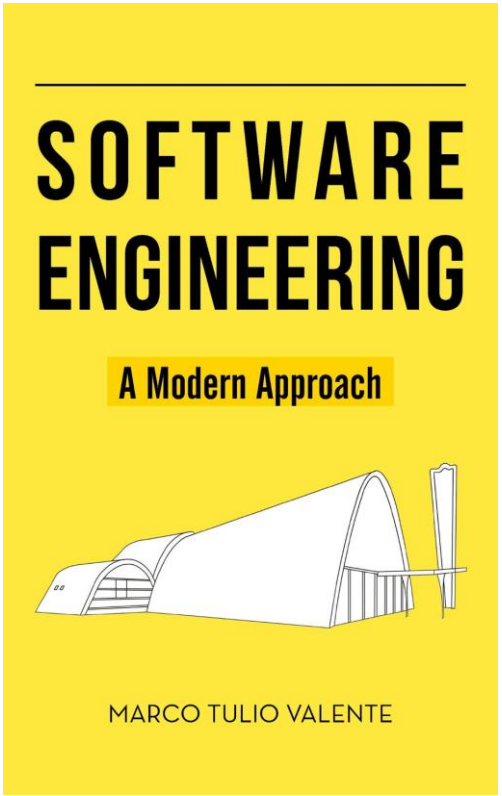


Chapter 10 - DevOps

Prof. Marco Tulio Valente

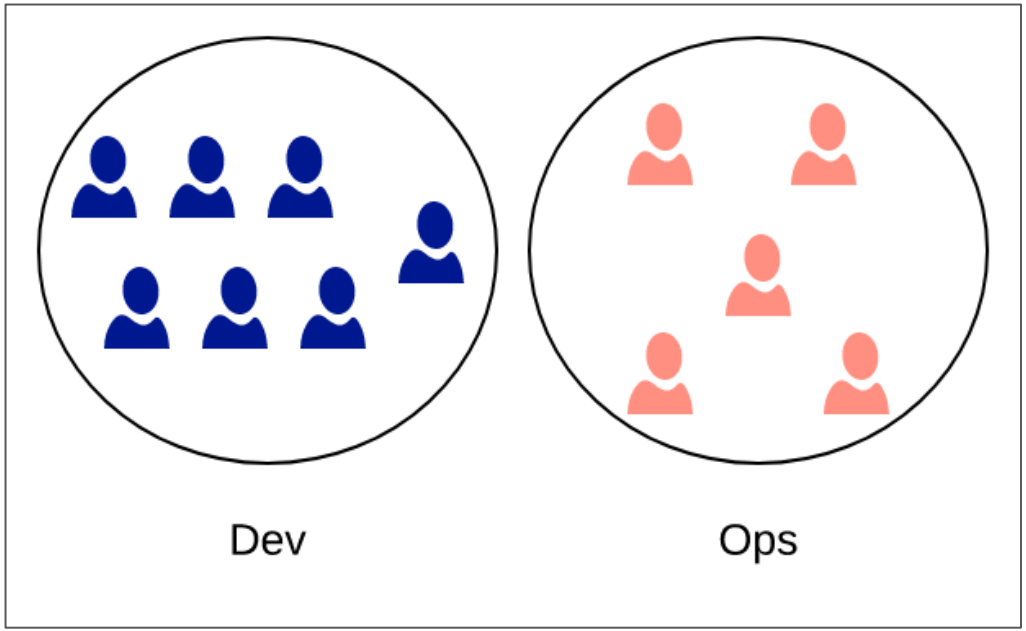
https://softengbook.org

Our situation in the course

- We defined and used a software process
- The requirements have been defined and implemented
- The design and architecture have been established
- Various tests have been developed and implemented
- We have completed many refactorings

Now we should complete the "last mile": To deploy the system, i.e., put it into production

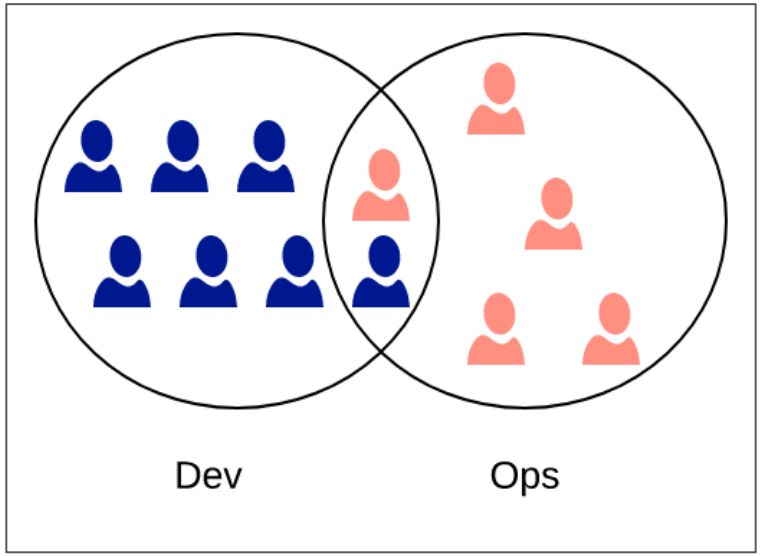
In the past, deployment was a challenging and high-risk process

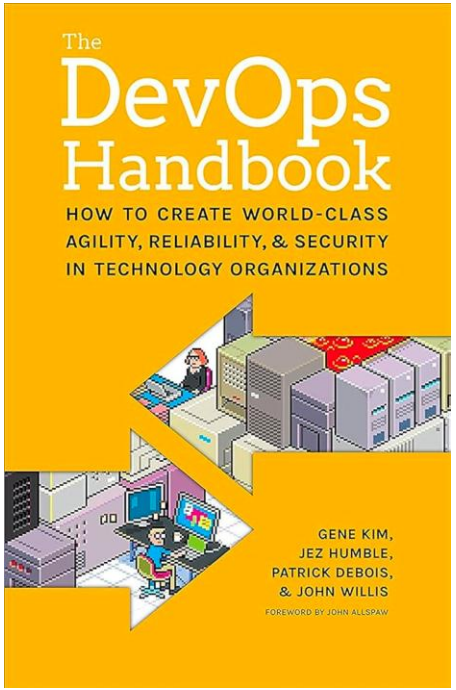


Two independent silos, with very little communication

Ops = system administrators, support, sysadmin, IT personnel, etc

Central idea of DevOps: Bridging the gap between Dev and Ops





"Imagine a world where product owners, development, QA, IT Operations, and Infosec work together, not just to aid each other, but to guarantee the overall success of the organization."

Primary objective: successful handover!

(deployment should start as soon as possible; be fully automated, etc)

Ops

Dev

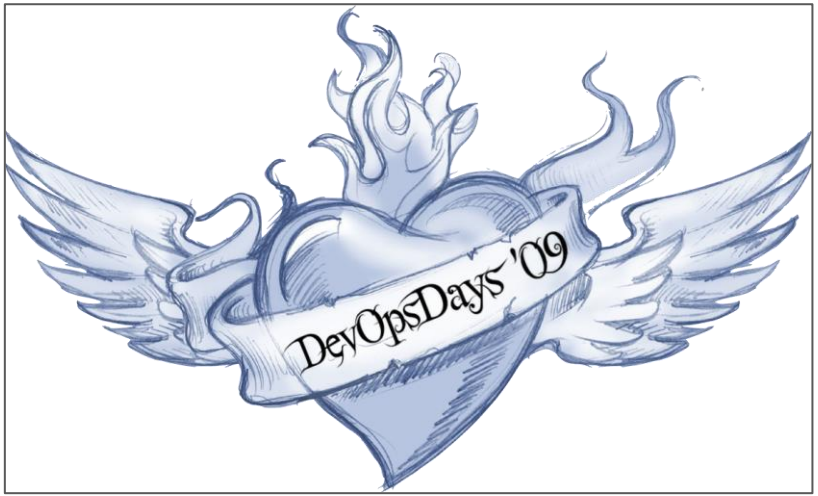
Objective: eliminate the "blame culture"

Dev: "The problem is not in my code, but in your server"

Ops: "The problem is not in my server, but in your code"

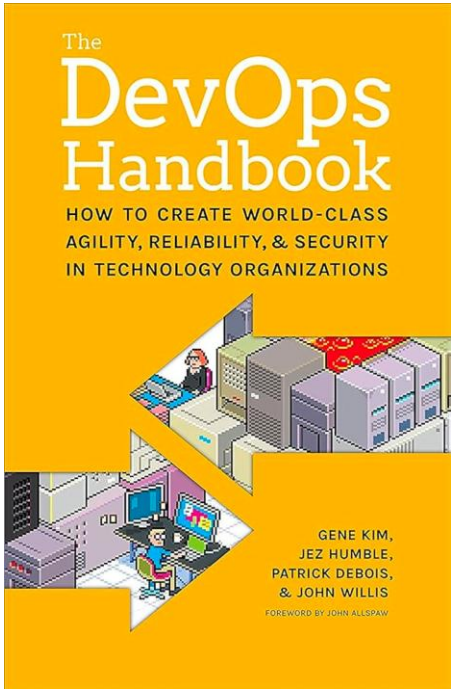
DevOps

- It's not a title or role; but a set of principles and practices
- Name emerged ~2009



DevOps Principles

- Foster collaboration between Devs and Ops teams
- Apply an agile mindset throughout the deployment phase
- Transform deployments into a routine operation
- Deploy software every day
- Automate the deployment process



"Instead of starting deployments at midnight on Friday and spending the weekend working to complete them, deployments occur on any business day when everyone is in the company and without customers noticing — except when they encounter new features and bug fixes."

