Git Branching Strategy Explained

1. What is a Git Branch?

A branch in Git represents an independent line of development. It acts as a pointer to a specific commit, allowing developers to work on different features, bug fixes, or releases without affecting the main codebase.

2. Types of Branches (from your notes)

Your notes indicate two primary types of branches:

A. Customer Branch

- Created specifically for a customer.
- Contains common software (base version) with customer-specific changes.
- Example branches: SBF_customer, SBH_customer, SBA_customer.
- Each branch evolves with unique updates and features.

B. Release Branch

- Created for a specific version of the application delivered to a customer.
- These branches are stable and undergo thorough validation.
- Named systematically, e.g., SBF_R_1.0.

3. Branch Workflow Example

From the diagrams, the general branching strategy and workflow includes:

- Main application has modules: Login, Sales, Account.
- Each module might have its own codebase or feature branches.
- Begin with a common base (e.g., state_branch_gray).
- Create customer-specific branches and release branches.
- Follow environment lifecycle: Dev -> Test -> Staging 1 -> Staging 2 -> Production.
- Testing phases: UAT and SIT before sprint execution.
- Releases composed of components like a, b, c.
- Example: Sprint 01 = a+b, Sprint 02 = a+c, Final = a+b+c.

4. Benefits of Using Branches

- Parallel development by multiple teams.
- Isolation of changes.
- Controlled release process.
- Customer-specific customization without disrupting the main codebase.

5. Git Commands for Branching

Useful Git commands to work with branches:

- Create a branch: git checkout -b feature-branch
- Switch branches: git checkout branch-name

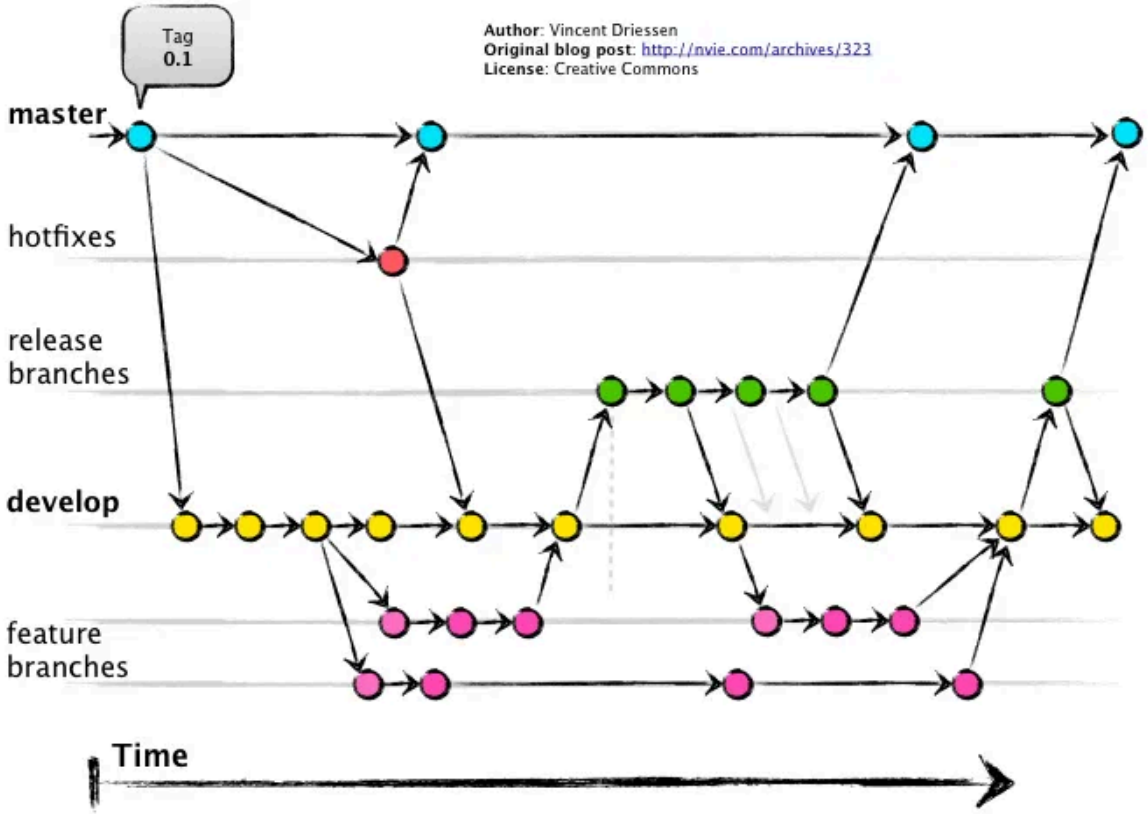
- List branches: git branch
- Merge branches: git checkout main && git merge feature-branch
- Delete a branch: git branch -d old-branch
- Push branch to remote: git push origin branch-name

