Branch Description

1. Main Branch (`main`)

- **Purpose**:
  - The `main` branch holds the **production-ready** version of your project.
  - It should always be **stable** and contain **tested** code that can be deployed to production.
  - This branch represents the official release of the project and should not be modified without proper review and testing.
- **Key Guidelines**:
  - **No direct commits**: Developers should never push directly to `main`. All changes should come from pull requests (PRs), usually from `staging` after final testing.
  - **Release-ready**: Every commit in `main` should be fully tested and considered ready for public use.
  - **Tagging**: Once new code is merged into `main`, it's a good practice to tag the commit with a version number for easier rollback or reference (e.g., `v1.0.0`).
  - **Emergency fixes**: Hotfixes for critical issues can be merged directly into `main` if necessary, but they should also be merged into `dev` and `staging` to maintain consistency across branches.

2. Development Branch (`dev`)

- **Purpose**:
  - The `dev` branch is the **integration** branch where developers consolidate their work before it moves on to further testing or production.
  - It serves as the main branch for ongoing feature development, bug fixes, and improvements.
- **Key Guidelines**:
  - **Starting point for feature branches**: New features, bug fixes, and other work should branch off from `dev`.
  - **Frequent updates**: The `dev` branch may have frequent updates and merges from multiple feature branches.
  - **Semi-stable**: While `dev` may not always be production-ready, it should be in a semi-stable state. Avoid breaking changes that would prevent testing or further development.
  - **Regular merges into staging**: Once the features and fixes in `dev` are complete and reviewed, they are merged into `staging` for testing.

3. Staging Branch (`staging`)

- **Purpose**:
  - The `staging` branch is the **pre-production** branch used for final testing. It contains code that is ready for the final round of testing before being deployed to production (merged into `main`).
  - It reflects what will be deployed, allowing developers and testers to ensure everything is functioning correctly under production-like conditions.
- **Key Guidelines**:
  - **Mirror production**: The code in `staging` should be as close as possible to what will be deployed in `main`. This includes testing configurations, integrations, and dependencies.
  - **Final bug fixes**: Any last-minute bug fixes or adjustments can be made in `staging`, but ideally, these changes should first be reflected in `dev` and then merged back into `staging`.
  - **Release candidate**: `staging` should be considered a "release candidate" branch, where only code that is feature-complete and stable is merged.
  - **Periodic testing**: Run integration tests, user acceptance testing, and performance tests on `staging` before merging into `main`.

4. Feature Branches (`feature/`)

- **Purpose**:
  - A feature branch is a temporary branch created to develop a specific feature or address a particular issue.
  - Feature branches allow developers to work on new features or fixes in isolation from the rest of the team.
- **Key Guidelines**:
  - **Branching from `dev`**: All feature branches should be created from the latest `dev` branch. This ensures they are built on the most up-to-date version of the code.
  - **Descriptive names**: Feature branches should have clear, descriptive names that explain what the branch is for. Common naming conventions include:
    - `feature/login-auth`
    - `feature/user-dashboard`
    - `feature/issue-1234`
  - **Isolated development**: Each feature branch should be focused on one task or feature. This keeps changes isolated and makes it easier to test and review.

    - **Merging into `dev`**: Once the feature is complete and tested in isolation, it is merged back into `dev`. Code reviews and tests are essential before merging to avoid introducing bugs.
    - **Short lifespan**: Feature branches should have a short lifespan and be deleted after merging into `dev` to avoid clutter in the repository.

5. Additional Branch Types (Optional)

If needed, your project could also adopt some additional branches depending on the workflow:

- **Hotfix Branches (`hotfix/`)**:
    - **Purpose**: Used to fix critical bugs or issues that need to be addressed immediately in production (`main`).
    - **Branching from `main`**: Hotfix branches are typically branched directly from `main` and merged back into both `main` and `dev` to ensure the fix is applied across the codebase.
    - **Naming Convention**: Use names like `hotfix/critical-issue` to specify the bug or problem being fixed.
- **Release Branches (`release/`)**:
    - **Purpose**: A release branch is created to prepare a new production release, often used for final adjustments.
    - **Branching from `staging`**: Release branches are created when a group of features is complete, and they are ready for deployment. It allows last-minute updates without affecting other branches.
    - **Merging into `main` and `dev`**: After testing and any final changes, release branches are merged into `main` for deployment and `dev` to keep it updated.
- **Bugfix Branches (`bugfix/`)**
    - **Purpose:** Bugfix branches are used to address non-critical bugs that arise during the development process but do not require an immediate fix in production.
    - **Branching from `dev`:** These branches are created from the `dev` branch and focus on fixing issues that have been identified during development or testing.
    - **Naming Convention:** Use names like `bugfix/issue-number-description` to specify the issue being resolved. For example, `bugfix/34-fix-button-click`.
    - **Merging into `dev`:** Once the bug has been resolved and tested, the bugfix branch should be merged back into `dev`, ensuring that the fixes are integrated into the ongoing development workflow.