DevOps Practices

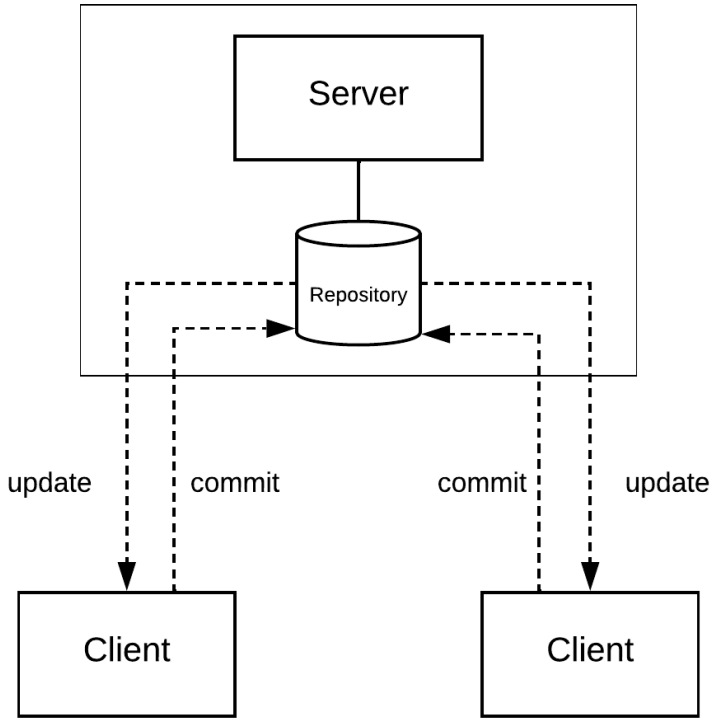
- Version Control
- Continuous Integration
- Branching Strategies
- Continuous Deployment
- Feature Flags

Version Control

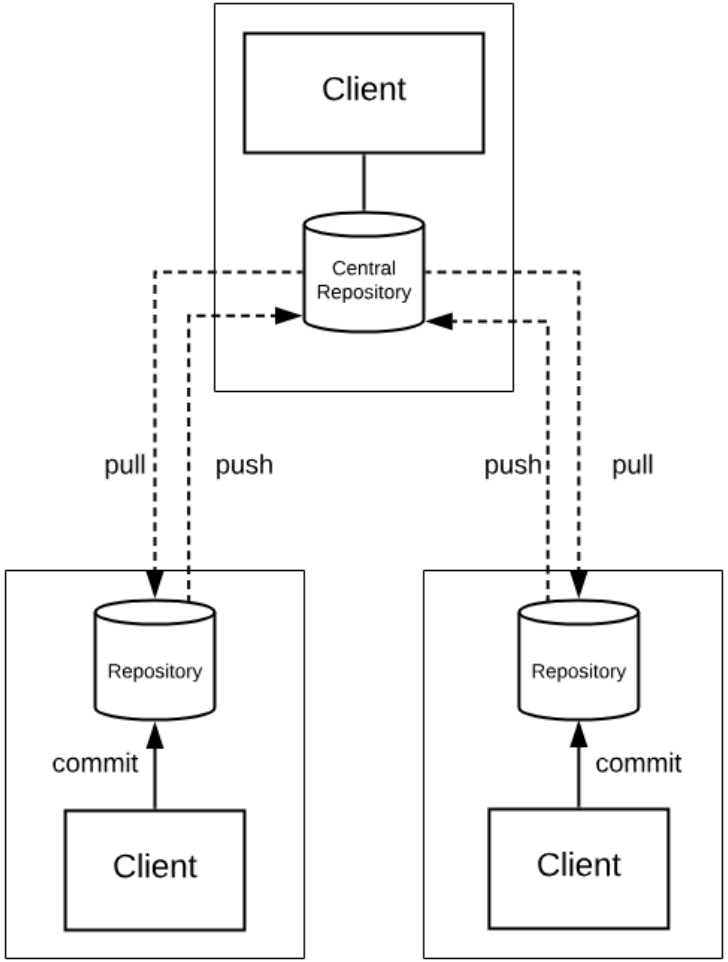
Version Control Systems (VCS)

- Essential for collaborative development
- They serve as the Source of Truth; maintaining the latest version
- Enables teams to recover previous versions

Centralized (example: svn, cvs)



Distributed
(example: git, mercurial)

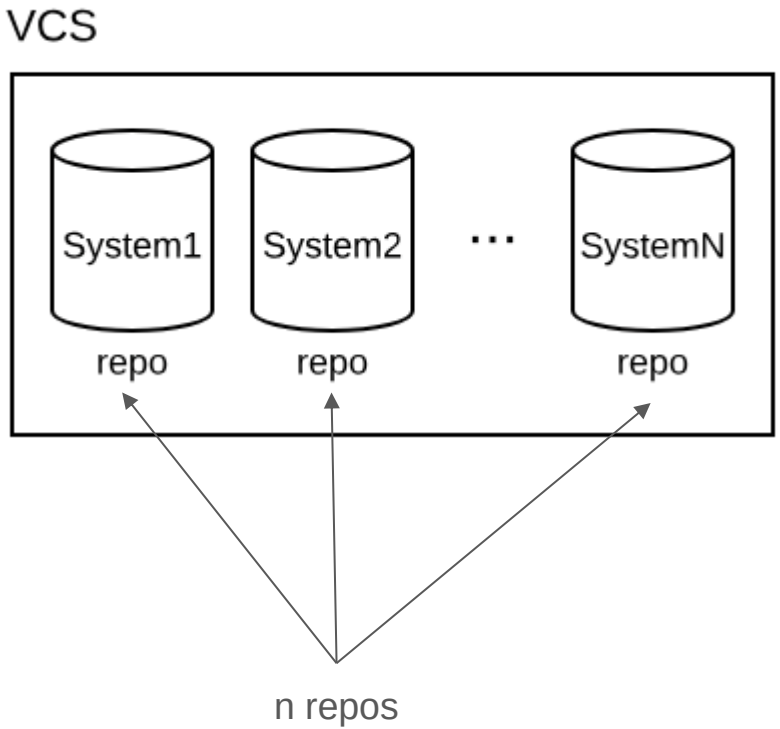


Advantages of DVCS

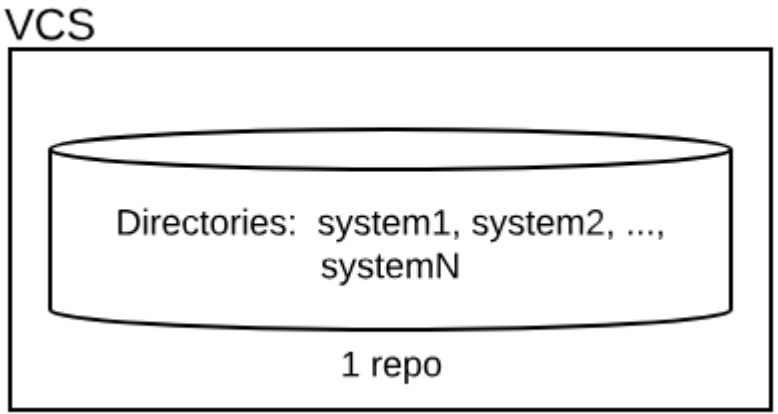
- Commits are faster; enabling devs to commit more often
- Each dev has a local repository, allowing offline work
- Supports alternative architectures: P2P, hierarchical, etc

Multirepos vs Monorepo

Multirepo (more commom)



Monorepo (less common; bigtechs)



Example: GitHub

Multirepos:

- my-org/system1
- my-org/system2
- my-org/system3

Monorepo:

- my-org/systems
- Folders:
  - system1
  - system2
  - system3

contributed articles

DOI:10.1145/2854146

Google's monolithic repository provides a common source of truth for tens of thousands of developers around the world.

BY RACHEL POTVIN AND JOSH LEVENBERG

Why Google Stores Billions of Lines of Code in a Single Repository

This article outlines the scale of that codebase and details Google's custom-built monolithic source repository and the reasons the model was chosen. Google uses a homegrown version-control system to host one large codebase visible to, and used by, most of the software developers in the company. This centralized system is the foundation of many of Google's developer workflows. Here, we provide background on the systems and workflows that make feasible managing and working productively with such a large repository. We explain Google's "trunk-based development" strategy and the support systems that structure workflow and keep Google's codebase healthy, including software for static analysis, code cleanup, and streamlined code review.

Google-Scale

Google's monolithic software repository, which is used by 95% of its software developers worldwide, meets the definition of an ultra-large-scale[4] system, providing evidence the single-source repository model can be scaled successfully.

The Google codebase includes approximately one billion files and has a history of approximately 35 million commits spanning Google's entire 18-year existence. The repository contains 86TB[a] of data, including approximately

a Total size of uncompressed content, excluding release branches.

Monorepos are primarily adopted by large tech companies

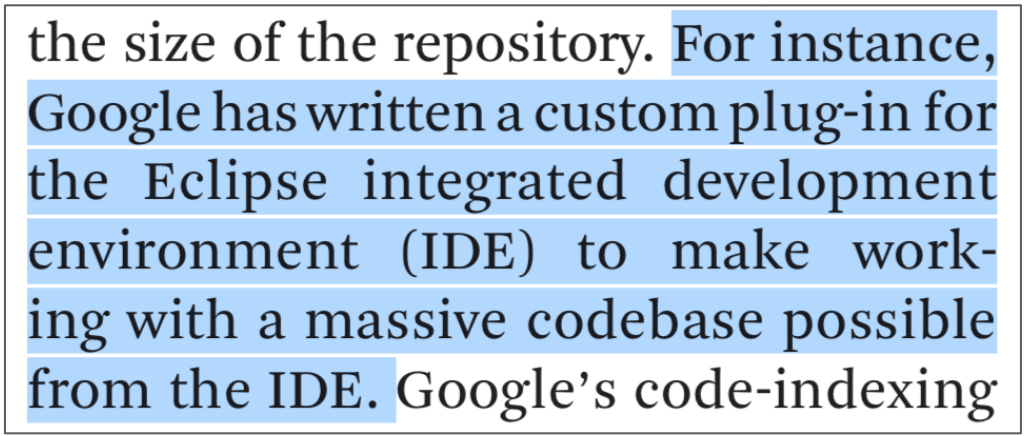
https://research.google/pubs/pub45424/

Advantages of Monorepos

- Provides a single source of truth
- Enables visibility and code reuse
- Ensures the same version of a library across all systems
- Supports atomic changes (1 commit can modify n systems)
- Enables large-scale refactorings

Disadvantage of Monorepos

- Requires specialized tools, such as online IDEs and build systems



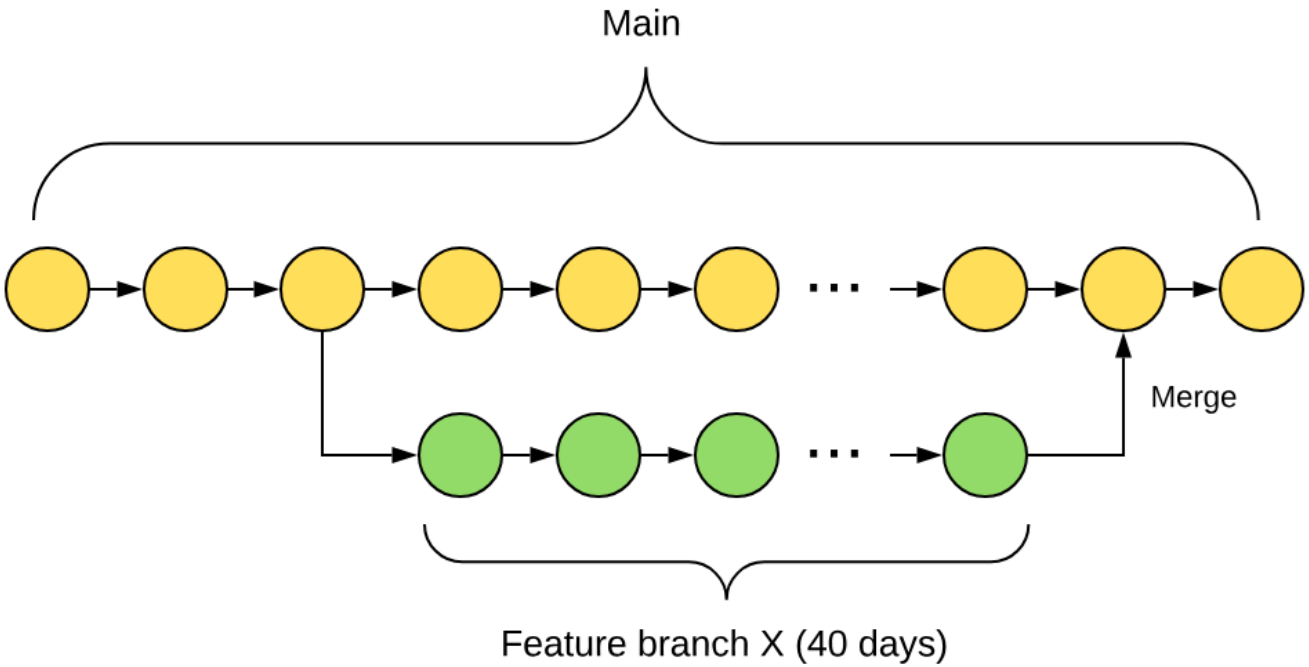
the size of the repository. For instance, Google has written a custom plug-in for the Eclipse integrated development environment (IDE) to make working with a massive codebase possible from the IDE. Google's code-indexing

