IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 5, Issue 1, December 2025
Impact Factor: 7.67

Analysis of Version Control Workflows

Sarthak Narayan Kadam[1] and Rohini S. Kapse[2]
M. Sc. (C.S)-II, Department of Computer Science and Applications[1]
Assistant Professor, Department of Computer Science and Applications[2]
MVPS K. T. H. M. College, Nashik.
Corresponding Author: Name: Sarthak Narayan Kadam
kadamsarthak96k@gmail.com
ORCID iD: https://orcid.org/0009-0000-3466-8621

Abstract: *Version control systems (VCS) are essential for modern software development. They help maintain code quality, track changes, and enable collaboration among distributed teams. Of the various systems available, Git has become popular because of its flexibility and distributed structure. Over time, different workflows have emerged, such as Git Flow, GitHub Flow, and Trunk-Based Development, to meet diverse project needs. This paper presents a study of version control workflows, focusing on their structure, merge management, collaboration styles, and suitability for different team sizes. Insights from recent studies highlight the strengths and weaknesses of each model regarding integration frequency, release stability, and conflict management. The findings show that while structured workflows like Git Flow ensure predictability and stability, lightweight models like GitHub Flow and trunk-based development improve speed and continuous integration. The paper wraps up with an evaluation of how adaptable these workflows are in modern DevOps environments and suggests future research directions for improving automated version control strategies.*

Keywords: Version Control System, Git Workflow, Git Flow, GitHub Flow, TrunkBased Development, Merge Conflicts, Continuous Integration

I. INTRODUCTION

Software development now involves large, distributed teams working together through shared repositories. Version Control Systems (VCS), such as Git, play a crucial role in coordinating changes and keeping code intact. However, how productive and efficient collaborative projects are relies not just on the tools but also on how workflows are organized. The shift from centralized to distributed branching models has greatly affected the software delivery process. Frameworks like Git Flow, GitHub Flow, and Trunk-Based Development have arisen to meet different needs in release control, agility, and collaboration. This research aims to systematically examine these workflows to understand their characteristics, benefits, and drawbacks in real-world settings. The paper is organized as follows: reviews related work, describes the proposed system and methodology, details the experimental setup, presents the results, and offers key insights and discusses future directions.

II. LITERATURE REVIEW

The Foundational Role and Evolution of Version Control: Version Control Systems (VCS) are crucial components of modern software development, providing the mechanism for managing complex codebases and ensuring efficient workflows. [1] Historically, before the advent of VCS, programmers relied on manual methods such as creating backups of files or employing naming conventions (e.g., analysis.sh, analysis02.sh) to distinguish between versions. This manual process was inherently inconsistent and difficult to manage, particularly when multiple developers collaborated on the same project. [2] [3] A VCS addresses these core issues by allowing developers to track the iterative changes made to their code. [3] This capability is fundamental because, during iterative development, scientists and engineers frequently need to experiment with new ideas without risking damage to the currently working code. [3] With a VCS, developers always have the option to revert to a specific past version of the code, which is essential for

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 5, Issue 1, December 2025 Impact Factor: 7.67

documenting and tracking which code version was used to produce specific results, particularly when revisiting a project months later. Furthermore, a key function of VCS is enabling collaboration, facilitating the automatic incorporation of changes made by multiple collaborators into the main codebase. Beyond software code, a VCS framework is also well-suited for tracking any plain-text files, including manuscripts, electronic lab notebooks, and protocols. [3]

The evolution of VCS is marked by a shift from early systems to increasingly complex, collaborative models. Early systems gave way to Centralized Version Control Systems (CVCS), which hosted the project repository on a central server. Developers would check out code, modify it, and commit changes back to this central server. However, centralized systems struggled to handle larger projects and teams distributed across different locations. The significant breakthrough was the introduction of Distributed Version Control Systems (DVCS), such as Git and Mercurial. DVCS architecture provides every developer with a full copy of the entire code repository, along with its history, enabling more advanced contemporary version control and distributed collaboration. Git, initially designed by Linus Torvalds in 2005 for coordinating Linux kernel development, exemplified this shift toward systems emphasizing efficiency, scalability, and decentralized tracking. [2] In the Git framework, the repository (or repo) refers to the current version of the tracked files alongside all previously saved versions. [3] Git saves conceptual snapshots of changes whenever a user instructs it to, and these snapshots are referred to as commits. Each commit is assigned an identifier, which can be used to compare two versions, restore a file to a previous state, or retrieve tracked files if they are accidentally deleted. Git's design enhances transparency, reproducibility, and openness to collaboration in science. [3]

VCS Workflows and Branching Strategies:

The inherent capability of modern VCS, particularly Git, to easily and quickly create branches allows development teams to customize their collaborative workflows. These workflows generally fall into two main categories: branch-based workflows and trunkbased workflows. [4]

Branch-Based Workflows:

Branch-based workflows define a set of remote branches, each with well-defined objectives (e.g., branches for features, integration, and the main line of development). Changes are migrated sequentially from the most isolated branches toward the main branch as the code matures. GitFlow stands out as the most popular example within this category.

