DevOps Fundamentals

Source Control

Agenda

Source Control

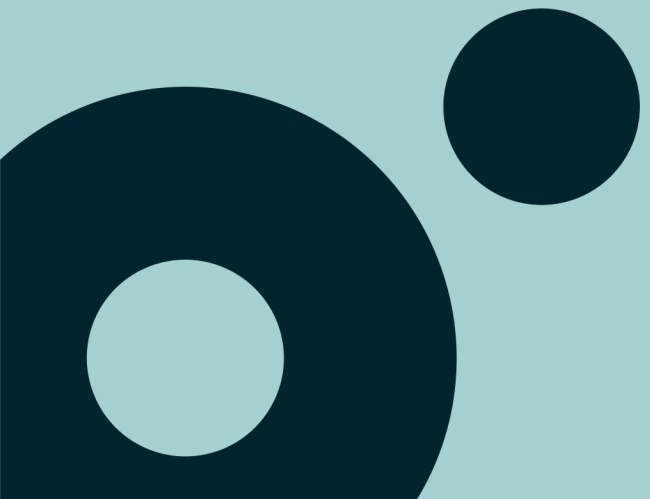
Git Overview

Pull Requests

Branching strategies

moOngy.

Source Control



Source Control

Source Control as a huge impact on DevOps

Everything as Code is a reality: Source code, pipelines, infra, docs, …

Single source of truth

To have a clear, updated and full picture of our solution we may rely on our source control

Two main types: Centralized vs Distributed

Source Control: Benefits

Full history of all the files (of all your system)

Branching

Simultaneous Development

Auditing

Allow the implementation of different automation strategies

Promote collaboration

Resiliency (remote servers)

moOngy.