For more information about Git, please refer to the appendix

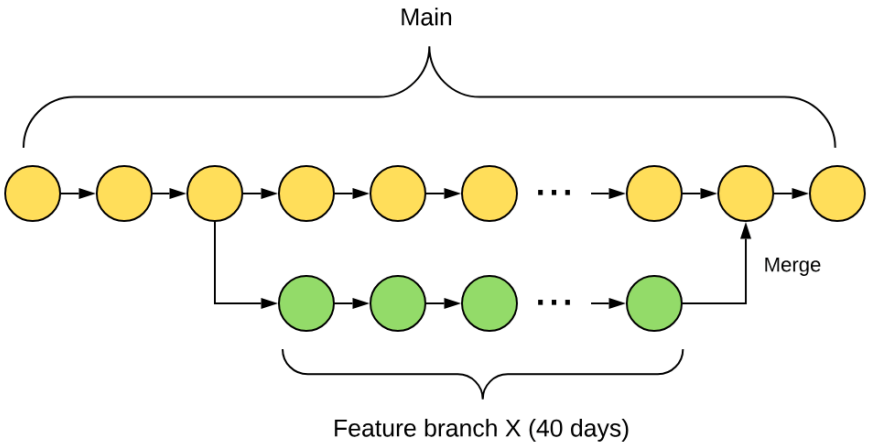
https://softengbook.org/chapterAp

Continuous Integration

In the past: feature branches were very common



Result after 40 days: **merge hell**



If a task causes pain, it's best not to let it accumulate; instead, tackle it daily

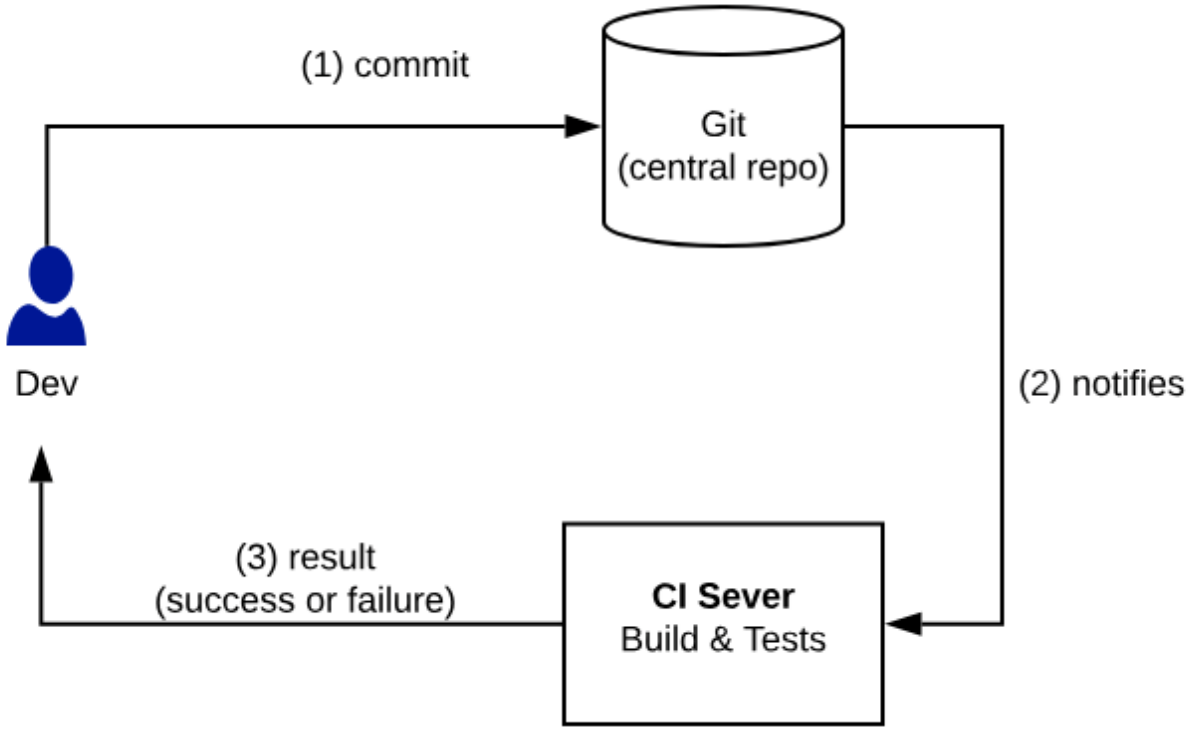
Continuous Integration (CI)

- First introduced in XP
- CI emphasizes frequent code integration into the main branch
- How often? Most authors recommend at least daily

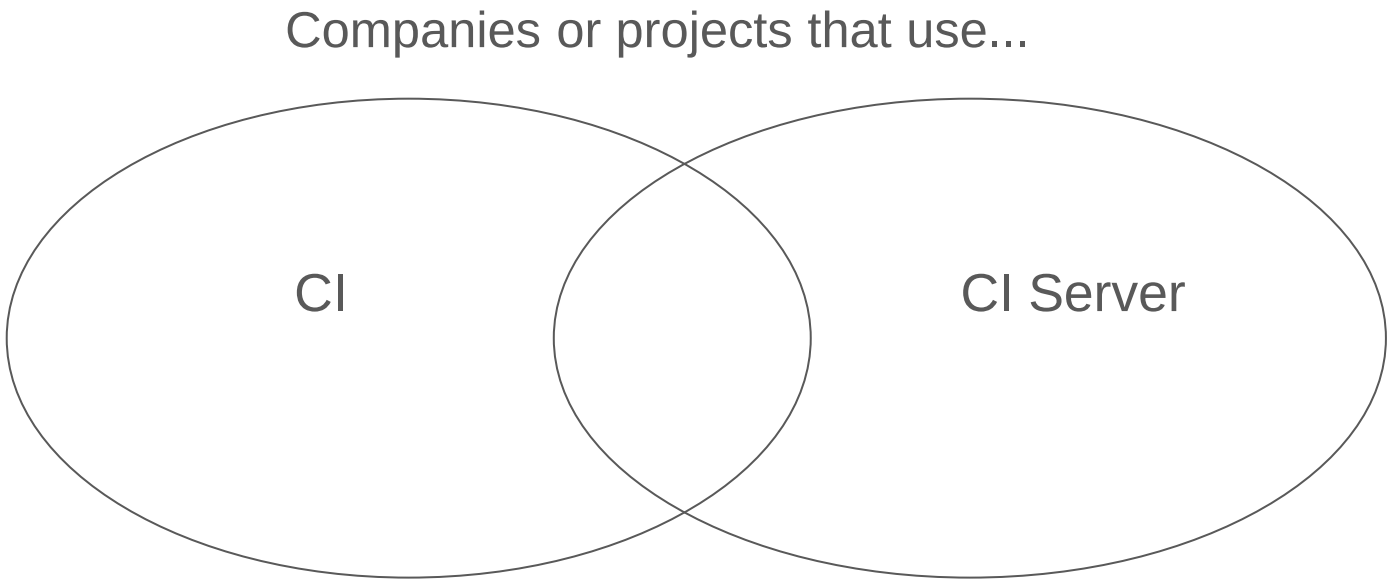
Key practices for effective CI implementation

- Automated builds
- Automated tests
- Pair programming

CI Servers


(1) commit
Git
(central repo)
Dev
(2) notifies
(3) result
(success or failure)
CI Sever
Build & Tests

Adopting CI is more than merely using a CI server



Branching Strategies

https://softengbook.org/articles/branching-strategies

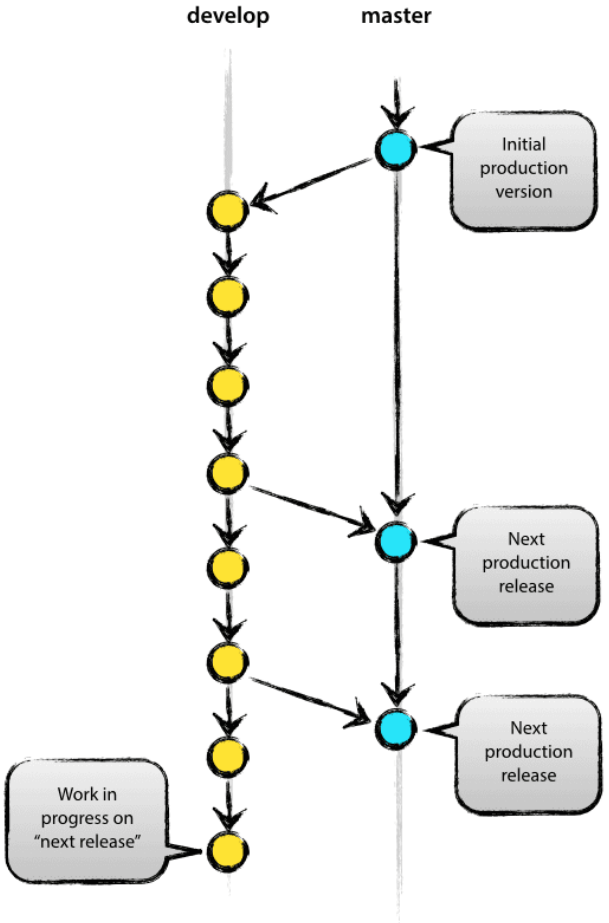
Branching Strategies

- How to organize and manage branches in a VCS
- Best practices for creating, merging, and deleting branches
- Common strategies:
  - Git-flow
  - GitHubFlow
  - Trunk-based Development

Git-Flow

Git-flow

- A widely-used branch strategy
- Two permanent branches:
  - master/main
  - develop



