*slides made in Canva

Studierendentage 2026

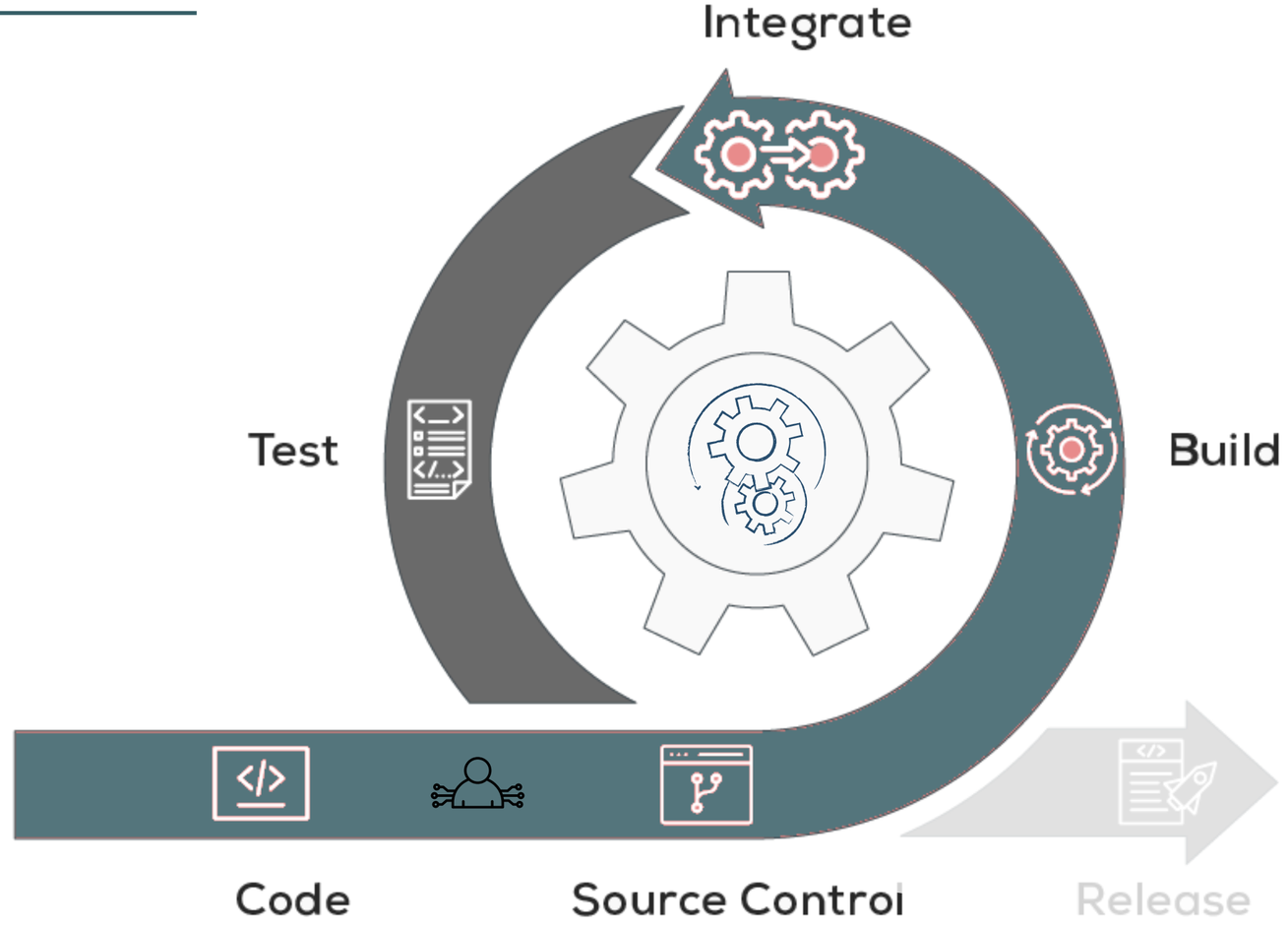
Version Control & Continuous Integration

for Analysis Preservation using Git and Docker

23rd - 27th March, 2026

Physics Institute, Heidelberg University

Tamasi Kar



About Me

Hi! I'm Tamasi Kar,
a Post Doc. in Experimental Particle Physics

- I am currently working on the Mu3e experiment with a particular focus on track reconstruction and data analysis.

Background:

- I’ve been primarily involved in the design and development of detector simulations and particle tracking algorithms for the past nine years.
- In the past, I have worked on the ATLAS experiment @ CERN and performed a physics analysis for the Future Circular Collider Study.
- Collaborative version control has been the backbone of the success of these projects.

23/03

Introduction to Docker

We’ll start with a brief overview of basic Linux commands and dive into docker containers, followed by a hands-on session.

24/03

Hands-on + Version Control

The hands-on session continues, where you’ll learn to containerize your code and run inside a container. An introduction to git will follow.

25/03

Collaborative git usage

You’ll create your own git repository on GitHub and dive deep into git by making changes to your code and recording them.

26/03

Continuous Integration

Introduction to Continuous Integration with Github followed by writing a CI pipeline code.

27/03

CI tutorial continuation

We’ll put everything we've learned so far together to produce an automated CI pipeline triggered with every commit

Preliminary Timeline

Every day: 13:30 – 16:30, 10-15 mins break after 45mins

Quick Review

Open your Terminal and try out some of these commands...

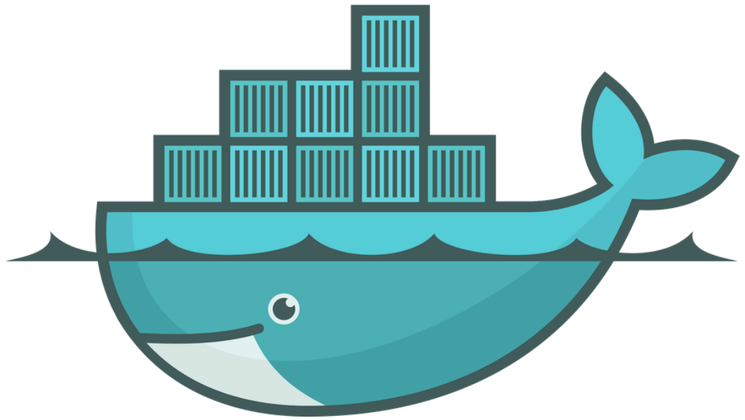
- mkdir -p docker-git-tutorial/data
- cd docker-git-tutorial

Introduction to Docker

Overview

- containers
- components
- installation
- exercises

Introduction to Docker



Official documentation: link

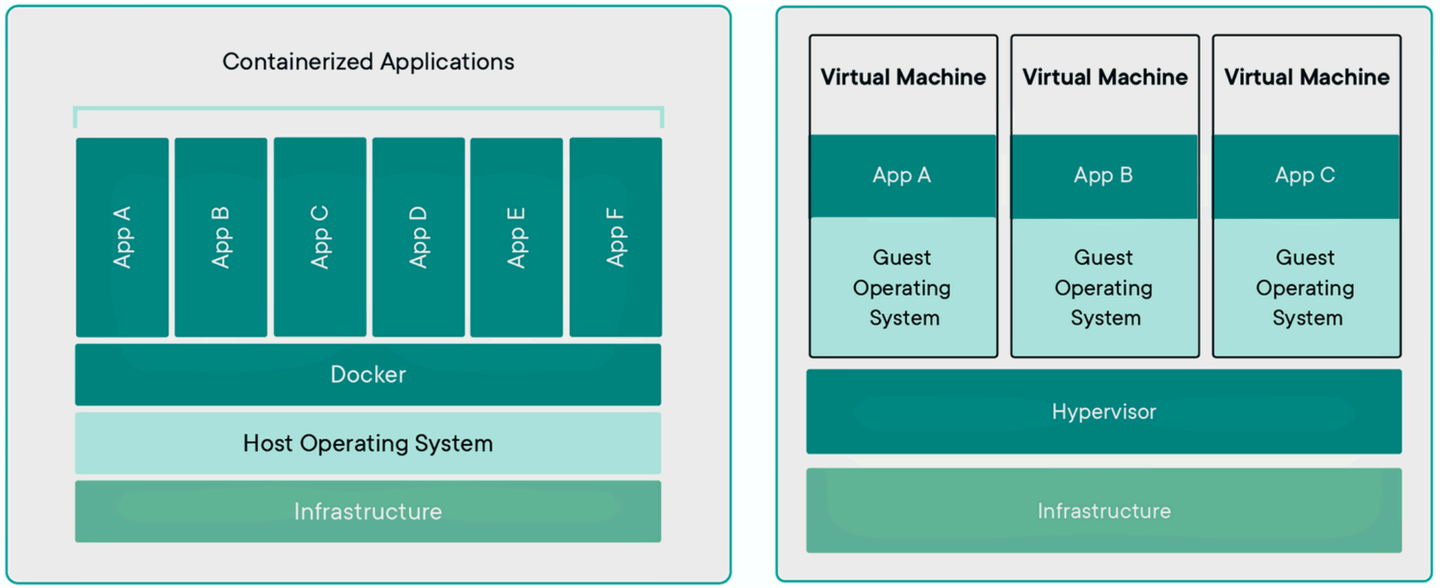
Have you run into the following problems?

- a colleague saying: “Works on my machine!”
- conflicting package versions, missing dependencies and libraries
- incompatible instructions due to a different operating system
- spent a lot of time on StackExchange and ChatGPT just to compile the code you received.

