Git Workflow Documentation with Examples

Introduction:

This document outlines our standardized Git workflow for project development, seamlessly integrating Git-flow with slide enhancements. Its purpose is to provide a comprehensive understanding of branching strategies, merging processes, and deployment procedures, fostering efficient collaboration and code management across our team.

Branching Model:

We adhere to a Git-flow branching model, strategically utilizing two long-lived branches:

- **main:** Represents the production-ready codeline, exclusively housing stable and thoroughly vetted releases. This ensures the highest quality and reliability in the production environment.
- **dev:** Serves as the primary development branch, creating a collaborative space for integrating and testing new features before deployment. It maintains a streamlined development process, enhancing efficiency and code stability."

1. Normal Workflow with QA and UAT Releases

1.1 Basic Normal Workflow

1.1.1 Create Feature Branches:

- Developers start by creating feature branches (e.g., `f1`, `f2`, `f3`) from the `dev` branch. These branches encapsulate the individual features being developed.

```
# Developer A
git checkout dev
git pull origin dev
git checkout -b f1

# Developer B
git checkout dev
git pull origin dev
git checkout -b f2

# Developer C
git checkout dev
git pull origin dev
git checkout -b f3
```

  - Each developer must take rebase every morning from the `dev` branch to keep the feature branch up to date.

```
# Developer A
git checkout f1
git pull origin dev
git rebase dev
```

  - If any conflict occurs during rebase, developer must resolve the conflict (communicate with other developers if necessary) and then continue the rebase process.

```
# Developer A
git rebase --continue
```

- After conflict resolve and rebase process complete, developer must push the changes to the remote repository without pull.

```
# Developer A
git push origin f1 --force
```

- If a feature requires collaboration from multiple developers, for instance, three developers assigned to feature `f1`, each developer can create their own feature branches from `f1`. For example:

```
# Developer A
git checkout -b f1-a f1

# Developer B
git checkout -b f1-b f1

# Developer C
git checkout -b f1-c f1
```

- Developers must perform a daily rebase from the main feature branch (f1) to synchronize their individual feature branches:

```
# Developer A
git checkout f1-a
git pull origin f1
git rebase f1

# Developer B
git checkout f1-b
git pull origin f1
git rebase f1

# Developer C
git checkout f1-c
git pull origin f1
git rebase f1
```

- Additionally, one developer (e.g., Developer A) is designated to perform a daily rebase from the `dev` branch to keep the main feature branch (`f1`) up to date:

```
# Developer A
git checkout f1
git pull origin dev
git rebase dev
```

- If any conflict occurs during rebase, developer must resolve the conflict and then continue the rebase process as mentioned above.

1.1.2 Work on Features:

- Developers work on their features independently within their respective branches.

1.1.3 Merge Request to Dev:

- Completed features are merged into the `dev` branch to consolidate the ongoing work by following **MR**.