Source: https://nvie.com/posts/a-successful-git-branching-model/

Permanent Branches

- Main/master/trunk: production-ready code
- Develop: integration branch for completed features pending QA approval

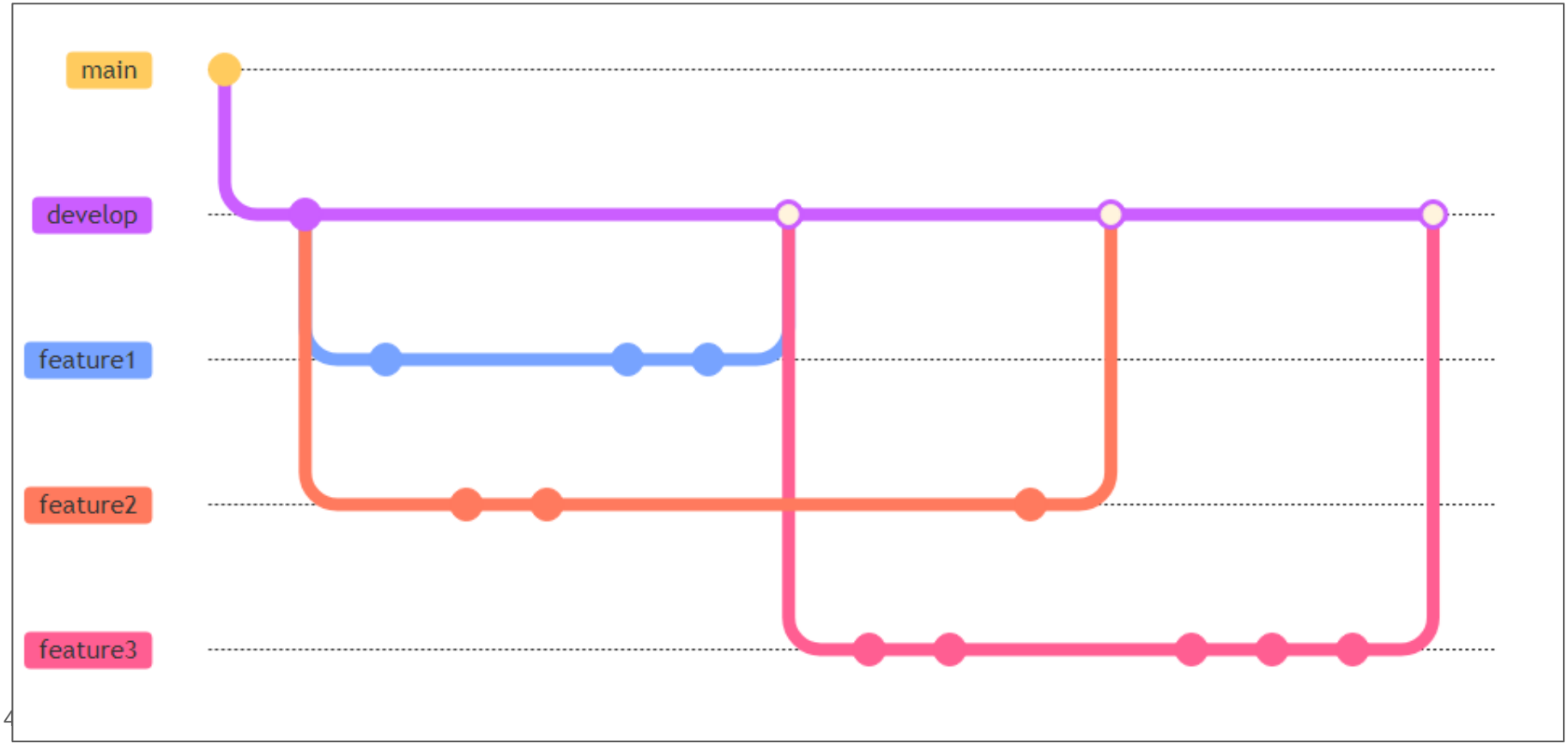
Temporary Branches

- Feature branches
- Release branches
- Hotfix branches

Feature Branches

- Used for implementing new features
- Branch flow:
  - Created from: develop
  - Merges into: develop
- Typically exists only in the developer's local repository

Feature Branches

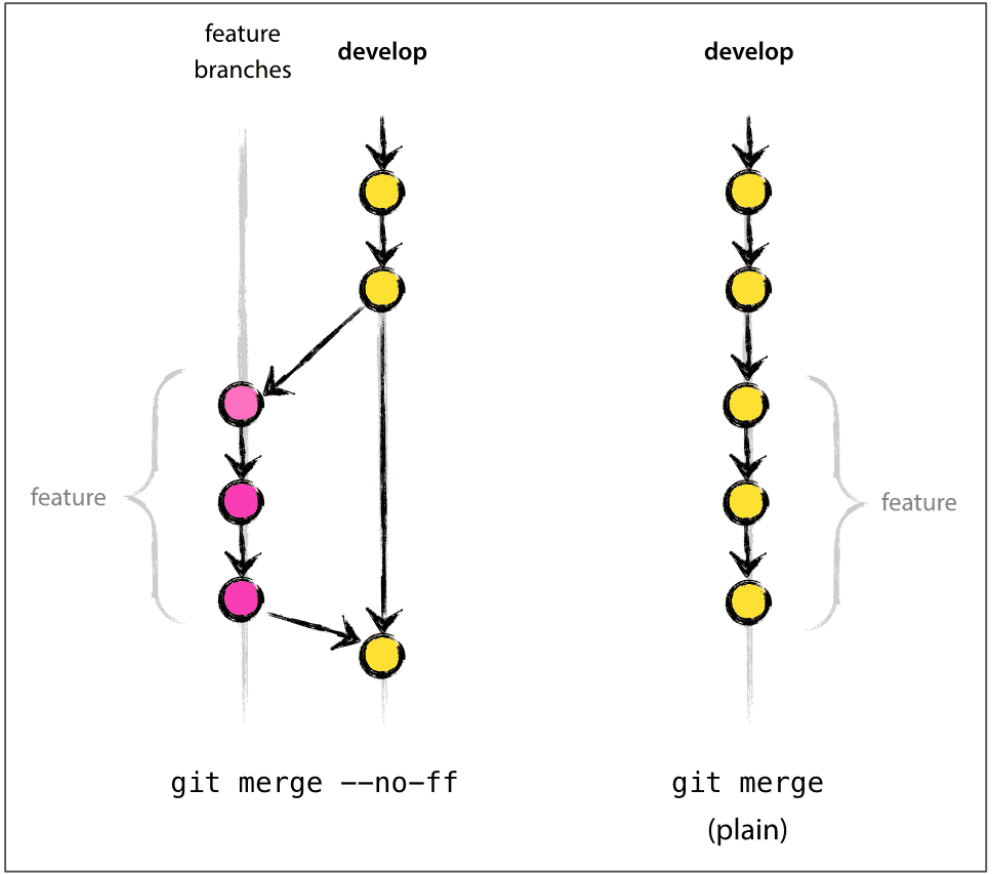


Commands for creating feature branches

```
git checkout -b feature-name develop   # creates feature branch from develop

[commits to implement feature]

git checkout develop                   # returns to develop

git merge --no-ff feature-name         # merges feature-name into develop
                                       # no-ff: no fast-forwarding (see next slide)

git branch -d feature-name             # deletes feature branch

git push origin develop                # Updates remote repo
```

git merge: with and without fast-forward

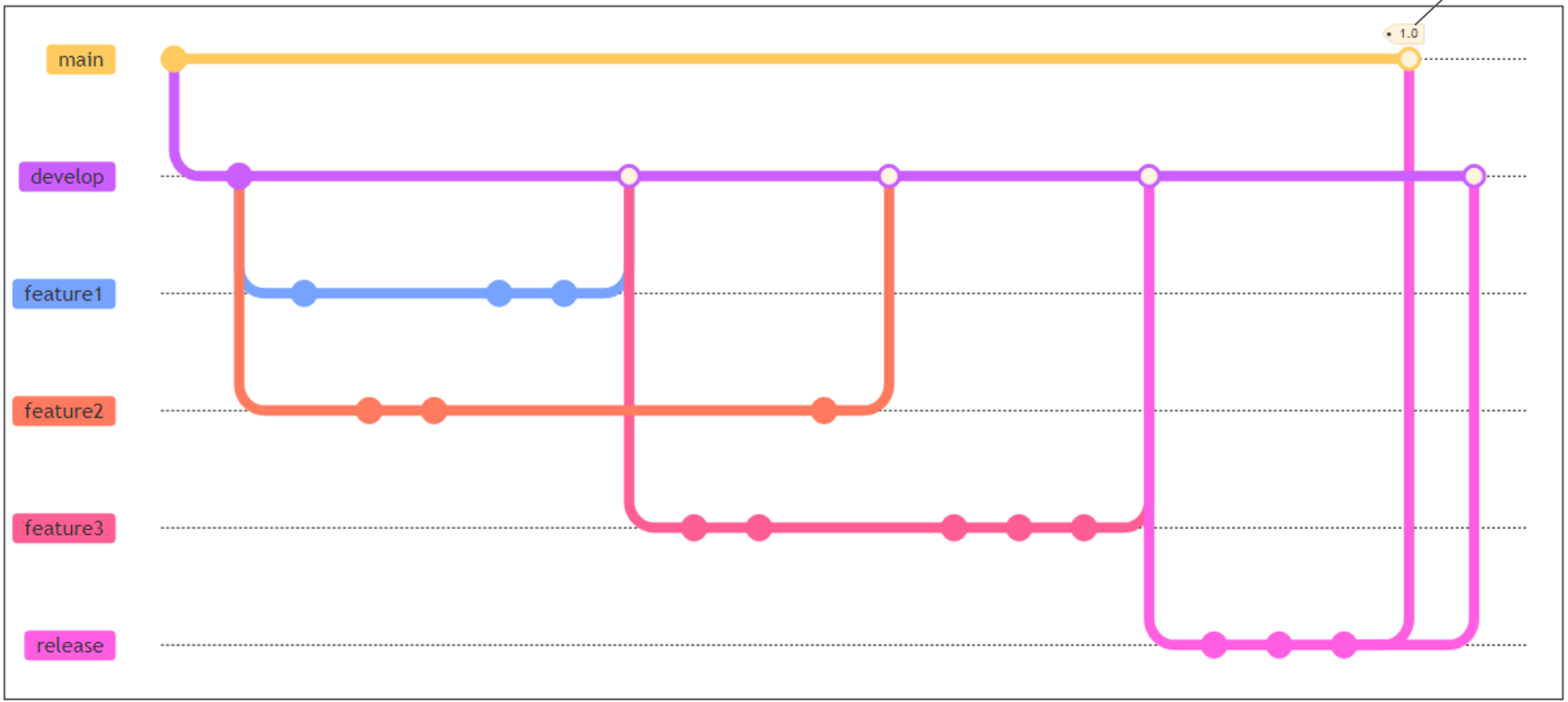
,



Release Branches

- Used to prepare a new release for customer approval
- Origin: develop
- Destination:
  - Merge into main (with the new release tag)
  - Merge into develop (with bug fixes)