Introduction to Docker

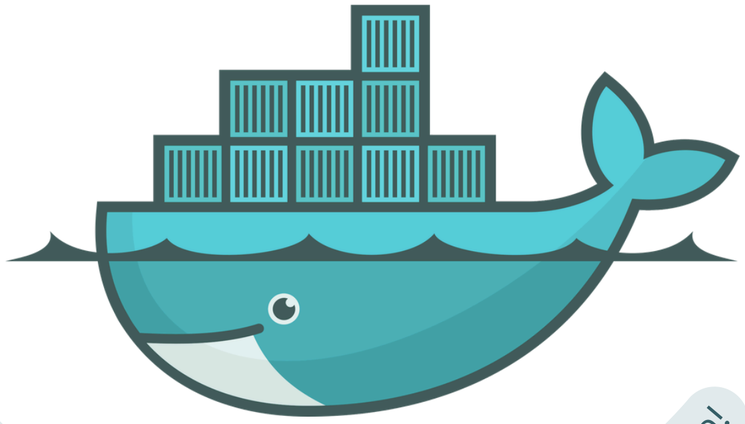
What is a Container?

An application and all its essential dependencies can be packaged into a single, isolated unit called a ***container***.



Container-based architecture vs virtual machines

You can share containers and be sure that everyone you share with gets the same container that works in the same way!



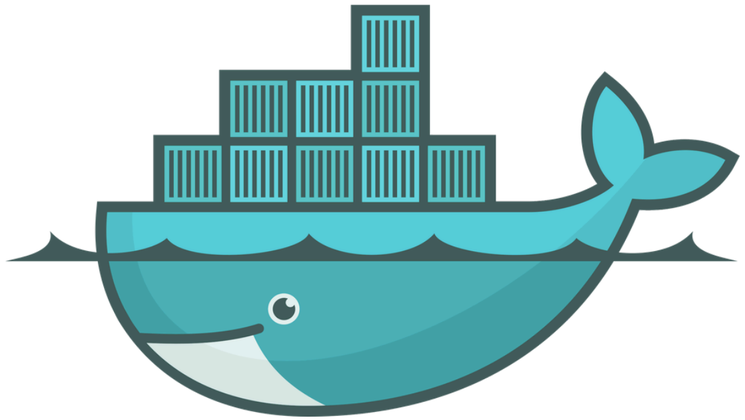
Official documentation: link

self-contained, isolated, independent, portable!

Containers:

- share the host machine's OS system kernel, and so don't require an OS per application.
- take up only as much memory as necessary, making them very lightweight and fast to spin up to run

Introduction to Docker



Official documentation: link

What is Docker?

It is an open-source ***containerization platform*** that provides the ability to package and run an application consistently on different platforms, ensuring reproducibility across systems.

- The isolation and security lets you run many containers simultaneously on a given host.
- It allows developers to work in standardized environments using local containers which provide your applications and services.

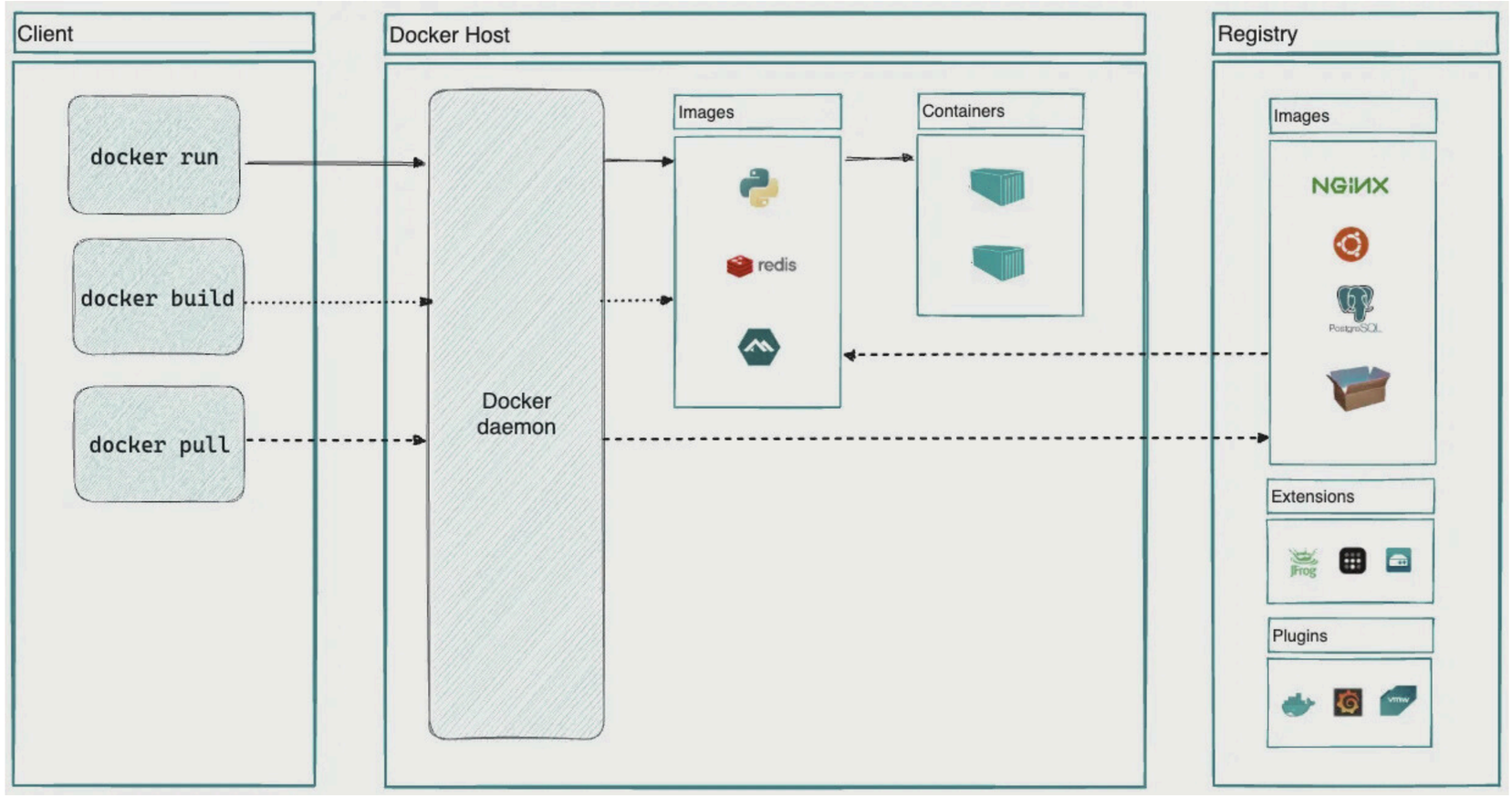
Alternatives to Docker:

Podman: Lightweight and rootless alternative. It is based on a daemonless architecture.

Apptainer: It is designed for scientific computing and HPC environments (emphasis on security).

Introduction to Docker

Docker Architecture

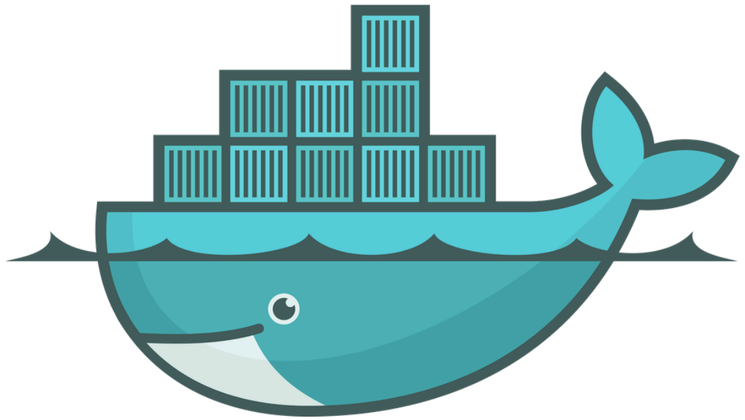


Official documentation: link

Docker:

- uses a client-server architecture
- client and daemon communicate using a REST API, over UNIX sockets or a network interface.

Introduction to Docker



Official documentation: link

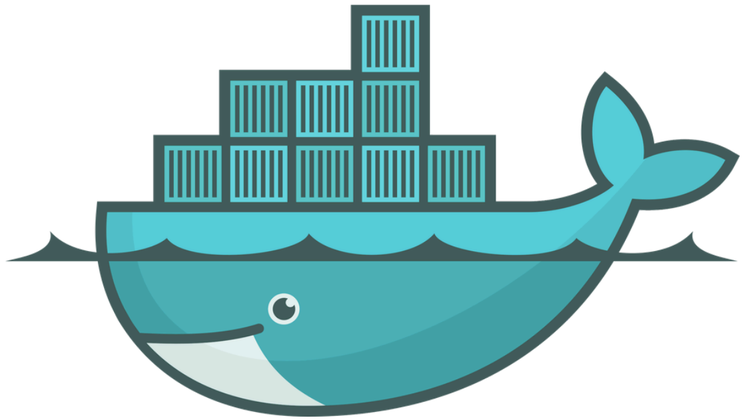
Docker Components:

- ***Docker Registry:*** *A storage distribution system for docker images, where the images can be stored in both public and private modes.*

- ***Docker Hub:*** A public registry that anyone can use. Docker looks for images on Docker Hub by default.

- ***Docker Engine:*** A core part of docker (server, REST API, client). Handles the creation and management of containers.

