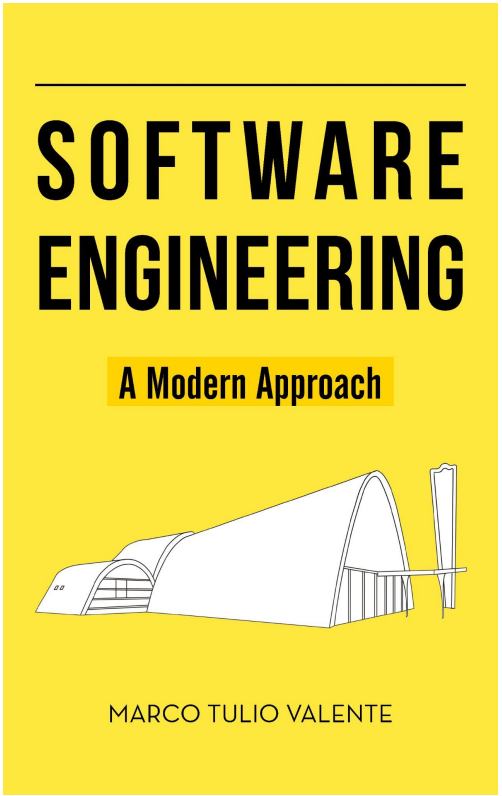


Chapter 10 - DevOps

Prof. Marco Tulio Valente

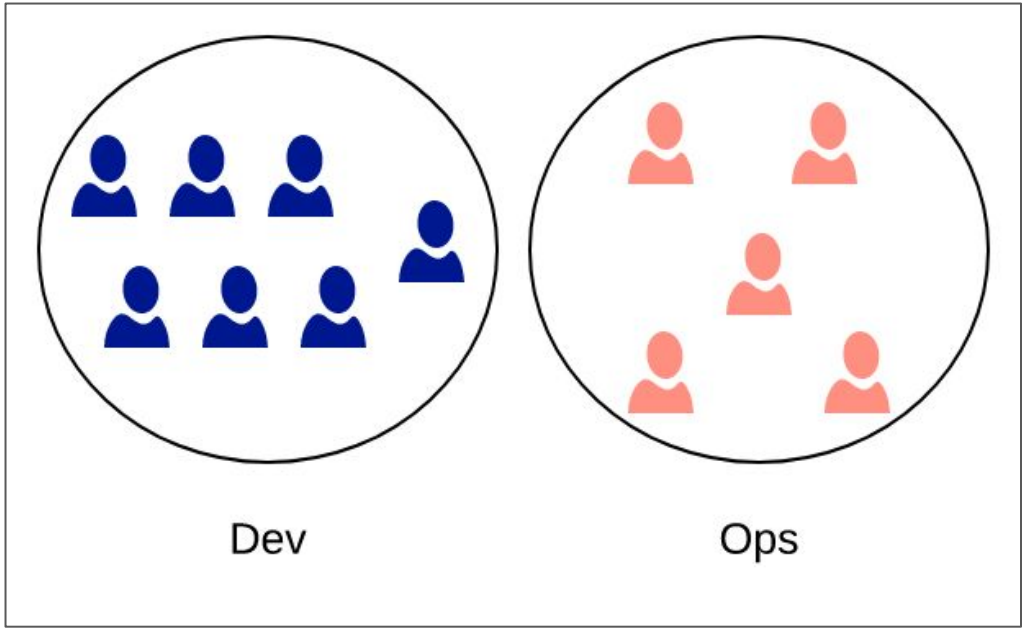
https://softengbook.org

Our Situation in the Course

- We defined and used a process to implement a system
- The requirements have been defined and implemented
- The design and architecture have been defined
- Various tests have been implemented
- And many refactorings have been performed

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

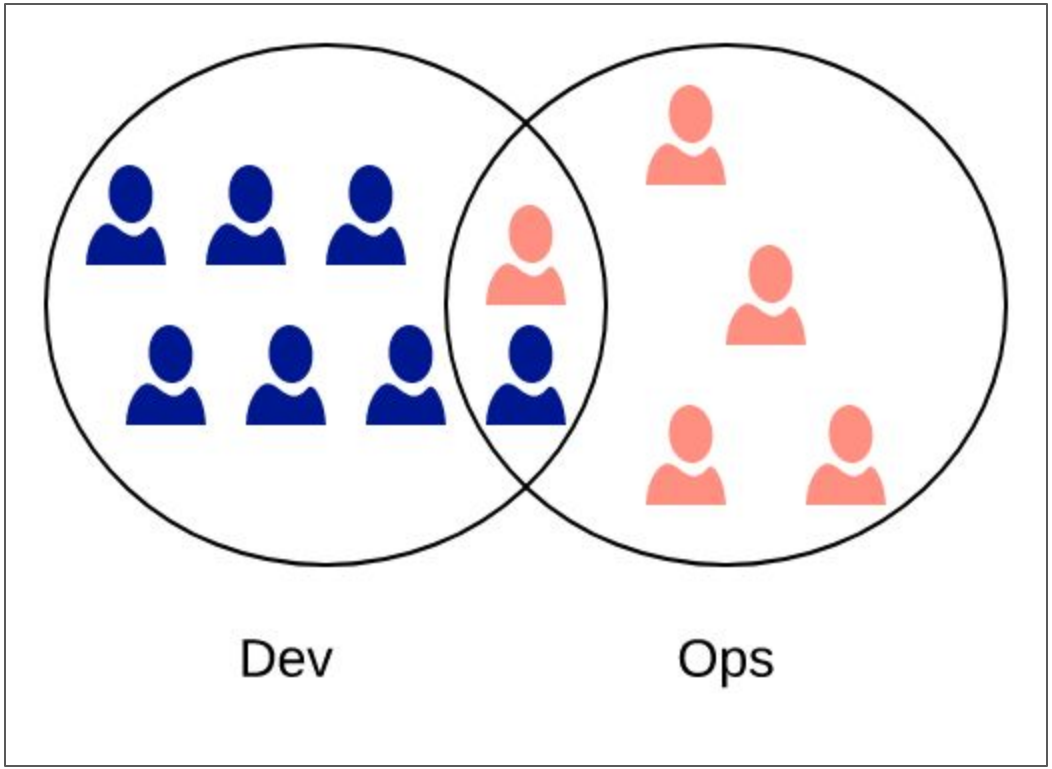
In the past, deployment was a traumatic process

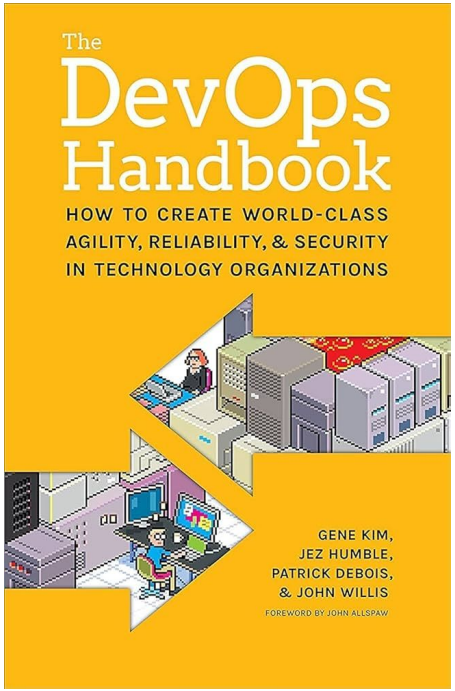


Two independent silos, with very little communication

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

Central idea of DevOps: bringing Dev and Ops closer





"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."

Objective: successful handover!

