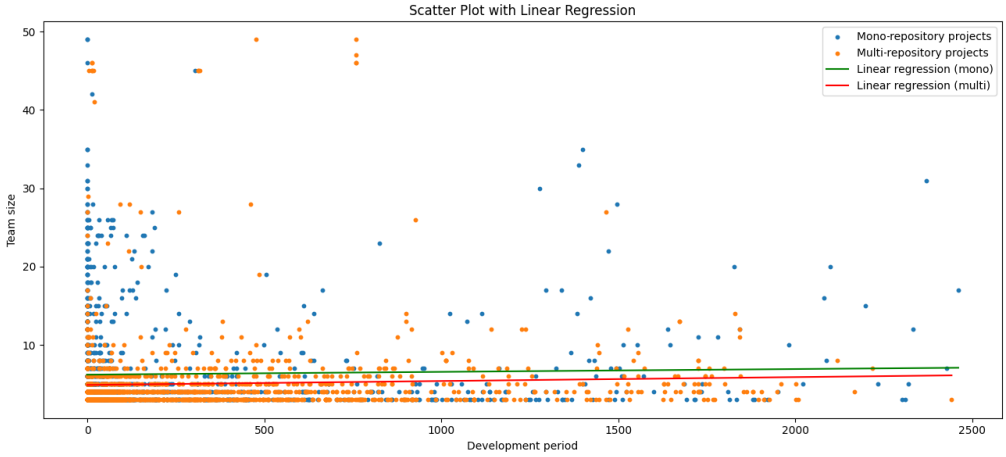

FIG. 9. Relation between development period and team size in mono-repository projects.

In the multi-repository case, the scenario differs. Almost 34% of the projects have a development period exceeding a year and this indicates that the multi-repository approach is mostly preferred in projects designed for longer development period. Furthermore, there is very weak correlation between team size and the development period in this approach, as it was in mono-repository one. So overall, one can clearly say that team size has a very weak connection with the development period.

8.3. RQ 3. Which branching strategy is preferred according to the repository structure?

The trunk-based approach is highly favored in mono-repository projects, with a significant usage percentage of 76.3%, while Git Flow is the least popular option at only 4.7%. Interestingly, in both the front-end and back-end sections of multi-repository projects, the trunk-based approach remains the most prevalent, although its usage percentage is lower compared to mono-repository projects. In multi-repository front-end projects, GitHub Flow emerges as the second most popular choice with a usage percentage of 27.5%, while Git Flow is the least favored at 14.9%. In multi-repository back-end projects, Git Flow and GitHub Flow have similar usage percentages at 15.0% and 24.2%, respectively.

These statistics highlight a strong inclination of developers towards the trunk-based approach in mono-repository projects. This approach is known for its simplicity and straightforwardness, enabling faster delivery of new features and bug fixes. It is worth noting that Git Flow, typically associated with more complex projects, is less commonly used in both mono-repository and multi-repository setups. This could be attributed to its steep learning curve, discouraging some developers from adopting it. On the other hand, GitHub Flow finds greater preference in multi-repository structures compared to mono-repository setups.

8.4. RQ 4. Which branching strategy is preferred according to the team size, programming language and development period?

Figure 3 shows the usage percentage of different programming languages in mono-repository projects. However, here we will talk about how these language preferences are distributed among different team sizes. Four major programming languages have been taken and their usage statistics are shown in Table 1.
Single developer projects:

- JavaScript: The majority of projects (13 504) are led by a single developer, indicating a strong preference for JavaScript in solo development scenarios. This suggests that JavaScript is widely adopted by individual developers who work on smaller projects.

Table 1. Statistics of team sizes in four major programming languages in mono-repository structure projects.

Programming languages	Single developer	Two developers	Between 3–10 developers	More than 10 developers
JavaScript	13 504	1125	576	98
Python	3762	349	148	21
TypeScript	1444	96	73	29
Ruby	387	49	42	37

- TypeScript, Python, and Ruby: While TypeScript, Python, and Ruby also have projects led by a single developer, their counts are comparatively lower. This could indicate that these languages are somewhat less commonly used for solo development or that they are preferred for larger projects that require multiple developers.

Two developer projects:

- JavaScript: With 1125 projects, JavaScript also demonstrates popularity in collaborative efforts involving two developers. This suggests that JavaScript is frequently chosen for small team collaborations.
- TypeScript and Python: TypeScript and Python have a smaller number of projects with two developers (96 and 349, respectively). However, these counts still indicate a preference for these languages in team settings, albeit to a lesser extent compared to JavaScript.
- Ruby: Ruby's count of 49 projects with two developers suggests a relatively lower preference for this language in team collaborations of this size.

