AI frameworks

About this class:

- Very applied class (5h CM vs 25hTP)
- ML Ops and Project oriented

- **Goal**: learn common practices to deploy ML applications

Two objectives:

- Learn some advanced machine learning concepts:
  - Conformal prediction
  - Out-of-distribution and Anomaly detection
  - Natural language processing and Vision language models
  - Retrieval augmented generation (RAG)
- Learn classic tools to deploy an AI application:
  - Git
  - REST API
  - Docker

About this class:

Teachers:

Paul Novello

Joseba Dalmau

Anthony Réveillac

David Bertoin

About this class:

- Organization:
  - 5 CM (1h15)
  - 10 TP (2h30) - including 2 dedicated to the project

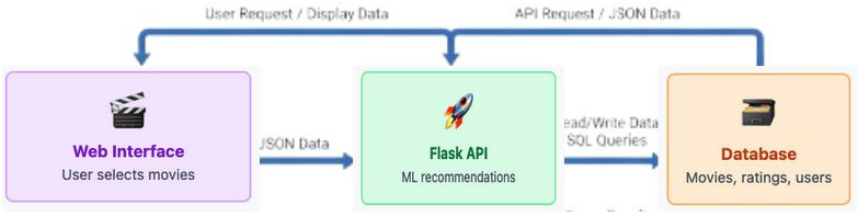
- Evaluation:
  - Quiz after each CM at the beginning of the following practical session
  - Final project

Project:

A common thread project (1 part for each CM-TP(s) block**):**

Deploy a movie recommendation platform.

- Group of 4 people
- Work on the project continuously



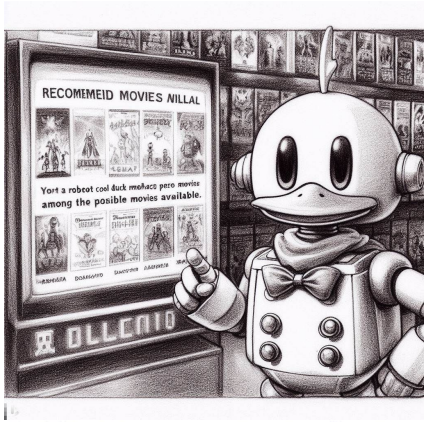
Project:

- **Block 1 :** Poster classification
  - **Goal:** Classify automatically movies genre according to their posters
- **Block 2:** Poster classification
  - **Goal:** Predict with confidence bounds
- **Block 3:** Poster classification
  - **Goal:** detect images that are not movie posters
- **Block 4:** Recommend similar movies
  - Goal: recommend similar movies using posters or movie plots
- **Block 5:** Movie Retrieval
  - Goal: Ask questions about movies

INSA TOULOUSE

WikiStat

From ML Model to Real Deployment



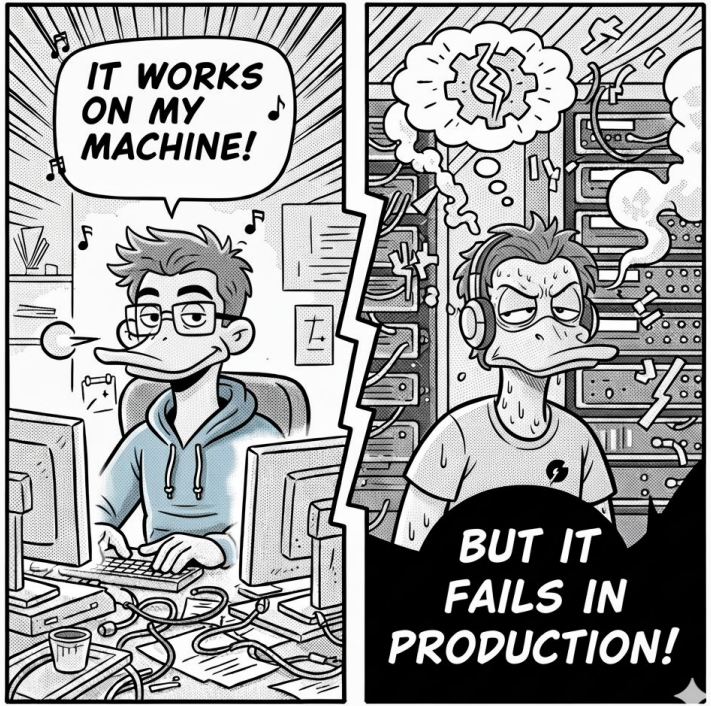
AI frameworks

Jupyter ≠ Production

Notebooks are for exploration, not deployment

Real-world ML needs:

- **Reproducibility** (same code runs everywhere)
- **Scalability** (serve many users, handle large requests)
- **Maintainability** (collaboration, updates, bug fixes)



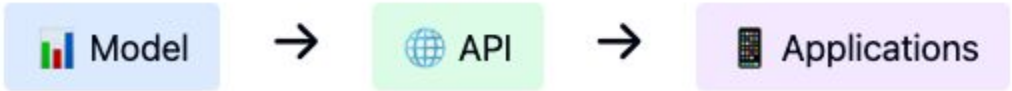
How would you deploy your model?

You trained a new, state of the art and complex deep learning model.

How would you pass it to people responsible for putting it into production?
They don’t know anything about deep learning…

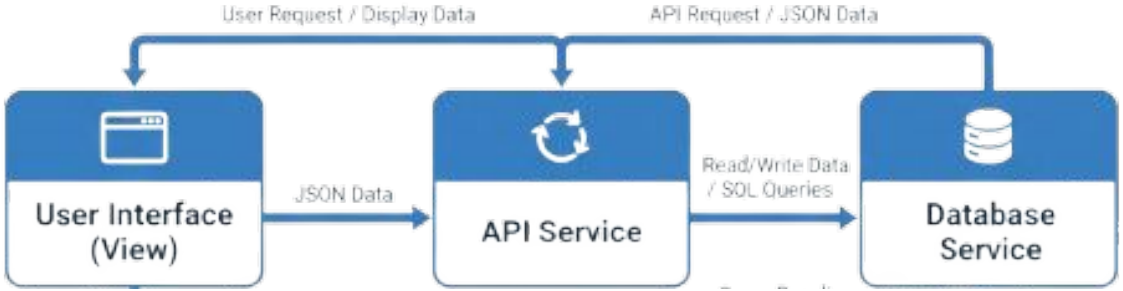
REST APIs for ML

Making Models Production-Ready



Client-Server Architecture

The client and server are separate, allowing them to evolve independently.



What is a REST API?

- **API: Application Programming Interface**
  -> contract between systems

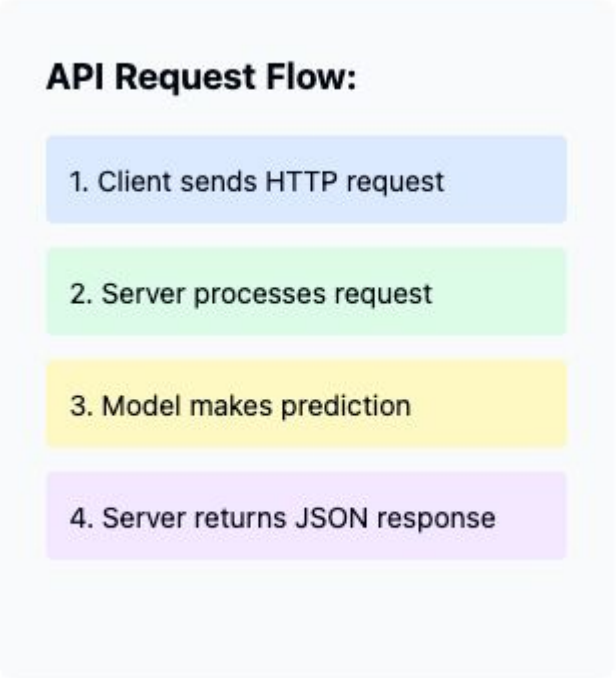
- **REST: Representational State Transfer**
  -> architectural style that uses standard http methods (GET, POST, PUT, DELETE)

REST API flow

The client:

- Makes an HTTP request to a URL
  - Using one of the standard HTTP methods (GET, PUT, POST, PATCH, DELETE, etc.)
  - With some content (usually JSON) in the body

- Waits for a response, which:
  - Indicates status via an HTTP response code
  - And usually has more JSON in the body.



Some other properties

- **Stateless:**
  Each request is independent and contain all the information needed to process the request
- **Cacheable:**
  Responses can be cached to improve performance and reduce server load.
- **Communicates with JSON (most common), HTML, XML, ….**
- **Resource-Based:**
  Built around resources each uniquely identified by a URL (endpoint).
- **Layered System**
  Can be built on multiple layers enhancing modularity (load-balancing, shared caches)

Why Use APIs for ML Models?

✅ Benefits

- **Language Agnostic:** Any language can call your API
- **Scalable:** Handle multiple requests simultaneously
- **Decoupled:** Model and application separate
- **Versioning:** Update models without breaking apps
- **Monitoring:** Track usage and performance

Exemple HTTP Methods in ML APIs

POST /predict

Make predictions with input data

```
POST /api/v1/predict
features: [1.2, 3.4, 0.8]
```

GET /info

Model metadata and version

```
GET /api/v1/info
model: v2.1, features: 10
```

GET /health

Check if model is running

```
GET /api/v1/health
Returns: status: healthy
```

POST /batch

Process multiple predictions

```
POST /api/v1/batch
samples: [[1,2], [3,4]]
```

Flask for ML APIs

Why Flask?

- **Simple:** Minimal, easy to understand
- **Flexible:** Add only what you need
- **Lightweight:** Small footprint
- **Mature:** Well-established ecosystem
- **Pythonic:** Clean, readable code

Alternatives

- **FastAPI:** Auto-docs, type safety
- **Django REST:** Full-featured framework
- **Tornado:** Async-focused

```
# Simple Flask example
from flask import Flask, request, jsonify
app = Flask(__name__)
@app.route('/predict', methods=['POST'])
def predict():
data = request.get_json()
features = data['features']
# Your ML prediction logic
prediction = 0.95
return jsonify('prediction': prediction)
if __name__ == '__main__':
app.run(debug=True)
```

Testing Your ML API

🐍 Python (requests)

```
import requests
url = "http://localhost:8000/predict"
data = features: [5.1, 3.5, 1.4, 0.2]
response = requests.post(url, json=data)
print(response.json())
```

🔧 cURL

```
curl -X POST
"http://localhost:8000/predict"
-H "Content-Type: application/json"
-d 'features: [5.1, 3.5, 1.4, 0.2]'
```

🌐 JavaScript (fetch)

```
const response = await fetch(
'http://localhost:8000/predict',
method: 'POST',
headers: 'Content-Type': 'application/json',
body: JSON.stringify(
features: [5.1, 3.5, 1.4, 0.2]
)
);
const result = await response.json();
```

📊 Flask Testing

Visit: `localhost:8000`

- Use Flask's test client
- pytest for automated testing
- Postman for manual testing
- curl for command line testing

Robust Error Handling

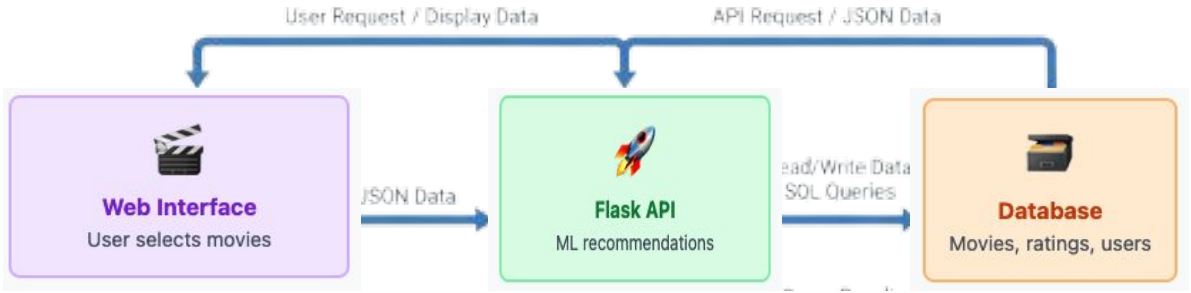
```
# Flask error handling examples
from flask import Flask, request, jsonify
import logging
app = Flask(__name__)
@app.route('/predict', methods=['POST'])
def predict():
try:
data = request.get_json()
# Validate input
if not data or 'features' not in data:
return jsonify('error': 'Missing features'), 400
features = data['features']
if len(features) != 4:
return jsonify(
'error': f'Expected 4 features, got len(features)'
), 400
if any(f is None for f in features):
return jsonify(
'error': 'Features cannot contain null values'
), 400
features_array = np.array(features).reshape(1, -1)
prediction = model.predict(features_array)[0]
return jsonify('prediction': float(prediction))
except ValueError as e:
app.logger.error(f'Prediction error: e')
return jsonify('error': 'Invalid input data'), 400
except Exception as e:
app.logger.error(f'Unexpected error: e')
return jsonify('error': 'Internal server error'), 500
```

400 Bad Request
Client errors

500 Server Error
Unexpected errors

Log Everything
For debugging

Project:

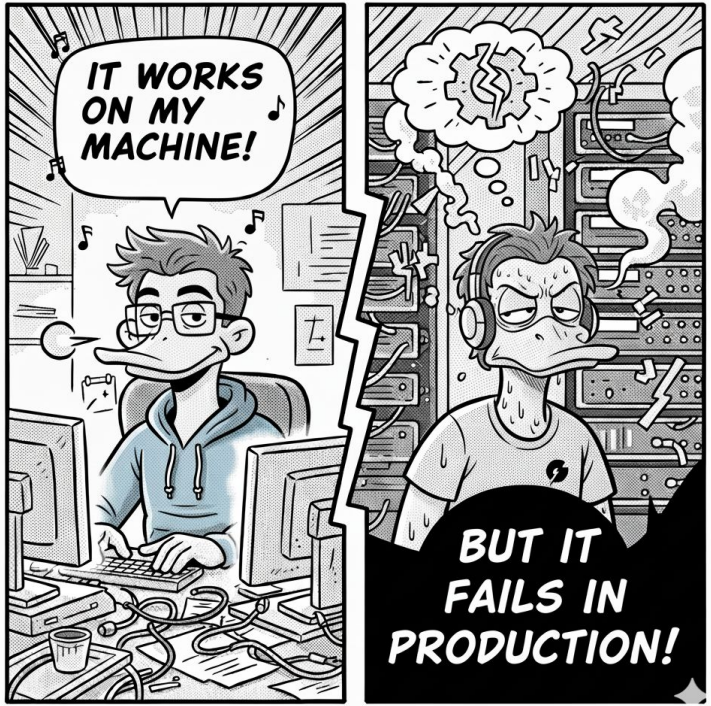

User Request / Display Data
API Request / JSON Data
Web Interface
User selects movies
JSON Data
Flask API
ML recommendations
ead/Write Data
SQL Queries
Database
Movies, ratings, users

Jupyter ≠ Production

Notebooks are for exploration, not deployment

Real-world ML needs:

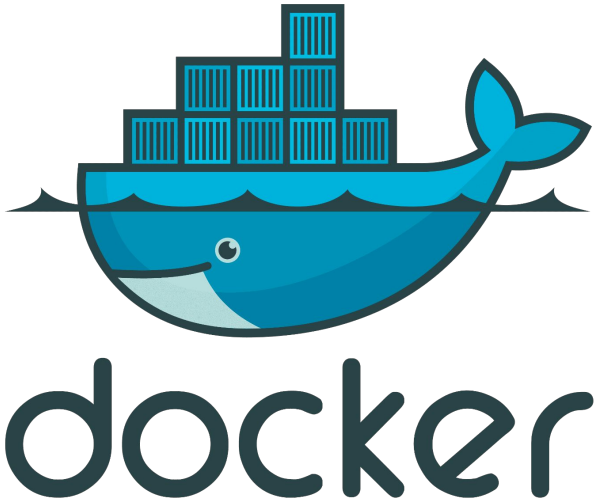
- **Reproducibility** (same code runs everywhere)
- **Scalability** (serve many users, handle large requests) -> Rest APIs (Flask/FastAPI), load balancing, cloud deployment (GCP)
- **Maintainability** (collaboration, updates, bug fixes)



How would make sure your model behaves as it should?

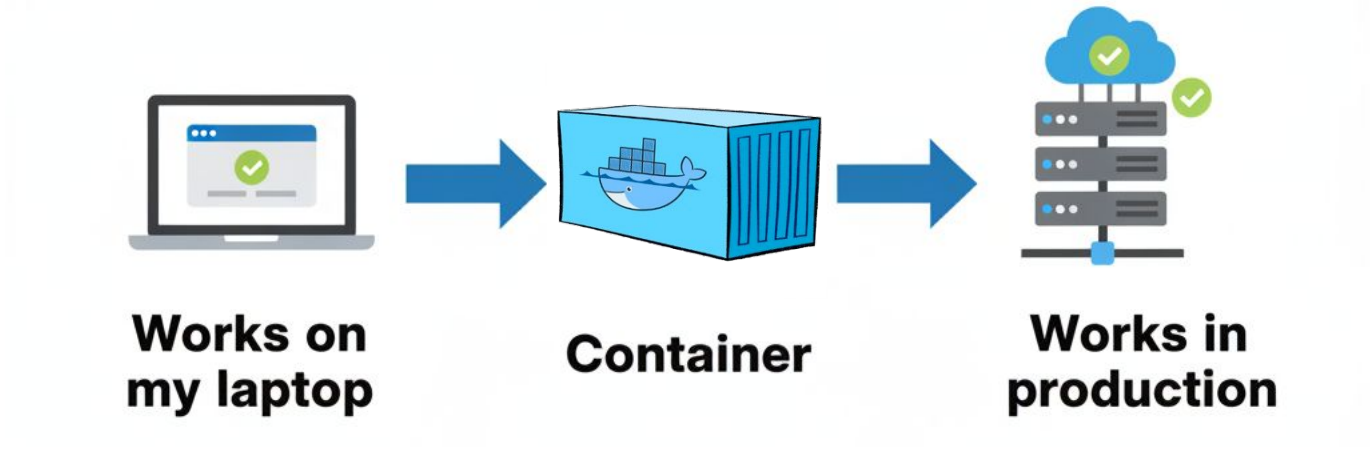
You will now deploy your model as a REST API.

How to make sure it works on the server as on your machine?
Think in terms of dependences.
How to scale easily?
What would you provide to the production team?


docker

Why Docker?

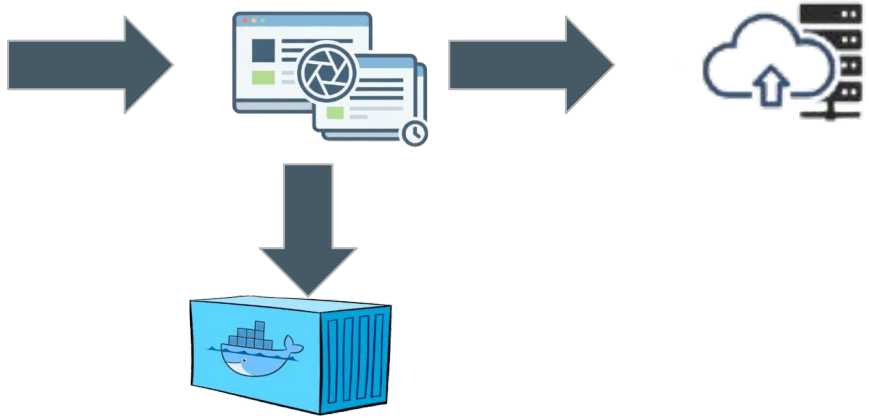
- **Reproducibility**: avoid “it works on my machine”
- **Portability**: build once, run anywhere
- **Isolation**: projects don’t conflict
- **Scalability**: run multiple containers in parallel



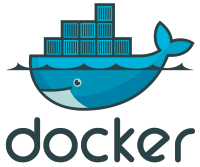
Key Docker Terminology

- **Image**: snapshot of environment (code + libs)
- **Container**: running instance of an image
- **Dockerfile**: recipe to build an image
- **Registry**: place to store/share images (e.g., Docker Hub)

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["python", "app.py"]
```

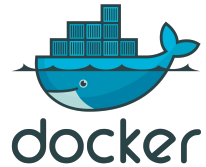


Docker Workflow



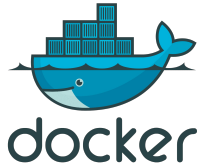
- Write Dockerfile
- Build image
- Run container
- Push image to registry
- Deploy (pull image on server/cluster and run containers)

Anatomy of a Dockerfile




```
FROM python:3.9-slim
WORKDIR /app
RUN apt-get update && apt-get install -y gcc
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY model.pkl .
COPY app.py .
EXPOSE 8000
CMD ["uvicorn", "app:app", "--host", "0.0.0.0"]
```

Building & Running Containers



- **Build and image:**

```
docker build -t my-ml-model:v1 .
```

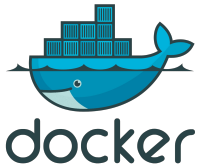
- **Run a container:**

```
docker run -p 8000:8000 my-ml-model:v1
```

- **Useful commands**

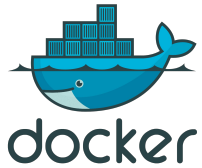
```
docker ps
docker images
docker logs container-id
docker exec -it container-id bash
```

How Docker Works (Under the Hood)



- **Containers share the host OS kernel (vs. VMs that run a full OS)**
- **Each container has:**
  - isolated filesystem
  - processes
  - networking
- **Images are layered for efficient caching**
- **Docker Engine orchestrates everything**

Docker Layers & Caching



- Each Dockerfile instruction = new layer
- Layers are cached and reused
- When something does change, all following layers get rebuilt.

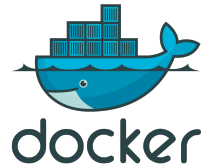
```
FROM python:3.9-slim
WORKDIR /app
RUN apt-get update && apt-get install -y gcc
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY model.pkl .
COPY app.py .
EXPOSE 8000
CMD ["uvicorn", "app:app", "--host", "0.0.0.0"]
```

Docker Layers & Caching

```
FROM python:3.9-slim
WORKDIR /app

RUN apt-get update && apt-get install -y gcc

COPY . .

