

# GitHub actions ci/cd

[risorse.vercel.app/lab/2024](https://risorse.vercel.app/lab/2024)

Alice Benatti, Samuele Musiani

>ADM  
staff

# CI/CD

# continuous integration continuous delivery

**per automatizzare la build e il testing del codice**

La Continuous Integration (CI) è la pratica di **automatizzare la compilazione** e il test del codice ogni volta che viene apportata una modifica.

Ogni nuovo commit del codice attiva un **processo di compilazione** e test automatizzato, spesso chiamato "**pipeline**", per segnalare eventuali difetti rilevati durante la compilazione o il test il più rapidamente possibile.



# Perché usarla?

- Automaticamente vengono fatti test sul codice
- Test centralizzati su tutta la repository
- Semplifica il debugging
- Migliora la produttività



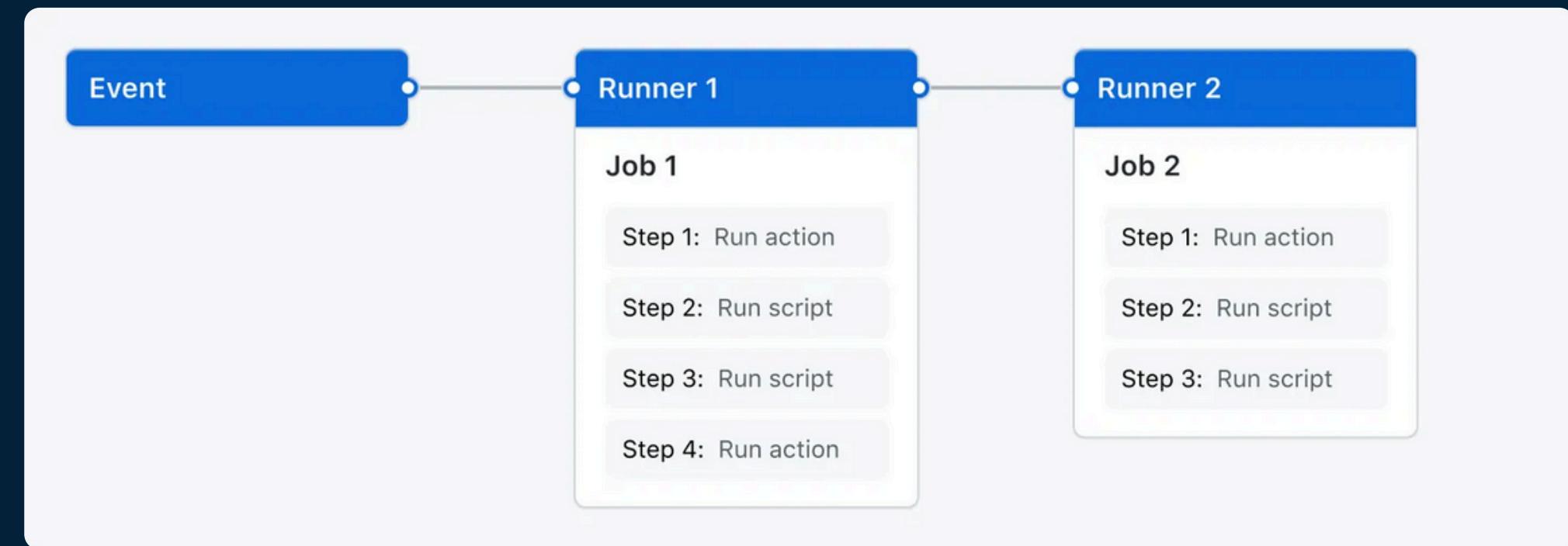
# GitHub Actions

GitHub ci mette a disposizione le *actions* in modo da **automatizzare i processi (CI)**

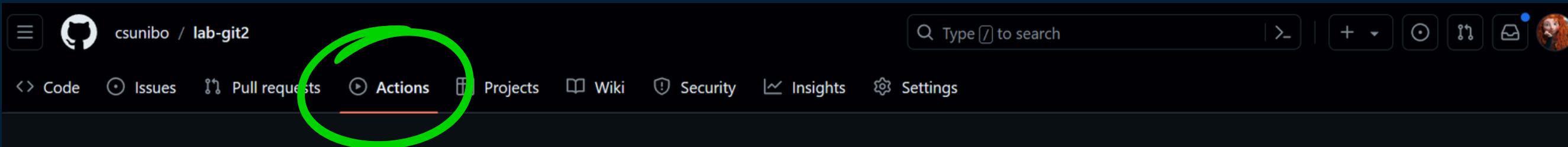
Anche altri sistemi di hosting di git permettono di avere dei workflow simili alle GitHub Actions.

