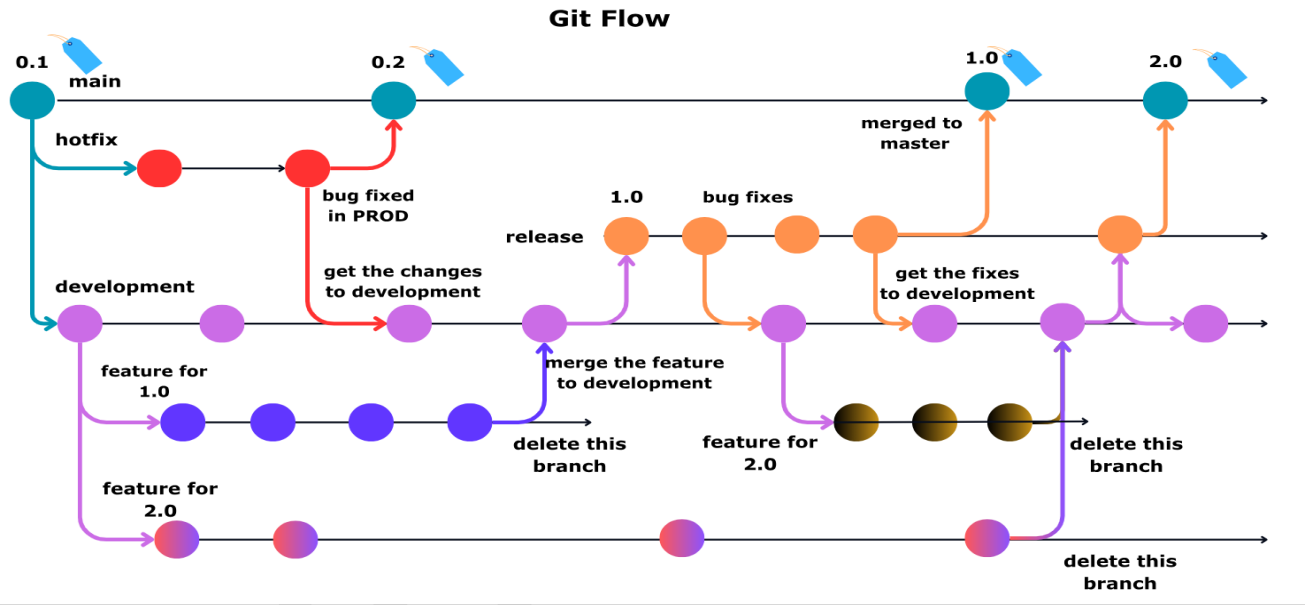
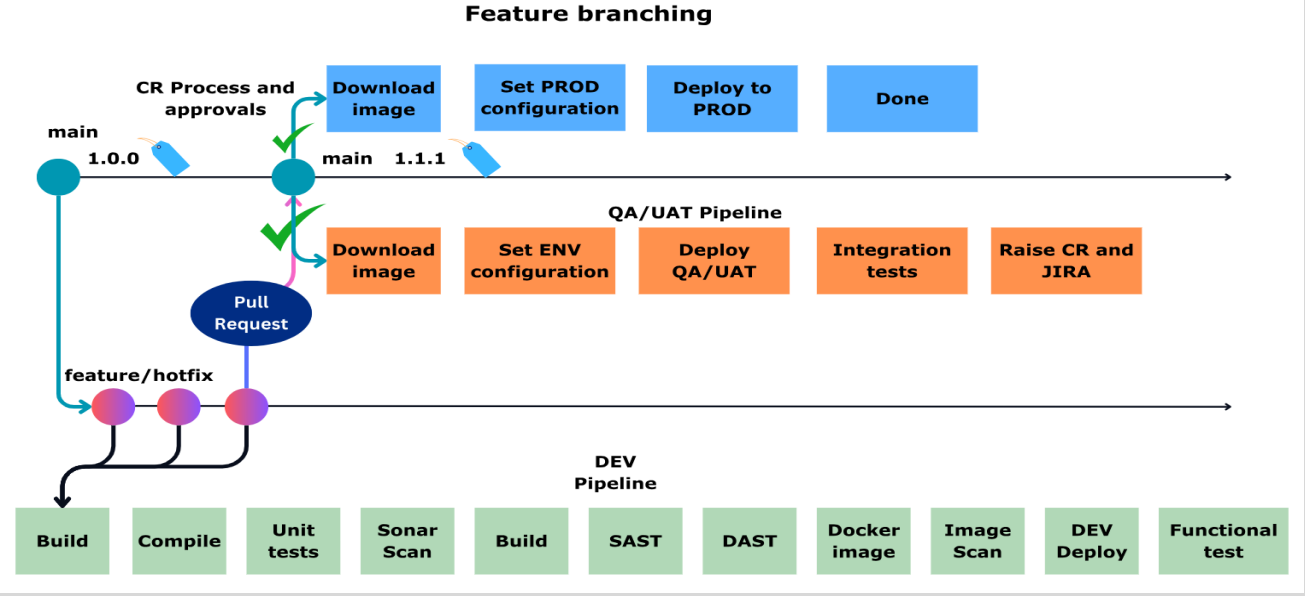
# Master Git Workflow: Branching Strategies and the Ultimate Guide to Git Reset vs Git Revert:

A **branching strategy** in Git is a workflow or set of practices that define how and when branches are created, merged, or deleted during the software development lifecycle. It helps manage code changes effectively, especially in teams working on large or complex projects.

## Popular Branching Strategies:



## 1. Git Flow

A robust workflow suitable for large projects with scheduled releases.

- **Main Branches:**
  - **main (or master)**: The production-ready branch.
  - **develop**: The integration branch for all features.
- **Supporting Branches:**
  - **Feature branches** (feature/*): Used for developing individual features.
  - **Release branches** (release/*): Prepare for a new release by finalizing and testing features.
  - **Hotfix branches** (hotfix/*): For urgent fixes on the production codebase.

**Flow:**

1. Start new features from **develop**.
2. Merge feature branches into **develop** after completion.
3. Create a release branch from **develop** when ready for testing.
4. Merge the **release** branch into **main** and tag it for production.
5. Use **hotfix** branches for critical production issues and merge into both **main** and **develop**.

**Pros:**

- Clear separation of concerns.
- Handles complex projects well.

**2. GitHub Flow**

A lightweight workflow commonly used for continuous deployment and smaller teams.

- **Main Branches:**
  - **main**: The only long-lived branch. Always deployable.
- **Flow:**

1. Create a branch for each feature or fix (e.g., feature/some-feature).
2. Commit changes to the feature branch.
3. Open a pull request (PR) for review and testing.
4. Merge the branch into main once approved and verified.
5. Deploy directly from main.

**Pros:**

- Simplicity and speed.
- Ideal for small teams or projects with CI/CD pipelines.

**Cons:**

- Less suited for projects with complex release cycles.

## Key Practices in Branching

1. **Branch Naming Conventions**:
    - Use prefixes like feature/, bugfix/, hotfix/, release/.
    - Examples:
        - feature/user-authentication
        - bugfix/fix-login-issue
2. **Code Reviews**:
    - Use pull requests for collaboration and quality checks.
    - Ensure all branches are reviewed before merging.
3. **Merge Strategies**:
    - **Squash merge**: Combines all commits into one for a clean history.
    - **Rebase and merge**: Keeps history linear.
    - **Merge commit**: Preserves the full history.
4. **CI/CD Integration**:
    - Automate testing and deployment pipelines for branches.
    - Use tools like Jenkins, GitHub Actions, GitLab CI/CD, or CircleCI.
5. **Branch Policies**:
    - Protect critical branches (main, develop).
    - Require status checks (e.g., tests pass) before merging.

| Choosing the Right Strategy | |
|---|---|
| **Criteria** | **Recommended Strategy** |
| **Small teams/projects** | GitHub Flow |
| **Large, complex projects** | Git Flow |
| **Continuous Deployment** | GitHub Flow |
| **DevOps-focused teams** | GitLab Flow |

## Git Reset Vs Git Revert:

Both git reset and git revert are used to undo changes in a Git repository, but they work differently and are used for distinct purposes. Here's a detailed comparison:

### 1. git reset

- **Purpose**: Moves the current branch's pointer backward in history, effectively "forgetting" certain commits. It is often used to undo commits in your local history before pushing to a shared repository.

### Modes of git reset:

1. **--soft**:
    - Keeps changes in the staging area (index).
    - Only moves the branch pointer.
    - Use case: "Oops, I need to amend this commit."
    - Example:

```
srinivasa.kp@BG4NB1882 MINGW64 /d/DevOps/daws-81s/repos (master)
$

srinivasa.kp@BG4NB1882 MINGW64 /d/DevOps/daws-81s/repos (master)
$ git reset --soft HEAD~1
```

2. **--mixed** (default):

- Unstages changes but keeps them in your working directory.
- Moves the branch pointer and un-stages changes.
- Use case: "I need to re-stage files before recommitting."
- Example:

```
srinivasa.kp@BG4NB1882 MINGW64 /d/DevOps/daws-81s/repos (master)
$

srinivasa.kp@BG4NB1882 MINGW64 /d/DevOps/daws-81s/repos (master)
$ git reset --mixed HEAD~1
```