Popular Git Branching Strategies

https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

1. Git Flow

Best for: Complex projects with scheduled releases.



https://medium.com/novai-devops-101/understanding-gitflow-a-simple-guide-to-git-branching-strategy-4f079c12edb9

Branches:
- main: Production-ready code.
- develop: Integration branch for features.
- feature/*: New features.
- release/*: Pre-production versions.
- hotfix/*: Quick fixes on production.

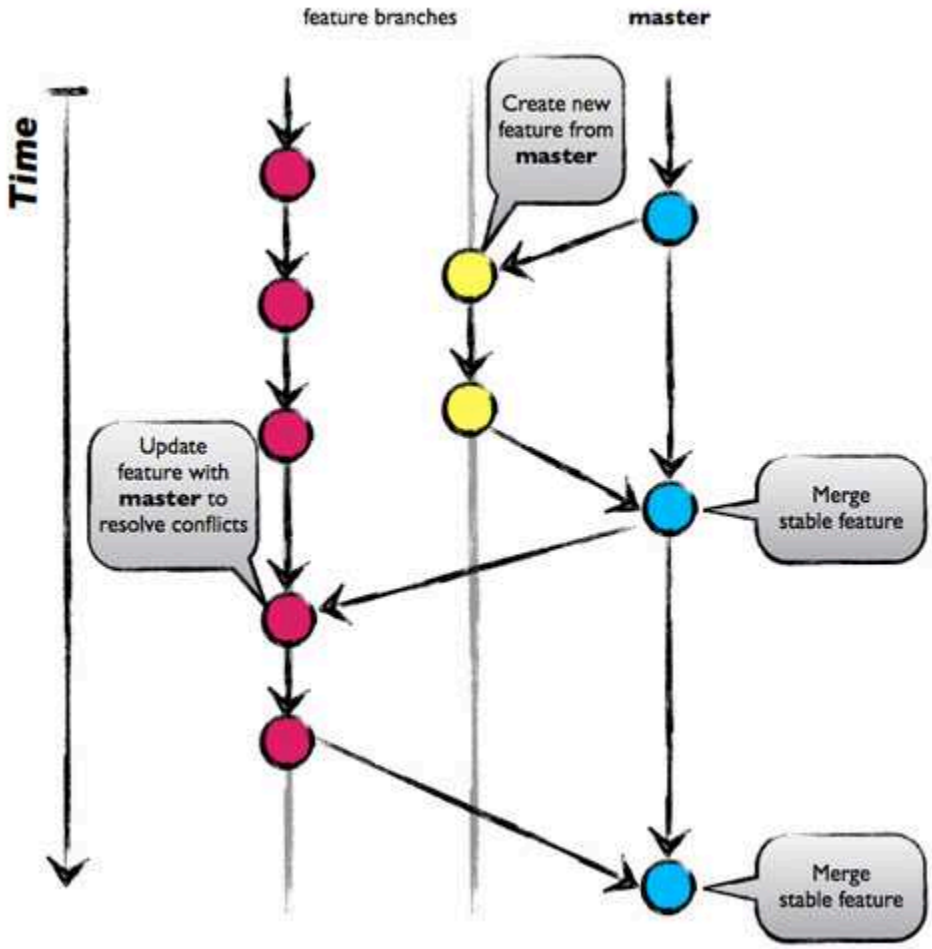
Workflow:
1. Start feature from develop.
2. Merge feature into develop.
3. Create release from develop.
4. Merge release to main and develop after testing.
5. Hotfix goes from main and merges back to main and develop.