Contengono uno o più processi che possono essere eseguiti in ordine sequenziale o in parallelo.

Ogni processo verrà eseguito all'interno del proprio runner di macchine virtuali o all'interno di un contenitore e ha uno o più passaggi che eseguono uno script definito dall'utente o un'azione.



# GitHub Actions



The screenshot shows the GitHub Actions interface for the repository 'csunibo / lab-git2'. The 'Actions' tab is highlighted with a green circle. The main heading is 'Get started with GitHub Actions'. Below it, there's a search bar with the placeholder 'Search workflows' and a link to 'Skip this and set up a workflow yourself →'. A section titled 'Suggested for this repository' displays six workflow templates:

- C/C++ with Make** By GitHub Actions: Build and test a C/C++ project using Make. Includes a 'Configure' button and a 'C' badge.
- Python Package using Anaconda** By GitHub Actions: Create and test a Python package on multiple Python versions using Anaconda for package management. Includes a 'Configure' button and a 'Python' badge.
- CMake based, multi-platform projects** By GitHub Actions: Build and test a CMake based project on multiple platforms. Includes a 'Configure' button and a 'C' badge.
- CMake based, single-platform projects** By GitHub Actions: Build and test a CMake based project on a single-platform. Includes a 'Configure' button and a 'C' badge.
- Publish Python Package** By GitHub Actions: Publish a Python Package to PyPI on release. Includes a 'Configure' button and a 'Python' badge.
- Django** By GitHub Actions: Build and Test a Django Project. Includes a 'Configure' button and a 'Python' badge.

At the bottom, there are 'Deployment' and 'View all' buttons.