# Install dependencies after copying all files
RUN pip install -r requirements.txt

EXPOSE 8000
CMD ["uvicorn", "app:app", "--host", "0.0.0.0"]
```

```
FROM python:3.9-slim
WORKDIR /app

RUN apt-get update && apt-get install -y gcc

# Copy only requirements first
COPY requirements.txt .

RUN pip install -r requirements.txt

# Now copy rest of the project
COPY model.pkl .
COPY app.py .

EXPOSE 8000
CMD ["uvicorn", "app:app", "--host", "0.0.0.0"]
```

Managing Large ML Models



✅ Solution 1: Volumes

```
# Mount model directory
docker run -v /host/models:/app/models myapi
# In your app
model = load_model('/app/models/model.pkl')
```

✓ Fast deployment
✓ Easy model updates
✓ Multiple model versions

📊 Size Comparison

Image with model:	2.5 GB
Image without model:	200 MB

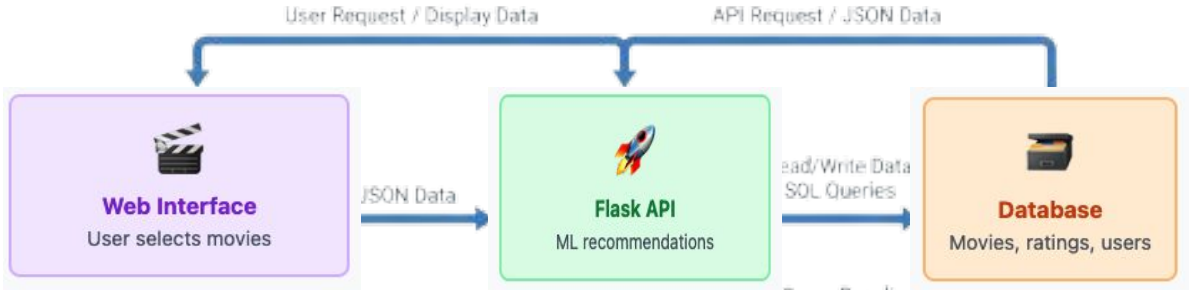
12x smaller! Faster builds, deploys, scaling.

✅ Solution 2: Cloud Storage

```
# Download at runtime
import boto3
s3 = boto3.client('s3')
s3.download_file(
'my-bucket', 'models/v2.pkl',
'/tmp/model.pkl'
)
```

✓ Centralized storage
✓ Version control
✓ Scalable

Docker compose



web/Dockerfile (React)

dockerfile

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

api/Dockerfile (Flask)

dockerfile

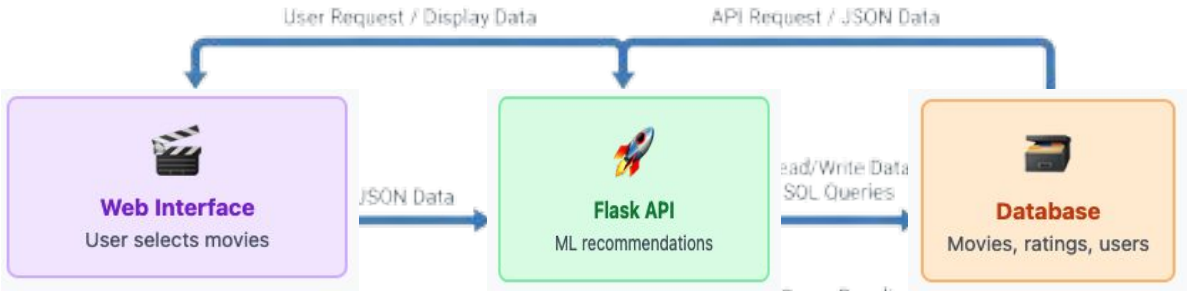
```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["python", "app.py"]
```

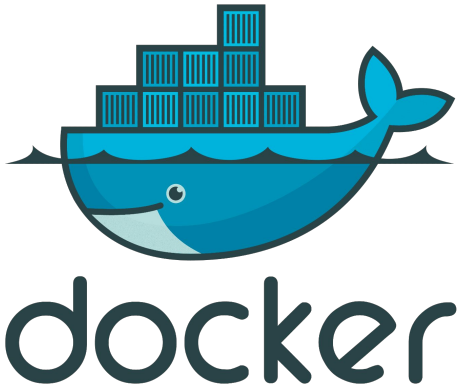
db/Dockerfile (Redis)

dockerfile

```
FROM redis:7-alpine
EXPOSE 6379
CMD ["redis-server", "--appendonly", "yes"]
```

Docker compose:





docker-compose.yml

yaml

```yaml
version: "3.9"
services:
  web:
    build: ./web
    ports:
      - "3000:3000"
    depends_on:
      - api

  api:
    build: ./api
    environment:
      REDIS_HOST: db
      REDIS_PORT: 6379
    ports:
      - "5000:5000"
    depends_on:
      - db