Pros:
- Clear structure.
- Good for multi-environment setups.

Cons:
- Overhead of managing multiple branches.

2. GitHub Flow

Best for: Continuous delivery and small teams.



https://www.abtasty.com/blog/git-branching-strategies/

Branches:
- main: Always deployable.
- feature/*: Short-lived feature branches.

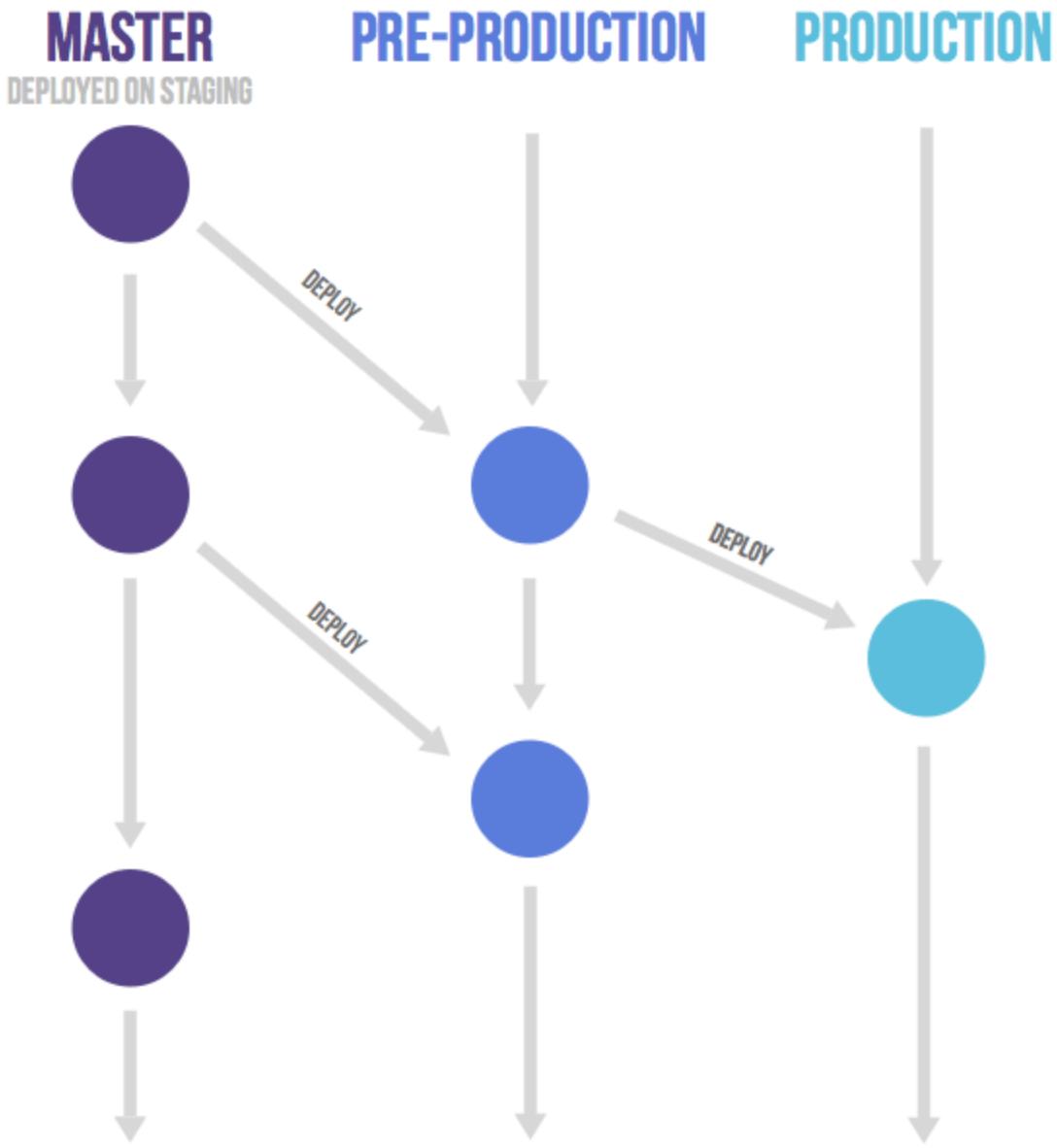
Workflow:
1. Create a branch from main.
2. Open a Pull Request (PR).
3. Review and test via CI/CD.
4. Merge into main.