## CONTINUOUS INTEGRATION

Build

Test

Merge

## CONTINUOUS DEPLOYMENT

deploy automatico



# WORKFLOW

RUNNER

JOB

Step

Step

Step

Step

RUNNER

JOB

Step

Step

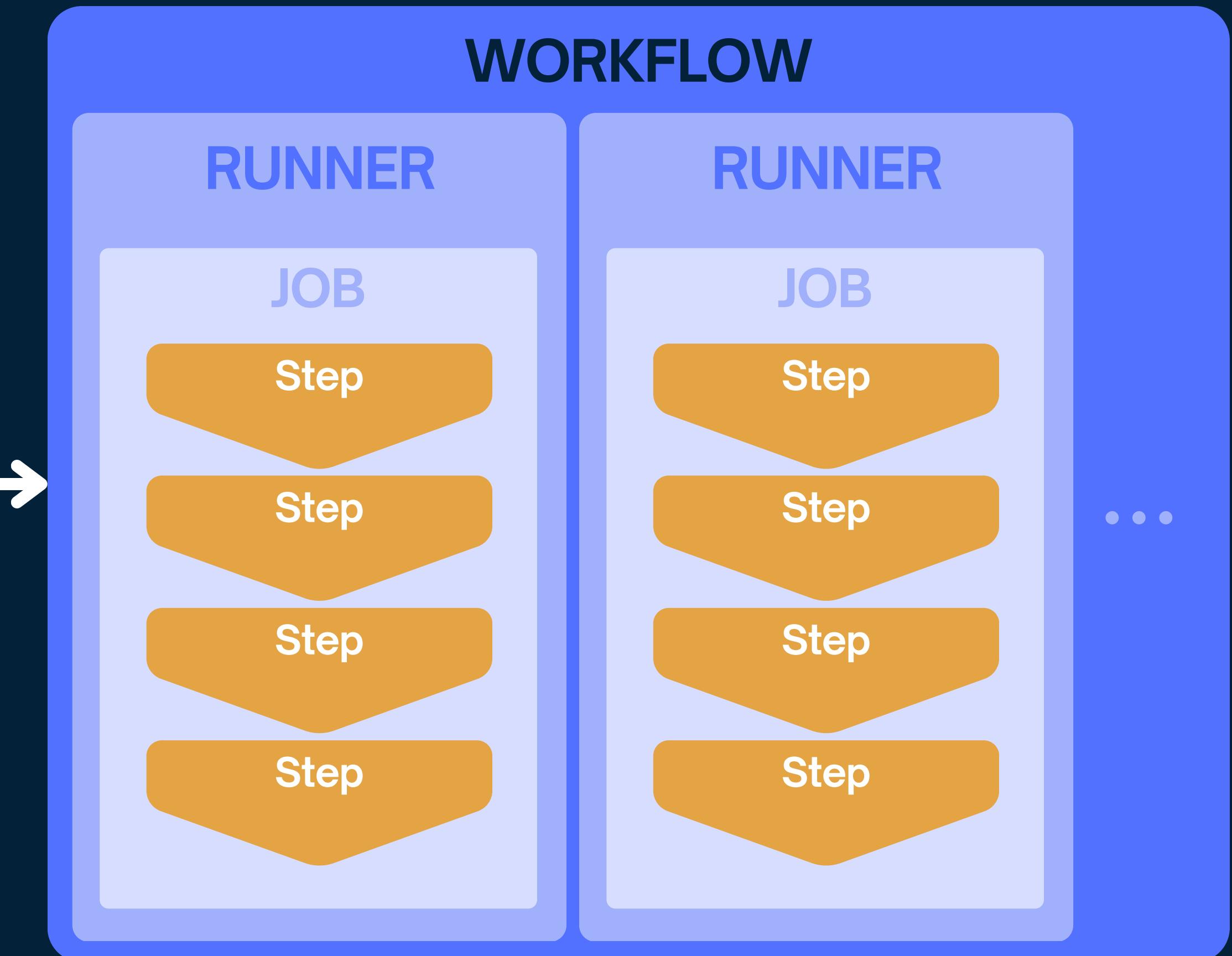
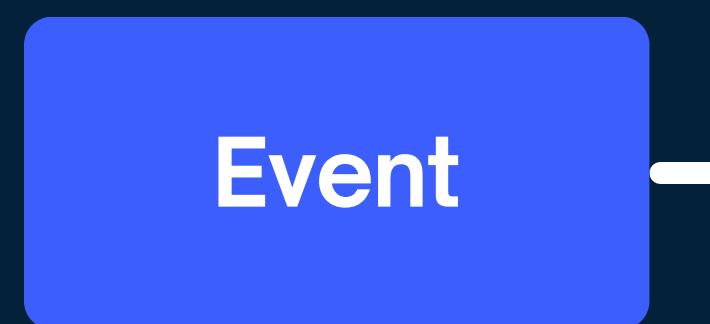
Step

Step

...

Event

provoca →



# Events

## Eventi che provocano l'esecuzione di un workflow

- **push**: tutte le volte che dei commit vengono pushati in remoto
- **pull\_request**: in base all'*activity type* scelto [created, opened, closed, ...]
- **fork**: tutte le volte che si fa una fork
- **issues**: in base all'*activity type* scelto [edited, opened, closed, ...]
- **workflow\_dispatch**: ci permette di eseguire l'action manualmente da github

Per info vedi la documentazione GitHub:

<https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows>

# Events | filters

## Filtri per restringere gli eventi

Alcuni eventi hanno dei filtri per avere un controllo maggiore su come viene provocato il workflow.

Per esempio *push*, può avere specificate alcune **branch**.

Posso quindi fare specifici workflows per main o branch che matchano un pattern definito.

Posso anche scegliere quali branch escludere.

Ci sono anche filtri sui **path** che permettono di provocare un workflow solo se alcuni file specifici/pattern vengono modificati.

Per info vedi la documentazione GitHub:

<https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows>

# Jobs

**compongono un workflow**

Un workflow è composto da uno o più jobs.

Di predefinito i jobs vengono eseguiti in parallelo.

Si possono avere quanti jobs si vuole all'interno dello stesso workflow.

Un job è composto da una o più azioni (comandi).

