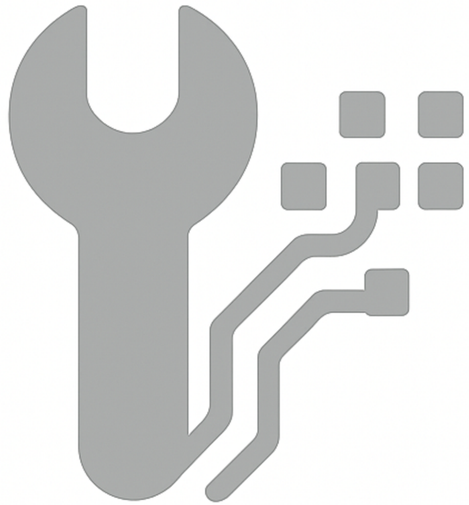
Session 1



WARSZTAT AI

Agenda

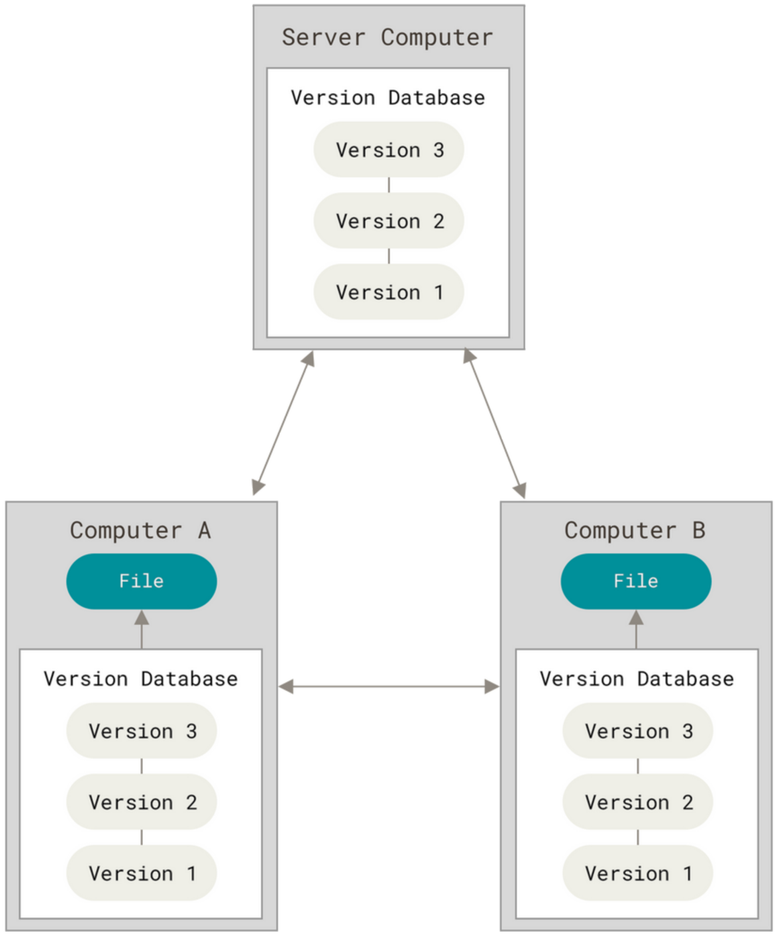
1. Git
2. Package and Environment Managers
3. Vertex AI

