MONO VS MULTI-REPO

WHY THE DEBATE IS HOTTER THAN EVER

AND HOW DEVELOPERS CAN CHOOSE THE RIGHT APPROACH

WRITTEN BY

JEREMY CASTILE, VICE PRESIDENT, GITKRAKEN

CONTRIBUTIONS FROM

KEVIN BOST, SENIOR SOFTWARE ARCHITECT,
MICROSOFT MVP & GITKRAKEN AMBASSADOR

TABLE OF CONTENTS

INTRODUCTION

As developers, we're all familiar with the importance of effective code management. And if we're being honest, we're more than familiar with the topic – we're intimate. It's present in our daily lives. It's one of the first things we learn in our job, and always a question when interviewing for the next one.

Why? Because good developers are always thinking about how to balance speed, flexibility, and maintainability, especially when working in a team.

The debate around monorepos vs multiple repos has only gotten hotter in the development community. See here, here, and here.

> **...it's important to consider key factors like dependency management, code sharing, and version control.**

When evaluating the right fit for your team, it's important to consider key factors like dependency management, code sharing, version control, and continuous integration (CI). It's also important to get current – and maybe challenge your assumptions – on how the latest collaboration tools are helping to streamline the workflow for teams using either a mono or multi repository approach.

We'll dive deep into each of these topics here, and by the end, you and your team will have the knowledge and insights to be able to decide which approach is best suited for your development needs.

So, let's dive in!

A BIT OF HISTORY

First things first, let's make sure we're on the same page with our terminology. A monorepo refers to a single, unified code repository that contains all the code for an organization's projects. And using multiple repos means that each project or service has its own separate repository.

Before the advent of modern Source Control Management (SCM) and Version Control System (VCS) tools, monorepos were the standard approach. As new distributed VCS like Git emerged, along with advances in CI systems, the multi-repo model gained momentum, particularly within the open-source community.

While prominent tech giants like Google, Facebook, and Microsoft (albeit to a lesser extent) continued to embrace monorepos, showcasing their effectiveness in certain situations, other industry leaders such as Amazon, Netflix, Uber, and Coca-Cola have adopted the multi-repo approach.

As software development projects have expanded and evolved, both strategies have faced their own challenges, with the multi-repo approach specifically grappling with issues of scalability and complexity management. In response, the monorepo approach has experienced a resurgence of sorts, offering a way to unify complex codebases and simplify dependency management. However, the monorepo approach brings its own set of challenges to the table, especially when accommodating numerous developers and requiring fine-grained access control.

WHY IS THIS AN IMPORTANT ISSUE FOR SOFTWARE TEAMS?

> **Simply put, this decision can have a massive impact on productivity, collaboration, and overall development process.**

There are benefits and drawbacks of each approach. The right choice for your team will depend on factors like your anticipated project requirements, team size, technical constraints, and more.

One of the biggest concerns team leads have is about making the “right” choice from the beginning. Once you’ve committed, you’ve essentially set your path for the team and all developers will need to adopt that direction. Switching from one approach to the other may be difficult because there are often many upstream and downstream tools that are connected to the current setup. So making a switch requires careful consideration of how these tools will be affected.

Another concern is that managing multiple repos can be a real headache for developers and teams. When you have code scattered across multiple repositories, it can be difficult to keep track of changes, dependencies, and versions, which can lead to errors and inconsistencies.

For example, imagine having to update a feature that requires changes to multiple repositories. You have to clone each repository, make the necessary changes, and then push those changes back to each repository separately. This can be a tedious and time-consuming process, especially if you have to deal with merge conflicts or coordinating deployments when there are shared dependencies.

In contrast, with a monorepo, all code and assets are stored in a single repository. This means that changes can be made to the entire codebase in one place, making it easier to manage dependencies, track changes, and avoid conflicts.

Of course, it's important to remember that monorepos are not a one-size-fits-all solution. Depending on the project's size and complexity, multiple repositories may still be the best approach.

As the push towards monorepos grows, it is also important to note that it doesn't have to be an all-or-nothing approach. Sometimes a mix of both monorepos and multiple repos can be used to address the specific needs of a particular project.

CAN I EASILY SWITCH FROM ONE TO THE OTHER

A common misconception about choosing between a monorepo and multiple repository structure is that the decision is permanent. But that's not entirely true. What works best for the project now might not be the case in the future. In fact, many companies and teams eventually decide to make the change. But be warned, a "rip and replace" approach can be wasteful, time-consuming, and stressful.

The best possible solution is one that can meet current needs and also be able to adapt to future changes. Having a well-structured codebase is critical when switching repository structures. It makes the transition smoother and minimizes the risk of errors.