Github workflow description (feature branch -> main)

1. Creating a Feature Branch

1. **Start from the `dev` branch**:
    - Ensure you are working on the latest version of the `dev` branch. This ensures your new feature is built on the most current code.
2. **Create a new feature branch**:
    - A new branch should be created from `dev` with a descriptive name based on the feature or task you are working on (e.g., `feature/user-authentication`). Using clear names helps the team understand the purpose of each branch.
3. **Develop your feature**:
    - Implement your feature or task, committing your changes locally as you progress. Regular commits with detailed messages are important for keeping a record of changes and making it easier to review the code later.
4. **Push the feature branch to GitHub**:
    - Once you have made progress, push your feature branch to GitHub. This allows others to track the work and offer feedback or collaborate if necessary.

Explanation:

Creating a feature branch for each new task allows for independent development without disrupting the main codebase. The isolation helps ensure that incomplete or unstable code doesn't affect the rest of the project.

2. Submitting a Pull Request (PR) to `dev`

Once a feature is complete, you will merge it back into the main development branch (`dev`). This is done via a **Pull Request (PR)**, which is a formal request to merge your changes into the project. PRs are critical for code review and testing.

Steps:

1. **Open a Pull Request (PR)**:
    - After pushing your branch to GitHub, create a pull request targeting the `dev` branch. In the PR description, explain the feature and list any changes made. Be specific about what was added or modified, and provide any context necessary for reviewers.
2. **Review Process**:
    - Other contributors or team members will review your PR. They may leave comments, suggest improvements, or request changes. This peer review process

is vital for catching potential bugs, ensuring code quality, and aligning with project standards.
    - If changes are requested, update your branch with the necessary changes and push the updated code. The pull request will automatically update.
3. **Merging the PR**:
    - Once the pull request has been approved, it can be merged into the `dev` branch. After merging, the feature branch is no longer needed and should be deleted to avoid clutter in the repository.

Explanation:

Pull requests serve as a checkpoint for reviewing code and ensuring that it meets the project's quality standards. The review process ensures that other developers can provide feedback and confirm that the feature is ready to be integrated into the development environment.

3. Merging `dev` into `staging`

1. **Update `dev`**:
    - Before initiating a pull request to merge into `staging`, ensure that the `dev` branch is up-to-date. This guarantees that the latest changes are included in the testing phase.
2. **Create a Pull Request to Merge into `staging`**:
    - Switch to the `staging` branch and create a pull request (PR) to merge the latest changes from `dev`. This allows you to review the code changes before merging and facilitates collaboration within the team.
3. **Testing on `staging`**:
    - Use the `staging` branch to conduct thorough testing, such as user acceptance testing (UAT), integration tests, or performance tests. Testing in the `staging` environment helps confirm that all the new features and fixes work together as expected.

Explanation: Merging `dev` into `staging` via a pull request allows the team to conduct thorough testing in an environment that simulates production, but without the risk of affecting live users. Testing on `staging` ensures that the new code is stable and ready for deployment.

4. Merging `staging` into `main`

1. **Ensure `staging` is Up-to-Date**:
    - Before creating a pull request to merge into `main`, confirm that the `staging` branch is fully up-to-date and includes all the latest features and fixes.
2. **Create a Pull Request to Merge `staging` into `main`**:
    - Switch to the `main` branch and initiate a pull request (PR) to merge the `staging` branch into it. This process allows for code review and ensures that newly tested code is incorporated into production.
3. **Tag the Release**:
    - After the PR is approved and the merge is complete, it's a good practice to create a version tag (e.g., `v1.0.0`). Tagging the release helps track deployments and makes it easier to roll back to a previous state if needed.
4. **Deploy the Code**:
    - Once merged into `main`, the code is now ready for deployment to live users. Depending on your deployment strategy, this may happen automatically or require manual intervention.

Explanation: Merging into `main` via a pull request signifies that the code is stable and ready for production. Creating version tags and deploying from `main` ensures that only fully tested and approved changes make it into the live environment.

5. Security

For managing a community repository, security is essential to ensure code quality and prevent malicious or accidental changes. Here's a set of best security practices for configuring branches in such repositories:

General Rules for All Branches

- **Restrict Direct Commits**: Ensure that no direct commits are allowed to any of the key branches (`dev`, `staging`, or `main`). All changes should happen through pull requests (PRs).
- **Require Status Checks and Reviews**: Require passing checks and reviews before merging, as described earlier for strict security.