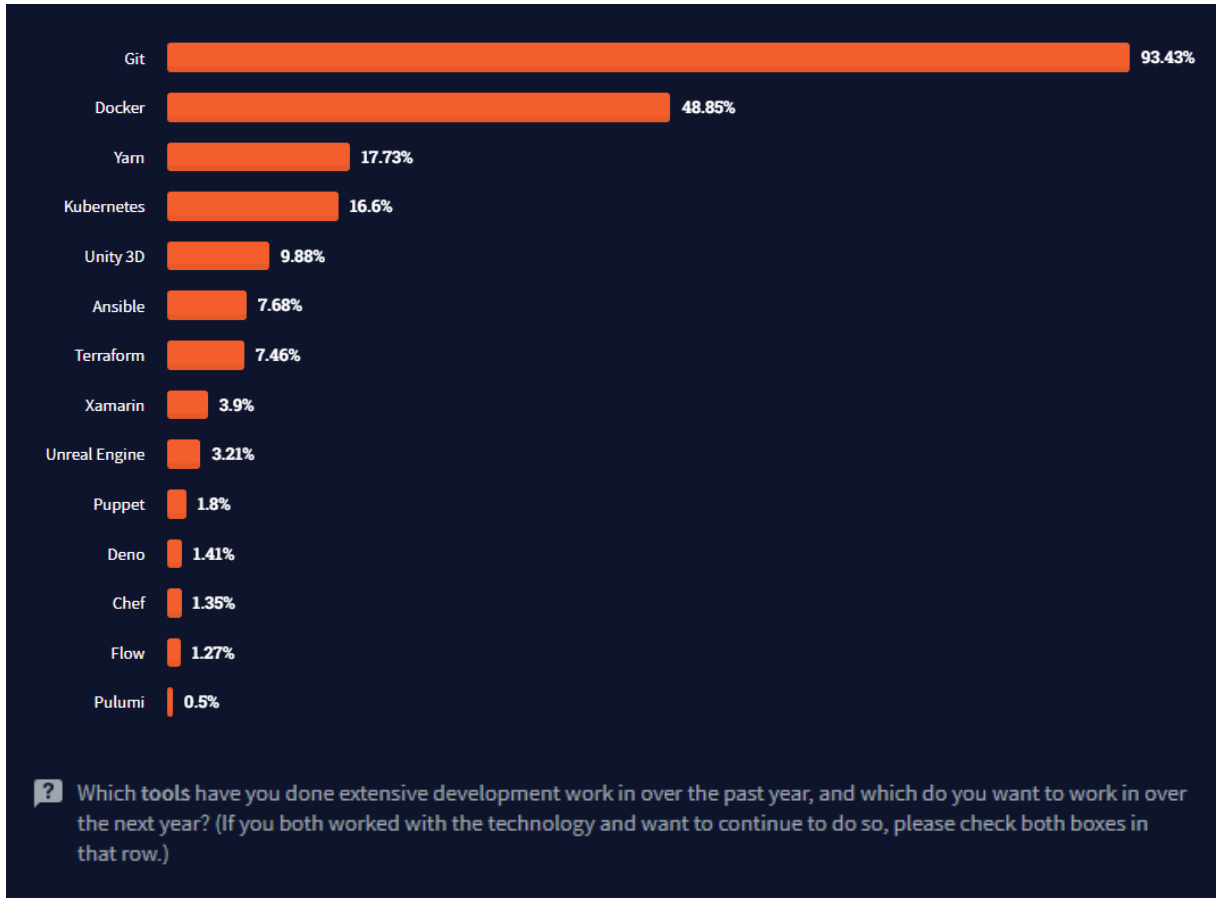
Source Control: Different Options

Centralized: Subversion, TFV

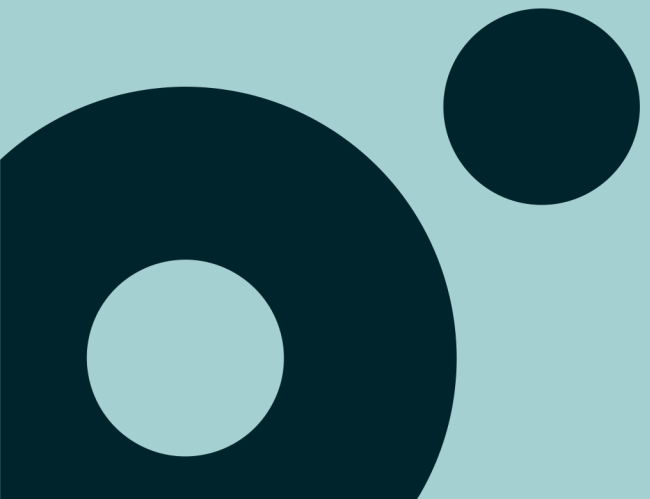
Distributed: Mercurial, Git

Git is by far the most used source control

Source: [Stack Overflow Survey 2021](), 76,253 responses



Git Overview



Git: Basics

Git manages an arbitrary structure of files and folders

Creates a '.git' folder to manage repo database and metadata

Records snapshots of file structure and contents

Treats its data like a stream of snapshots

All changes are made on local repository

When feel comfortable with local changes can propagate changes to remote repository

Changes on remote repositories are always made explicitly

Git: How to Start?

Full copy (clone) of a remote repo into a local repo

```
git clone <repo_url>
```

On clone command you receive not only the files but all metadata (folder .git)

Metadata includes all branches, file versions and additional configuration

With this you may navigate locally on repo history

This command creates an upstream between local branch and remote branch to make it easier to propagate the changes

On security side, you may run this command using HTTPS or SSH protocol and using your credentials/ssh keys

