# FROM CHAOS TO FLOW

## GIT & BRANCHING STRATEGIES

By Lajin M J, CoE

For Techversant Infotech


पथ निर्माण विभाग, बिहार
ROAD CONSTRUCTION DEPARTMENT, BIHAR
तारामंडल
PLANETARIUM
नेहरू मार्ग
NEHRU MARG
पटना संग्रहालय
PATNA MUSEUM
गांधी मैदान
GANDHI MAIDAN
डाकबंगला चौराहा
DAKBUNGLOW
राजेन्द्र पथ
RAJENDRA PATH

# WHY
## WE NEED VERSION CONTROL

### Track Changes

Complete history with **timestamps** and **author** info

### Team Collaboration

Multiple developers on **same project** without conflicts

### Easy Revert

Quickly **undo mistakes** and return to any version

### Branching

Create **parallel versions** for features and experiments

### Backup

Your code is **safely stored** in multiple locations

### Performance

Operations are **fast and efficient** even with large projects

# OVERVIEW

## HOW GIT TRACKS AND MANAGES YOUR CHANGES


**Add Files**

```
git add file.txt
```

**Commit Changes**

```
git commit -m "Message"
```

**Push to Remote**

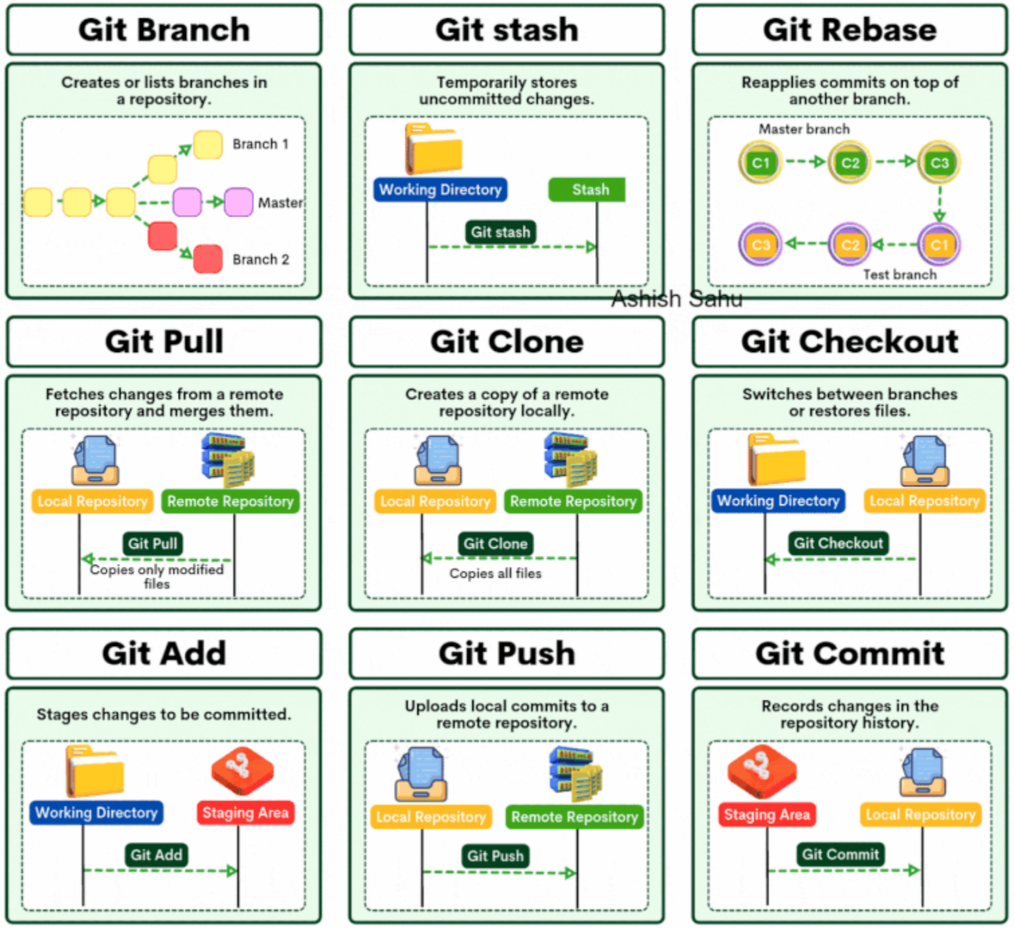
```
git push origin main
```

**Pull Changes**

```
git pull origin main
```

# GIT COMMANDS

## UNDERSTANDING THE BASIC GIT COMMANDS




Real-World Example Flow

```
# Clone project
git clone https://github.com/techversant-infos/branching-strategy.git
cd branching-strategy

# Create a new feature branch
git checkout -b feature/add-ICF-102-login

# Work and stage changes
git add .
git commit -m "feat(auth): add OTP-based login"

# Sync with main before pushing
git fetch origin
git rebase origin/main

# Push branch for PR
git push origin feature/add-ICF-102-login

# After approval
git checkout main
git merge feature/add-ICF-102-login
```