  db:
    build: ./db
    ports:
      - "6379:6379"
```

Jupyter ≠ Production

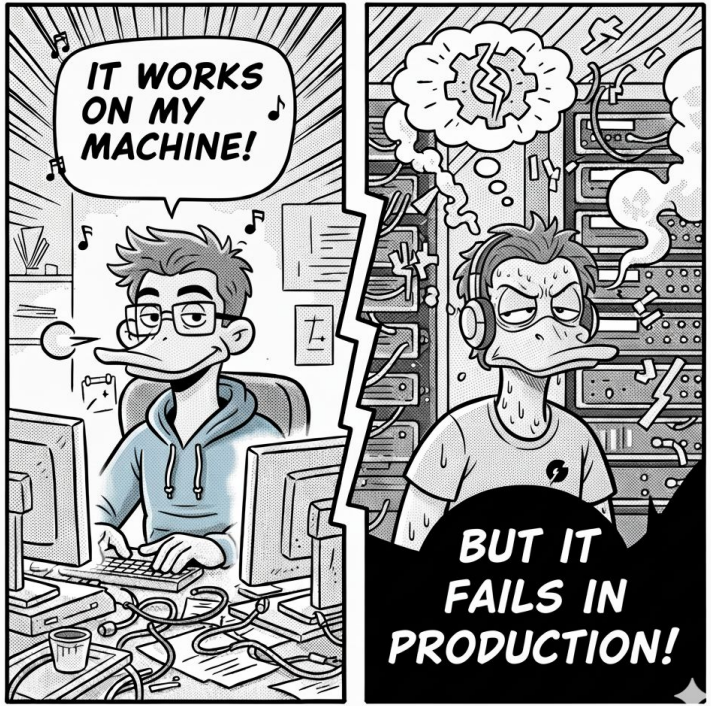
Notebooks are for exploration, not deployment

Real-world ML needs:

- **Reproducibility** (same code runs everywhere)
  -> Containers (Docker)

- **Scalability** (serve many users, handle large requests)
  -> Rest APIs (Flask/FastAPI), load balancing, cloud deployment (GCP)

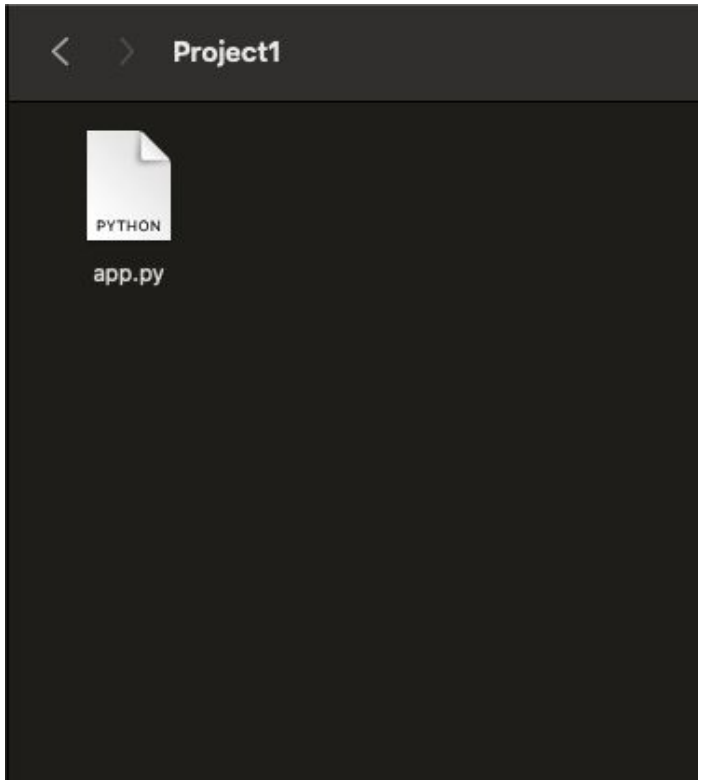
- **Maintainability** (collaboration, updates, bug fixes)



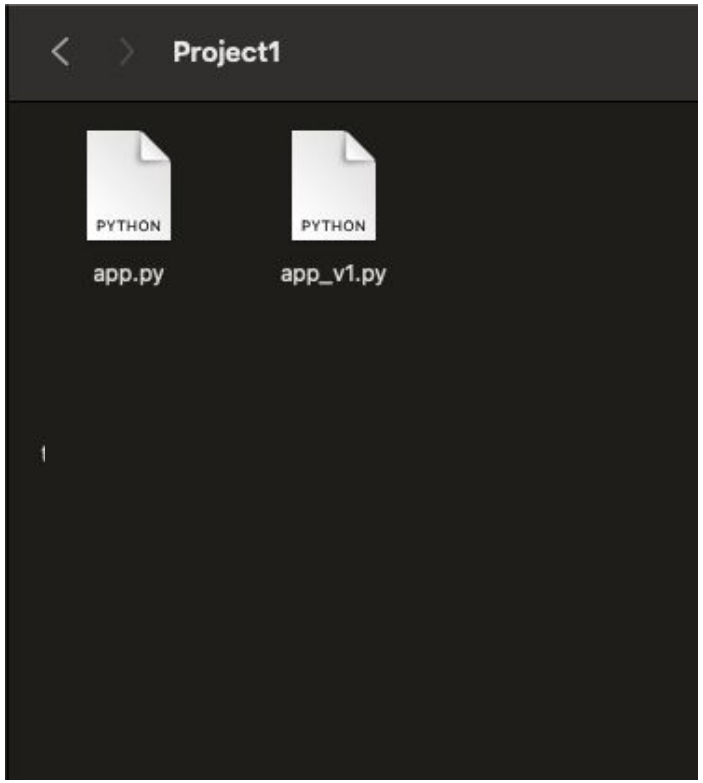

git

Version control systems and Git

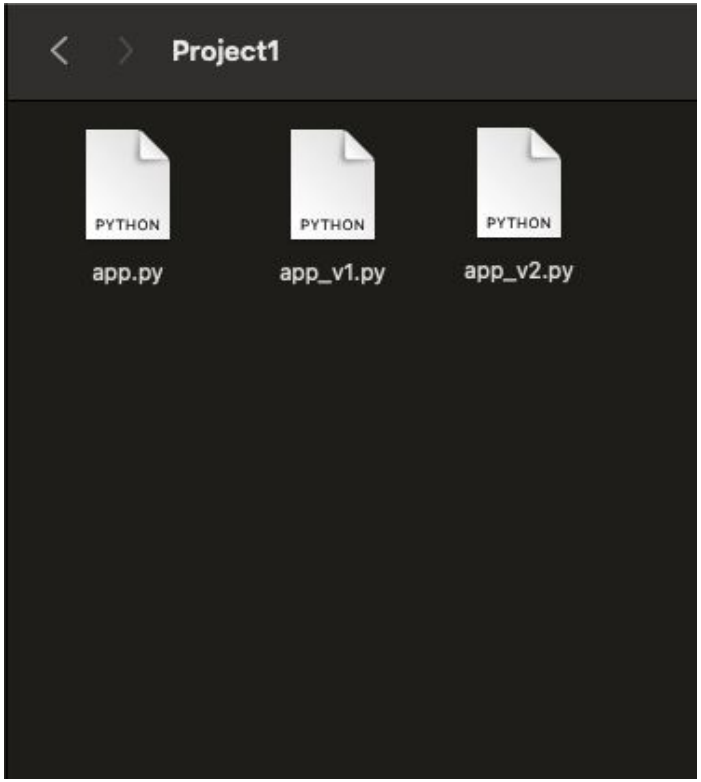
The typical young developer version control systems



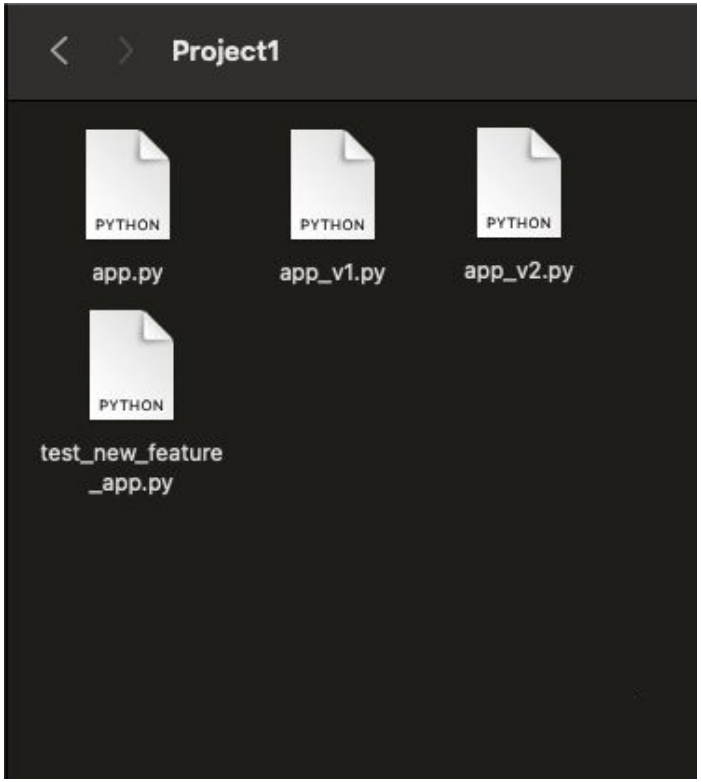
The typical young developer version control systems



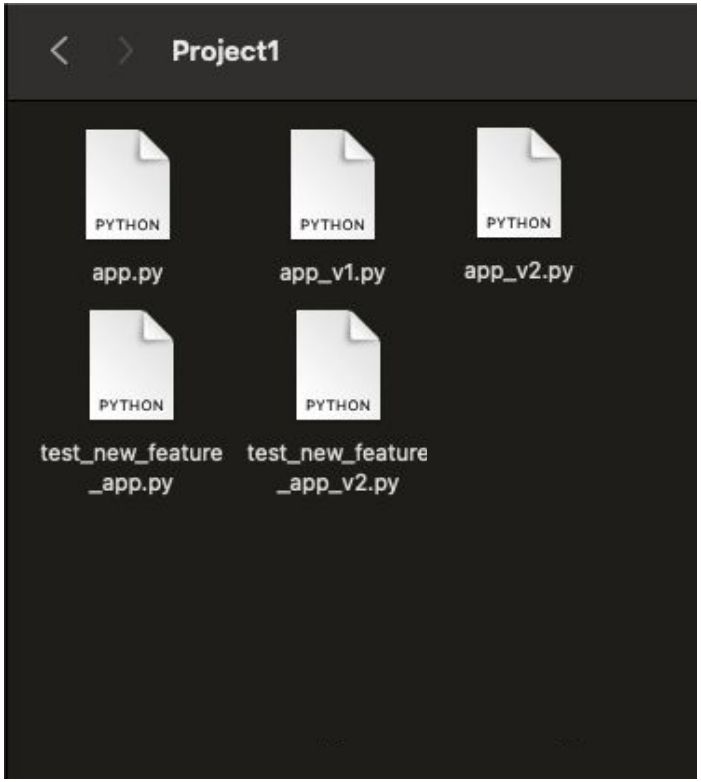
The typical young developer version control systems



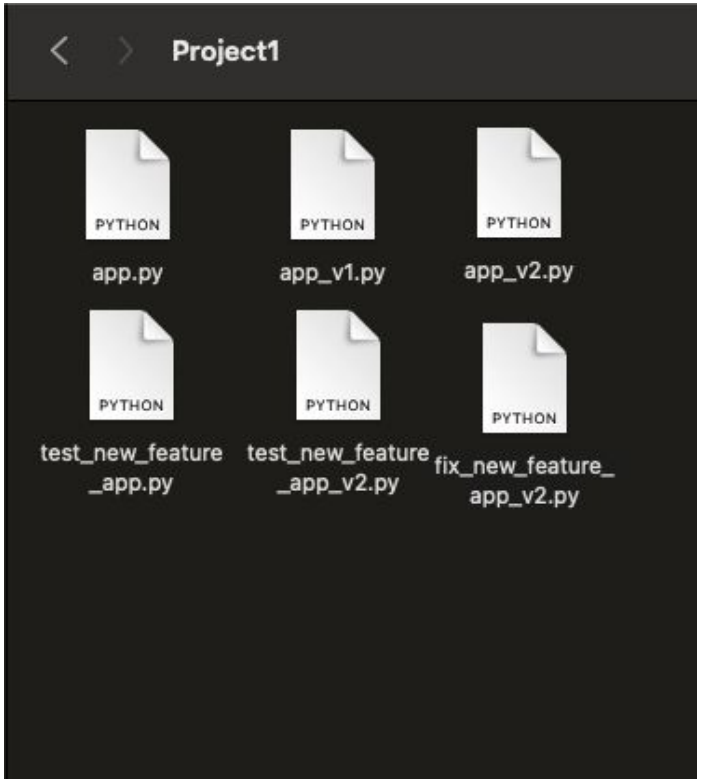
The typical young developer version control systems



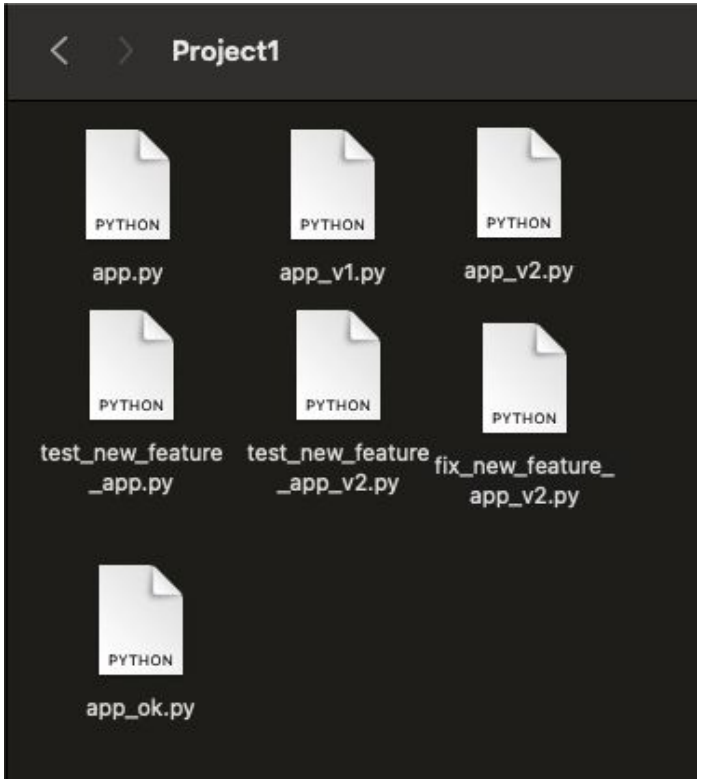
The typical young developer version control systems



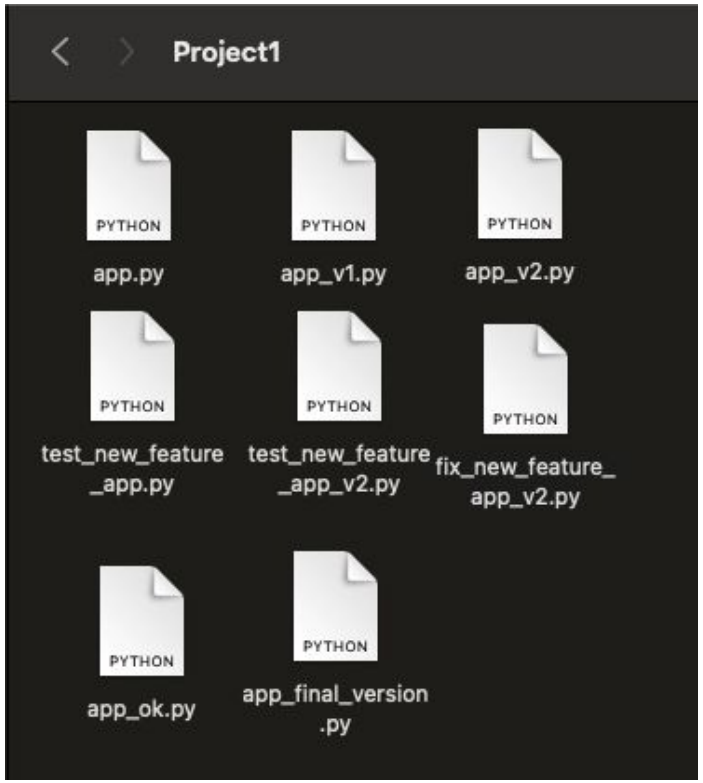
The typical young developer version control systems



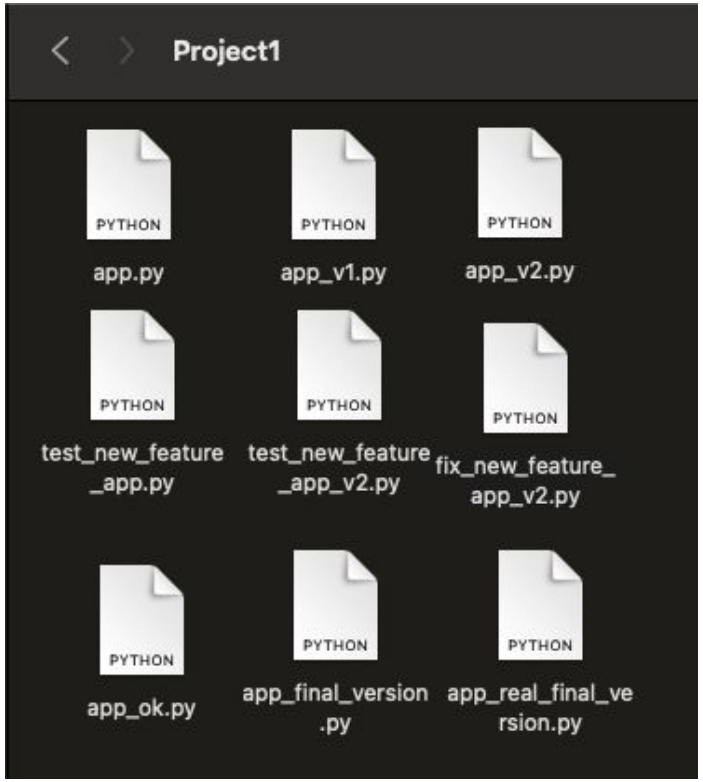
The typical young developer version control systems



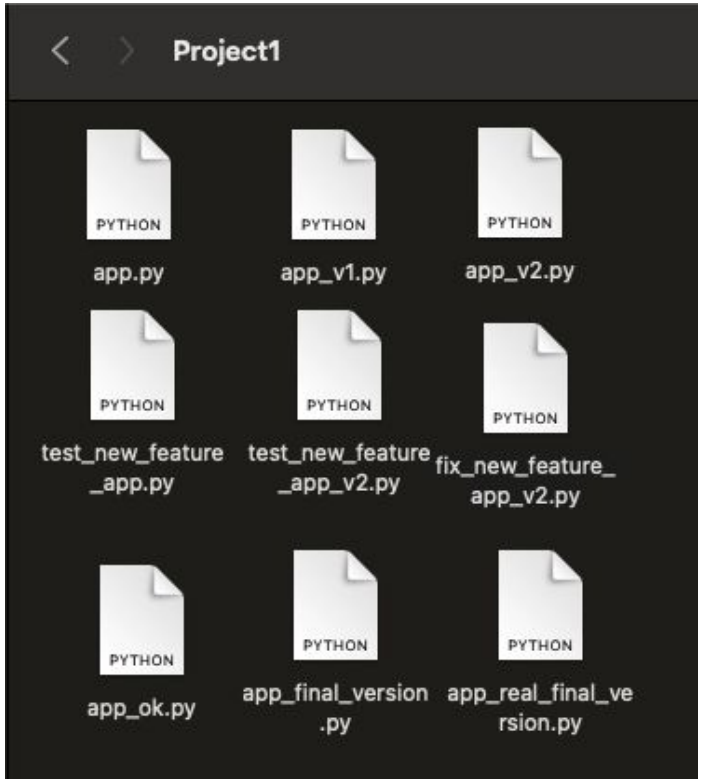
The typical young developer version control systems



The typical young developer version control systems

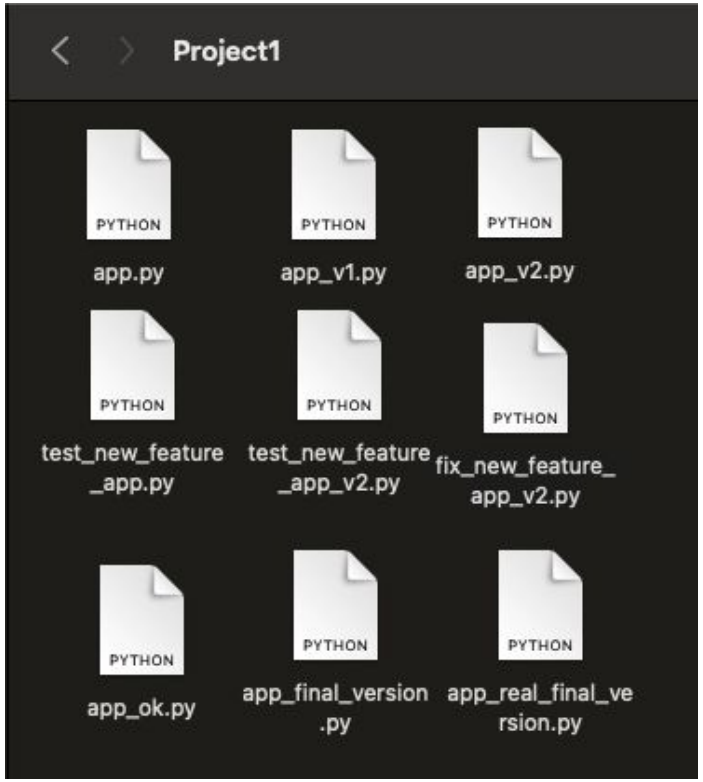


The typical young developer version control systems



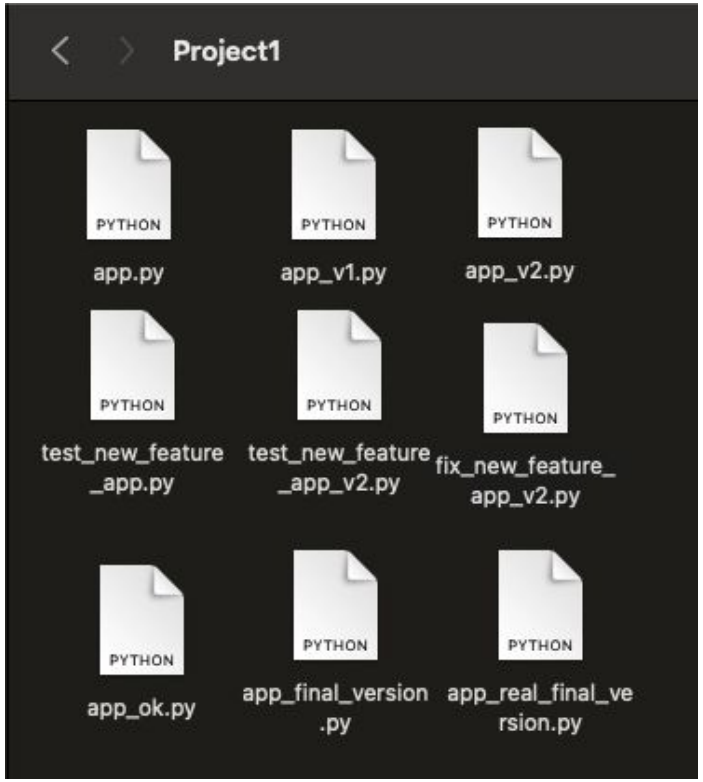
- Hard to compare changes over time

The typical young developer version control systems



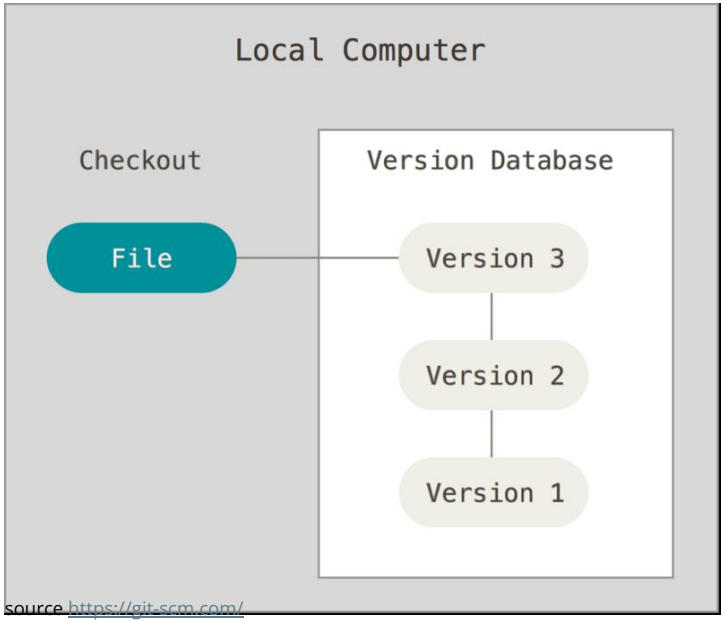
- Hard to compare changes over time
- Hard to remember what each version does

The typical young developer version control systems



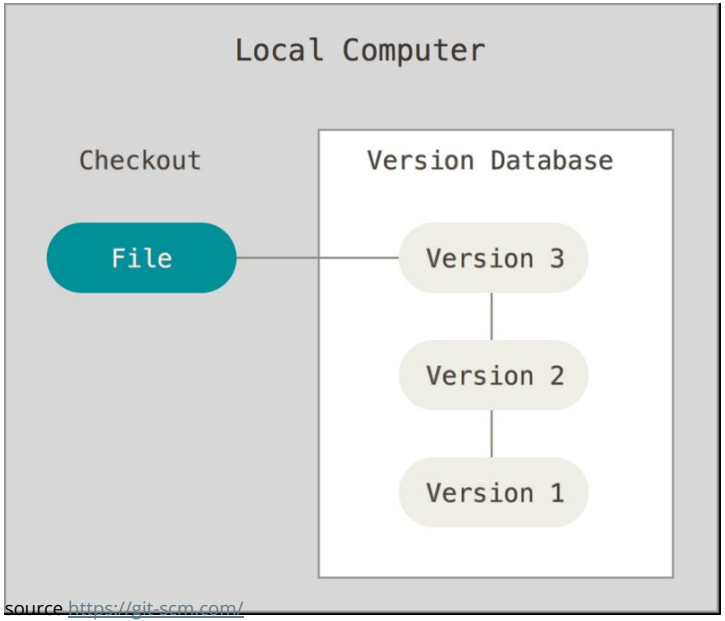
- Hard to compare changes over time
- Hard to remember what each version does
- Hard to revert to a desired state

Local Version Control Systems



source https://git-scm.com/

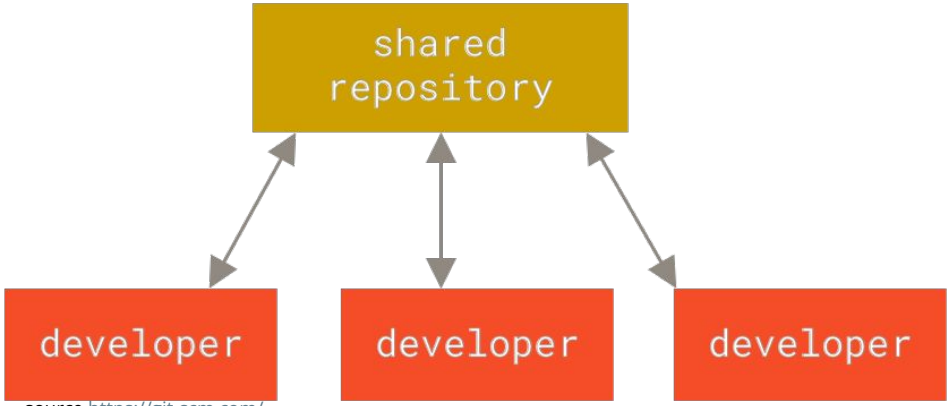
Local Version Control Systems



source https://git-scm.com/

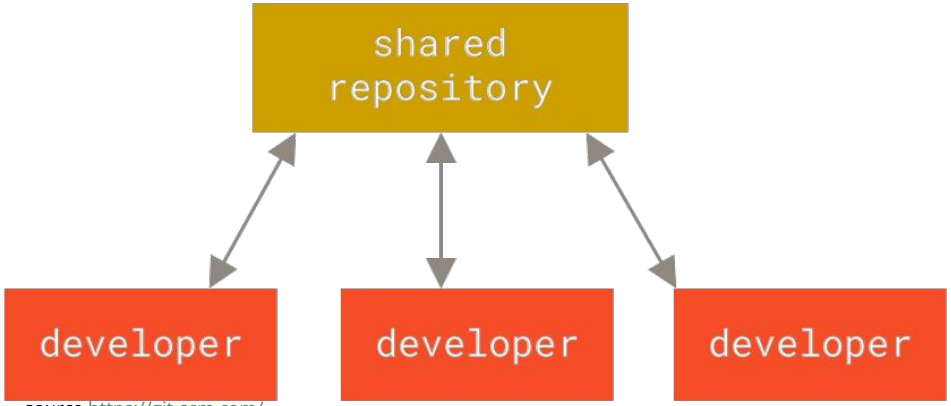
- How to work in team?
- What if my disk get corrupted?

Centralized Version Control Systems



source https://git-scm.com/

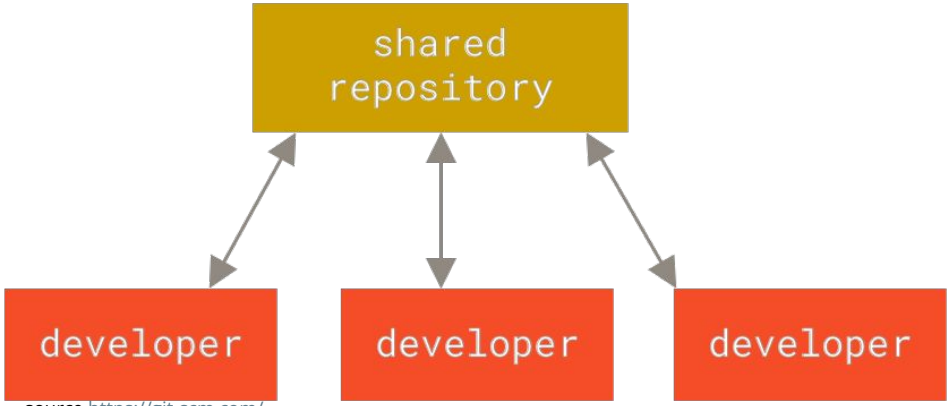
Centralized Version Control Systems



source https://git-scm.com/

- Everyone knows (to a certain degree) what other people are doing in the project

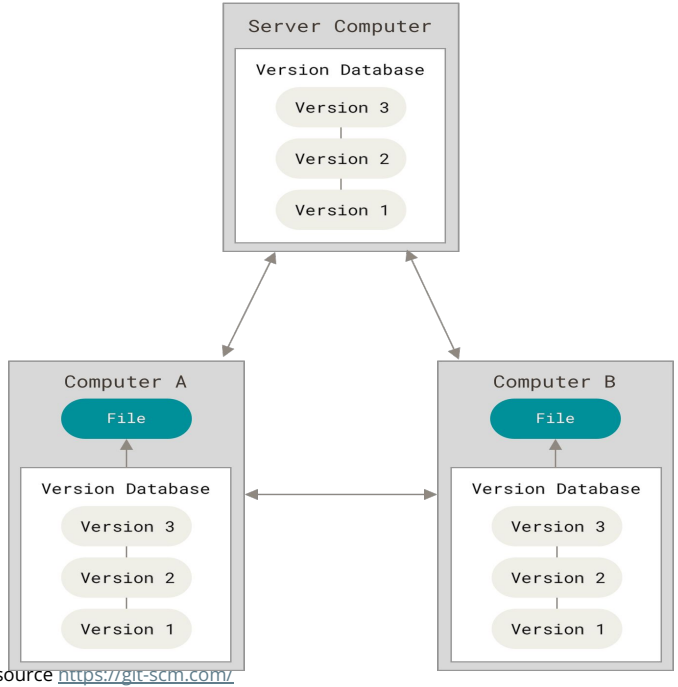
Centralized Version Control Systems



source https://git-scm.com/

- Everyone knows (to a certain degree) what other people are doing in the project
- Single point of failure

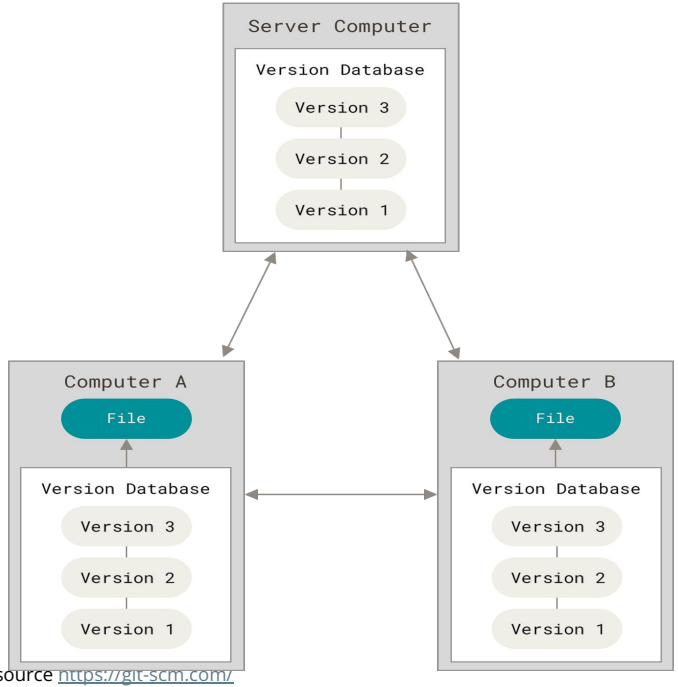
Distributed Version Control Systems



source https://git-scm.com/

- Client checks out the entire codebase history instead of latest snapshot
- Every clone is a full backup

Distributed Version Control Systems



source https://git-scm.com/

- Client checks out the entire codebase history instead of latest snapshot
- Every clone is a full backup
- Each client can be used to restore if the server fails
- Easy to collaborate
- Can deal with different remote repositories and different teams

Git

- Distributed Version Control Systems
- Created in 2005 by the Linux development community (and in particular Linus Torvalds)
- Aiming for the following goals:
  - Speed
  - Simple design
  - Support for non linear development (branching)
  - Fully distributed
  - Able to handle large projects efficiently (like Linux kernel)

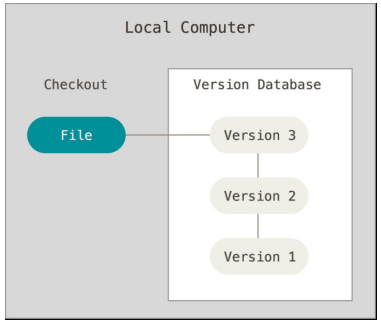
Git ≠ GitHub

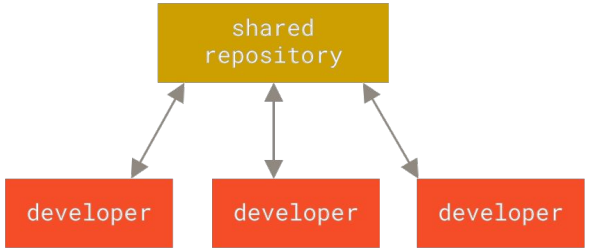
