Paper OS15
Be the Cosmic Guardian of Your Codebase with GIT

Shuai Wu, MSD (UK) Limited, London, UK
Brian M. Lang, MSD, Zurich, Switzerland
Carl Herremans, MSD, Brussels, Belgium
Christian Koul, MSD, Zurich, Switzerland

ABSTRACT

Changes in computing environments, automation, and programming languages, particularly in drug development, have driven the widespread adoption of open-source technologies. Among these, version control systems like Git have become highly popular. However, fully understanding and using Git's extensive capabilities can be challenging, especially for beginners who often stick to basic commands like git add, git commit, and git push. This paper addresses this gap by using a metaphorical approach, drawing parallels between Git and the Sacred Timeline from Marvel's *Loki*. By likening the Git main branch to the Sacred Timeline and feature branches to alternate timelines, this paper provides a relatable framework that enhances programmers' comprehension of Git's branching, merging, and commit history. This analogy helps illustrate the importance of maintaining code integrity and resolving conflicts, much like the Time Variance Authority (TVA) safeguards the multiverse's coherence.

INTRODUCTION

In software development, managing multiple branches in a version control system like Git can be as complex as navigating alternate timelines in a multiverse. Each branch represents a different path in the development process, and without careful management, these paths can quickly become chaotic. Inspired by the Marvel Cinematic Universe's *Loki* series, where the Sacred Timeline is managed by the TVA to ensure order, we find a compelling analogy for understanding Git's branching structure.

In this analogy, the Git main branch represents the primary path leading to a production-ready code product, akin to the Sacred Timeline. The main branch, like the Sacred Timeline, must be meticulously managed to ensure stability and integrity. Feature branches can be seen as alternate timelines, which may eventually be merged back into the main branch or pruned if they do not align.

This paper delves into this analogy, using the concept of the Sacred Timeline to illuminate key principles and best practices for managing Git branches. By likening developers to the TVA, we offer fresh insights into maintaining order in codebases, ensuring main branch stability, and effectively handling feature development and integration.

PARALLEL BETWEEN *LOKI* AND GIT

In the TV series *Loki*, various elements can be metaphorically compared to functionalities in Git, enriching our understanding of version control in software development. By exploring these parallels, we can gain deeper insights into effectively managing version control.

Loki: *Loki*, the God of Mischief, manipulates time and creates alternate versions of himself, known as variants. This mirrors how developers create branches for new features in Git. For example, when developers start a feature branch to experiment with a new idea, they create a version of the code that may later merge back into the main branch or be discarded, similar to *Loki*'s variants exploring different realities.

Sacred Timeline: The Sacred Timeline represents the primary and stable sequence of events within the multiverse, which the TVA works to preserve. This is akin to the Git main branch, which contains the production-ready code. Maintaining the stability of the main branch is crucial, just as ensuring the Sacred Timeline remains orderly. For instance, the main branch must hold code that is free from critical issues, ensuring the product's integrity.

Alternate Timelines: These are the divergent paths that emerge when actions cause deviations from the Sacred Timeline. In Git, feature branches represent these alternate timelines, created to explore new developments. A developer might create a feature branch to add a new function, which can either be merged into the main branch if it aligns with project goals or pruned if it does not, much like managing alternate timelines.

TVA: The Time Variance Authority (TVA) acts as the custodian of the Sacred Timeline, monitoring and correcting any anomalies or disruptions, referred to as Nexus Events. This parallels a centralized remote repository and its development team, which serve as custodians of the main branch. Code reviewers, like TVA agents, ensure the codebase remains stable and consistent by resolving conflicts and maintaining quality during code reviews. For example, a pull request undergoes scrutiny to ensure it meets the project's standards before merging.

Nexus Events: Nexus Events are critical moments when a timeline deviates from its predetermined path, leading to potential chaos. The TVA intervenes in these events to reset or prune the timelines, restoring order. In Git, Nexus Events are akin to merge conflicts, representing critical points where deviations or inconsistencies arise and require resolution to maintain stability. For instance, if two developers modify the same line of code in different branches, a merge conflict occurs that must be manually resolved, ensuring the code remains coherent.

By drawing these parallels, we gain a vivid framework to better visualize Git's structure and management practices. Imagine developers as mischievous Lokis, each experimenting with different features and enhancements by creating and managing various branches, akin to *Loki*'s exploration of alternate timelines. Similar to how *Loki*'s variants embody different versions of himself, these branches represent different versions of the code, each with the potential to be integrated or discarded based on their alignment with the project's goals.