Pros:
- Simple and fast.
- Great for continuous deployment.

Cons:
- No explicit release or staging branches.

3. GitLab Flow

Best for: Combining feature-driven and environment-based development.

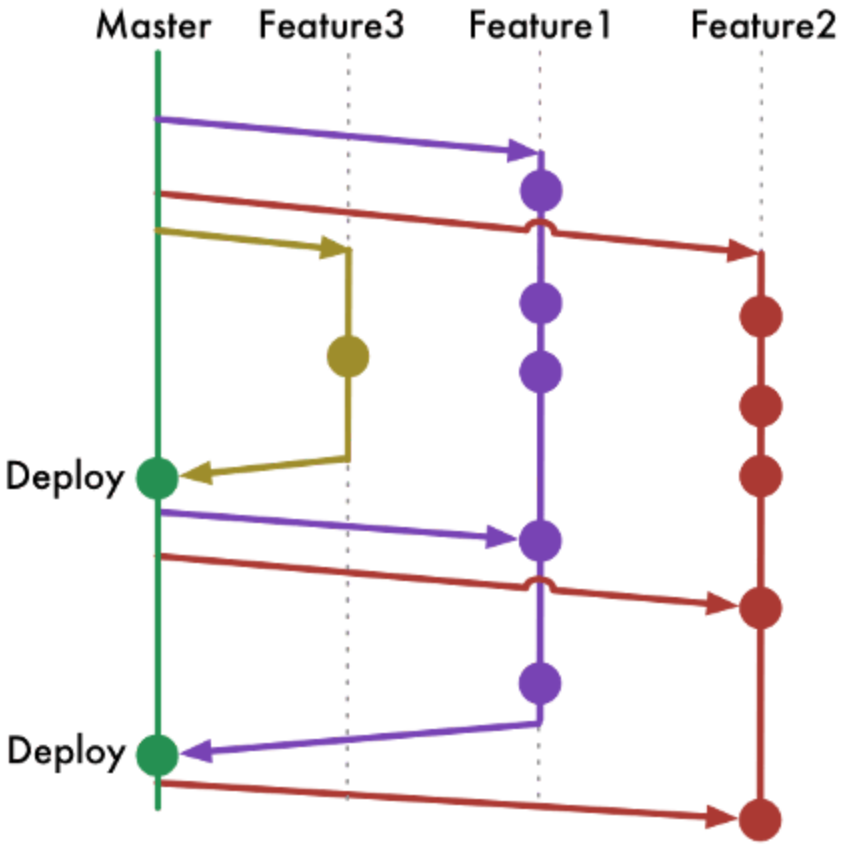


Combines Git Flow with environment branches like staging and production.

Pros:
- Flexible for Dev and Ops teams.
- Great for end-to-end CI/CD pipelines.

4. Trunk-Based Development

Best for: High-frequency deployment, modern DevOps teams.



Branches:
- main or trunk: Single shared branch.
- Short-lived feature branches or direct commits.

Workflow:
1. Developers commit directly or via short-lived branches.
2. Use feature flags to hide incomplete features.
3. Deploy continuously.

Pros:
- Encourages fast integration.
- Fewer merge conflicts.

Cons:
- Requires robust CI/CD and testing automation.

5. Release Branching

Best for: Teams maintaining multiple versions/releases.

Branches:
- main or master

- release/1.x, release/2.x: For versioned support.

Bugfixes and features are backported as needed.

Pros:
- Good for long-term support and versioning.

Cons:
- Requires careful management of bugfixes across versions.