IJIRCCE

e-ISSN: 2320-9801 | p-ISSN: 2320-9798

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN COMPUTER & COMMUNICATION ENGINEERING

Volume 11, Issue 11, November 2023

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

Impact Factor: 8.379

International Journal of Innovative Research in Computer and Communication Engineering

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

Repo Refactor: A Smart Guide for Engineers to Split Monorepo Efficiently

Twinkle Joshi

Dynatrace, Michigan, USA

ABSTRACT: Monorepos, which consolidate multiple projects into a single repository, have gained popularity for their streamlined code sharing, centralized dependency management, and simplified CI/CD workflows. However, as monorepos scales, they introduce significant challenges, including performance bottlenecks, complex dependency resolution, and CI/CD inefficiencies. This article explores effective techniques for splitting monorepos into smaller, manageable repositories, such as using git filter-repo, git subtree, and manual extraction methods.

The article also includes a detailed step-by-step guide for splitting repositories. A real-world case study demonstrates how decoupling a React and Angular application resolved dependency conflicts, enhanced scalability, and improved CI/CD workflows. Additionally, it outlines actionable recommendations for assessing monorepo suitability, defining clear boundaries, optimizing CI/CD, and managing team transitions. By identifying common gaps in current practices—such as inconsistent dependency management, limited cross-repository CI/CD integration, and insufficient tooling—this paper offers practical solutions and best practices to help engineering teams optimize their repository strategies effectively.

KEYWORDS: Monorepo, Repository Splitting, Repository refactoring, Version Control, Code Decoupling, Scalability, Performance Optimization, Commit History Preservation, Repository Architecture, Git Operations, Modular Development, Codebase Management, Development Workflow, Repository Performance, Monorepo Challenges, Repository Migration, Dependency Conflicts

I. INTRODUCTION

1.1. Monorepos Unveiled: Benefits, Challenges, and Strategic Insights

What is a Monorepo?

A monorepo is a version control strategy that houses multiple projects or applications within a single repository. It is distinct from polyrepos (where each project resides in a separate repository) and monoliths (a single, tightly coupled application). Monorepos enable streamlined code sharing, cross-project refactoring, and deployment flexibility, making them applicable across various development scenarios, such as frontend, backend, and infrastructure projects.

Advantages of Monorepos

- **Code Reuse**: Encourages code sharing and reduces duplication, ensuring consistency across projects.
- **Collaboration**: Simplifies developer access to multiple projects within the same repository.
- **Dependency Management**: All projects use a single package manager and version control system.
- **Atomic Commits**: Monorepos enable atomic commits and releases, ensuring that changes across multiple projects are always in sync.
- **Improved Testing**: Allows simultaneous testing across all projects for comprehensive quality assurance.

Disadvantages of Monorepos

- **Complexity**: It can become complex as the codebase grows, making it challenging to navigate and understand.
- **Slower Cloning**: Initial repository cloning can be time-consuming, especially for large repositories.
- **Specialized Tooling**: Requires complex tooling to handle dependencies, builds, and versioning.
- **Risk of Over-Engineering**: Integrating unrelated projects can lead to unnecessary complexity.
- **Performance Issues**:
  - **Commits**: A large number of commits slows operations like git log or git blame.
  - **Refs**: A high number of refs in a monorepo can significantly impact Git performance, especially during ref advertisements and operations like git clone, git fetch, and git push.
  - **Tracked Files**: The .git/index keeps track of all files, influencing the outcomes of commands like git status and git commit.
  - **Large Files**: Media assets impact repository performance during cloning and updates.
- **Tight Coupling**: Encourages brittle software that makes abstraction mistakes harder to fix.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

- **Scalability**:Managing millions of lines of code and rapid submissions (e.g., thousands of weekly commits) places significant demands on version control systems.

When to Use Monorepos
Monorepos are suitable for:
- **Shared Codebases**: Projects with substantial shared dependencies and resources.
- **CI/CD Pipelines**: Centralized build, test, and deployment processes.
- **Collaborative Teams**: Development teams working closely on interconnected projects.
- **Cross-Project Refactoring**: Frequent refactoring across multiple applications.