Git

About Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

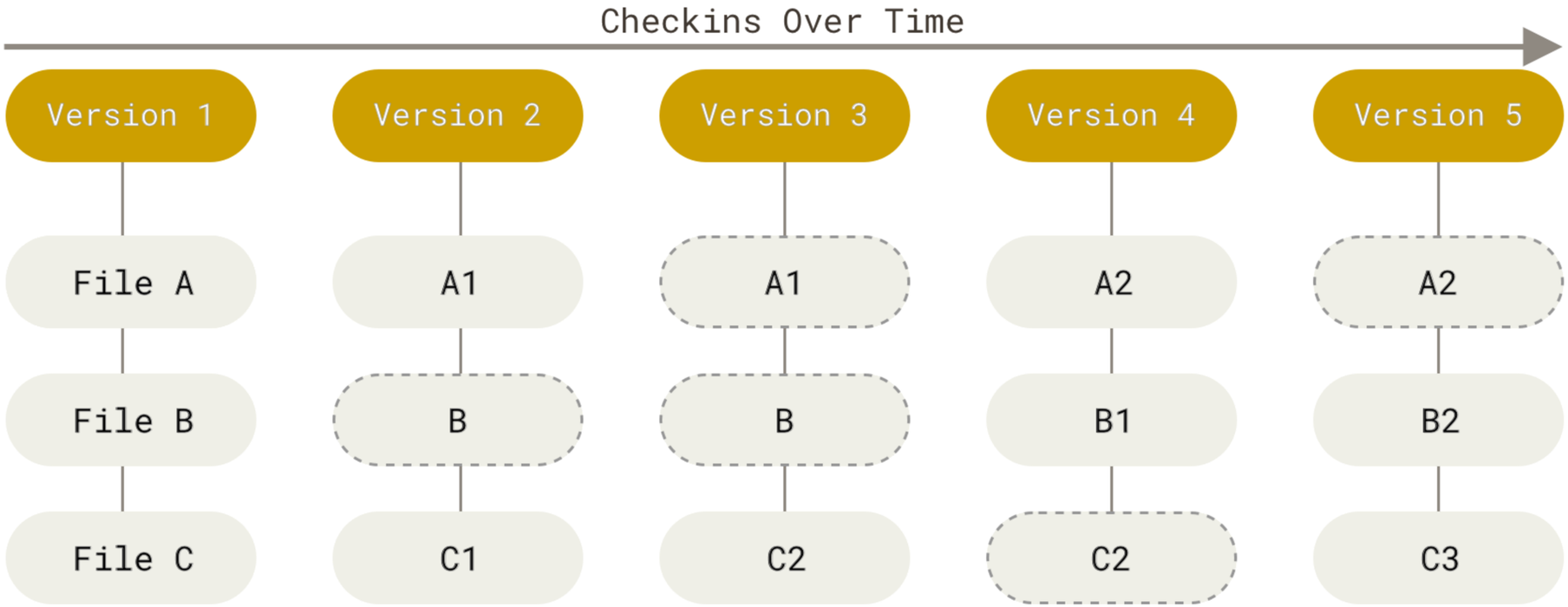
Git is a Distributed Version Control System. Instead of just downloading the latest files, every clone of a Git repository contains the entire history of the project. This means any copy can fully restore the repository if the server is lost.



Distributed Version Control System

Snapshots

Git stores data as a series of snapshots of your project. Each commit is like taking a picture of all your files at a specific moment in time. If a file hasn't changed, Git doesn't save it again but simply links to the previous version.

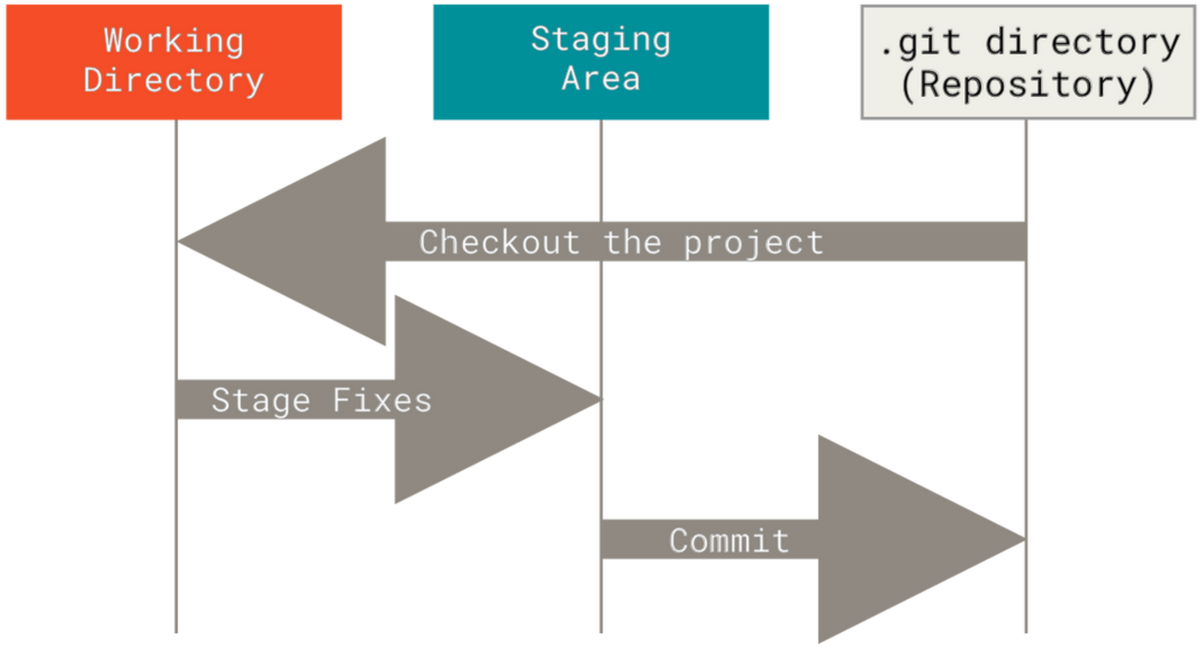


Storing data as snapshots of the project over time

The Three States

Git has three main states that your files can reside in: modified, staged, and committed:

- **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.