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

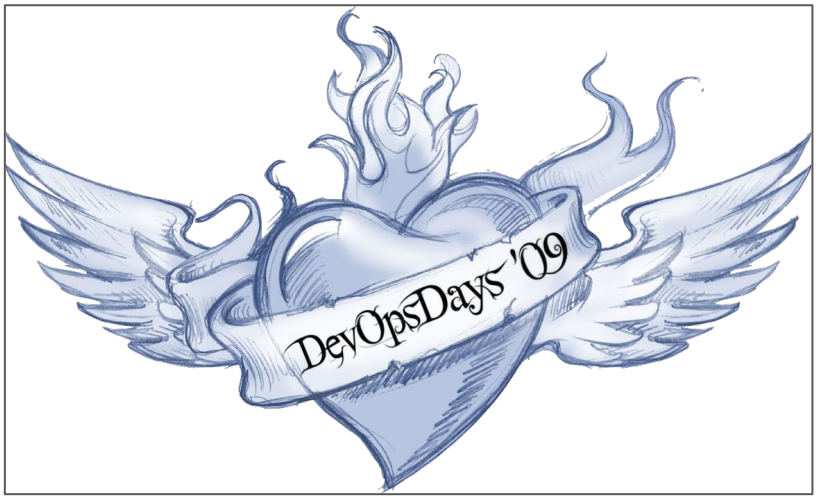
Objective: end the blame game

Dev: the problem is not my code, but your server

Ops: the problem is not my server, but your code

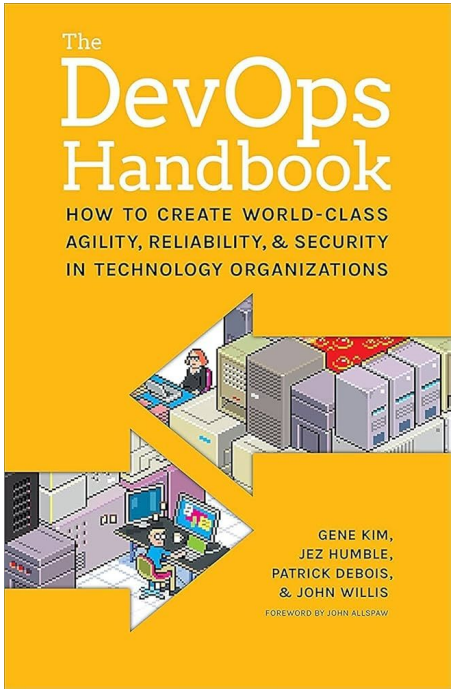
DevOps

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



DevOps Principles

- Bring Devs and Ops closer, from the start of the project
- Follow an agile mindset also in the deployment phase
- Turn deployments into a non-event
- Deploy parts of a system 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

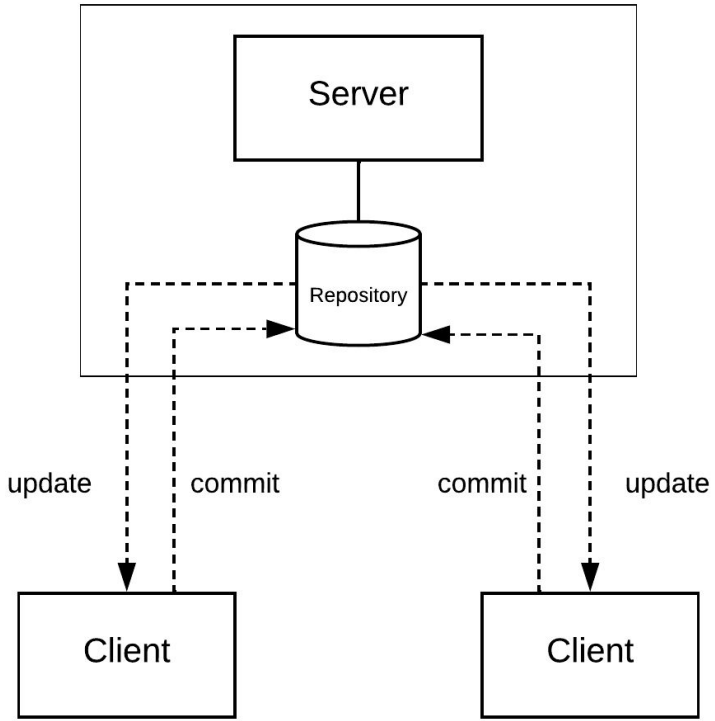
- Version Control
- Continuous Integration
- Branching Strategies
- Continuous Deployment
- Feature Flags

Version Control

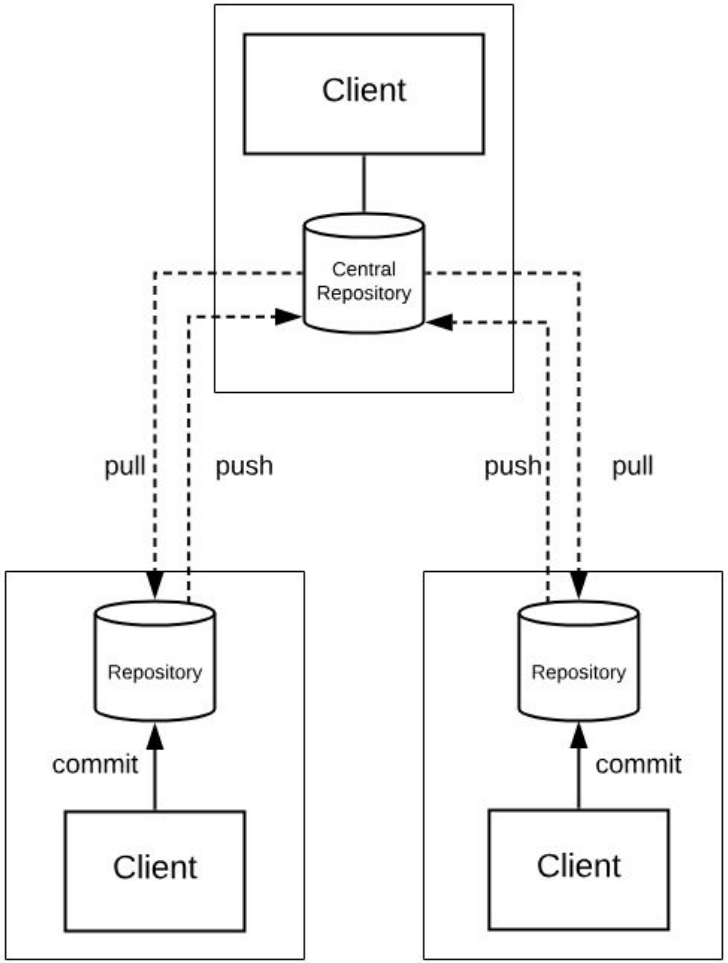
Version Control

- Essential for collaborative development
- VCS: Source of Truth; stores the latest version
- Allows to recover previous versions

Centralized (example: svn, cvs)



Distributed
(example: git, mercurial)

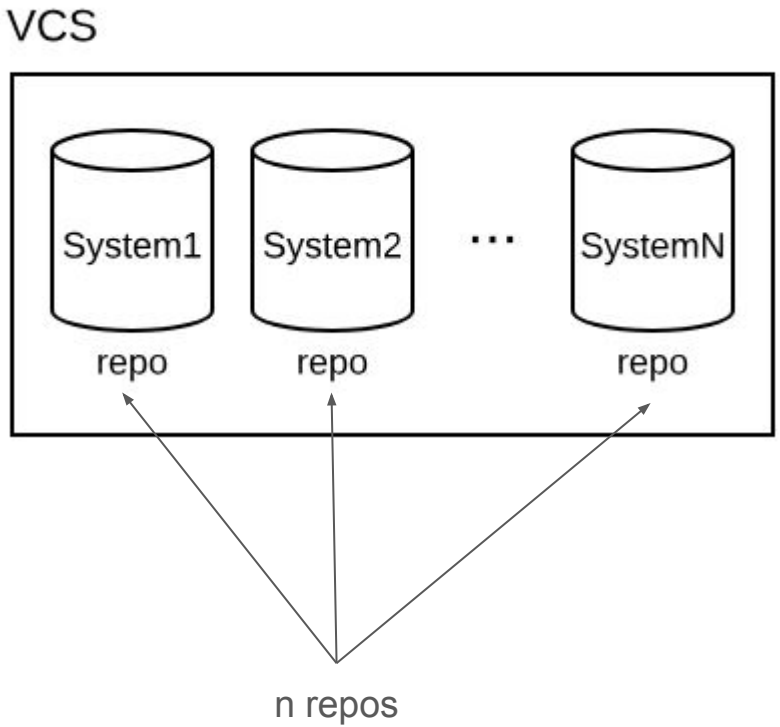


Advantages of DVCS

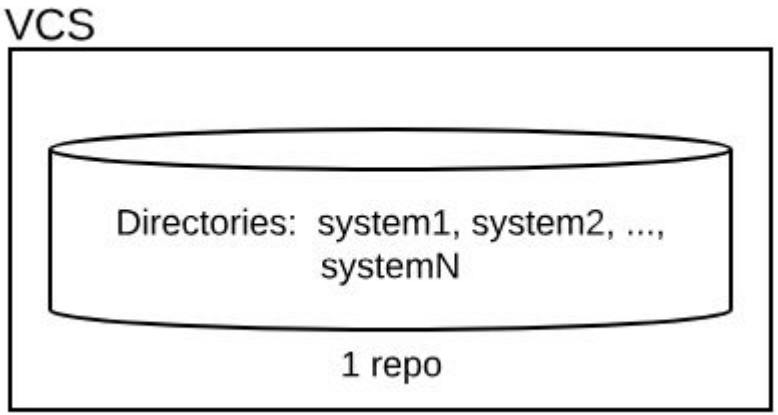
- Commits are faster; devs can make commits more often
- There is a local VCS; thus, devs can work offline
- Supports alternative architectures: P2P, hierarchical, etc

