



A PORSCHE COMPANY

GIT & DOCKER & CI/CD: MODERN DEVELOPMENT WORKFLOWS - FROM LOCAL TO PRODUCTION

CLUJ-NAPOCA, 15.11.2025

GUEST MANAGEMENT FORM

Welcome to MHP!

Please scan the QR code below to **register your visit** and be informed about important **safety instructions** for our office premises.



SCAN ME

AGENDA

GIT

Versioning control

CI/CD

Continuous integration – Continuous Development Introduction and Basics

Docker

Run your application on any environment

GIT

Introduction to Git

What is Git?

Distributed version control system designed to handle projects of any size with speed and efficiency

Created by **Linus Torvalds in 2005** (the creator of Linux) to manage Linux kernel development

Tracks changes to files over time, maintaining a complete history

Enables seamless collaboration between multiple developers

Free and open-source software

Why Use Git?

Track history: Complete record of who changed what, when, and why

Collaborate: Multiple people can work on the same project without conflicts

Work in parallel: Create branches for new features without affecting the main codebase

Revert changes: Easily go back to previous versions if something breaks

Work offline: Most operations can be performed without internet connection

Backup: Multiple copies of the codebase exist on different machines

Git vs. Other Version Control Systems

Distributed vs. Centralized:

- Git: Every developer has a complete local copy (distributed)
- SVN/CVS: Single central repository (centralized)