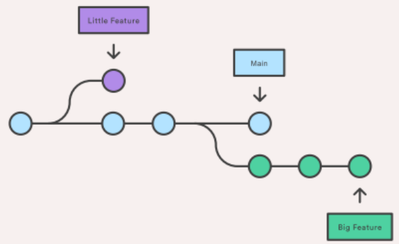
# BRANCHING

## CREATING AND MANAGING BRANCHES

### What is a Branch?

A **branch** is an independent line of development that allows you to work on features without affecting the main codebase

**Branch Visualization**




### Why Use Branches?

- Isolate new features
- Fix bugs safely
- Experiment without risk

### Create & Switch Branch

Create a new branch and switch to it

```
git checkout -b feature-branch
# Creates and switches to new branch
```

### Switch Branches

Move between different branches

```
git checkout main
git switch feature-branch
```

### Merge Branches

Combine changes from one branch into another

```
git checkout main
git merge feature-branch
# Merges feature-branch into main
```

Use **git branch -d** to delete merged branches

# COLLABORATION

## PULLING, MERGING, AND RESOLVING CONFLICTS

### Syncing with Remote

#### Pull Changes

Fetch and merge changes from remote repository

```
git pull origin main
# Fetches and merges remote changes
```

Always **pull before pushing** to avoid conflicts

#### Fetch Changes

Download changes without merging

```
git fetch origin
git merge origin/main
# Two-step alternative to pull
```

### Merging & Conflicts

#### Merge Branches

Combine changes from different branches

```
git checkout main
git merge feature-branch
# Merges feature-branch into main
```

#### Resolving Conflicts

When Git can't automatically merge changes

1. Open conflicted files in editor
2. Resolve conflicts manually
3. Stage resolved files
4. Complete the merge with commit

Use **git status** to see conflicted files

# MERGE CONFLICTS

## WHAT A MERGE CONFLICT LOOKS LIKE


That's exactly what happens in Git — it's called a **merge conflict**.

**Git shows something like this inside the file:**

```
<<<<<<< HEAD
driver.findElement(By.id("username")).sendKeys("admin");
driver.findElement(By.id("password")).sendKeys("admin123");
=======
LoginPage loginPage = new LoginPage(driver);
loginPage.login("admin", "admin123");
>>>>>>>feature/login-page-refactor
```

**Let's decode this:**

- `<<<<<<< HEAD` → Code from **your current branch** (e.g., master)
- `=======` → Divider between both versions
- `>>>>>>>` feature/login-page-refactor → Code from the **other branch**