Configure `dev` Branch Settings

The `dev` branch is where active development occurs, and this is the only branch from which you want to merge into `staging`.

- **Branch Protection Settings**:
    1. Require **Pull Request** for all changes to `dev`.
    2. **Require 2 reviews** before merging. This ensures that multiple contributors review the new code.
    3. **Allow PR merges only from feature branches**:
        - Instruct contributors to create feature branches (e.g., `feature/branch_name`), which are then merged into `dev` via PRs.
    4. **Require all status checks** to pass (automated tests, CI checks, etc.).
    5. **Restrict force pushes** and **disallow branch deletion** for `dev`.

Configure `staging` Branch Settings

The `staging` branch is used for final integration and testing. Only PRs from `dev` should be merged into `staging`.

- **Branch Protection Settings**:
    1. Require **Pull Request** for all changes to `staging`.
    2. **Allow merges only from the `dev` branch**:
        - To restrict this, you can configure workflows (e.g., GitHub Actions) that block any PRs not originating from `dev`.
        - Manually enforce the rule that merges into `staging` can only come from `dev`. Contributors should follow this process, or you can use automated checks.
    3. **Require 3 reviews** before merging to ensure thorough code review (e.g., from developers, testers, and security).
    4. **Require all status checks** to pass (including security and integration tests).
    5. **Disallow force pushes** and **branch deletions**.
    6. **Require branches to be up to date** before merging to ensure `staging` is always consistent with the latest state of `dev`.

Configure `main` Branch Settings

The `main` branch represents production-ready code. It should only accept merges from `staging`, except for hotfix branches which can be merged directly into `main` to resolve urgent issues.

- **Branch Protection Settings**:
    1. Require **Pull Request** for all changes to `main`.
    2. **Allow merges only from `staging` and hotfix branches**:
        - Implement this rule manually via branch policies or configure a workflow using GitHub Actions or a similar tool to enforce that merges into `main` can only come from:

        - `staging`: For regular updates.
        - `hotfix/*`: For urgent hotfixes that must be applied directly.
    3. **Require 4 reviews** (e.g., from developers, QA, security, and project owners) before merging to `main`.
    4. **Require all status checks to pass**, ensuring no broken code is deployed to production.
    5. **Require signed commits and signed tags** for additional verification.
    6. **Disallow force pushes** and **branch deletions**.
    7. **Restrict who can push**: Only senior maintainers or admins should have permission to merge into `main`.
    8. **Allow hotfix feature branches**:
        - Ensure hotfix branches follow a naming convention (e.g., `hotfix/bug_name`).
        - Allow PRs from `hotfix` branches directly into `main` for urgent patches. These PRs must still follow the same rigorous review and status check requirements as `staging` merges.

Additional Security Settings for All Branches

- **Require 2FA for All Contributors**: Enforce two-factor authentication for every contributor.
- **Enable CodeQL Analysis**: Use GitHub's advanced security features to scan your codebase for vulnerabilities.
- **Dependabot Alerts**: Ensure Dependabot is set up to monitor your dependencies for security vulnerabilities across all branches.

Summary of Configuration

Branch	Allowed Merges From	Reviews Required	Status Checks	Signed Commits	Branch Rules
`dev`	Feature branches	2	Yes	Yes	No direct commits, disallow force push
`staging`	`dev`	3	Yes	Yes	No direct commits, only from `dev`
`main`	`staging`, `hotfix/*`	4	Yes	Yes	No direct commits, only from `staging` or `hotfix/*`