Performance: Git is significantly faster for most operations

Branching: Git excels at branch management and merging

Offline work: Git allows full functionality without server connection

Data integrity: Git uses SHA-1 hashing to ensure data integrity

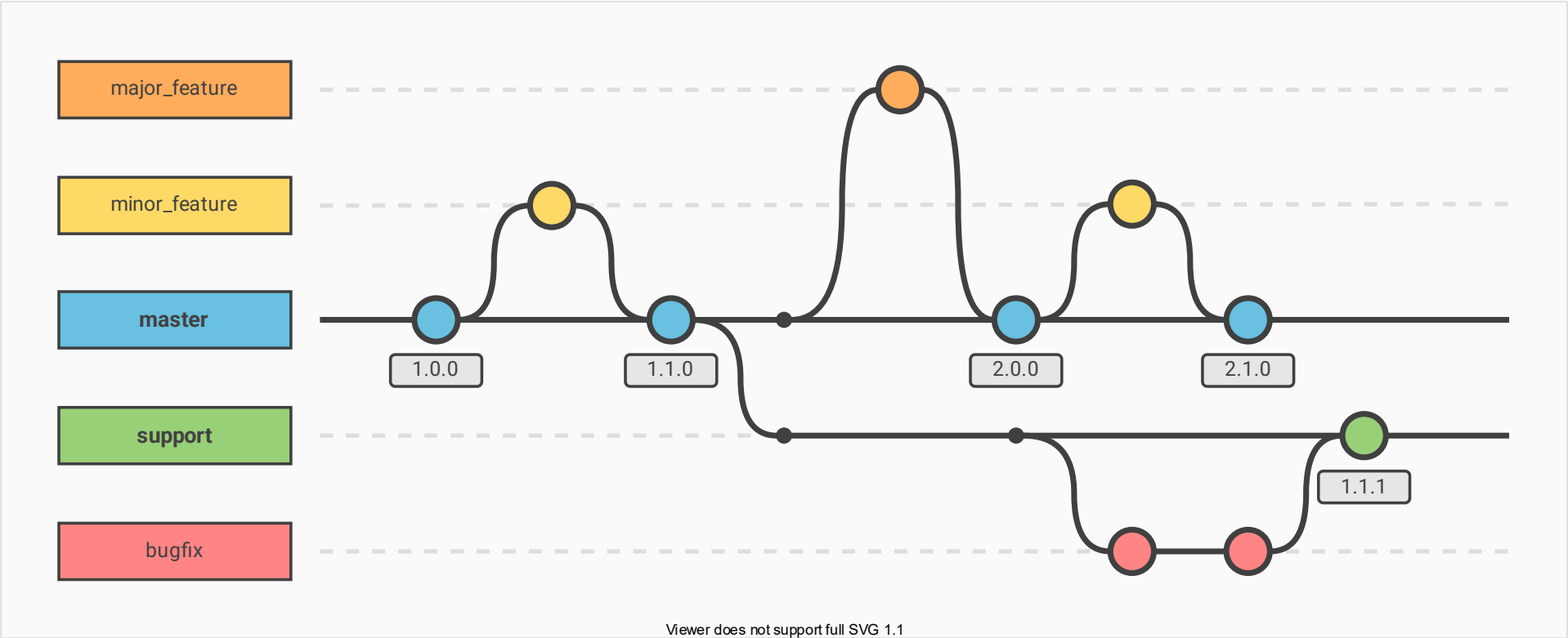
GIT

Git Basics

Git Terminology

- **Repository (repo):** Collection of files and the history of changes made to those files
- **Commit:** Snapshot of changes at a specific point in time
- **Branch:** Independent line of development
- **Merge:** Combining changes from different branches
- **Clone:** Full copy of a repository including all history
- **Pull:** Fetch changes from a remote repository and merge them
- **Push:** Send local commits to a remote repository
- **Working directory:** Files you're currently working on
- **Staging area:** Intermediate area where changes are prepared for commit
- **Stash:** takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy

Git Workflow



- **Working directory:** Where you edit files
- **Staging area (Index):** Where you prepare changes for commit
 - git add moves changes from Working Directory to Staging Area
- **Repository:** Where committed changes are stored
 - git commit moves changes from Staging Area to Repository
- **Remote repository:** Shared repository (like on GitHub)
 - git push sends commits to Remote Repository
 - git pull retrieves commits from Remote Repository

Basic Git Commands

- git **init**: Create a new repository
- git **init** my-project
- git **clone**: Copy an existing repository
- git **clone** <https://github.com/username/repository.git>
- git **add**: Add files to staging area
- git **add** filename.txt # Add specific file
- git **commit**: Commit changes to repository
- git **commit** -m "Add new feature"
- git **status**: Check status of working directory
- git **status**
- git **log**: View commit history
- git **log** --oneline # Compact view

GIT

Installation and configuration

Download and Install Git Bash

1. Visit the official Git website: <https://git-scm.com/downloads>
2. Click on "Windows" to download the installer
3. Run the installer (.exe file)
4. Accept the license agreement
5. Choose the installation location (default is usually fine)
6. Select components (make sure "Git Bash" is checked)
7. Choose the default editor (select your preferred editor)
8. Choose "Git from the command line and also from 3rd-party software" option
9. Choose the SSH executable (use the OpenSSH option)
10. Choose HTTPS transport backend (use the OpenSSL library)
11. Configure line ending conversions (select "Checkout Windows-style, commit Unix-style line endings")
12. Configure the terminal emulator (select "Use MinTTY")
13. Configure extra options (enable file system caching and Git Credential Manager)
14. Complete the installation

Download and Install Git Bash

Set your identity

- `git config --global user.name "Your Name"`
- `git config --global user.email "your.email@example.com"`

Set default branch name to 'main'

- `git config --global init.defaultBranch main`

Configure line ending behavior for Windows

- `git config --global core.autocrlf true`

Add color to Git output

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

Configure credential helper for Windows

- `git config --global credential.helper wincred`

Download and Install Git Bash

Set up useful aliases

- `git config --global alias.co checkout`
- `git config --global alias.br branch`
- `git config --global alias.ci commit`
- `git config --global alias.st status`
- `git config --global alias.unstage 'reset HEAD --'`
- `git config --global alias.last 'log -1 HEAD'`

Pull behavior (avoid unexpected merge commits)

- `git config --global pull.rebase false`

Show current configuration

- `git config --list`

Creating an SSH Key

- SSH keys provide a secure way to authenticate with Git servers without entering your password each time. Here's a step-by-step guide to create and set up an SSH key:

1. Creating an SSH Key

2. Open your terminal or Git Bash

- **Generate a new SSH key** (replace "your_email@example.com" with your actual email):
 - Bash `ssh-keygen -t ed25519 -C "your_email@example.com"`
 - If you're using an older system that doesn't support Ed25519:
 - Bash `ssh-keygen -t rsa -b 4096 -C your_email@example.com`
- 3. ****When prompted for a location****, press Enter to accept the default file location:
Enter file in which to save the key (/c/Users/YOUR_USERNAME/.ssh/id_ed25519): [Press Enter]
- 4. ****Create a secure passphrase**** when prompted (recommended but optional):
Enter passphrase (empty for no passphrase): [Type a passphrase]
Enter same passphrase again: [Type passphrase again]