Git: Versioning

Git uses the concept of commit to create a new version of a set of files

A new “full” snapshot is created. Git don’t use deltas

This strategy allows to easily and fast compare different versions of one file

If a file is changed, a new full copy is stored

If a file didn’t change, only a pointer is represented on Git database

Each commit (version) is identified by a “commit hash” or SHA

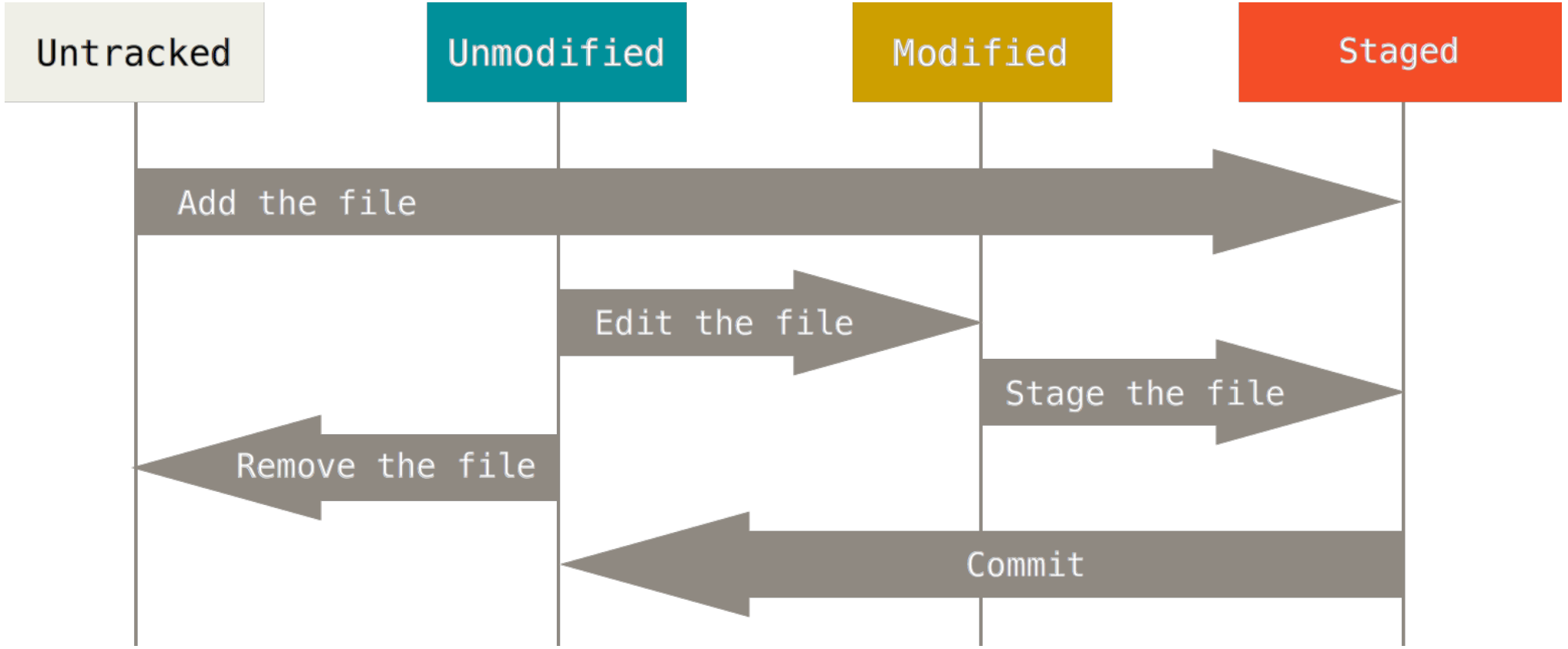
Every change is made locally only