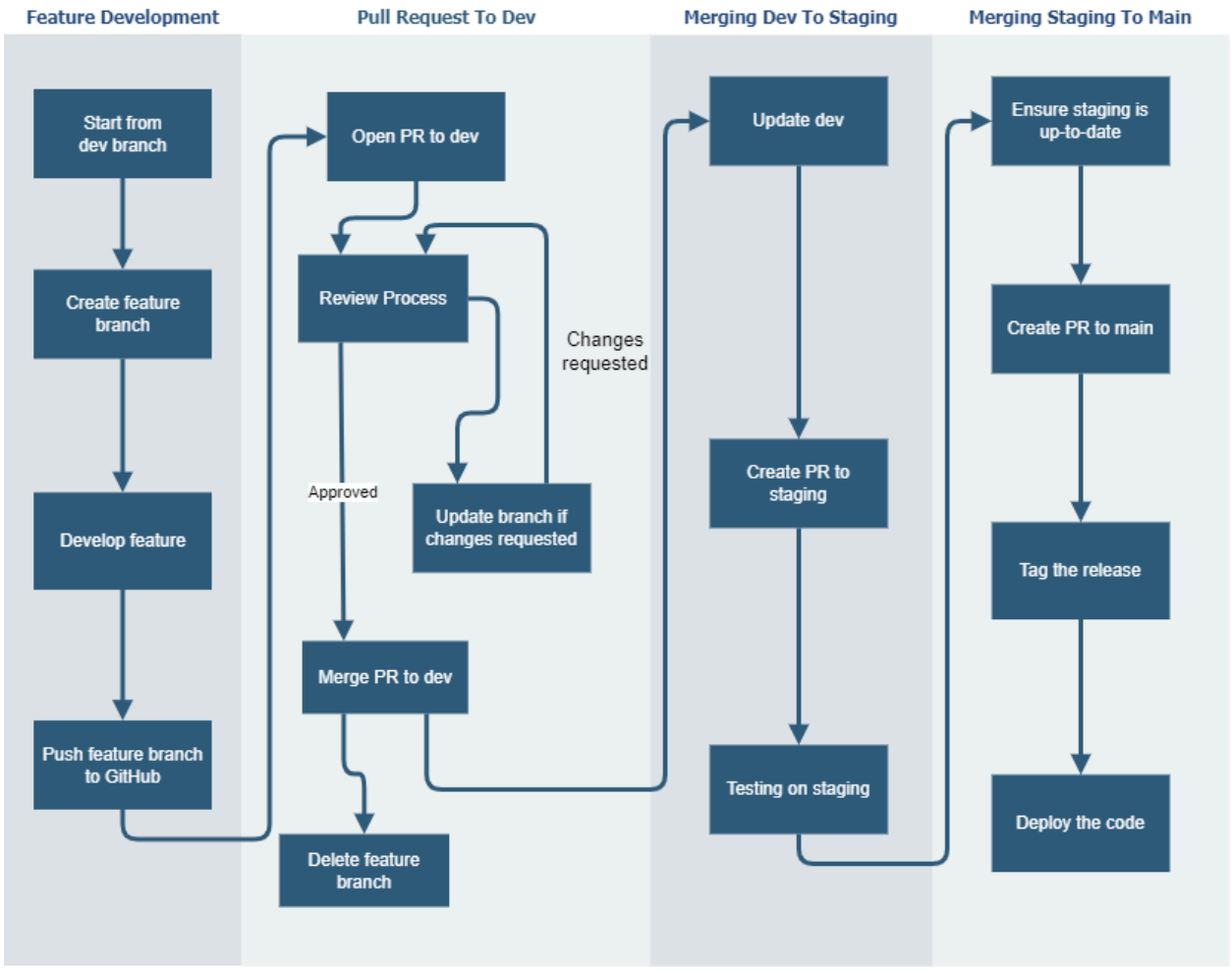
5. Best Practices for GitHub Workflow

- **Use Branches**: Always develop new features in their own branch. Don't work directly on `dev`, `staging`, or `main` to avoid accidental changes to the project's main codebase.
- **Regular Commits**: Commit your changes frequently, with clear and descriptive messages. This makes it easier to track progress and troubleshoot if needed.
- **Peer Reviews**: All code should be reviewed by peers before merging. This ensures that another set of eyes has verified the work, reducing the chances of bugs and errors in production.
- **Continuous Integration (CI)**: Use automated testing tools integrated with GitHub (e.g., GitHub Actions) to automatically run tests on each pull request. This helps identify issues early in the development process.
- **Keep Branches Updated**: Regularly pull updates from `dev` into your feature branches to keep them aligned with the latest changes. This prevents conflicts when it's time to merge.
- **Tag Releases**: Use Git tags to mark important releases, especially when merging into `main`. Tags allow for easier tracking and deployment of specific versions of the project.

Summary of the Workflow

1. **Feature Development**: Work on new features in a separate feature branch.
2. **Pull Request**: Create a PR to merge the feature into `dev`, followed by code review and testing.
3. **Staging**: Merge `dev` into `staging` for final testing in a production-like environment.
4. **Main**: Once testing is complete, merge `staging` into `main` for production release.
5. **Release**: Tag and deploy the code from `main`.

This workflow keeps the project organized, ensures a high level of code quality, and allows for easy collaboration within the community. Each step is designed to prevent issues from reaching production while allowing contributors to work independently on their respective tasks.


Feature Development
Pull Request To Dev
Merging Dev To Staging
Merging Staging To Main
Start from dev branch
Create feature branch
Develop feature
Push feature branch to GitHub
Open PR to dev
Review Process
Changes requested
Approved
Update branch if changes requested
Merge PR to dev
Delete feature branch
Update dev
Create PR to staging
Testing on staging
Ensure staging is up-to-date
Create PR to main
Tag the release
Deploy the code