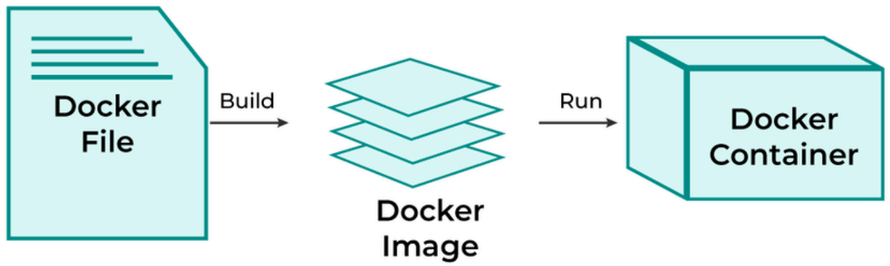
Introduction to Docker



Official documentation: link

Docker Objects:

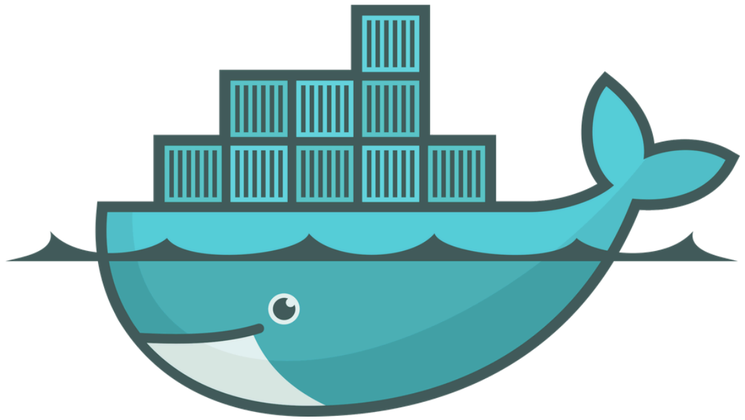
- ***Dockerfile:*** A script containing instructions to build a docker image
- ***Docker Image:*** Read-Only. Used to create containers containing application code & dependencies.
- ***Docker Container:*** A runnable instance of an image. One can create, start, stop, move, or delete a container. Can connect it to networks, attach storage to it, create a new image from current state.



Introduction to Docker

Docker Installation:

https://docs.docker.com/get-started/get-docker/



Official documentation: link

Home / Get started / Get Docker

Get Docker

Docker is an open platform for developing, shipping, and running applications.

Docker allows you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications.

By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

You can download and install Docker on multiple platforms. Refer to the following section and choose the best installation path for you.

> **Docker Desktop terms**
>
> Commercial use of Docker Desktop in larger enterprises (more than 250 employees OR more than $10 million USD in annual revenue) requires a paid subscription.

Docker Desktop for Mac

A native application using the macOS sandbox security model that delivers all Docker tools to your Mac.

Docker Desktop for Windows

A native Windows application that delivers all Docker tools to your Windows computer.

Docker Desktop for Linux

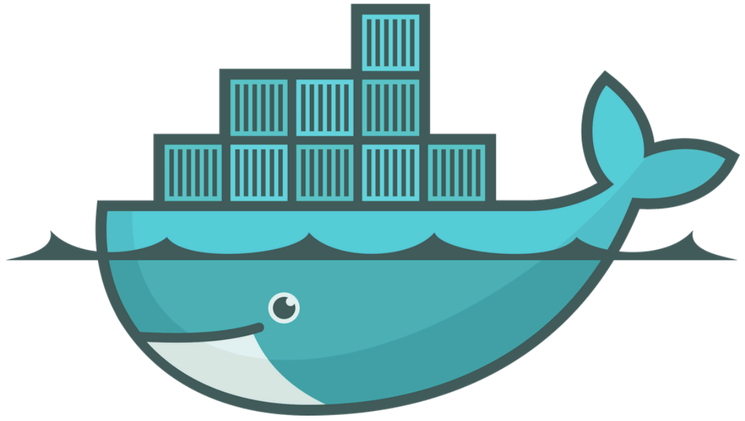
A native Linux application that delivers all Docker tools to your Linux computer.

> **Note**
>
> If you're looking for information on how to install Docker Engine, see Docker Engine installation overview.

Exercises:

Follow the tutorial:
docker-tutorial-and-exercises

google_doc_questions & suggestions



Official documentation: link

Example Docker Commands

Assuming that Docker is installed, the following commands can be used to interact with

- First open a terminal and check if Docker is installed by running `docker --version`
- Run the following commands in the terminal:

Start docker service by running the following or open Docker Desktop

```
sudo systemctl start docker
```

Check if Docker service is running by running

```
sudo systemctl status docker
```

Pull a Docker image from Docker Hub by running

```
docker pull hello-world
```

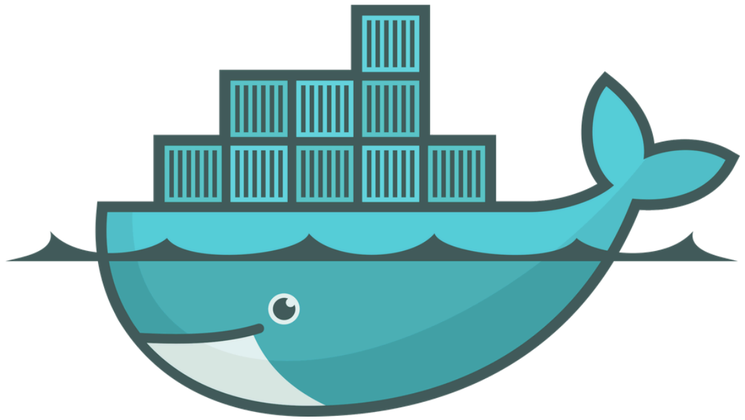
Run the Docker image by running

```
docker run hello-world
```

Quick Recap

Follow the tutorial:
docker-tutorial-and-exercises

google_doc_questions & suggestions



Official documentation: link

- Docker Objects
  - docker file → build
  - docker image → run
  - docker container

- You can create many Docker containers for a given Docker image

- Copy files and dirs from and into the container

- Installing required libraries and software in the image
- Defining a workDir

Version Control with Git

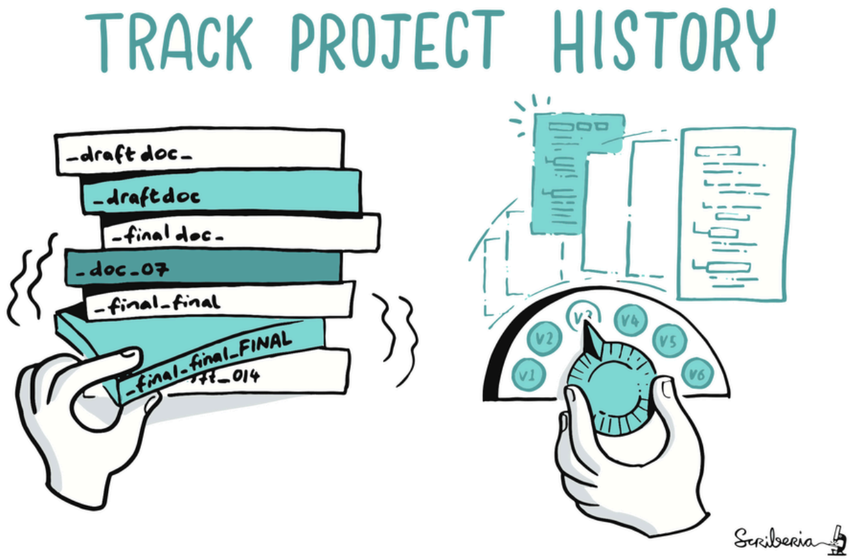
Overview

- version control basics
- git (core concepts)
- git branching
- exercises

Introduction to Version Control

What is Version Control (source control)?

It is a practice of tracking and managing changes to your software code over time.



Ref: The Turing Way Community. This illustration is created by Scriberia with the Turing Way community and used under a CC-BY 4.0 licence. DOI: 10.5281/zenodo.3332807

Version Control Systems (VCS):

- Software tools that help developers to work in teams and thus faster, smarter and efficiently.
- History tracking allows developers to turn back to earlier versions and fix a mistake without disrupting the team members.
- Examples include Google Docs, Overleaf for document tracking.
- Advanced VCS like SVN, Mercurial and **Git** offer powerful tools.

https://www.atlassian.com/git/tutorials/what-is-version-control

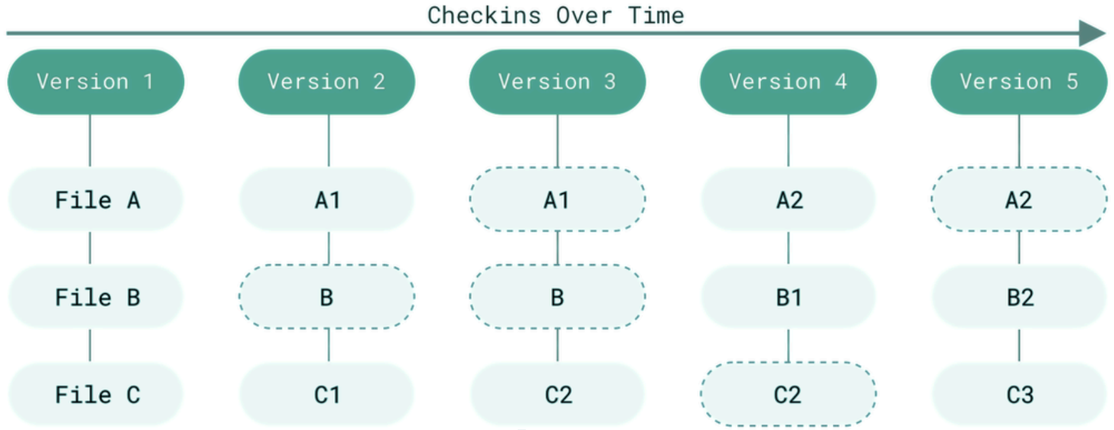
Software Version Control using Git

What is Git?

Git is by far the most widely used modern VCS (**D**istributed **VCS**) that is a mature and actively maintained open-source project.

- Originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel.
- Has been designed with performance, security and flexibility in mind (**agile development**).
- Being DVCS, it stores **snapshots** and not differences.

Ref: Chacon, S., & Straub, B. Pro Git (Version 2) [Computer software].