Creating an SSH Key

Start the SSH agent:

- Bash eval "\$(ssh-agent -s)"

Add your SSH key to the agent: For Ed25519 keys:

- Bash ssh-add ~/.ssh/id_ed25519
- **Copying Your SSH Public Key** - you'll need to add your public key to your Git hosting service:
- Bash clip < ~/.ssh/id_ed25519.pub -> *# Copy to clipboard*

Adding Your SSH Key to Git Hosting Service GitHub:

- Go to your GitHub account → Settings → SSH and GPG keys
- Click "New SSH key"
- Enter a title (like "My Laptop")
- Paste your SSH public key
- Click "Add SSH key"

Creating an SSH Key

Testing Your Connection

- To verify your SSH connection: **For GitHub:**
- Bash `ssh -T git@github.com`
- You should receive a success message if everything is set up correctly.

Changing Your Repository URL to Use SSH

- If you've already cloned a repository using HTTPS, you can switch to SSH:
- Bash `git remote -v` *# Check current remote URL*
- *# Change to SSH*
`git remote set-url origin git@github.com:username/repository.git`
- Replace the URL with your actual repository SSH URL.

Create a repository

- Create repository in UI

- Clone your own repository

git clone <https://github.com/username/repository-name.git> or

git clone [git@github.com:username/repository.git](https://github.com/username/repository.git)

git clone <https://github.com/EvelinnP/MHPWorkshop.git>

git clone [git@github.com:EvelinnP/MHPWorkshop.git](https://github.com/EvelinnP/MHPWorkshop.git)

GIT

Branching & Merging

Git Branching

- Branches allow parallel development streams
- Branches are lightweight pointers to commits

Creating branches:

git branch feature-name # Create branch git checkout -b feature-name

Switching branches:

git checkout branch-name # Switch to existing branch

Listing branches:

git branch # List local branches git branch -a

Default branch: Traditionally master, now often main

Git Branching

Create Branch

git checkout -b branch-name

Step 1: Check Your Current Status

- First, check which files have been modified and need to be committed:
- git status
- This will show:
 - Modified files that aren't staged yet
 - New files that aren't tracked
 - Files already staged for commit

Step 2: Stage Your Changes

- You need to stage the files you want to include in this commit:
- *# To stage all changes*
git add .
- *# OR to stage specific files*
git add filename1 filename2
- *# OR to stage specific directories*
git add directory/

Git Branching

Step 3: Commit Your Changes

- Now commit the staged changes with a descriptive message:
- `git commit -m "Your descriptive commit message here"`

Tips for good commit messages:

- Be specific about what changes you made
- Use present tense ("Add feature" not "Added feature")
- Keep it concise but informative
- Start with a verb (e.g., "Fix", "Add", "Update", "Implement")

Step 4: Verify Your Commit

- Check that your commit was successful:
- *# Show the last commit*
`git log -1`
- *# OR show a summary of recent commits*
`git log --oneline -5`

Step 5: Push Your Branch (When Ready)

- When you're ready to share your changes, push your branch to the remote repository:
- `git push -u origin your-branch-name`

Complete Example Workflow

- *# Check status*
`git status`
- *# Stage all changes*
`git add .`
- *# Create commit*
`git commit -m "Add login functionality to user dashboard"`
- *# Verify commit*
`git log -1`
- *# Push to remote*
`git push -u origin your-branch-name`

Push with Upstream Flag --- important

Use this command to push your branch and set the upstream tracking:

git push --set-upstream origin feature

How to Delete a Git Branch

- *# Switch to another branch (like main or master)*
- git checkout main
- *# Then try deleting the branch* git branch -d branch-name

Deleting a Local Branch

- *# Delete a local branch (only if it's fully merged)*
git branch -d branch-name
- *# Force delete a local branch (even if not merged)*
git branch -D branch-name