Interaction with remote repo is done explicitly

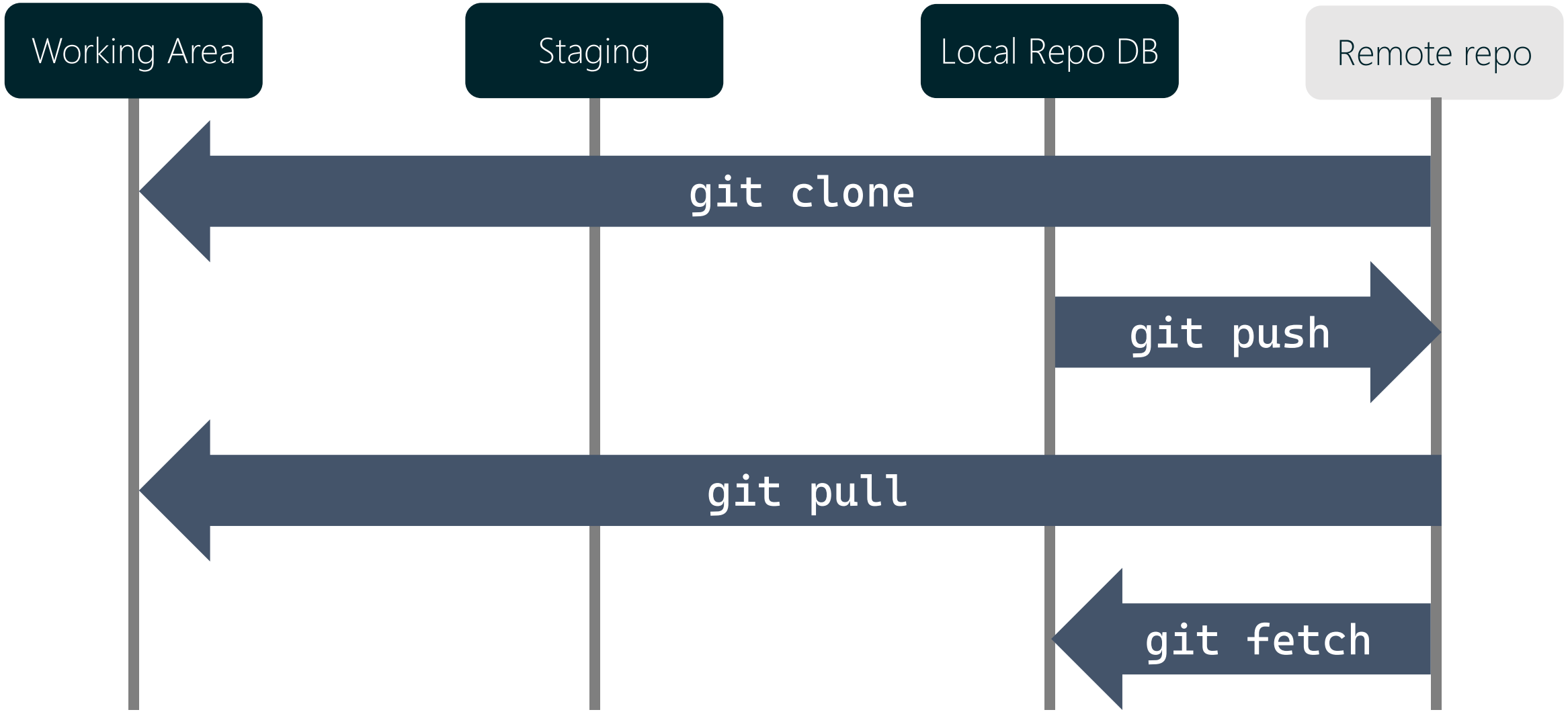
Git: States

State	Description
Committed	Safely stored in Git's database
Staged	Marked in its current version to go into your next snapshot ('commit')
Modified	Changed but not staged
Untracked	New file added to the folder but not under git control

Git: Commit


Untracked
Unmodified
Modified
Staged
Add the file
Edit the file
Stage the file
Remove the file
Commit