When to Avoid Monorepos
Monorepos are not ideal for:
- **Small Projects**: Independent applications without shared dependencies.
- **Independent Teams**: Teams with minimal collaboration or code sharing.
- **Security Concerns**: Scenarios requiring granular access controls.
- **Legacy Code**: Codebases difficult to refactor or integrate.
- **Performance Concerns**: Large repositories that impact overall scalability.

Mitigation Strategies
To address challenges in large monorepos:
- **Remove Refs**:
  - Delete unused branches/tags to declutter repositories while maintaining commit history.
- **Improve File Management**:
  - Use fast local storage and advanced techniques like file system notifications for efficiency.
- **Use Git LFS**:
  - Store large files externally as placeholders, reducing repository size and cloning impact.
- **Split the Repository**:
  - Break monorepos into smaller, focused repositories by identifying logical boundaries, such as modules with similar release cycles.

Arguments Against Monorepos (Matt Klein's View)
Matt Klein critiques monorepos, emphasizing that:
- Monorepos provide no unique benefits at scale compared to polyrepos.
- Tight coupling leads to brittle software and complicates collaboration with open-source projects.
- Scaling version control for large monorepos is highly resource-intensive and unnecessary, as similar tools are required for polyrepos.

1.2. Rationale for Splitting a Monorepo
While monorepos offer advantages like code sharing, centralized dependency management, and cross-project refactoring, their scalability challenges often justify splitting into smaller repositories. Here are the key points summarized:
- **Scalability Concerns**:
  - **Performance Issues**: Large repositories can suffer from slow operations (e.g., cloning, commits, git log) due to the sheer volume of tracked files, branches, and commits.
  - **Codebase Complexity**: Managing millions of lines of code becomes increasingly challenging, particularly with rapid and frequent commits.
- **Team Autonomy & Decoupling**:
  - **Tight Coupling Risks**: Monorepos can result in brittle software, where unrelated projects are unnecessarily linked, increasing the likelihood of abstraction mistakes.
  - **Independent Development**: Splitting allows teams to work independently, avoiding bottlenecks tied to unrelated code or teams.
- **CI/CD Optimization**:
  - **Tailored Pipelines**: Separate repositories enable customized workflows and pipelines for specific projects, improving build and deployment efficiency.
- **Mitigating Monorepo Challenges**:
  - **Simplified Tooling**: Smaller repositories reduce the need for complex dependency management tools.
  - **Improved Performance**: Removing unused refs, splitting into focused modules, or using external storage (e.g., Git LFS for large files) may ease some challenges but splitting often becomes the more practical long-term solution.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

- **When Monorepos Fall Short**:
  - **Independent Projects**: If projects have minimal shared dependencies or development overlap, splitting provides clearer boundaries.
  - **Security and Access Control**: Splitting repositories offers finer-grained permissions compared to the all-encompassing access required in a monorepo.
- **Community Critiques (e.g., Matt Klein)**:
  - Monorepos don't scale efficiently at very large sizes and offer no unique benefits compared to polyrepos. Splitting repositories avoids the intensive resource demands of managing a massive, centralized repository.

References:

- https://medium.com/ableneo/monorepo-pros-cons-tools-2e6f86939be1
- https://medium.com/@mattklein123/monorepos-please-dont-e9a279be011b
- https://www.atlassian.com/git/tutorials/monorepos

II. PROBLEM STATEMENT.

2.1. Challenges specific to splitting repositories from a monorepo

Splitting repositories from a monorepo can be a complex process, and several challenges are commonly encountered:

- **Preserving Commit History**:
  - Extracting a subdirectory while retaining its full Git history can be tricky. Tools like git filter-repo or git subtree help, but ensuring accuracy and completeness requires careful handling.
- **Branch and Tag Discrepancy**:
  - The new repository often doesn't include all branches and tags from the original monorepo, which can lead to missing contextual data for some developers.
- **Dependency Management**:
  - Monorepos often include shared dependencies across projects. After splitting, managing and resolving these dependencies for the new repository can be challenging and may require additional setup.
- **CI/CD pipeline**:
  - Monorepos typically use centralized CI/CD pipelines. Post-split, these systems may need significant restructuring to accommodate the new repository.
- **Effort Coordination**:
  - Teams working on different parts of the monorepo may need to coordinate efforts to ensure smooth transitions, especially if the split affects collaborative workflows.
- **Breaking Changes and Integration**:
  - Ensuring compatibility between the split repository and the remaining monorepo components can lead to breaking changes. Testing and integration need to be handled meticulously.

2.2. Gaps in current knowledge or practices.

Here are potential gaps in current knowledge and practices regarding monorepos and repository splitting:

- **Dependency Management Challenges**
  - **Complex Dependency Resolution**: There is limited standardized guidance on resolving intricate dependency conflicts in monorepos when applications rely on different library versions.
  - **Best Practices**: While splitting repositories alleviates this, there is a lack of documented best practices for handling dependencies post-split, especially for shared libraries.
- **Optimal Repository Design**
  - **Granularity**: There's little consensus on how to determine the optimal level of granularity when splitting repositories—e.g., whether to split by projects, modules, or teams.
  - **Architecture Evolution**: How to transition from a tightly coupled monorepo architecture to a modular design without significant downtime or disruptions remains a challenge.
- **CI/CD Pipelines**
  - **Cross-Repository Integration**: The gap lies in efficient methods to coordinate CI/CD pipelines across multiple repositories, especially for projects that still share interdependent components.
  - **Tooling**: Limited tools exist to manage holistic workflows that span multiple repositories, causing inefficiencies during deployment.
- **Resource Allocation**
  - **Team Adaptation**: Transitioning to polyrepos requires adjustments in team workflows. However, there is limited research on how teams can efficiently adapt, retrain, or reallocate their focus after the split.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

- **Operational Costs**: The additional cost of managing multiple repositories, such as infrastructure, tools, and manpower, is often overlooked in practice.

- **Design System Updates**
  - **Compatibility Across Apps**: In cases of different design systems, there is insufficient documentation on how design systems can be harmonized or transitioned during a repository split without disrupting development cycles.
  - **Tracking Changes**: Tools or methods for keeping design updates consistent across repositories remain underdeveloped.
- **Scalability Limitations**
  - **Scaling Decisions**: While splitting helps scalability, the criteria for determining when a repository split is necessary versus investing in monorepo optimization is not well-documented.
  - **Version Control Performance**: Techniques to improve performance in large repositories without resorting to splits are still an area for exploration.
- **Cultural and Collaboration Changes**
  - **Team Communication**: Splitting repositories may create silos and reduce collaboration. Best practices for maintaining clear communication channels post-split are not widely explored.
  - **Knowledge Retention**: There is minimal guidance on how to retain and share knowledge across decoupled teams working in polyrepo structures.

Filling these gaps could involve more research, case studies, and tooling advancements to better support developers in optimizing their repository strategies.

III. TECHNIQUES, APPROACHES AND STEPS

3.1. Existing techniques for splitting repos.

Here are some common techniques for splitting repositories:

1. **Using git filter-repo (recommended method)**:
   a. This is a modern and efficient tool for filtering repository history.
   b. Allows you to extract specific folders or subdirectories and turn them into a standalone repository.
   c. Ideal for keeping commit history intact.
2. **Using git subtree**:
   a. A built-in Git feature that simplifies splitting a repository.
   b. Supports extracting a subdirectory while preserving commit history.
   c. Requires less manual setup than other methods..
3. **Manual extraction using git clone and selective commits**:
   a. Clone the original repository and manually move the desired files or folders to a new repository.
   b. This method doesn't retain the original commit history unless combined with advanced filtering tools.

Different methodologies are suited for varying levels of complexity and requirements, such as preserving history, branches, and tags.