Deleting a Remote Branch

- *# Delete a remote branch*
git push origin --delete branch-name

Git Merging

- Merging combines changes from different branches
- **Process:**
git checkout main # Switch to destination branch
git merge feature-branch # Merge changes from feature branch
- **Merge conflicts:** Occur when same lines are changed in both branches
 - Git marks conflicts in files
 - Manually edit files to resolve conflicts
 - Use git add to mark as resolved
 - Complete with git commit
- **Types of merges:**
 - **Fast-forward:** When target branch hasn't changed
 - **Recursive/3-way:** When both branches have changes

Git Merging

- Edit the same file with different text on every branch and push it
- Decide in which branch you want to merge it

Git commands:

- git merge conflict-branch
- git status
- git mergetool
- git add file.txt
- git commit

Git Rebase

- Alternative to merging that creates linear history

Basic rebase:

git checkout feature-branch

git rebase main

When to use:

- Rebase: For cleaner history, especially for feature branches before PR
- Merge: For integrating completed features into main branch

Golden rule: Don't rebase branches that others are working on

- Creates cleaner project history but rewrites commit history

Git Rebase/Git Merge

When to Choose Rebase

- For feature branches only you are working on
- When you want to maintain a clean, linear project history
- Before submitting a pull request to ensure your PR can be applied cleanly
- When your commits need cleanup before being shared
- For keeping a feature branch up-to-date with main during development

When to Choose Merge

- For public/shared branches that others are working on
- When the exact history of the branch is important
- When you want to preserve the context that work was done in parallel
- For long-running branches (like release branches)
- When you're less comfortable with Git (merging is generally safer)

GIT

Repositories

Repositories

Online hosting services for Git repositories:

- **GitHub**: Most popular, owned by Microsoft
- **GitLab**: Complete DevOps platform, open-source edition available
- **Bitbucket**: Integrates with other Atlassian products

Working with remotes:

```
git remote add origin https://github.com/username/repo.git # Add remote
```

```
git remote -v # List remotes
```

```
git push -u origin main # Push to remote
```

```
git pull origin main # Pull from remote
```

- Remotes enable collaboration between developers
- Can have multiple remotes for a single repository

Pull Requests/Merge Requests

- Mechanism for reviewing code before it's merged

Typical workflow:

- Fork repository (on GitHub/GitLab)
- Clone your fork
- Create a feature branch
- Make changes and push to your fork
- Create Pull Request (PR) or Merge Request (MR)

Code review process:

- Reviewers comment on code
- Automated tests run
- Discussion and revisions

Merging: Once approved, changes are merged into main branch

- Different platforms have different terminology but similar concepts

GIT

Commit Best Practices

Commit Best Practices

Atomic commits: One logical change per commit

Clear commit messages:

- First line: Brief summary (50 chars or less)
- Blank line
- Detailed explanation if needed

Conventional commit format:

feat: add user authentication

fix: resolve memory leak in data processor

docs: update installation instructions

Commit frequency: Commit often, push when feature is complete

Don't commit:

- Generated files
- Dependencies
- Environment configuration
- Credentials or secrets

CI/CD

CI/CD Introduction

Introduction to CI/CD (Continuous Integration & Continuous Deployment)

- **What Is CI/CD?**
- **CI/CD** stands for:
- **Continuous Integration (CI)** — automatically **building and testing code** whenever someone makes changes.
- **Continuous Deployment (CD)** — automatically **deploying those changes** to production
- (or a staging environment) once they pass all tests.
- It's all about **automation**, **consistency**, and **speed** — reducing the manual work between writing code and seeing it live.

