COMP30022

IT Project

- Development Practice

Development

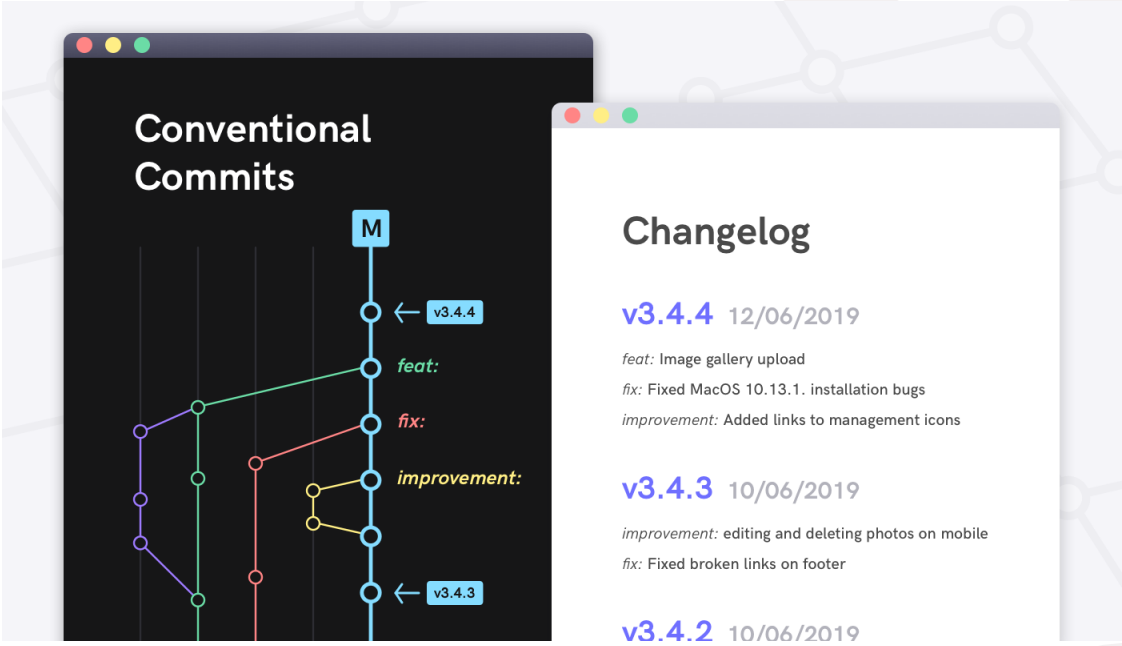
Git Review

- Basic
  - https://youtu.be/USjZcfj8yxE?si=eUhcTIYBncKLnae_
  - https://youtu.be/HkdAHXoRtos?si=w20qFqqLKZOJvAs3
- Pull Request
  - https://youtu.be/z8CYDyFqzp0?si=EvBH_E4fVqwW57dN
  - https://youtu.be/rgbCcBNZcdQ?si=_F87AUuqs3-e1EjW

Conventional Commits

- A standardised commit message convention for clear, semantic, and machine-readable versioning
- Streamline versioning, release management, and improve overall project communication

- Resources:
  - https://www.pixelmatters.com/blog/conventional-commits-dynamic-changelog
  - https://simi.studio/en/git-commit-message-guidelines/



Structure

- <type>(optional scope): <description>

Types

- `feat` : new features
- `fix` : bug fixes
- `chore` : routine tasks

```
➜  commit-msg-linter-test git:(master) ✗ gcam "update README"

  ************* Invalid Git Commit Message **************
  commit message: update README
  correct format: <type>[scope]: <subject>
  example: docs: update README to add developer tips

  type:
    feat     A new feature.
    fix      A bug fix.
    docs     Documentation only changes.
    style    Changes that do not affect the meaning of the code (white-space, formatting,
    refactor A code change that neither fixes a bug nor adds a feature.
    test     Adding missing tests or correcting existing ones.
    chore    Changes to the build process or auxiliary tools and libraries such as docume
    perf     A code change that improves performance.
    ci       Changes to your CI configuration files and scripts.
    build    Changes that affect the build system or external dependencies (example scope
    temp     Temporary commit that won't be included in your CHANGELOG.

  scope:
    Optional, can be anything specifying the scope of the commit change.
    For example $location, $browser, $compile, $rootScope, ngHref, ngClick, ngView, etc.
    In App Development, scope can be a page, a module or a component.

  subject:
    Brief summary of the change in present tense. Not capitalized. No period at the end.
```

GIT COMMIT MESSAGE GUIDELINES

Conventional Commits

A standardised commit message convention for **clear**, semantic, and **machine-readable** versioning