- Distributed Version Control Systems

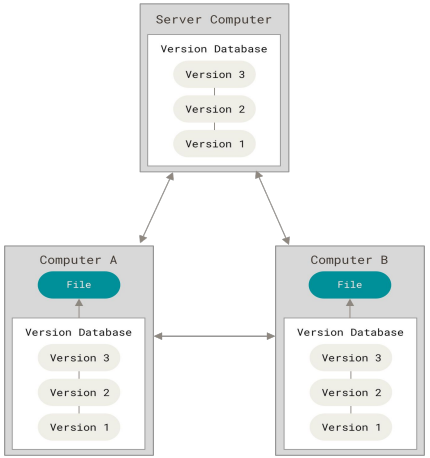
- Platform that allows developers to create, store, manage and share their code using Git

Previousy

- Version control systems



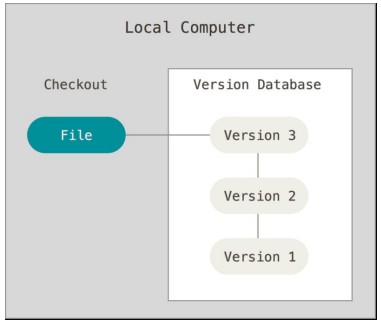


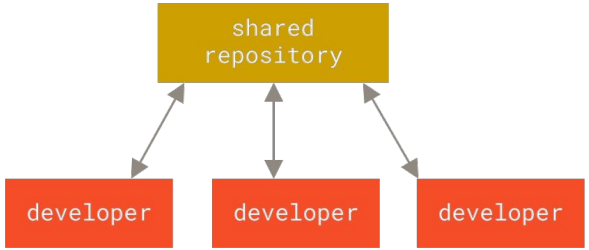


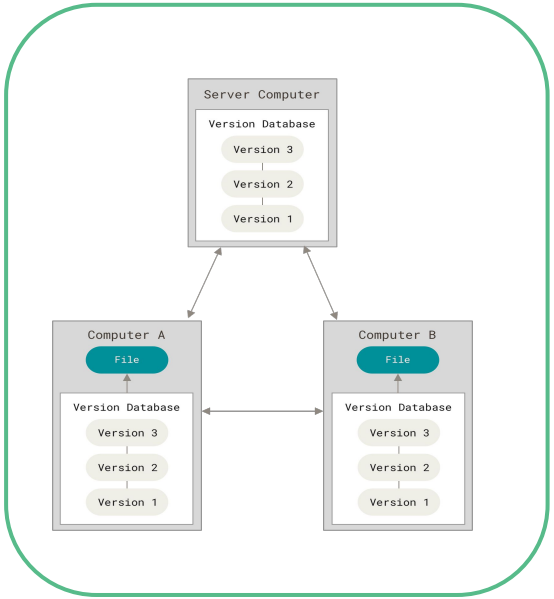
source https://git-scm.com/

Previousy

- Version control systems



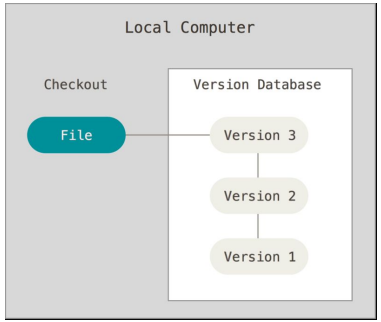


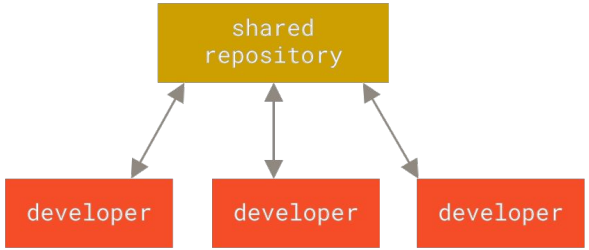


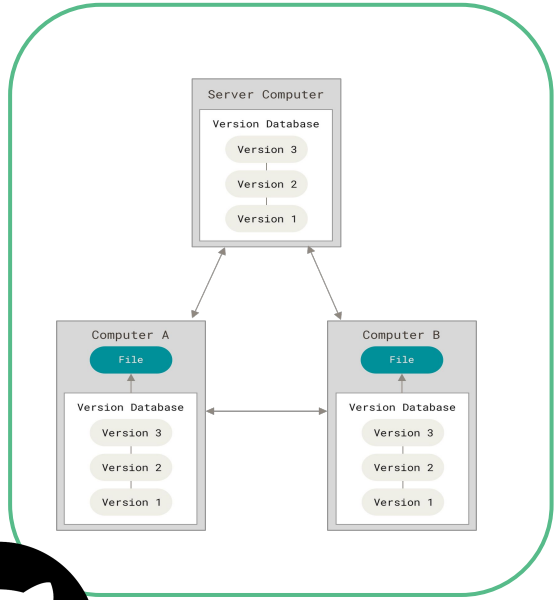
source https://git-scm.com/

Previousy

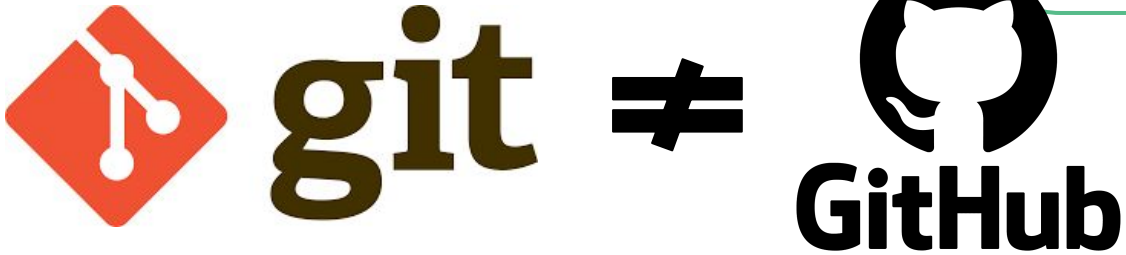
- Version control systems







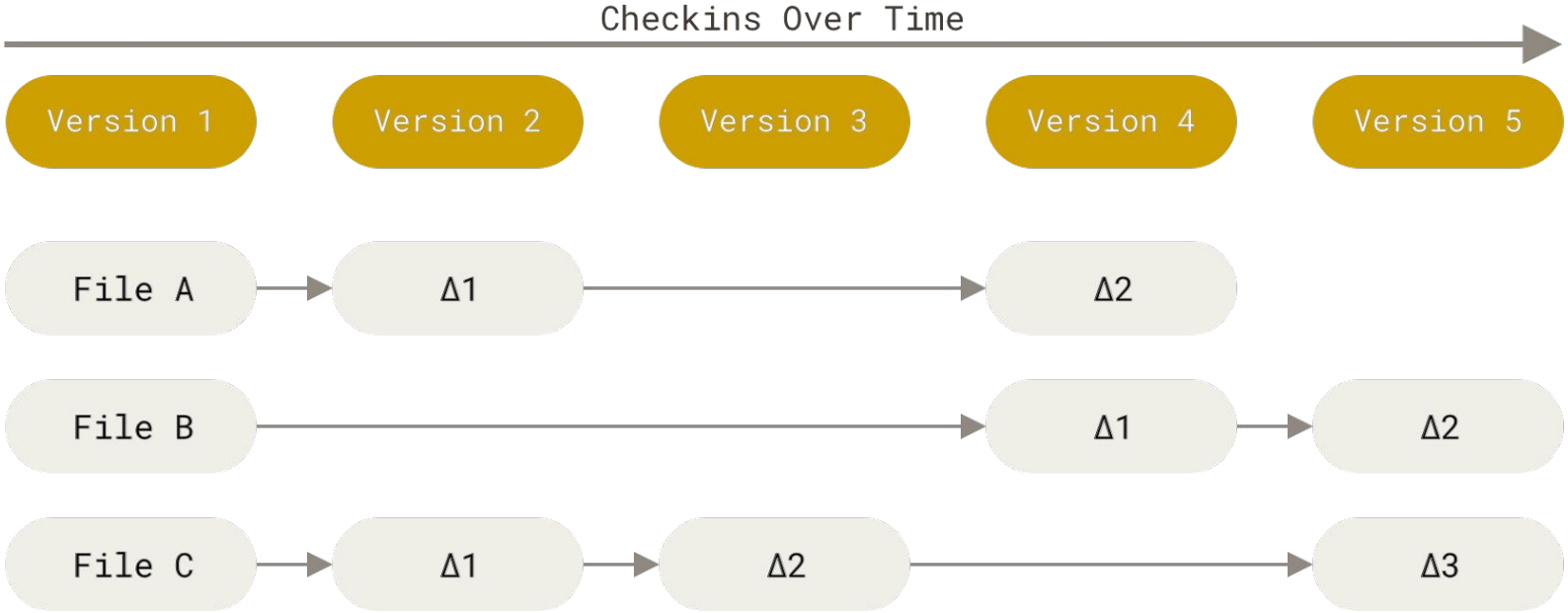
source https://git-scm.com/



Git’s basic principles

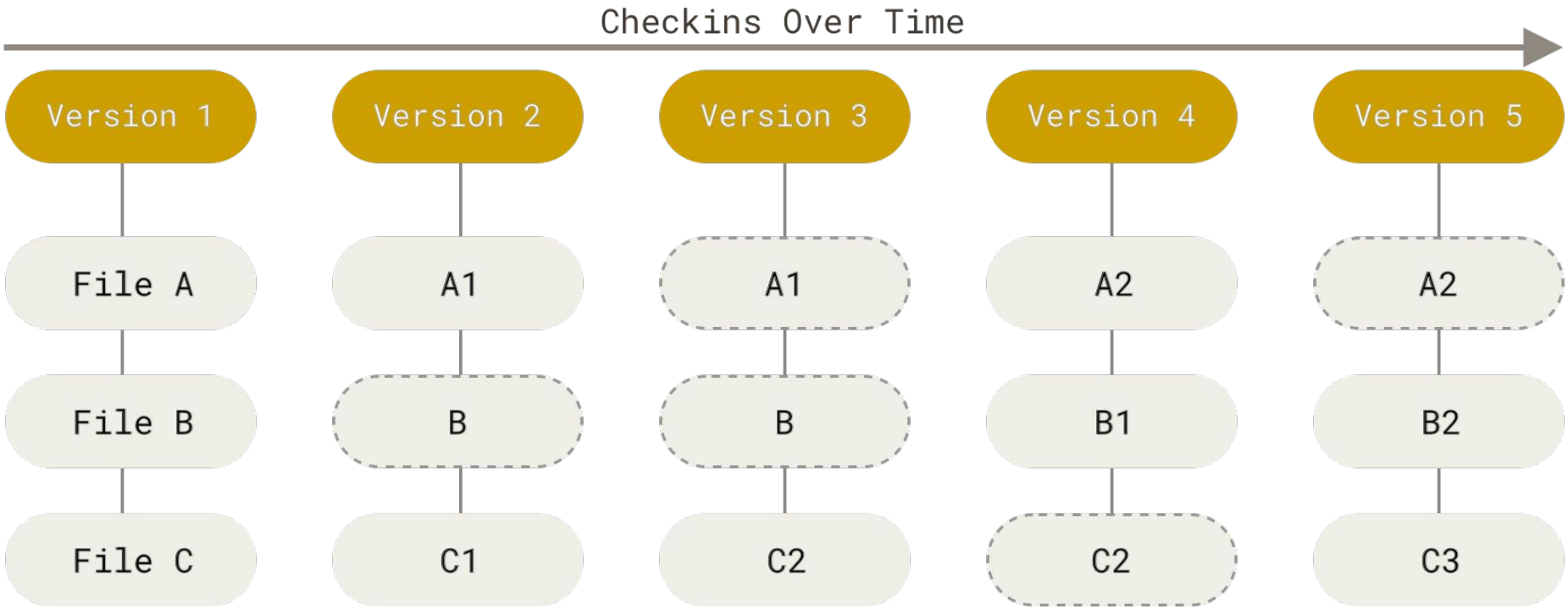
Traditional VCS

- Delta-based version control



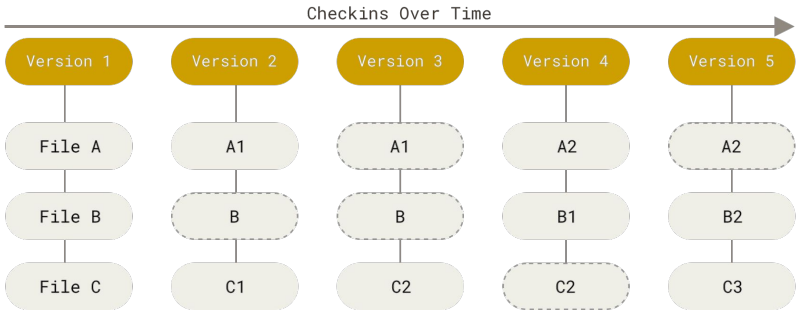
source https://git-scm.com/

Git's commits



source https://git-scm.com/

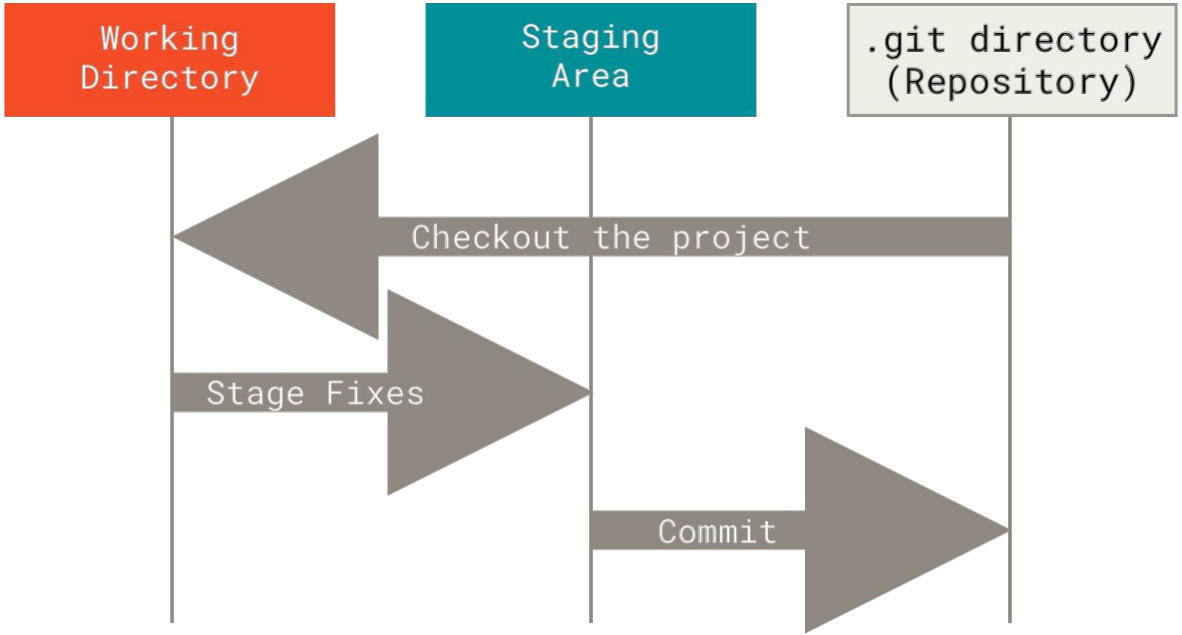
Git's commits



source https://git-scm.com/

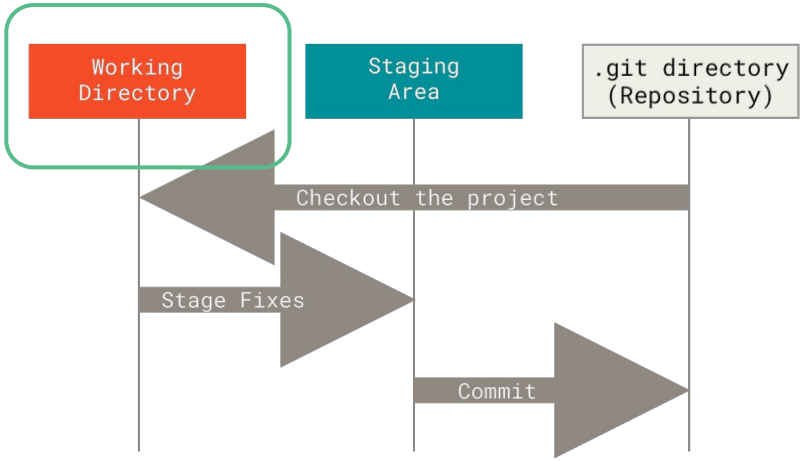
- Snapshot of a mini filesystem
- Store references to these snapshots
- Makes it extremely powerful for branching
- Almost every operations are local
- Integrity checksum mechanisms
  => impossible to modify a file without Git knowing it
- Generally only adds data

Git 3 sections



source https://git-scm.com/

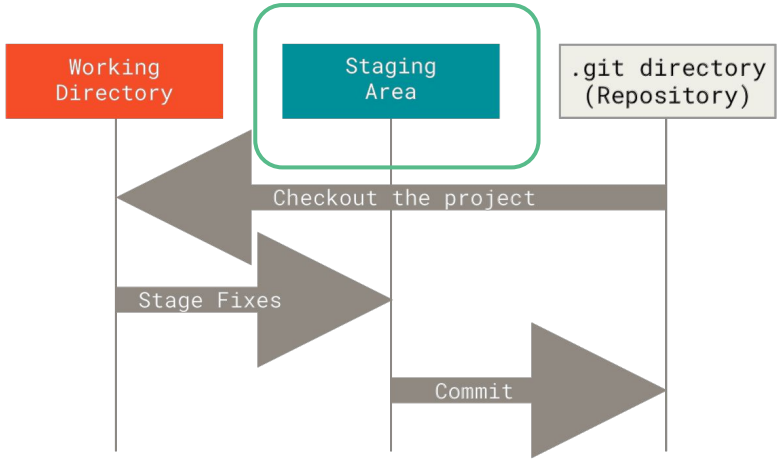
The Working Directory



- Single checkout of one version of the project
- Files are pulled out of the Git directory and placed on disk
- You can use or modify these files

source https://git-scm.com/

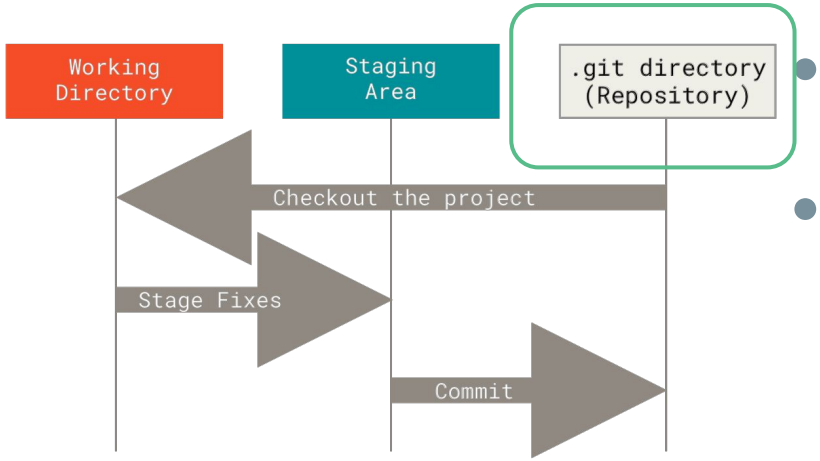
The Staging Area



source https://git-scm.com/

- Single file, usually contained in Git directory
- Stores information about what will go into the next commit
- Sometimes referred as “**index**”

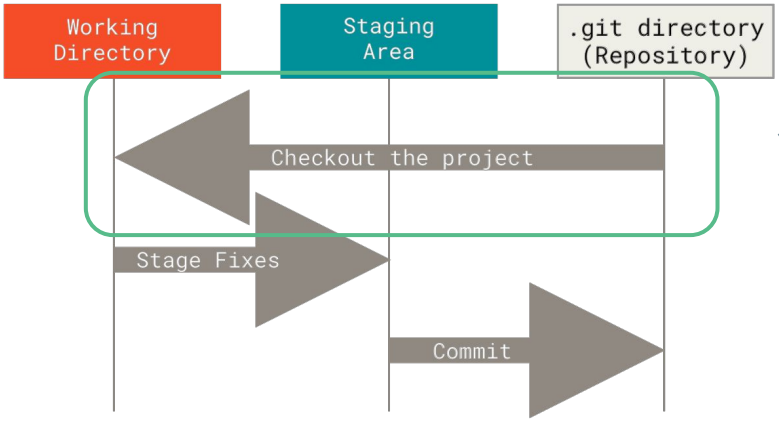
The Git Directory



- Where Git stores the metadata and object database for your project.
- Is what you copy when doing a **git clone**

source https://git-scm.com/

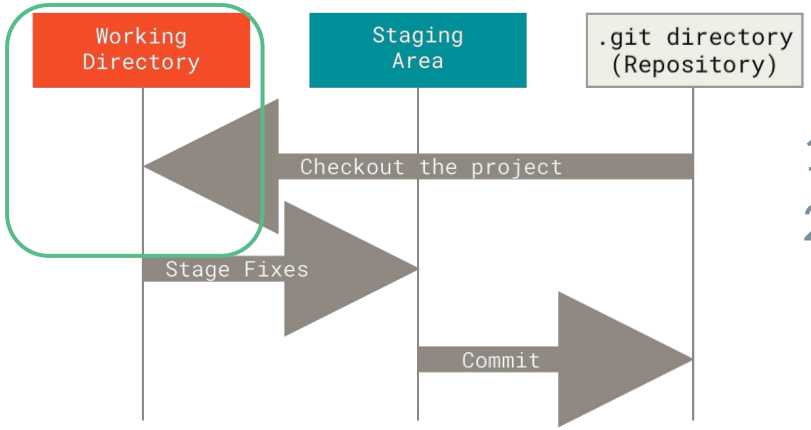
Typical workflow



1. You checkout from a specific commit

source https://git-scm.com/

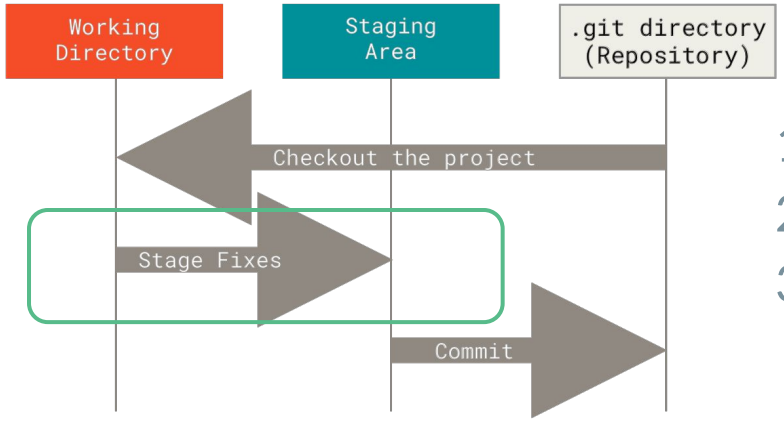
Typical workflow



1. You checkout from a specific commit
2. You modify/create some files

source https://git-scm.com/

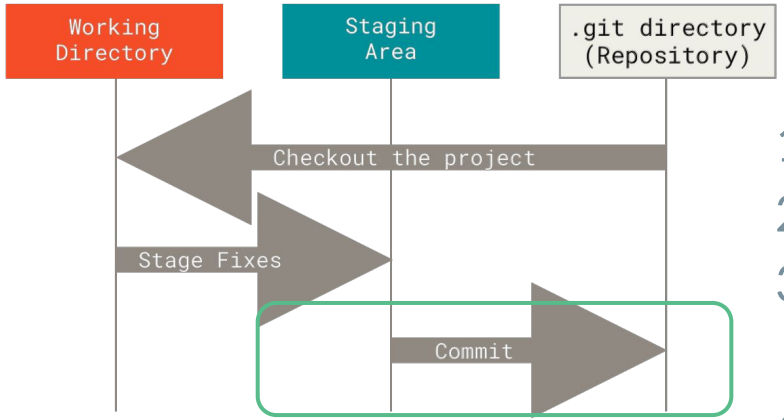
Typical workflow



1. You checkout from a specific commit
2. You modify/create some files
3. You add to the staging areas the changes you want to be part of the next commit

source https://git-scm.com/

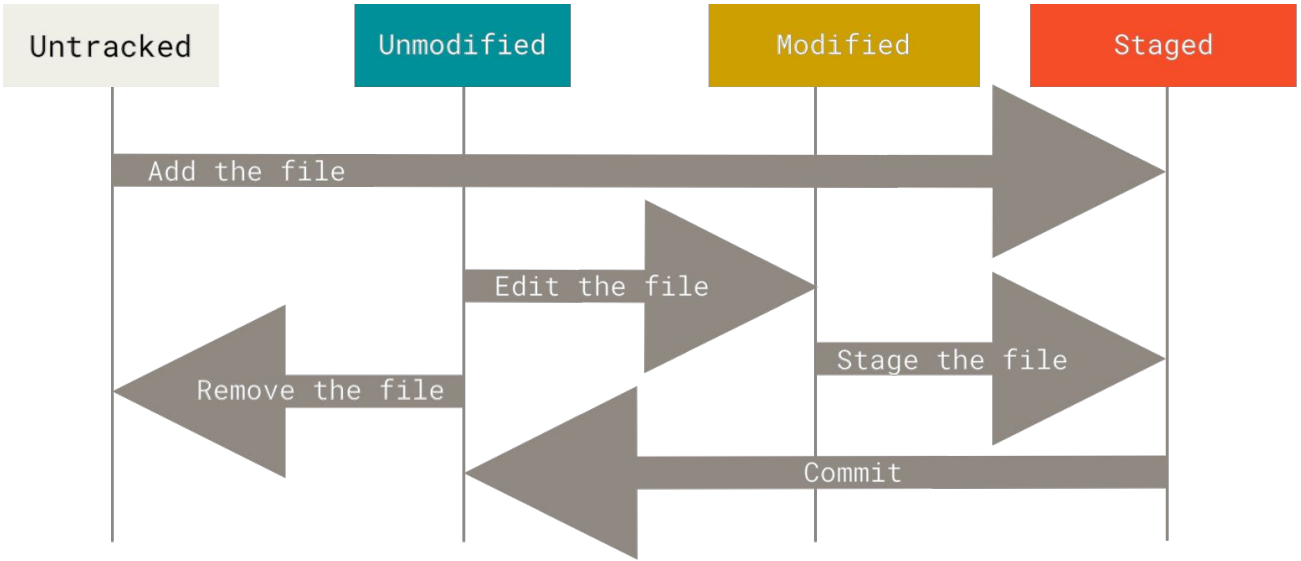
Typical workflow



source https://git-scm.com/

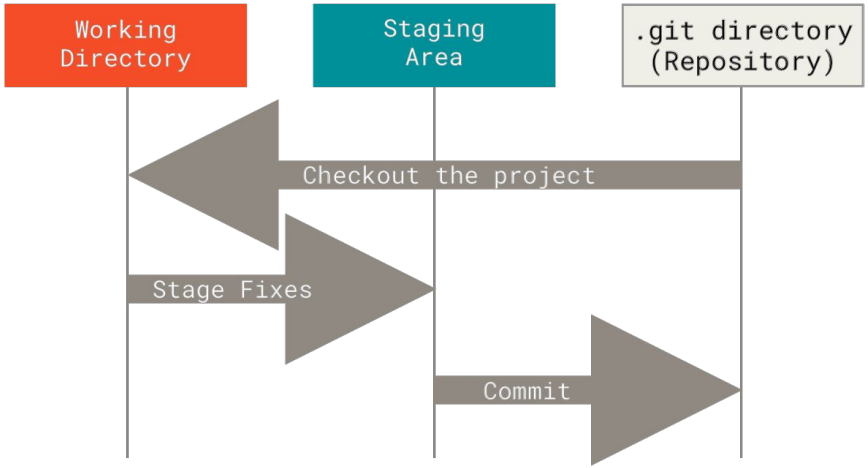
1. You checkout from a specific commit
2. You modify/create some files
3. You add to the staging areas the changes you want to be part of the next commit
4. You commit these changes and git stores them in the git diectory

Git files 4 states



source https://git-scm.com/

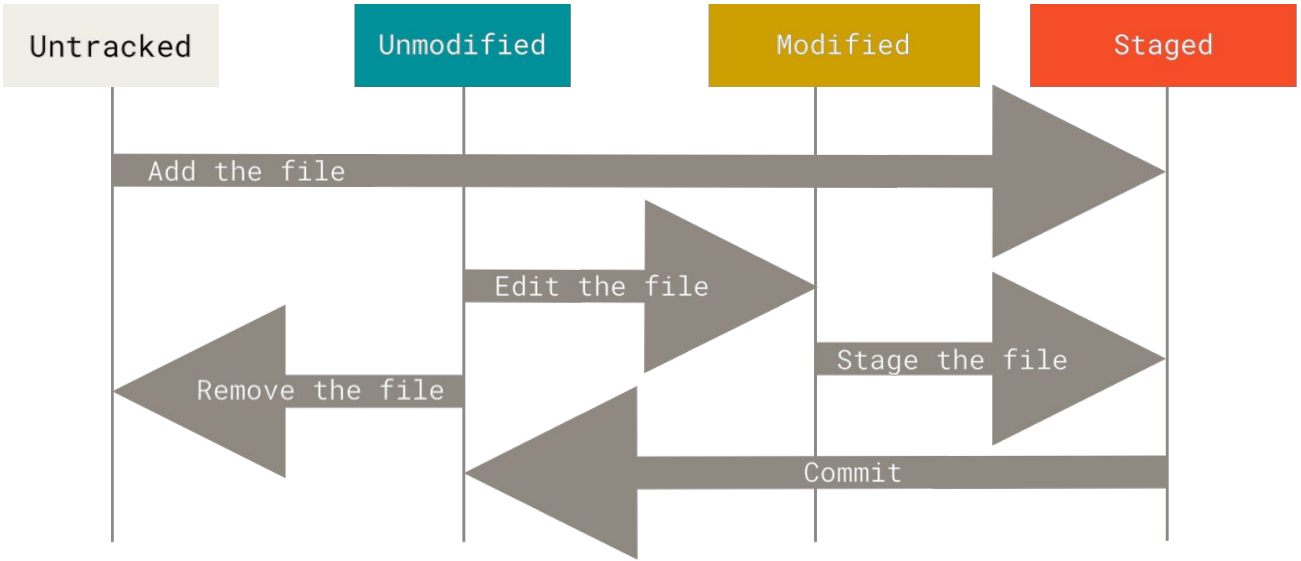
Git files 4 states



source https://git-scm.com/

- Untracked
- Unmodified
- Modified
- Staged

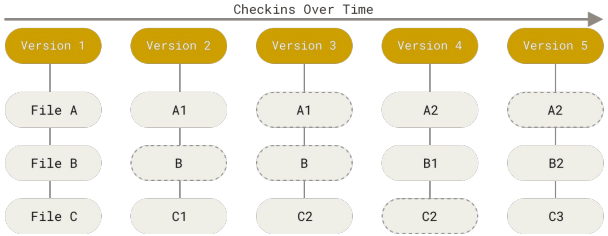
Git files 4 states

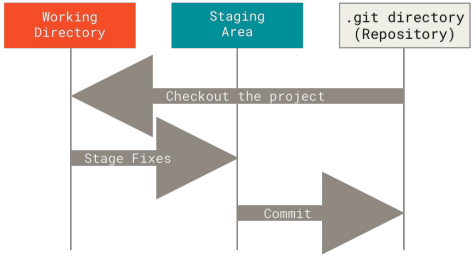


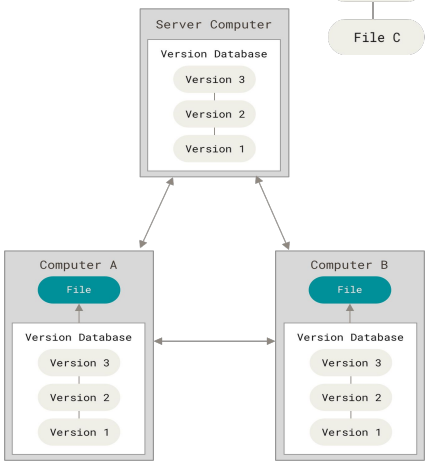
source https://git-scm.com/

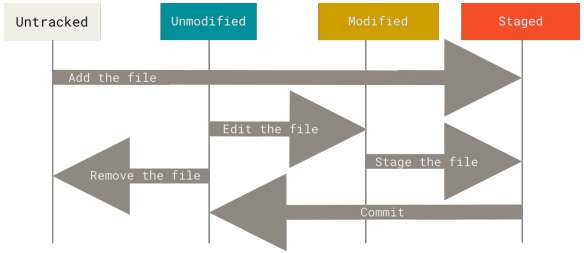
Previously

- Version control systems and Git
- Commits
- Working areas
- Four states









Initialize a repository

Two ways:

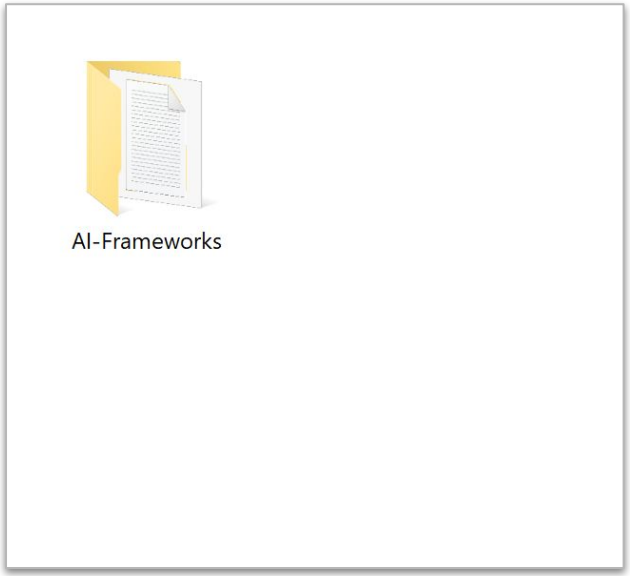
- Initialize a repository in an existing directory
- Clone an existing repository

Cloning an existing repository:

```
$ git clone https://github.com/DavidBert/AIF.git
```

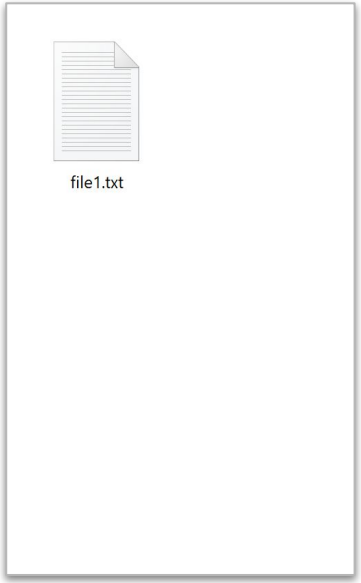
Cloning an existing repository:

```
$ git clone https://github.com/DavidBert/AIF.git
```



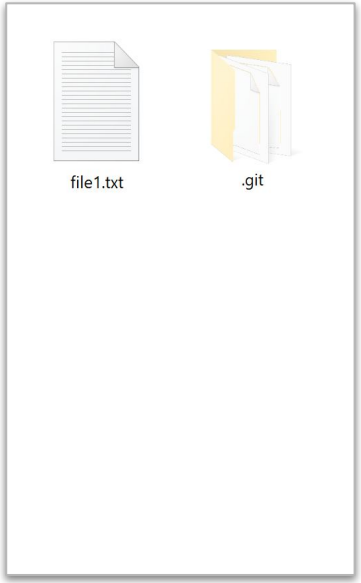
Initialize a repository from an existing directory:

```
$ git init
```

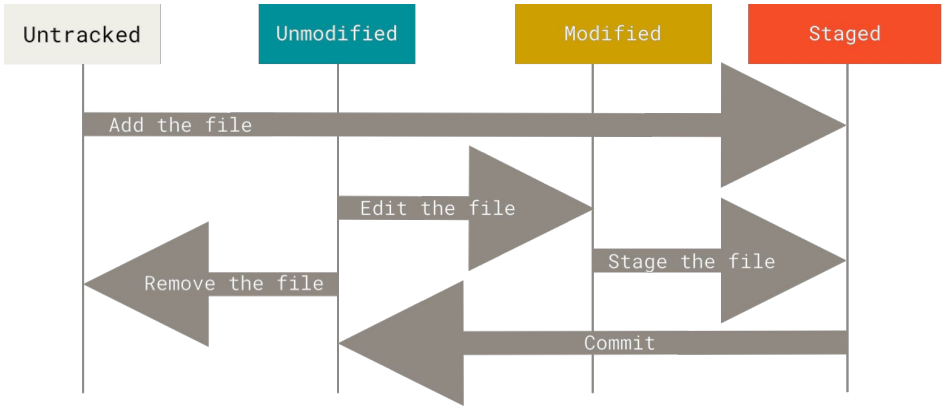


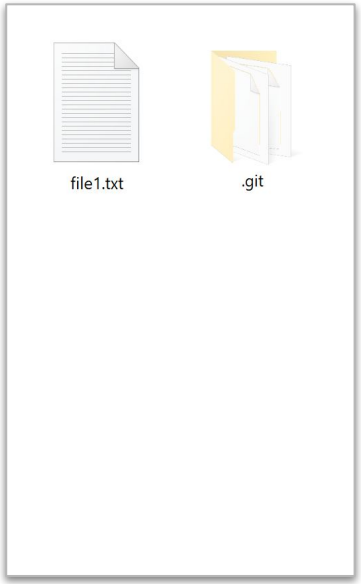
Initialize a repository from an existing directory:

```
$ git init
```



Check current status:

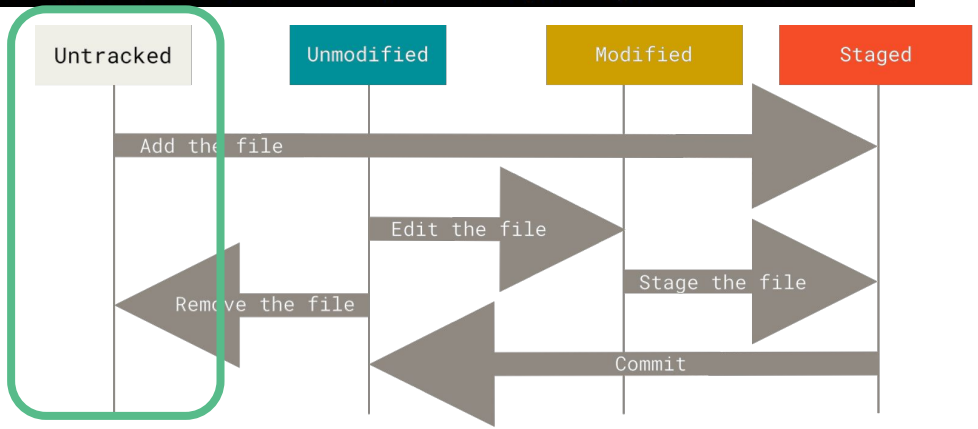


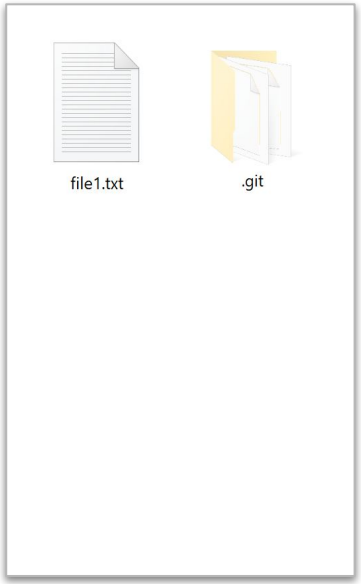


Check current status:

```
$ git status
```

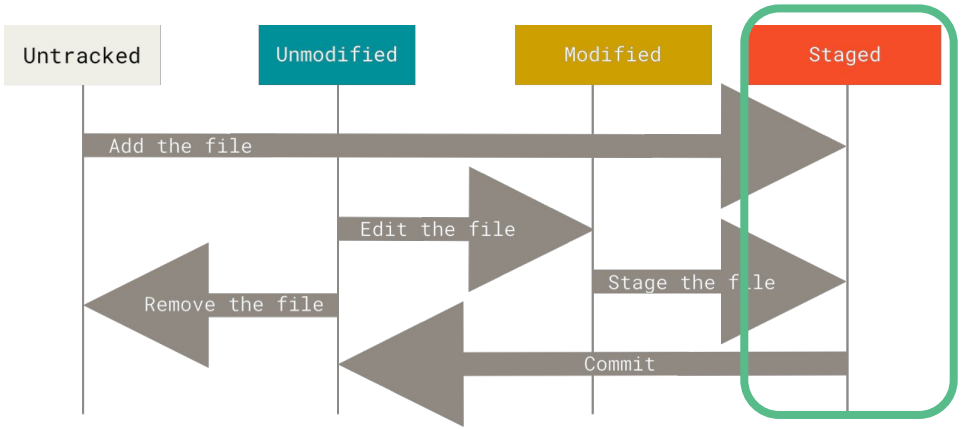
```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        file1.txt

nothing added to commit but untracked files present (use "git add" to track)
```

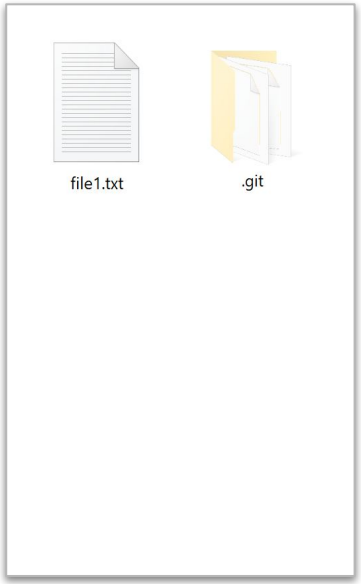




Tracking a file:

```
$git add file1.txt
```

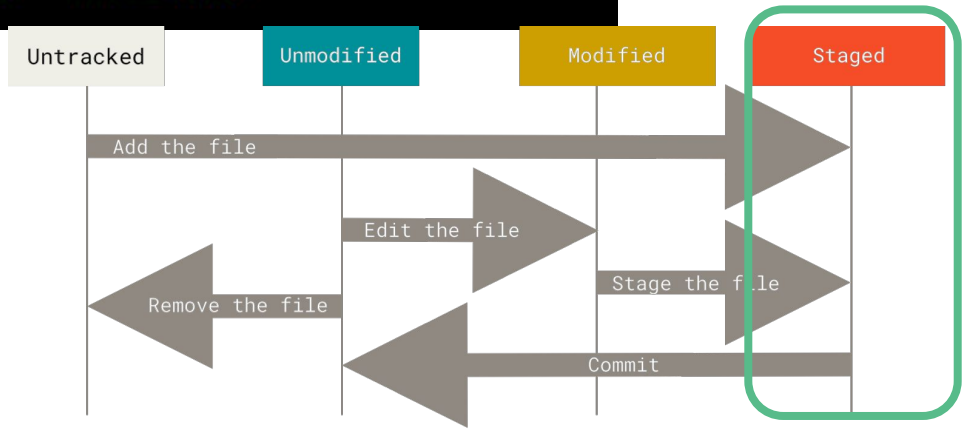


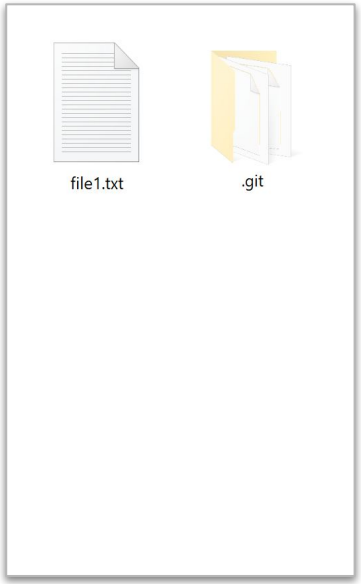


Tracking a file:

```
$git add file1.txt
$git status
```

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   file1.txt
```

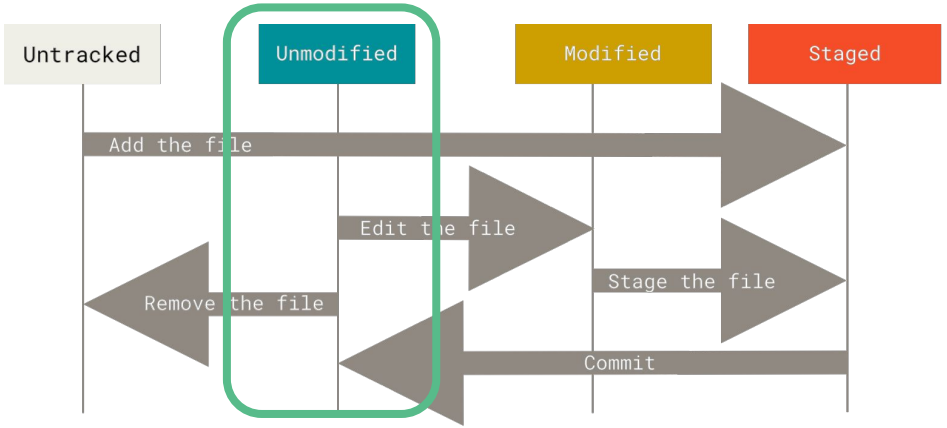


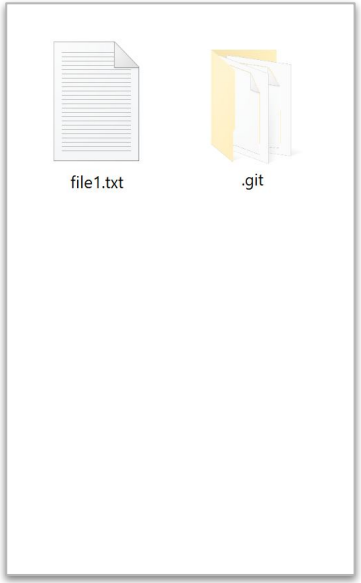


Commit changes:

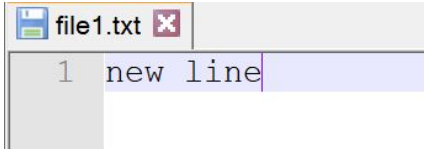
```
$git commit -m "first commit"
[master (root-commit) 43176a7] first commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt
```

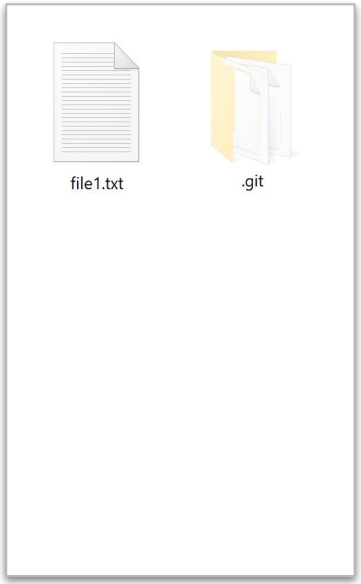
```
$ git status
On branch master
nothing to commit, working tree clean
```





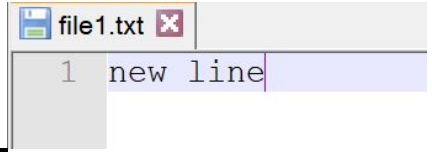
Recording changes to the repository:

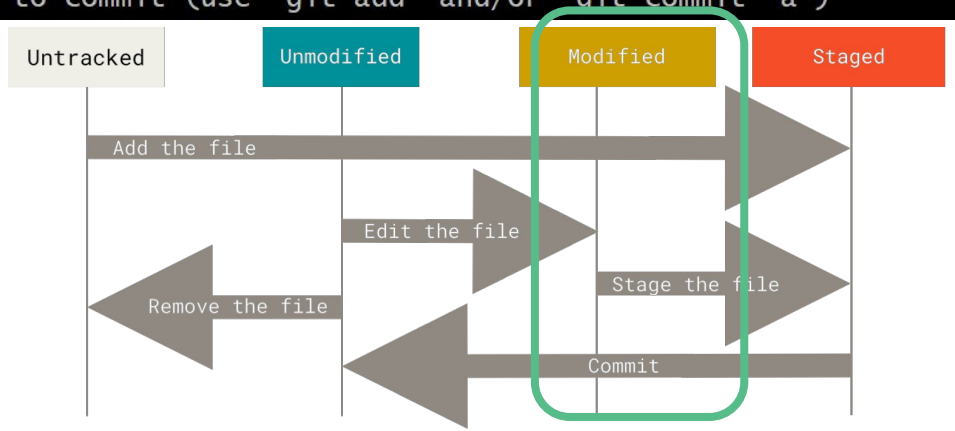


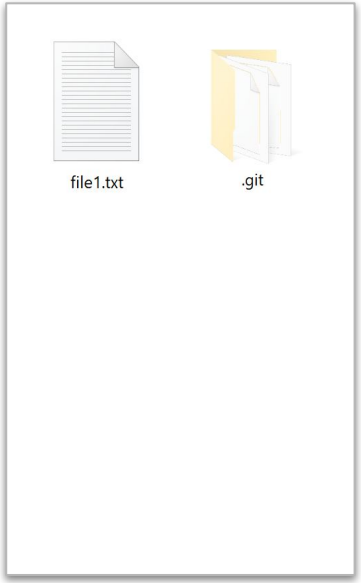


Recording Changes to the Repository

```
$ git status
on branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```



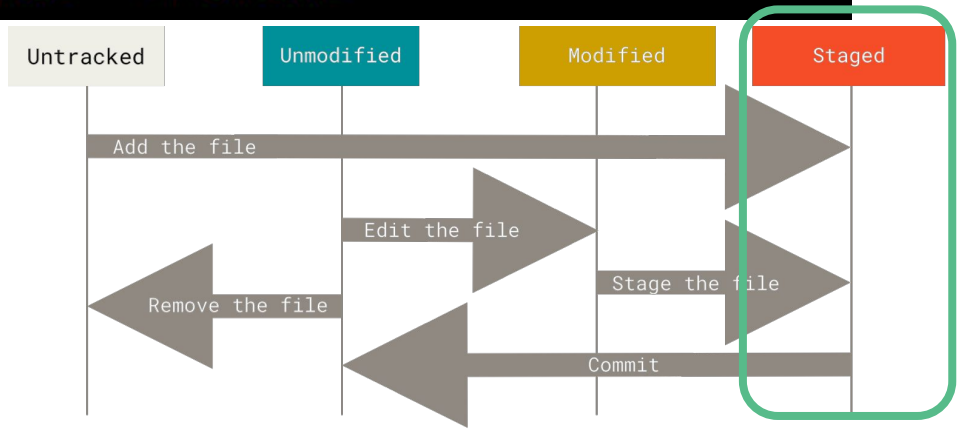


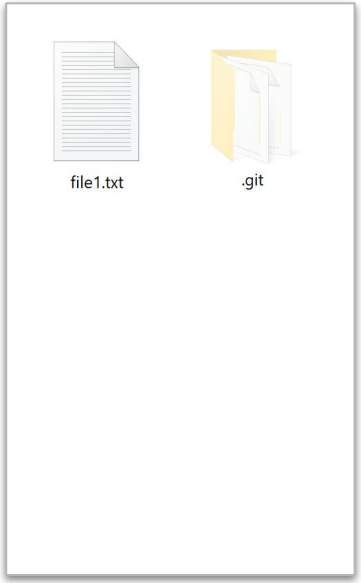


Recording Changes to the Repository