• GitFlow Structure:

GitFlow is characterized as a robust branching model suitable for projects with reasonably well-defined release cycles. [2] It typically employs two main branches: the master branch, which holds production-ready code, and the develop branch, which serves as the integration point for new features. [2] Supporting branches include feature branches (for new development), release branches (for pre-release preparation), and hotfix branches (for immediate production bug fixes). [2] [4]

• Suitability and Advantages:

Branch-based development tends to suit less experienced and larger teams better. [4] Factors promoting its usage include organized commit flow, better control over commits, and organized repository structure (Code Organization). It supports flexible and parallel feature development through self-containment on dedicated branches. This model is particularly valued in restricted sectors like finance, which require higher security and compliance, as it allows for the separation of environments (e.g., development, quality assurance, pre-production) before changes reach production. [4] [2] This approach is frequently related to an explicit concern for code quality.

Trunk-Based Workflows:

In contrast, trunk-based workflows utilize a single remote branch (the trunk) where developers integrate their changes directly and frequently, often multiple times per day. This model ensures the codebase is perpetually available on demand. Trunk-based development is favored for fast-paced projects with experienced developers and smaller teams.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 5, Issue 1, December 2025 Impact Factor: 7.67

Choosing the optimal workflow is a complex task, as the selection must maximize productivity while promoting software quality. The choice ultimately depends on circumstances such as project size, needs, team seniority, and organizational development guidelines. [2] [4]

VCS Integration with Modern Development Practices:

VCS tools are integral to modern collaborative software development, especially when adopting agile methodologies and DevOps practices. The integration of version control into DevOps signifies a shift where VCS transcends simple change tracking to become an essential component of the entire software delivery chain. [2] [1] VCS facilitates Continuous Integration (CI) and Continuous Deployment (CD) pipelines. Effective version control practices are critical in DevOps environments, which require systems capable of managing extensive, rapidly recurring changes. The practice of branching and utilizing Feature Toggles allows teams to handle different features simultaneously without disrupting the principal code line, thereby enabling parallel development that speeds up the development cycle. [2] VCS integration also improves visibility and accountability across the development stack, ensuring teams know who made which modifications, when, and why. This makes the VCS the single source of truth for code and artifacts. Looking forward, trends anticipate closer integration between VCS and other DevOps tools, such as CI/CD platforms and issue tracking systems. Furthermore, there is a growing recognition that version control should be extended beyond code to include non-code artifacts like configurations and datasets. [3] [2]

The Challenge of Merge Conflicts:

Despite the sophisticated collaboration tools offered by VCS, merge conflicts remain an inevitable and disruptive aspect of development. [5] A merge conflict occurs when concurrent changes interfere textually. [6] Empirical evidence shows that conflicts are frequent; in open-source projects, merge conflicts occur in approximately 19 percent of all merges. Resolving them is often difficult, time-consuming, and error-prone. [7] [5] [8] [9]

Unstructured Merging and Conflict Causes:

The core reason for textual conflicts is that most modern VCS, including Git, Mercurial, and SVN, treat software artifacts as plain text and rely on line-by-line merging (unstructured merging). [8] [9] Git's default strategy uses a three-way merge algorithm on the text level. [10] While unstructured merging is advantageous for being language-independent and applicable to all textual artifacts, it fails when multiple changes occur to the same lines because it ignores the syntactic and semantic meaning of the code. [8] When a textual merging tool encounters two different changes to the same line of code, it reports a conflict. [9] These conflicts impose significant costs, frequently stalling continuous integration pipelines. [6] Furthermore, conflicts that are missed by current merge tools (indirect conflicts) might not be discovered until testing or building, or they might even be released, leading to unexpected software behavior. [8] [5]

Conflict Categorization and Impact on Quality:

Empirical studies have sought to quantify and categorize merge conflicts to understand their root causes. [7] Based on the type of changes causing the conflict, categories include:

- **SEMANTIC :** These conflicts involve changes to program logic or structure, such as refactoring like renaming variables. They require the most developer reasoning and typically account for the largest proportion of merge conflicts (nearly 60 percent in one study). They often result in large differences in the Abstract Syntax Tree (AST).
- **FORMATTING :** Conflicts arising solely from changes in formatting, such as whitespace. These constituted approximately 23.21 percent of conflicts in one corpus.
- **DISJOINT :** involving independent changes.
- **DELETE :** where code modified on one branch is deleted on the other.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 5, Issue 1, December 2025 Impact Factor: 7.67

- **COMMENTS :** Changes limited only to comments. A crucial finding from empirical research is the strong link between merge conflicts and subsequent software quality issues. Code that was involved in a merge conflict has a higher likelihood—specifically a 2× higher chance—of being buggy in the future. Factors correlated with increased bug proneness include the size of the change, the number of committers involved in the merge, and changes made to central files. In terms of resolution, developers primarily use the ADAPTED strategy (60.82 percent), meaning they manually modify existing lines or add new lines to resolve the conflict. For SEMANTIC conflicts, the ADAPTED strategy is required approximately 80 percent of the time. The Social Dimension of Conflicts Recent research has increasingly focused on social assets (developer roles and relationships) as predictors of merge conflicts. Empirical studies show that developer roles are statistically related to the emergence of merge conflicts. [7] [11]

• Conflict-Prone Roles:
Contributors at the project level and occasional contributors at the merge-scenario level are found to cause proportionally more conflicts. The combination of a top contributor (project level) and an occasional contributor (merge-scenario level) touching the source branch is particularly conflict-prone, leading to conflicts in 32.31 percent of scenarios. [11]

• Branch Focus:
Contributions made to the source branch are significantly more conflict-prone than contributions to the target branch.

• Prediction:
Prediction models that combine both technical (e.g., number of simultaneously changed files) and social assets demonstrate high effectiveness, achieving accuracy of 0.92 and perfect recall (1.00) for predicting conflicts, suggesting that the integration of developer role information is vital for improving speculative merging techniques. [11] Advanced Merging and Conflict Resolution Techniques: Given the limitations of traditional unstructured merging, researchers have proposed numerous techniques categorized as structured, semi-structured, refactoring-aware, and machine learning-based approaches. Structured and Semi-Structured Merging To address the shortcomings of textual merging, researchers investigate methods that leverage the inherent structure of the code. • Structured Merging techniques transform the code into an Abstract Syntax Tree (AST), converting the merging problem into the integration of nodes and edges on the AST. Examples include Mastery and Spork. [5] [10] • Semistructured Merging aims for a middle ground, exploiting structural information only when the unstructured merge fails. [7] By representing artifacts partly as text and partly as trees, this approach achieves a level of language-independence while improving precision over purely unstructured methods. [8] Studies confirm that semi-structured merge can significantly reduce the number of merge conflicts, sometimes by half [8] [9]. Examples include FSTMerge and JDime. JDime specifically introduces a structured merge approach with auto-tuning that dynamically switches between unstructured and structured merging based on the presence of conflicts. [8] [10]
Refactoring-Aware Merging: Merge conflicts involving refactorings are noted as being larger and harder to resolve. Refactoring-aware merging techniques are designed to handle these specific cases: • Operation-based Refactoring-aware Merging: This technique, pioneered by MolhadoRef, treats refactorings as operations, inverting and replaying them while considering their semantics. [5] RefMerge, a rejuvenated, Git-based implementation of this approach, supports 17 refactoring types. [8] • Graph-based Merging: IntelliMerge is an example of a graph-based refactoring-aware technique that uses semantic analysis to transform source code into a program element graph. [10] Comparative studies have shown the effectiveness of these advanced techniques. In one large-scale comparison, RefMerge resolved or reduced conflicts in 25 percent of merge scenarios, slightly outperforming IntelliMerge (24 percent). Notably, RefMerge worsened the conflict situation (increased conflicting lines of code or files) in a smaller proportion of scenarios (11 percent) compared to IntelliMerge (30 percent). Furthermore, RefMerge demonstrated an ability to reduce the number of false positives reported compared to Git and IntelliMerge, while entirely eliminating false negatives in a qualitative analysis. [8]

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 5, Issue 1, December 2025 Impact Factor: 7.67