Structure

- <type>(optional scope): <description>

Types

- `feat` : new features
- `fix` : bug fixes
- `chore` : routine tasks

Examples

Commit message with description and breaking change footer

```
feat: allow provided config object to extend other configs

BREAKING CHANGE: `extends` key in config file is now used for extending other config files
```

Commit message with `!` to draw attention to breaking change

```
feat!: send an email to the customer when a product is shipped
```

Commit message with scope and `!` to draw attention to breaking change

```
feat(api)!: send an email to the customer when a product is shipped
```

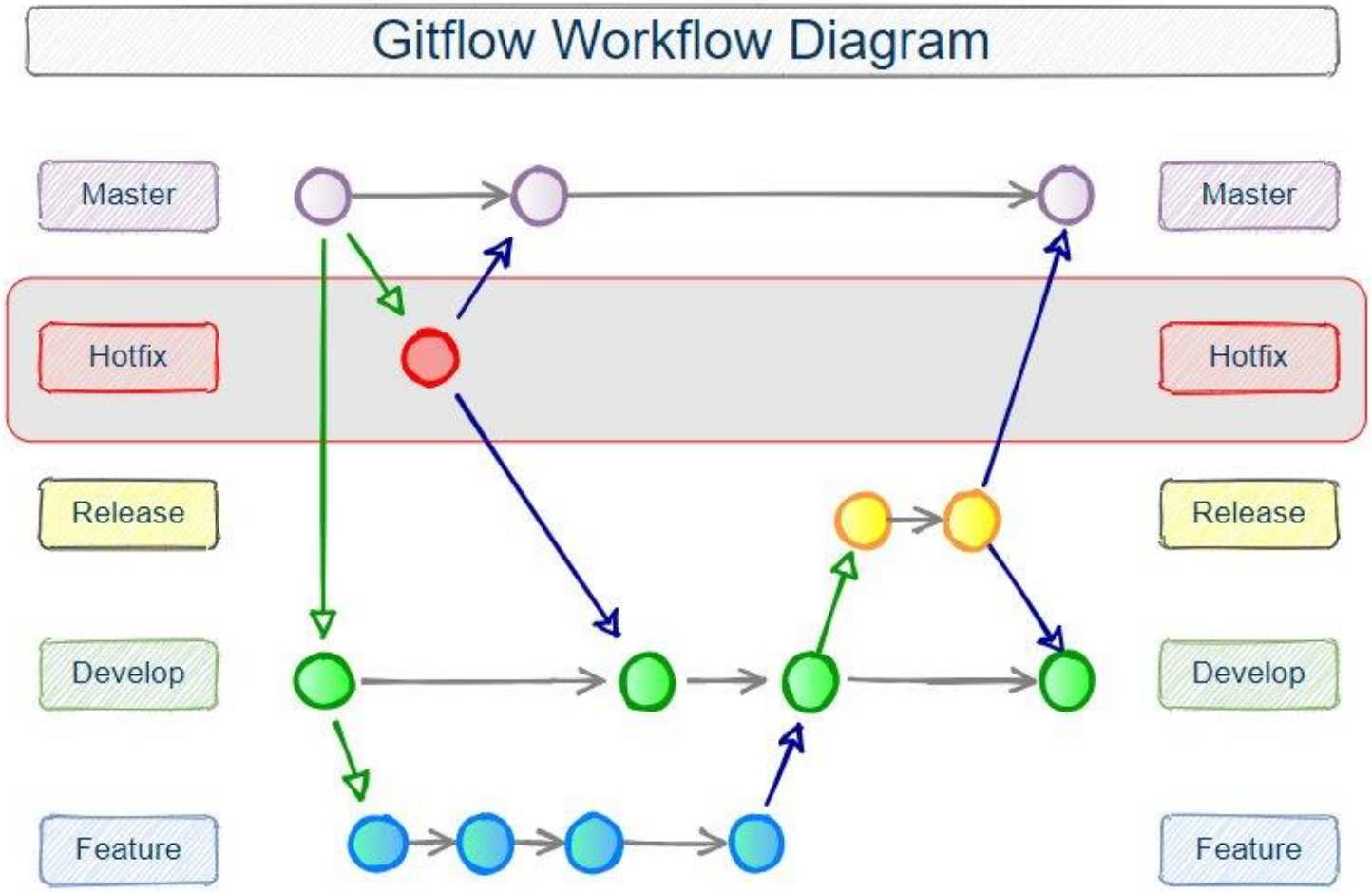
Commit message with both `!` and BREAKING CHANGE footer

```
chore!: drop support for Node 6

BREAKING CHANGE: use JavaScript features not available in Node 6.
```

