The Hoang

--MLOps-- Version Control

Understand and Implement Production-Grade Machine Learning Operations

Lecture

- Problem Statement?
- What is Version Control?
- Why Version Control is important?
- Fundamentals of Version Control with Git.
- Version Control in Data Engineering.
- Version Control in MLOps.
- Hands-On Exercises.
- Best Practices & Tools to Explore.

Problem Statement

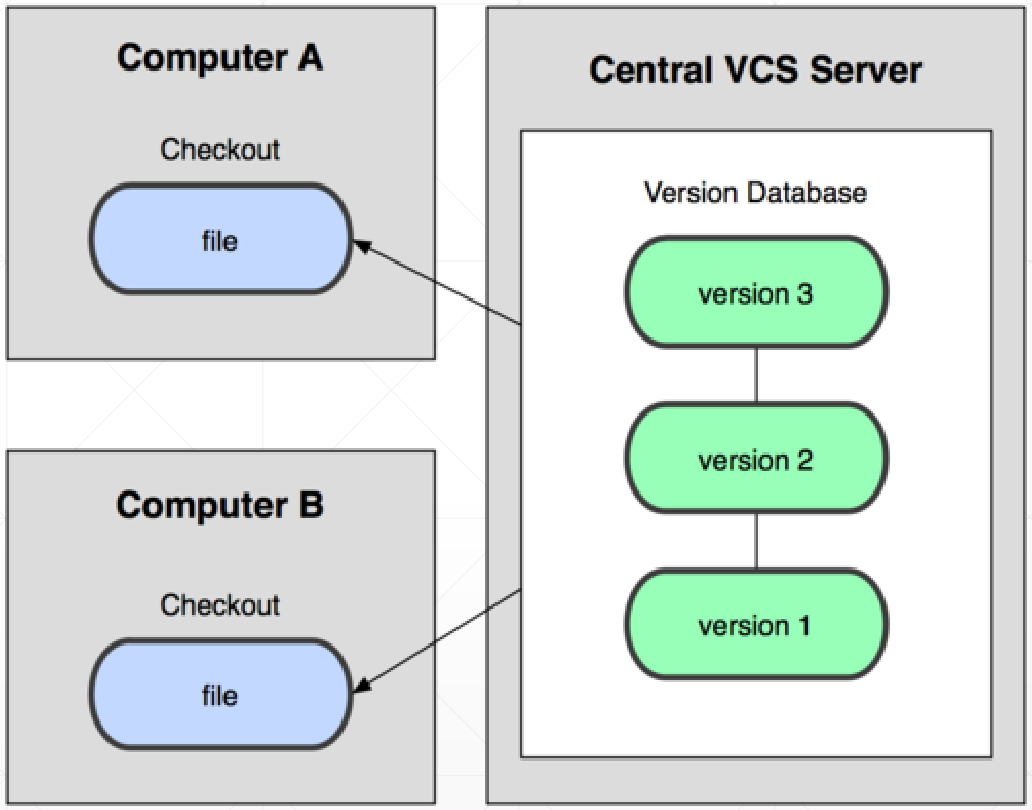
Cannot collaborate as software projects grow in size and complexity:

- Number of programmers working on the same codebase increased
- Overwriting each other’s work
- Losing track of historical changes
- Cannot recover from mistakes or bugs; no backups

What is Version Control?

- Version control is a system that tracks **changes** to files over time. It allows multiple developers or team members to **collaborate**, manage changes, and maintain a **history** of their work.