Release Branch (last branch in the figure)



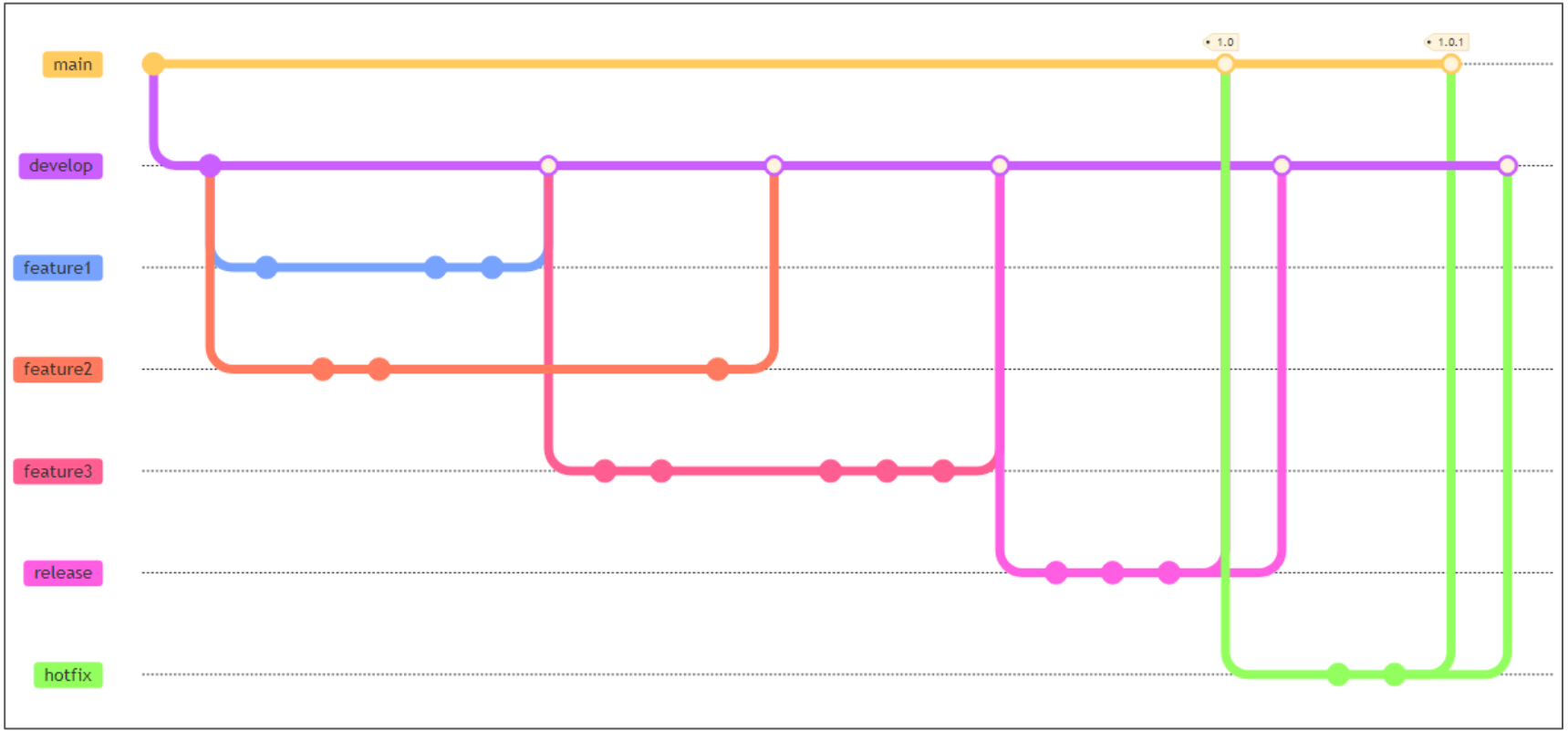
Commands for creating release branches

```
git checkout -b release-1.0 develop   # creates release branch from develop

[release commits]

git checkout main                     # switch to main
git merge --no-ff release-1.0         # merges into main
git tag -a 1.0                        # adds tag to main

git checkout develop                  # switch to develop
git merge --no-ff release-1.0         # merges into develop

git branch -d release-1.0             # removes release branch

git push origin develop               # pushes develop to remote repo
git push origin main                  # pushes main to remote repo
```

Hotfix Branches

- Used to fix critical bugs detected in production
- Origin: main (via the tag where the bug was reported)
- Destination:
  - Merge into main (with new version tag)
  - Merge into develop

Hotfix Branches (last branch in the figure)

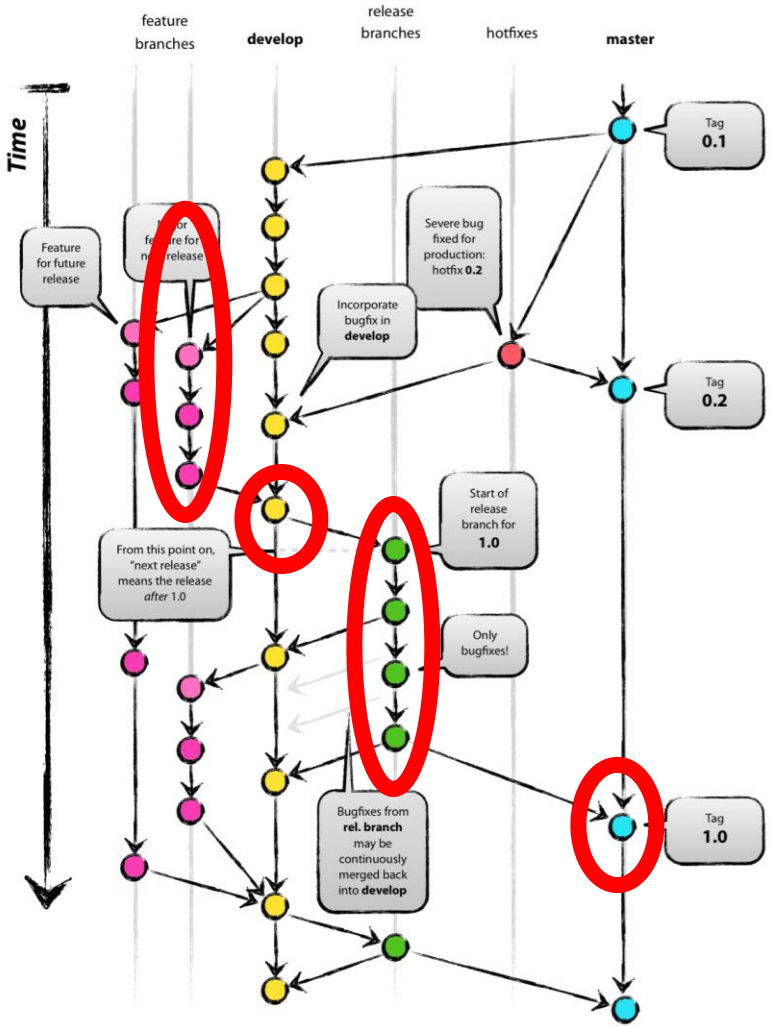


Commands for creating hotfix branches

```
git checkout -b hotfix-1.2.1 main   # creates hotfix branch from main

[hotfix commits]

git checkout main                   # switches to main
git merge --no-ff hotfix-1.2.1      # merges hotfix branch into main
git tag -a 1.2.1                    # adds tag to main

git checkout develop                # switches to develop
git merge --no-ff hotfix-1.2.1      # merges hotfix branch into develop

git branch -d hotfix-1.2.1          # deletes hotfix branch

git push origin develop             # pushes develop to remote repo
git push origin main                # pushes main to remote repo
```

Git-flow: Summary

Feature ⇒ Develop ⇒ Release ⇒
⇒ Main



Git-flow: Usage and disadvantages

- Recommended when:
  - You have several customers with different versions
  - You maintain manual testing and QA teams
  - Releases need customer approval
- Disadvantages:
  - Can lead to long-lived branches and increased conflicts
  - Results in longer customer feedback cycles

Exercises

1. Can we implement Continuous Integration (CI) with Git-flow? Explain your reasoning.
2. How can CI servers be integrated into Git-flow projects? Which branches should be monitored by CI servers?

GitHub Flow

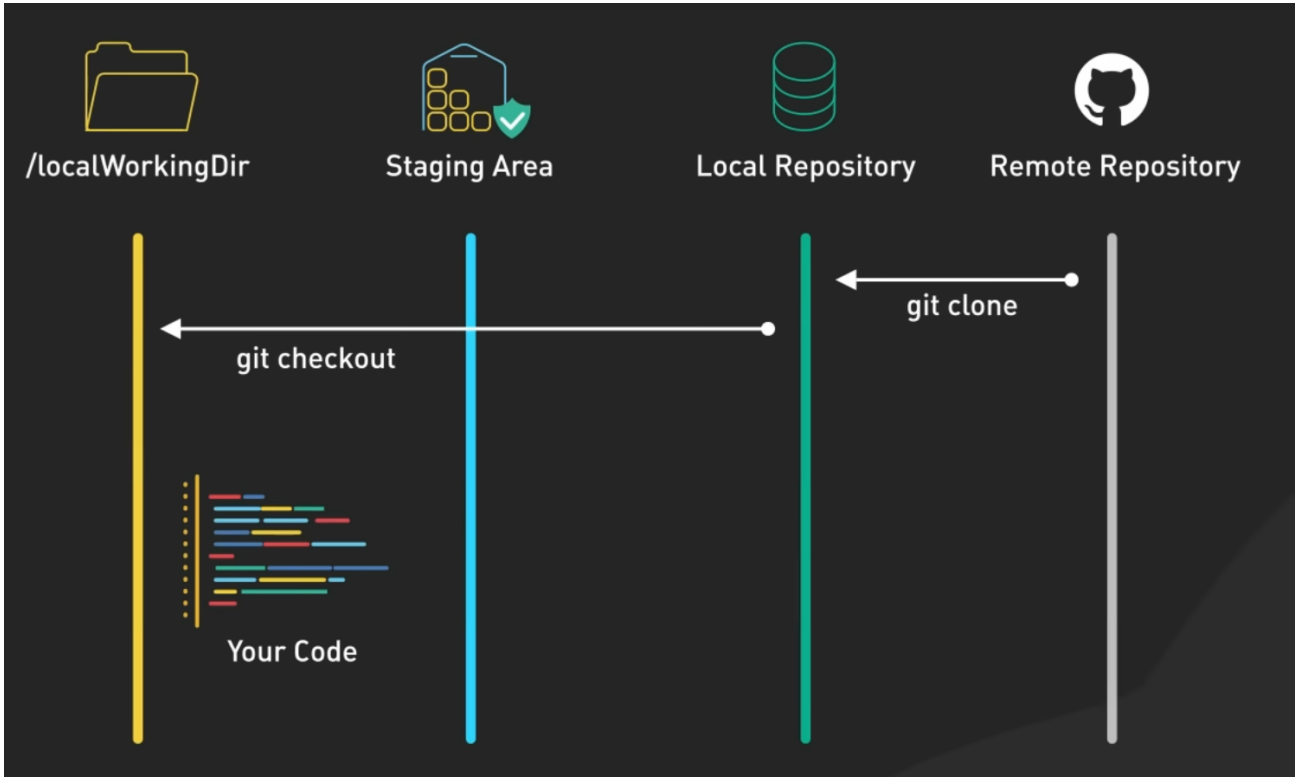
GitHub Flow

- A simplified workflow common in GitHub projects
- Key characteristics:
  - Only feature and main branches
  - Pull Requests (PRs)
  - No develop, release, and hotfix branches