Multirepos vs monorepo

Multirepo (more commom)



Monorepo (less common; bigtechs)



Example: GitHub

Multirepos:

- aserg-ufmg/system1
- aserg-ufmg/system2
- aserg-ufmg/system3

Monorepo:

- aserg-ufmg/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 mainly used by large tech companies

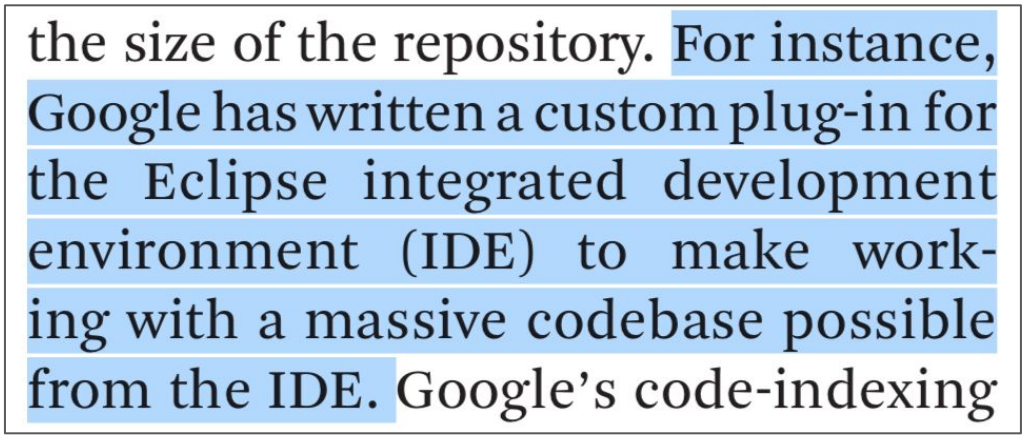
https://research.google/pubs/pub45424/

Advantages of Monorepos

- A single source of truth
- Promote visibility and code reuse
- The same version of a library is used in all systems
- Changes are always atomic (1 commit can change n systems)
- Facilitate large-scale refactorings

Disadvantage of Monorepos

- Require custom tools. Example: online IDEs



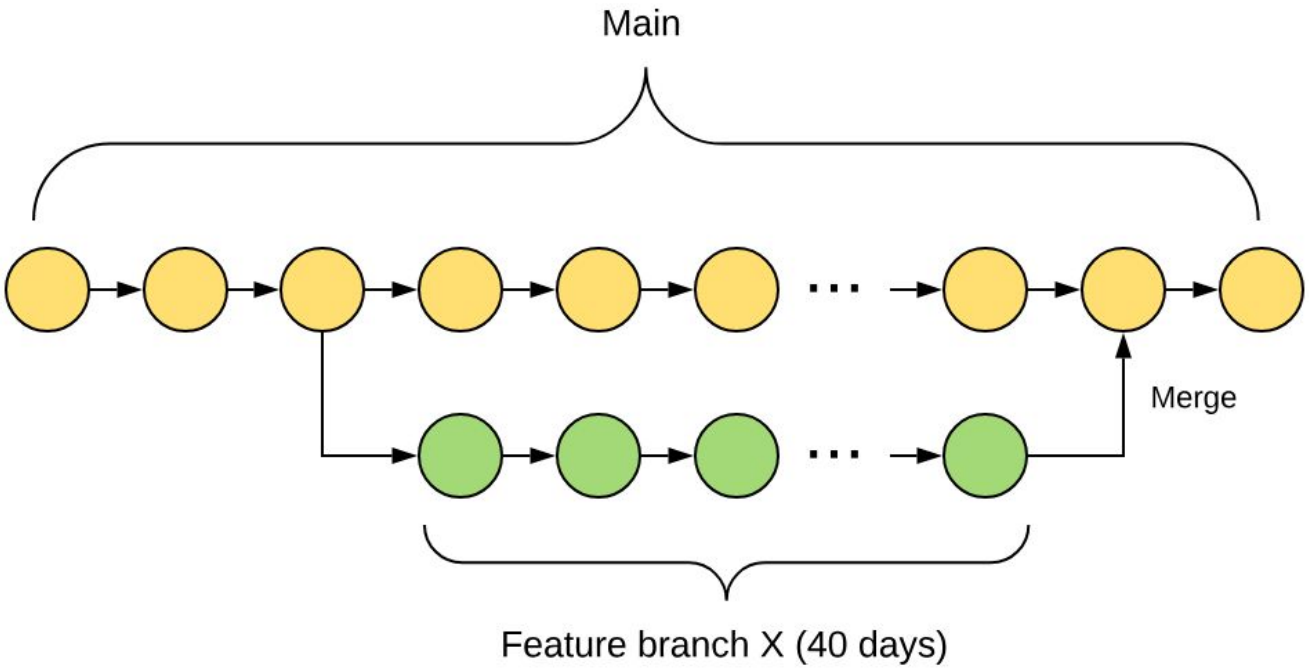
the size of the repository. For instance, Google has written a custom plug-in for the Eclipse integrated development environment (IDE) to make working with a massive codebase possible from the IDE. Google's code-indexing

Details about git in the appendix

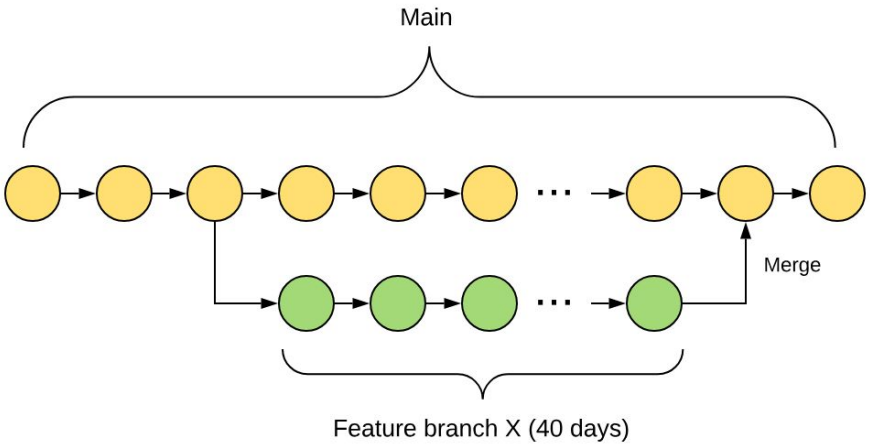
https://softengbook.org/chapterAp

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, and instead, tackle it every day

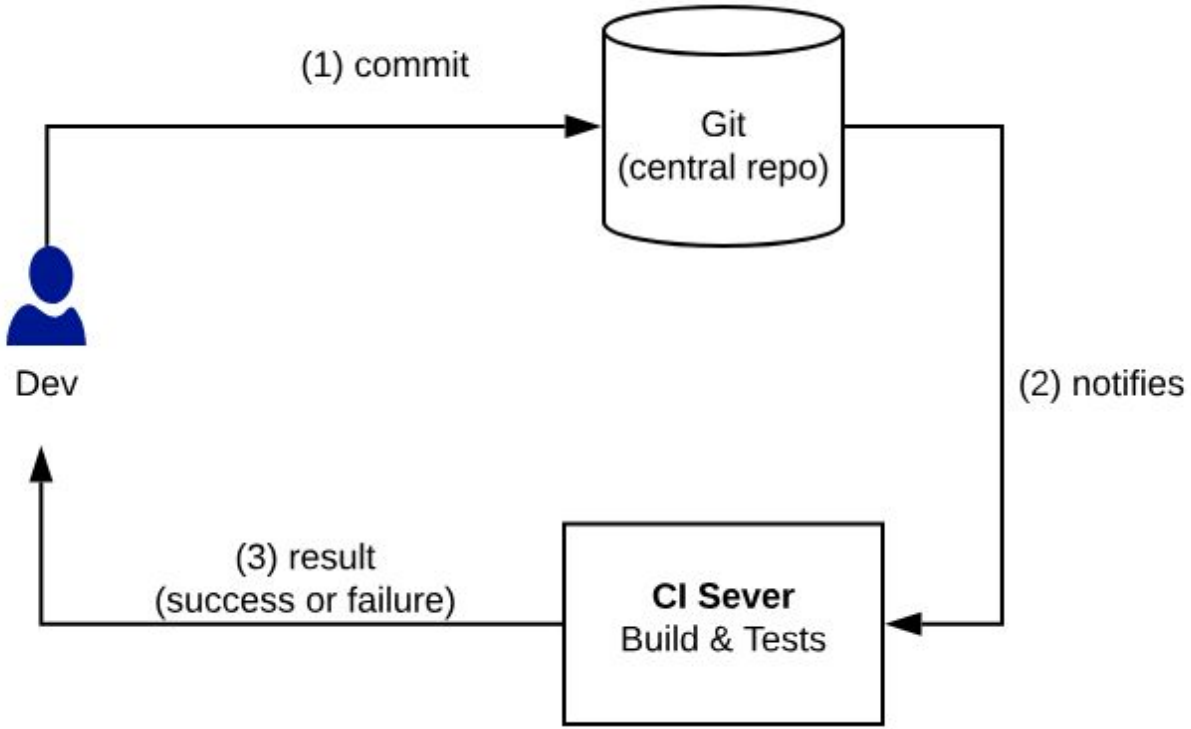
Continuous Integration (CI)