**Per info vedi la documentazione GitHub:**

**<https://docs.github.com/en/actions/using-jobs/using-jobs-in-a-workflow>**

# YAML

**yet another markup language**

È un linguaggio di markup che viene usato per i file di configurazione.

```
foo: bar
pleh: help
stuff:
  foo: bar
  bar: foo
```

I workflow di GitHub vengono scritti in YAML

Per info vedi la documentazione GitHub:

<https://docs.github.com/en/actions/using-jobs/using-jobs-in-a-workflow>

```
name: Basic workflow
```

→ **Nome del workflow**

```
on:
```

```
  push:
```

```
    branches:
```

```
      - main
```

```
jobs:
```

```
  basic-job:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

```
      - name: Hello World
```

```
        run: echo "Hello world"
```

→ **Events**

→ **Filtri**

→ **primo Jobs**



# .github/workflows

**cartella da creare per contenere tutti i workflow della repository**

Contiene tutti i file *.yaml* dei vari workflow.

È importante creare files separati per ogni workflow.

Eventuali workflow esterni alla directory non verranno considerati da GitHub come Actions.

# Runs-on

## immagine su cui esegue il job

GitHub quando esegue i workflow alloca una Virtual Machine dedicata. Possiamo decidere il suo sistema operativo con l'opzione *runs-on*

- ***ubuntu-latest***
- ***windows-latest***
- ***macos-latest***
- ...

Per info vedi la documentazione GitHub:

<https://docs.github.com/en/actions/using-jobs/using-jobs-in-a-workflow>

# Steps

**sono le azioni che compongo i jobs**

In questo caso è presente un unico step che esegue il comando:

*echo "Hello world"*

Ci sono anche altri tipi di step oltre a run, il più comune è **uses** che esegue un workflow già definito.

...

jobs:

basic-job:

runs-on: ubuntu-latest

steps:

- name: Hello World

run: 'echo "Hello world"'



# Steps | uses

**sono le azioni che compongo i jobs**

È molto comune voler eseguire dei comandi/azioni sulla propria repository, dovremmo quindi fare un clone sul runner.

Con **uses** possiamo farlo usando un'action preesistente (p.e. checkout)

...

jobs:

basic-job:

runs-on: ubuntu-latest

steps:

- name: Clone repo

uses: actions/checkout@v4



# Steps | uses

# le più popolari

**sono le azioni che compongo i jobs**

Jamesives/github-pages-deploy-action@4.1.4 deploy di una cartella nella gh-pages branch

actions/setup-python@v2 setup un environment python

github/super-linter@v4 eseguire una combinazione di linter generando un'unico report

Qui ne trovi tante altre:

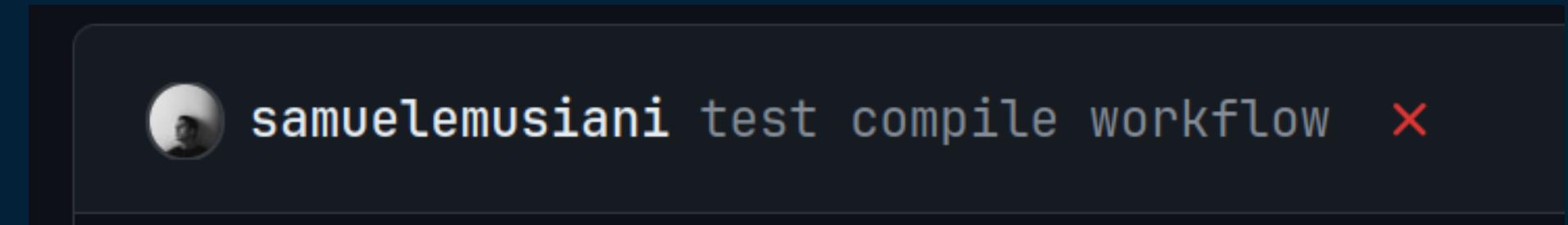
<https://github.com/sdras/awesome-actions>

# ✖ Action failed

Le actions possono riscontrare degli errori

Quando un comando del workflow produce un errore, l'actions viene bloccata da GitHub e le viene assegnata lo status di “**failed**”

Tutti gli steps successivi non verranno runnati.



# Events | Workflow dispatch

ci permette di eseguire l'actions manualmente

