The Journal of CIEES Volume 2, Issue 2, December 2022, pp. 13 – 18, ISSN: 2738-7283 (Print), 2738-7291 (Online)
DOI: 10.48149/jciees.2022.2.2.2

Multi Repository Management Tools

Ulvi Shakikhanli [1*] and Vilmos Bilicki [1]
[1] Doctoral School of Computer Science, Faculty of Science and Informatics, University of Szeged, H-6720, Szeged, Hungary

*E-mail of corresponding author: ulvi@inf.u-szeged.hu

Abstract: Multi repository is one of the most preferred structures in the development process. There are numerous tools available for managing Multi Repo projects. The use of these tools can affect the development period and productivity. In the first place, this paper explains Version Control Systems and the concept of branching together the most essential tools in this field. The second, the fingerprints or signatures of some of those tools are described for identifying repositories. Specific measurements have been calculated according to those tools and gave us a clearer view of the usage index of those tools. The study of actual data in the paper yields exact conclusions for the most essential features for Version Control Systems tools.

Keywords: Multi Repository Management, Version Control System, Github Mining, Development Productivity

Received 18 September 2022
Accepted 18 November 2022
Published 20 December 2022

1. Introduction

Multi Repository structures are popular among developer teams and companies. As a basic definition, it can be explained that the Multi Repository structure manages all packages and source code components in several repositories [1]. This allows each group of developers to focus on their part of the project and work independently. However, this approach introduces some difficulties. Since the project is separated into different modules, some tasks like testing, debugging, versioning, and even sometimes launching can create serious problems, which will delay the development. Comparison of the Multi Repository approach with Mono and other aspects of the Multi Repository structure was analyzed in detail in [2].

To overcome these and other related problems, software developers use Multi Repository Management Tools (MRMT). There is no unique description for this concept. The general definition is provided her for MRMT based on its main characteristics. MRMT is a software or service provider, which stores projects in one or more repositories and makes developer team collaboration possible. It is evident that MRMTs have much more capabilities than these two, but all of them are based on a basis of these two basic notions. Name storage of the project and help the developer team to work together.

Among the MRMTs, Github is the most popular one, and other tools may be split in two main categories: Major and Minor Multi Repository Management tools. Major tools have their own local server for storing repositories, and can work independently from Github. However, Minor tools have very limited feature capacity and cannot work without Github. This paper focuses on the characteristics of these tools and present new findings about their general use. It is worth mentioning that there are dozens of such tools, but this paper will focus only on the most popular and promising ones. The followings are main questions which paper focuses on:

RQ 1: *What is version control?*
RQ 2: *What are the main features of Github?*
RQ 3: *Which VCSs have apparent advantages over Github?*
RQ 4: *Which tools increase Multi Repository Management process of Github, and what are their signatures?*
RQ 5: *What are the main features of Minor MRMTs?*

2. Literature review

Most of the information about Multi Repository management tools in the literature is present in different internet forums or websites rather than in academic papers. Most of the Version Control Systems (VCSs) may also be treated as MRMTs. In short, it can be said that VCSs permit the acceleration and simplification of the software development process, and allow new workflows [7]. There are two main approaches in VCSs namely, centralized and distributed. The centralized Version Control System (CVCS) stores the development history in a central server, while the Distributed one (DVCS) does the opposite; that is, it saves copies of repositories in all machines [8]. Each of them has its pros and cons. This is why the choice of what type of VCS should be used can affect the whole development period as regards the size and intensity of commits. Another article [9] widely explains at which level this can occur. However, the authors only briefly mention a few most popular tools, and this paper gives a more detailed review here. There ares some of them, which had to be mentioned, for example, Github [10] and Mercurial [11]. These tools will be discussed in later sections in a more detailed way, but it has to be mentioned that Github has a

© The Author(s). Published by CIEES Association Press. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 4.0 Generic License.

considerable role in the VCS field, with a market share of nearly 85% [12]. In this paper, Github Mining will be implemented to answer the previously mentioned questions. However, it is not an easy task. Since Github hosts millions of repositories, there are several cons about doing mining among them [13]. However, as described in the mentioned paper this problem can be avoided by implementing additional steps in presented algorithm.