- Proposed by XP
- As the name suggests, CI recommends integrating code frequently
- But how often?
  - There is no consensus, but most authors recommend at least once a day

Best practices when using CI

- Automated builds
- Automated tests
- Pair programming

CI Servers

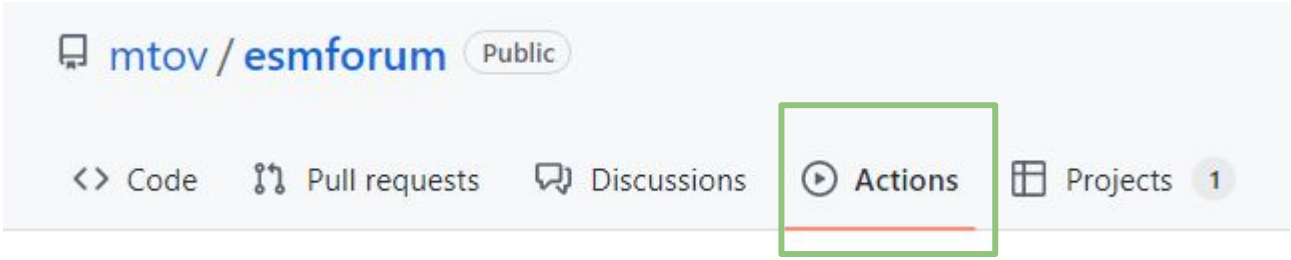

(1) commit
Git
(central repo)
Dev
(2) notifies
(3) result
(success or failure)
CI Sever
Build & Tests

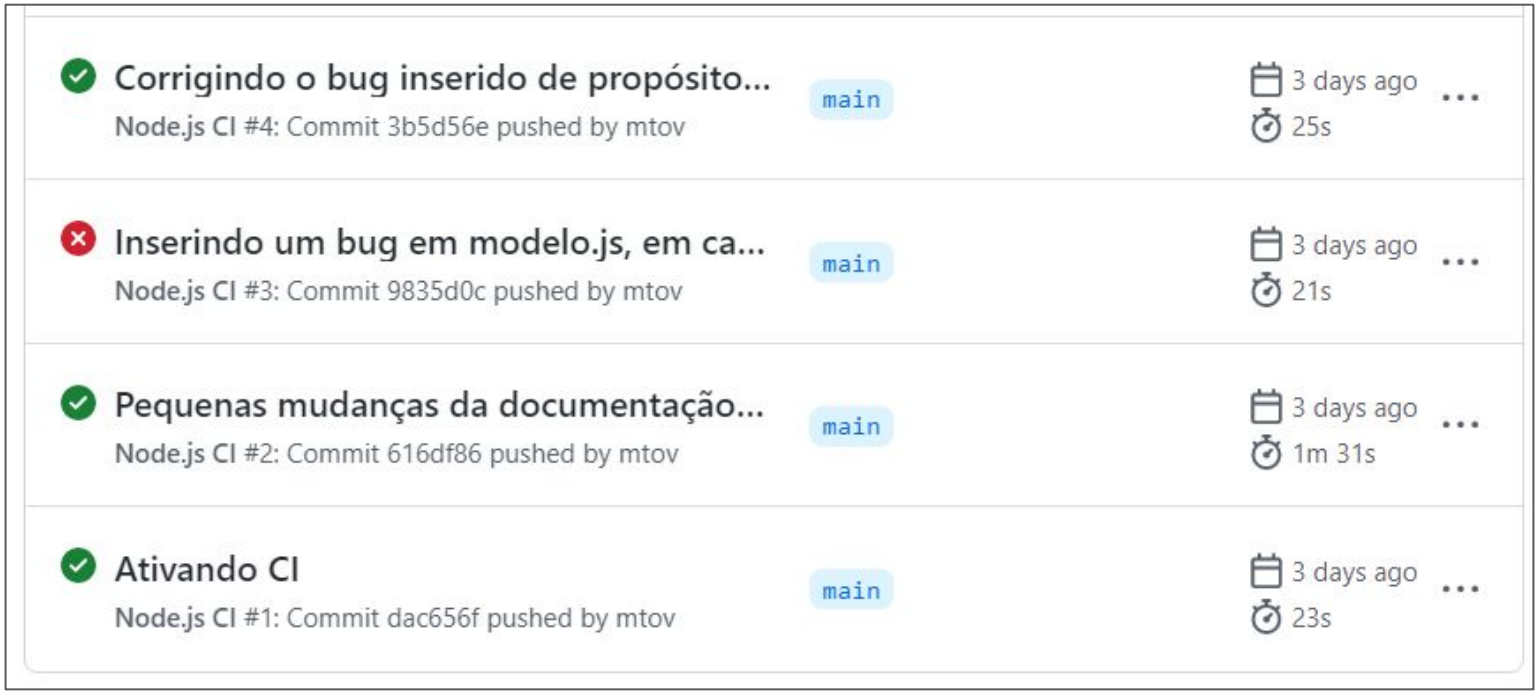
Example: GitHub Actions Configuration File

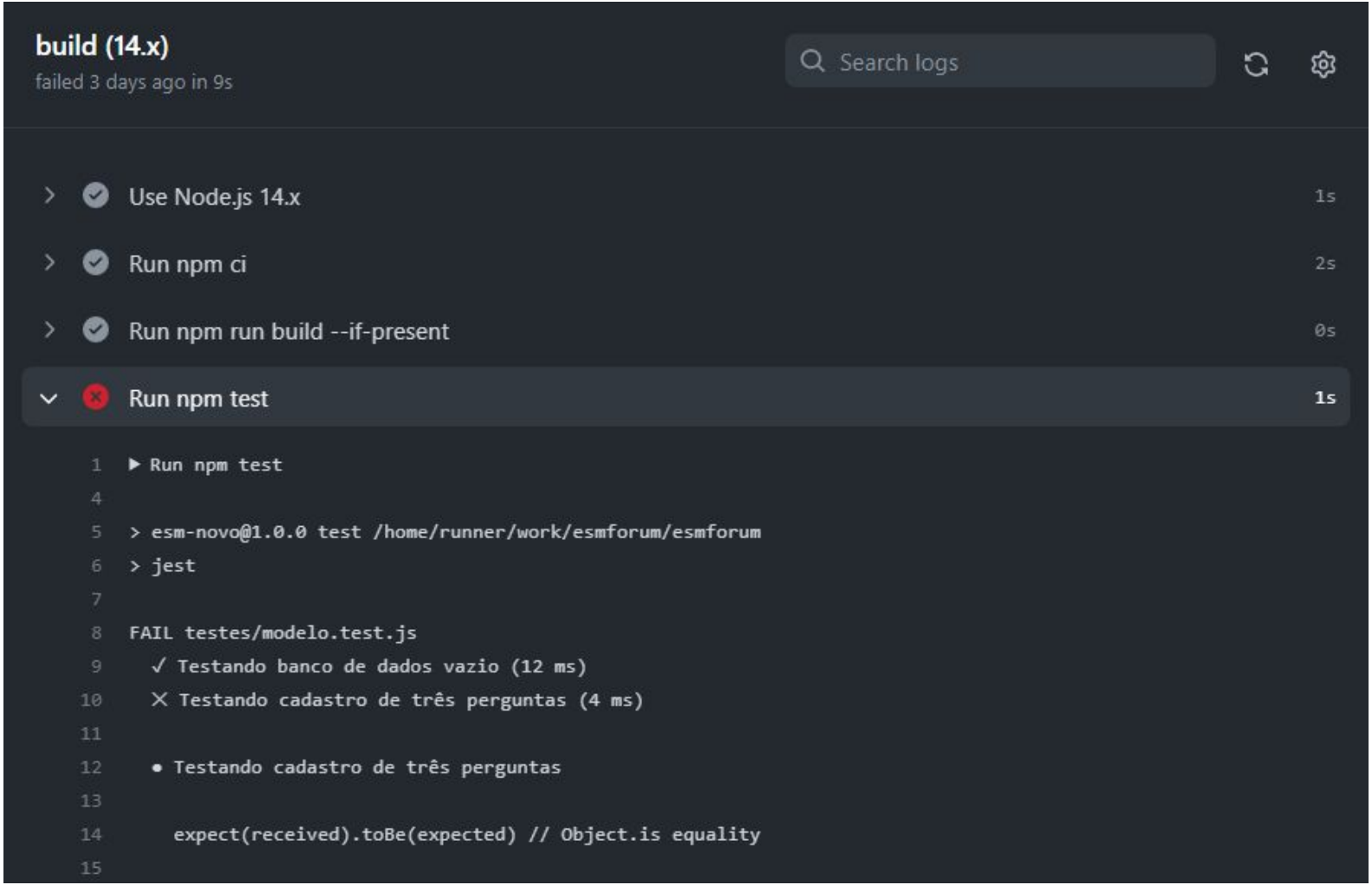
```
on:
 push:
   branches: [ "main" ]
 pull_request:
   branches: [ "main" ]

jobs:
 build:
   runs-on: ubuntu-latest
   strategy:
     matrix:
       node-version: [14.x, 16.x, 18.x]
```

(cont.)