Snapshots

File — link to an unchanged file from a previous version

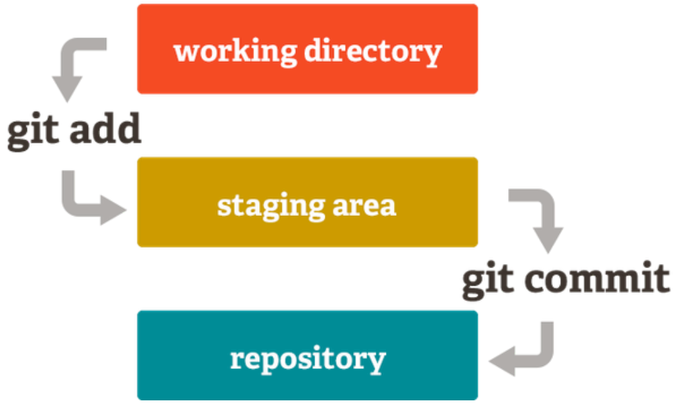
File — changed file from a new version

https://www.atlassian.com/git/tutorials/what-is-git

Software Version Control using Git

Git's Architecture

- **Working Directory**: the directory on your local machine/computer where you make changes.
- **Staging Area**: an intermediate area where commits can be formatted and reviewed before committing.
- **Repository**: where Git permanently stores all the snapshots and history (.git folder on your local machine).



Ref: https://git-scm.com/about/staging-area

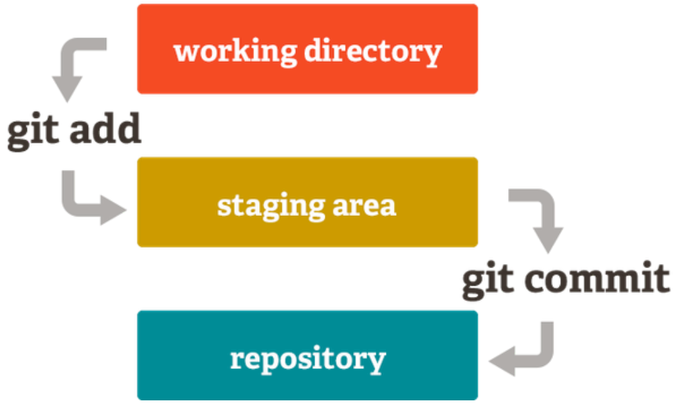
What is a commit?

- pointer to a snapshot of working directory at a certain point in time.
- includes metadata like: the author, commit message, and a reference to previous commit.

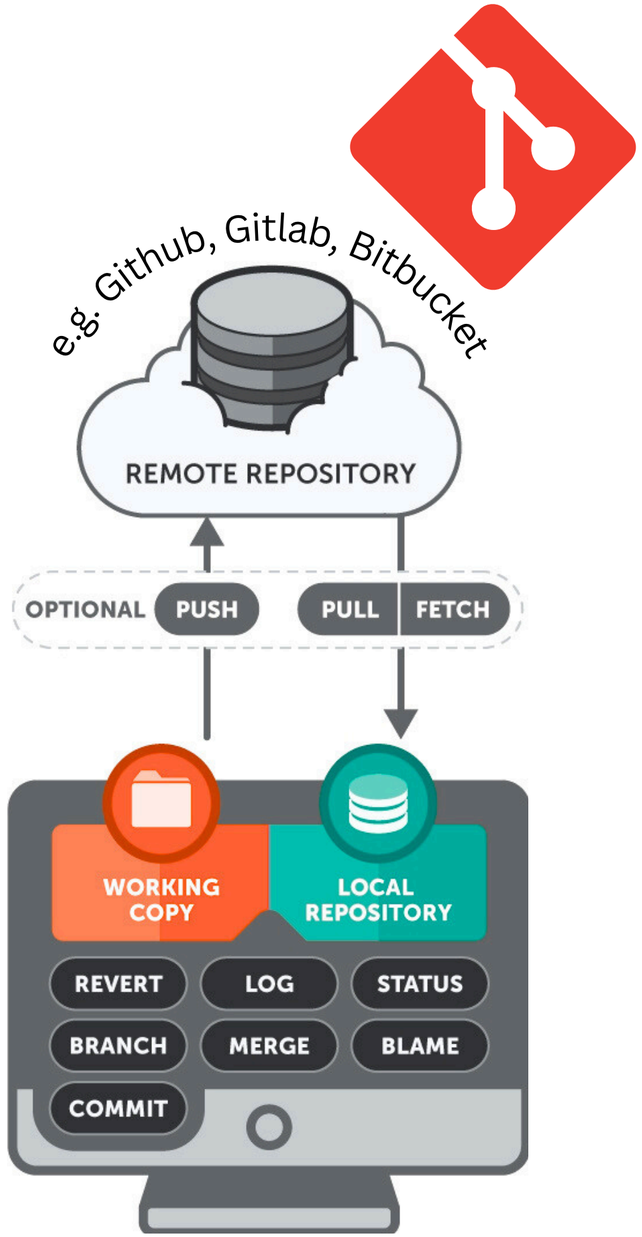
Software Version Control using Git

Git's Architecture

- **Working Directory**: the directory on your local machine/computer where you make changes.
- **Staging Area**: an intermediate area where commits can be formatted and reviewed before committing.
- **Repository**: where Git permanently stores all the snapshots and history (.git folder on your local machine).



Ref: https://git-scm.com/about/staging-area

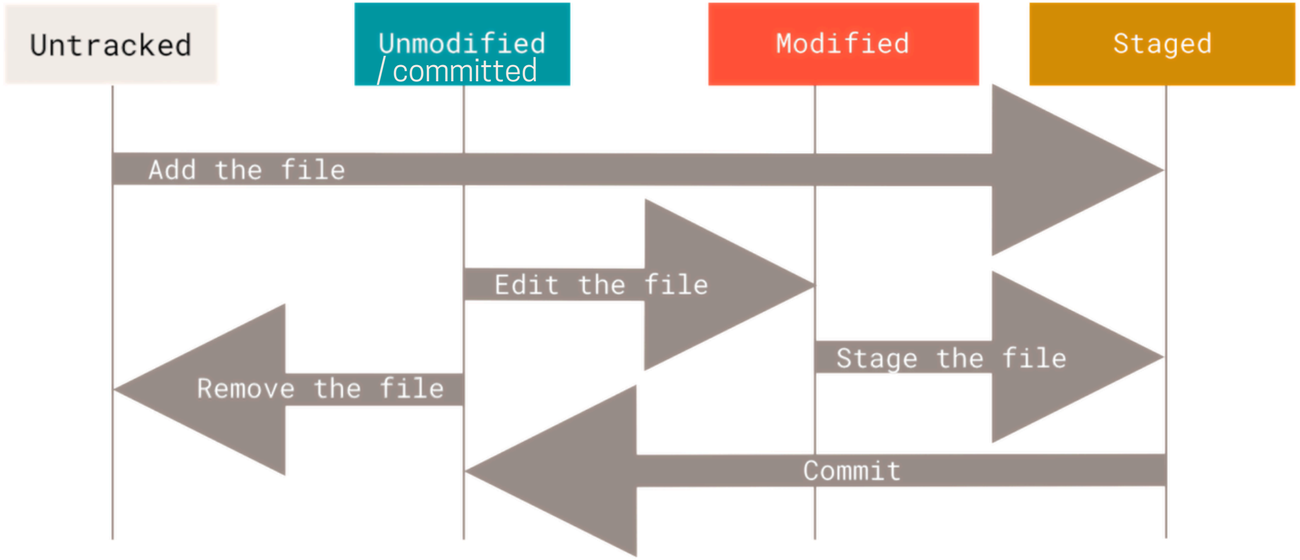


Ref: M. Soni, DevOps for Web Development

Software Version Control using Git

The Three Main States (that your files can reside in)

- **Modified**: means that you have changed the file but have not committed it to your database yet.
- **Staged**: means that you have marked a modified file in its current version to go into your next commit snapshot.
- **Committed**: means that the data is safely stored in your local database (.git folder on your local machine).

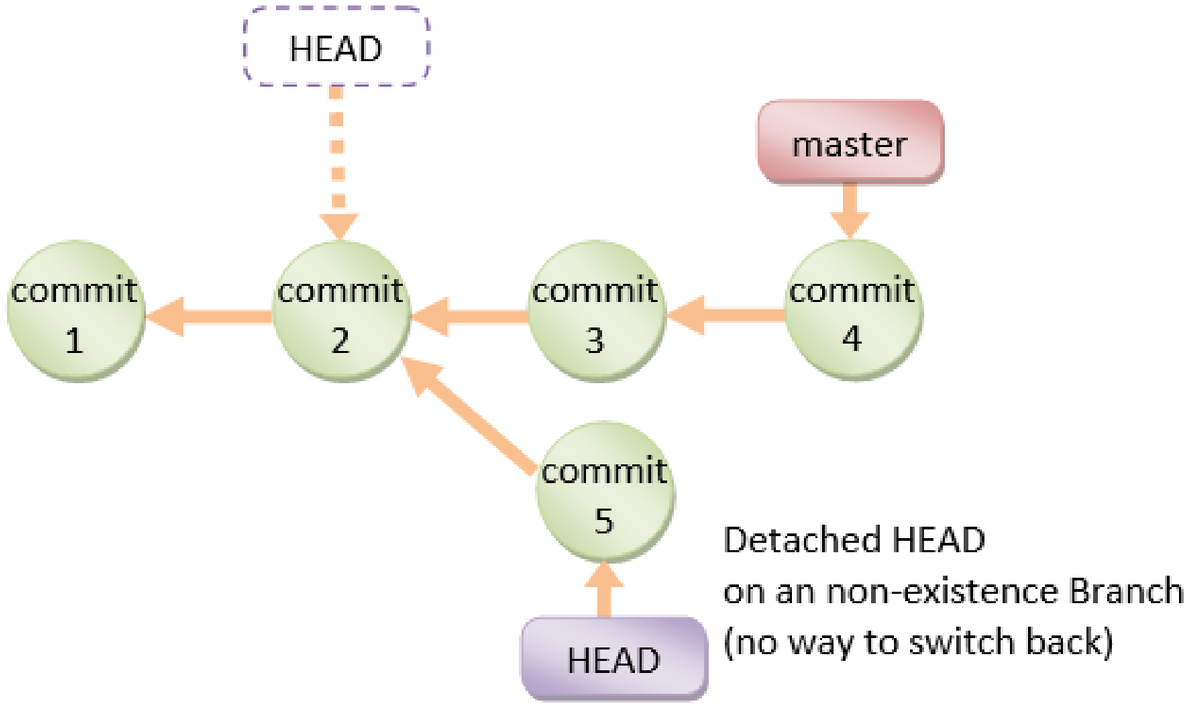


Ref: Chacon, S., & Straub, B. Pro Git (Version 2) [Computer software]v2

Software Version Control using Git

Git Branch

A branch in Git is simply a lightweight movable pointer (**HEAD**) to one of the commits. The default branch is **master**.