The centralized repository, acting as the Time Variance Authority (TVA), serves as the vigilant custodian of the codebase. The TVA's role in ensuring the integrity of the Sacred Timeline is mirrored by the development team's responsibility in maintaining the stability and consistency of the main branch. Code reviewers step into the shoes of TVA agents, meticulously examining pull requests to resolve conflicts and ensure that newly introduced changes do not destabilize the project. Their intervention is crucial during critical moments, much like the TVA's handling of Nexus Events, ensuring that the codebase remains coherent and functional.

Moreover, understanding these analogies helps underscore the importance of best practices in version control. Just as the TVA must manage and prune alternate timelines to prevent chaos, developers must judiciously handle feature branches and resolve merge conflicts to maintain a smooth, orderly, and high-quality main branch. This broader perspective not only enhances our comprehension of Git's functionalities but also emphasizes the collaborative effort required to keep a project on track. Development, much like the multiverse, is a complex, dynamic process that necessitates vigilance, coordination, and a deep understanding of both the creative and regulatory facets involved.

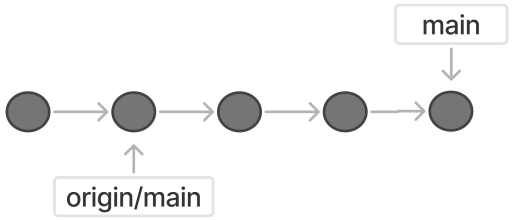
BRANCH MANAGEMENT IN GIT

In the Marvel series *Loki*, the Time Variance Authority (TVA) ensures the stability of the Sacred Timeline by managing various alternate timelines. Similarly, in Git, effective version control and collaboration hinge on understanding the relationships between local and remote repositories and between different branches. By drawing on parallels from the *Loki* series, we can explore these concepts in an engaging and memorable way.

LOCAL VS REMOTE REPOSITORIES

Just as the TVA maintains a central view of all timelines, the remote repository in Git serves as the definitive source of truth for the codebase. The local repository, akin to a developer's personal timeline, must stay in sync with the remote to ensure consistency and stability. Maintaining this sync requires a series of actions: adding changes, committing them to the local history, pushing them to the remote, and pulling updates from the remote repository.

Developers stage changes by using the `git add <filename>` command, or `git add .` to stage all changes. This step is like noting down important events in a personal timeline. Once the changes are staged, the next step is to commit them using `git commit -m ""commit message""`, which saves these changes to the local repository. This mirrors the TVA's process of updating their records to reflect new events in a timeline.



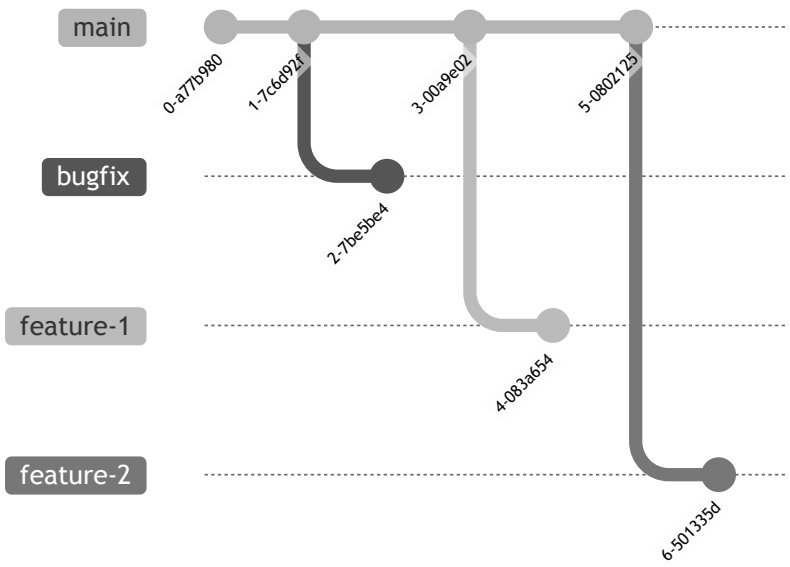
To ensure that these local changes are reflected in the overarching history maintained by the TVA, developers must push their commits to the remote repository using `git push origin main`. Conversely, to stay informed of changes made by others, developers pull updates from the remote repository with `git pull origin main`. This continuous cycle of adding, committing, pushing, and pulling keeps the local and remote repositories synchronized, just as the TVA ensures coherence across multiple timelines.