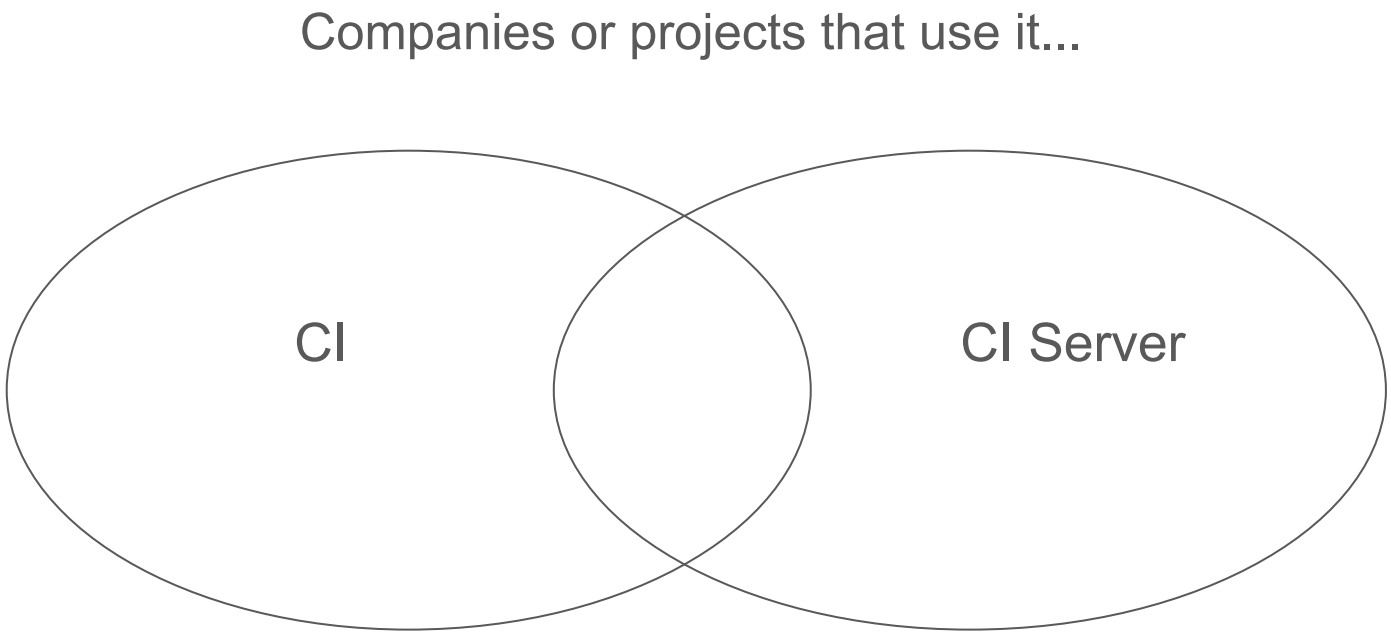
```
steps:
  - uses: actions/checkout@v3
  - name: Use Node.js ${{ matrix.node-version }}
    uses: actions/setup-node@v3
    with:
      node-version: ${{ matrix.node-version }}
      cache: 'npm'
  - run: npm ci
  - run: npm run build --if-present
  - run: npm test
```


mtov / esmforum
Public
Code
Pull requests
Discussions
Actions
Projects
1


Corrigindo o bug inserido de propósito...
Node.js CI #4: Commit 3b5d56e pushed by mtov
main
3 days ago
25s
Inserindo um bug em modelo.js, em ca...
Node.js CI #3: Commit 9835d0c pushed by mtov
main
3 days ago
21s
Pequenas mudanças da documentação...
Node.js CI #2: Commit 616df86 pushed by mtov
main
3 days ago
1m 31s
Ativando CI
Node.js CI #1: Commit dac656f pushed by mtov
main
3 days ago
23s


build (14.x)
failed 3 days ago in 9s
Search logs
Use Node.js 14.x
1s
Run npm ci
2s
Run npm run build --if-present
0s
Run npm test
1s
1 ▶ Run npm test
4
5 > esm-novo@1.0.0 test /home/runner/work/esmforum/esmforum
6 > jest
7
8 FAIL testes/modelo.test.js
9 ✓ Testando banco de dados vazio (12 ms)
10 ✕ Testando cadastro de três perguntas (4 ms)
11
12 • Testando cadastro de três perguntas
13
14 expect(received).toBe(expected) // Object.is equality
15

Important: do not confuse adopting CI with only using a CI server



Branching Strategies

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

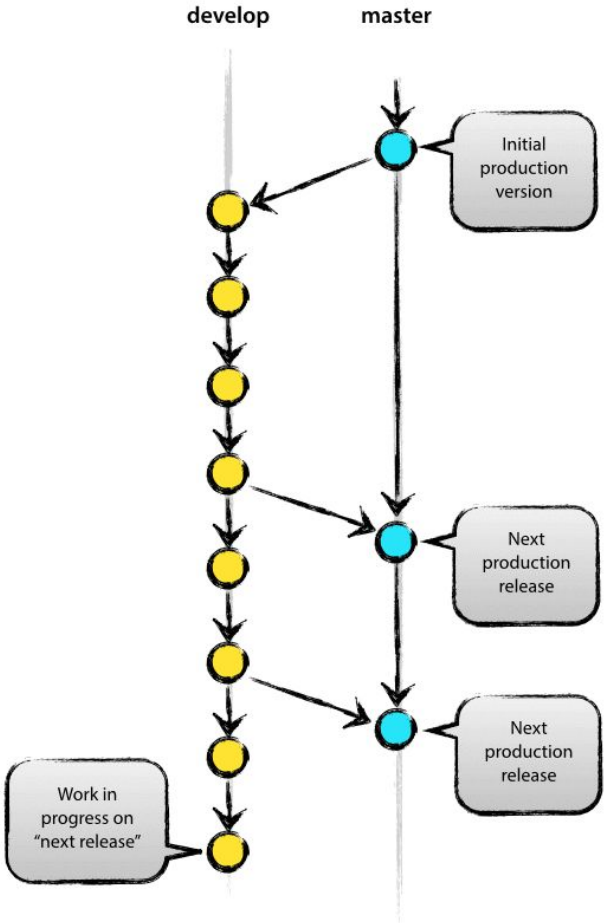
Branching Strategies

- How to organize and manage branches in a VCS
- Why, when, and how to create, merge, and delete branches
- Main strategies:
  - Git-flow
  - GitHubFlow
  - Trunk-based Development

Git-Flow

Git-flow

- Widely-used branch strategy
- Two permanent branches:
  - Main
  - Develop



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

Permanent Branches

- Main: code that is ready for production; also called master or trunk
- Develop: features that are implemented, but haven't passed a final test, for example, by the QA team