For instance, if a development team decides to switch from a monorepo structure to multiple repositories, a well-structured codebase would have clear boundaries between different parts of the code. This would help the team figure out which parts belong in which repository. Plus, having consistent naming conventions for modules, packages, and directories would make it easier to locate and organize files in the new repository structure.

On the other hand, if a development team decides to switch from multiple repositories to a monorepo structure, a well-structured codebase would have a clear separation of concerns between different projects or services. This would help the team identify which parts of the code belong in which package or directory in the new monorepo structure. Again, having consistent naming conventions and dependencies would make it easier to reorganize everything without breaking dependencies, reducing the risk of introducing bugs or errors during the migration process.

Switching from one repository structure to another can be a daunting task, especially when it comes to the potential loss of history in Git repositories. Losing this valuable information can set back the development process, causing unnecessary delays and frustration. However, by carefully planning the transition and ensuring you have a well-structured codebase, it's possible to retain the essential historical data and make a smooth switch to the new repository structure.

DO DEVELOPERS EVER GET TO HAVE A SAY?

When it comes to choosing the repository structure, many developers wonder if they ever get a say. In most cases, the decision comes down to the level of team autonomy within the organization. While the decision is sometimes made by team leads or lead architects, when a company has a culture that emphasizes team autonomy, developers are usually given the freedom to make decisions about the repository structure, or at a minimum, are invited to participate in the conversation.

On the flip side, some organizations have a top-down command and control approach where all decisions are made by the CTO or other top executives. In such organizations, developers don't get a say in the repository structure and have to work with whatever is dictated to them.

While the decision of whether developers get to choose the repository structure depends on the level of team autonomy, the end goal should always be the same: success for the business. By fostering a culture of collaboration and input, organizations can make the best decisions for their project and their teams.

COMPARISION GUIDE

MONOREPO

{ *Mono* : **Single** } { *Repo* : **Repository** }

PROS +

- Atomic commits
- Make changes affecting multiple services at once
- Structural simplicity
- Common directory structure
- More efficient code review process

CONS -

- Requires smarter build scripts
- Large numbers of PRs can pile up
- Can create bottlenecks
- Can be time-consuming and costly.
- Can be challenging to navigate and find things

MULTI-REPO

{ *Multi* : **Multiple** } { *Repo* : **Repository** }

PROS +

- Simplified development process
- Logical and technical barriers in the codebase
- Gets code into production faster
- Reduces conflict between developers
- Faster build times and releases

CONS -

- Can be overwhelming
- Difficult to manage changes across repositories
- Difficult to coordinate changes across multiple teams
- More challenging collaboration and communication
- Debugging can be time-consuming and frustrating
- Can impact developers ability to focus

GET THE BEST OF BOTH WORLDS

Try GitKraken Client for Free

WHAT ARE THE BENEFITS OF USING A MONOREPO?

As we mentioned in the introduction, monorepos have become especially popular among organizations that develop many software products because they can manage all their code in one place. One of the key benefits of using a monorepo is the ability to create atomic commits. Atomic commits allow for grouping a set of changes together into a single commit, making it easier to track changes and roll them back if needed. Additionally, with all the code in one place, changes can be made that affect multiple services at once, resulting in significant time savings.

Another advantage of using a monorepo is its structural simplicity. By keeping things that often change together in one place, similar to the .NET structure that can have a single solution file with many projects in it, a monorepo makes it easier for developers to manage and maintain their code. The common directory structure provided by a monorepo enables developers to work more efficiently and ensures that all changes are made in one central location.

Furthermore, a monorepo simplifies code sharing. With everything stored in a common directory structure, sharing code via project references or file linking becomes much easier. It is also easier to review the changes as a whole. Instead of having to sift through many smaller changes spread across multiple repositories, reviewers can easily review one big change, leading to a more efficient code review process.

WHAT ARE THE CHALLENGES OF USING A MONOREPO?

Using a monorepo also comes with its own set of challenges. While having all your code in one place might seem like a good idea, it can quickly become overwhelming. One major challenge with monorepos is managing build systems. Since build systems trigger on changes, you need to ensure that small changes don't trigger massive builds. This means build scripts have to be smarter, and you have to navigate different subfolders, so builds can become more complicated.

Setting up CI tools for a monorepo can also be difficult since all pull requests (PRs) are in one place, making collaboration and organization challenging. With everything in one repo, a large number of PRs can pile up, requiring you to tag and organize them for clarity. Additionally, if projects need to evolve independently, this becomes challenging with a monorepo. For example, editing a piece of shared code may trigger the build pipeline to build everything, causing bottlenecks that slow things down. This can result in a snowball effect that only gets heavier.

Dependency management is another issue with a monorepo. Upgrading to a new version of a dependency requires everything across the codebase to switch, which can be time-consuming and costly. If it's only needed for a small fix, it can feel like a high cost. Additionally, dealing with the large size of the repo can create friction for developers, making it challenging to navigate and find things.