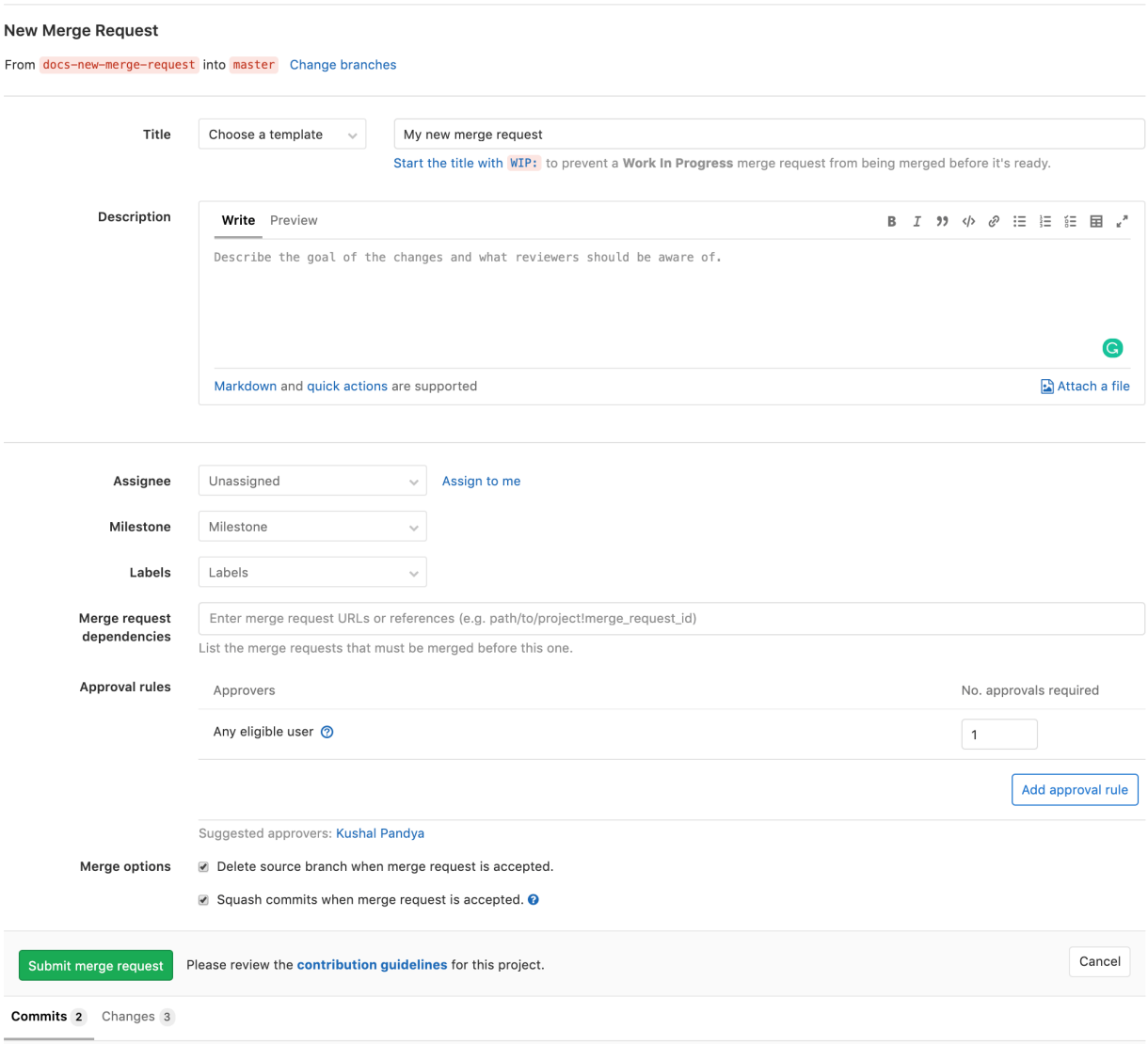
- Developers **must** follow these steps to give a **MR** (Merge Request) to the `dev` branch from their feature branch (e.g., `f1`).
    - To give **MR** to the `dev` branch, developer need to push all commit to their feature branch (e.g., `f1`) to the remote repository.

      ```
      git add .
      git commit -m "commit message"
      git push origin f1
      ```

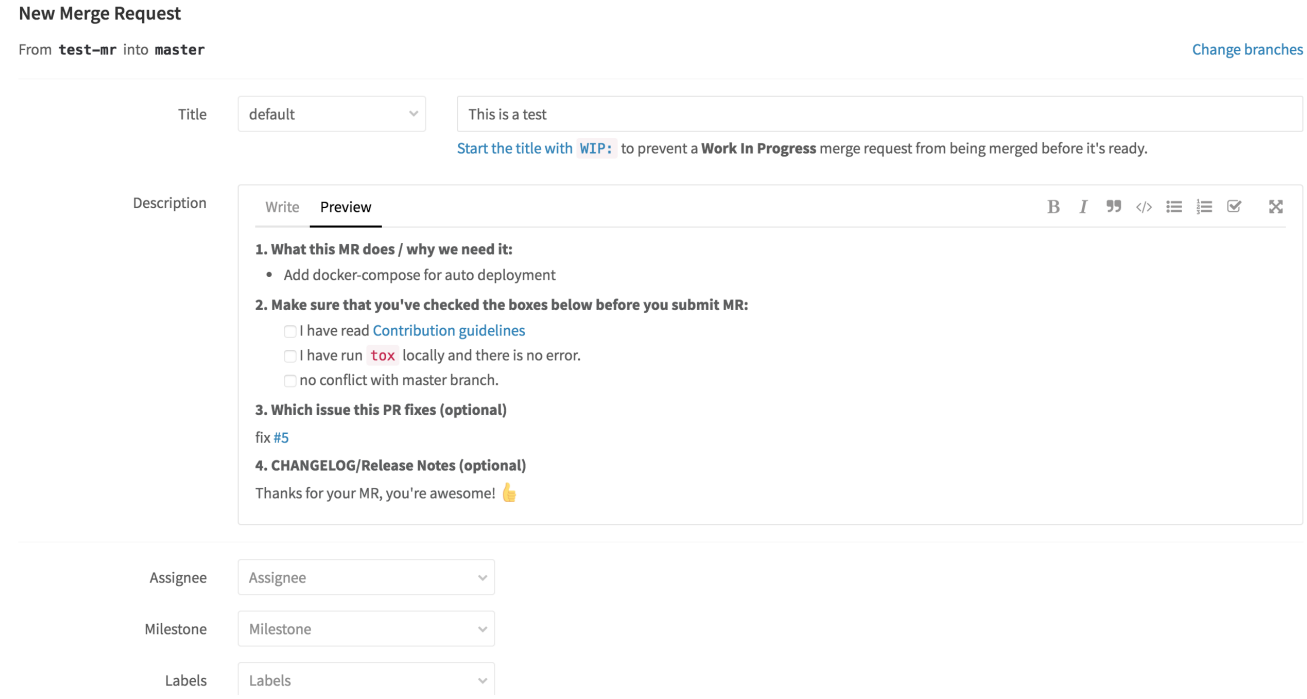
    - In the output, GitLab will prompt you with a direct link for creating a merge request:

      ```
      remote: To create a merge request for docs-new-merge-request, visit:
      remote:   https://gitlab-instance.com/my-group/my-project/merge_requests/new?
      merge_request%5Bsource_branch%5D=my-new-branch
      ```

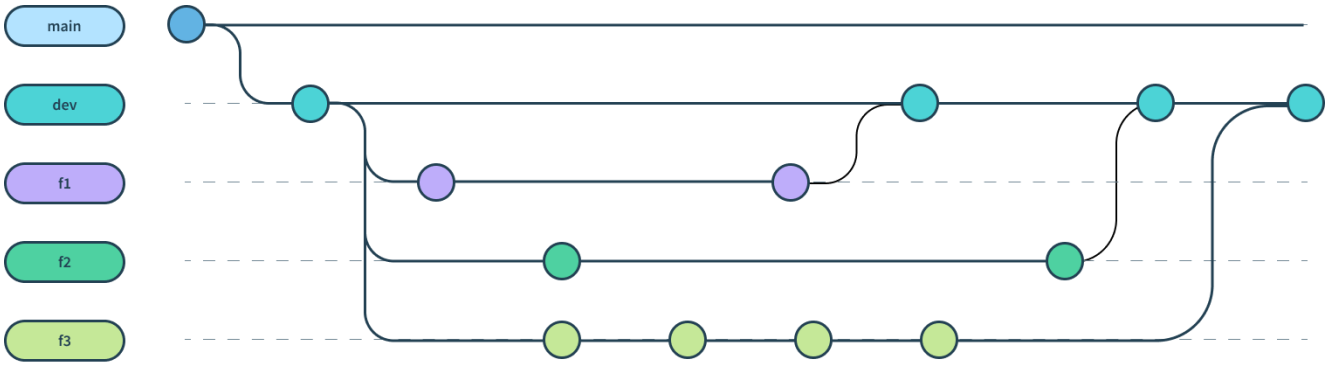
      Copy that link and paste it in your browser, and the New Merge Request page will be displayed. here is example image of the New Merge Request page:

    - During **MR** developer must include proper description of the changes in the **MR**.



Git flow diagram for developers workflow:



- If multiple developers are working on a single feature, such as `f1-a`, `f1-b`, and `f1-c`, they should first merge their individual branches into the main feature branch `f1`:

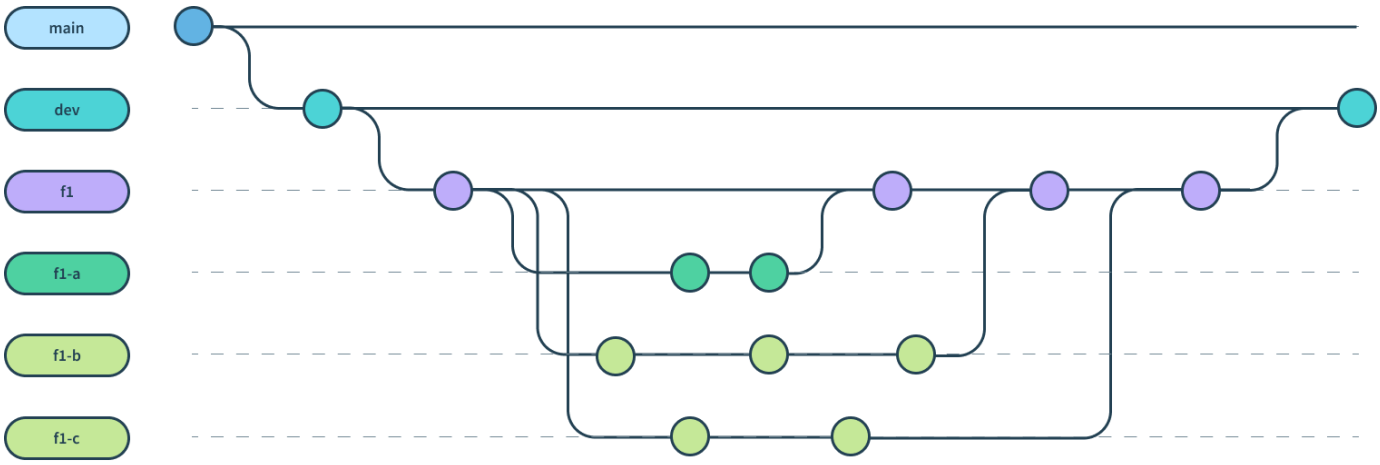
```
# Developer A (f1-a branch)
git checkout f1
git pull origin f1
git merge --squash f1-a
git commit -m "Merge branch 'f1-a' into f1"
git push origin f1

# Developer B (f1-b branch)
git checkout f1
git pull origin f1
git merge --squash f1-b
git commit -m "Merge branch 'f1-b' into f1"
git push origin f1

# Developer C (f1-c branch)
git checkout f1
git pull origin f1
git merge --squash f1-c
git commit -m "Merge branch 'f1-c' into f1"
git push origin f1
```

- After merging individual changes into the main feature branch (`f1`), one developer (e.g., Developer A) should create a **MR** to merge the feature into the `dev` branch:
- Assigned developer then review the merge request, ensuring the changes are accurate and meet the project's coding standards.
- Once the merge request is approved, the changes are merged into the `dev` branch.
- To know more about **MR** (Merge Request), please visit GitLab Merge Request

Git flow diagram for multiple developers in a single feature workflow:



1.1.4. **QA Testing:**

- A QA branch e.g.,(`qa_f1_f2_f3`) is created from `dev` for comprehensive testing by the QA team.

```
git checkout -b qa_f1_f2_f3 dev
```

- Push the QA deployment branch to the QA environment:

```
git push origin qa_f1_f2_f3
```

- QA team tests the changes in the QA environment.
- If issues are found during QA testing, they should be fixed directly in the QA deployment branch (`qa_f1_f2_f3`):

```
# Fix issues in the QA branch
git add .
# Make necessary changes
git commit -m "Proper commit message"
git push origin qa_f1_f2_f3
```

- Once QA testing is successful, the QA deployment branch (`qa_f1_f2_f3`) should be merged into the `dev` branch to ensure that the tested changes are included in the ongoing development.

1.1.5. **UAT Release:**

- Upon successful QA testing, a UAT Release branch e.g.,(`UAT_Release_f1_f2_f3`) is created from (`qa_f1_f2_f3`). This branch serves as the basis for User Acceptance Testing (UAT) before production deployment.

```
git checkout -b UAT_Release_f1_f2_f3 qa_f1_f2_f3
```

1.1.6 UAT to Main:

- After UAT approval, the UAT Release branch (e.g., `UAT_Release_f1_f2_f3`) is merged into `main` , marking the completion of the feature development cycle.
  - Once the UAT is successfully completed and the UAT Release branch is approved for production, it is merged into the `main` branch:

```
git checkout main
git merge --squash UAT_Release_f1_f2_f3
```

  - Additionally, a tag is created to mark the specific version associated with the UAT Release. This tag serves as a snapshot of the codebase at the point of the UAT-approved release:

```
git tag -a v1.0.0 -m "UAT_Release_f1_f2_f3 Approved"
git push origin main --tags
```

  - The created tag (`v1.0.0` in this example) provides a reference point for the specific release and is especially useful for tracking and reverting to specific versions in the future.

1.2 Alternative Paths

1.2.a Handling Issues in QA

- If issues arise during QA, they are fixed in the given e.g. `qa_f1_f2_f3` branch before proceeding with UAT.