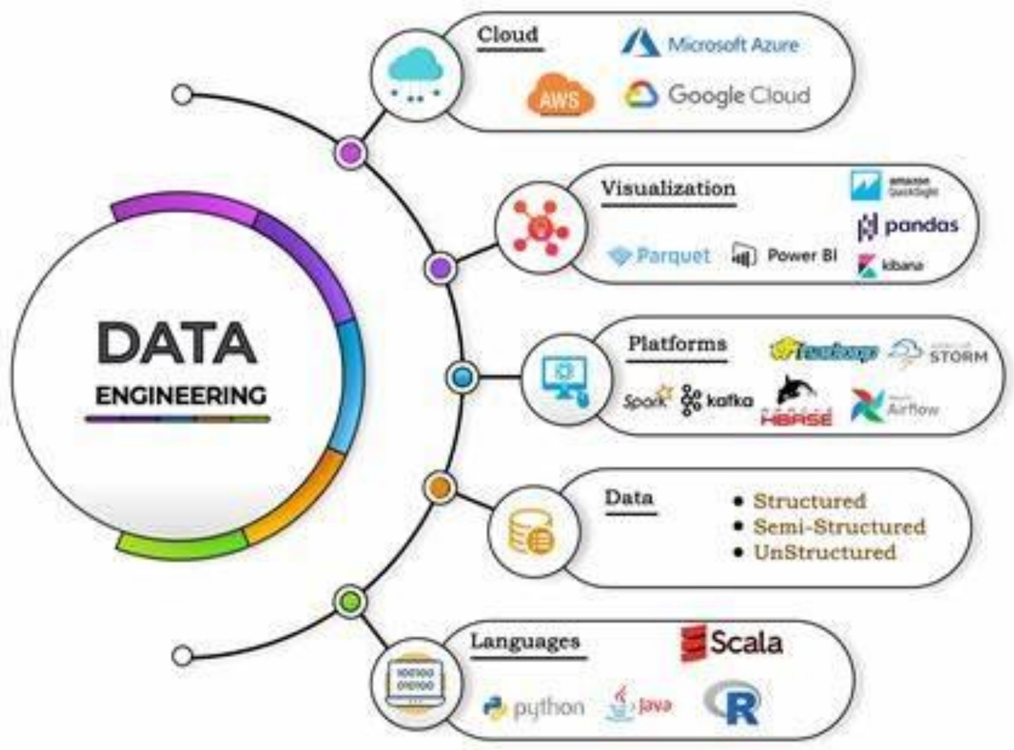
- Think of version control as a "*time machine*" for your code and data projects:
  - You can **see what changed**, **who changed it**, and **when**.
  - If something breaks, you can **revert** to a previous version.
  - It helps ensure **collaboration** without overwriting each other's work.



Why is Version Control? – 1/2

In Data Engineering:

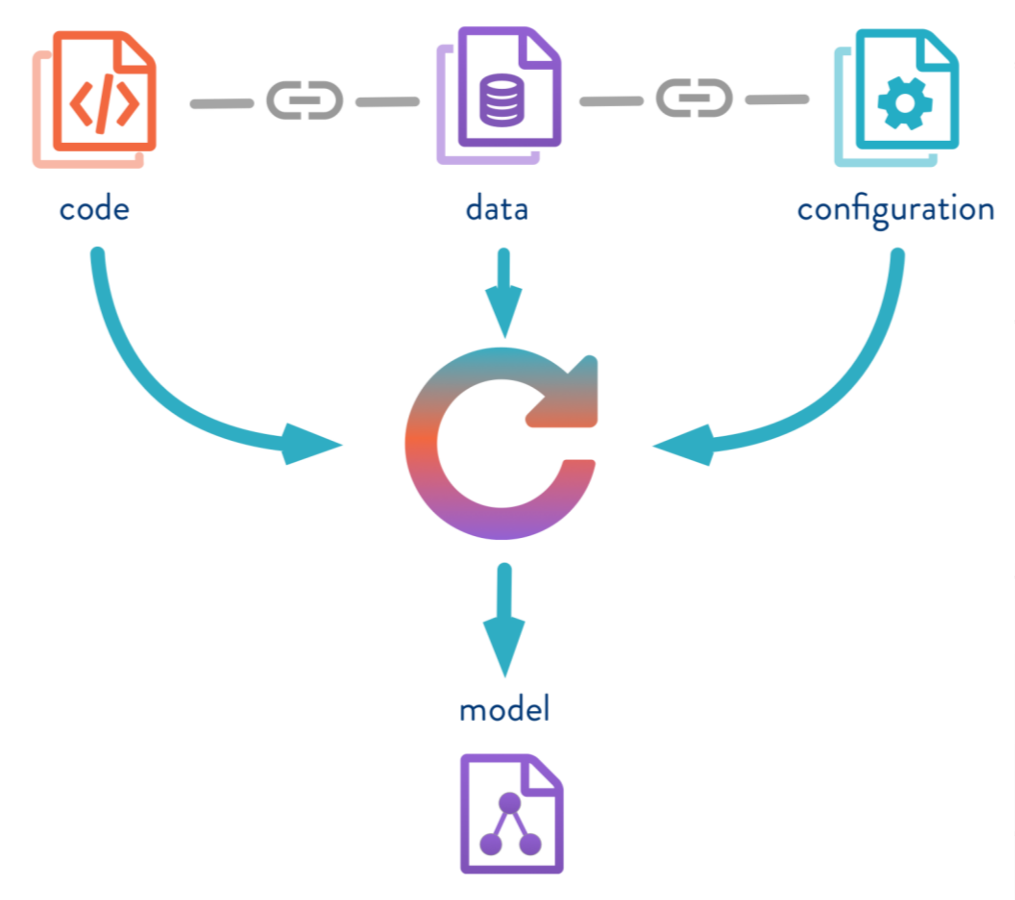
- Data pipelines often evolve (e.g., schema changes, bug fixes, or performance improvements). Version control tracks these changes.
- Infrastructure as Code (IaC) tools like Terraform or dbt (data build tool) rely on version control to manage configurations.
- Collaboration between team members on ETL (Extract, Transform, Load) scripts.