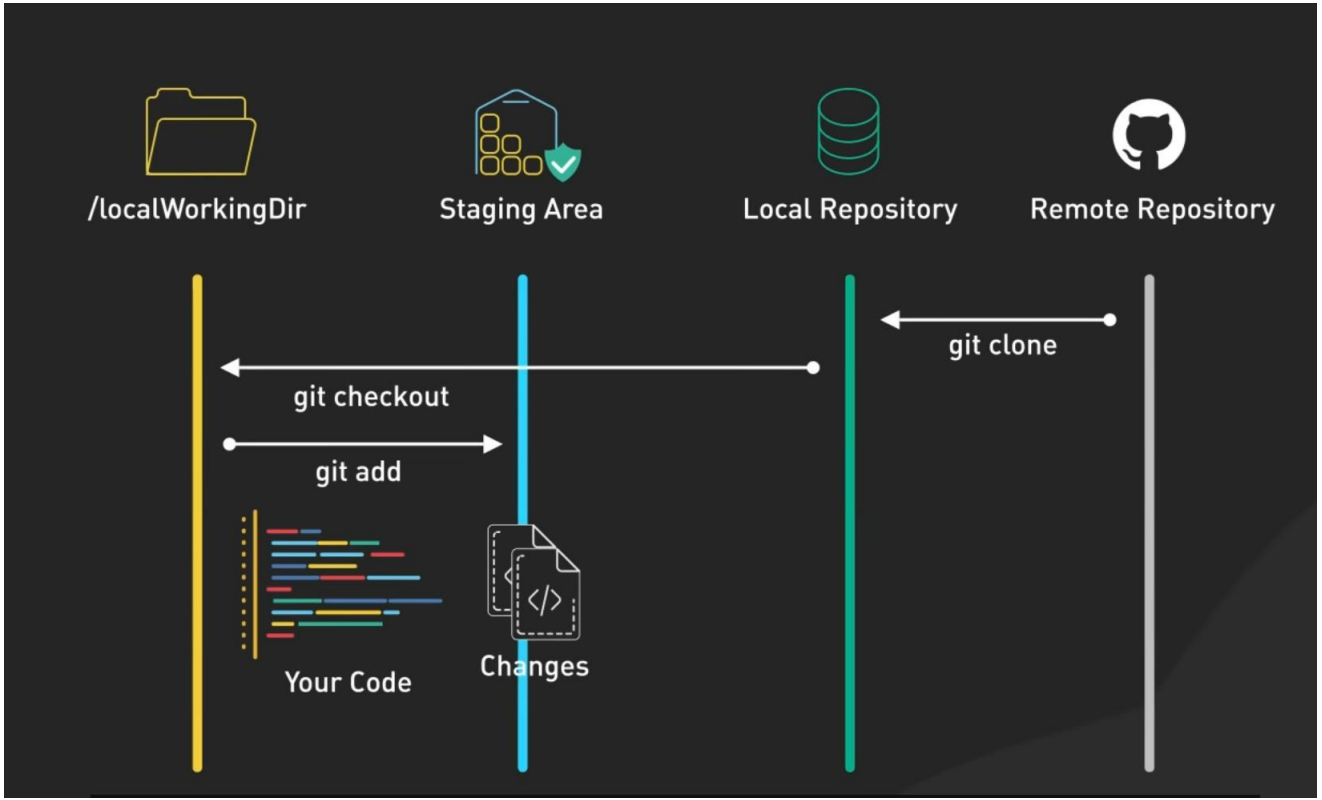
GitHub Flow Steps

1. Dev creates a feature branch in their local repository
2. Dev implements the feature
3. Dev pushes the branch to GitHub
4. Dev creates a PR on GitHub
5. PR enables code review by team members
6. Reviewer examines changes and merges the PR into main

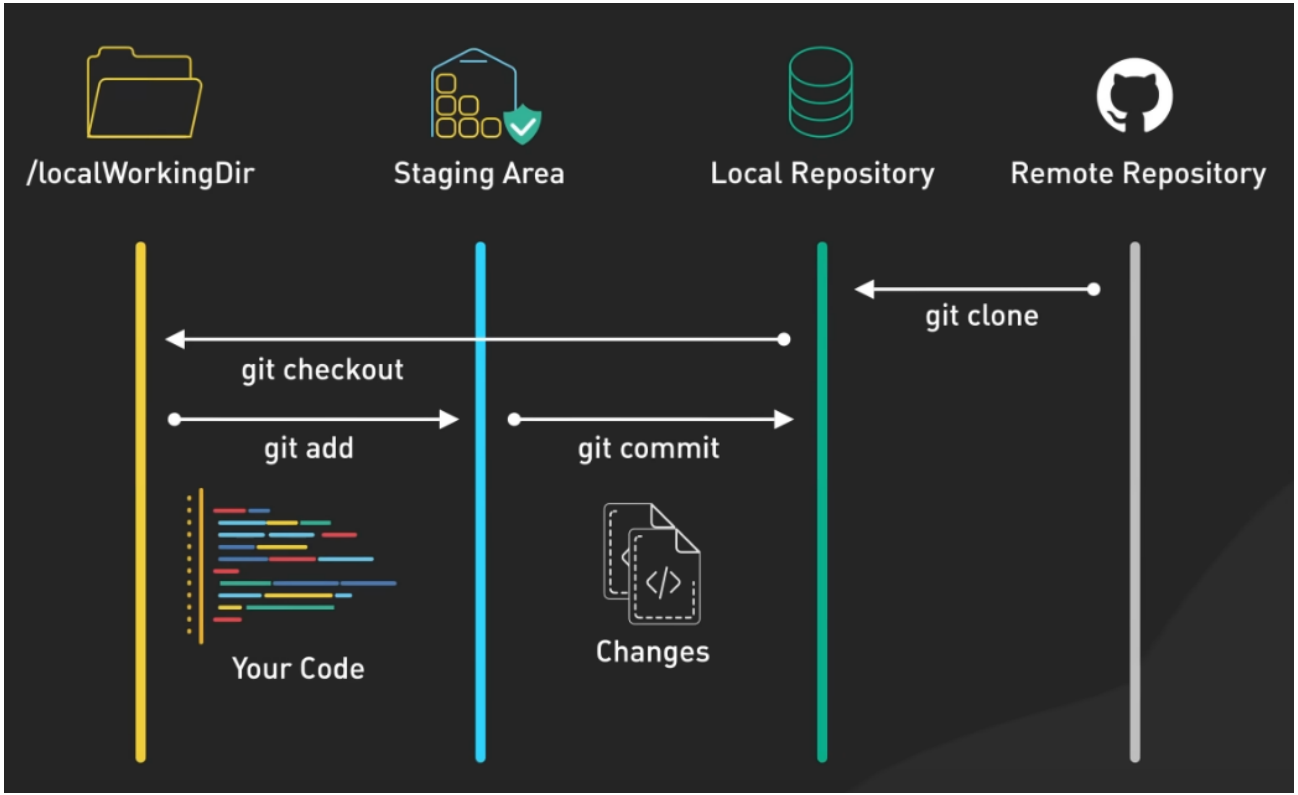
GitHub-flow


/localWorkingDir
Staging Area
Local Repository
Remote Repository
git clone
git checkout
Your Code

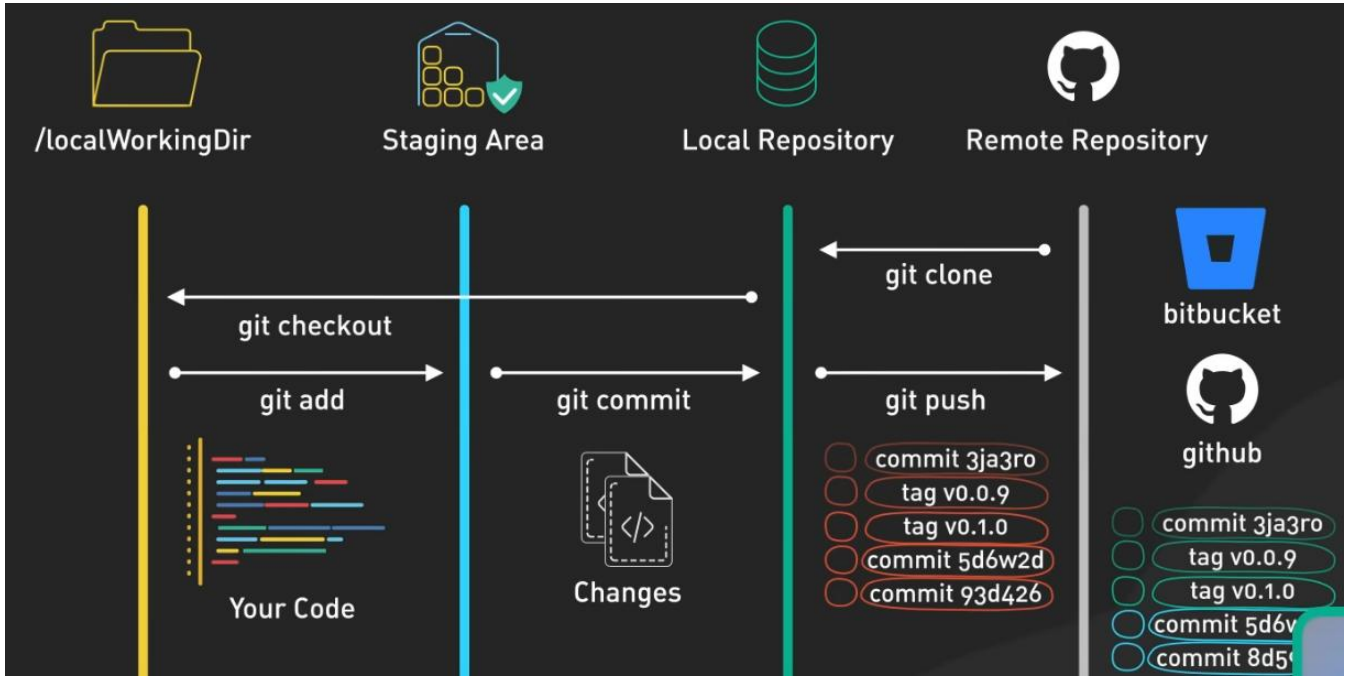
GitHub-flow


/localWorkingDir
Staging Area
Local Repository
Remote Repository
git clone
git checkout
git add
Your Code
Changes