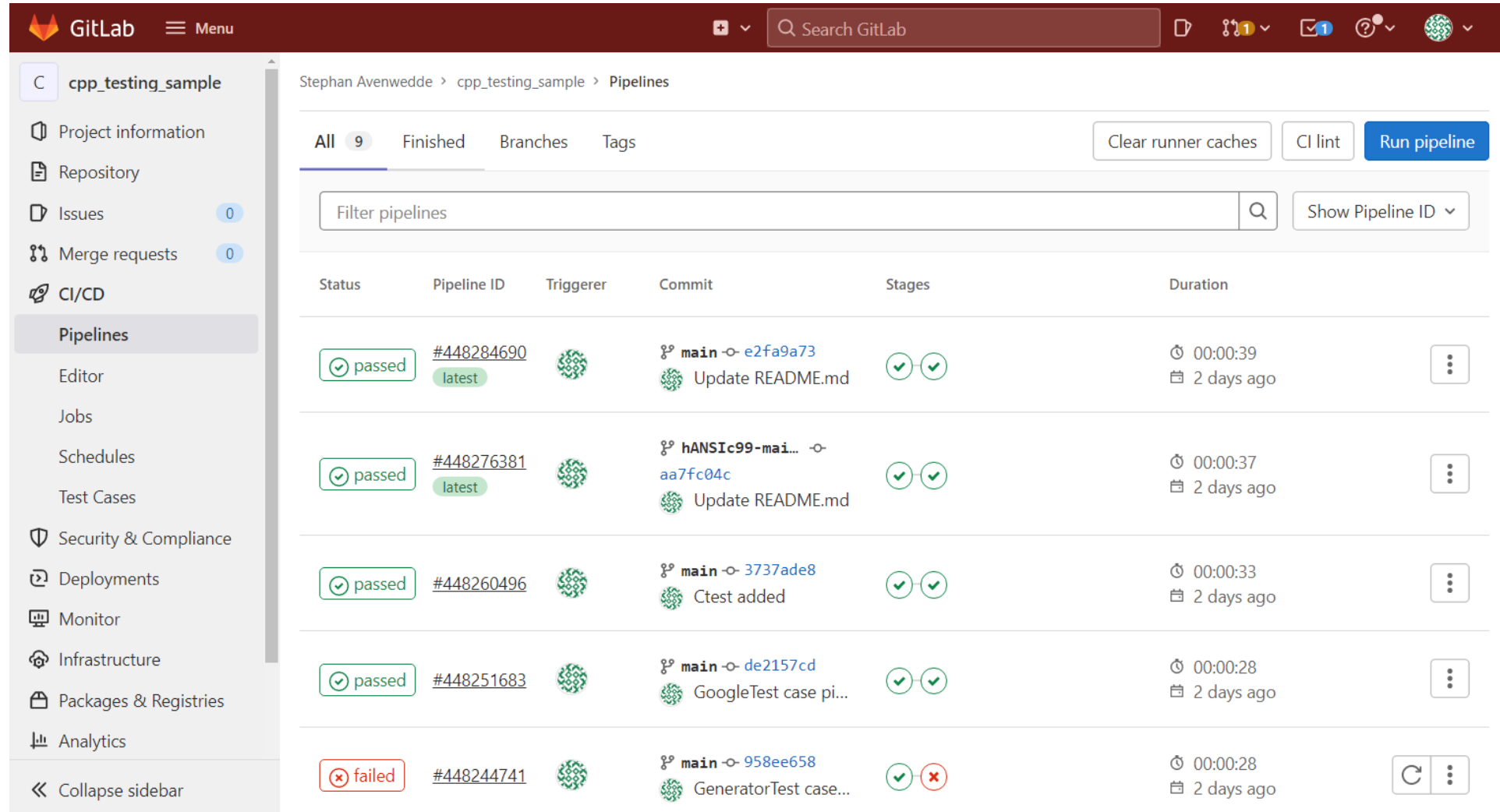
Why use CI/CD?

- Without CI/CD:
 - Developers manually build, test, and deploy code — time-consuming and error-prone.
- With CI/CD:
 - Every code change is automatically tested.
 - Bugs are caught early.
 - Deployments are faster, safer, and more repeatable.
 - In short: **we write code** → **the pipeline does the rest**.

Popular CI/CD platforms

- **Jenkins** – The classic, highly extensible, open-source CI/CD tool.
- **GitLab CI/CD** – Integrated with GitLab, very popular for pipelines and DevOps.
- **GitHub Actions** – GitHub-native, fast adoption for automation.
- **CircleCI** – Cloud-based, simple and fast for containerized apps.
- **Travis CI** – Popular in open-source projects, easy YAML configuration.