1.2.b Handling UAT Failures and Enhancements

- In case of UAT failures requiring enhancements, changes are introduced in the `qa_f1_f2_f3` branch and then tested again and create a new UAT Release branch e.g.,(`UAT_Release_f1_f2_f3_enhance`) from the

updated `qa_f1_f2_f3` branch.

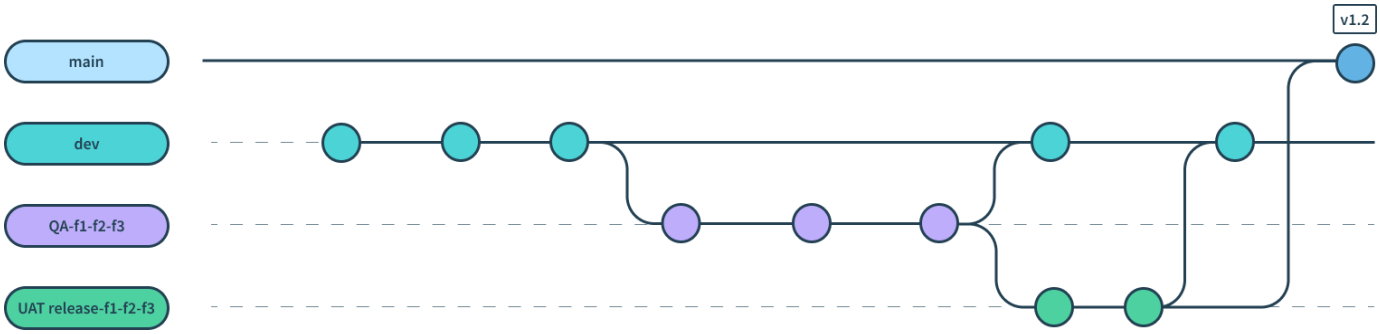
- Once the UAT is successfully completed and the UAT Release branch is approved for production, it is merged into the `main` branch with TAG and also merged into the `dev` branch to ensure that the tested changes are included in the ongoing development:

```
git checkout main
git merge --squash UAT_Release_f1_f2_f3_enhance

git tag -a v12.1.1 -m "UAT_Release_f1_f2_f3_enhance Approved"
git push origin main --tags

git checkout dev
git merge --squash UAT_Release_f1_f2_f3_enhance
```

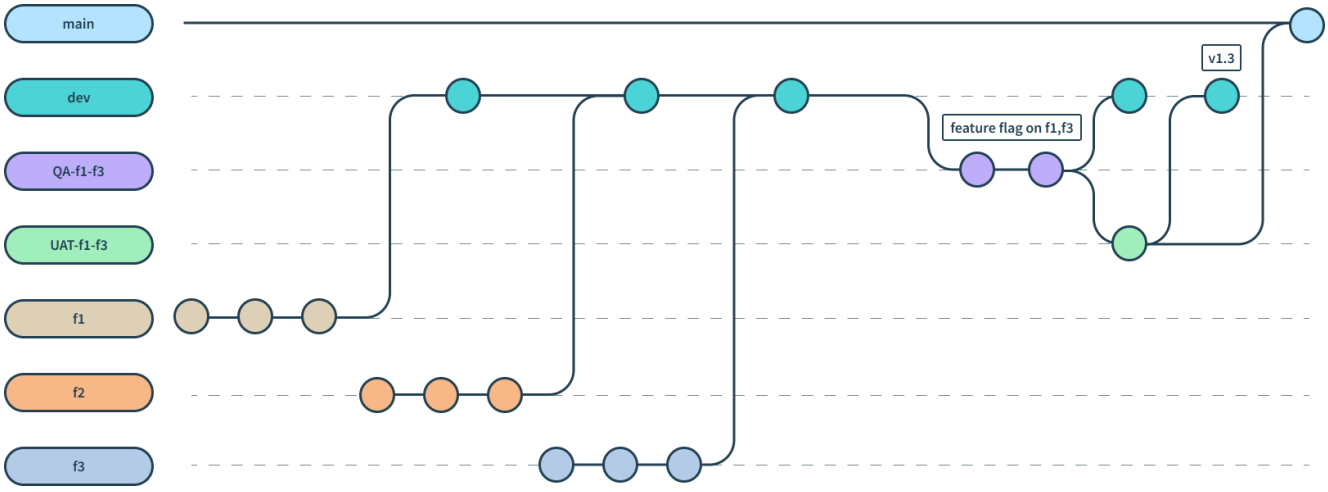
Git flow diagram for release cycle workflow:



1.2.c Feature Flagging

- Feature flagging is implemented in the e.g.,(`qa_fi_f3`) branch to control the accessibility of specific features during testing.

