Lab 3 Report: Collaborative Development
(Student 3)

Sawan Kumar

1. Literature Review

In open-source software development, contributors work collaboratively without traditional organizational hierarchy. This decentralized structure introduces challenges related to coordination, code consistency, and conflict management. To address such issues, Git and well-defined workflows are essential. Git supports distributed development by enabling branching, version tracking, code reviews, and controlled integration.

Code maintainers play a central role in large-scale open-source projects. They review contributions, check for code quality, enforce consistency, and ensure that only verified changes are merged into the main branch. Without maintainers, the project could become unstable due to conflicting ideas, inconsistent coding styles, or accidental overwrites. Thus, maintainers act as gatekeepers who protect the long-term health and direction of the project.

Branching allows contributors to work independently on separate features without disturbing the stable version of the codebase. Contributors may not personally know or trust each other, so isolated development serves as a social mechanism to prevent conflicts and accidental interference. When the work is ready, merging reintroduces the feature into the stable branch, often accompanied by pull requests that serve as communication tools for discussion, review, and feedback.

In professional environments, similar workflows such as GitFlow and feature branching are used. These workflows ensure that code integration happens systematically and only after proper validation. Code reviews also serve as quality gates, catching bugs early and ensuring that changes align with the project's standards.

Overall, Git workflows serve both technical and social functions. They prevent conflicts, support transparency, enforce accountability, and protect the maintainability of the project. Without structured branching, review processes, and maintainers, collaborative software development would not scale effectively.

2. Part A: Version Control Log (Student 3)

Below is the Git command log executed by Student 3 while working on the `feature-dev` branch and generating a deliberate merge conflict.

S.No	Git Command and Description
1	`git init` – Initialize repository
2	`git add .` – Stage project files
3	`git commit -m "Initial commit"` – Save project baseline
4	`git checkout -b feature-dev` – Create development branch (Student 3 role)
5	Edit `config.txt` – Modify Version line to generate conflict
6	`git add config.txt` – Stage development changes
7	`git commit -m "Dev config update + version change"` – Commit Student 3's work
8	`git checkout main` – Switch to main for integration
9	`git merge feature-prod` – Merge prod first (done by Student 1)
10	`git merge feature-test` – Merge test branch
11	`git merge feature-dev` – Merge dev branch → conflict occurs
12	Manually edit `config.txt` – Resolve conflict
13	`git add config.txt` – Stage resolved file
14	`git commit -m "Resolved merge conflict from feature-dev"` – Finalize resolution

3. Part B: Critical Analysis (Student 3)

As Student 3, my primary responsibility was to simulate a merge conflict by modifying the same line of the configuration file that other students were also editing. This ensured that when all branches were merged, Git would detect conflicting changes and require manual resolution.

How the Merge Conflict Occurred

The conflict occurred because the `Version` line inside `config.txt` was modified differently in multiple branches:

- Student 1 updated it to: `1.0.1-stable`
- I (Student 3) updated it to: `1.1.0-alpha`

When Student 1 attempted to merge `feature-dev` into `main`, Git detected that both branches changed the same line in incompatible ways. Therefore, Git could not automatically decide which version to keep and inserted conflict markers into the file.

Conflict Markers in config.txt

```
<<<<<<< HEAD
Version = 1.0.1-stable
=======
Version = 1.1.0-alpha
>>>>>>> feature-dev
```

These markers show:

- `HEAD` = the version already in main branch
- The lower block = my version from `feature-dev`

Manual Conflict Resolution

To resolve the conflict, I opened `config.txt`, removed the conflict markers, and selected the correct version string. For project stability, the production version (`1.0.1-stable`) was chosen while preserving all environment configuration blocks (Test, Dev, Prod). After resolving the conflict:

- I saved the file
- Staged it with `git add config.txt`
- Committed the fix using `git commit`

This produced a clean and stable merge.

4. Appendix: Final config.txt

```
# Configuration File V1.0

[General]
Project_Name = My_Service
```

```
Version = 1.0.1-stable
Environment = DEFAULT

[Test_Settings]
Logging_Level = DEBUG
Database_URL = http://test-db:5432

[Dev_Settings]
Logging_Level = TRACE
Database_URL = http://localhost:8080

[Prod_Settings]
Logging_Level = INFO
Database_URL = https://prod-db:5432
```