Machine Learning for Merge Resolution and Prediction:

A new dimension in merge conflict resolution involves utilizing machine learning. DeepMerge is a data-driven approach that frames resolution as a sequence-to-sequence machine learning problem. [11] This approach is motivated by the observation that a vast majority (80–87 percent) of resolutions consist merely of re-arranging lines from the conflicting region without introducing new code. DeepMerge employs an edit-aware embedding and a pointer network design to synthesize resolutions. DeepMerge successfully predicted correct resolutions for 37 percent of non-trivial JavaScript merges in a test set, representing a 9x improvement over a state-of-the-art semi-structured technique, and achieved 78 percent accuracy for small merges (up to 3 lines). [6] Additionally, auxiliary tools like WizardMerge focus on providing suggestions rather than prescribing resolutions, recognizing the unpredictability of developers' ultimate needs. WizardMerge uses dependency analysis based on LLVM Intermediate Representation (IR) to group conflicts by relevance and assign fix priorities. Crucially, WizardMerge aims to detect Violated Diff Code Blocks (DCBs)—code automatically applied by Git that may still introduce unexpected results or latent bugs because necessary dependent definitions are missing. WizardMerge showed it could reduce manual merging time costs by 23.85 percent. [10]

Merging vs. Rebasing :

While explicit merge (git merge) is the traditional integration method, which preserves history by creating a merge commit, [9] rebasing (git rebase) is also widely used, particularly in pull requests hosted on platforms like GitHub. Rebasing rewrites the evolutionary history of commits, essentially serving as an implicit merge. Developers utilize rebasing to synchronize with an updated base branch and maintain a clean, linear evolutionary history. [12] Empirical studies confirm that textual conflicts arise in rebases (24.3 percent–26.2 percent) at a rate similar to explicit merges. Developers tend to resolve these conflicts similarly to explicit merges, often combining existing lines/tokens rather than introducing new code. However, a significant difference noted is that developers often introduce new changes during the rebase process in 34.2 percent of non-conflict rebase scenarios, posing new challenges for validation compared to explicit merges. [12] The continuous analysis of VCS tools and practices—from fundamental version tracking and sophisticated branching workflows to the complex challenge of conflict resolution and the adoption of advanced automated merging techniques—remains a vibrant area of software engineering research.

III. PROPOSED SYSTEM

This study proposes a comparative framework for evaluating different Git-based workflows using four key metrics:

Merge Frequency: How often code is integrated into the main branch.

Stability: The reliability of builds after integration.

Release Management Complexity: The ease of managing multiple versions or environments.

Collaboration Overhead: The communication and coordination required among developers.

Each workflow, Git Flow, GitHub Flow, and Trunk-Based Development is analyzed against these criteria. The framework also incorporates automation and continuous integration (CI) considerations to measure modern workflow efficiency. The aim is to identify workflow patterns that balance flexibility, stability, and maintainability across teams of varying sizes and project complexities.

IV. EXPERIMENT

4.1. Experimental Setup

A controlled simulation was conducted using Git repositories to emulate small- and medium-scale team collaboration. Each workflow was implemented independently using identical project codebases. Git logs were analyzed to measure:

- Number of merges per week
- Merge conflicts encountered
- Average resolution time
- Build success rate

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 5, Issue 1, December 2025 Impact Factor: 7.67

4.2. Tools Used

- Git
- GitHub for hosting repositories

4.3. Workflow Simulation

- **Git Flow:** Multiple long-lived branches (main, develop, feature, release) were used.
- **GitHub Flow:** Direct branching from the main branch, with short-lived feature branches.
- **Trunk-Based Development:** Frequent commits directly to the main branch with automated testing.

V. RESULTS AND DISCUSSION

The results reveal clear differences among the three workflows.

Table 1: Comparison of Git-Based Workflows

Workflow	Merge Frequency	Stability	CI/CD Support	Ideal Team Size
Git Flow	Low	High	Moderate	Large
GitHub Flow	High	Moderate	High	Medium
Trunk-Based	Very High	Moderate	Excellent	Small