Between 3 and 10 developer projects:

- JavaScript: With 576 projects falling within this team size range, JavaScript shows a moderate preference for larger team collaborations. This implies that JavaScript is widely used in both small and medium-sized development teams.
- TypeScript, Python, and Ruby: TypeScript (73 projects) and Python (148 projects) also demonstrate a preference for team collaborations within this size range. While Ruby has a lower count of 42 projects, it still suggests some usage in medium-sized teams.

More than 10 developer projects:

- JavaScript: Despite its popularity in smaller team sizes, JavaScript's count of 98 projects with more than 10 developers indicates that it is also used in larger-scale development efforts.
- TypeScript and Python: TypeScript (29 projects) and Python (21 projects) show a relatively lower preference for projects with more than 10 devel-

opers. This could indicate that these languages are used less frequently in larger team settings or that they are more commonly associated with smaller to medium-sized projects.

- Ruby: Ruby's count of 37 projects with more than 10 developers suggests a higher preference for large-scale collaborations within the Ruby community.

Finally, we also can demonstrate the usage of different branching strategies by different developer teams.

Table 2. Statistics of team sizes in three major branching strategies in mono-repository structure projects.

Branching strategy	Single developer	Two developers	Between 3–10 developers	More than 10 developers
Trunk-based	16 113	1308	464	86
GitHub Flow	2622	508	492	99
Git Flow	581	144	144	10

Trunk-based:

- Single developer: With 16 113 projects, the trunk-based branching strategy is highly preferred by individual developers. This suggests that solo developers often adopt a simpler approach with fewer branches, favoring the trunk-based strategy.
- Two developers: Trunk-based branching is also popular among teams of two developers, with 1308 projects. This indicates that small teams prefer a streamlined development process without complex branching structures.
- Between 3 and 10 developers: Although the count decreases to 464 projects, the preference for trunk-based branching remains notable among medium-sized teams. This indicates a continued preference for a simplified development workflow without extensive branching.
- More than 10 developers: Even with larger teams, trunk-based branching is still utilized in 86 projects. This suggests that some organizations or projects with significant team sizes prefer a streamlined and less complex branching approach.

GitHub Flow:

- Single developer: GitHub Flow is adopted in 2622 projects by individual developers. This suggests a notable preference for this branching strategy among solo developers, potentially due to its simplicity and ease.
- Two developers: With 508 projects, GitHub Flow is also favored by small teams. This indicates that small collaborative efforts appreciate the simplicity and efficiency provided by GitHub Flow.

- Between 3 and 10 developers: GitHub Flow remains a preferred branching strategy among medium-sized teams, with 492 projects. This suggests that teams of this size find value in the straightforward and flexible workflow provided by GitHub Flow.
- More than 10 developers: GitHub Flow is adopted in 99 projects with more than 10 developers, indicating that some larger teams also appreciate the streamlined approach offered by this branching strategy.

Git Flow:

- Single developer: Git Flow is used in 581 projects by individual developers. Although the count is lower compared to trunk-based and GitHub Flow, it still indicates a preference for a more structured and feature-oriented branching strategy even in solo development scenarios.
- Two developers: With 144 projects, Git Flow is favored by small teams. This suggests that some small teams find value in the feature branching and release management aspects provided by Git Flow.
- Between 3 and 10 developers: Git Flow is also preferred by medium-sized teams, with 144 projects. This indicates that teams of this size appreciate the clear separation of features and releases provided by Git Flow.
- More than 10 developers: Git Flow is employed in 10 projects with more than 10 developers. This suggests that some larger teams still find value in the more structured and formalized branching approach of Git Flow.

Overall, the analysis highlights the following preferences for branching strategies by different team sizes:

- Trunk-based: Preferred by solo developers and teams of all sizes, indicating a preference for a simpler development workflow.
- GitHub Flow: Preferred across all team sizes, showcasing its flexibility and ease of use for both small and large teams.
- Git Flow: Exhibits a preference among developers working individually or in small to medium-sized teams, emphasizing the value placed on feature branching and release management.

9. Conclusion

The analysis reveals several noteworthy points differentiating mono-repository and multi-repository projects. The first point of interest is the team size distribution between both projects. It is clear from the statistics that mono-repository projects typically require fewer developers than multi-repository projects. In fact, over 85% of mono-repository projects have only one developer, compared to 62% in multi-repository projects. This suggests that mono-repository projects

are more likely to be undertaken by individual developers or small teams, whereas multi-repository projects are more often the result of collaboration among larger teams.

Furthermore, when considering projects with more than three developers, the difference between the two types of repositories becomes even more apparent. Only 5% of mono-repository projects have 3–10 developers, while 19.5% of multi-repository projects have such a team size. Additionally, projects with over 10 developers constitute 0.9% in mono-repository and 1.6% in multi-repository projects. This indicates that multi-repository projects are more likely to be undertaken by larger teams, perhaps due to the increased complexity and scope of such projects.