Git Branches:

are essential in software development as they allow developers to work on different features or bug fixes simultaneously without affecting the main codebase.

https://www3.ntu.edu.sg/home/ehchua/programming/howto/Git_HowTo.html

Software Version Control using Git

Git Branch

A branch in Git is simply a lightweight movable pointer (**HEAD**) to one of the commits. The default branch is **master**.

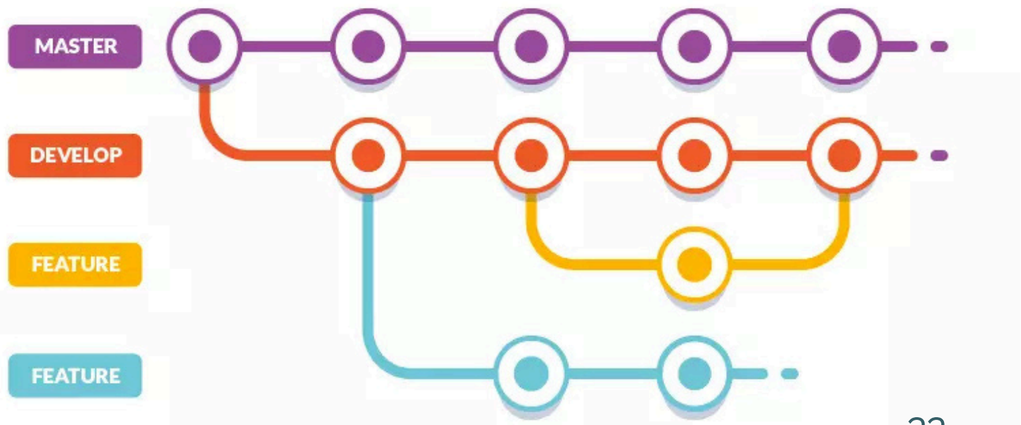
Git Branches:

are essential in software development as they allow developers to work on different features or bug fixes simultaneously without affecting the main codebase.

Branching strategies:

Depending on the size and project requirements, a particular branching strategy can be good or bad.

- Zero Branch Strategy (master): e.g. personal projects, unstable.
- Development Branch Strategy: small size projects, stable, multiple features can't be developed concurrently.
- Gitflow Branch Strategy: large teams, easy to track active features and releases, overkill for small projects.



Ref: https://www.sitepoint.com/use-git-branches-buddy/

Software Version Control using Git

What will you typically do using Git?

- **Clone** repository from Github/Gitlab
- Create your own **branch** and switch (recommended)
- Work in local workspace (edit files on your PC)
- **Add** changes and **commit** changes (.git repository: local)
- **Push** commit(s) to remote repository (Github/Gitlab)

Typical Git Commands:

- git clone <url_of_the_remote_repo>
- git branch <branch_name>
- git checkout <branch_name>
- git add <modified_file(s)>
- git commit -m “<meaningful message>”
- git pull
- git push

Git take aways

Things to keep in mind

- Pull before you push
  Otherwise you will end up with merge conflicts
- Write meaningful commit messages (“Fixed a bug” is not meaningful)
- Commit regularly
- Work on your branch and merge master into your branch regularly
- Don’t break the master branch

-

Git Configuration

To be done once on a machine

https://training.github.com/downloads/github-git-cheat-sheet.pdf

Configure user information for all local repositories

```
$ git config --global user.name "[name]"
```

Sets the name you want attached to your commit transactions

```
$ git config --global user.email "[email address]"
```

Sets the email you want attached to your commit transactions

```
$ git config --global color.ui auto
```

Enables helpful colorization of command line output

Starting a Git Project

https://training.github.com/downloads/github-git-cheat-sheet.pdf

When starting out with a new repository, you only need to do it once; either locally, then push to GitHub, or by cloning an existing repository.

`$ git init`

Turn an existing directory into a git repository

`$ git clone [url]`

Clone (download) a repository that already exists on GitHub, including all of the files, branches, and commits

Branching in Git

https://training.github.com/downloads/github-git-cheat-sheet.pdf

Branches are an important part of working with Git. Any commits you make will be made on the branch you're currently "checked out" to. Use `git status` to see which branch that is.

`$ git branch [branch-name]`

Creates a new branch

`$ git checkout [branch-name]`

Switches to the specified branch and updates the working directory

`$ git merge [branch]`

Combines the specified branch's history into the current branch. This is usually done in pull requests, but is an important Git operation.

`$ git branch -d [branch-name]`

Deletes the specified branch

Day-To-Day Work

https://training.github.com/downloads/github-git-cheat-sheet.pdf

Browse and inspect the evolution of project files

`$ git log`

Lists version history for the current branch

`$ git log --follow [file]`

Lists version history for a file, including renames

`$ git diff [first-branch]...[second-branch]`

Shows content differences between two branches

`$ git show [commit]`

Outputs metadata and content changes of the specified commit

`$ git add [file]`

Snapshots the file in preparation for versioning

`$ git commit -m "[descriptive message]"`

Records file snapshots permanently in version history

Synchronize

https://training.github.com/downloads/github-git-cheat-sheet.pdf

Synchronize your local repository with the remote repository on GitHub.com

`$ git fetch`

Downloads all history from the remote tracking branches

`$ git merge`

Combines remote tracking branch into current local branch

`$ git push`

Uploads all local branch commits to GitHub

`$ git pull`

Updates your current local working branch with all new commits from the corresponding remote branch on GitHub. `git pull` is a combination of `git fetch` and `git merge`

Fixing Mistakes

https://git-scm.com/docs/git-restore
https://git-scm.com/docs/git-reset
https://git-scm.com/docs/git-revert

How to (safely) go back when there is a problem with your code/commits?

- **git restore** remove changes not yet committed
- **git reset** move HEAD to a specified state (can be done keeping or cancelling the changes in the working area)
- **git revert** record a new commit, reverting the changes of a previous one

Fixing Mistakes

https://git-scm.com/docs/git-remote

How to manage remote tracked repositories?
e.g. changing https → ssh or adding upstream repository to local HEAD

- **git remote -vv**
- **git remote add** <origin/upstream> <url>
- **git remote set-url** <origin/upstream> <url>

Exercises:

Follow the tutorial:
git-tutorial-and-exercises

README

Git Play Ground

This is a repository for you to familiarize yourself with git. Some instructions/commands below introduce you to the very basics of git. But of course there's much more...

Setting up your git environment

Create a GitHub account

If you do not have one follow the instructions here to create a GitHub account.

Add your SSH key to your GitHub account

This step is optional but highly recommended if you want to avoid entering username and password every time you push (pull) to (from) remote.
It includes the following steps:

1. Generate an SSH key on your machine (steps 1 and 2 of prerequisites)
2. Add the SSH key to your GitHub account (steps 1-9 of Adding a new SSH key to your account) Follow the instructions here to add SSH key to your GitHub account.

Install git

Follow the instructions here to install git on your machine. You can also use the docker container we created previously. The container has git installed and is ready to use.

Windows users should use the Unix subsystem (WSL) terminal (e.g. Ubuntu) and follow install instructions for Linux.

Check git installation by running `git --version` in your terminal. You should see the version of git installed on your machine.

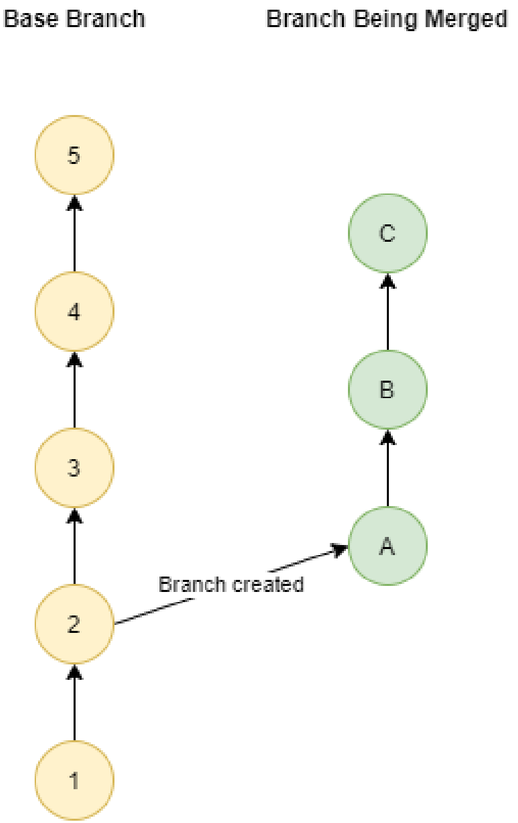
Configure git (to be done once on a machine)