```
$git add file1.txt
$git status
```

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file1.txt
```

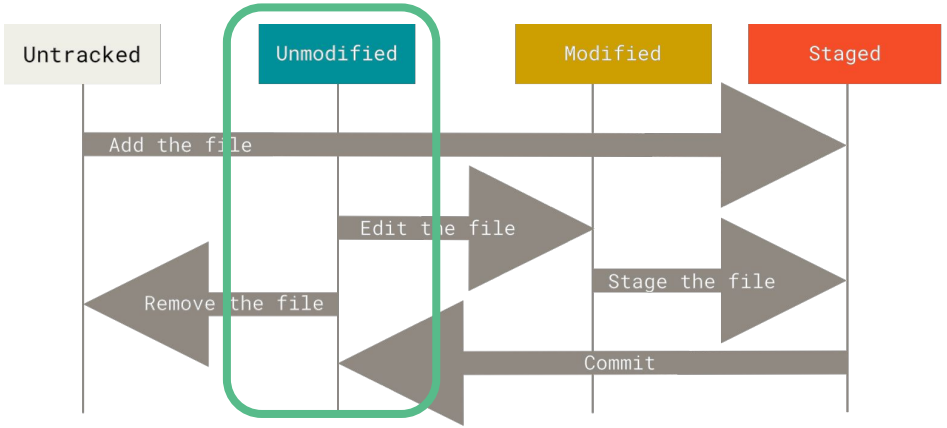


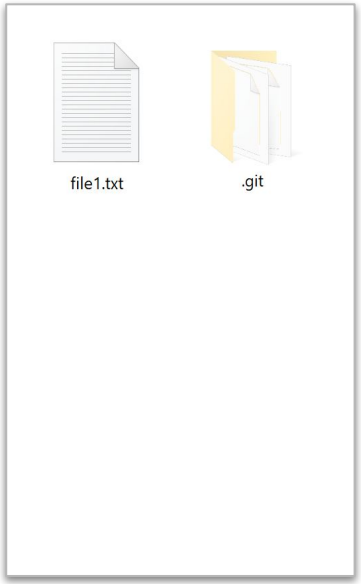


Recording Changes to the Repository

```
$git commit -m "add new line"
```

```
[master 802715a] add new line
 1 file changed, 1 insertion(+)
```

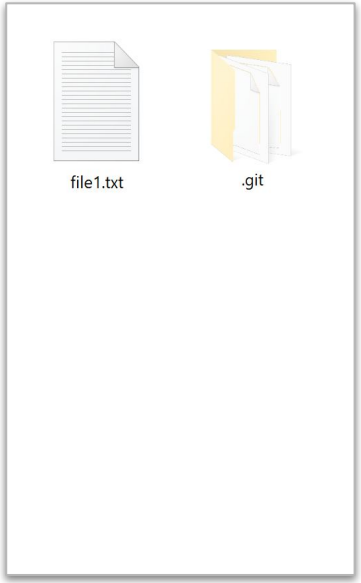




Check differencies

file1.txt

```
1 modified new line
2 a second new line
```



Check differencies

file1.txt

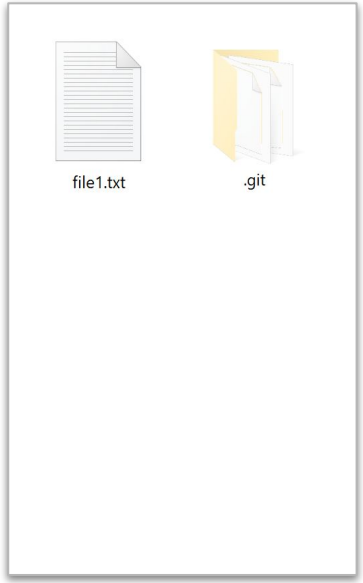
```
1 modified new line
2 a second new line
```

$ git status

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will
  (use "git checkout -- <file>..." to discard

        modified:   file1.txt

no changes added to commit (use "git add" and/
```

$git diff

```
diff --git a/file1.txt b/file1.txt
index c6568ea..b9ca418 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1,2 @@
-new line
\ No newline at end of file
+modified new line
+a second new line
\ No newline at end of file
```

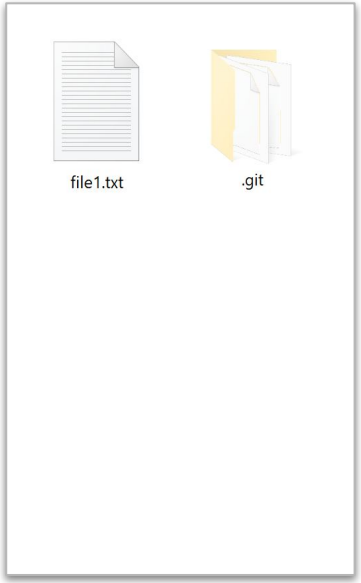


Recording Changes to the Repository

file1.txt

```
1  modified new line
2  a second new line
```

```
$git commit -m "add second line"
[master 53c470a] add second line
 1 file changed, 2 insertions(+), 1 deletion(-)
```

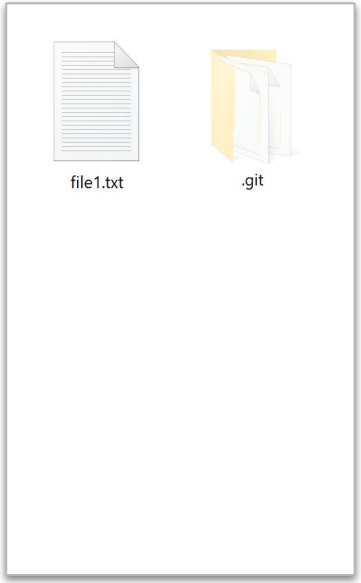


You forgot something?

file1.txt

```
1 modified new line
2 a second new line
3 a third new line
```

```
$git add file1.txt
$git commit --ammend
```



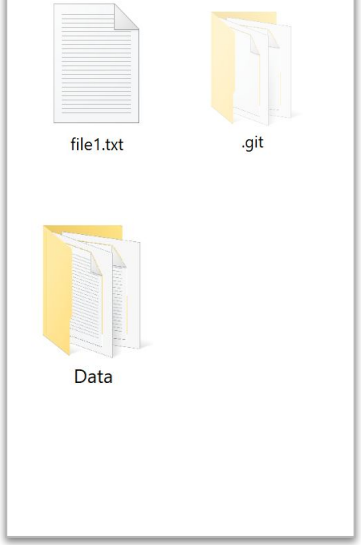
Ignoring Files

```
$ git status
```

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Data/

nothing added to commit but untracked files present (use "git add" to track)
```

.gitignore

```
1  Data/
```



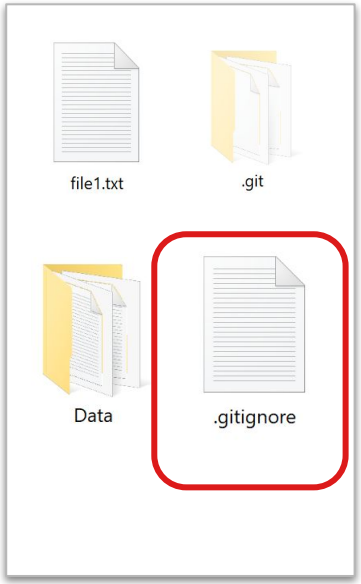
Ignoring Files

```
$ git status
```

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Data/

nothing added to commit but untracked files present (use "git add" to track)
```

.gitignore

```
1  Data/
```



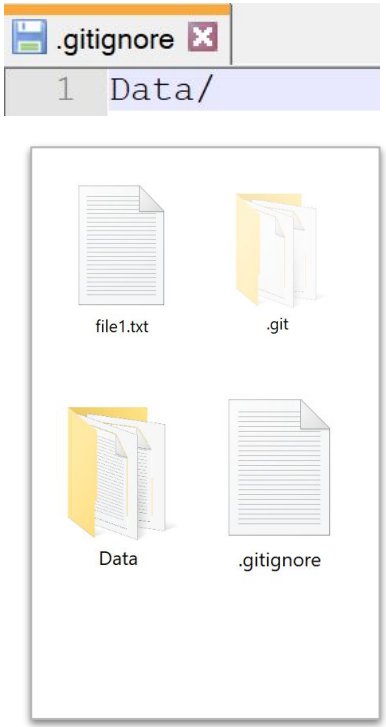
Ignoring Files

$ git status

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Data/

nothing added to commit but untracked files present (use "git add" to track)
```

$ git status

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

nothing added to commit but untracked files present (use "git add" to track)

david.bertoin@B20181627 MINGW64 ~/Workspace/Sandbox/INSA/git_intro (master)
$ git commit -a -m 'add gitignore'
On branch master
Untracked files:
        .gitignore

nothing added to commit but untracked files present
```



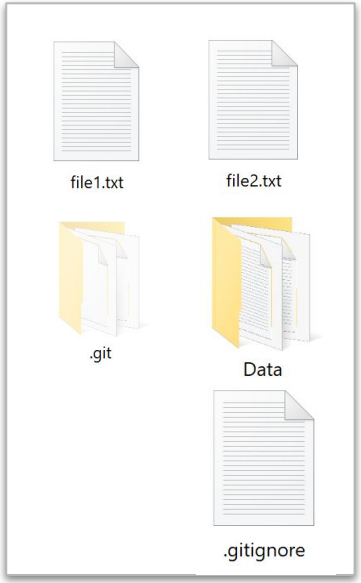
Delete Files

```
$ git status
```

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        file2.txt
```

```
$git add file2.txt
$git commit -m "add file2"
```

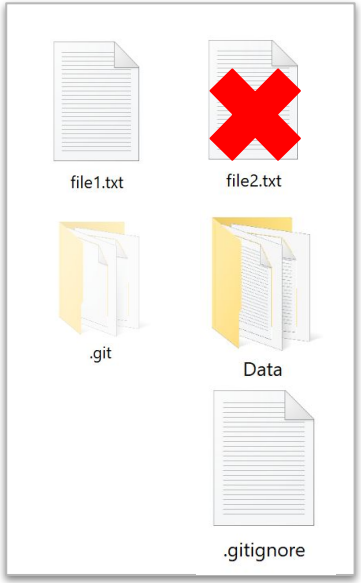
```
[master 1d6e521] add file2
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file2.txt
```



Delete Files

```
$ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    file2.txt
```



Delete Files

$ git status

```
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    file2.txt
```

$git rm file2.txt

$ git status

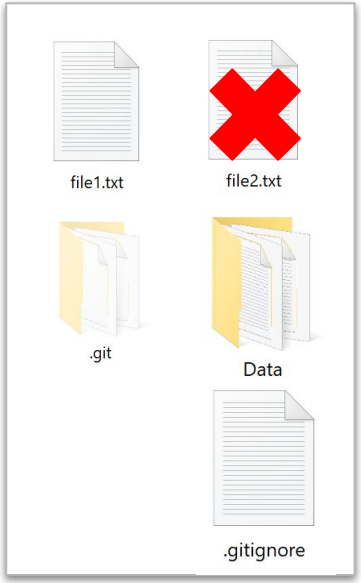
```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    file2.txt
```

$git commit -m 'delete file2'

```
[master 630bdcb] delete file 2
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 file2.txt
```

$ git status

```
On branch master
nothing to commit, working tree clean
```

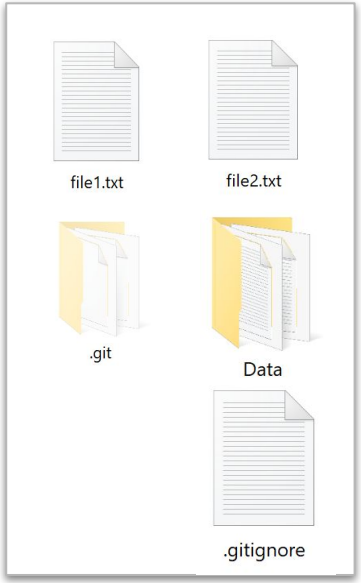


Delete Files

```
$git rm file2.txt --cached
```

```
$ git status
```

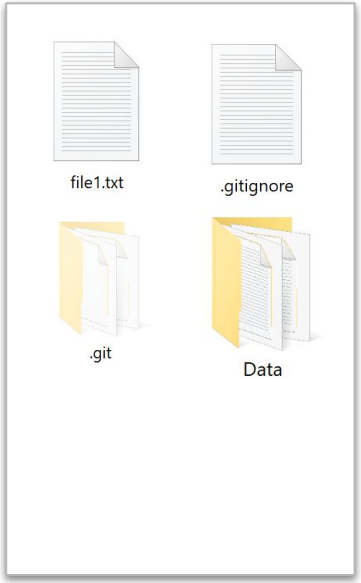
```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    file2.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        file2.txt
```



Unstage Files

```
$git add file1.txt
$git status
```

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file1.txt
```



Unstage Files

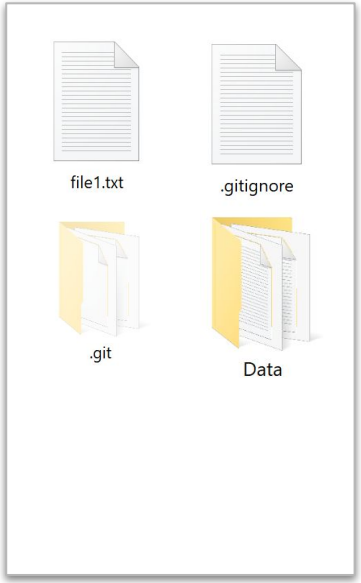
```
$git add file1.txt
$git status
```

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file1.txt
```

```
$git reset HEAD file1.txt
```

```
$ git status
```

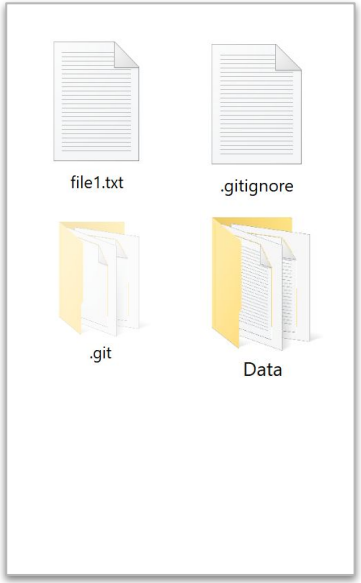
```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   file1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```



Commit History

```
$git log
```

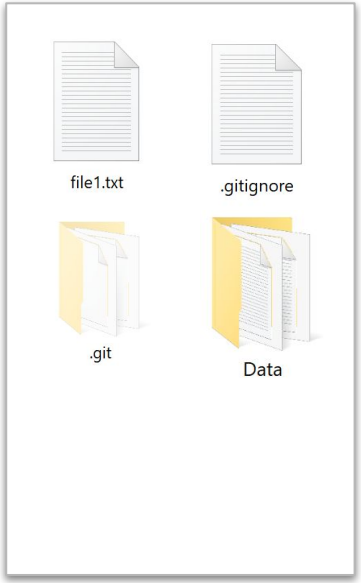
```
commit 4e9240e61210332bf2ed7dc7416dac10ee532d19 (HEAD -> master)
Author: David <davideurofr1@gmail.com>
Date:   Wed Oct 20 14:54:37 2021 +0200

    delete file2

commit 8490684539ab79ac8d5a7e7017bd832d06d13381
Author: David <davideurofr1@gmail.com>
Date:   Wed Oct 20 14:16:48 2021 +0200

    add file2

commit 630bdcbace8363b90603acbc5ba208356fc084d6
Author: David <davideurofr1@gmail.com>
Date:   Wed Oct 20 14:12:55 2021 +0200

    delete file 2

commit 1d6e5219628c3babc3a3add647002d1131036306
Author: David <davideurofr1@gmail.com>
```



Commit History

```
$git log - p 2
```

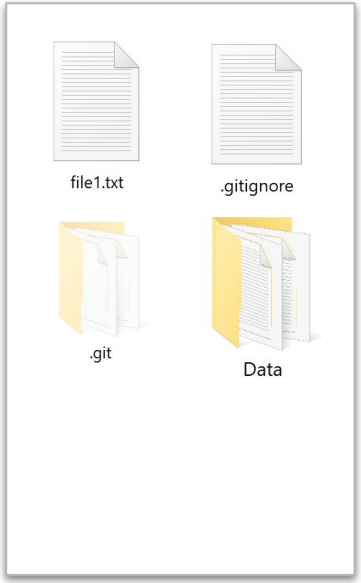
```
commit 4e9240e61210332bf2ed7dc7416dac10ee532d19 (HEAD -> master)
Author: David <davideurofr1@gmail.com>
Date:   Wed Oct 20 14:54:37 2021 +0200

    delete file2

diff --git a/file2.txt b/file2.txt
deleted file mode 100644
index e69de29..0000000

commit 8490684539ab79ac8d5a7e7017bd832d06d13381
Author: David <davideurofr1@gmail.com>
Date:   Wed Oct 20 14:16:48 2021 +0200

    add file2

diff --git a/file2.txt b/file2.txt
new file mode 100644
index 0000000..e69de29
```



Commit History

```
$ git log --pretty=oneline
4e9240e61210332bf2ed7dc7416dac10ee532d19 (HEAD -> master) delete file2
8490684539ab79ac8d5a7e7017bd832d06d13381 add file2
630bdcbace8363b90603acbc5ba208356fc084d6 delete file 2
1d6e5219628c3babc3a3add647002d1131036306 add file2
30d73ae295a60b31147506141253bbed29a88720 add gitignore
53c470a9c90ca4b9a2a858ac5c19608f8b0a9064 add second line
802715a9ca986627a220bcea6a98ed4ca7e25499 add new line
43176a7ebd79fa5aebe12e1f987a7766eacb64e1 first commit
```

```
$ git log --pretty=format:"%h - %an, %ar : %s"
4e9240e - David, 23 minutes ago : delete file2
8490684 - David, 61 minutes ago : add file2
630bdcb - David, 65 minutes ago : delete file 2
1d6e521 - David, 71 minutes ago : add file2
30d73ae - David, 78 minutes ago : add gitignore
53c470a - David, 28 hours ago : add second line
802715a - David, 29 hours ago : add new line
43176a7 - David, 29 hours ago : first commit
```



Remote directory

Remote directory

- Show remote repositories

```
$git remote -v
origin
```

Remote directory

- Show remote repositories

```
$git remote -v
origin
```

```
$git remote -v
origin  https://github.com/DavidBert/AIF.git (fetch)
origin  https://github.com/DavidBert/AIF.git (push)
```

Remote directory

- Show remote repositories

```
$git remote -v
origin
```

```
$git remote -v
origin  https://github.com/DavidBert/AIF.git (fetch)
origin  https://github.com/DavidBert/AIF.git (push)
```

- Add remote repositories

```
$git remote add AIF https://github.com/DavidBert/AIF
```

Remote directory

- Show remote repositories

```
$git remote -v
origin
```

```
$git remote -v
origin  https://github.com/DavidBert/AIF.git (fetch)
origin  https://github.com/DavidBert/AIF.git (push)
```

- Add remote repositories

```
$git remote add AIF https://github.com/DavidBert/AIF
```

- Push to remote repository

```
$git push origin main
```

Remote directory

- Show remote repositories

```
$git remote -v
origin
```

```
$git remote -v
origin  https://github.com/DavidBert/AIF.git (fetch)
origin  https://github.com/DavidBert/AIF.git (push)
```

- Add remote repositories