References:

- https://github.com/newren/git-filter-repo
- https://medium.com/@jeevansathisocial/extract-subdirectory-from-git-repository-without-losing-history-3de8aed359a4#:~:text=The%20%60git%20subtree%60%20command%20provides,%60%20%E2%80%94%20prefix%60%20option

3.2. Template and considerations for splitting repo

Splitting a monorepo into two separate repositories requires careful planning to ensure a smooth transition and minimal disruption to the development workflow. Here's a step-by-step guide, including commands and tips for updating the CI/CD pipeline:

Step 1: Analyze and Plan

1. **Identify the split boundaries**: Determine which parts of the monorepo will go into each new repository.
2. **Dependency mapping**: List all dependencies between the components to ensure nothing breaks after the split.
3. **Backup the monorepo**: Create a backup to avoid accidental data loss.

Step 2: Create New Repositories

Create new repositories in your desired platform (e.g., GitHub), create a new repository manually through the UI or via commands.

Step 3: Split the Monorepo

1. Use repo splitting techniques to extract specific directories
2. Ensure the commit history is preserved in the new repositories.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

Step 4: Update CI/CD Pipelines

1. **Duplicate the existing pipeline configuration**: Copy the pipeline YAML or configuration files to the new repositories.
2. **Modify repository references**:
   - Update the repository paths in the CI/CD configuration to point to the new repositories.
3. **Adjust triggers**: Ensure the pipelines are triggered only for relevant branches or directories.
4. **Test the pipelines**: Run the pipelines to verify that builds, tests, and deployments work as expected.

Step 5: Communicate and Transition

1. **Notify the team**: Inform developers about the changes and provide updated repository URLs.
2. **Update documentation**: Include instructions for cloning and working with the new repositories.
3. **Monitor and resolve issues**: Address any problems that arise during the transition.

By following these steps, you can split a monorepo into two repositories while maintaining a seamless developer experience.

3.3. Ways to split repos using various techniques and addressing some challenges:

3.3.1 Do not preserve history in new repo

Step 1: Create a New Repository

- Use GitHub or another platform to create an empty repository for the new code.

Step 2: Clone the New repo

- Clone the new repo to your local machine:

```
git clone <new-repo-url>
```

Step 3: Copy the Desired Directory

- **cd** to parent folder where both repo exists
- Use **xcopy** to copy the specific directory to a new folder.

```
xcopy <directory-to-split> <new-repo-folder> /E
```

  - /E: Copies all subdirectories, including empty ones.
- Examples:

1. Copy entire repo to new-repo

```
xcopy original-repo new-repo /E
```

2. Copy specific project

```
xcopy original-repo\project-a new-repo\project-a /E
```

```
Output:
Does new-repo\project-a specify a filename or directory name on the target
(F = file, D = directory)? D
(hit 'D' if you want it to be copied as directory)
```

3. Copy files only from one project. Let's take an example: files from 'project-a'

```
xcopy original-repo\project-a new-repo /E
```

Step 4: Navigate to the New Repository Folder

```
cd <new-repo-folder>
```

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

Step 5: Push the changes to New repository

- Initialize Git in the new directory:

```
git init
```

- Add the files to the repository:

```
git add .
```

- Commit the files:

```
git commit -m "<commit-message>"
```

- Push the code to the remote repository:

```
git push -u origin main
```

References:

- https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/cc771254(v=ws.11)

3.3.2. Preserve history in new repo - detailed step-by-step guide for copying all branches and tags from an original repository to a new repository:

Option 1: Copy selected Branches and Tags (Standard Approach)

Step1: Clone the Original Repository into a New Directory:

```
git clone <url to original-repo> new-dir
```

Step 2: Move into the New Directory:

```
cd new-dir
```

Step 3: List All Branches in the Original Repository:

```
git branch -a
```

Step 4: Checkout Each Branch You Want to Copy:

```
git checkout branch-name
```

Step 5: Fetch All Tags from the Original Repository:

```
git fetch –tags
```

Step 6: Verify Your Local Tags and Branches:

```
git tag
```

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

```
git branch -a
```

Step 7: Remove the Link to the Original Repository:

```
git remote rm origin
```