Merge Types

https://lukemerrett.com/different-merge-types-in-git/

You do a **git pull** and have **divergent branches**

```
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 2), reused 2 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (6/6), 2.08 KiB | 1.04 MiB/s, done.
From github.com:tkar-git/git-playground
 * branch            master     -> FETCH_HEAD
 * [new branch]      master     -> upstream/master
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge (the default strategy)
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```



Merge Types

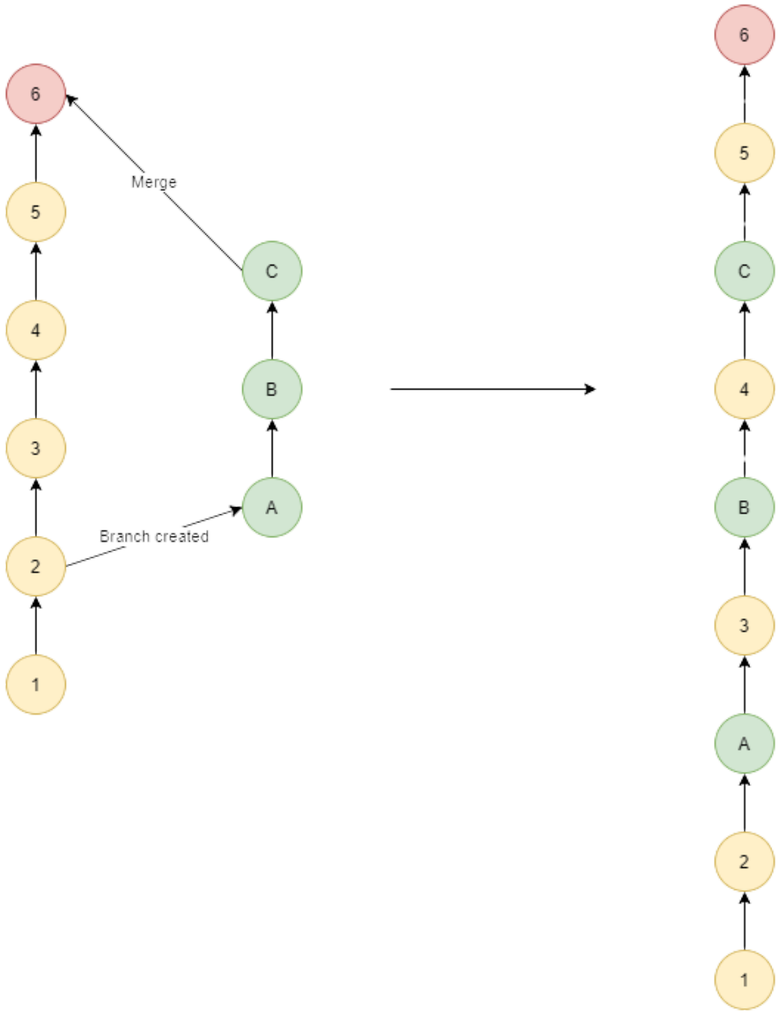
https://lukemerrett.com/different-merge-types-in-git/

1. **git pull --ff-only** will only do "fast-forward" updates: it fails if your local branch has diverged from the remote branch. This is the default.
2. **git pull --rebase** runs git rebase
3. **git pull --no-rebase** runs git merge.
4. **git pull --squash** runs git merge --squash

Merge

https://lukemerrett.com/different-merge-types-in-git/

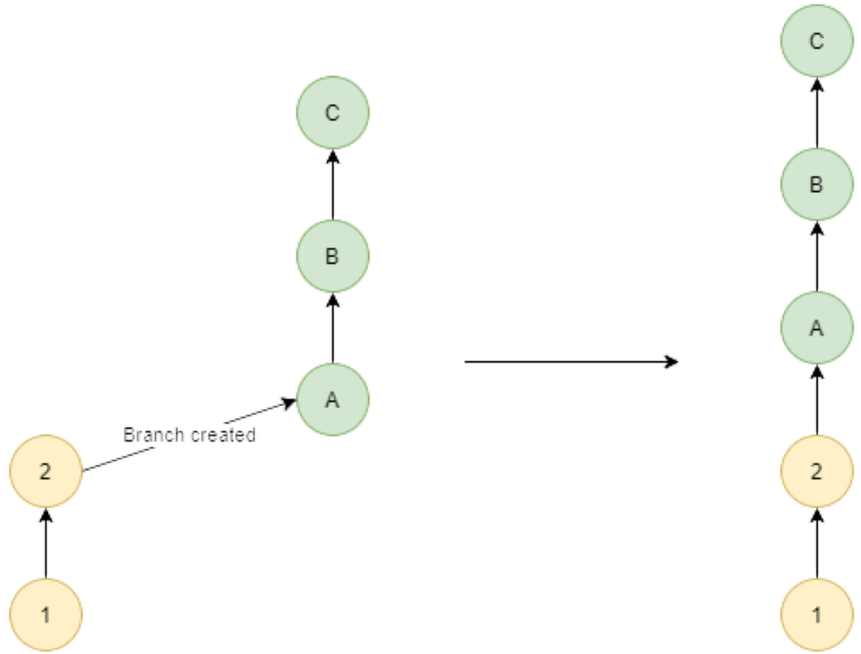
Commits from your branch are written into the base branch history based on their timestamp, a merge commit is created



Fast Forward Merge

https://lukemerrett.com/different-merge-types-in-git/

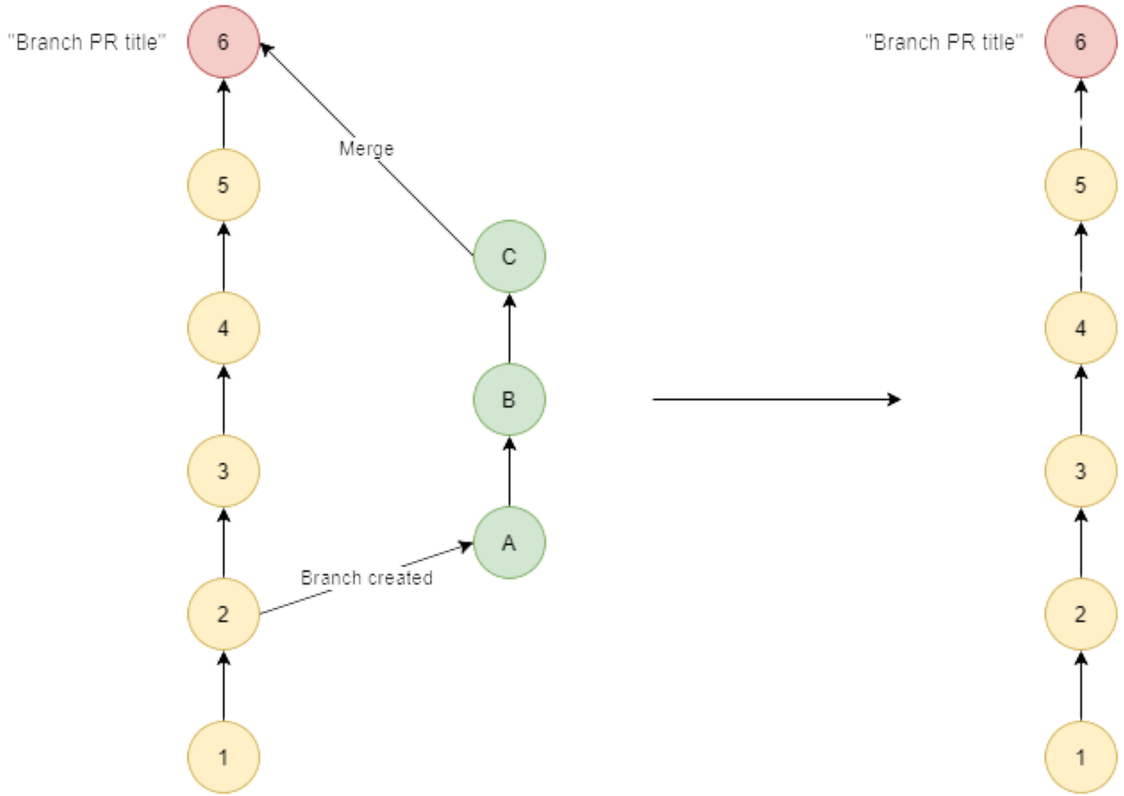
Same as merge but without a merge commit, can be done if there are no commits on the base branch



Squash and Merge

https://lukemerrett.com/different-merge-types-in-git/

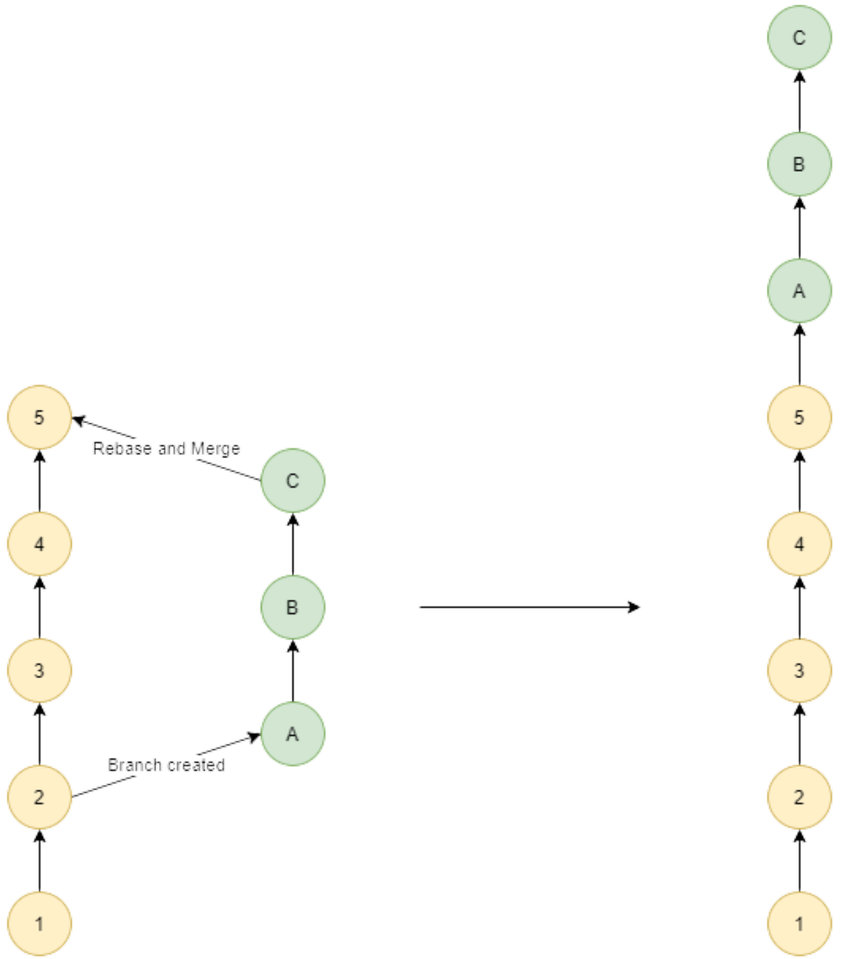
Same as merge, but the commits from the merged branch are “squashed” in a single merge commit



Rebase and Merge

https://lukemerrett.com/different-merge-types-in-git/

The point where your branch was created is moved at the end of the base branch, and the commits from your branch are reapplied one at a time



Group Exercise:

Collaborative Data Analysis Pipeline

Work as a team to build a small data analysis pipeline using Git.

By the end, your group should have:

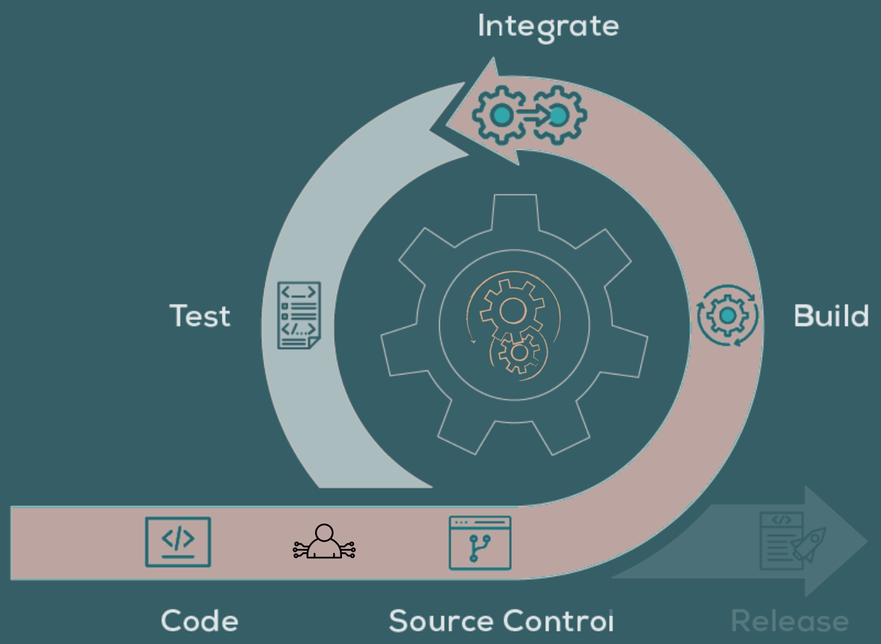
- a working pipeline
- a clean main/master branch
- all work merged via Pull Requests

Repository Structure:

```
miniAnaPipeline/
├──── generator.py   # use a random number generator to create data
├──── data.csv       # x, y, z
├──── process.py     # read and compute, e.g. r, r_perp, θ, φ and write to proceessed.csv
├──── analyze.py     # read processed.csv, determine mean, mode and plot all variables
├──── config.txt     # e.g. scale_factor = 2.0
└──── README.md      # add documentation for your repository
```

Repo url:

git@github.com:tkar-git/miniAnaPipeline.git



Continuous Integration

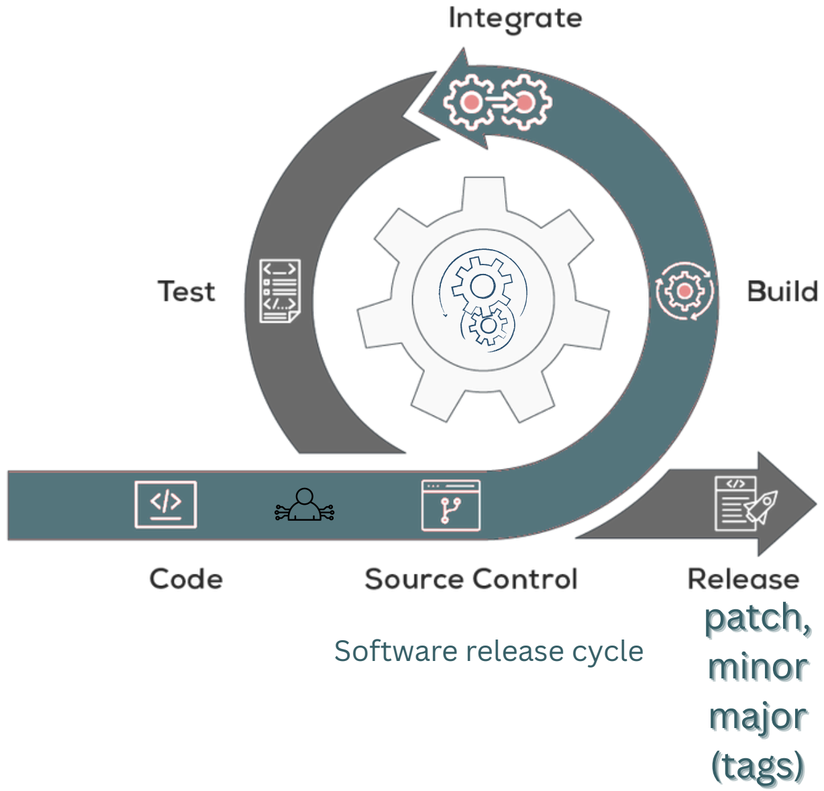
Overview

- Concept
- Exit codes
- YAML and GitHub Actions
- exercises

Continuous Integration (CI)

Typical Software Release Cycle

It follows a structured process that helps teams plan, build, test, and deploy software efficiently. Here's a standard flow:



Software release cycle

Typical software release cycle:

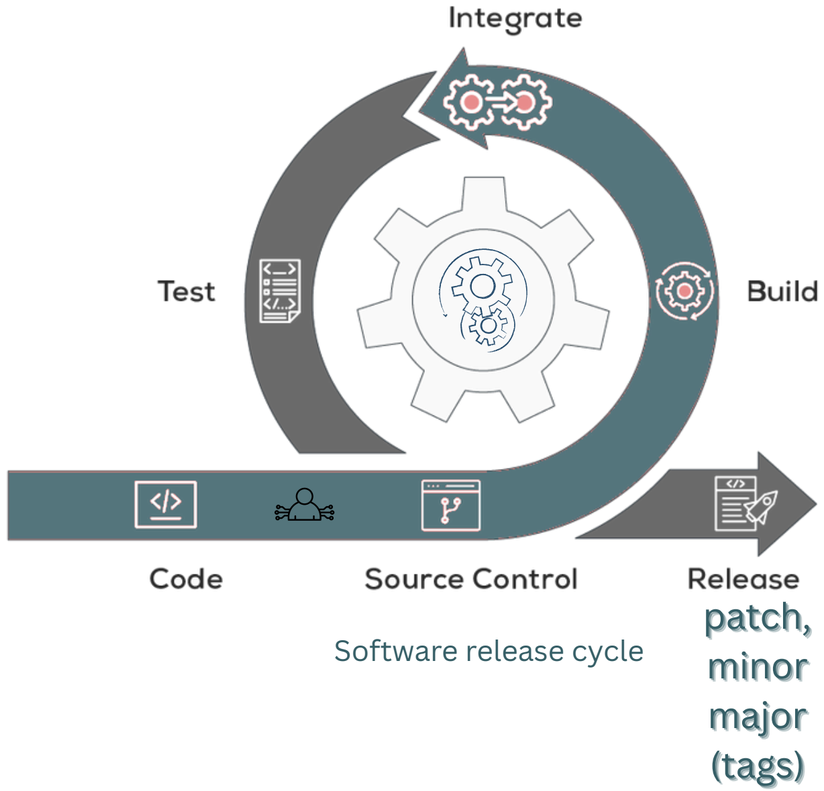
- Planning and Design
- Code Development (including Code Integration)
- Code Building with dependencies
- Code testing
- Release
- Maintainance

https://www.atlassian.com/continuous-delivery/continuous-integration

Continuous Integration (CI)

Typical Software Release Cycle

It follows a structured process that helps teams plan, build, test, and deploy software efficiently. Here's a standard flow:



Software release cycle

Typical software release cycle:

- Planning and Design
- Code Development (including Code Integration)
- Code Building with dependencies
- Code testing → today's task!
- Release
- Maintainance

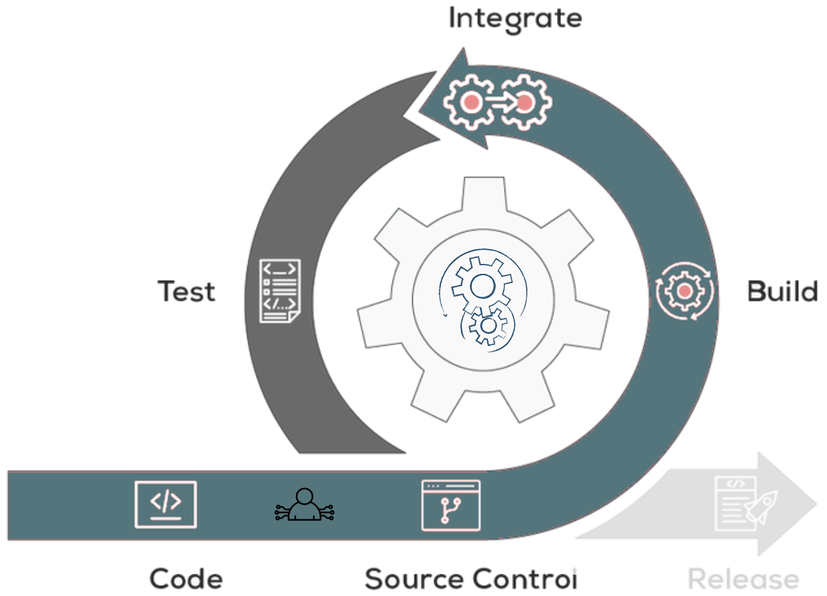
https://www.atlassian.com/continuous-delivery/continuous-integration

Continuous Integration (CI)

What is CI?

CI is the practice of **automating** the **integration** of **code changes** from multiple **contributors** into a **single software project.**

→ every time a contributor provides new changes to your codebase, those changes are tested to make sure they don't "**break**" anything.



Why is CI Important?

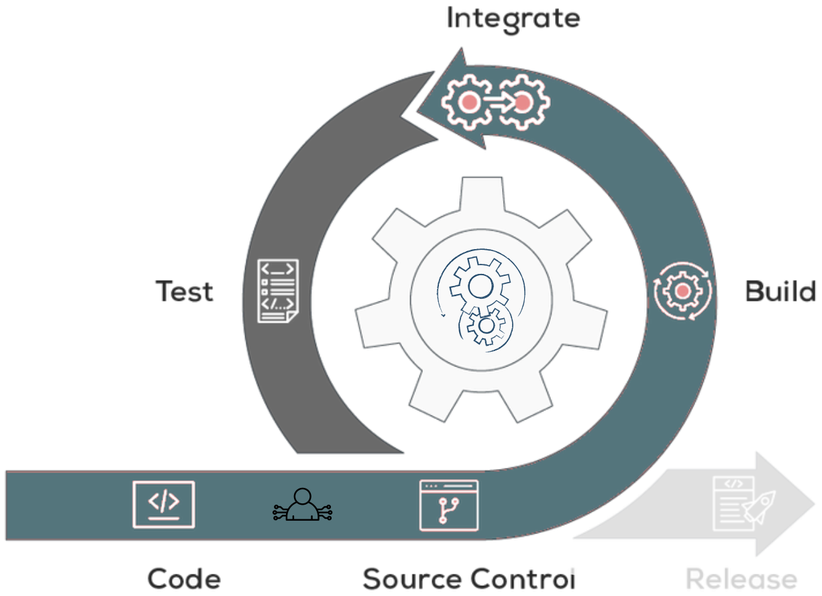
- Early bug detection due to automatic tests (e.g. for every push)
- Prevents worked on my machine situations
- Stable and fast integration
- Faster software releases (developers are more confident)
- Automated Quality Control
- Better Team Collaboration

https://www.atlassian.com/continuous-delivery/continuous-integration

Without CI:

- Last-minute bugs slipping into production.
- Delayed releases due to unstable code
- Tedious manual testing and debugging
- Poor team coordination and stressful merges

Continuous Integration (CI)



YAML and Exit Codes

YAML (**Y**AML **A**in't **M**arkup **L**anguage, originally standing for Yet Another Markup Language) is a human-readable data-serialization language.

↳ It is commonly used for configuration files and in applications where data is being stored or transmitted. CI systems' modus operandi typically rely on YAML for configuration.

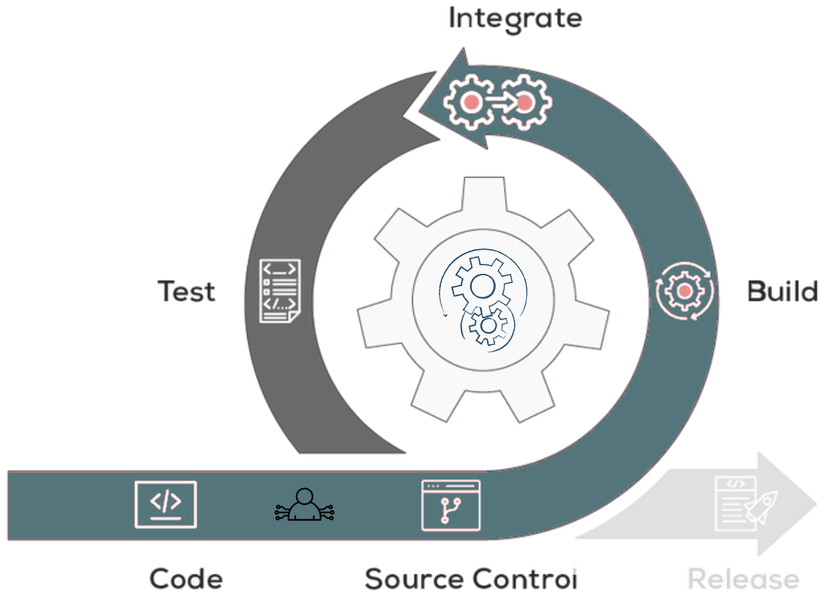
Exit Codes are integer values returned by a command or a program to indicate the result of its execution.

↳ An exit code of zero refers to a successful execution

↳ A non-zero exit code refers to a failure.

https://www.atlassian.com/continuous-delivery/continuous-integration

Continuous Integration (CI)



Why do we need tests, e.g. pytest?
"Your code can run perfectly and still be completely wrong and tests are how you prove it isn't."

Two different kinds:

1. **Runtime checks**: tests inside your main code:
   - validate inputs (file, path existence, and array bounds)
   - prevent crashes
   - early exit with clear error messages

Does not test the correctness of the results!

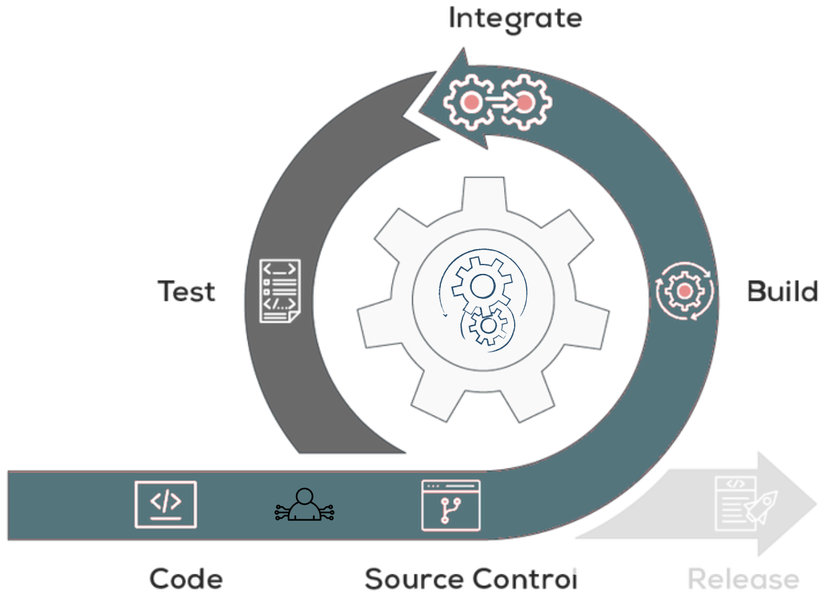
2. **Correctness checks:** tests in the test directory
   - tests if the algorithms/functions function as intended
   - known input and output to detect bugs in the function definition
   - keep the main code logic clean

https://www.atlassian.com/continuous-delivery/continuous-integration

Continuous Integration (CI)

Why do we need tests, e.g. pytest?

“Your code can run perfectly and still be completely wrong and tests are how you prove it isn’t.”