**It's Git saying: "**I found two versions of this part — please tell me which one to keep."

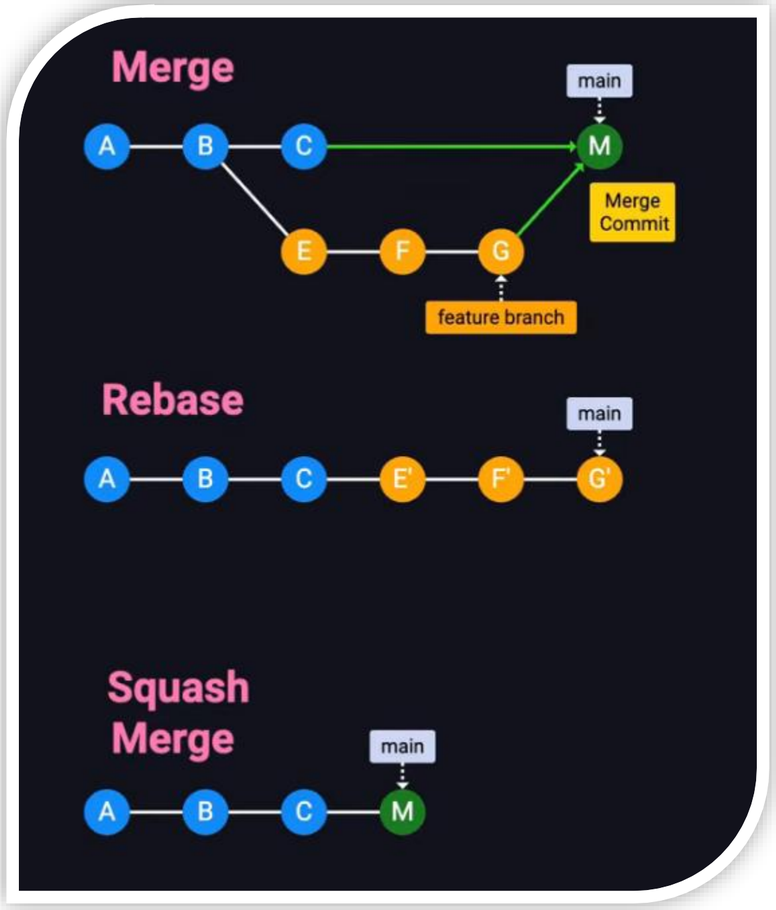
# RESOLVING CONFLICTS

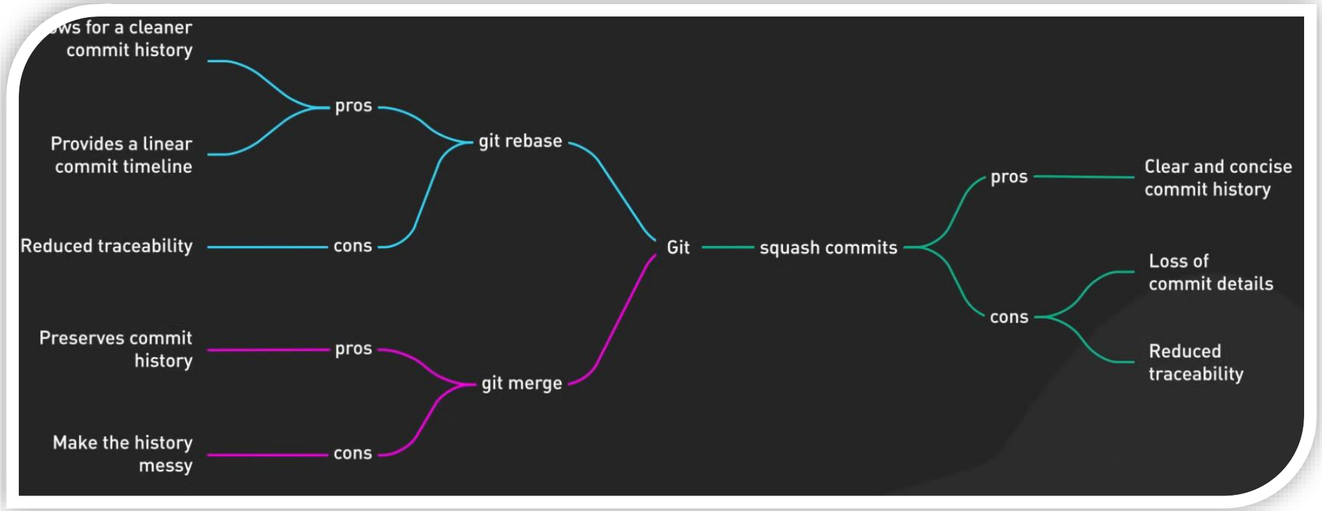
## HOW TO RESOLVE CONFLICTS

| Command | Meaning |
| --- | --- |
| `git checkout master` | Switch to master branch |
| `git pull origin master` | Get the latest version of master |
| `git merge feature/login-page-refactor` | Try to merge the feature branch |
| **Conflict happens** | You fix it manually in the file |
| `git add src/test/java/tests/LoginTest.java` | Tell Git that you've fixed the conflict |
| `git commit -m "Resolved merge conflict using POM structure"` | Save your fix |
| `git push origin master` | Send your fixed code to GitHub |