Step 8: Link to the New Repository:

```
git remote add origin <url-to-new-repo>
```

Step 9: Push All Branches to the New Repository:

```
git push origin --all
```

Step 10: Push All Tags to the New Repository:

```
git push --tags
```

Step 11: To allow main to track origin/main (this is needed to set up the tracking if you add a remote to an existing repository)

```
git branch --set-upstream-to=origin/main main
```

Option 2: Clone Entire Repository (Short Approach)

To copy the entire repository, including all branches and tags, with fewer steps:

Step 1: Clone the Original Repository Using --mirror:

```
git clone --mirror <url-to-original-repo> new-dir
```

Step 2: Move into the Temporary Directory:

```
cd new-dir
```

Step 3: **Link to the New Repository**:

```
git remote set-url origin <url-to-new-repo>
```

Step 4: Push Everything (Branches and Tags) to the New Repository:

```
git push origin --mirror
```

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

Step 5: To start using the new mirrored repo

```
git clone <url-to-new-mirrored-repo>
```

References:

- https://www.atlassian.com/git/tutorials/git-move-repository

3.3.3. Preserve history of only folders moved to new repo

Step1: Ensure you have Git version 2.22.0 or later.
Step2: Clone the Original Repository into a New Directory:

```
git clone <url to original-repo> new-dir
```

Step 3: Navigate to the cloned repository:

```
cd <new-repo-folder>
```

Step 4: Install git-filter-repo (if not already installed).

- To filter out the subfolder from the rest of the files in the repository

```
git filter-repo --path FOLDER-NAME/
```

  - **FOLDER-NAME:** The folder within your project where you'd like to create a separate repository.
  - The repository should now only contain the files that were in your subfolder(s).
- to set the subfolder as the root of the new repository:

```
git filter-repo --subdirectory-filter FOLDER-NAME
```

Step 5: Create a new repository on GitHub and copy its remote URL.
Step 6: Add the new repository as a remote:

```
git remote add origin <url-to-new-repo>
```

Step 7: Verify the remote URL:

```
git remote -v
```

Step 8: Push the changes to the new GitHub repository:

```
git push -u origin <branch-name>
```

Note: The new repository will retain the folder's history but won't include the original repository's branches and tags.
References:

- https://docs.github.com/en/get-started/using-git/splitting-a-subfolder-out-into-a-new-repository

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

3.4.4. Copy commits after splitting repo

Step 1: Add the old repo as a remote repository

```
git remote add <oldrepo> <url-to-old-repo>
```

Step 2: Get the old repo commits

```
git remote update
```

Step 3: Examine the whole tree

```
git log --all --oneline --graph --decorate
```

Step 4: Copy (cherry-pick) the commits from the old repo into your new local one

```
git cherry-pick sha-of-commit-one
git cherry-pick sha-of-commit-two
git cherry-pick sha-of-commit-three
```

Step 5: Check your local repo is correct

```
git log
```

Step 6: Send your new tree (repo state) to github

```
git push origin master
```

Step 7: Remove the now-unneeded reference to oldrepo

```
git remote remove oldrepo
```

References:

- https://stackoverflow.com/questions/37471740/how-to-copy-commits-from-one-git-repo-to-another

IV. CASE STUDY: DECOUPLING APPLICATIONS FROM A MONOREPO TO RESOLVE DEPENDENCY CONFLICTS

Problem

The project housed both a React and an Angular application within a monorepo. The Angular application relied on an outdated design system that was no longer actively maintained, while the React application utilized a newer design system requiring frequent updates to ensure the latest design improvements. This dual reliance on different design systems created significant dependency conflicts:

- **Dependency Misalignment**: Each application required different versions of shared libraries, which made it impossible to upgrade packages without disrupting one or both apps.
- **Design System Stagnation**: The React application could not fully adopt updates from the new design system, causing inconsistencies and limiting the use of new components.
- **Future Decommissioning of Angular App**: The Angular app was slated to be phased out, adding complexity in maintaining its dependencies while ensuring no disruptions to the React app.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