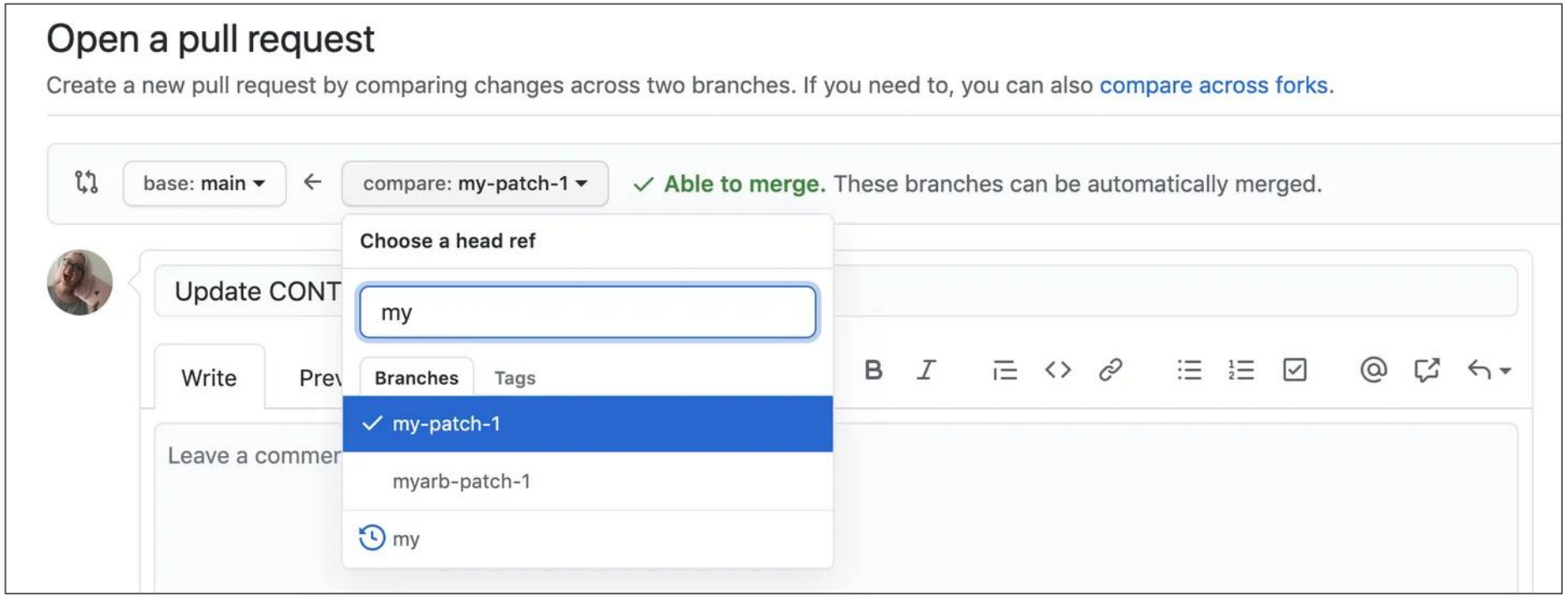
GitHub-flow


/localWorkingDir
Staging Area
Local Repository
Remote Repository
git clone
git checkout
git add
git commit
Your Code
Changes

GitHub-flow


/localWorkingDir
Staging Area
Local Repository
Remote Repository
git checkout
git add
git commit
git clone
git push
bitbucket
github
Your Code
Changes
commit 3ja3ro
tag v0.0.9
tag v0.1.0
commit 5d6w2d
commit 93d426
commit 3ja3ro
tag v0.0.9
tag v0.1.0
commit 5d6v
commit 8d5

Pull Request



GitHub Flow: Usage and Disadvantage

- When to use: systems with only one version in production, such as Web systems
- Challenges: PRs can take a long time to be reviewed

GitHub Flow: Cook Book

Assumptions:

•You have successfully cloned the repository using

```
git clone https://github.com/snsmssss/Agile-SE.git.
```

•You have created a separate branch for your feature development (best practice). If you *haven't* created a separate branch, you're developing directly on your local main branch, which is generally discouraged.

•You are comfortable using the command line.

Detailed Steps:

1.Navigate to the local repository:

```
cd Agile-SE
```

•**Create a feature branch (if you haven't already):** If you *didn't* create a branch before, do so now. *Don't* develop directly on main. This makes it easier to isolate your changes and manage pull requests.

Replace `<feature_branch_name> with a descriptive name for your feature`

```
(e.g., add-new-feature, fix-bug
git checkout -b <feature_branch_name>
```

For example:

```
git checkout -b add-documentation
```

•**Make your changes:** Make the necessary code modifications, additions, and deletions related to your feature in your working directory.

Stage your changes: Use the `git add command to stage the files you have modified:`

```
git add .
```

Or, to add specific files:

```
git add file1.txt file2.py
```

•**Commit your changes:** Commit the staged changes with a descriptive commit message:

```
git commit -m "Implement feature: Add new documentation"
```

•**Push your feature branch to the remote repository:**

```
git push -u origin <feature_branch_name>
```

The `-u flag (short for --set-upstream) sets up a tracking connection between your local branch and the remote branch. This means that in the future, you can simply use git push and git pull without specifying the remote and branch name.`

•**Create a Pull Request (PR) on GitHub:**

•Go to the GitHub website for your repository：

```
https://github.com/snsmssss/Agile-SE
```

•You should see a banner at the top suggesting you create a pull request from your feature branch.

•Click the "Compare & pull request" button.

•Provide a clear title and description for your pull request, explaining the changes you have made.

•Review the "Files changed" tab to ensure that all your changes are included.

•Click "**`Create pull request`**".

•**Code Review:** Wait for the repository maintainers (likely yourself, in this case,

•unless it's a collaborative project) to review your pull request. They may provide feedback or request changes.

•**Address Review Feedback (if any):** If reviewers request changes:

•Make the necessary changes in your local feature branch.

•Stage and commit the changes:

```
bash
git add .
git commit -m "Address review feedback: fix typo, improve comments"
```

•Push the updated branch to the remote repository:

```
bash
git push
```

The changes automatically appear in the pull request.

•**Merge the Pull Request:** Once the review is complete and any necessary changes have been made, the repository maintainer (again, likely yourself) will merge your pull request into the `main branch`.

•On the GitHub website, go to your pull request and click "**`Merge pull request`**".

•You can choose to "Squash and merge", "Merge", or "Rebase and merge". "Squash and merge" is often a good option, as it combines all commits on your feature branch into a single commit on the main branch, resulting in a cleaner commit history. Choose the option that best suits your workflow.

Confirm the merge.

•(Optionally) Delete the feature branch after merging (if it's no longer needed).

•**Update your local** `main branch:` After the pull request has been merged, update your local main branch:

```
bash
git checkout main   # Switch to the main branch
git pull origin main # Pull the latest changes from the remote main branch
```

•**(Optional) Delete your local feature branch:**

```
bash
git branch -d <feature_branch_name>
```

This deletes the local branch. Be sure you have pushed all changes and merged the PR *before* deleting the branch.

If you get an error that the branch hasn't been merged, use

```
git branch -D <feature_branch_name> (capital -D)
```

only if you are absolutely sure you want to delete the branch, even if it hasn't been merged.

Explanation of Key Commands

git checkout -b <feature_branch_name>: Creates and switches to a new branch.

git add .: Stages all changes in the current directory.

git commit -m "Your commit message": Commits the staged changes.

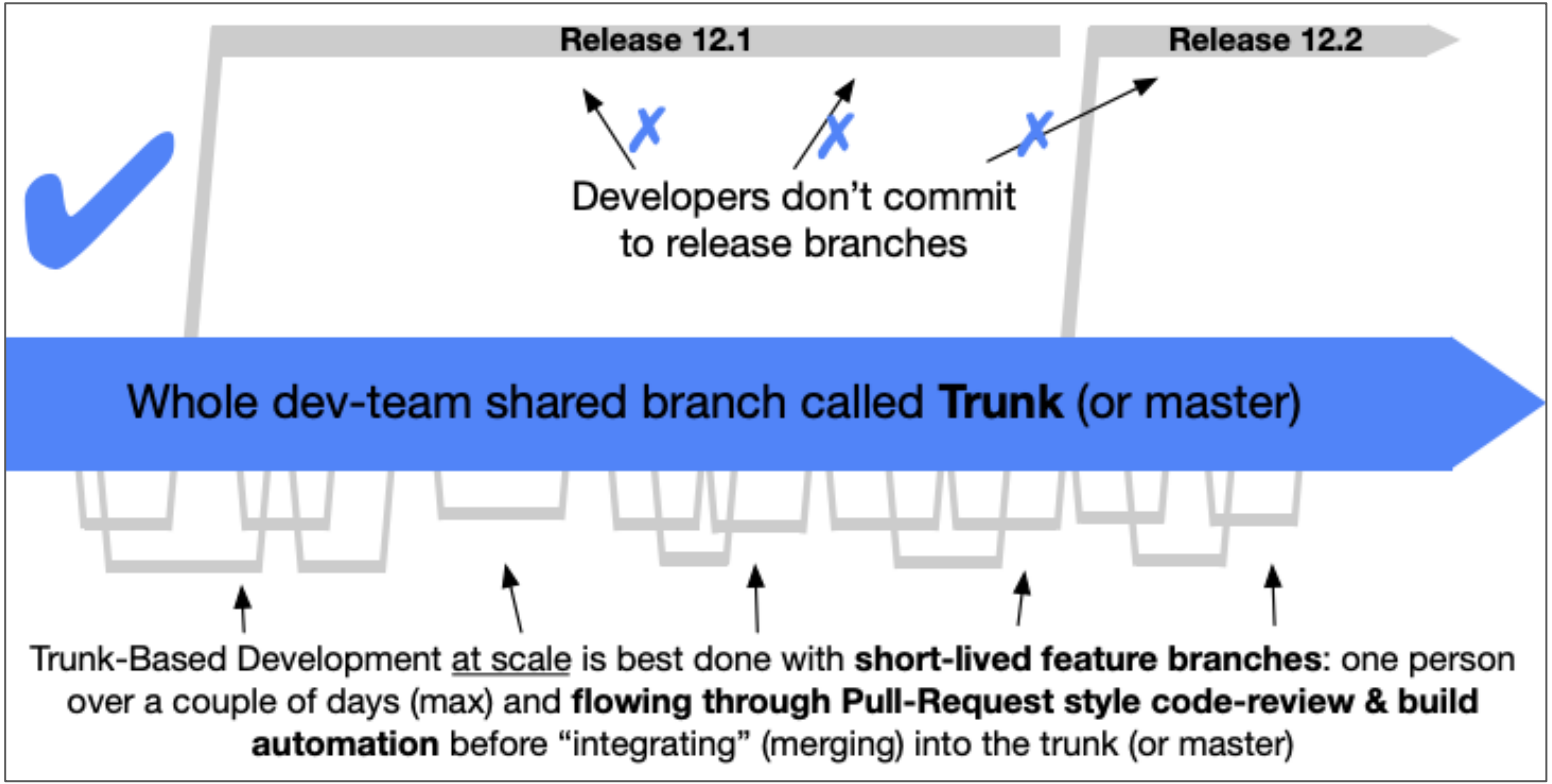
git push -u origin <feature_branch_name>: Pushes the branch to the remote repository.

git pull origin main: Updates your local main branch with the remote main branch.