```python
#!/usr/bin/env python
import numpy as np

def compute_std(x):
    mean = sum(x) / len(x)
    return np.sqrt(sum(xi - mean)**2 for xi in x) / len(x)
```

```python
#!/usr/bin/env python
import numpy as np

def compute_std(x):
    mean = sum(x) / len(x)
    return np.sqrt(sum(xi - mean)**2 for xi in x) / (len(x) - 1)
```


https://www.atlassian.com/continuous-delivery/continuous-integration

Exercises:

Follow the tutorial:

ci-tutorial-and-exercises

workflow-syntax

- testing on multiple OS
- strategy: matrix
- test stable and experimental OS/versions
- how do you test if you plots are produced
- and produced correctly

CI pipeline

A CI pipeline is a series of automated steps that are executed whenever code changes are made. The pipeline typically includes the following stages:

1. **Build**: The code is compiled and built into an executable or deployable artifact. This step ensures that the code can be successfully built without any errors.
2. **Test**: Automated tests are run to verify that the code behaves as expected. This step helps catch bugs and ensures that new changes do not break existing functionality.
3. **Deploy**: The built artifact is deployed to a staging or production environment. This step ensures that the code is ready for release and can be tested in a real-world environment.

CI configuration

The CI configuration file defines the steps and stages of the CI pipeline. The configuration file is typically written in YAML or JSON format and includes the following sections:

```yaml
name: <name of your workflow>

on: <event or list of events>

jobs:
  job_1:
    name: <name of the first job>
    runs-on: <type of machine to run the job on>
    steps:
      - name: <step 1>
        run: |
          <commands>
      - name: <step 2>
        run: |
          <commands>
  job_2:
    name: <name of the second job>
    runs-on: <type of machine to run the job on>
    steps:
```