3. Differences between mono and multi repository approaches

Starting from small individual projects to huge cooperative applications, the first task of developers is to choose the structure of the project. There are currently two main approaches to this. They are called Mono Repository, and Multi Repository approaches. Some big companies like Google have a long history of using the Mono Repository approach for their projects. Still the study described in [3] gives a clear view of the advantages and disadvantages of both methods. A comparison can be made according to the three main features of both methods, as provided in [3].

Visibility – According to the survey among Google developers, this may be considered the main advantage of having a Mono Repository structure. Developers mentioned the need for code to be accessible to all of them, which allows them to use the same code for all components and use the same development approach. Of course, this is also possible in the Multi Repository approach. Still, since each part of the project is located in a different repository, special permission has to be given.

Choice of developer tools, consistent style, and toolchain – As also mentioned in the article Google has created its own tools. Since the whole project is in a unique repository, this means that all the developers have to use the tools, which were created or recommended by Google. It is also regarded as one of the greatest tradeoffs of the Mono Repository approach. Developers mentioned in the survey that sometimes they were forced to use tools, that they were not familiar with or comfortable. It may be beneficial to use a unique style and development language for the whole project, but it restricts the freedom of the developers, and this can have adverse effects.

A version of dependencies – This is quite similar to above-mentioned themes. The Mono Repository approach forces developers to use the latest dependency and sometimes induces crashes or problems during the development phase. However, dependencies of different code portions do not vary in Multi Repository structures, and this allows developers to circumvent the diamond dependency problem.

Cognitive load issue - This issue is handled by both repository structures using different approaches. The Mono Repository approach attempts to achieve this goal by increasing the visibility of code and tools. At the same time, the Multi repo approach uses separate small codes, and this increases the build and development velocity.

Besides all these mentioned themes, both repository structure shows different behavior in several cases, which are described in [2]. In the end, it clearly can be said that the choice between two-structure types is not a solid line and has to be decided by checking different tradeoffs.

4. Version control systems

During the development period, there may be situations where developers end up with thousands of code, and in some cases, they would like to check older versions of their source code. Version Control Systems (VCSs) are widely used for this purpose. It is worth to mention that VCS are tools, that keep the tracking information of files and they access and collaborate between developers. There are two main types of VCS, namely Centralized Version Control Systems and Distributed Version Control Systems.

Centralized Version Control Systems – uses one central repository approach.

With these, a central computer or server stores all the files and versions of the project. Users can access some specific files or entire repositories to work. After finishing their work, the user has to "*push*" their changes with a commit message. After pushing these changes, other users have to "*update*" their files according to the new version of the repository. It should be mentioned that CVCS stores just the last version of a project, so other users always have to keep up to date about changes in the source code. During the development phase, users can use branches to implementing new features or functionalities. Sometimes branches may crash, but it will not affect the work process of the whole project. *Branches* can be merged into the source code after the testing and implementation of features. *Branches will be discussed* in more detail below.

There are two main disadvantages of CVCSs, namely, single-point risk and low speed. As it mentioned above, CVCS contains only one central server for storing repositories. If it goes down, the whole project will be inaccessible, and the development process will stop. In addition, using one server creates another problem during the development phase. Users have to 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.

Distributed Version Control Systems - This is a new approach for CVCS, and does not require any central server. It means that every user can have a full copy of the repository and use it without asking for permission from the main server. It also runs faster than CVCS because almost all the commands run on the local machine, so users do not need any network connection. Most of the terms described in CVCS are also applied in DVCS. One of the most significant drawbacks of this approach may be memory consumption. Since it stores the whole project, it will consume a lot more memory, but DVCS uses compression for decrease the repository size. This approach is faster, more flexible, and reliable. If one repository goes down then one of the users can upload their own version within seconds. Because of all of these properties, DVCS is used a lot in the development industry nowadays, and there are massive platforms that provide these services. These include Github and Mercury.