  Git flow diagram for specific feature release cycle workflow:



2. Hot Fixes

2.1 Main Hot Fix

2.1.1. **Create Hot Fix Branch:**

- A hotfix branch e.g.,(`master_hot_fix`) is created from the `main` branch for addressing critical issues in production.

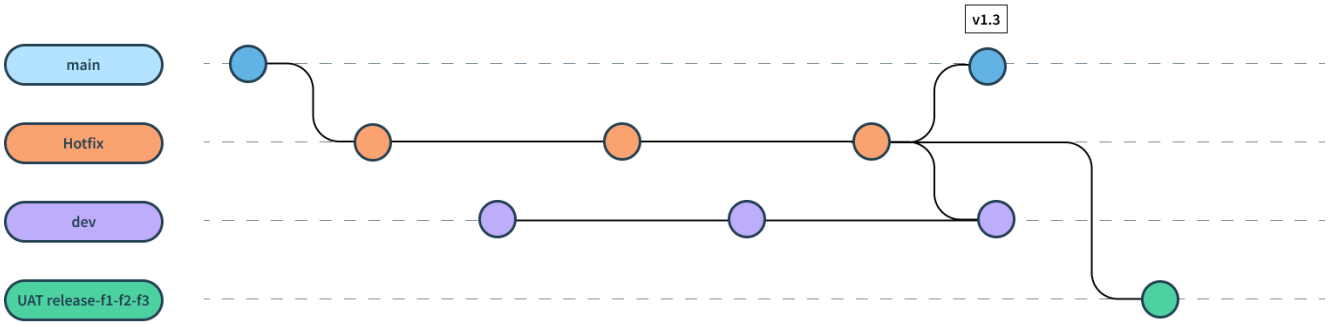
```
git checkout main
git pull origin main
git checkout -b master_hot_fix
```

2.1.2. **Fix and Merge:**

- The hotfix branch is used to fix the issue and is merged back into both `main` and `dev`.

```
git checkout main
git merge --squash master_hot_fix
git checkout dev
git merge --squash master_hot_fix
```

Git flow diagram for hotfix release cycle workflow:



3. Daily Rebase

3.1. **Rebase from Dev:**

- Developers are encouraged to perform a daily rebase from the `dev` branch to synchronize their feature branches with the latest changes.

```
git checkout f1
git pull origin dev
git rebase dev
```

  - If any conflict occurs during rebase, developer must resolve the conflict and then continue the rebase process.

```
# Developer A
git rebase --continue
```

  - After conflict resolve and rebase process complete, developer must push the changes to the remote repository without pull.

```
# Developer A
git push origin f1 --force
```

4. Commit Guidelines

4.1. **Single Logical Change:**

- Commits should represent a single logical change. If a commit contains multiple unrelated changes, the merge request will be cancelled.

- Writing clear and concise commit messages is essential for maintaining a clean and understandable Git history. Here's an example of a well-structured and descriptive commit message:

```
feat: implement user authentication with OAuth2

- Added OAuth2 authentication for user login
- Integrated third-party OAuth provider for secure authentication
- Updated user model to store OAuth credentials
- Implemented login and logout endpoints for OAuth authentication

Fixes #123
```

In this example:

- **Type:** `feat` indicates that it's a new feature.
- **Summary:** "implement user authentication with OAuth2" succinctly describes the purpose of the commit.
- **Details:** Bulleted list provides additional information about the changes made.
- **Reference:** "Fixes #123" references an associated issue number for more context.

Remember, commit messages should be clear, specific, and focused on a single logical change. They help others (and your future self) understand the purpose and impact of each commit.

1.2 Environment Management:

1.2.1 Dev Environment:

- Dedicated for developers' testing. Code is deployed into the development environment after feature branches are merged into dev ensuring a systematic process where developers' changes undergo testing in the development environment after approval and merging into the dev branch..

1.2.2 QA Environment:

- QA deployments are prohibited directly from the `dev` branch to maintain a clear separation between development and testing environments.
- QA testing should be conducted in a dedicated QA environment.

1.2.3 Hotfix Environment:

- Environment closely resembling the production setup. Allows comprehensive testing in an environment mirroring production conditions.

Conclusion:

By following this Git workflow, we aim to maintain a structured development process that facilitates collaboration, testing, and reliable production releases. Adhering to these guidelines ensures a clean version history and efficient management of feature development and bug fixes.