Why is Version Control? – 2/2

In MLOps:

- Machine learning models evolve with time. Things to track:
  - **Code**: Changes in the model's architecture or training script.
  - **Data**: Changes in training datasets or feature engineering steps.
  - **Experiments**: Which hyperparameters were used to train a specific model.
- Tools like **Git** (for code) and **DVC** (Data Version Control) are critical.
- Versioning ensures reproducibility in experiments.

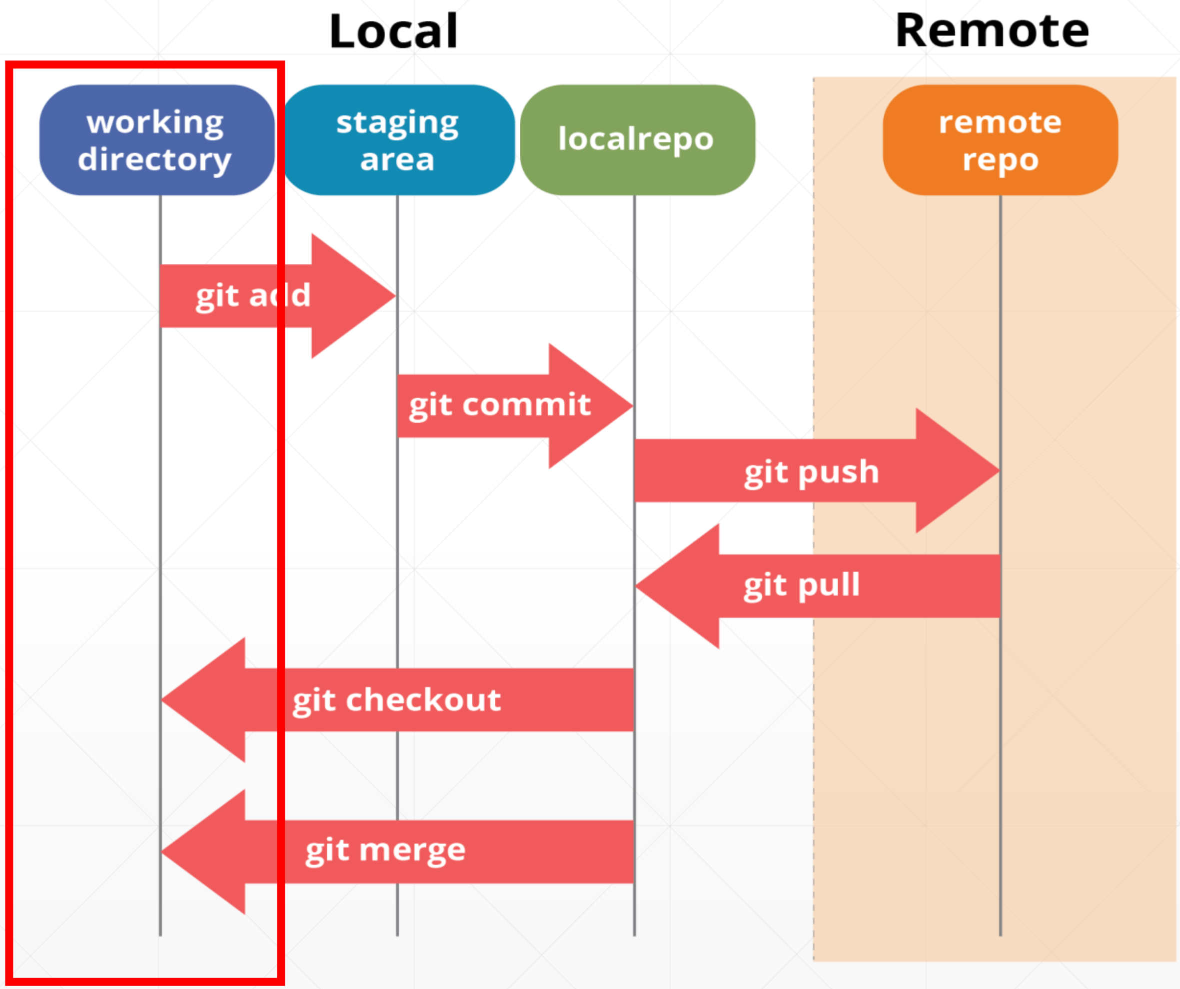


Fundamentals of Version Control with Git

Core Components (1/3)

Working Tree (Working directory)

- This is the local directory where you modify files. It contains the actual files that you are currently working on, including any changes you make.
- Files in this area are considered "untracked" until they are added to the staging area.

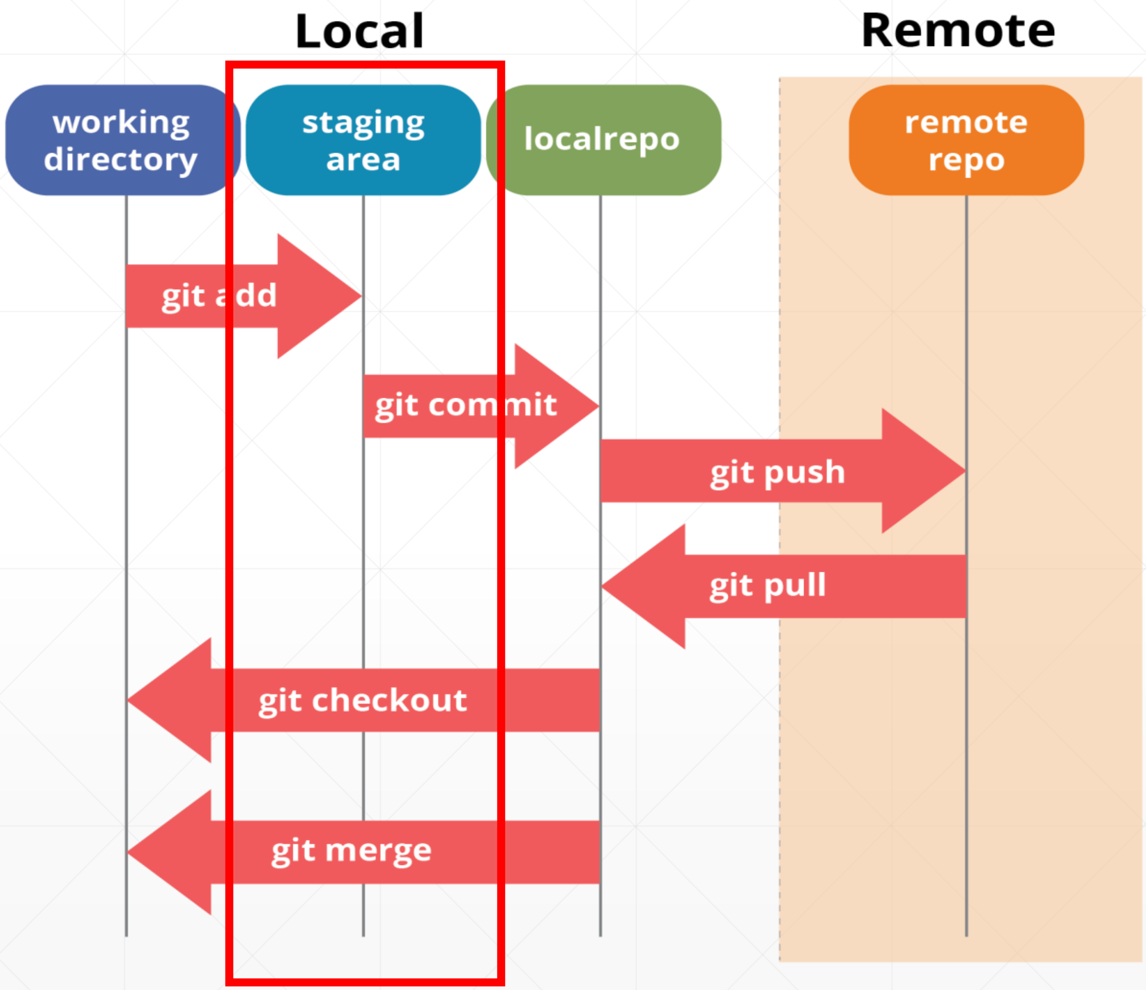


Fundamentals of Version Control with Git

Core Components (2/3)