5. Branching

A branch can be understood as a side development process independent from the main repository. Developers create a *branch*, an isolated workspace, from a particular state of the source code. They can share this branch and work on their tasks without affecting the rest of the project and later merge (or integrate) their changes back into the main line of development [4]. It is worth to mention that not all branches have to be merged to the main development line. In some cases, branches may be created just for testing purposes. Some of the different branching strategies will be explained in the below in more detail.

6. Branching strategies

The results of a survey are presented, based on interviews of developers and what they have learned about branching in the development process. From this, the basic idea about the purpose of branching and its types can be retrieved. After that, branching strategies will be explained in detail. As described in [5], there are several types of branches and they mostly have names related to their purpose. Five of the most common strategies:

1) *Release* - This branch type is mainly used for saving specific release versions of projects;

2) *Experiment* - It contains mostly experimental codes and implementations which are not included in the release version;

3) *Feature* – It is for implementing new features. It had to be merged only after the testing and verification phases;

4) *Bug fix* – As suggested by the name, this branch is created for fixing bugs and errors in projects;

5) *Contributor* - This branch is mainly used by individual developers to separate their work before sharing it with others.

Besides all of these branching types, several clear branching strategies are used to improve productivity and the quality of collaboration during the development phase.

The article the blog [6] describes the main characteristics of the branching strategy:

- It provides a clear path for the development process from initial changes to production;
- It allows users to create workflows that lead to structured releases;
- It permits parallel development;
- It optimizes developer workflow without adding any overhead;
- It enables faster release cycles;
- It efficiently integrates with all DevOps practices and tools, such as different versions control systems.

Three main branching strategies:

Git Flow – this strategy supports multi branches for managing source code. There are two main groups for these branches:

a) Primary branches – They are divided into two parts: master, which keeps the main production code, and development, which is created for implementing the development process.

b) Support branches – They are divided into three parts: feature branches for implementing new features, a hotfix is for dealing with errors and bugs, and release, which is for preparation of release.

The advantages of this strategy are standard support among Git tools, a clear naming strategy, and being ideal while handling multiple version development. On the other hand, disadvantages can be that history becomes a bit complex and hard to read and not very useful when developers need one version of release.

Github Flow – As suggested by the name, the strategy is offered by Github itself. With this strategy, there is one main master branch, and it is used as the main development branch. Users have to create additional branches for bug and feature implementation and all of them at the end have to be merged to the main branch after successful testing. The advantage of this strategy are that it is clear and straightforward. The git history will not become more complex, as it has been mentioned above. However, simplicity may also be a disadvantage for this strategy because it is unsuitable for a difficult development process.

It should be added that besides these popular features, there might be several other strategies based on the development culture of the company and the tool used for version control.

7. Major multi repository management tools

As described in the literature review, Github is one of the most popular VCSs in the market.

7.1. Main features of Github

Several main features make it so popular. These are:

a) Collaborative coding - Since it is one of the most essential features of VCS, Github focuses on this area. It enables substantial developer teams to work together. There are several UI elements and tools for tracking the work of other developers, and this allows the whole team to work in a synchronized way. Due to this feature, developers can add comments to their and changes of other developers, which increases teamwork and development quality.

b) Automation - this feature mainly focuses on the automation of testing, building, labelling and issue labeling. Developers can add commands for testing, and other phases of automation, which saves enormous amounts of time and increases the quality of code besides keeping the build process in line.

c) Security - This feature combines several aspects of the previous two. It provides critical reviews and code scanning. Although Github hosts a vast amount of open source, projects it also has a Private repository feature which is mainly designed for companies and commercial projects.

d) Project management - As it is understandable from its name, this feature helps developers manage their repositories. It should not be mixed up with MRMT, which was presented earlier. It covers managing a single repository, its commits, and labels, and so on.

Besides these features, Github has a few more features, it provides all these almost free, and this has attracted millions of developers and created a vast developer community. Since it is so popular among developers, it stores more than 128 million public repositories. Because of this, even other big MRMTs store their repositories in Github.

VCSs like Beanstalk, Helix core, Mercurial and Gerrit have some advantages over Github. All of these tools use Github as storage, but they also have public and private servers, which allows them to store their repositories. These features are listed below:

Beanstalk - Unlike Github, it mainly focuses on private repositories, and this is why Beanstalk gets business more than Github.