Branching

- Resources
  - https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
  - https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/Gitflow-release-branch-process-start-finish
  - https://medium.com/@sreekanth.thummala/choosing-the-right-git-branching-strategy-a-comparative-analysis-f5e635443423



In industrial or professional software development, branching conventions and techniques are critical for maintaining a clean, organized, and efficient workflow in Git. Here are some widely used branching strategies and conventions:

1. Git Flow

Git Flow is a popular branching model introduced by Vincent Driessen. It is suitable for projects with a release cycle and is widely adopted in many industries. The key branches in Git Flow are:

- **Main (Master)**: The `**main**` branch (sometimes called `**master**`) is the stable branch where the source code always reflects a production-ready state.
- **Develop**: The `**develop**` branch serves as an integration branch for features. It reflects the latest delivered development changes for the next release.
- **Feature Branches**: These branches are created from `**develop**` for working on individual features. Once a feature is complete, it is merged back into `**develop**`.
- **Release Branches**: When a new release is ready, a `**release**` branch is created from `**develop**`. This branch is used for final testing and bug fixes before merging into `**main**`.
- **Hotfix Branches**: These branches are created from `**main**` to address critical issues in production. After the fix, they are merged back into both `**main**` and `**develop**`.

In industrial or professional software development, branching conventions and techniques are critical for maintaining a clean, organized, and efficient workflow in Git. Here are some widely used branching strategies and conventions:

1. Git Flow

2. GitHub Flow

GitHub Flow is a simpler branching model, ideal for continuous delivery and projects that are regularly deployed. Key elements include:

- **Main Branch**: The `**main**` branch contains the production-ready code. All changes that are deployed come from this branch.
- **Feature Branches**: Every new feature, bug fix, or experiment is developed in a separate branch created from `**main**`. These branches are usually named after the feature or issue they address.
- **Pull Requests**: Once a feature is complete, a pull request is opened to merge the feature branch into `**main**`. Code review and automated testing typically happen at this stage.
- **Continuous Deployment**: After the pull request is merged, the changes can be automatically deployed to production.

In industrial or professional software development, branching conventions and techniques are critical for maintaining a clean, organized, and efficient workflow in Git. Here are some widely used branching strategies and conventions:

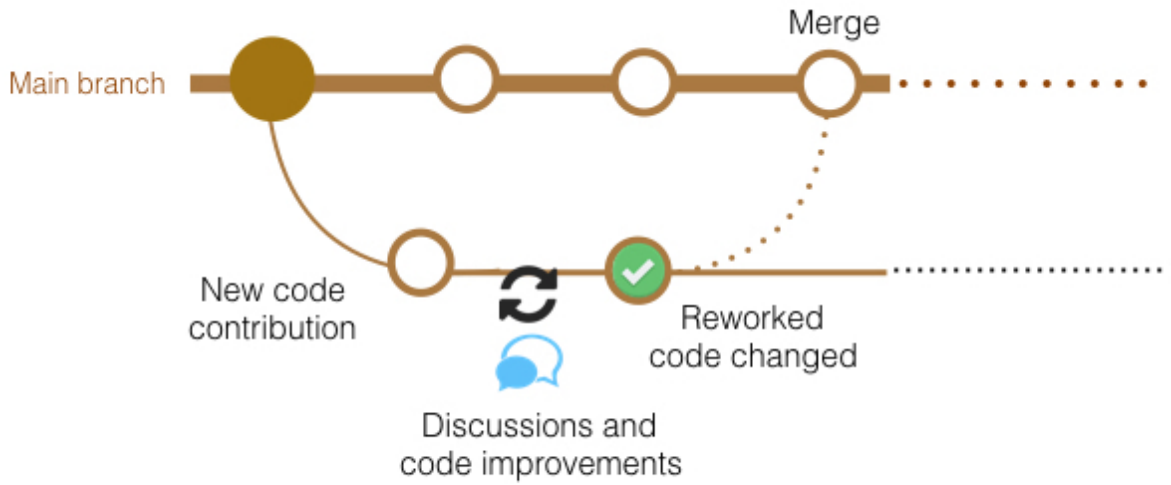
Best Practices for Branching Conventions

- **Consistent Naming**: Use consistent and descriptive naming conventions for branches, like `**feature/login-page**`, `**bugfix/fix-signup-error**`, or `**release/v1.0.0**`.
- **Small, Focused Changes**: Keep branches small and focused on a single feature or fix to simplify merging and code reviews.
- **Regular Merging**: Regularly merge changes from `**main**` into feature branches to keep them up-to-date and minimize conflicts.
- **Automation**: Leverage CI/CD tools to automatically test and deploy changes when branches are merged.