# MERGE, REBASE & SQUASH

RESOLVING CONFLICTS

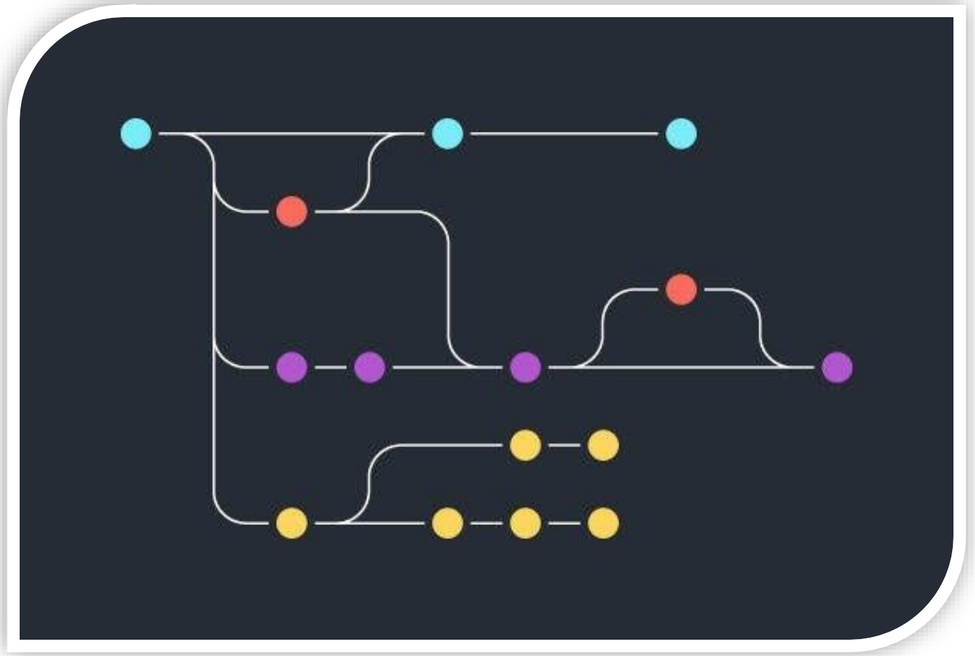


# WHAT
## IS BRACHING STRATEGY

A **branching strategy** is a structured way to manage how code changes are developed, tested, and merged in a Git repository.

It defines **where developers commit code**, **how they collaborate**, and **how features move from idea → production**.



# COMMON BRANCHING
# STRATEGIES

| Workflow | Ideal For | Key Users (2025) | Complexity |
|---|---|---|---|
| **GitHub Flow** | SaaS, CI/CD-heavy teams | GitHub, Vercel, Netlify | ⭐ |
| **GitLab Flow** | Teams aligning dev → staging → prod | GitLab, enterprise CI users | ⭐⭐ |
| **Git Flow (Classic)** | Release-based products | Atlassian, legacy apps | ⭐⭐⭐ |
| **Trunk-Based Development** | High-speed teams, microservices | Google, Meta, Uber | ⭐⭐⭐⭐ |
| **Feature Branching (Basic)** | Small to mid teams | Most startups | ⭐ |

1. GitFlow
2. GitHub Flow
3. GitLab Flow
4. Trunk Based Dev
5. Feature Branching

# WHAT IS **GITFLOW**

**Best For:**

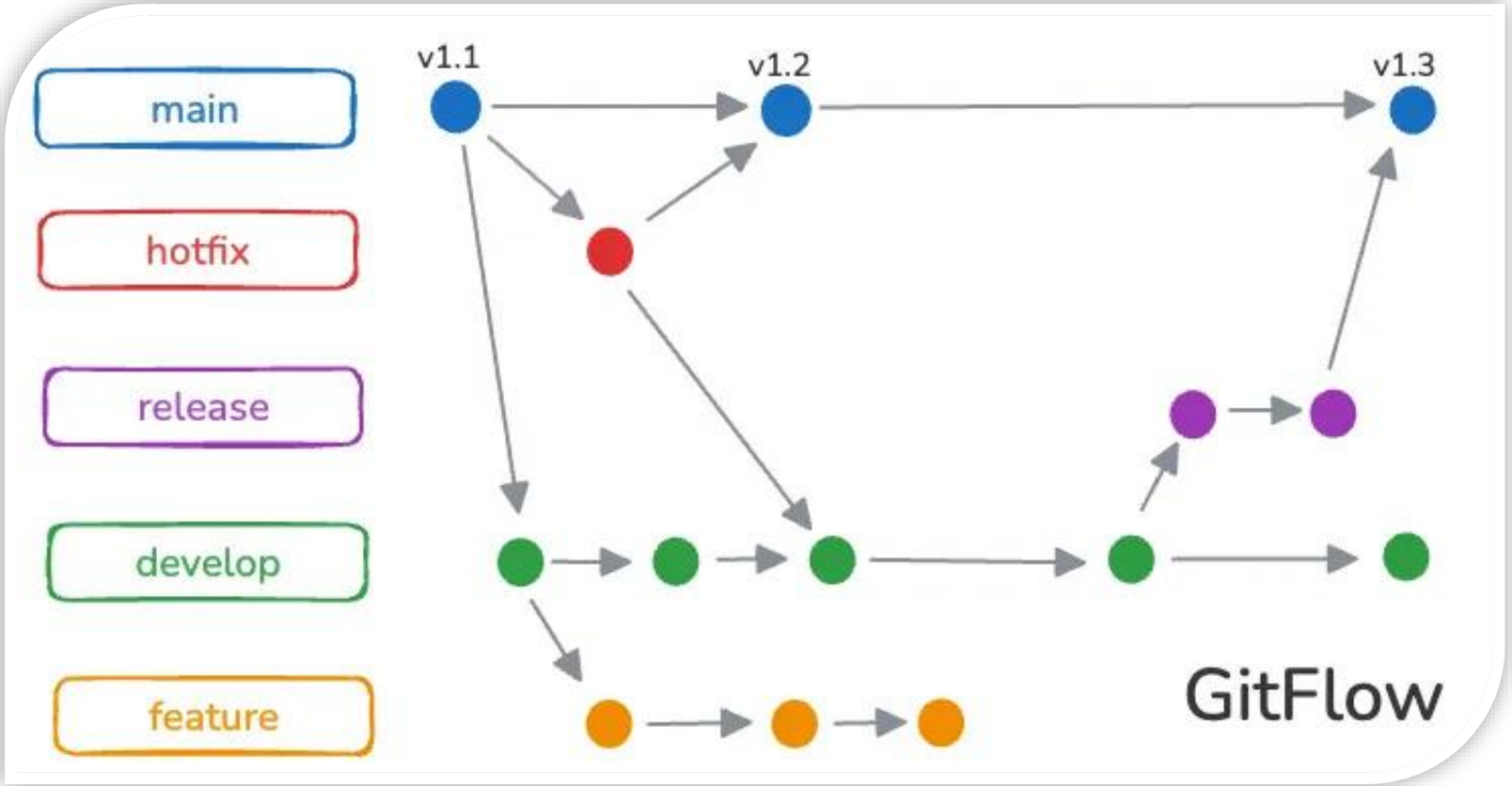
- Apps with planned releases (e.g., mobile, enterprise)
- Teams needing clear release versions