Helix core - One key difference is that it is centralized and not distributed like Github. It means Helix core stores all project files in one machine so not all developers will have a copy of the whole project as they do in Github.

Gerrit - The code review is a key advantage of Gerrit over Github. Unlike Github, where users can have only one commit in a pull request, so the developer will not get confused when they come to check it.

Mercurial - The learning period for Mercurial is much simpler than Github. Mercurial is viewed by many as more secure than Github. However, it should be mentioned that the same security level can also be achieved in Github, but it takes more effort and requires more knowledge.

7.2. Main aspects of MRMTs

During analysis of the features of the above-mentioned tools, it became evident that some of them have similar aims, and this could prove helpful in identifying their main aspects. In addition, these aspects may be crucial because they highlight the weaknesses of Github.

1) *It updates several repositories at the same time* - This is essential for the majority of tools. They do it in different ways, but in the end, all of them help users to update, (by "update" it means pull, push, etc.) several repositories at once. It saves a lot of time for developers and project owners, which overall increases the development period. It can also cause some problems in big projects, but most of the tools have different approaches to reduce this theatre.

2) *It monitors the development process* - Github gives the user a detailed view of the development period and collaboration of developers among themselves. Still it cannot do the same with several repositories at the same time. Many tools were created for this purpose. They give a clear view of the contributors, work description of individual developers, commits, and so on. This information becomes crucial during the management of Multi Repository projects. The quality of this information may be different for each tool, but this is undoubtedly valuable since it is not possible to get it in Github.

3) *It manages third-party elements*. This part is not fully covered in the managing Multi Repository project, but it is helpful in this area as well. Managing dependencies and different packages may become a nightmare owing to the MR project development, and there is no proper command or tool inside the Github platform to handle it. However, the tools, which are mentioned here, can do it quickly, and they can save a lot of time for developers.

4) *It improves review and pull requests*. This feature is not common among MRMTs, but there are several tools, that do a great job and highlight the weakness of Github and other services. It is mainly used for big projects with tens of contributors. In this case, it becomes impracticable for project owners to manage all the pull requests, and perhaps not every contributor updates their projects according to new merges. Hence, this functionality seeks to overcome this issue by making the process more straightforward and more understandable.

5) *It manages the project build process*. This is quite similar to the automation feature of Github. The main difference here is that the tools focus on building several repositories at the same time to make projects compatible.

In Figure 1, it can be seen how popular these features are among users. The number of users has been calculated for each tool. It should be added that some tools may use one or more features from above list. If so, the user share of this tool will be added to all the features to maintain fairness. The features will be represented by their ordinal number given in the previous section.

Table 1. Github Multi Repository Management procedures.

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

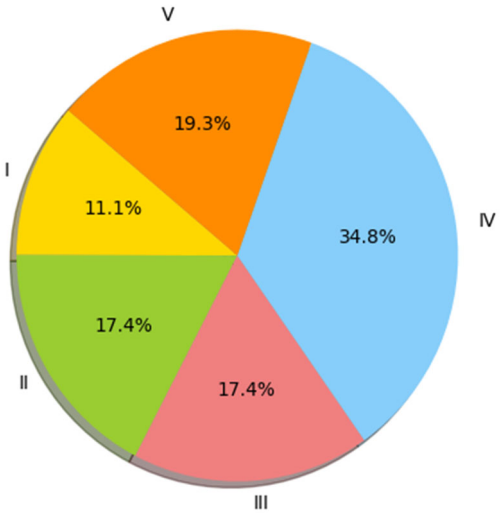

Fig. 1. Percentage of features.