KEEPING BRANCHES UP-TO-DATE

As the TVA monitors alternate timelines to maintain order, developers must keep their feature and main branches up-to-date to avoid conflicts. Fetching the latest changes from the remote repository using `git fetch origin` is like the TVA gathering updates from different timelines. To align the feature branch with the changes in the main branch, developers use `git rebase main`, similar to how the TVA aligns an alternate timeline with the Sacred Timeline. Finally, merging changes involves resolving any conflicts that arise, akin to the TVA addressing Nexus Events that threaten the stability of the multiverse.

BRANCHING IN GIT

Branches in Git function like alternate timelines in the *Loki* series, allowing developers to work on new features or bug fixes independently without affecting the main codebase. The main branch represents the Sacred Timeline, the stable and official history of the codebase.



To create a new branch for adding a new feature, developers use the command `git branch branch-name`. This command is like creating an alternate timeline for a specific purpose. Switching to this new branch with `git checkout branch-name` allows developers to work on it independently. Since main branch represents the stability of codebase and under protection, other development branches should only be merged into with pull requests (GitHub or bitbucket) or merge requests (GitLab) to request the review from reviewers. After a thorough review is completed, they could be merged back into the main branch with `git checkout main` followed by `git merge branch-name`, similar to how the TVA might reintegrate an alternate timeline back into the Sacred Timeline.

HANDLING CONFLICTS

Conflicts in Git are akin to Nexus Events in the *Loki* series. They occur when changes in different branches impact the same lines of code, requiring intervention to resolve. During a merge or rebase, Git will highlight these conflicts, marking the conflicting sections with indicators like `<<<<<<< HEAD`.

Developers must manually resolve these conflicts by editing the files and deciding which changes to keep, combining the different changes, or creating a new integrated solution. Once resolved, the developer commits the changes to finalize the conflict resolution, ensuring a stable and consistent codebase.

For example, consider a function to calculate the mean in an R script. If the main branch and a feature branch both modify this function, a conflict will arise:

```
# analysis.R

# Function to calculate the mean
calculate_mean <- function(x) {
<<<<<<< HEAD
  print(""Calculating mean..."")
  mean(x, na.rm = TRUE)
=======
  if (length(x) == 0) {
    return(NA)
  }
  mean(x, na.rm = TRUE)
>>>>>>> bugfix
}
```

DIVERGENCE AND CONFLICT SOLUTIONS

When local and remote branches diverge, it's akin to different timelines developing independently. To maintain synchronization, developers may need to reset their local branch and pull the latest updates using commands like `git reset --soft HEAD~1` followed by `git pull origin main`. Handling divergence conflicts involves fetching changes from the remote repository with `git fetch`, merging them into the local branch, and resolving any conflicts that arise.

DIFFERENCE BETWEEN REVERT AND RESET

Understanding the difference between the `revert` and `reset` commands is crucial for proper version control. The `revert` command is used to create a new commit that effectively undoes the changes made in a prior commit. It achieves this by creating a new commit that reflects the inverse of the undesired changes, thereby preserving the commit history. Importantly, this process allows for the safe and non-destructive removal of specific changes while maintaining the integrity of the project history, much like the TVA correcting minor deviations without disrupting the timeline. The revert command is suited for scenarios where it's necessary to correct errors within the commit history without altering the existing commits. Conversely, the `reset` command is utilized to manipulate the commit history by moving the current branch reference to a specified commit. There are different modes of `git reset-soft, mixed, and hard`, each with unique effects on the state of the working directory, staging area, and commit history, akin to the TVA resetting an entire timeline to an earlier state. It's important to exercise caution when using git reset, particularly the hard reset option, as it has the potential to permanently discard local changes and alter the commit history. With the Git integration, the developers should be careful but also not too worried about making mistakes. That's why we could leverage `revert` and `reset` commands if we made the wrong commits.

By integrating these best practices and drawing on the engaging analogy of the *Loki* series, developers can ensure their contributions are in sync with the remote repository, maintaining a stable and orderly codebase. Following these guidelines helps manage branches effectively, prevent conflicts, and resolve any issues that arise, much like the TVA managing the multiverse.