Index or Staging Area

- The index is where you prepare changes before committing them. It allows you to review changes and selectively stage parts of files.
- You can add files to the index using the command git add <filename>, which marks them for inclusion in the next commit.



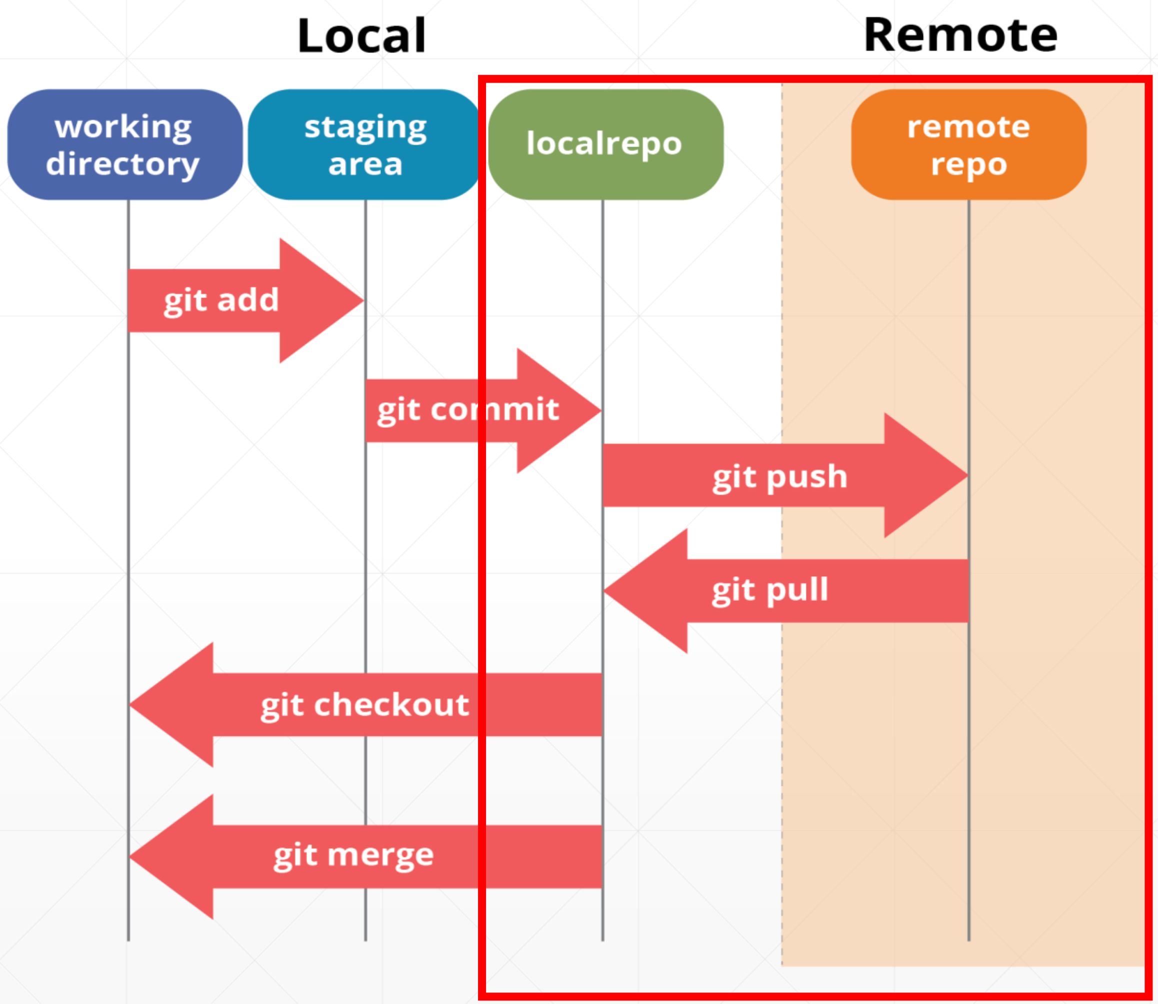
Fundamentals of Version Control with Git

Core Components (3/3)

Repository:

- The repository is where Git stores the complete history of your project, including all commits, branches, and tags.
- Each commit represents a snapshot of your project at a specific point in time, allowing you to track changes over time and collaborate with others effectively

There are local and remote repositories.