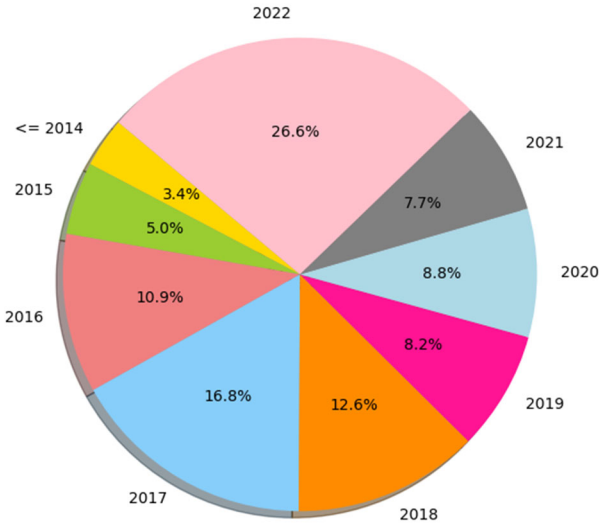

Fig. 2. The general usage percentage of tools.

Figure 1 can provide a good overview of which features are needed most for MRMTs. Nearly 35% of the users employ tools with the *Improve review and pull request feature*. This may be the greatest weakness of the Github platform since there is a considerable demand for tools like these. There are several reasons for this but the main one is the branching system. Of course, it is a key feature, but this brings several problems. For example, there may be vast amounts of branches in Multi Repository projects, and the review and pull request process can become a challenge for managers in this scenario. The percentage of the other three features is almost the same. This means that features like the *Monitoring the development process, managing third-party elements, and managing the project build process* have nearly the same importance for users, but none of them is as important as the fourth feature. The least important feature is the first one presented in the last section. It is surprising that updating all the repositories at once is not a priority for users.

These results may be helpful for a developer who wants to create a new MRMT. It offers us a clear view of the needs of the market and users, and it highlights the weakness of popular VCS and MRMTs.

8. Measurements of MRMTs

The database was created using the signatures of the tools above-mentioned and different statistics can be obtained with the help.

1) *Usage of tools over the years*.

Figure 2 provides a precise picture of the usage level. As can be seen, there was a considerable increase in both 2017 and 2022.

This paper analyzed projects only created until October of 2022 but still usage of tools in this year is significantly higher than others are. One of the main reasons for that can be increase in the usage of VCSs and different MRMTs. In addition, it seems that different weaknesses of several Major MRMTs are become more obvious, and developers rely more on 3rd party Minor tools. First years during the appearances of these tools, most of them have been developed by individual developer by non-commercial purposes, and that can be reason for low usage of these tools until 2016. Between the years 2016-2018 usage of tools increased significantly comparing previous years and that is the period where several problems with Major VCSs become more and more obvious. Starting from 2019 the usage of these tools decreased again, this is the same period when Github added new features, which more or less solved some problems, and Minor tools again lost their popularity.

All of these changes during the years shows the importance of Minor tools in this field. Some tools for example “Pull” even inspired Github to add similar features and one can say that these tools has a huge impact shaping this industry.

9. Discussion and conclusion

Here, the VCSs, MRMTs, branching, and different branching strategies have been reviewed. It should be admitted that most of these have already been presented in other research studies. Still none explained concepts like branching and versioning together and revealed connections between them as it done here. There are some excellent VCSs and MRM (Multi Repository Management) tools like Github, Mercury and Beanstalk. Still here, this paper also presented some less-known yet effective tools for Multi Repository management. They can boost productivity of development with a big community and several powerful features, but they have five main drawbacks, which are also the main features of minor MRMTs. These are the following:

i) It should update several repositories at once - Which is not possible in major VCSs, and such features can only be incorporated by implementing minor MRMTs like “Mepo”, “MR”, and “Pull”.

ii) When it comes to monitoring the development process - Github has some UI elements needed to support this feature. Still it lacks cohesion between different repositories at once, and developers need to use other tools like “Lerna”.

iii) It should manage third-party elements - Unfortunately, major MRMTs, including Github, do not pay too much attention to this topic, and leave it to the

discretion of project owners. This is why some less important tools like "Talan CLI" or "Lerna" have been presented to solve this issue.

iv) Improve review and pull request – Here, the importance of pull request and branching in the first section have been mentioned and primary tools have some disadvantages in this area too. Pull requests may become a challenge when there are tens of branches, and merge requests and this often creates bottlenecks and reduces the speed of development. Tools like "Mention-bot" and "Zappr" which are presented before could overcome this problem.

v) Manage project build process - This function usually causes problems during the build process, as suggested by its name. Tools like Github use *Automation* to solve it, but it is not effective when there are several repositories, and all of them have to be used at the same time. "DevOps & Swarm-mode" and "Mrgit" are the tools that may be useful in such cases.