**Best Practices:**

- Tag releases
- Use consistent naming (release/x.x.x)
- Merge hotfixes into both main & develop

**Avoid when:**

- Fast-paced CI/CD teams
- Small teams (too heavy)



# WHAT IS
# GITHUB FLOW

**Best For:**

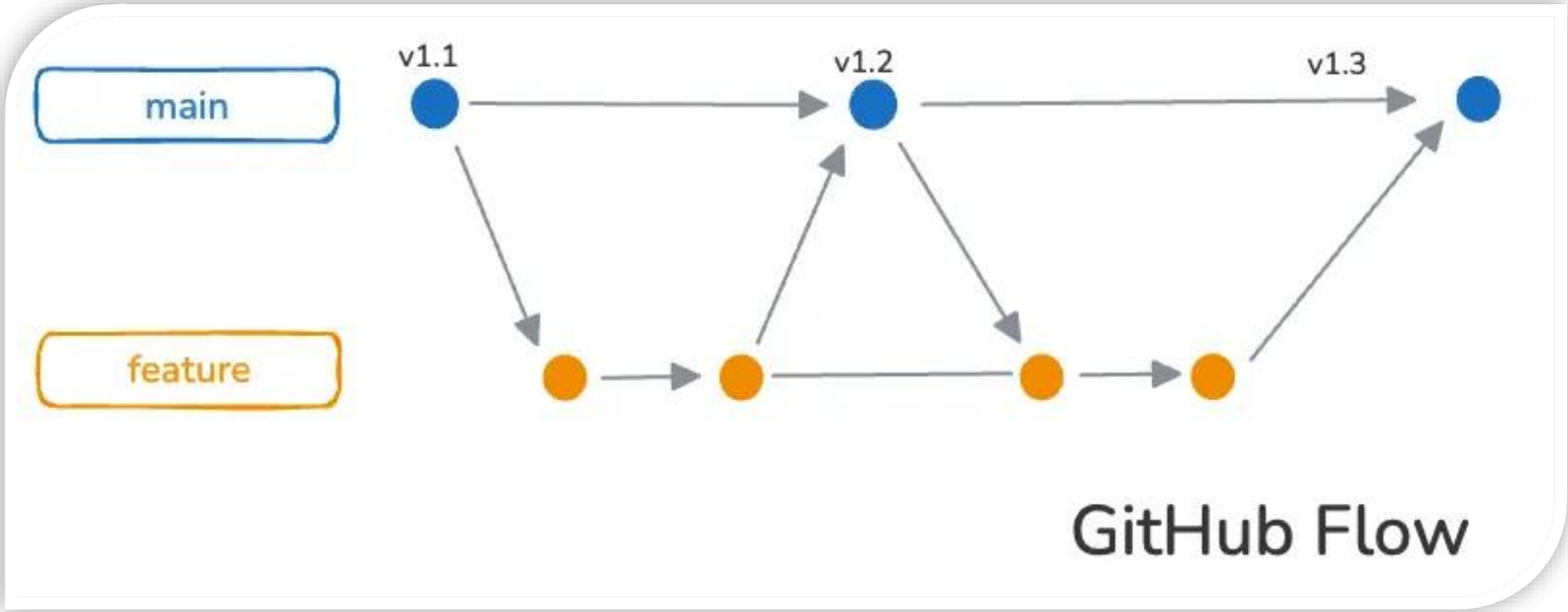
- Web apps with continuous deployment
- Teams using GitHub Actions, Vercel, or Netlify