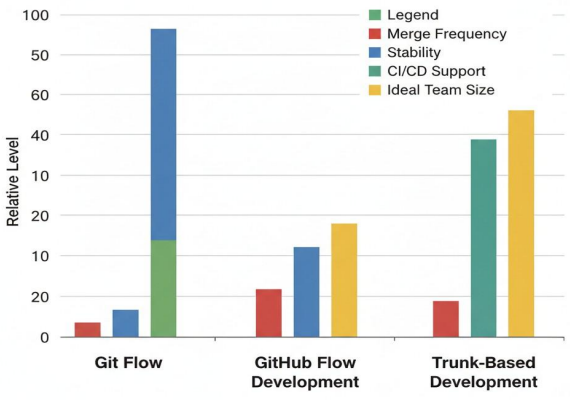
5.1. Findings

• **Git Flow:** Provided strong release control but delayed integration and created more maintenance work.
• **GitHub Flow:** Offered a good balance of stability and speed but required continuous testing.
• **Trunk-Based Development:** Reduced merge conflicts and improved collaboration speed, although it relied significantly on automation.

5.2. Graphical Interpretation

A conceptual graph (Figure 1) illustrates the inverse relationship between merge frequency and stability. As integration frequency rises, stability tends to vary, but overall delivery speed improves.
These findings support the observations of Nelson et al. (2019), who noted that early integration helps reduce merge conflicts. The proposed framework validates existing theories while providing new experimental insights.

Figure 1: Workflow Comparison Chart (Git Flow vs vs Trunk-Bow vs Trunk-Flow)



IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 5, Issue 1, December 2025 Impact Factor: 7.67

VI. CONCLUSION

This research shows that the effectiveness of version control workflows mainly depends on project size, level of automation, and team collaboration styles. Trunk-Based Development proves to be the most effective for agile environments, while Git Flow is better suited for structured, long-term management. GitHub Flow strikes a balance, making it ideal for mid-sized teams working in DevOps.

VII. FUTURE SCOPE

Future research could expand this study by:

- Using data from real-world repositories through Git APIs.
- Exploring AI-driven models to predict conflicts and automate merge resolutions.
- Conducting surveys in organizations to understand the link between developer satisfaction and workflow choice.
- Integrating DevOps metrics like deployment frequency and lead time to evaluate continuous delivery efficiency.

REFERENCES

[1] P. G. A. N. Gowda, "Git branching and release strategies," International Journal of Innovative Research in Engineering & Multidisciplinary Physical Sciences, 10 (5), 1–8. https://doi. org/10.5281/zenodo, vol. 14221771, 2022.
[2] S. K. Devineni et al., "Version control systems (vcs) the pillars of modern software development: Analyzing the past, present, and anticipating future trends," International Journal of Science and Research (IJSR), pp. 1816–1829, 2020.
[3] J. D. Blischak, E. R. Davenport, and G. Wilson, "A quick introduction to version control with git and github," PLoS computational biology, vol. 12, no. 1, p. e1004668, 2016.
[4] P. Lopes, P. Accioly, P. Borba, and V. Menezes, "Choosing the right git workflow: A comparative analysis of trunk-based vs. branch-based approaches," arXiv preprint arXiv:2507.08943, 2025.
[5] N. Nelson, C. Brindescu, S. McKee, A. Sarma, and D. Dig, "The life-cycle of merge conflicts: processes, barriers, and strategies," Empirical Software Engineering, vol. 24, no. 5, pp. 2863–2906, 2019.
[6] E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, and S. Lahiri, "Deepmerge: Learning to merge programs," IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 1599–1614, 2022.
[7] C. Brindescu, I. Ahmed, C. Jensen, and A. Sarma, "An empirical investigation into merge conflicts and their effect on software quality," Empirical Software Engineering, vol. 25, no. 1, pp. 562–590, 2020.
[8] M. Ellis, S. Nadi, and D. Dig, "Operation-based refactoring-aware merging: An empirical evaluation," IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 2698–2721, 2022.
[9] M. Owhadi-Kareshk, S. Nadi, and J. Rubin, "Predicting merge conflicts in collaborative software development," in 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–11, IEEE, 2019.
[10] Q. Zhang, J. Li, J. Lin, J. Ding, L. Lin, and C. Qian, "Wizardmerge–save us from merging without any clues," arXiv preprint arXiv:2407.02818, 2024.
[11] G. Vale, H. Costa, and S. Apel, "Predicting merge conflicts considering social and technical assets," Empirical Software Engineering, vol. 29, no. 1, p. 24, 2024.
[12] T. Ji, L. Chen, X. Yi, and X. Mao, "Understanding merge conflicts and resolutions in git rebases," in 2020 IEEE 31st international symposium on software reliability engineering (ISSRE), pp. 70–80, IEEE, 2020