Besides introducing less essential tools and their properties, this paper also presented "*signatures*" of these tools and provided sound methods for other researchers interested in this area. In the last section, two fundamental values were introduced, and they tell us something about the importance of tools and their use ratio over the past few years.

This paper has analyzed several terms like *Version Control, Branching, Branching strategies* and *Minor Multi Repository Management tools*. Main features of VCSs have been analyzed. A unique Database has been created and several analyses, which has never been shown or discussed before, have been introduced in this paper. It has been proved that Minor tools has huge impact on VCSs, and therefore has to be taken account during project development process.

Threads to validity

It has to be taken account that measurements shown in this paper are created according to the collected repositories from Github. As it has been described in previous chapters, those repositories are identified according to the *signature files* presented in repository folder. However, it is known that some tools do not necessary need some configuration tool. Even in some cases, there is no such file, and that is why identifying usage of those tools inside repository is almost not possible. In addition, this is a rapidly growing field of programming and new tools emerge almost each month. Thus, popularity ratio of tools and needs of MRMT industry can be much different in upcoming years.

References

[1] Scott, P L. (2017). *Mono-repo or multi-repo? Why choose one, when you can have both?* Medium.com, 2017. [Online]. Available at: https://patrickleet.medium.com/mono-repo-or-multi-repo-why-choose-one-whe.

[2] Shakikhanli, U., and Bilicki, V. (2022). Comparison between mono and multi repository structures, *Pollack Periodica*, vol. 17, issue 3, pp. 7-12. doi: https://doi.org/10.1556/606.2022.00526.

[3] Otte, S. (2009). *Version Control Systems*, Available at: https://www.mi.fu-berlin.de/inf/groups/ag-tech/teaching/2008-09_WS/S_19565_Proseminar_Technische_Informatik/otte09version.pdf.

[4] Rama Rao N. and Chandra Sekharaiah K. (2016). A Methodological Review Based Version Control System with Evolutionary Research for Software Processes. *In Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies (ICTCS '16)*. pp. 1–6. https://doi.org/10.1145/2905055.2905072.

[5] Brindescu, C. C., Codoban, M., Shmarkatiuk, S. and Dig, D. (2014). How do centralized and distributed version control systems impact software changes? *In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. pp. 322–333. https://doi.org/10.1145/2568225.2568322.

[6] Github. http://www.github.com/ Accessed March, 2022.

[7] Mercurial. https://www.mercurial-scm.org. Accessed March 25, 2022.

[8] Slintel. https://www.slintel.com/tech/source-code-management/github-market-share. Accessed March 20, 2022.

[9] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining GitHub, *MSR 2014: Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 92–101, https://doi.org/10.1145/2597073.2597074.

[10] Jaspan, C., Jorde, M., Knight, A., Sadowski, C., Smith, E. K., Winter, C., and Murphy-Hill, E. (2018). Advantages and disadvantages of a monolithic repository: a case study at google. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18),* pp. 225–234. https://doi.org/10.1145/3183519.3183550.

[11] Barr, E.T., Bird, C., Rigby, P.C., Hindle, A., German, D.M., Devanbu, P. (2012). Cohesive and Isolated Development with Branches. In: de Lara, J., Zisman, A. (eds) *Fundamental Approaches to Software Engineering. FASE 2012. Lecture Notes in Computer Science*, vol. 7212. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-28872-2_22.

[12] Phillips, S., Sillito, J. and Walker, R. (2011). Branching and merging: an investigation into current version control practices. *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering,* pp.9–15. https://doi.org/10.1145/1984642.1984645.

[13] *DevOps Branching Strategies Explained*: Available at: https://www.bmc.com/blogs/devops-branching-strategies/, Accessed March 22, 2022.