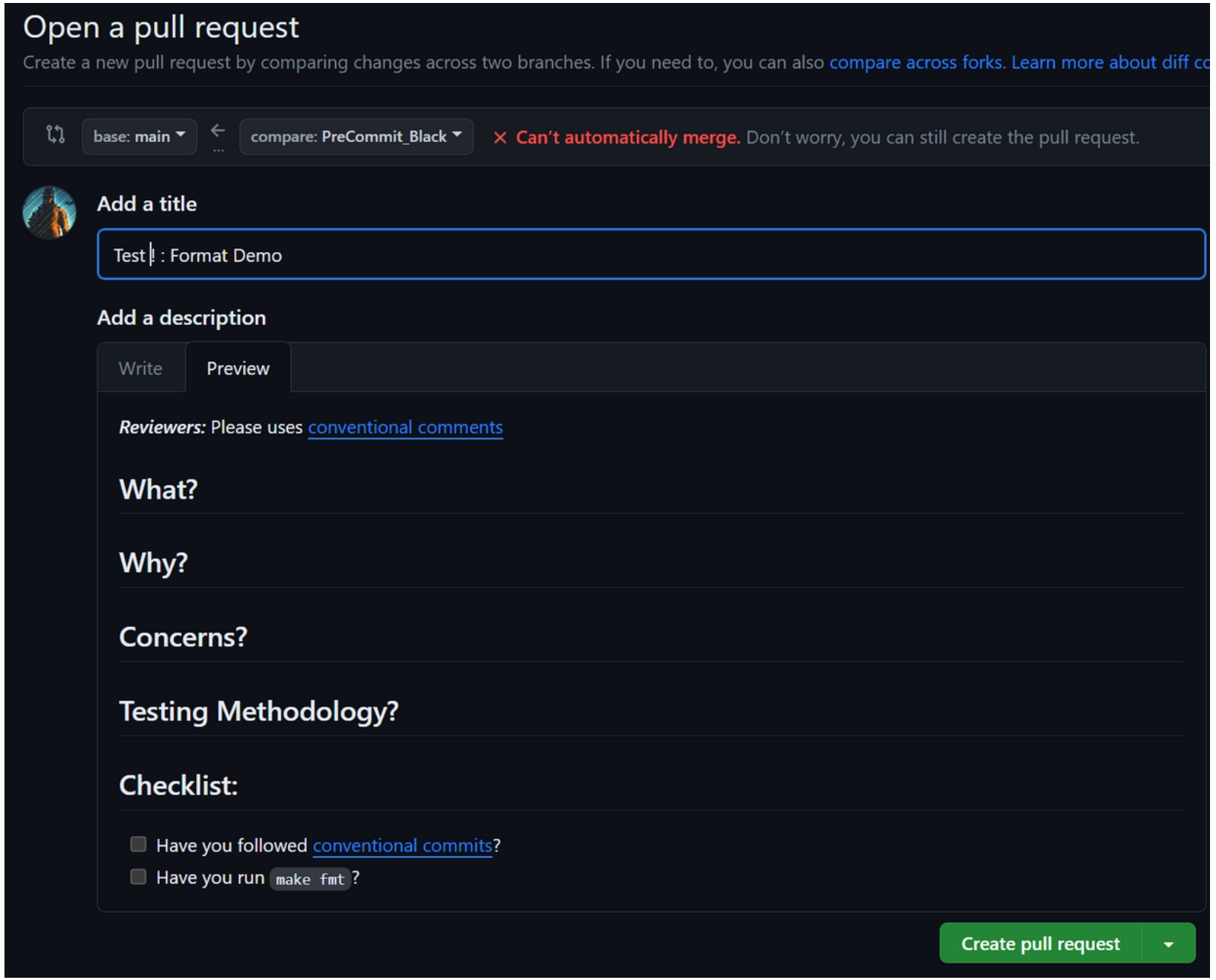
These strategies help teams manage complex projects efficiently, ensure code quality, and streamline the development process. The choice of branching strategy depends on the team's workflow, the project's complexity, and the deployment frequency.

Pull Request (PR) & Code Review

A **proposed code change** submitted by a contributor **for review** and integration into the main codebase



Simplified Pull Request process


Open a pull request
Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. Learn more about diff co
base: main
compare: PreCommit_Black
Can't automatically merge. Don't worry, you can still create the pull request.
Add a title
Test! : Format Demo
Add a description
Write
Preview
Reviewers: Please uses conventional comments
What?
Why?
Concerns?
Testing Methodology?
Checklist:
Have you followed conventional commits?
Have you run make fmt?
Create pull request

Thank you