**Best Practices:**

- Always deploy from main
- Keep branches short-lived
- Automate linting & testing

**Avoid when:**

- Complex release schedules
- Multiple environments



# WHAT IS
# GITLAB FLOW

**Best For:**

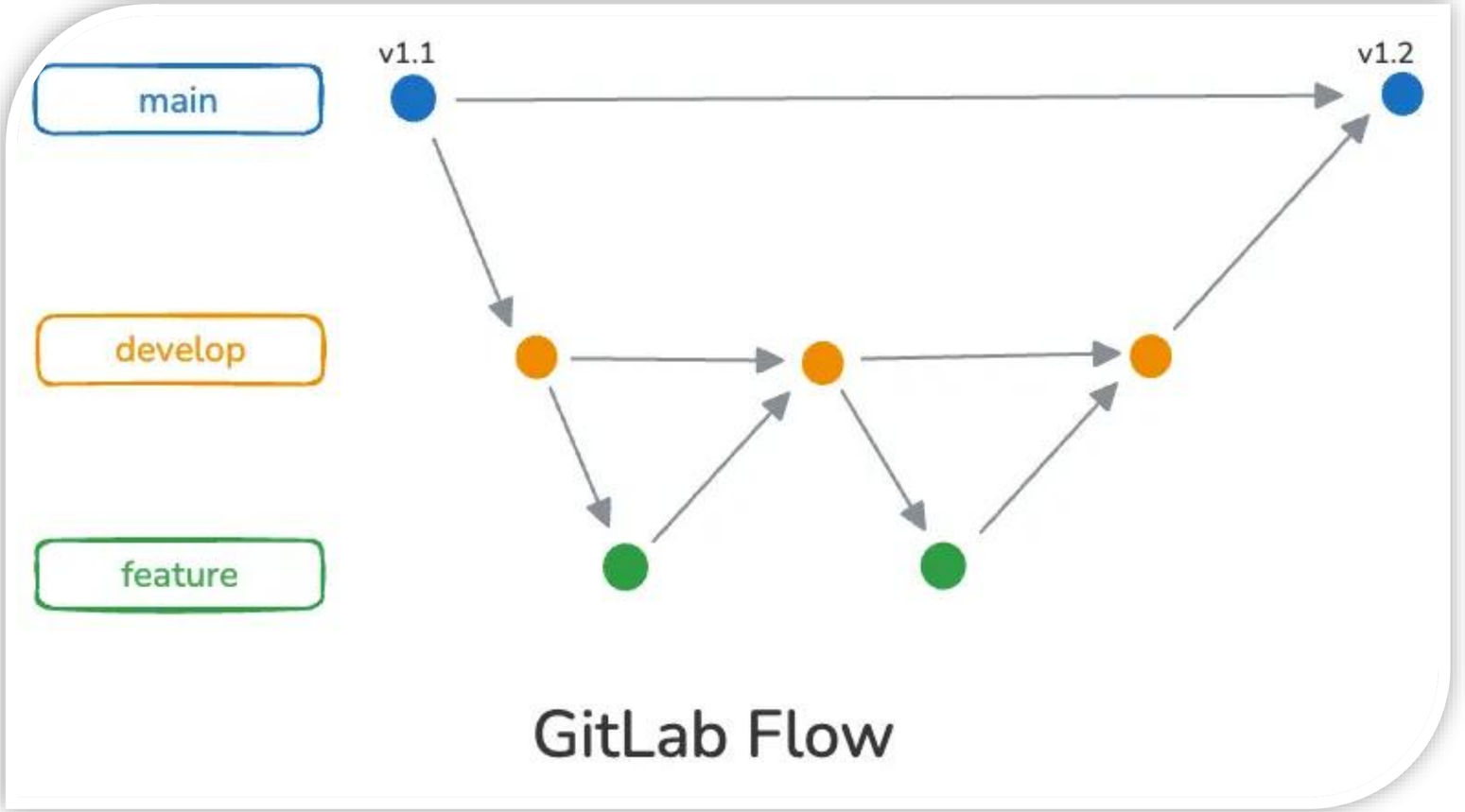
- Teams aligning code branches with environment stages
- GitLab CI/CD users