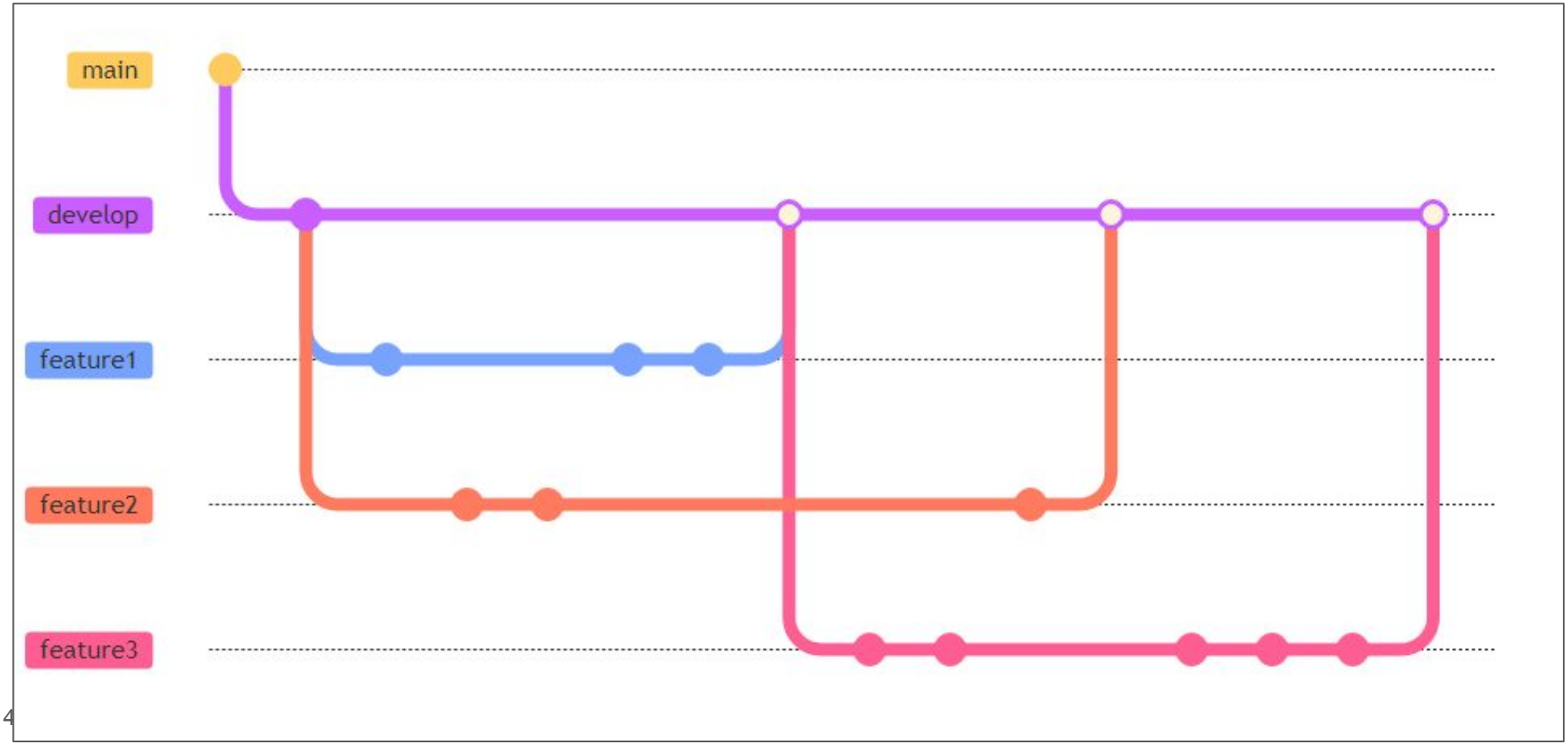
Temporary Branches

- Feature branches
- Release branches
- Hotfix branches

Feature Branches

- Branches to implement a new feature
  - Origin: develop
  - Destination: merged back into develop
- Often, only exist in the dev's local repository

Feature Branches

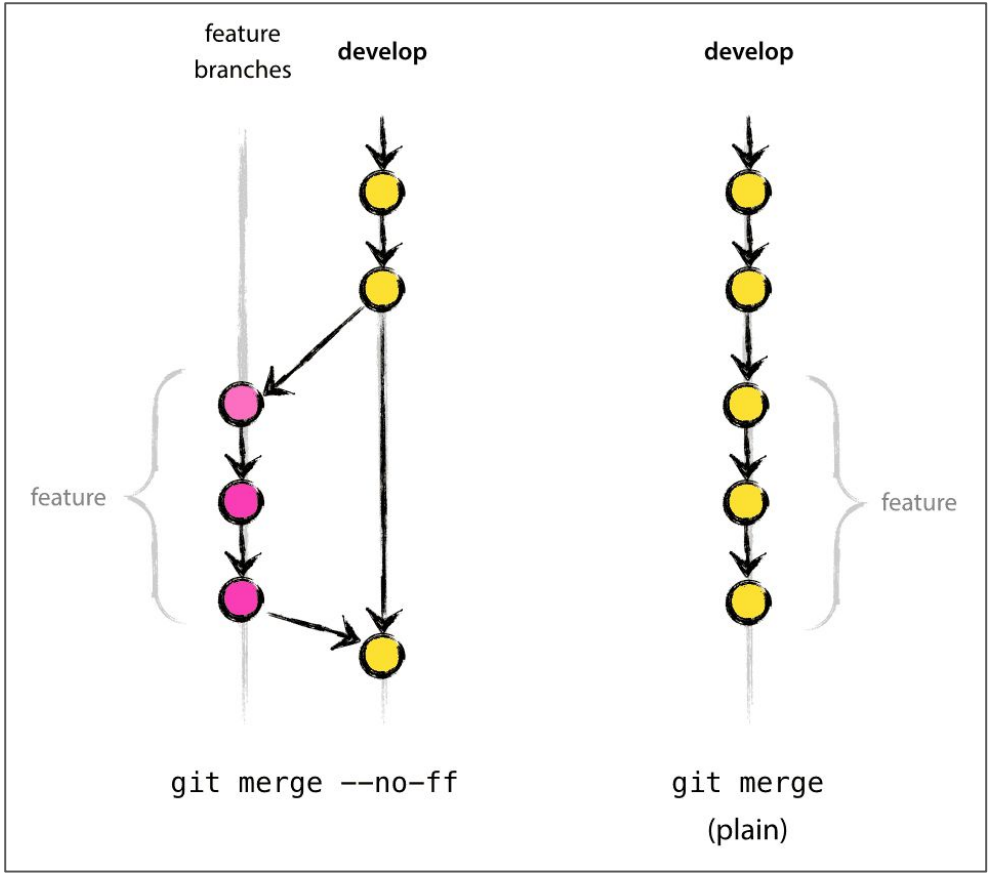


Commands to create feature branches

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

[commits to implement feature]

git checkout develop                   % switches 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             % removes feature branch

git push origin develop                % pushes develop to remote repo
```

git merge: with and without fast-forward

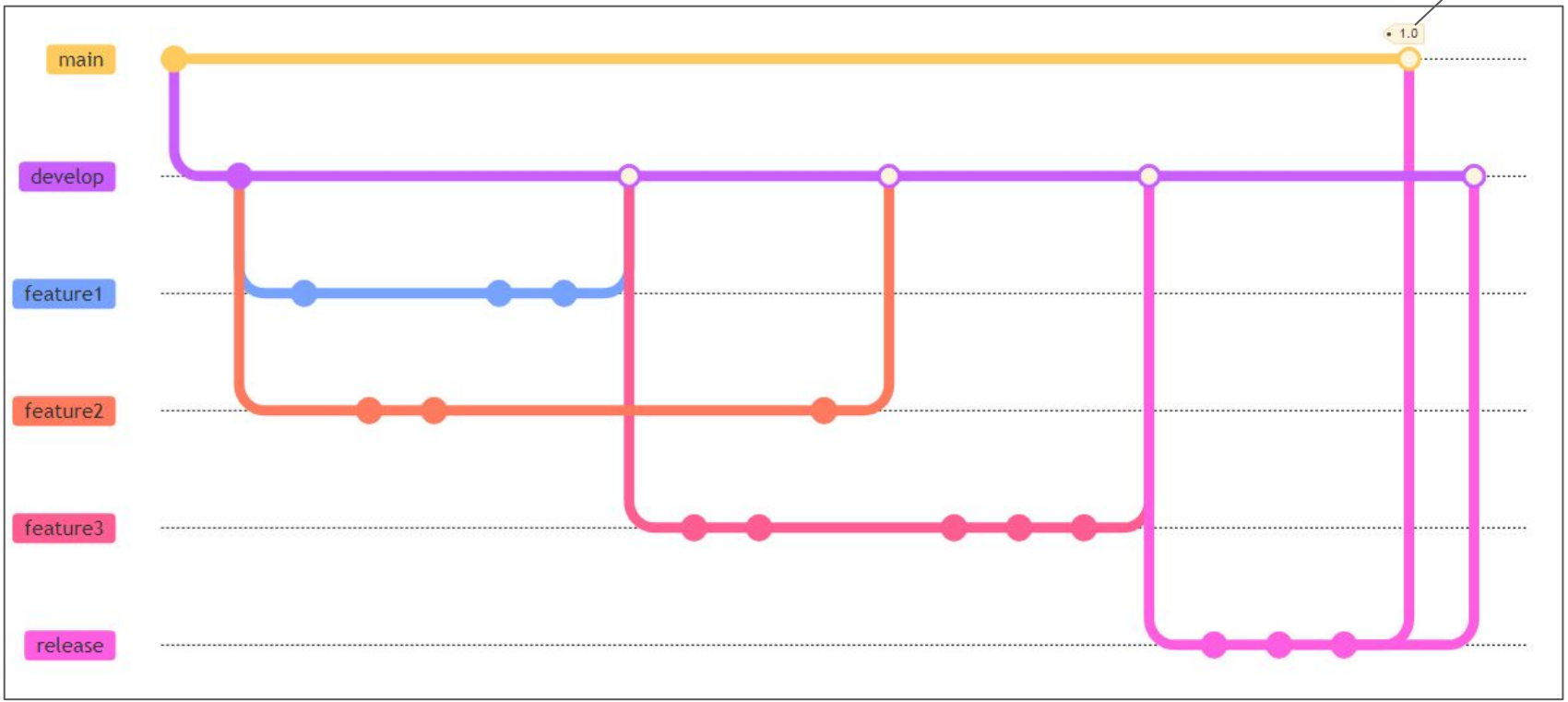
’



Release Branches

- Used to prepare a new release to be approved by customers
- Origin: develop
- Destination:
  - merge into main (with the new release tag)
  - also merge into develop (with bug fixes)

Release Branch (last branch in the figure)



Commands to create 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 (github)
git push origin main                  % pushes main to remote repo (github)
```