```
$git remote add AIF https://github.com/DavidBert/AIF
```

- Push to remote repository

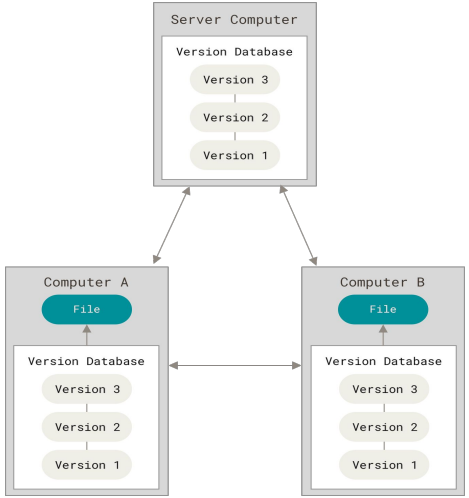
```
$git push origin main
```

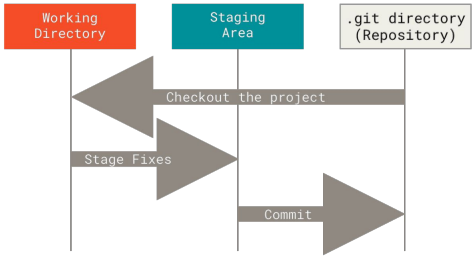
- Get data from remote repository

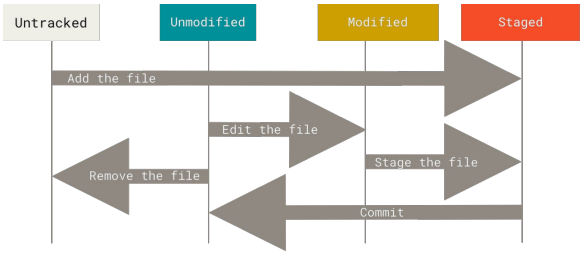
```
$git fetch origin
$git pull origin
```

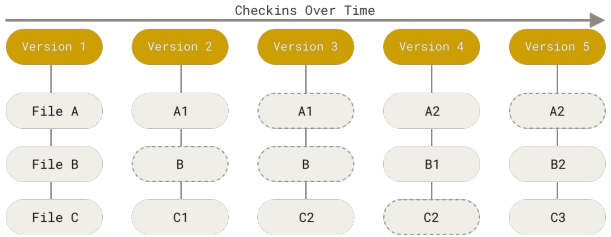
Previously

- Version control systems and Git
- Commits
- Working areas
- Four states
- Basic git commands

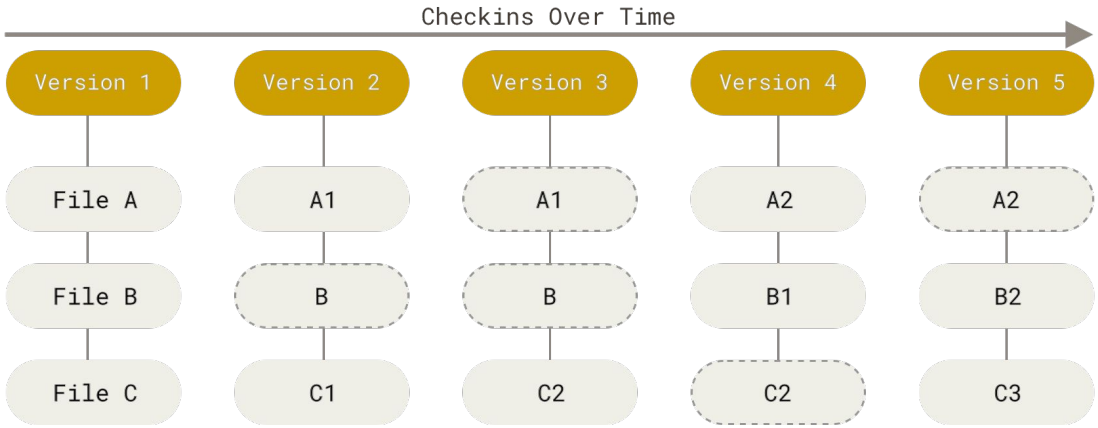




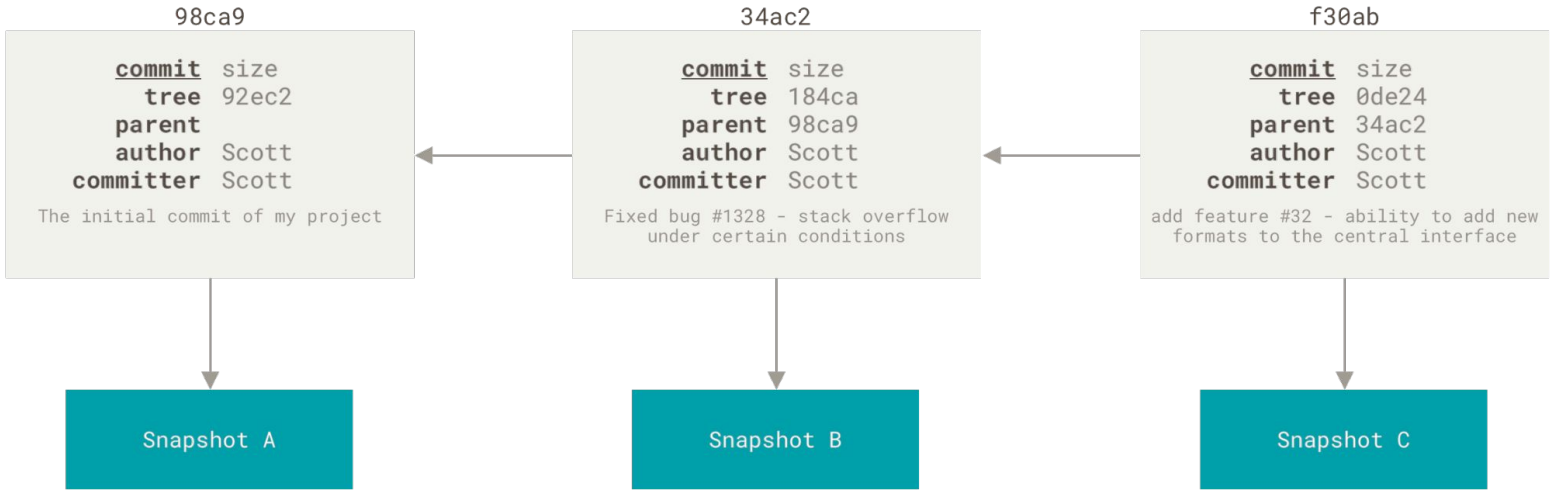




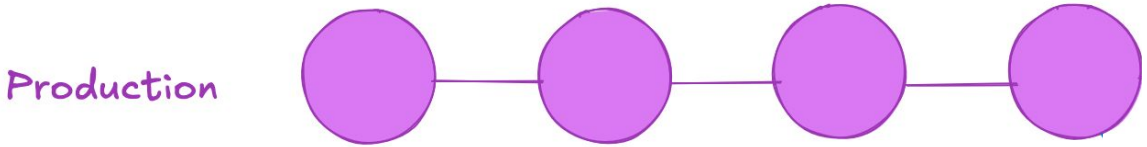
Previously


Checkins Over Time
Version 1
Version 2
Version 3
Version 4
Version 5
File A
File B
File C
A1
B
C1
A1
B
C2
A2
B1
C2
A2
B2
C3

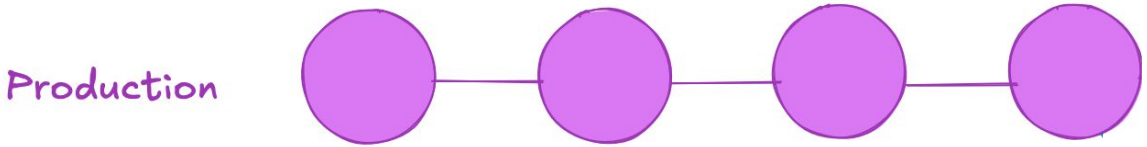
History as a graph



Branching

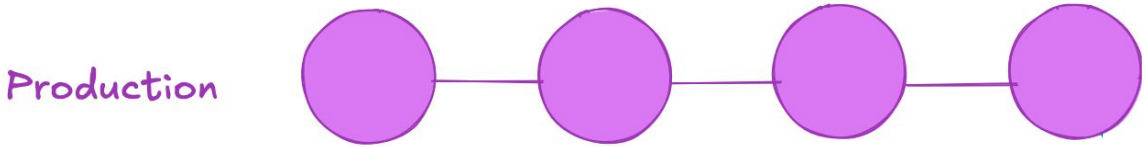


Branching



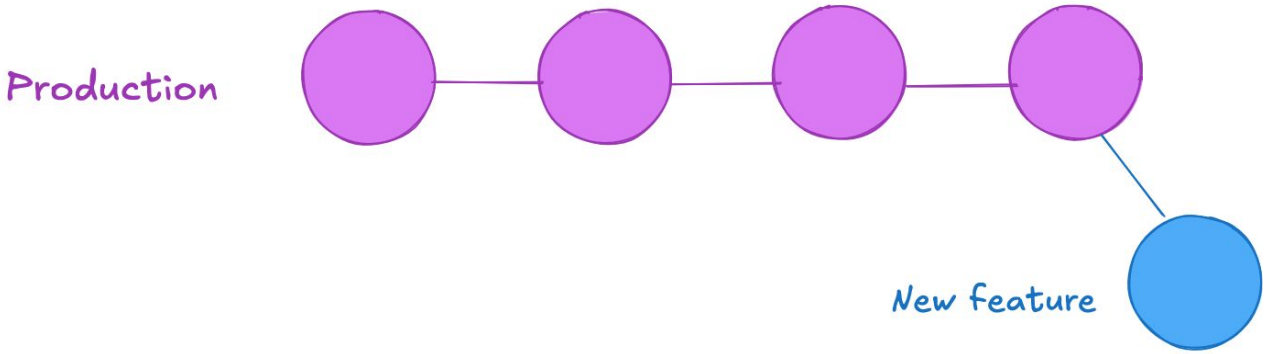
- A cool library used by many users
- Want to add a new complicated feature with huge changes in the codebase
- How to develop with help of code versioning without impacting the production code?

Branching

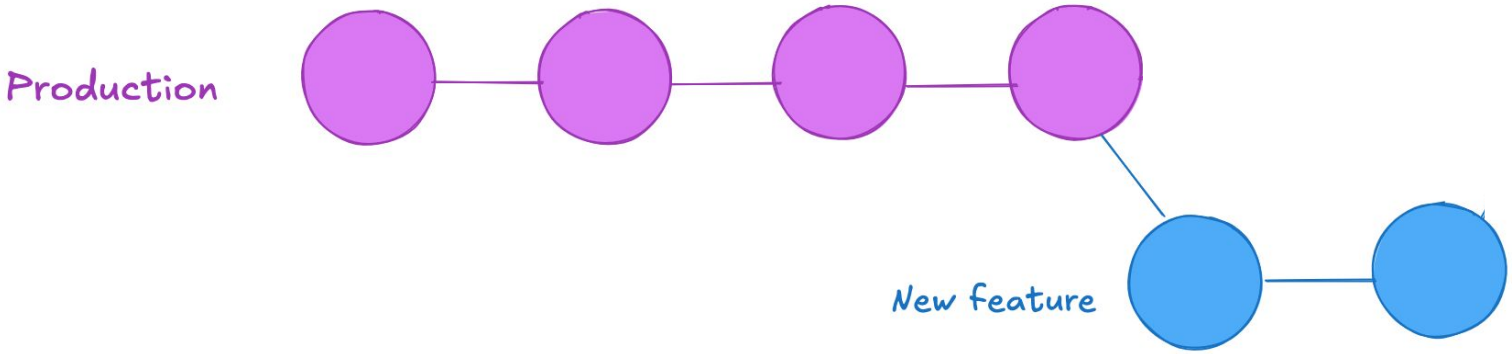


- A cool library used by many users
- Want to add a new complicated feature with huge changes in the codebase
- How to develop with help of code versioning without impacting the production code?
  => Work on a parallel version of the code

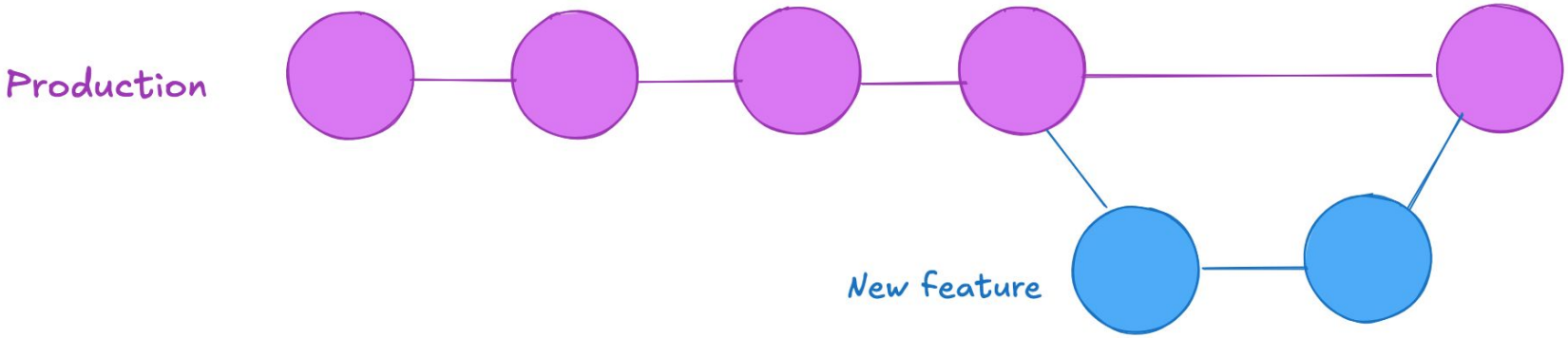
Branching



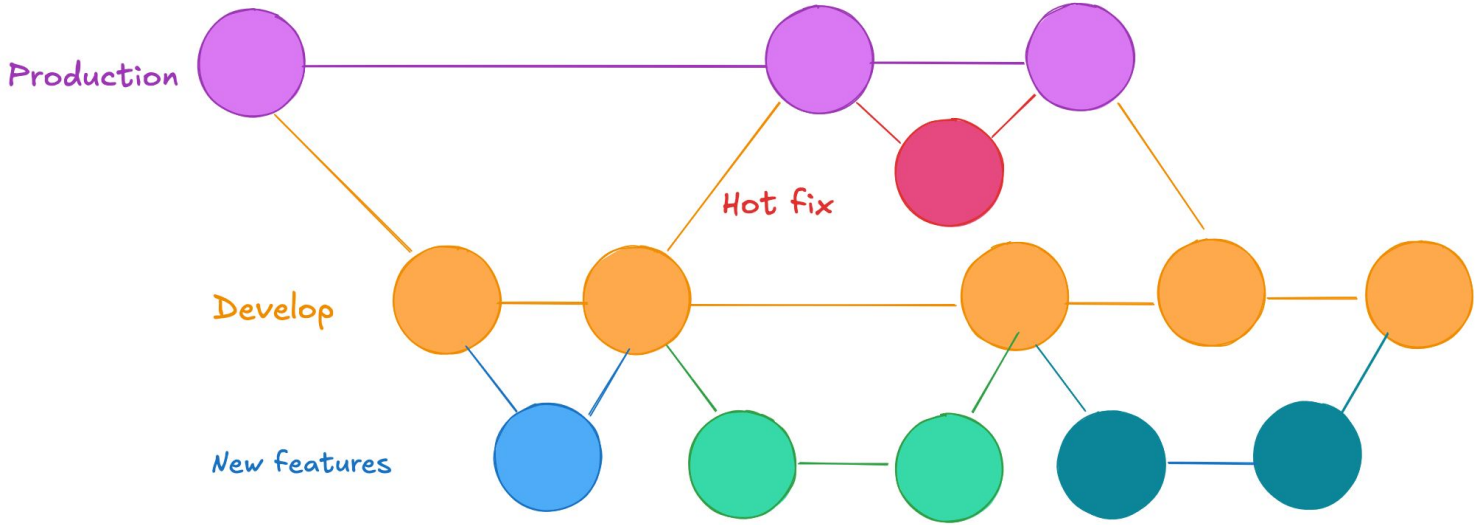
Branching



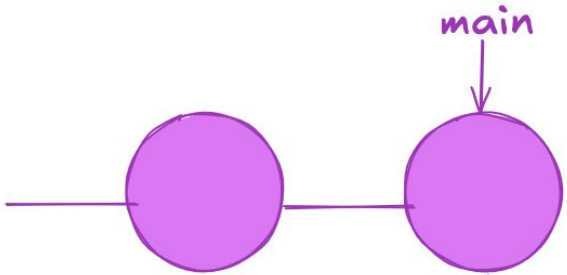
Branching



Branching

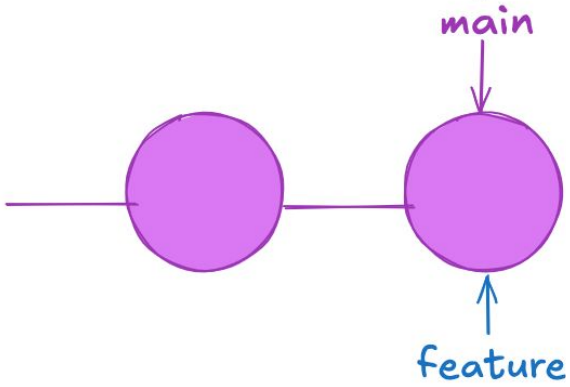


Branching

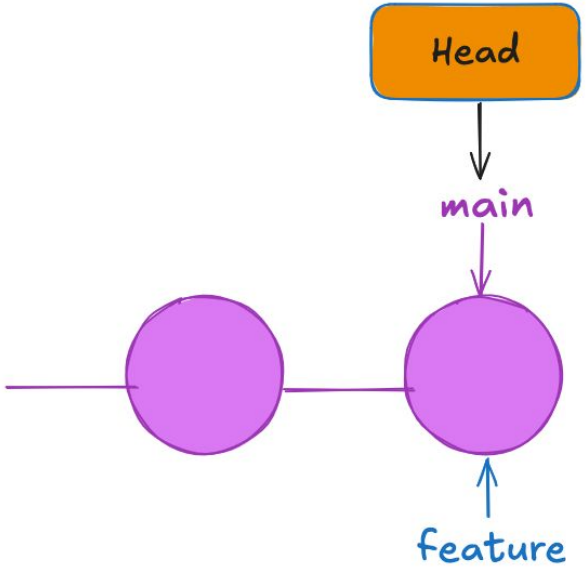


Branching

```
$git branch feature
```

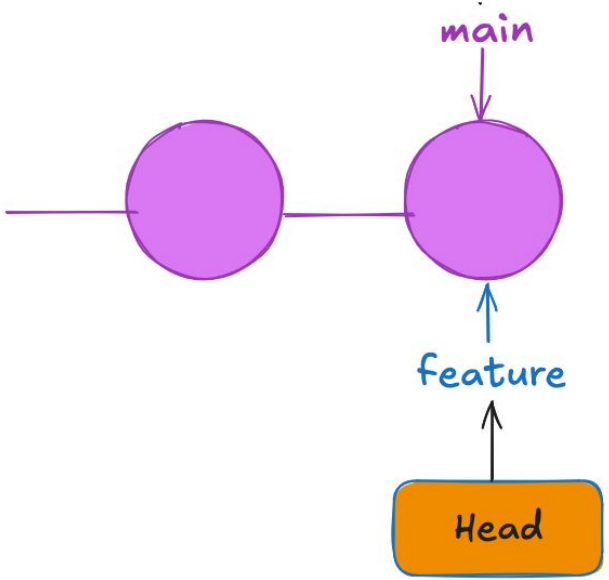


Branching




```
$git branch feature
```

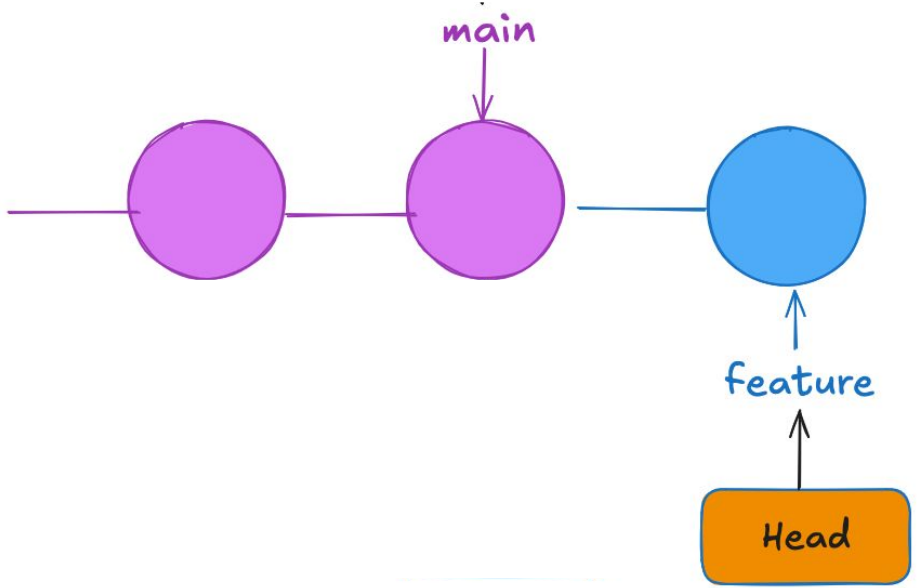
Branching



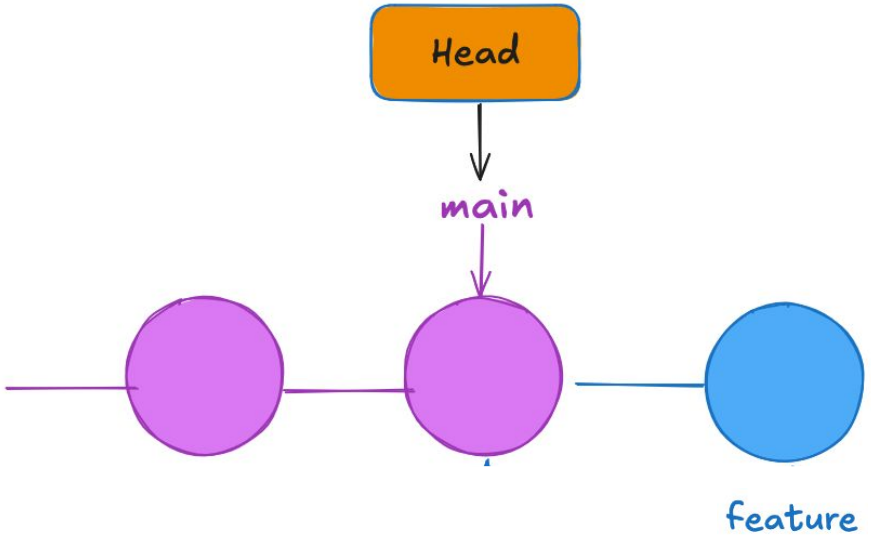