Fundamentals of Version Control with Git

Basic Git Commands

Initialize a Repository - Initializes a new Git repository in the current directory.

```
> git init
```

Clone a Repository - Clones a repository from a remote server to your local machine

```
git clone <repository_url>
```

Add Files to Staging Area:

```
> git add <file>
> git add . # Adds all modified and new files Adds specific files or all files to the staging area
```

Commit changes - Creates a new commit with the changes in the staging area and specifies the commit message inline:

```
> git commit -m "commit message"
```

Check Status - Shows the current state of your repository, including tracked and untracked files :

```
> git status
```

Fundamentals of Version Control with Git

Git Commands: Branching and Merging

- Create a New Branch
  - `git branch <branch-name>` *#Creates a new branch with the specified name.*
- Switch to a Branch
  - `git checkout <branch-name>` *#Switches to the specified branch.*
- Merge Branches
  - `git merge <branch>` *#Merges the specified branch into the current branch.*

Fundamentals of Version Control with Git

Git Commands: Remote Repositories

- Fetch Changes
  - `git fetch` *#Retrieves changes from a remote repository.*
- Pull Changes
  - `git pull` *#Fetches changes from the remote repository and merges them into the current branch.*
- Push Changes
  - `git push` *#Pushes local commits to the remote repository.*

Fundamentals of Version Control with Git

Git Commands: Managing History

- Display History

  > `git log` *#Display the commit history of the current branch.*

- Revert a Commit

  > `git revert <commit>` *#Creates a new commit that undoes the changes introduced by the specified commit.*

Fundamentals of Version Control with Git

Git Commands: Utilities

- Check Differences

  `> git diff` *#Shows the changes between the working directory and staging area.*

- Stash Changes

  `> git stash` *#Stashes the changes in the working directory, to switch to a different branch or commit without committing the changes.*

*For more details, checkout "**Git Cheat Sheet**" in the reference material folder.*

Version Control in Data Engineering Projects

1. Managing ETL Pipelines

- Store SQL scripts, Python ETL code, and configuration files in Git.
- Use branching for testing schema changes or new transformations.

2. Infrastructure as Code (IaC)

- Tools like Terraform or dbt allow you to version control infrastructure configurations.
- Example: Store your dbt models and pipelines in Git for collaborative editing.

Version Control in MLOps Projects

1. Data Version Control (DVC)

- Datasets are large and binary, so Git isn’t efficient. Use DVC to version datasets and models.

2. Experiment Tracking

- Combine Git with tools like MLflow or Weights and Biases for end-to-end experiment tracking.
- Each experiment can be tied to a Git commit for reproducibility.

Hands-On Exercise 1

1. Install Git on your machine (if not already installed).
2. Create a new repository.
3. Run git init in an empty folder.
4. Add a Python or SQL file (e.g., etl_script.py) with a simple function.
5. Stage and commit the file
6. Create a free GitHub / GitLab account and create a new repo.
7. Add the remote URL to your local repo
8. [**Homework**] Open the Git Commands Cheat Sheet, practice all other commands.

Hands-On Exercise 2

1. Install DVC
2. Initialize DVC in your Git repo
3. Add a dataset (e.g., data/train.csv) to version control
4. [**Homework**] Push the dataset to a remote storage (e.g., S3, Google Drive)

Best Practices

1. Commit Often: Make small, logical commits with clear messages.
2. Use Branches: Separate development work from the main production code.
3. Automate Testing: Use CI/CD pipelines to test code on every commit.
4. Tag Releases: Clearly define production-ready versions.
5. Track Experiments: Keep track of model training runs and datasets.

Tools to Explore

1. Git: Core version control tool.
2. GitHub/GitLab: Platforms for hosting Git repositories.
3. DVC: Data versioning for ML workflows.
4. MLflow: Experiment tracking and reproducibility.
5. Terraform/dbt: Infrastructure and data pipeline versioning.