**Best Practices:**

- Enforce merge requests
- Protect production branch
- Automate staging → prod promotions

**Avoid when:**

- You need ultra-fast, continuous releases
- No real environment promotion

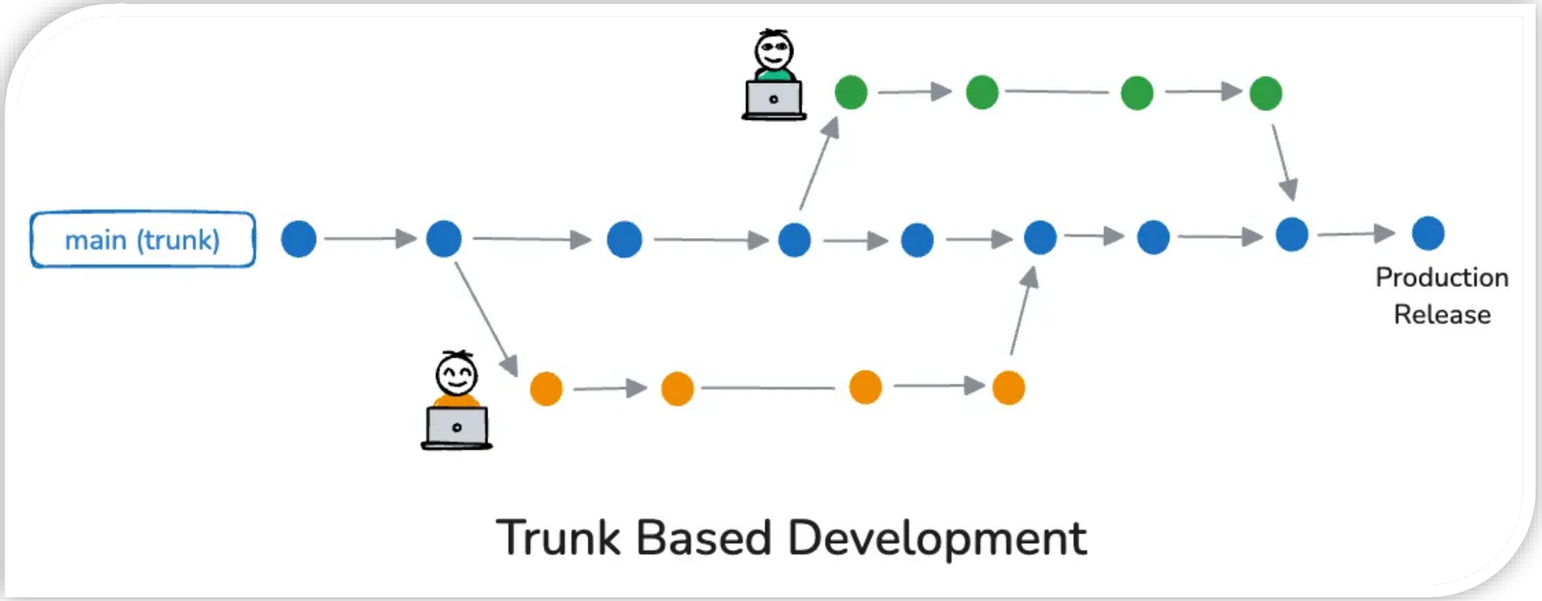


# WHAT IS
# TRUNK BASED DEV

**Best For:**
- Mature CI/CD pipelines
- Large, distributed teams

**Best Practices:**
- Feature toggles for unfinished code
- Frequent merges (daily)
- Strong automated testing

**Avoid when:**
- Teams without robust CI/CD
- Manual QA bottlenecks



WHAT IS

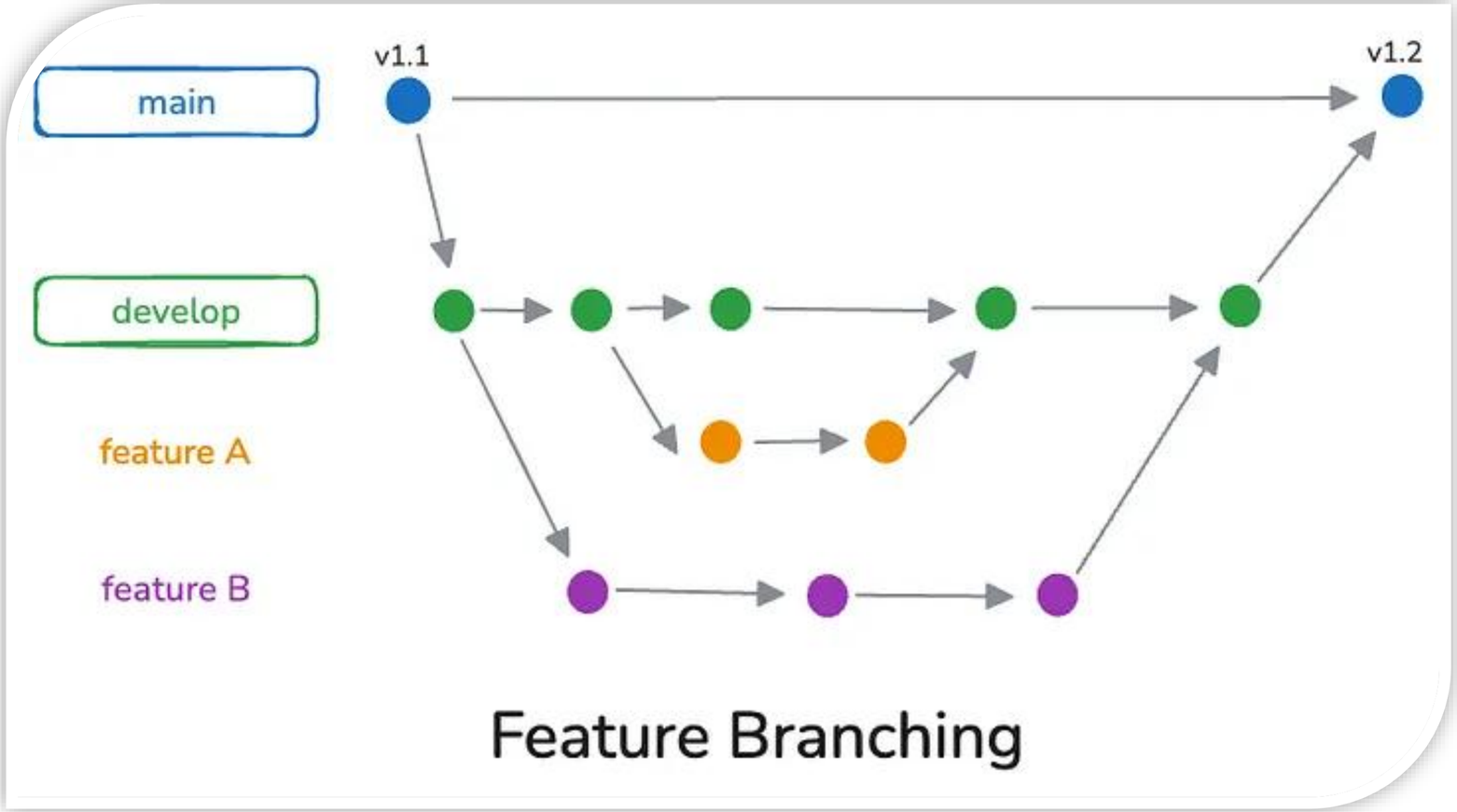
# FEATURE BRANCHING

**Best For:**

- Startups, small teams
- Ad-hoc release cycles

**Best Practices:**

- Use descriptive branch names (feature/user-auth)
- Keep branches under a week
- Use draft PRs for visibility



# CHOOSING THE RIGHT **STRATEGY**

1. How many **active contributors** commit daily?
2. Do you **ship** continuously or in fixed versions?
3. Do you have automated tests and **pipelines**?
4. Do you have dev/staging/prod **environments**?
5. Compliance: Are **audits** or version tags mandatory?

## DECISION MATRIX

| Team Size | Release Frequency | CI/CD Maturity | Recommended Workflow |
|---|---|---|---|
| Small | Low | Low | Feature Branching |
| Small | High | Medium | GitHub Flow |
| Medium | Medium | High | GitLab Flow |
| Large | High | Very High | Trunk-Based Dev |
| Enterprise | Low | Medium | Git Flow |
| Open Source | Open Source | Open Source | Open Source |

GIT

# BEST PRACTICES

- **main** is always deployable.
- Short-lived feature branches.
- PR review and CI validation are mandatory.
- Build once → Promote through environments.
- Every production deployment must be **tagged**.
- Follow **Conventional Commits** for clarity and automation.
- Keep the process transparent and reproducible.
- Use *.nocommit* or *.gitignore* to prevent committing sensitive or local-only files.
- Never commit environment variables or credentials; use .env.example templates.
- All projects must have *.editorconfig* and *.gitattributes* to maintain consistency.
- Handbook

# THANK YOU