3. **--hard**:

- Deletes all changes in the working directory and staging area. **Irreversible**!
- Completely resets the branch pointer and the working state.
- Use case: "I want to completely discard all changes."
- Example:

```
srinivasa.kp@BG4NB1882 MINGW64 /d/DevOps/daws-81s/repos (master)
$

srinivasa.kp@BG4NB1882 MINGW64 /d/DevOps/daws-81s/repos (master)
$ git reset --hard HEAD~1
```

**Key Points for git reset:**

- **Rewrites history**: Changes the repository's history and can be dangerous in shared repositories.
- **Affects local changes**: It alters the local repository state, including the working directory and index.

**2. git revert**

- **Purpose**: Creates a new commit that "reverses" the effects of a previous commit without altering the history.

**How it works:**

- Git applies an "inverse" patch to undo the changes from a specific commit.
- This action is **safe for shared repositories** because history is preserved.

**Example:**

To revert a commit:

```
srinivasa.kp@BG4NB1882 MINGW64 /d/DevOps/daws-81s/repos (master)
$

srinivasa.kp@BG4NB1882 MINGW64 /d/DevOps/daws-81s/repos (master)
$ git revert <commit-hash>
```

Git will open your editor to write a commit message for the revert. After saving, a new commit will be added.

## Key Points for git revert:

- **Does not rewrite history**: Keeps the repository's history intact and transparent.
- **Safe for collaboration**: Preferred for undoing changes in shared repositories.

| Comparison Table | | |
|---|---|---|
| **Feature** | **git reset** | **git revert** |
| **Action** | Moves the branch pointer backward. | Creates a new commit to undo changes. |
| **Effect on History** | Rewrites history (potentially destructive). | Preserves history (non-destructive). |
| **Collaboration** | Risky in shared repositories. | Safe for shared repositories. |
| **Use Case** | Undo commits locally. | Undo specific changes permanently. |
| **Modes** | --soft, --mixed, --hard. | Single mode (creates a new commit). |
| **Reversible** | Partially (depends on the mode). | Yes, can revert the revert commit. |

## Example Scenarios

1. **Local repository fix (before pushing):**
    - Use git reset to remove unwanted commits.
    - Example: git reset --soft HEAD~2 to undo the last 2 commits and keep changes staged.

2. **Shared repository fix:**
    - Use git revert to undo a commit without affecting others' work.
    - Example: git revert <commit-hash> to undo a specific commit.

**Final Advice:**

- **Use git reset for local-only changes** when you're sure no one else depends on the history.
- **Use git revert for undoing changes in shared repositories**, as it's safer and preserves history.

**Conclusion: Git Branching Strategies and Commands**

Branching strategies and Git commands like git reset and git revert are essential tools for managing code in software development. Here's a concise summary:

**Branching Strategy**

Branching strategies organize the workflow in Git, making collaboration smoother and code management efficient. Each strategy serves specific needs:

- **Git Flow**: Best for **structured, complex projects** with defined release cycles, ensuring clear separation of features, releases, and fixes.
- **GitHub Flow**: Ideal for **continuous delivery** and **small teams**, with simplicity at its core, enabling frequent and lightweight updates.

- **GitLab Flow**: Bridges the gap for **DevOps-focused teams**, integrating environments like staging or pre-production into the workflow.

**Key Practices:**

1. **Branch Naming**: Use consistent naming conventions for clarity (e.g., feature/login-page or hotfix/payment-error).
2. **Code Reviews**: Collaborate via pull requests to ensure quality and adherence to standards.
3. **Merge Strategies**: Use the appropriate merge type (squash, rebase, or commit merge) based on team needs.
4. **CI/CD Integration**: Automate testing and deployment to maintain high development velocity and reliability.
5. **Branch Protection**: Safeguard critical branches (main, develop) with status checks.

**Git Reset vs. Git Revert**

These commands handle undoing changes but serve different purposes:

- **git reset**: Alters commit history locally. Use cautiously in **private or local repositories**.
- **git revert**: Creates a new commit to undo changes. Preferred for **shared repositories** as it preserves history.

**When to Use:**

- Use **git reset** when fixing **local commits** that haven't been pushed yet.
- Use **git revert** to safely undo changes in **shared or collaborative environments**.

**Final Advice:**

- Choose a branching strategy that aligns with your project needs, balancing complexity and flexibility.
- Use git reset for precise, local adjustments, but avoid rewriting shared history.
- Use git revert for safe, collaborative undoing of changes without disrupting others' work.