Solution

The team opted to split the monorepo into two independent repositories—one for the React app and another for the Angular app. This separation provided the following immediate benefits:

- **Independent Dependency Management**:
  - Each app could define its own package versions without impacting the other.
  - The React app could continuously adopt updates to the new design system without being constrained by the Angular app's outdated requirements.
- **Facilitated Decommissioning of Angular App**:
  - Decoupling the repositories simplified the shutdown process for the Angular application.
  - Teams could phase out the Angular app incrementally, without any risk to the React app's functionality.

Other Advantages of Splitting the Monorepo

- **Enhanced Scalability**:
  - Teams could scale each application independently, tailoring updates, tools, and configurations to the unique needs of each project.
- **Improved CI/CD Pipelines**:
  - Separate repositories enabled customized CI/CD workflows for each app, optimizing build, test, and deployment pipelines.
  - Reducing the scope of builds resulted in faster pipelines and minimized risks associated with deployments.
- **Reduced Complexity**:
  - The monorepo's tightly coupled structure was replaced with focused, manageable repositories, reducing maintenance overhead.
  - Version control operations (e.g., cloning, commits, branching) became faster and more efficient with smaller repositories.
- **Future-Proofing**:
  - Isolated repositories provided a foundation for adding new features or applications without concerns about legacy constraints.
  - Migration or integration of external systems became simpler, as repositories represented discrete components.
- **Streamlined Collaboration**:
  - Developers could work on their respective apps without requiring full knowledge of the monorepo structure or unrelated components.

V. RECOMMENDATIONS

Here are actionable suggestions for teams considering splitting their monorepo:

- **Assess the Need for Splitting**
  - **Identify Pain Points**: Analyze issues like dependency conflicts, performance bottlenecks, build times, or scalability challenges. Confirm whether these problems warrant splitting the repository or optimizing the existing monorepo.
  - **Evaluate Team Structure**: Assess if your teams work independently enough to benefit from separate repositories or if they rely heavily on shared code and workflows.
- **Define Clear Boundaries**
  - **Logical Separation**: Split the repository along functional or architectural boundaries, such as individual apps, services, or modules with distinct dependency requirements.
  - **Release Cycle Alignment**: Choose splits based on components that share similar release cycles to minimize cross-repository coordination.
- **Plan for Dependency Management**
  - **Shared Libraries**: Establish a system for managing shared code, such as versioned libraries stored in a separate repository or package registry.
  - **Document Dependencies**: Clearly document inter-repository dependencies to avoid confusion and streamline updates.
- **Optimize CI/CD Pipelines**
  - **Create Tailored Pipelines**: Build specific pipelines for each repository to reduce build scope and improve deployment efficiency.
  - **Integration Testing**: Set up mechanisms to test compatibility between decoupled repositories if they share resources or functionality.

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

- **Prepare for Tooling Changes**
  - **Version Control Adjustments**: Familiarize your team with version control tools and workflows suited for polyrepos, such as managing multiple remotes or using subtree techniques.
  - **Dependency Tracking**: Adopt tools that simplify cross-repository dependency tracking and integration.
- **Facilitate Team Transition**
  - **Training and Communication**: Educate teams on the rationale and benefits of splitting. Provide clear guidance on new workflows and repository conventions.
  - **Knowledge Retention**: Document legacy knowledge to ensure smooth collaboration across repositories.
- **Test the Transition Process**
  - **Start Incrementally**: Begin with splitting one or two components as a pilot to identify challenges and refine processes.
  - **Monitor Metrics**: Track performance indicators such as build times, merge conflicts, and development velocity to gauge the effectiveness of the split.
- **Mitigate Risks**
  - **Backup and Version History**: Ensure backups of the original monorepo and preserve version history during the split for traceability.
  - **Dependency Conflicts**: Anticipate and address potential dependency conflicts between repositories early in the transition.
- **Reflect and Optimize**
  - **Post-Split Review**: Evaluate the outcomes of splitting repositories, including team collaboration, performance improvements, and dependency management. Use insights to refine workflows.