For instance, in a large monorepo with hundreds of developers where everything changes frequently, teams need a solution to filter out unnecessary noise and focus on their specific tasks.

WHAT ARE THE BENEFITS OF USING MULTIPLE REPOSITORIES?

The multiple repository approach involves dividing code across – you guessed it – multiple repositories. This approach offers a range of benefits, including a simplified development process, logical and technical barriers in the codebase, more efficient deployments, and improved collaboration.

One of the primary advantages of the multi-repository approach is that it simplifies the development process. By breaking down the code into smaller chunks, developers can tackle even the largest systems without feeling overwhelmed. This approach is particularly useful for microservices, where each team or developer might be responsible for just one microservice, as well as applications with reusable libraries in separate repositories. And, when the development and test environments are suitably automated for deployments, having a multiple repository structure allows for faster build times, releases, and overall project progress.

Using separate repositories helps create logical barriers both from a technological and a team standpoint, creating a separation of concerns that is especially beneficial for microservices. This approach also makes it easier to manage notifications, which is often a shortcoming of Git hosting providers not allowing for a more fine-grained approach. A multiple repository structure helps reduce the number of PRs for areas that don't change often, which in turn makes it easier to isolate changes to a codebase as well as information about these changes that a developer is subscribed to.

In addition, a multiple-repo approach simplifies deployment since changes to a particular repository trigger CI to run, allowing developers to get their code into production faster. Another significant benefit of a multi-repo structure is speed, as the smaller build runs more quickly when making changes.

Finally, a multi-repo approach is also useful for collaborative projects, as it reduces the risk of conflicts between developers. A large monorepo might seem like a good idea at first, but it can quickly become overwhelming and challenging to manage. The multi-repo approach sets up clear boundaries, as mentioned above, so developers do not accidentally stray into where other developers are working, leading to more efficient and productive work. Overall, a multi-repo approach is a great way to simplify and streamline development projects, particularly those with multiple services or teams.

WHAT ARE THE CHALLENGES OF USING MULTIPLE REPOSITORIES?

Using multiple repositories can present several challenges. Firstly, managing multiple repositories can be overwhelming, especially when there are hundreds of them to keep track of. It can be challenging to know which repositories to access to find the required code, leading to disorganization and difficulties in managing changes across repositories. This often results in significant developer overhead.

Another issue with using multiple repositories is coordinating changes across multiple teams. This can become especially problematic when work is being done across many different parts of the codebase. For example, if one team makes a change that affects another team's code, it can cause significant delays and require extensive coordination to ensure everything remains in sync. It can also result in a large number of PRs and merge conflicts, making it challenging to keep everything up to date and working correctly.

This also means collaboration and communication can become more challenging when working with multiple repositories. When each team is working in their own repository, it can be tough to keep everyone on the same page and ensure that changes are being made correctly. This can lead to a lack of visibility and difficulty in coordinating work, particularly when multiple teams are working on different parts of the same project.

Another point here is that often when changes require coordination across teams, the features need to be well planned out and executed. Team A may have to deploy a version of an API that just returns mock data simply so that Team B can start coding against it. It is this level of planning that is often missed between teams.

Debugging can also be a significant challenge associated with a multi-repo structure. In the event of an issue, it might be necessary to examine several microservices to identify the source of the problem. This can be frustrating and time-consuming, making it difficult to understand the root cause of bugs. This is one reason why some teams prefer using monorepos, as it's easier to manage changes and keep track of everything when all the code is in one place.

While there are solutions available to help developers manage the challenges associated with using a multi-repo structure, many of these solutions fall short in separating the signal from the noise. One major challenge, as we covered above, is the sheer volume of changes and notifications that can pile up when working across multiple repositories. Tools like Graphite and "PR stacking" can help streamline the process of making changes and managing pull requests, but they may not provide a complete solution. Developers need to be able to filter out unnecessary noise and focus on their specific tasks to be productive. Unfortunately, many existing solutions do not offer this capability, making it difficult for developers to manage complex codebases efficiently. Without effective tools to separate the signal from the noise, developers can quickly become overwhelmed and find it difficult to stay organized and focused.