Hotfix Branches

- Branches to fix critical bugs detected in production
- Origin: main (via the tag where the bug was reported)
- Destination:
  - merge into master (with new tag)
  - also merge into develop

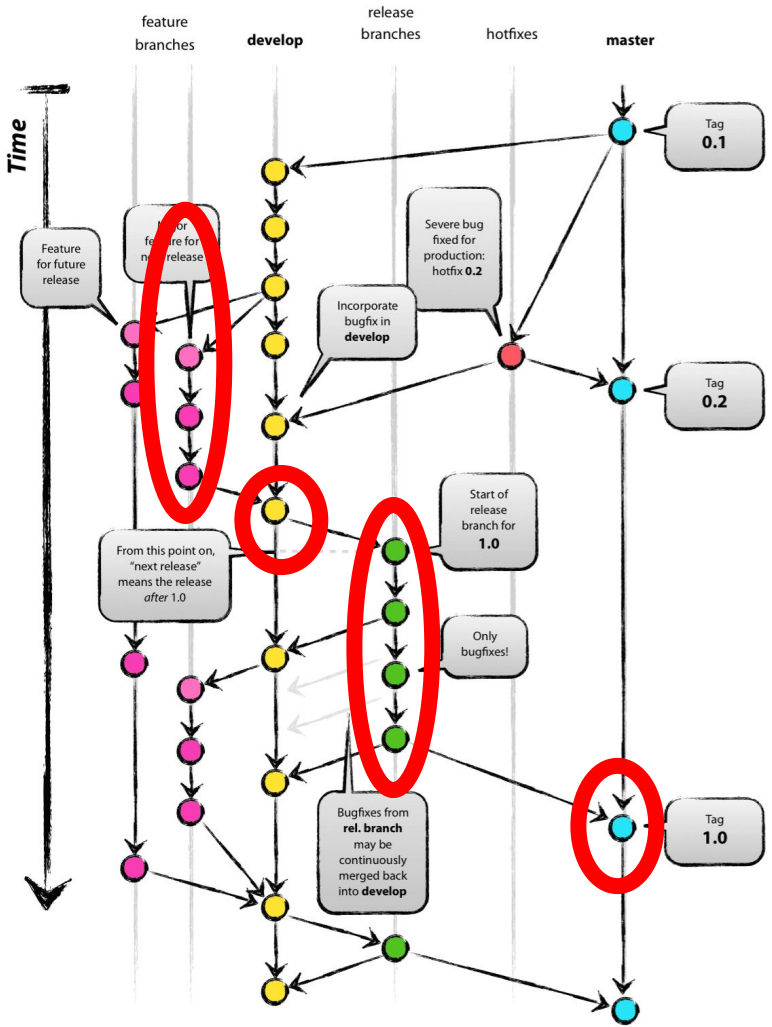
Hotfix Branches (last branch in the figure)

Commands to create hotfix branches

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

[commits do hotfix]