Git: Using remote repo



Git: Branching

One of the most powerful features of git

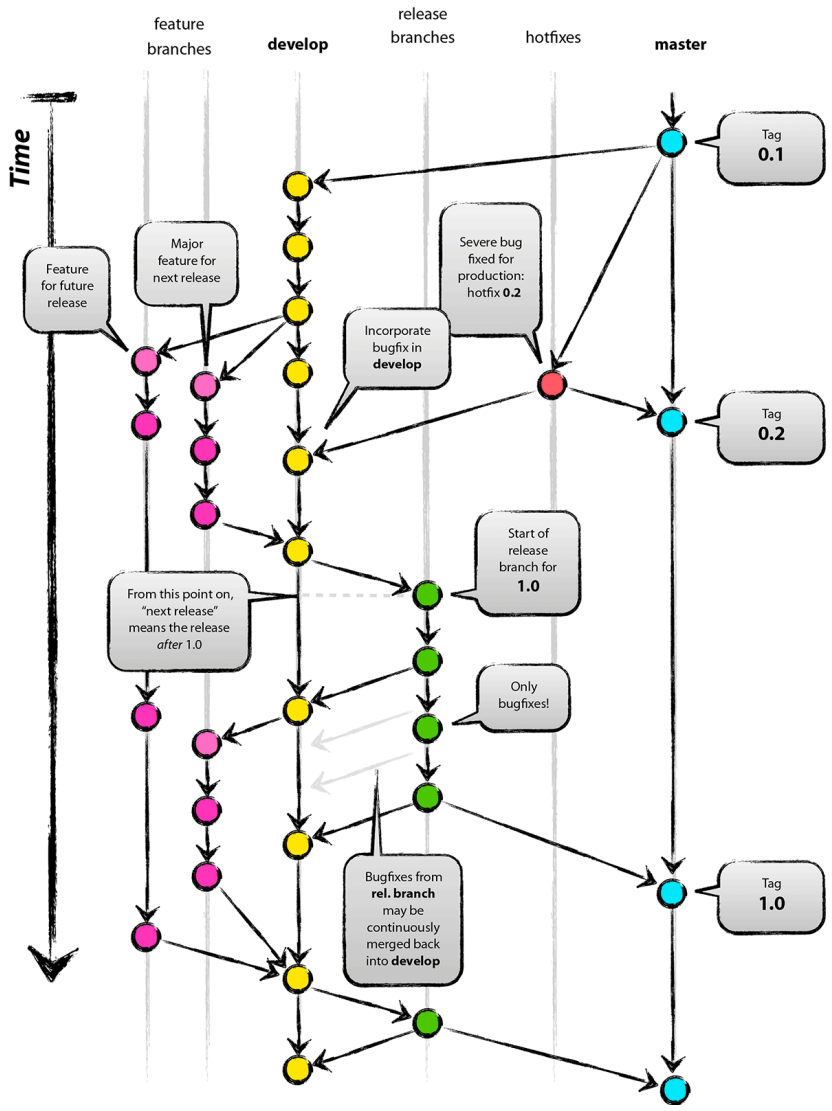
Ease and “cheap” to create new branches

A new branch is only a pointer to the last commit (no need to duplicate files)

Every commit exists inside a branch. The default branch is now called “main”

Easily allow parallel and distributed development without causing any impact on the main codebase

Keep your branch list clean! Ease to create, easier to delete!



moOngy.

Git: Merging

When you want to add the changes made on one branch into another

Git makes this process as much automatic as possible

All non-competing changes are merged automatically

Competing changes causes merge conflicts that needs to be solved manually

Merge conflicts only occurs when same file is changed on same lines in both branches

Merge can be done in 3 ways:

- Pull from remote repo (already seen)
- Explicit merge
- Pul request (next chapter)

Git: Explicit Merge

Using git commands to initiate merge process

Needs to be carefully done because keeping git history clean is important to help navigate on its log

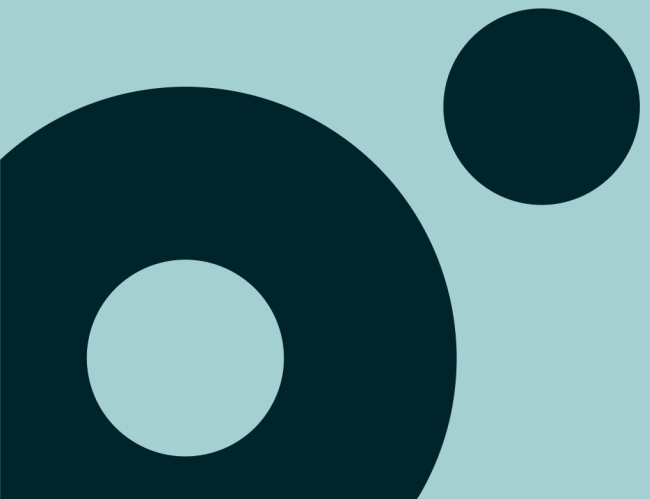
2 types of merging

- Fast forward
- Merge Commit

When using merge commit, additional strategies can be done to keep history clean like doing a squash commit (you will see it on pull requests)

Demo: Using Repo

Branching Strategies



Branching Strategies

Branching strategy means the way you handle and manage branches on remote repo

This definition is crucial to have a clean and stable remote repo

Needs to be explained and accepted by all team members in order to make it happen

Have a crucial role on CI/CD automation processes

On local repos, each one may have different and their own strategy but need to follow remote strategy every time wants to propagate (push) its changes

Trunk Based Development

Use a trunk (main) branch where you have always a buildable and deployable version