GITKRAKEN WORKSPACES: COLLABORATIVE CODING REDEFINED

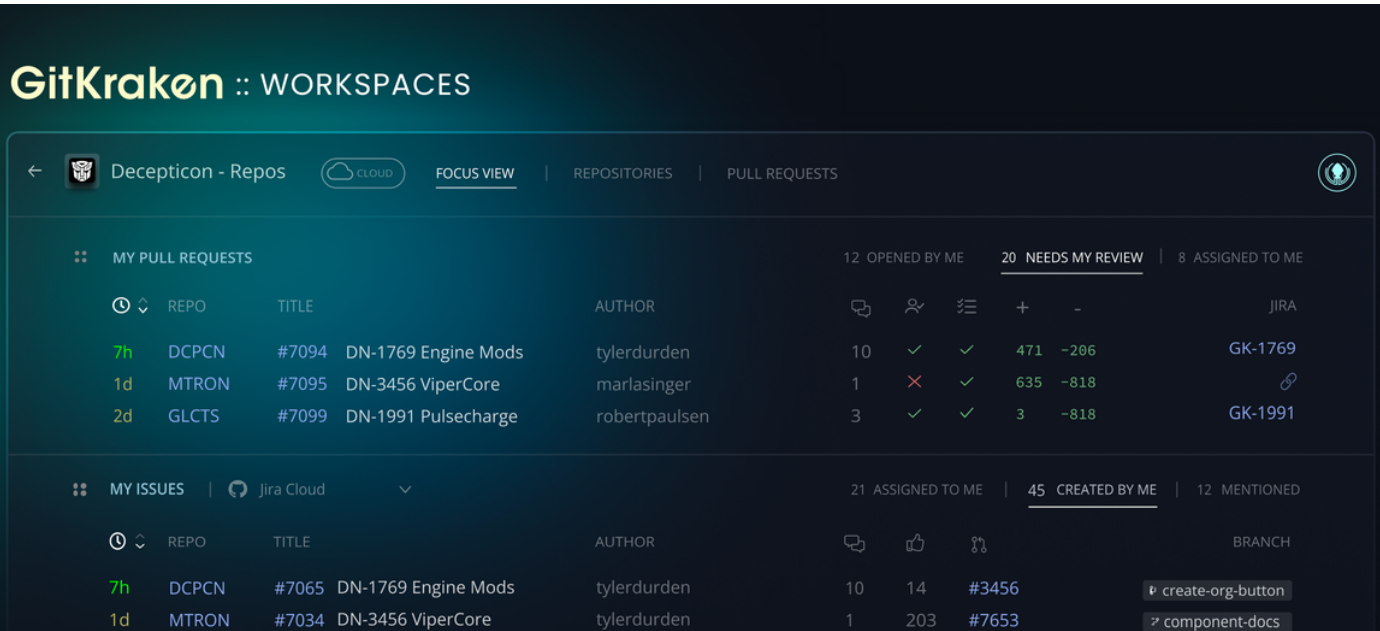
As we've explored throughout this ebook, working with multiple repositories can be a challenge, and adopting a monorepo structure is not always the best solution. That's where GitKraken Workspaces can help to fill the gap, offering a more flexible and efficient approach to organizing your development work.

With GitKraken Workspaces, you can group together all the repositories you're working on and aggregate all the pull requests, issues, and other important information coming from those repos in one place. This can simulate some of the benefits – without inheriting the drawbacks – of using a monorepo while maintaining the flexibility of a multi-repo approach. Workspaces allow you and your team to streamline collaboration and communication while maintaining a clear and efficient codebase.

One of the key advantages of Workspaces is their flexibility. You can group things together dynamically, change your Workspace, or create a new one if your needs change. You can also have many different Workspaces that slice things in different ways, and you can do this all on your own if you want to or share them with your team. Workspaces allow you to connect with multiple systems, including Jira issues, making your Workspace more efficient to work in.

Onboarding new developers is also easier with Workspaces. You can share your Workspace with new developers, and they can quickly get up and running by cloning all the repos in the Workspace. Plus, you can share Workspaces with your team and use the Team View, which shows all pull requests and issues for the repositories in your Workspace – giving you a high-level view of your team's coding efforts. Workspaces let you see changes and pull requests across all the repos you need to be paying attention to, and adding or removing repos from your Workspace is easy and flexible.

Overall, GitKraken Workspaces offer a flexible and dynamic approach to managing your development work, and they're a great alternative to using a monorepo structure. You can try Workspaces for free by downloading GitKraken Client here.



CONCLUSION

The choice between using a monorepo or multiple repositories for your development projects is an important decision that can greatly impact your team's workflow. Both approaches have their benefits and challenges, and the right choice for your team will depend on a variety of factors. However, it's worth noting that switching from one approach to another is not impossible, but should be done with careful planning and consideration.

It's important to involve developers in the decision-making process and ensure that everyone's opinions are heard. Finally, tools like GitKraken Workspaces can help streamline your team's workflow and improve collaboration, regardless of which approach you ultimately decide to go with. We hope this ebook has provided you with the insights you need to make an informed decision for your team's development needs.

QUESTIONS? COMMENTS?

CONNECT WITH THE AUTHOR

JEREMY CASTILE, VP, GITKRAKEN
linkedin.com/in/jeremycastile/