The second point to consider is the difference in development period between the two types of repositories. The statistics suggest that mono-repository projects are more frequently used in shorter duration projects, with more than half of the projects falling into the less than six-month duration category. In contrast, multi-repository projects exhibit a higher usage percentage in longer-term projects, with 12% of such projects lasting between 12 and 24 months, and 8% lasting more than 24 months. This suggests that mono-repository projects are typically preferred for smaller-scale or simpler projects, while multi-repository projects are more suitable for larger and more complex projects that require more time to complete.

Finally, the statistics on branching strategies reveal some interesting insights into the preferences of developers when it comes to managing code changes. In mono-repository projects, the trunk-based approach is by far the most common, with a usage percentage of 76.3%, while the Git Flow approach is used the least at 4.7%. On the other hand, in both the front-end and back-end parts of multi-repository projects, trunk-based remains the most used approach, but its usage percentage is lower than that in mono-repository projects. In the front-end part of multi-repository projects, GitHub Flow emerges as the second most used approach at 27.5%, while Git Flow is the least used at 14.9%. In the back-end part of multi-repository projects, Git Flow and GitHub Flow have similar usage percentages, with 15.0% and 24.2%, respectively.

Overall, these statistics suggest that developers prefer the trunk-based approach in mono-repository projects. This approach is typically simpler and more straightforward than other branching strategies, and it enables faster delivery of new features and bug fixes. It is also worth noting that the Git Flow approach, which is often associated with more complex projects, is less commonly used in both mono-repository and multi-repository projects. This could be due to its relatively steep learning curve, which may deter some developers from using it. On the other hand, GitHub Flow is preferred more in multi-repository structures rather than in mono-repository ones.

In conclusion, the provided statistics reveal some interesting differences between mono-repository and multi-repository projects, particularly in terms of team size and development duration. While mono-repository projects are typically smaller in scale and require fewer developers, multi-repository projects are more complex and often involve larger teams. The trunk-based branching strategy is the most used approach in both types of repositories, likely due to its simplicity and speed. However, the notable usage of GitHub Flow in both the front-end and back-end part of the multi-repository project suggests this branching strategy's suitability for this type of repository structure projects.

Based on the findings and insights gained from this study on branching strategies and the comparison between mono- and multi-repository structures, future research should focus on investigating the impact of different branching strategies on the productivity of software development teams. This future work will involve conducting empirical studies and collecting quantitative and qualitative data to assess the effectiveness of various branching strategies in terms of enhancing productivity, reducing conflicts, improving collaboration, and enabling efficient release management. Furthermore, exploring the interaction between branching strategies and other factors such as team size, project complexity, and development methodologies will provide a comprehensive understanding of how these strategies influence productivity in different contexts. The outcomes of this research will contribute to the body of knowledge in software engineering and provide valuable insights for practitioners in selecting the most suitable branching strategy to optimize their development processes.

10. Threats to validity

The present study acknowledges several potential threats to the validity of the findings. First, the reliability of the collected data from GitHub repositories could be of concern. While efforts were made to select repositories with a good reputation and active development communities, it is possible that some projects may contain inaccuracies or inconsistencies in their documentation or issue tracking. To mitigate this threat, multiple measures were taken, including the double-checking of repositories, conducting thorough data cleansing and validation processes. However, it is important to note that some degree of error or misinterpretation may still exist.

Despite these potential threats to validity, our study provides valuable insights into optimizing branching strategies in mono and multi-repository environments. By acknowledging and addressing these threats, we strive to enhance the reliability and applicability of our findings, ultimately contributing to the body of knowledge in software engineering and aiding practitioners in making well-informed decisions regarding branching strategies.

Appendix

Pull requests:
https://api.github.com/repos/{user name}/{repository name}/pulls?state=all
Issues:
https://api.github.com/repos/{user name}/{repository name}/issues?state=all
Commits:
https://api.github.com/repos/{user name}/{repository name}/commits
Events:
https://api.github.com/repos/{user name}/{repository name}/events
Pull comments:
https://api.github.com/repos/{user name}/{repository name}/pulls/comments
Issue comments:
https://api.github.com/repos/{user name}/{repository name}/issues/comments