```
name: Basic workflow
```

```
on:
```

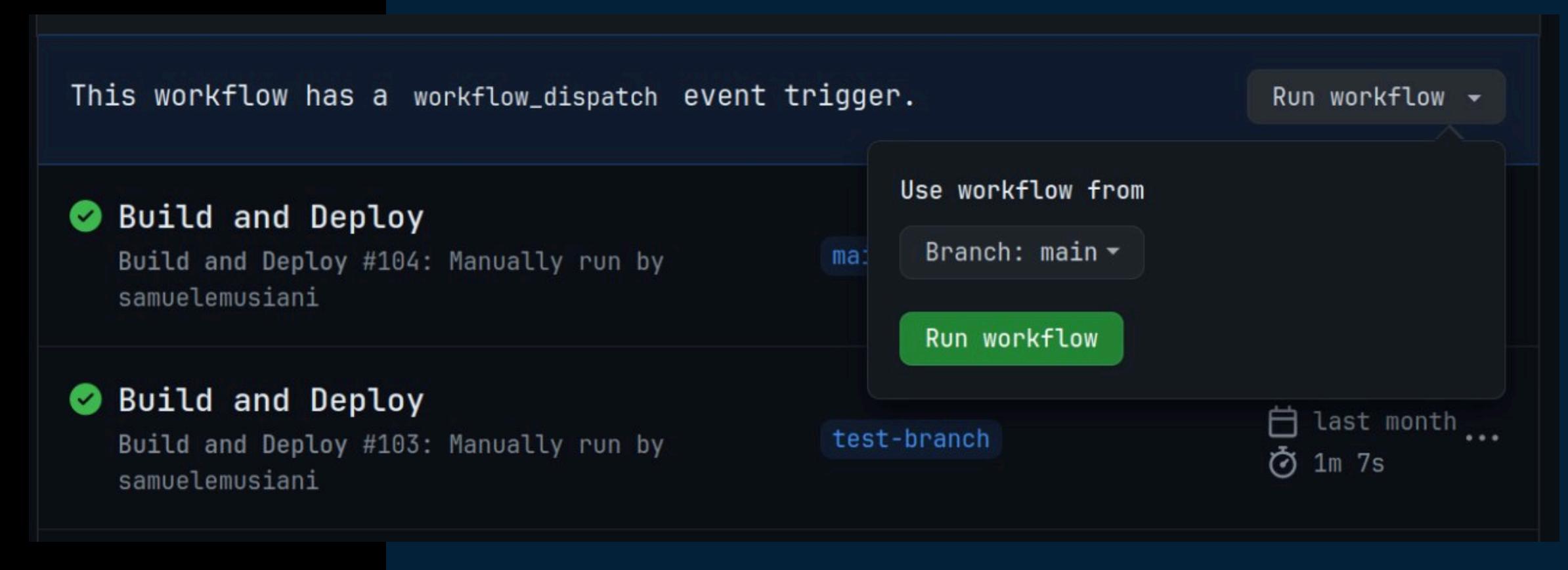
```
push:
```

```
  branches:
```

```
    - main
```

```
workflow_dispatch:
```

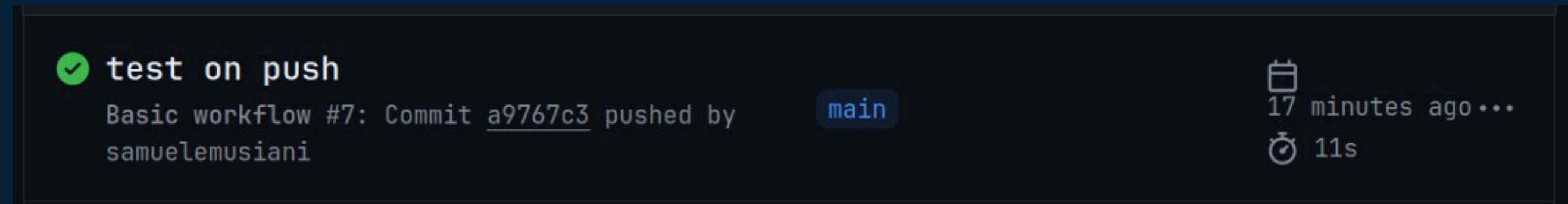
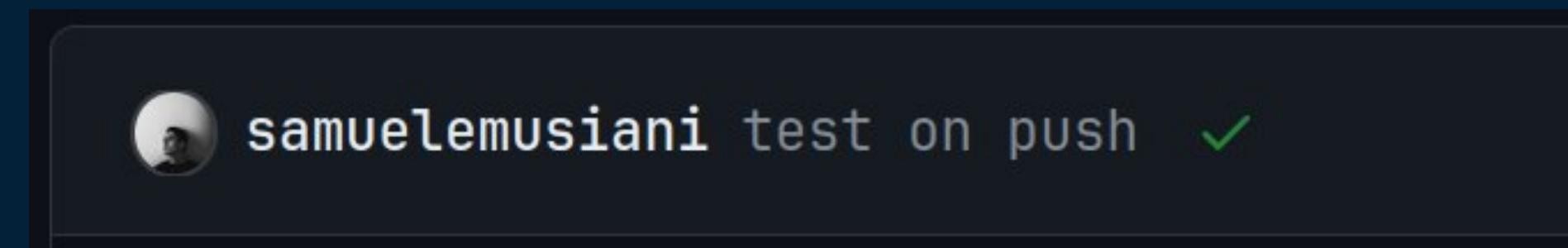
```
...
```





# Action successful

Quando tutti i comandi del workflow vengono eseguiti senza errori, la action termina correttamente.



# Espressioni \${{ ... }}

Le action possono avere variabili e segreti

Per usarli si usa una sintassi speciale formata da   \${{ <nome\_variabile> }}

Esempio:

pippo = [1, 2, 3]

Per usarla: \${{ pippo }}

**Per info vedi la documentazione GitHub:**

**<https://docs.github.com/en/actions/learn-github-actions/expressions>**

# Strategy | matrix

**un job può essere eseguito su specifici range**

Se si vuole testare il progetto su più versioni diverse, invece di creare tante actions è possibile usare le **strategy**.

Il set di passaggi del processo verrà eseguito su ognuna di queste configurazioni.