Basic commands

Setting Up & Getting a Repository

- **init**: Creates a brand new Git repository in your current folder. Use this to start tracking a new project.
- **clone**: Copies an existing repository from a remote server (like GitHub) to your local machine. This is how you get a project to start working on it.

The Basic Workflow (Making Changes)

- **add**: Moves changes from your working directory to the staging area. It prepares files for a commit.
- **commit**: Permanently takes the changes from the staging area and saves them as a snapshot in the repository's history. Always include a message.

Basic commands

Reviewing & Inspecting Changes

- **status**: Shows an overview of which files are modified, staged, or untracked. It tells you the current state of your working directory.
- **diff**: Shows the precise line-by-line differences between files (working directory vs. staged, or between commits). It answers "what exactly did I change?".
- **show**: Displays detailed information (metadata and changes) about a specific commit.

Branches & Context Switching

- **branch**: Lists all existing branches. Also used to create or delete branches. A branch is a separate line of development.
- **checkout**: Switches your working directory to a different branch or an older commit. It changes your context to work on something else.
- **merge**: Integrates changes from one branch into your current active branch. It's used to combine the work of different branches (e.g., merging a feature into the main branch).

Basic commands

Sharing & Updating (Remote Repositories)

- **push**: Uploads your local commits to a remote repository (like GitHub). This shares your work with others.
- **pull**: The opposite of push. It downloads new changes from the remote repository and immediately merges them into your current local branch. Use this to get updates from others.

Undoing Things & Troubleshooting

- **reset**: A powerful command to undo changes. It can unstage files (git reset HEAD <file>) or even move your branch pointer to discard recent commits (use with caution!).
- **help**: Shows the help manual for any Git command. For example, git help reset explains all the options for the reset command in detail.

Package and Environment Managers

Why Use Package and Environment Managers?

Managing packages and environments is essential to keep your projects organized, reproducible, and conflict-free. Tools like conda, pip, and venv help you:

- Install and update libraries easily.
- Isolate project dependencies to avoid conflicts.
- Share your setup with others for reproducibility.
- Manage complex dependencies, including non-Python libraries.

This ensures your code works consistently across different machines and projects.

Package and Environment Management Tools

Miniconda
Miniconda is a minimal installer provided by Anaconda. Use this installer if you want to install most packages yourself.

Anaconda Distribution
Anaconda Distribution is a full featured installer that comes with a suite of packages for data science, as well as Anaconda Navigator, a GUI application for working with conda environments.

Miniforge
Miniforge is an installer maintained by the conda-forge community that comes preconfigured for use with the conda-forge channel. To learn more about conda-forge, visit their website.

pip + venv
Pip is the default Python package manager, and venv is the built-in tool for creating isolated environments. Together, they are lightweight and ideal for pure Python projects.

System requirements (Conda)

- A supported operating systems: Windows, macOS, or Linux
- For Miniconda or Miniforge: 400 MB disk space
- For Anaconda: Minimum 3 GB disk space to download and install
- For Windows: Windows 8.1 or newer for Python 3.9

Concepts: Packages

A package is a compressed .tar.bz2 or .conda file that contains:

- system-level libraries,
- Python or other modules,
- executable programs and components,
- metadata in the info/ directory,
- a set of files installed into a defined prefix.

Conda manages dependencies between packages and platforms, and the package format is identical across all operating systems.

Package structure

Concepts: Channels

A channel is the place where conda packages are stored and managed. Packages are downloaded from remote channels, which are simply URLs pointing to directories with packages. By default, conda uses the main channel (sometimes requiring a license), but there are also community channels such as conda-forge, which is free and open to everyone. Channels can be customized, for example to use private or internal repositories. Conda-forge works like PyPI but provides automated builds and peer-reviewed recipes, ensuring consistency and quality.

Concepts: Environments

An environment is a directory that holds a specific set of installed packages. Each environment is isolated — changing one does not affect the others. For example, you can keep NumPy 1.7 in one environment and NumPy 1.6 in another for legacy testing. Environments can be easily activated or deactivated to switch between them, and they can also be shared with others using an environment.yaml file.

Cheatsheet