Trunk-based Development (TBD)

Trunk-based Development (TBD)

- Development should occur directly on the main (or trunk)
- No develop branches
- Goal: minimize merge conflicts



Source: https://trunkbaseddevelopment.com

"Almost all development occurs at the HEAD of the repository, not on branches. This helps identify integration problems early and minimizes the amount of merging work needed. It also makes it much easier and faster to push out security fixes."

"All front-end engineers work on a single stable branch of the code, which also promotes rapid development, since no effort is spent on merging long-lived branches into the trunk."

Continuous Deployment

Continuous Deployment (CD)

- CI: integrate code frequently
- CD: integrated code goes immediately into production
- Goal: rapid experimentation and feedback!

How to keep partial implementations from reaching customers?

Feature Flags (also called feature toggles)

```
featureX = false;
...
if (featureX)
   "here is incomplete code for X"
...
if (featureX)
   "more incomplete code for X"
```


While the feature is being developed!

When the code is complete: enable the flag

```
featureX = true;
...
if (featureX)
   "here is incomplete code for X"
...
if (featureX)
   "more incomplete code for X"
```

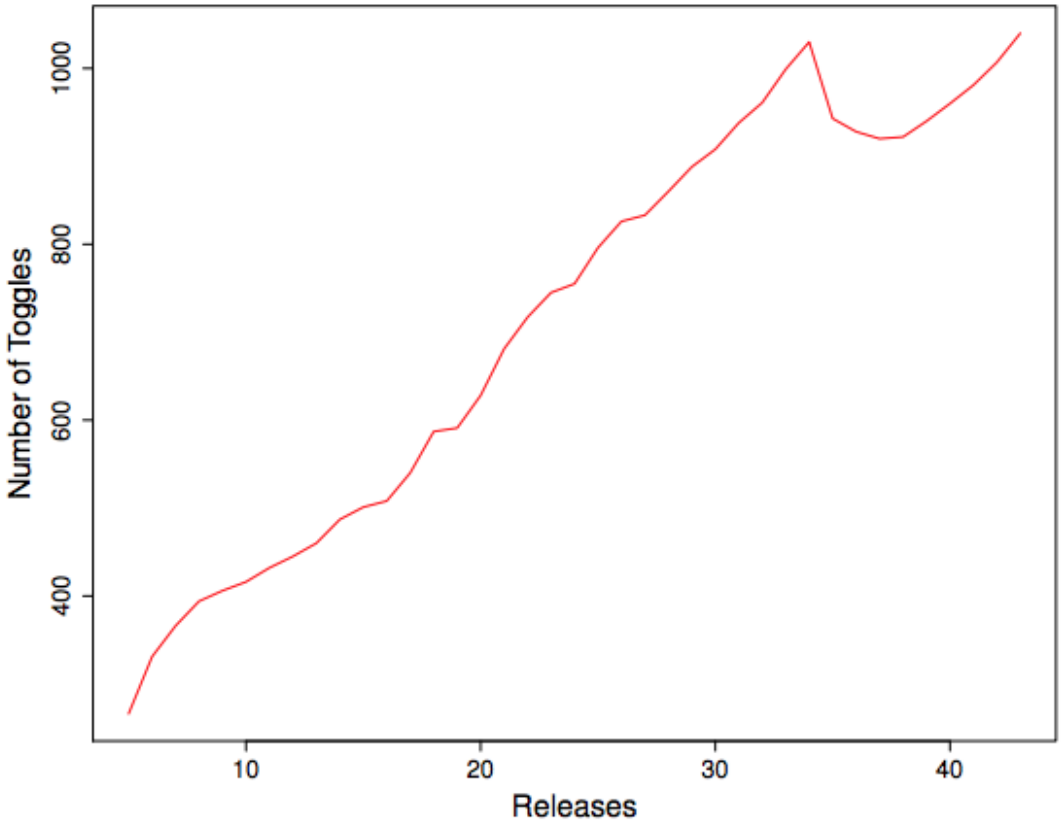


Figure 2: Number of unique toggles per release of Google Chrome.

Md Tajmilur Rahman et al. Feature toggles: practitioner practices and a case study. MSR 2016.

Exercises

1. Assume the following function:

```
String highlight_text(String text, String word) {
    // “text” is a text in markdown
    // search all instances of “word” in “text”
    // convert word to bold (**word**, in markdown)
}
```

Assume that you are working in your local repo on a code that calls “highlight_text”. Describe a change (push) made to this function, by another developer, that:

(a) causes a compilation error in your code (after a pull)?

(b) causes a logic error in your code (after a pull)?

2. Define (and distinguish) the following practices:

- Continuous Integration
- Continuous Delivery
- Continuous Deployment

Continuous
Integration
(example: daily)

GitHub

Continuous
Deployment
(automatically)

Production
Server

Continuous
Integration
(example: daily)

GitHub

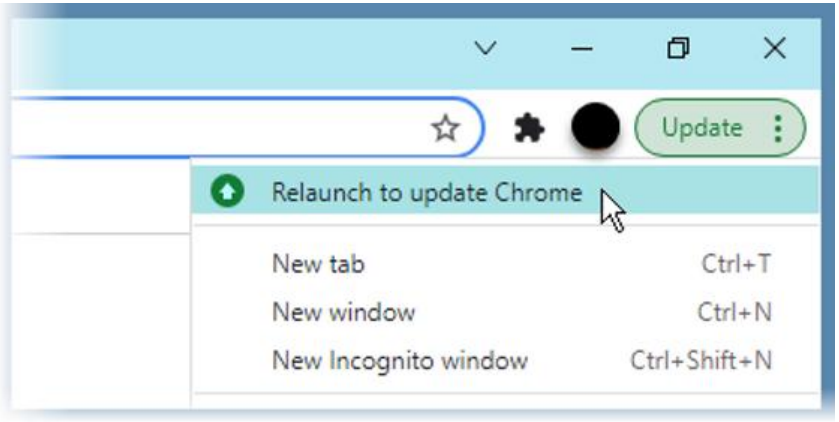
Continuous Delivery
(deployment must be
manually approved)

Production
Server

3. Suppose you were hired by a company that produces printers and became responsible for defining the DevOps practices adopted in the implementation of the printers' drivers.

Which of the following practices would you recommend in this case: continuous deployment or continuous delivery? Provide a brief justification.

4. In a browser like Chrome, which practice is more suitable: Continuous Delivery or Deployment? Provide a brief justification.



5. What is the best type of system for using Continuous Deployment? Justify.

6. Languages like C support conditional compilation directives like #ifdef and #endif. What are the key differences between these directives and feature flags?

```
#include <iostream>

#ifdef _WIN32
    #include <windows.h>
    void clearScreen() {
        system("cls");
    }
#else
    #include <unistd.h>
    void clearScreen() {
        system("clear");
    }
#endif
```

```
int main() {
    std::cout << "This program will
       clear the screen in 3 secs" <<
       std::endl;
    sleep(3);
    clearScreen();
    std::cout << "Screen cleared!"
}
```

To compile:

```
g++ -D__unix__ -o clear_screen clear_screen.cpp
```

7. In the context of TBD, feature flags are used to disable implementations that are not ready for production. However, in other contexts, feature flags can be used to enable or disable general features. Provide an example of a system and describe some of the features that can be turned on or off.

8. What are the key differences between an A/B Test and a canary release?

In summary, feature flags are used to:

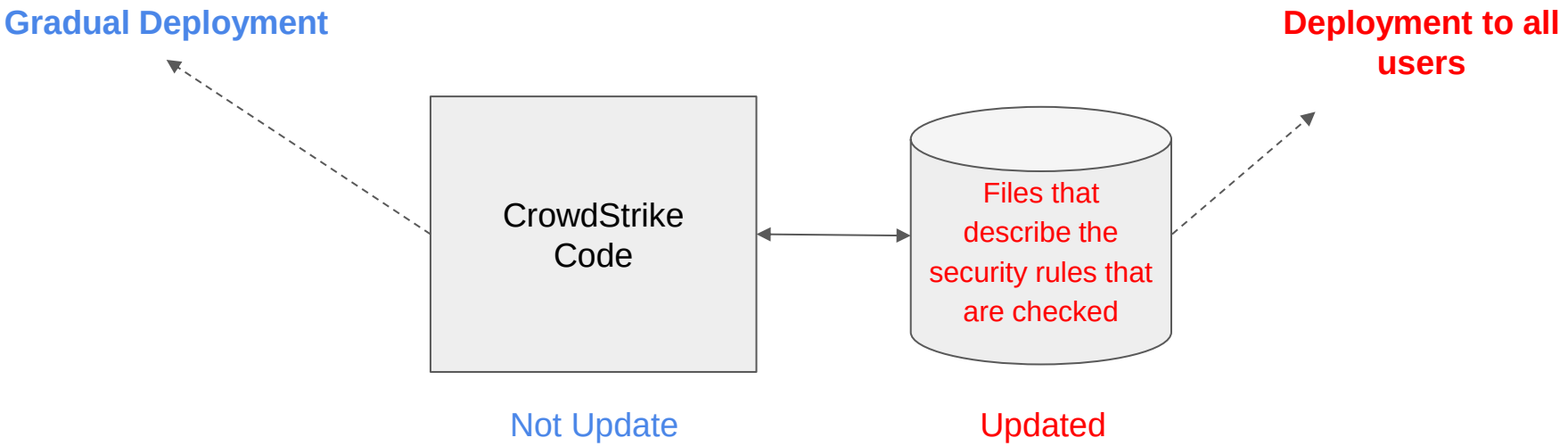
1. Control the release of untested or incomplete features when using Continuous Deployment (our focus in Ch. 10)
2. Enable/disable optional features
3. Conduct A/B testing
4. Implement canary releases

9. Complete this table assuming a company that uses git-flow.

Type of Branch	Origin Branch	Destination Branch(es)
Feature		
Release		
Hotfix		

10. During July 2024, a failure in one of CrowdStrike's attack-protection systems caused "blue screens" on over eight million Windows machines worldwide. Discuss which DevOps practice could have been used to prevent this incident.

.


Gradual Deployment
Deployment to all users
CrowdStrike Code
Files that describe the security rules that are checked
Not Update
Updated

End