```
...
jobs:
  node-test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node: [18.x, 19.x, 20.x]
    steps:
      - name: Setup node ${{ matrix.node }}
        uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node }}
```

# If

**lo step a cui si riferisce viene eseguito se la condizione è soddisfatta**

Impedisce l'esecuzione di uno step a meno che non venga soddisfatta una condizione.

if usa un'espressione senza il blocco \${{ ... }}.

...

jobs:

  basic-job:

  strategy:

  matrix:

    os: [macos-latest, ubuntu-latest]

    runs-on: \${{ matrix.os }}

  steps:

    - name: Runner

      run: 'echo \${{ runner.os }}'

    - name: Test linux

      if: runner.os == 'Linux'

      run: 'echo "Siamo su Linux!!!"'

    - name: Test mac

      if: runner.os == 'macOS'

      run: 'echo "Siamo su mac!!!"'

# if

The image shows two side-by-side GitHub Actions job runs. Both runs have a status of "succeeded".

**Job 1: basic-job (macos-latest)**

- Set up job
- Runner**
  - 1 ► Run echo macOS
  - 4 macOS
- Test linux**
- Test mac**
  - 1 ► Run echo "Siamo su mac!!"
  - 4 Siamo su mac!!
- Complete job
- Cleaning up orphan processes

**Job 2: basic-job (ubuntu-latest)**

- Set up job
- Runner**
  - 1 ► Run echo Linux
  - 4 Linux
- Test linux**
  - 1 ► Run echo "Siamo su Linux!!"
  - 4 Siamo su Linux!!
- Test mac**
- Complete job
- Cleaning up orphan processes

**Annotations:**

- A large white arrow points from the word "skipped" to the "Test mac" step in the first job run.
- A large white arrow points from the word "skipped" to the "Test mac" step in the second job run.

# Secrets

**variabili/token/password che possiamo usare in modo sicuro nei jobs**

Può essere necessario a volte usare token, password per usare dei comandi come ssh, API, chiave di deploy...

```
...
jobs:
  basic-job:
    runs-on: ubuntu-latest
    steps:
      - name: Clone repo
        run: ssh debian@cs.unibo.it
        with:
          password: ${{ secrets.NOME_SECRETS }}
```

# Secrets

Per usarli sono da inserire all'interno delle “Settings” della repository.

The screenshot shows the GitHub repository settings page. At the top, there are several navigation links: Issues (1), Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. The Settings link is highlighted with a yellow circle. On the left, a sidebar lists various repository management options: General, Access, Collaborators, Code and automation, Branches, Tags, Rules, Actions, Webhooks, Environments, Codespaces, Pages, Security, Code security and analysis, Deploy keys, and Secrets and variables. A large yellow arrow points from the text in the main content area to the 'Secrets and variables' link in the sidebar. In the main content area, there are two sections: 'Actions secrets and variables' and 'Repository secrets'. The 'Actions secrets and variables' section contains a description of what secrets and variables are, a note about workflow triggers, and tabs for 'Secrets' (selected) and 'Variables'. The 'Repository secrets' section shows a message stating 'This repository has no secrets.' and a green button labeled 'New repository secret'.

Actions secrets and variables

Secrets and variables allow you to manage reusable configuration data. Secrets are **encrypted** and are used for sensitive data. [Learn more about encrypted secrets](#). Variables are shown as plain text and are used for **non-sensitive** data. [Learn more about variables](#).

Anyone with collaborator access to this repository can use these secrets and variables for actions. They are not passed to workflows that are triggered by a pull request from a fork.

Secrets    Variables

Environment secrets

This repository has no environment secrets.

Manage environment secrets

Repository secrets

This repository has no secrets.

New repository secret

Proviamo a creare delle actions per la repository [cartabinaria/lab-git2](#)  
creiamo una fork della repo

Cloniamo la fork sul nostro pc in locale