FURTHER GIT WORKFLOW EXAMPLES

SCENARIO 1: SYNCHRONIZING LOCAL AND REMOTE BRANCHES

A developer working on a local branch called `local-branch` needs to synchronize with the remote branch `origin/main`. This process is akin to the TVA in *Loki* ensuring that local timelines are consistent with the Sacred Timeline, maintaining an up-to-date and accurate reflection of events.

To synchronize the local branch with the remote branch:

```
git checkout local-branch
git pull origin main
```

`git pull origin main` is a combination command that first retrieves updates from the remote branch (fetch) and then merges these updates into the local branch (merge). By pulling changes regularly, the developer ensures that their local branch stays up-to-date with the latest developments from the remote repository. This practice is similar to the TVA's continuous monitoring and realignment of timelines to ensure consistency and accuracy.

If conflicts arise during the pull process, Git will pause and provide instructions on how to resolve them. The developer must resolve these conflicts by manually editing the files, staging the resolved changes with `git add`, and then completing the merge with `git commit`.

```
# After resolving conflicts in the files
git add <conflicted-files>
```

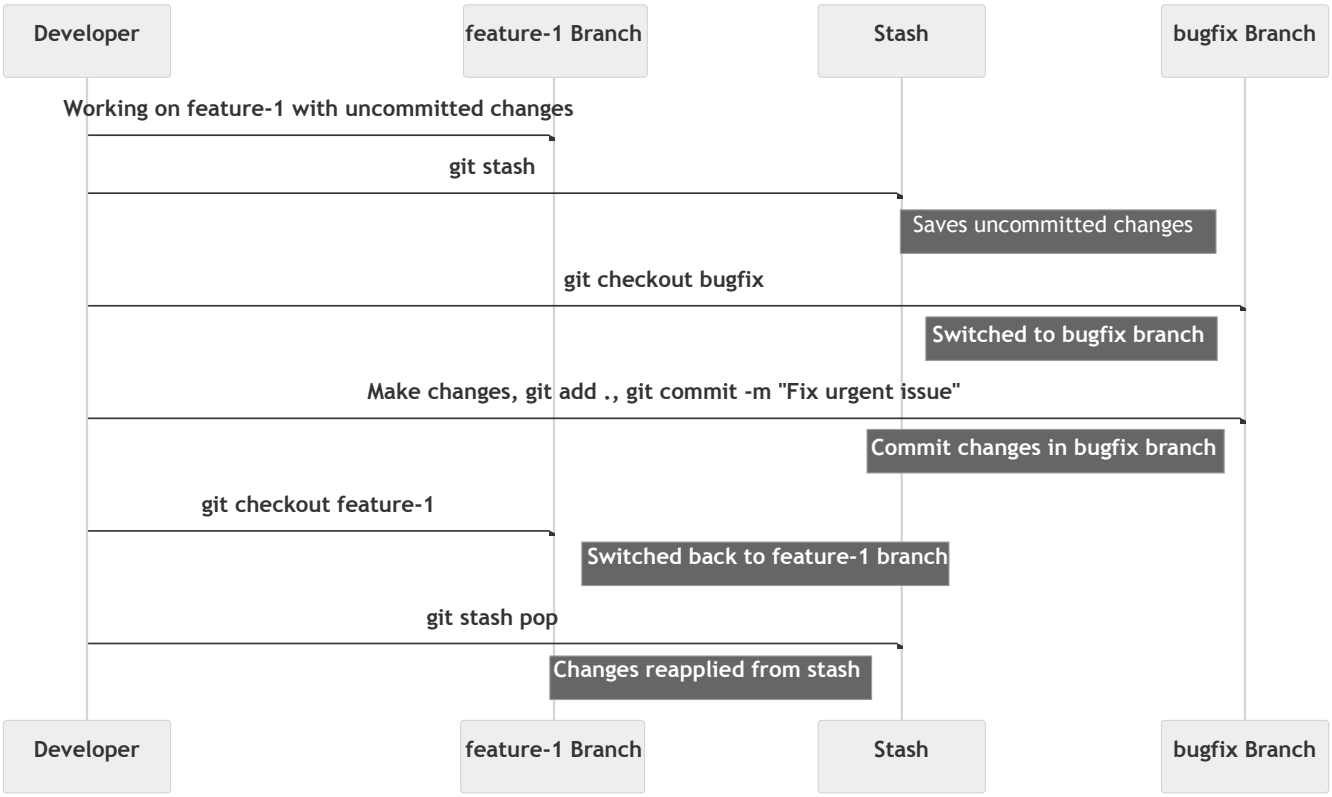
```
git commit -m ""Resolved merge conflicts""
```

SCENARIO 2: USING STASH TO MANAGE CHANGES

Like *Loki*'s ability to manipulate time and different realities, allowing him to jump into different timelines, developers can work on different branches even if they have changes in one branch using `git stash` and `git stash pop`. The `git stash` command saves the current state of the working directory and index, useful for temporarily setting aside changes without committing them. While, the `git stash pop` command retrieves the most recently stashed changes, allowing developers to continue working on the previous task.