Every new development starts from this branch with a creation of a new branch

At some point in time, several branches may exist with different developments

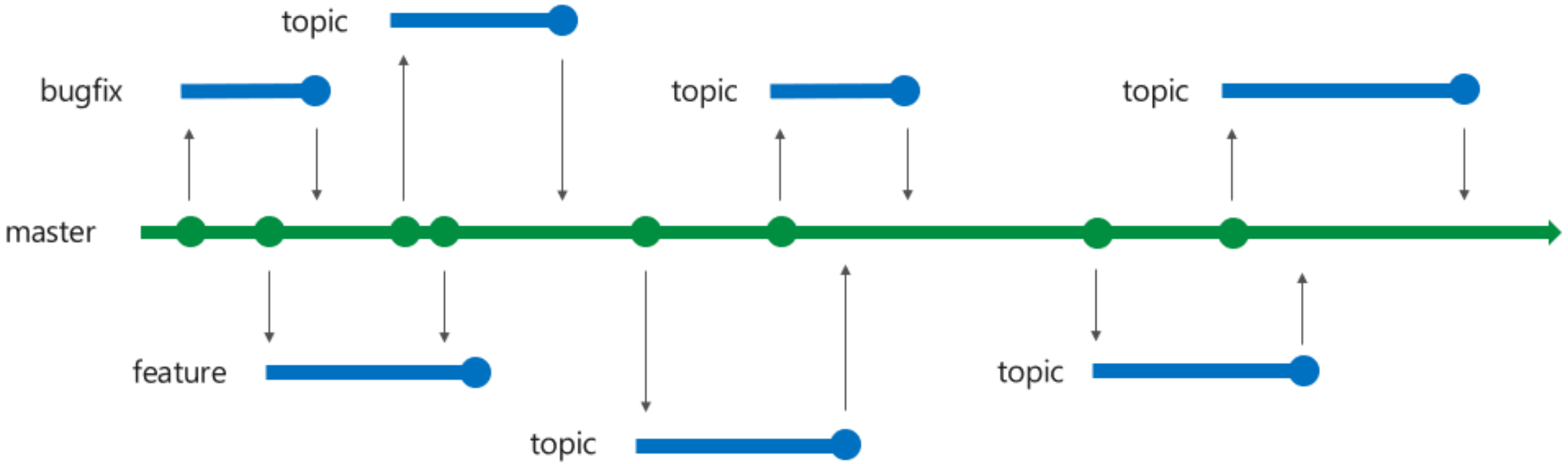
When reaching a stable development, a PR must be open to start merging process

You must never change directly your trunk branch

New branches may follow a topic (short-lived branches) or feature (long-lived branches) strategy