Gitlab pipeline example



GitLab Menu

Stephan Avenwedde > cpp_testing_sample > Pipelines

All 9 Finished Branches Tags

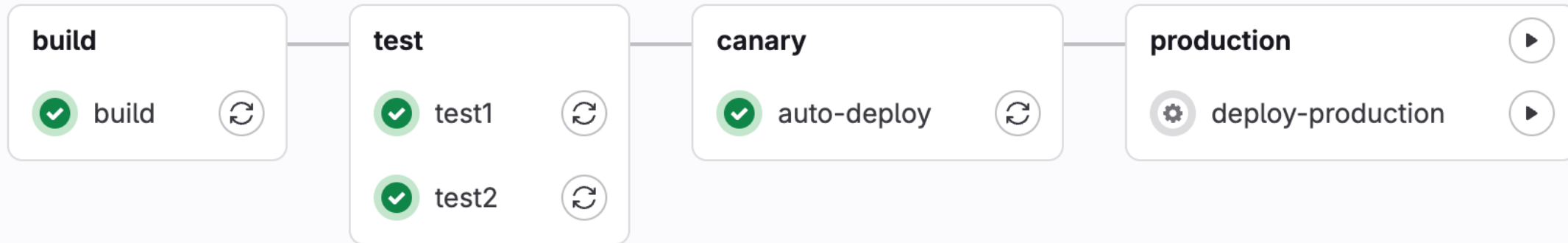
Clear runner caches CI lint Run pipeline

Filter pipelines

Status	Pipeline ID	Triggerer	Commit	Stages	Duration
passed	#448284690 latest		main -> e2fa9a73 Update README.md	✓ ✓	00:00:39 2 days ago
passed	#448276381 latest		hANSIc99-mai... -> aa7fc04c Update README.md	✓ ✓	00:00:37 2 days ago
passed	#448260496		main -> 3737ade8 Ctest added	✓ ✓	00:00:33 2 days ago
passed	#448251683		main -> de2157cd GoogleTest case pi...	✓ ✓	00:00:28 2 days ago
failed	#448244741		main -> 958ee658 GeneratorTest case...	✓ ✗	00:00:28 2 days ago

Gitlab pipeline example - detail

Pipeline Jobs 5 Tests 0



Docker

Docker Basics

What is Docker and why use it?

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

Docker lets developers package an application and all its dependencies — like libraries, frameworks, and configuration files — into a single container. This container can then be run on any system that has Docker installed, ensuring the app behaves the same everywhere.

Why use Docker?

Consistency: Works the same on any machine.

Isolation: Each app runs in its own container, separate from others.

Portability: Run anywhere — on your laptop, server, or the cloud.

Speed: Starts up faster than virtual machines.

Docker Terminology

Docker: Platform for building, running, and managing containerized applications

Container: Lightweight, standalone runtime environment that packages code and dependencies

Image: Read-only template used to create containers

Dockerfile: Text file containing instructions to build a Docker image

Layer: Each instruction in a Dockerfile adds a new cached layer to the image

Tag: Label that identifies a specific version of an image (e.g., `myapp:v1.0`)