```
$git checkout feature
```

Branching

```
$git commit -m "add feature"
```

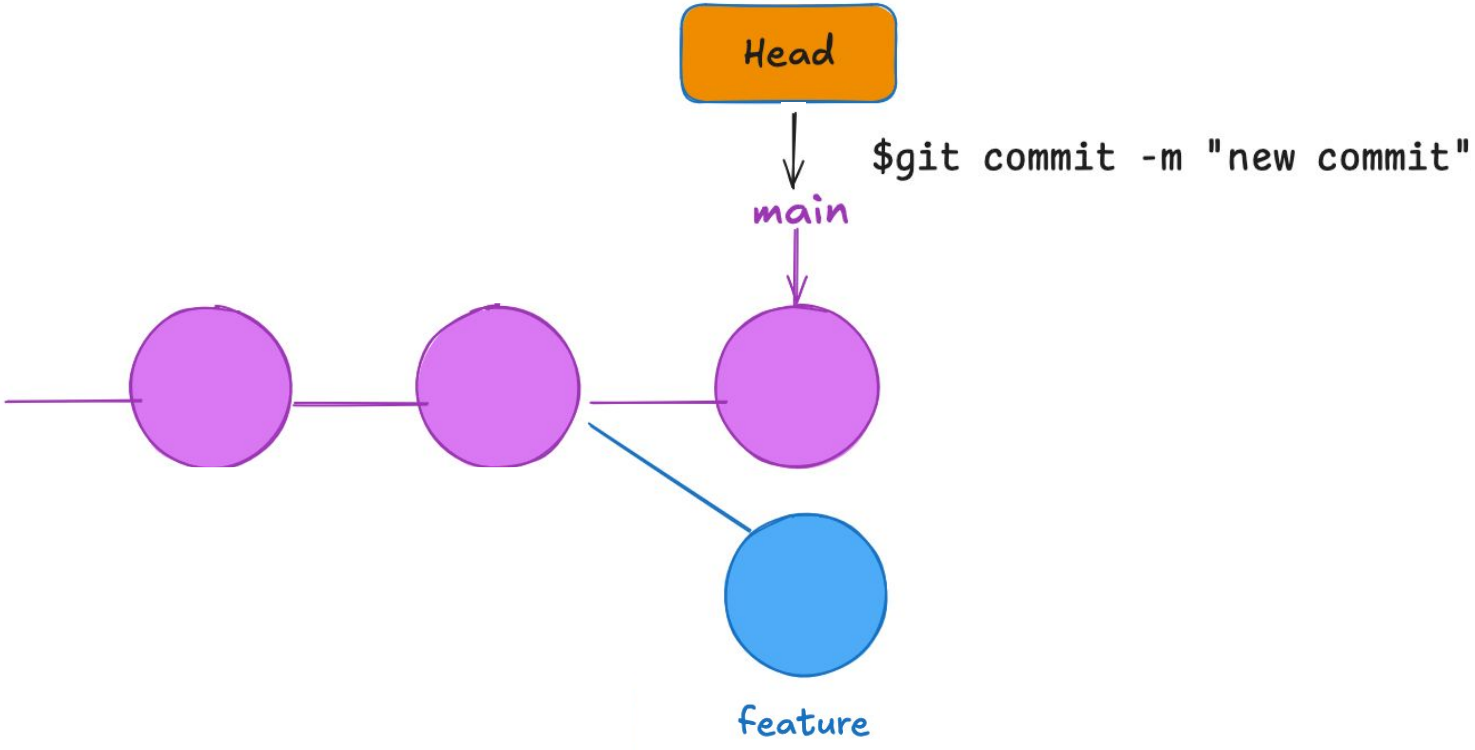


Branching

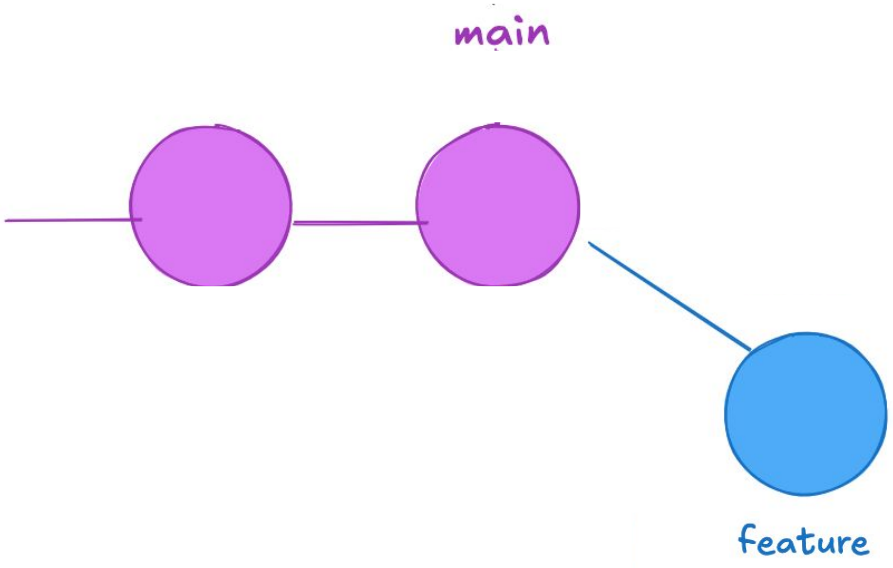



```
$git checkout main
```

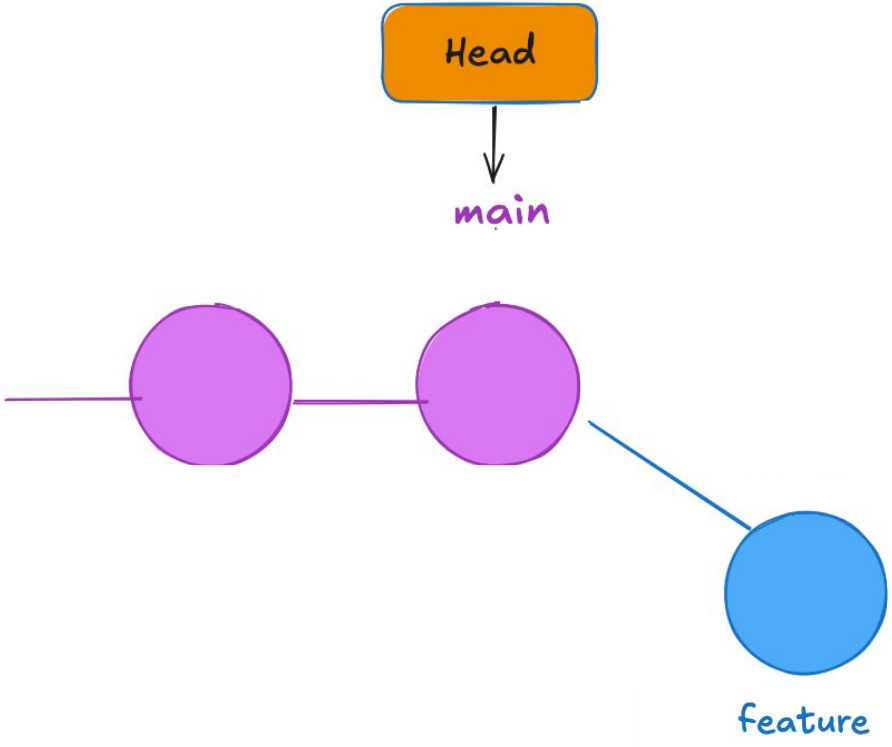
Branching



Merging: the easy way

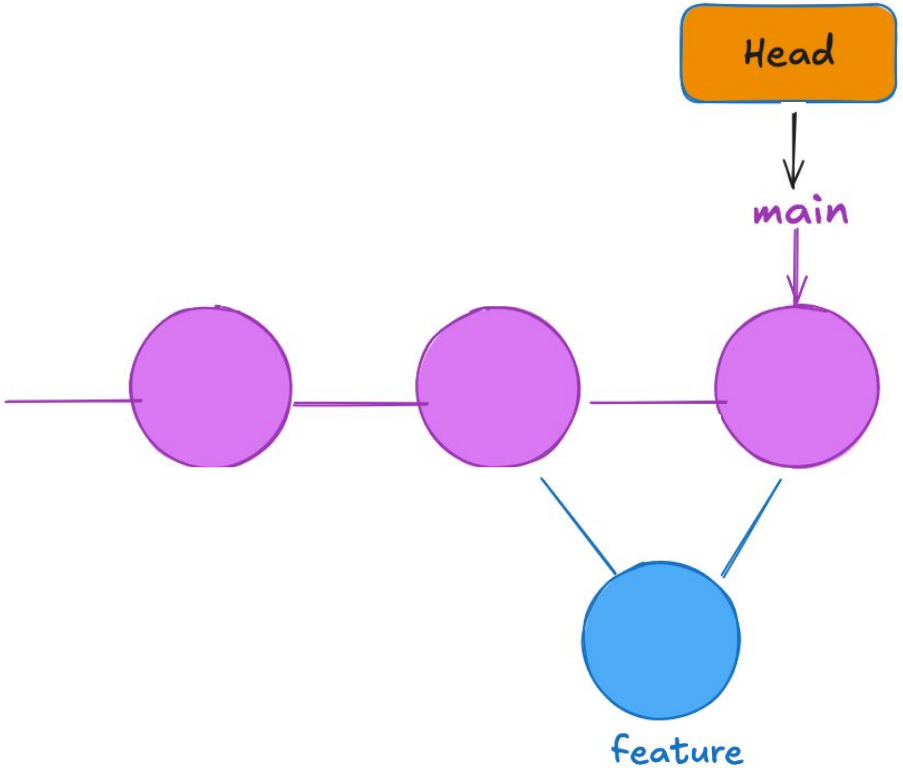


Merging: the easy way



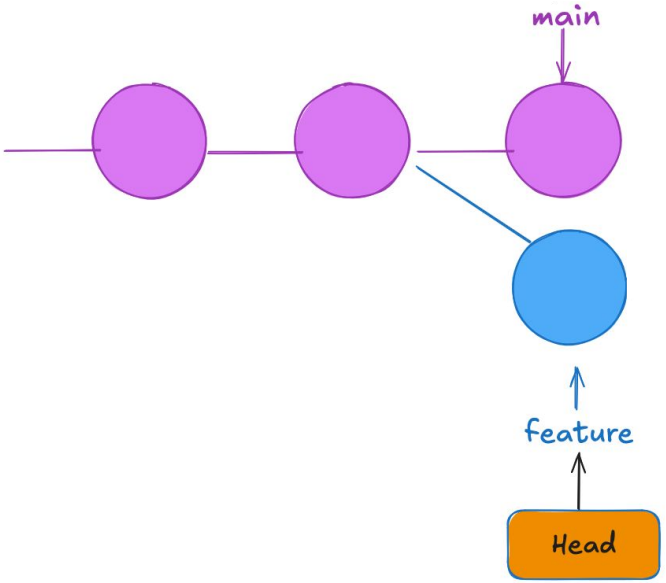

```
$git checkout main
```

Merging: the easy way

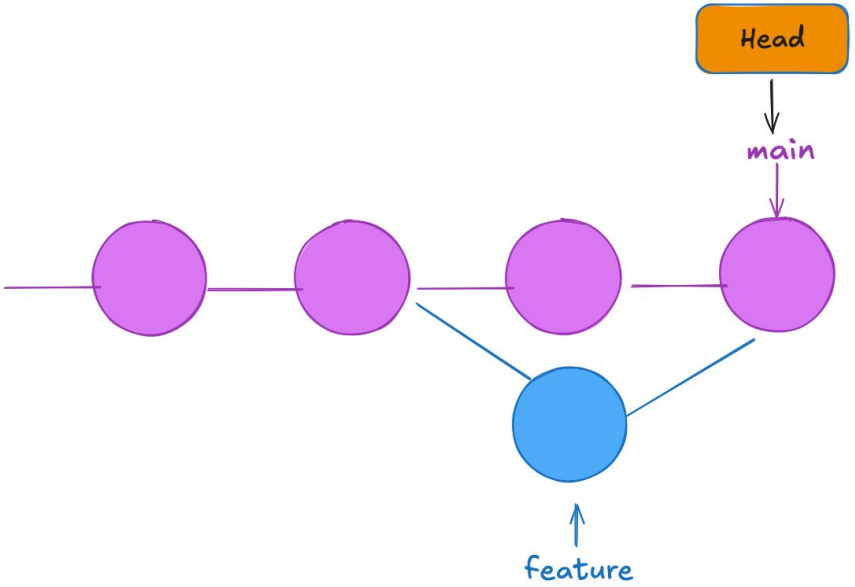



```
$git merge feature
```

Merging: the more complicated way

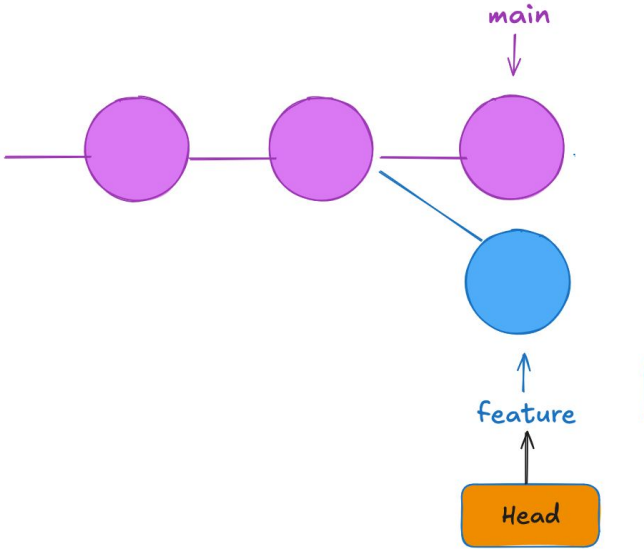


Merging: the more complicated way



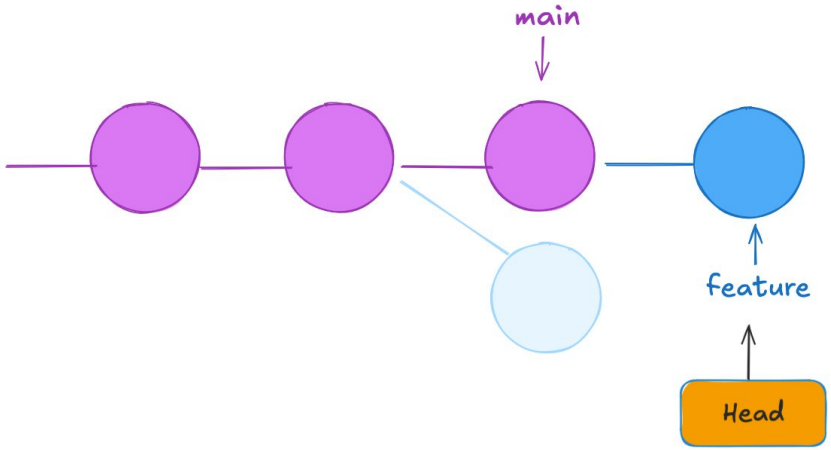

```
$git checkout main
$git merge feature
```

Rebasing




```
$git checkout feature
$git rebase main
```

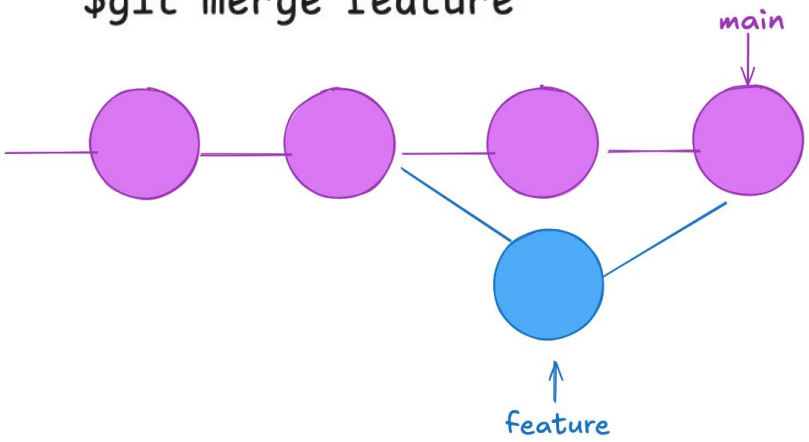
Rebasing



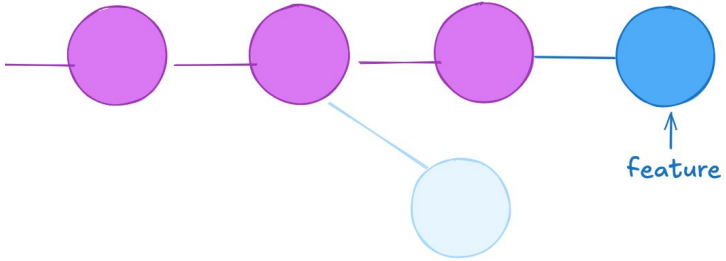

```
$git checkout feature
$git rebase main
```

Rebasing

```
$git checkout main
$git merge feature
```

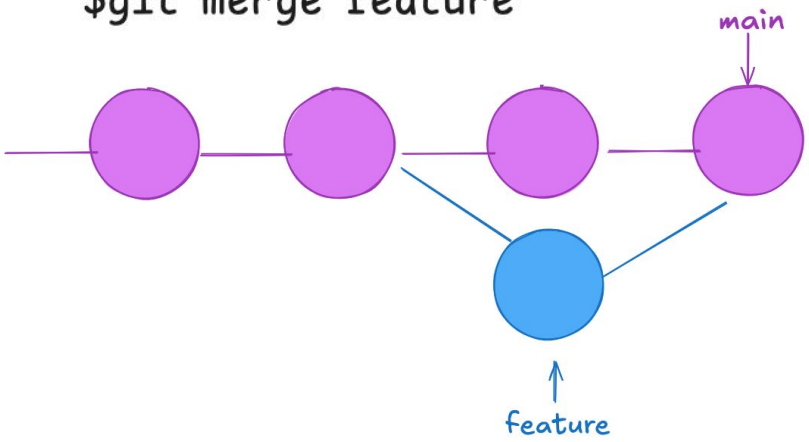


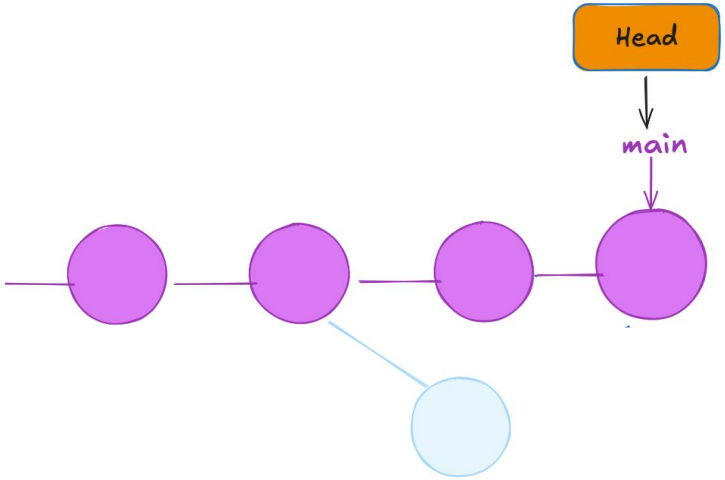

```
$git checkout feature
$git rebase main
```



Rebasing

```
$git checkout main
$git merge feature
```

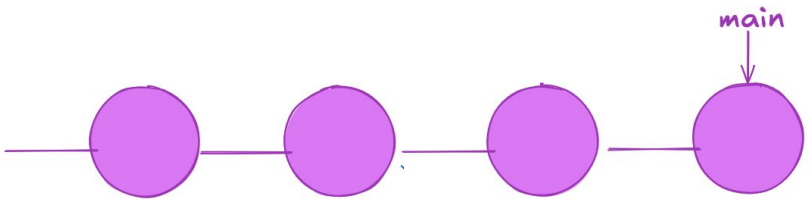


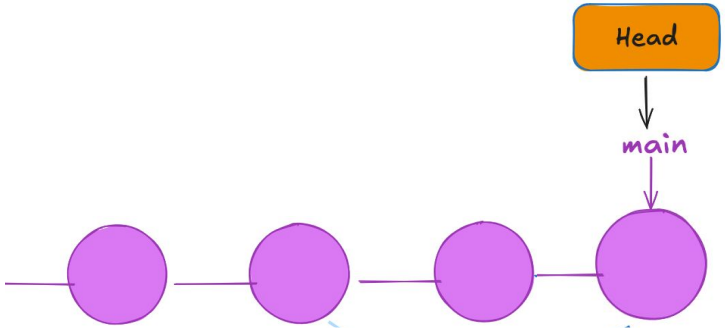



```
$git checkout main
$git merge feature
```

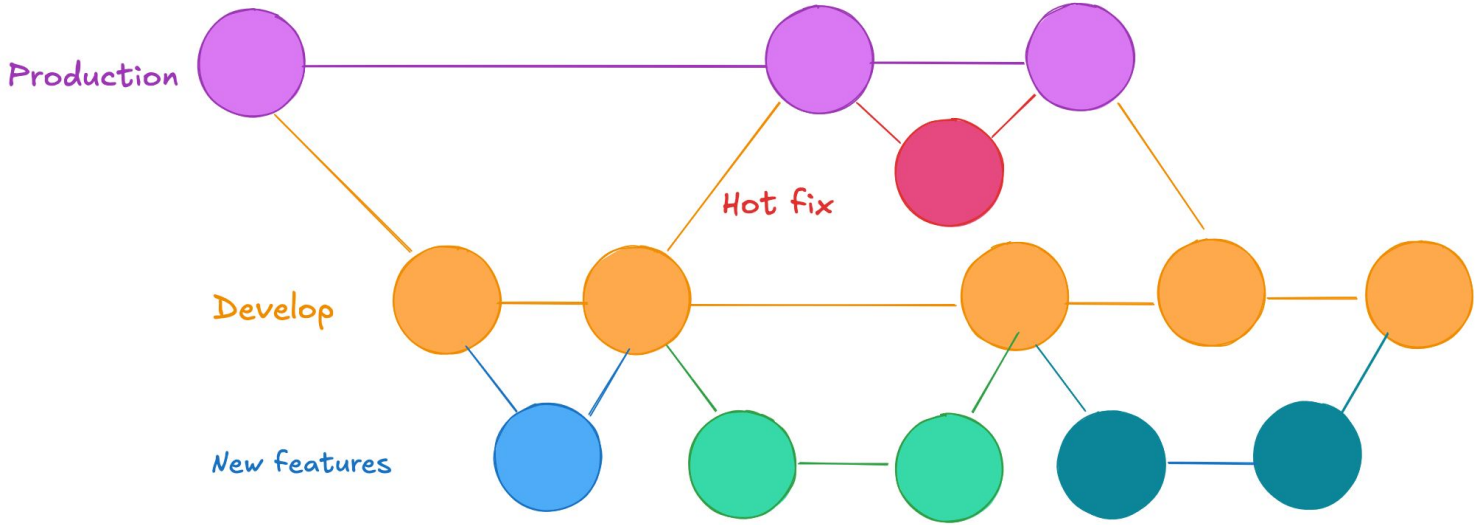
Delete your branches

```
$git branch -d feature
```





Branching



Project: do branches and pull requests!

- **Block 1 :** Poster classification
  - **Goal:** Classify automatically movies genre according to their posters
- **Block 2:** Poster classification
  - **Goal:** Predict with confidence bounds
- **Block 3:** Poster classification
  - **Goal:** detect images that are not movie posters
- **Block 4:** Recommend similar movies
  - Goal: recommend similar movies using posters or movie plots
- **Block 5:** Movie Retrieval
  - Goal: Ask questions about movies

Jupyter ≠ Production

Notebooks are for exploration, not deployment

Real-world ML needs:

- **Reproducibility** (same code runs everywhere)
  -> Containers (Docker)

- **Scalability** (serve many users, handle large requests)
  -> Rest APIs (Flask/FastAPI), load balancing, cloud deployment (GCP)

- **Maintainability** (collaboration, updates, bug fixes)
  -> Version control (Git)