Imagine working on `feature-1` and needing to address an urgent issue in another branch. You can use `git stash` to set aside your current changes, akin to *Loki* hiding something temporarily. Switch to `bugfix` with `git checkout bugfix`, fix the bug, and commit the changes.

After resolving the urgent issue, switch back to `feature-1` with `git checkout feature-1` and use `git stash pop` to retrieve your previously stashed changes. This is like *Loki* revealing hidden elements from another reality, allowing you to pick up where you left off without any loss of progress.



By using `git stash` and `git stash pop`, you efficiently manage multiple tasks without disrupting your workflow.

SCENARIO 3: MERGING FEATURE BRANCHES

Merging combines the changes from the feature branch into the main branch without altering the commit history. This process is akin to how the TVA ensures that events from different timelines are integrated without disrupting the established history of the Sacred Timeline.

Imagine a scenario where a developer is working on a branch named `feature-branch` and wants to integrate the completed work into the main branch. To do this, the developer uses the merge command:

```
git checkout main
git merge feature-branch
```

This is akin to the TVA integrating events from an alternate timeline into the Sacred Timeline without altering the historical context. If conflicts arise during the merge, Git will pause and provide instructions on how to resolve them. The developer must resolve these conflicts by manually editing the files, staging the resolved changes with `git add`, and then completing the merge with `git commit`. This is similar to how the TVA resolves discrepancies between timelines to ensure a coherent history.

```
# After resolving conflicts in the files
git add <conflicted-files>
git commit -m ""Resolved merge conflicts""
```

SCENARIO 4: REBASING A FEATURE BRANCH

Rebasing involves reapplying commits from a feature branch on top of the main branch, creating a linear history. This is similar to how the TVA in *Loki* might realign events in a divergent timeline to match the Sacred Timeline, ensuring a clean, linear sequence of events.

Imagine a developer is working on a new feature in a branch called `feature-branch`, and the main branch has advanced since the `feature-branch` was created. To realign the feature branch with the latest changes from the main branch, the developer can use the rebase command:

```
git checkout feature-branch
git rebase main
```

During the rebase process, Git will attempt to apply each commit from the feature branch on top of the main branch. If conflicts arise, the developer must resolve them before continuing the rebase, much like the TVA handling various Nexus Events in divergent timelines to ensure they are resolved and reintegrated back into the Sacred Timeline. Once the rebase is complete, the feature branch will have a clean, linear history that includes the latest changes from the main branch, just as the TVA ensures a seamless and orderly timeline in the multiverse.

SCENARIO 5: CHERRY-PICKING SPECIFIC COMMITS

Cherry-picking allows developers to apply specific commits from one branch to another. This can be useful when only certain changes are needed in a different context. For example, during development, when specific changes need to be applied to multiple branches, git cherry-pick provides a means to selectively apply these changes to each branch as needed. Imagine the TVA deciding to extract specific events from an alternate timeline to integrate into the Sacred Timeline.

```
git cherry-pick <commit-hash>
```

This command takes a specific commit from one branch and applies it to the current branch, facilitates the selective and precise application of individual commits across different branches, providing flexibility and control over the commit history and code changes, much like selectively realigning specific events in the multiverse.

SCENARIO 6: INTERACTIVE REBASE FOR COMMIT HISTORY CLEANUP

Interactive rebase can be used to rewrite commit history, squash commits, or edit commit messages. It's like the TVA revisiting a history and erasing or reordering, or merging events.

```
git rebase -i HEAD~n
```

This opens an interactive interface where you can choose to squash, edit, or reword commits, helping maintain a clean and understandable commit history.

BEST PRACTICES FOR GIT USAGE

In Git, managing your project history and collaborating effectively with your team can be likened to the dynamic interplay between *Loki*'s manipulation of timelines and the TVA's meticulous oversight. Below are key practices to ensure your Git workflows remain efficient and your project history clean.