References

1. D. Arve, Branching strategies with distributed version control in agile projects, pp. 1–12, 2010, https://fileadmin.cs.lth.se/cs/Personal/Lars_Bendix/Teaching/Lund/Coaching-course/2015-16/Reports/2010/Arve.pdf.
2. GitKraken, What is the best Git branch strategy?, 2023, https://www.gitkraken.com/learn/git/best-practices/git-branch-strategy.
3. J.D. Blackburn, G.D. Scudder, L.N. Van Wassenhove, Improving speed and productivity of software development: a global survey of software developers, *IEEE Transactions on Software Engineering*, **22**(12): 875–885, 1996, doi: 10.1109/32.553636.
4. R. Baskerville *et al.*, Balancing quality and agility in Internet speed software development, [in:] *Proceedings of the International Conference on Information Systems*, ICIS 2002, Barcelona, Spain, December 15–18, 2002, https://aisel.aisnet.org/icis2002/89.
5. R.T. Ogawa, B. Malen, Towards rigor in reviews of multivocal literatures: Applying the exploratory case study method, *Review of Educational Research*, **61**(3): 265–286, 1991, doi: 10.3102/00346543061003265.
6. C. Jaspan *et al.*, Advantages and disadvantages of a monolithic repository: A case study at Google, [in:] *ICSE-SEIP '18: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pp. 225–234, 2018, doi: 10.1145/3183519.3183550.
7. U. Shakikhanli, V. Bilicki, Comparison between mono and multi repository structures, *Pollack Periodica*, **17**(3): 7–12, 2022, doi: 10.1556/606.2022.00526.
8. A. Lucido, Mono-repo to multi-repo and back again, 2017, retrieved Jan 25, 2019 from https://www.youtube.com/watch?v=lV8-1S28ycM.
9. G. Korlam, One for all, all for one – The journey to Android monorepo at Uber, 2017, retrieved Jan 25, 2019 from https://speakerdeck.com/kageiit/one-for-all-all-for-onethe-journey-to-android-monorepo-at-uber.
10. D. Spinellis, Version control systems, *IEEE Software*, **22**(5): 108–109, 2005, doi: 10.1109/MS.2005.140.

11. N.N. Zolkifli, A. Ngah, A. Deraman, Version control system: A review, *Procedia Computer Science*, **135**: 408–415, 2018, doi: 10.1016/j.procs.2018.08.191.

12. S. Otte, Version control systems, 12 pages, 2009, https://www.mi.fu-berlin.de/inf/groups/ag-tech/teaching/2008-09_WS/S_19565_Proseminar_Technische_Informatik/otte09version.pdf.

13. B. Berliner, CVS II: Parallelizing software development, [in:] *Proceedings of the USENIX Winter 1990 Technical Conference*, Berkeley, USA, USENIX Association, pp. 341–352, 1990.

14. B. Cannon, B. Warsaw, S.J. Turnbull, A. Vassalotti, *Migrating from Subversion to a distributed VCS*, PEP 0374, Python Foundation, 1990, draft retrieved from http://www.python.org/dev/peps/pep-0374/.

15. I.C. Clatworthy, Distributed version control: Why and how, [in:] *Proceedings of Open Source Development Conference (OSDC)*, 7 pages, 2007.

16. B. De Alwis, J. Sillito, Why are software projects moving from centralized to decentralized version control systems?, [in:] *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, Vancouver, BC, Canada, pp. 36–39, 2009, doi: 10.1109/CHASE.2009.5071408.

17. A. Koc, A.U. Tansel, A survey of version control systems, *ICEME 2011*, 6 pages, 2011.

18. V. Kovalenko, F. Palomba, A. Bacchelli, Mining file histories: Should we consider branches?, [in:] *ASE '18: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, September 3–7, Montpellier, France, pp. 202–213, 2018, doi: 10.1145/3238147.3238169.

19. E.T. Barr, C. Bird, P.C. Rigby, A. Hindle, D.M. German, P. Devanbu, Cohesive and isolated development with branches, [in:] *FASE'12: Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, pp. 316–331, 2012, doi: 10.1007/978-3-642-28872-2_22.

20. W. Zou, W. Zhang, X. Xia, R. Holmes, Z. Chen, Branch use in practice: A large-scale empirical study of 2,923 projects on GitHub, [in:] *Proceedings of the 19th IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Sofia, Bulgaria, pp. 306–317, 2019, doi: 10.1109/QRS.2019.00047.

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

22. U. Shakikhanli, V. Bilicki, Multi repository management tools, *The Journal of CIEES*, **2**(2): 13–18, 2022, doi: 10.48149/jciees.2022.2.2.2.

23. GitHub, github/renaming: Guidance for changing the default branch name for GitHub repositories, https://github.com/github/renaming.

24. U. Shakikhanli, V. Bilicki, Machine learning model for identification of frontend and backend repositories in Github, *Multidisciplinary Science Journal*, **5**: e2023ss0106, 2023, doi: 10.31893/multiscience.2023ss0106.

Received June 6, 2023; revised version December 7, 2023;
accepted December 7, 2023; published online February 1, 2024.