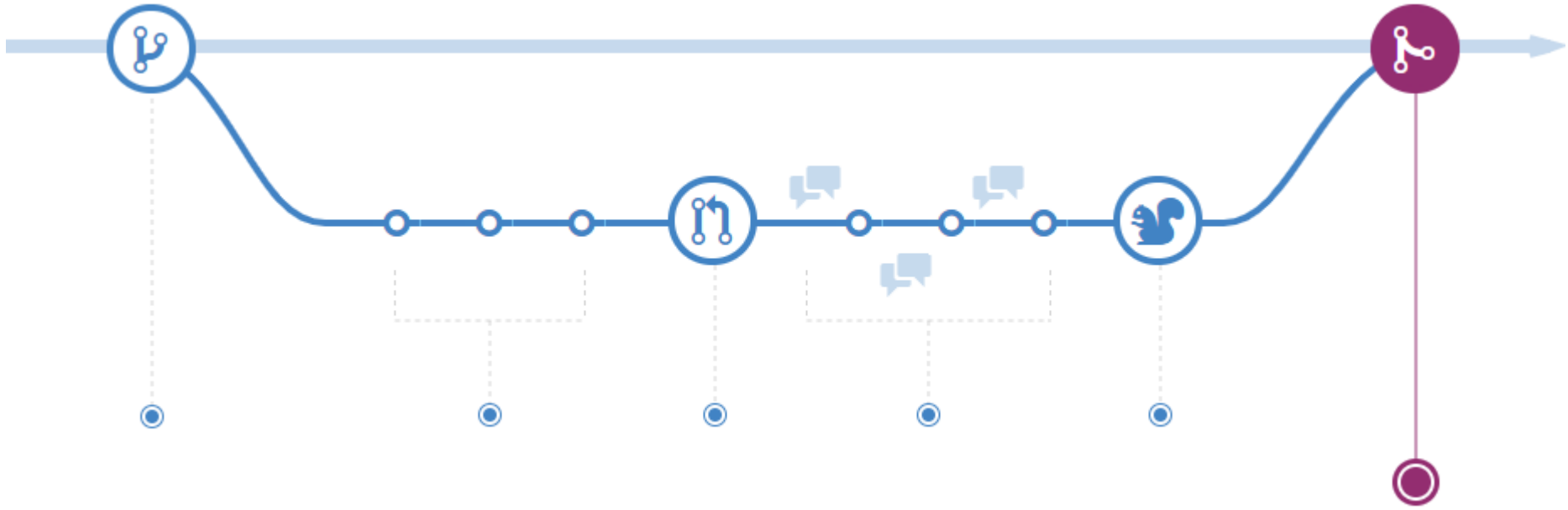
Trunk Based Development

Trunk Based Development

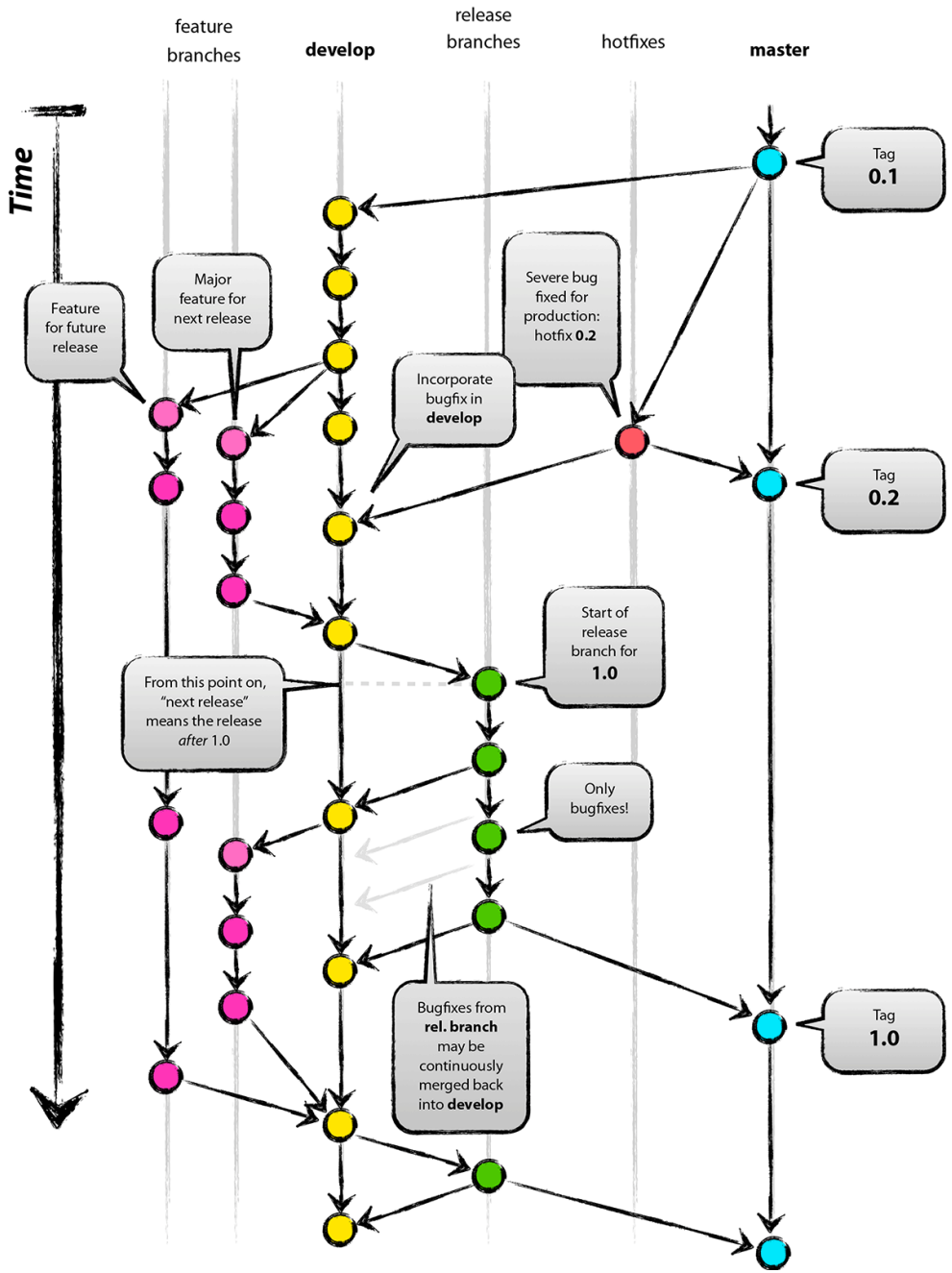


GitHub Flow

Understanding the GitHub flow · GitHub Guides



GitFlow



[GitFlow explained]

moOngy.