Pull Before Push: Always pulling the latest changes from the remote repository before pushing your changes ensures your local repository is synchronized with the remote one. This practice minimizes the risk of conflicts by integrating updates from others early, similar to how the TVA ensures all timelines are synchronized (Loeliger and McCullough 2012). For example, running `git pull origin main` before `git push origin main` keeps your work aligned with the latest project state, reducing the likelihood of disruptive conflicts.

Regularly Merge with Main Branch: Regularly merging the main branch into your feature branch keeps it up-to-date with the latest changes. This proactive approach reduces the complexity of future merges and allows you to handle conflicts incrementally, much like the TVA ensuring timelines are consistently aligned (Bird et al. 2009). Incorporating changes incrementally prevents the accumulation of conflicts, making integration smoother and less error-prone.

Rebase vs. Merge: Rebase is akin to *Loki*'s ability to manipulate events and create a smooth, linear timeline. When you rebase your feature branch with changes from the main branch, you're rewriting the commit history to appear as though changes were made sequentially. This keeps the project history clean and easier to follow [Chacon and Straub (2014)

In contrast, merging is like the TVA preserving the branching structure and all the events that transpired. When you merge completed features back into the main branch, you maintain the full development history, providing an accurate and comprehensive record of the project's evolution (Brooks 1995). This transparency is invaluable for auditing and understanding the development process, akin to the TVA's records of all events across timelines.

Effective Communication and the Role of Code Reviewers (*Loki*/TVA): Effective communication among developers is crucial, akin to how *Loki* and the TVA must communicate to maintain order. Regular discussions about ongoing changes and integration plans help avoid redundant work and conflicts. Tools like instant messaging, email, and regular stand-up meetings facilitate this communication. For instance, syncing with your team regularly about branch updates ensures everyone is aware of the latest changes and can plan their work accordingly.

The role of code reviewers can be seen as the TVA's vigilant oversight. Code reviewers maintain code quality and ensure adherence to best practices by reviewing pull requests and providing feedback. Automated tools can assist, but the human aspect of code reviews ensures only high-quality, conflict-free code is merged, maintaining the project's integrity (Rigby and Bird 2013)

CONCLUSION

The analogy between *Loki*'s Sacred Timeline and Git branches offers a unique and engaging framework for understanding the complexities of branch management in software development. By comparing the main branch to the Sacred Timeline and feature branches to alternate timelines, developers can better grasp the importance of resolving conflicts, synchronizing changes, and maintaining code integrity.

This paper has detailed several best practices for Git usage, emphasizing the roles of rebase and merge, the necessity of synchronizing local and remote repositories through the "pull before push" practice, and the benefits of regularly merging the main branch into feature branches during development. By implementing these strategies, developers can ensure a cleaner, more understandable project history, much like the TVA's safeguarding of the multiverse's coherence.

Equally critical is the emphasis on effective communication and the role of code reviewers, likened to the TVA's vigilant oversight. By actively communicating about ongoing changes, potential conflicts, and integration plans through tools like messaging apps, email threads, and regular stand-up meetings, teams can avoid redundant work and smooth out the development process. Code reviewers act as gatekeepers of code quality, ensuring that only high-quality, conflict-free code is merged into the main branch, thereby maintaining the integrity of the codebase.

In summary, just as the TVA safeguards the multiverse, developers must be the cosmic guardians of their codebase, ensuring a stable and coherent history for their projects. By following best practices for branching, merging, and collaboration, teams can navigate the challenges of software development with greater confidence and efficiency. Understanding and implementing these strategies will lead to a well-managed and harmonious codebase, ultimately contributing to the success and stability of development projects.

REFERENCES

Bird, Christian, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. 2009. "Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista." *Communications of the ACM* 52 (8): 85–93. https://doi.org/10.1109/ICSE.2009.5070550.

Brooks, Frederick P. 1995. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional.

Chacon, Scott, and Ben Straub. 2014. *Pro Git*. Apress.

Loeliger, Jon, and Matthew McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, Inc.

Rigby, Peter C, and Christian Bird. 2013. "Convergent Contemporary Software Peer Review Practices." *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 202–12.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Shuai Wu
Email: shuai.wu@msd.com.
Company: MSD
Address: 120 Moorgate, London, EC2M 6UR, United Kindom

Brian M. Lang
Email: brian.lang@msd.com.
Company: MSD
Address: The Circle 66, 8058, Zurich, Switzerland