Volume: Persistent storage used to save data outside a container's lifecycle

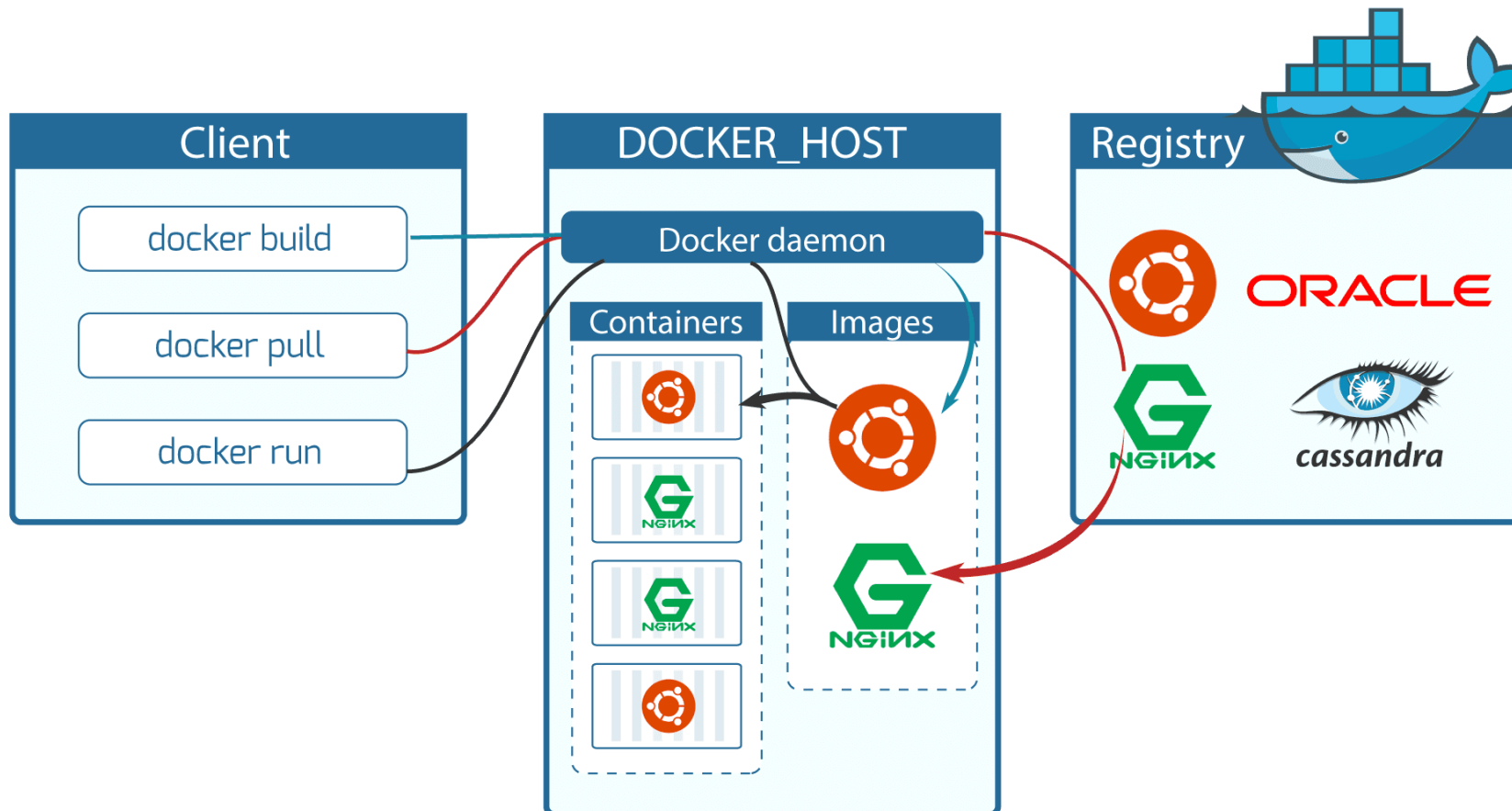
Network: Virtual network that allows containers to communicate with each other

Registry: Service for storing and distributing Docker images (e.g., Docker Hub, ECR)

Repository: Collection of related Docker images with different tags

How does Docker work?

DOCKER COMPONENTS



Dockerfile example

```
# Use an official Node.js runtime as the base image
FROM node:18-alpine

# Set the working directory inside the container
WORKDIR /app

# Copy package files and install dependencies
COPY package*.json ./

RUN npm install --production

# Copy the rest of the app's source code
COPY . .

# Expose port 3000 to the host
EXPOSE 3000

# Define the command to start the app
CMD ["npm", "start"]
```

Basic Docker commands

Checking Docker setup:

<code>docker version</code>	# Show Docker version info
<code>docker info</code>	# Display system-wide information

Working with images:

<code>docker pull nginx</code>	# Download image from Docker Hub
<code>docker images</code>	# List all downloaded images
<code>docker rmi nginx</code>	# Remove an image
<code>docker build -t myapp .</code>	# Build an image from Dockerfile
<code>docker tag myapp myrepo/myapp:v1</code>	# Tag an image for a registry
<code>docker push myrepo/myapp:v1</code>	# Push image to a registry

Basic Docker commands (part 2)

Running containers:

```
docker run nginx                # Run a container from an image
docker run -d -p 8080:80 nginx  # Run container detached and map port 8080→80
docker ps                      # List running containers
docker ps -a                   # List all containers (including stopped)
docker stop mycontainer        # Stop a running container
docker start mycontainer        # Start a stopped container
docker restart mycontainer      # Restart a container
docker rm mycontainer          # Remove a stopped container
```

Inspecting & debugging:

```
docker logs mycontainer        # Show container logs
docker exec -it mycontainer bash # Open a shell inside container
docker inspect mycontainer      # Show detailed info about container
```

Quick links

- **Git**
- **CI/CD**
- **Docker**



Thank you for your
attention !