Branching Strategies: Best Practices

Do small commits to get a better history to navigate

PRs must be part of the process

PRs must be small enough to have functional expression but not be a burden on review process

After PR is complete you may feel comfortable in deleting your branch

Not needed branch must be always deleted

Branching Strategies: Best Practices

Define a naming strategy

- Group similar branches (bugfix, topic, feature)
- Define different policies for each branch type
- May use personal branches identifying them with user identifier

Always define a flow that match your team skills, size and maturity

- Start with trunk-based development
- Keep your flow simple and add complexity only if needed
- Always develop using branches

Git: File management

Having your repo as a single source of truth is crucial to have a good file management strategy

Git is great to handle textual files and do snapshots but is really bad handling binary files

Since every commit is a snapshot and not deltas, every time a binary file change, a full copy is done

The size of your repo starts to grow a lot a make operations with remote repo to became heavy

Git: .gitignore

.gitignore file allows to define regular expression to define which files must be ignored on commit command

As best practice, first file to be created on the repo must be .gitignore file

Rules are applied from top-down file structure

Can use exceptions to commit specific files

You may have several .gitignore files on your repo. For each folder, a merge between all .gitignore files that exists starting on that folder until reaching repo base folder

Exists a lot of pre-defined .gitignore files based on framework/IDE

Visit [github/gitignore: A collection of useful .gitignore templates]

Git: .gitignore

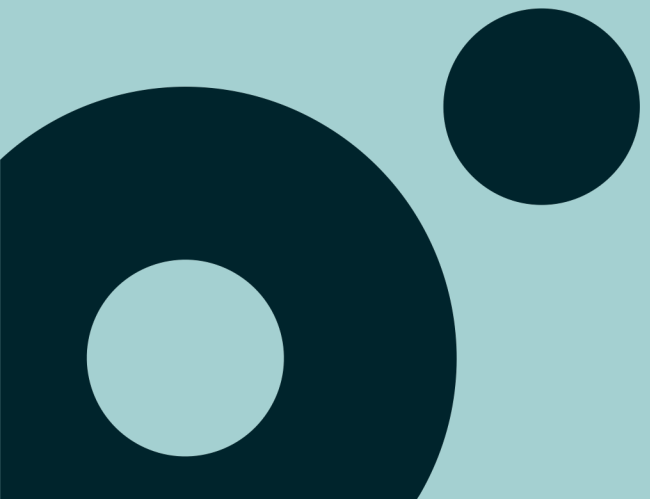
```
# excluir todos os ficheiros de extensão .tmp
*.tmp


# excluir todos os ficheiros começados por 'erro'
erro*

# excluir ficheiro chamado 'ignorar.txt'
ignorar.txt

# sobrepor configuração para incluir o ficheiro 'erro-principal.tmp'
!erro-principal.tmp
```

Pull Requests



Pull Requests

Pull requests is not a Git feature but instead a feature very popular on all Git Service Providers (GitHub, GitLab, Azure DevOps, …)

Feature created by GitHub to make easier code review within (distributed) teams

Pull requests exists to enable peer reviews bringing more quality to all produced code

Each team element can bring its own experience, knowledge and skills to all code developed within the team

At the end of a pull request you get a merge between two branches

Must not be used for one element (team lead) validation but can be used to manage which feature/improvement will be part of main version

Pull Requests

With Pull Requests you may protect your main branch and only allow updates after code being reviewed

Using PRs is a crucial practice to enable collaboration between developers inside a team

Organizations that want to go “one step further” into innersourcing, PRs are mandatory to handle collaboration between teams inside organization

Additional features you get from pull requests

- Better definition of product timeline handling which PRs you want to close
- Ask for review/approval from 1+ team elements
- Make mandatory to have a reference to a work item
- Automatic execution of a CI pipeline to test build phase before merge
- Able to have CD pipeline integrated to make functional testing

Demo: Pull Request

Lab

Lab 1: Start using your repo

Learning Objectives

- Create a new repo on GitHub
- Configure a new repo to use Pull Requests
- Create an Issue
- Create a Pull Request
- Complete a Pull Request
- Clean up your local repo to be ready for new developments

Markdown version: https://github.com/tasb/devops-with-github-training/blob/main/labs/lab01.md

HTML version: https://tasb.github.io/devops-with-github-training/labs/lab01.html

moOngy.
Minds on the move

Rua Sousa Martins, nº 10

1050-218 Lisboa | Portugal