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 (github)
git push origin master              % pushes main to remote repo (github)
```

Git-flow: Summary

Feature ⇒ Develop ⇒ Release ⇒
⇒ Main



Git-flow: Usage and disadvantages

- Recommended when:
  - Several customers with different versions
  - Manual testing and QA teams
  - Releases require customer approval
- Disadvantages:
  - Tendency to have long-lived branches and more conflicts
  - And longer customer feedback cycles

GitHub Flow

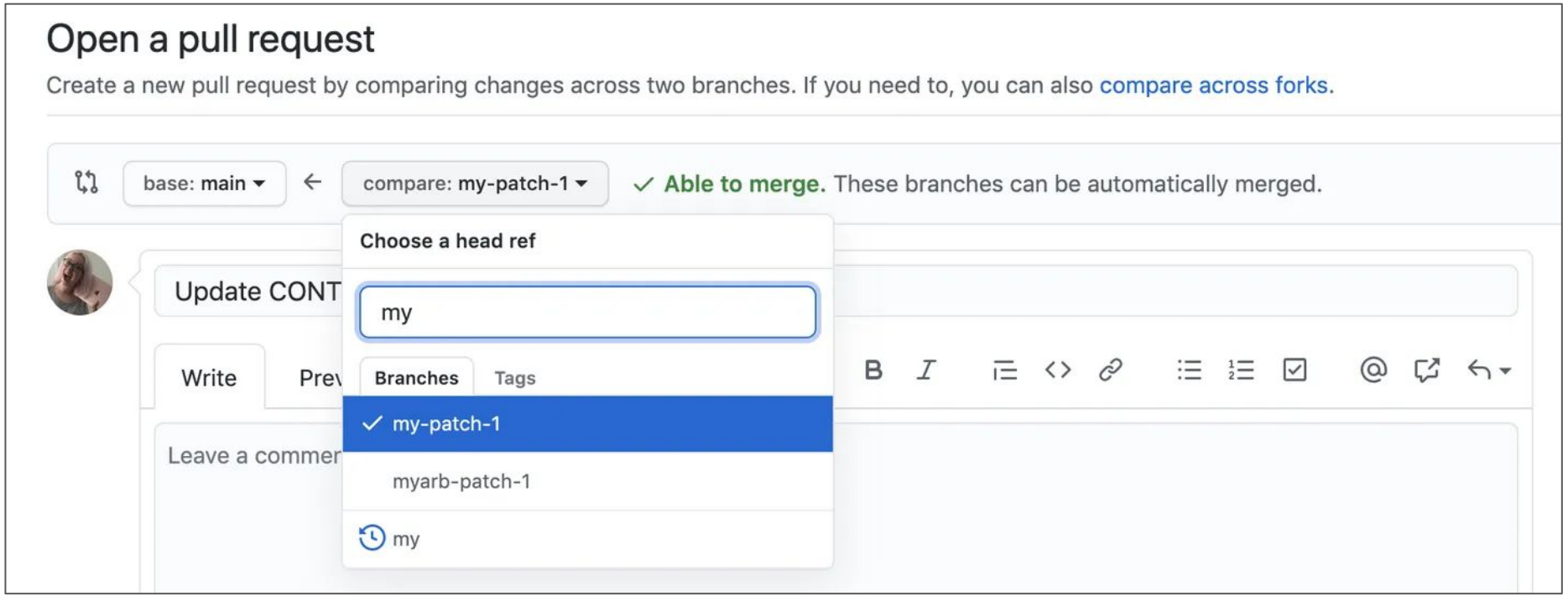
GitHub Flow

- Common flow when using GitHub
- Simplified Git-Flow:
  - No develop, release, and hotfix branches
  - Only feature and main branches
- But with support for Pull Requests (PR)

GitHub Flow Steps

- Dev creates a "feature branch" in their local repo
- Implements a feature
- Pushes the branch to GitHub
- Goes to GitHub and opens a Pull Request (PR)
    - PR: request for someone to review the branch
- Reviewer (other dev) reviews and merges the PR into main

Pull Request



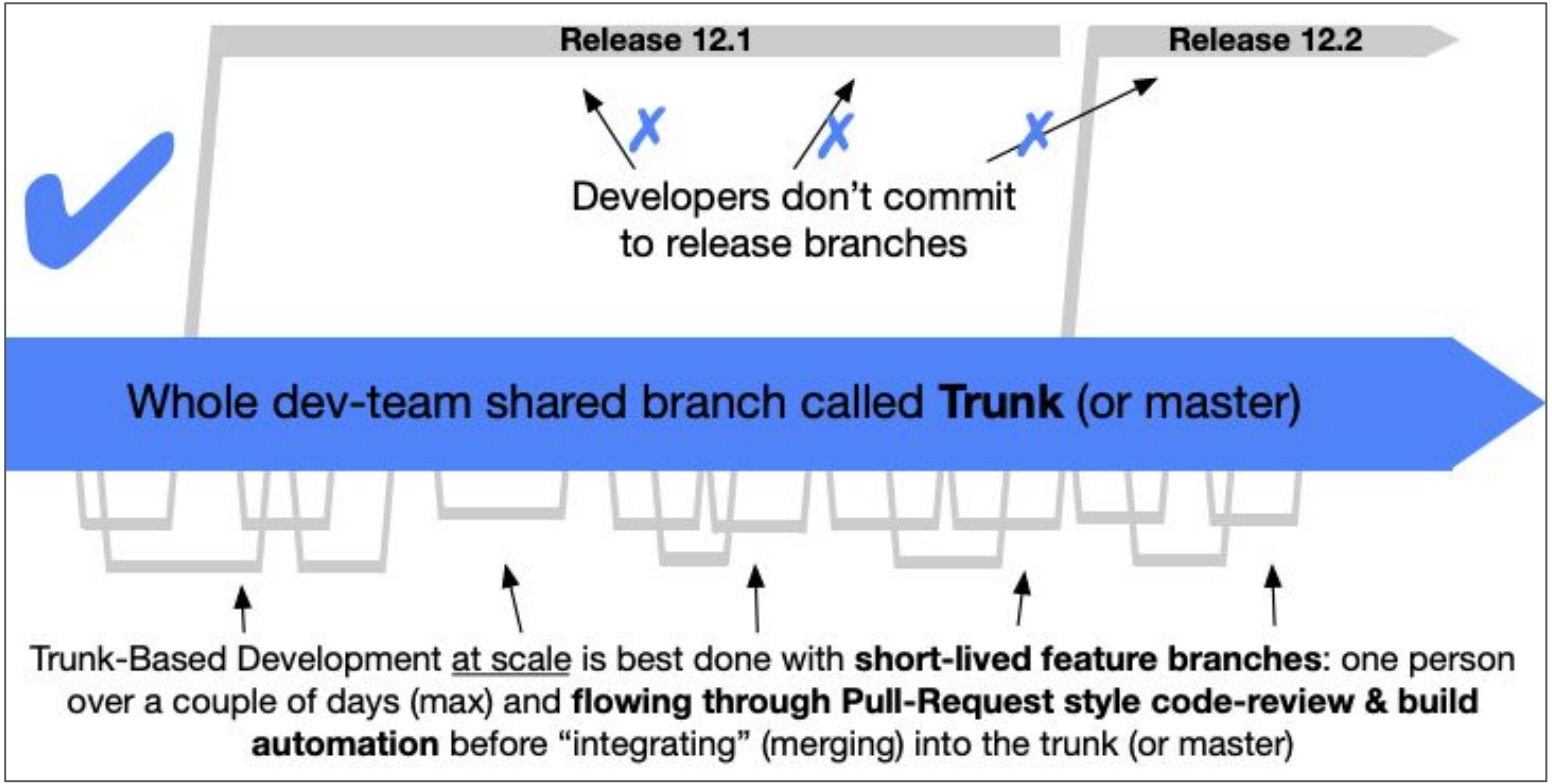
GitHub Flow: Usage and Disadvantage

- When to use:
  - Systems with only one version in production
  - Example: Web systems
- Disadvantage:
  - PRs may take a long time to be reviewed

Trunk-based Development (TBD)

Trunk-based Development (TBD)

- Since merges can cause conflicts, TBD advocates:
  - No develop branches
  - All implementation occurs directly on the main branch
- Main branch: also called trunk or master



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: experiment and feedback!

How to prevent my 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 ready: enable the flag

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

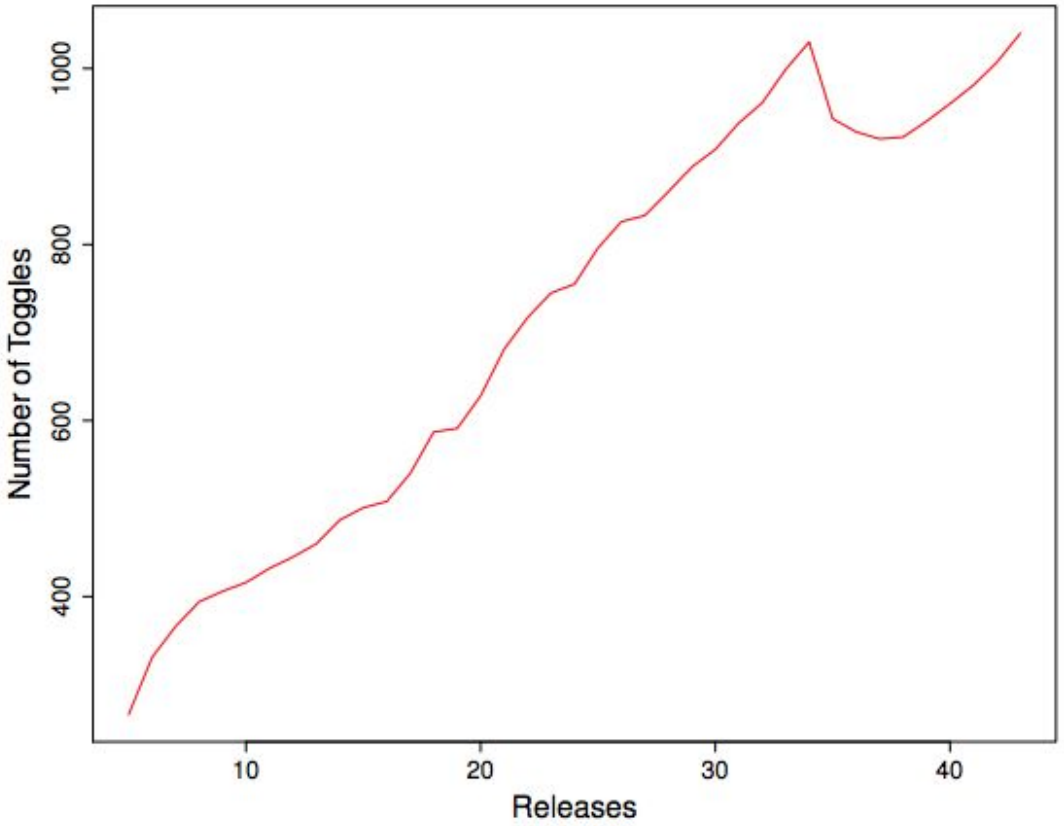


Figure 2: Number of unique toggles per release of Google Chrome.

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

Branch by Abstraction

- Technique to make changes in a system:
  - Keeping the current implementation running
  - Without creating branches
- Idea:
  - Simulate a branch in the code
  - Through abstractions and duplication of code

Example: changing the implementation of a function f

1. Rename `f` to `f_old`
2. Create the following new function (or abstraction):

```
void f() {
  f_old();        // used by the rest of the code
  // f_new();     // used during the implementation/test of the change
}
```

3. In the local repo, implement and test `f_new`, switching comments
4. When ready, delete `f_old` and `f`; and rename `f_new` to `f`

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 dev, 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

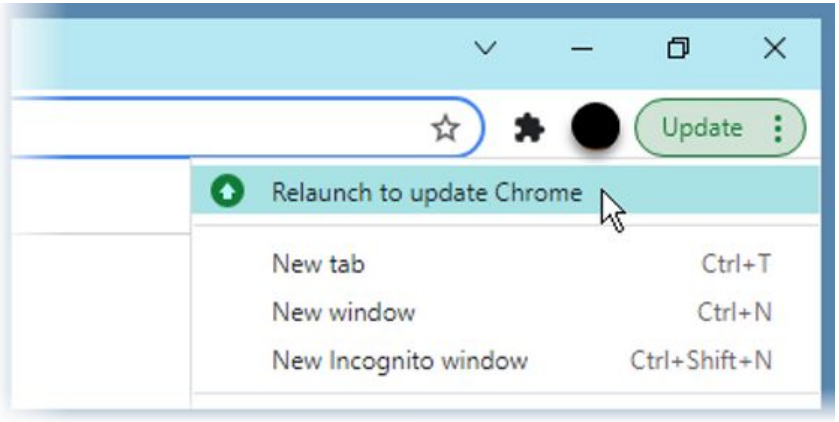
Continuous Delivery
(deployment must be
manually approved)

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? Justify.

4. In a browser like Chrome, is it better to use Continuous Delivery or Continuous Deployment?



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 is the difference 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!"
}
```

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

8. What's the difference between an A/B Test and a canary release?

In summary, feature flags are used to:

1. Control the release of untested or incompleted 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 the following table assuming a company that uses git-flow.

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

10. Assume you are responsible for implementing a change in a function `f`.

For this, you decided to use a branch by abstraction strategy. Thus, you created a copy of `f`, called `f_new`. Still assume that f calls a function `g`.

(a) If a bug is fixed in `g`, by another dev, which git command should you use to get the new version of `g`.

(b) Now suppose your change in `f` requires a change in `g` as well. What should you do in this case?

End