Splitting repositories can be a transformative step for teams, but careful planning and execution are key to maximizing its benefits while minimizing disruptions.

VI. CONCLUSION

While monorepos offer significant advantages in terms of streamlined code sharing, centralized dependency management, and simplified CI/CD workflows, they become increasingly complex and difficult to manage at scale. Performance bottlenecks, slow version control operations, and intricate dependency conflicts often make monorepos less efficient for large or loosely coupled projects.

Splitting a monorepo into smaller, more focused repositories provides a practical solution to these challenges. By decoupling repositories, teams can independently manage dependencies, scale CI/CD pipelines, and enhance overall development efficiency. The step-by-step techniques outlined, including git filter-repo, git subtree, and manual extraction, enable teams to preserve commit history and transition smoothly.

The case study highlights the tangible benefits of splitting, such as resolving dependency misalignment, improving scalability, and streamlining CI/CD processes. However, successful monorepo splitting requires careful planning, clear boundary definition, and efficient dependency management. Addressing post-split challenges—such as cross-repository CI/CD coordination, knowledge retention, and team collaboration—ensures minimal disruption and long-term effectiveness.

Ultimately, while monorepos can be powerful for tightly integrated projects, they are not a one-size-fits-all solution. Teams must evaluate their repository strategies based on project complexity, scalability needs, and collaboration patterns. By leveraging the recommended techniques and best practices, organizations can strike the right balance between monorepo efficiency and polyrepo flexibility, fostering more agile and scalable software development.

REFERENCES

1. Ableneo, "Monorepo: Pros, cons, and tools," *Medium*, Sep. 28, 2023. https://medium.com/ableneo/monorepo-pros-cons-tools-2e6f86939be1
2. M. Klein, "Monorepos, please don't," *Medium*, Jan. 2, 2019. https://medium.com/@mattklein123/monorepos-please-dont-e9a279be011b
3. Atlassian, "Monorepos explained: What they are and how to manage them," *Atlassian Git Tutorials*. https://www.atlassian.com/git/tutorials/monorepos
4. Newren, "Git filter-repo," *GitHub*. https://github.com/newren/git-filter-repo

| e-ISSN: 2320-9801, p-ISSN: 2320-9798| www.ijircce.com | |Impact Factor: 8.379 | Monthly Peer Reviewed & Referred Journal |

|| Volume 11, Issue 11, November 2023 ||

| DOI: 10.15680/IJIRCCE.2023.1111003 |

5. J. Sathish, "Extract subdirectory from git repository without losing history," *Medium*, May 9, 2022. https://medium.com/@jeevansathisocial/extract-subdirectory-from-git-repository-without-losing-history-3de8aed359a4
6. Microsoft, "Windows Server 2012 R2 and 2012," *Microsoft Learn*, Aug. 31, 2016. https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/cc771254(v=ws.11)
7. Atlassian, "Git move repository," *Atlassian Git Tutorials*. https://www.atlassian.com/git/tutorials/git-move-repository
8. GitHub, "Splitting a subfolder out into a new repository," *GitHub Docs*. https://docs.github.com/en/get-started/using-git/splitting-a-subfolder-out-into-a-new-repository
9. "How to copy commits from one git repo to another," *Stack Overflow*, May 26, 2016. https://stackoverflow.com/questions/37471740/how-to-copy-commits-from-one-git-repo-to-another

INNO SPACE
SJIF Scientific Journal Impact Factor

Impact Factor: 8.379

doi cross ref

ISSN INTERNATIONAL STANDARD SERIAL NUMBER INDIA

निस्केयर NISCAIR

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN COMPUTER & COMMUNICATION ENGINEERING

9940 572 462 6381 907 438 ijircce@gmail.com

www.ijircce.com

Scan to save the contact details