```
$ git clone git@github.com:<nome_utente>/lab-git2
```

Se avete già clonato la fork in locale nel precedente laboratorio:

1. aprite la fork su github
2. premere su “sync fork”
3. premere su “update fork”
4. Aprite la repository in locale e mandate un *git pull*

Nella nostra repository vediamo che la cartella *workflow* è già stata creata da noi, se la apriamo troviamo il nostro *basic\_workflow.yaml*

# Creiamo il nostro primo workflow

Vogliamo realizzare un'aktion per compilare *file.cpp*

1. creo il file *compile.yaml*
2. diamo un **nome** all'aktion
3. vogliamo che **per ogni commit** *file.cpp* compili
  - a. **clonare la repo,**

usando **un'aktion già pronta**

```
name: Compile c++  
on:  
  push  
  
jobs:  
  basic-job:  
    runs-on: ubuntu-latest  
    steps:  
      - name: Clone the repo  
        uses: actions/checkout@v4
```

# Creiamo il nostro primo workflow

Vogliamo realizzare un'aktion per compilare *file.cpp*

1. creo il file *compile.yaml*
2. diamo un **nome** all'aktion
3. vogliamo che **per ogni commit** *file.cpp* compili
  - a. **clonare la repo**,  
usando **un'aktion già pronta**
  - b. **installiamo il compilatore g++**

```
name: Compile c++  
  
on:  
  push  
  
jobs:  
  basic-job:  
    runs-on: ubuntu-latest  
    steps:  
      - name: Clone the repo  
        uses: actions/checkout@v4  
  
      - name: Install g++  
        run: sudo apt install gcc
```

# Creiamo il nostro primo workflow

Vogliamo realizzare un'aktion per compilare *file.cpp*

1. creo il file *compile.yaml*
2. diamo un **nome** all'aktion
3. vogliamo che **per ogni commit** *file.cpp* compili
  - a. **clonare la repo**,  
usando **un'aktion già pronta**
  - b. **installiamo il compilatore g++**
  - c. usiamo il classico comando per **compilare il file**

```
name: Compile c++  
  
on:  
  push  
  
jobs:  
  basic-job:  
    runs-on: ubuntu-latest  
    steps:  
      - name: Clone the repo  
        uses: actions/checkout@v4  
  
      - name: Install g++  
        run: sudo apt install gcc  
  
      - name: Compile the file  
        run: g++ file.cpp
```

# Needs

**un workflow per essere eseguito ha bisogno che prima ne sia eseguito un altro**

- Identifica tutti i processi che devono essere completati correttamente prima di essere eseguiti.
- Può essere una stringa o una matrice di stringhe.
- Se un processo ha esito negativo, tutti i processi che lo richiedono vengono ignorati, a meno che non utilizzino un'istruzione condizionale che determina la continuazione del processo.



```
...  
jobs:  
  basic-job:  
    runs-on: ubuntu-latest  
    steps:  
      - name: Clone repo  
        run: ssh debian@cs.unibo.it  
...  
  build:  
    name: Build the project  
    runs-on: ubuntu-latest  
    needs: basic-job  
    steps:  
      ...
```

# GitHub Action Limits

Product	Storage	Minutes per month
<b>GitHub</b>	<b>500 MB</b>	<b>2,000</b>
<b>GitHub Pro</b>	<b>1 GB</b>	<b>3,000</b>
<b>GitHub Free for organizations</b>	<b>500 MB</b>	<b>2,000</b>
<b>GitHub Team</b>	<b>2 GB</b>	<b>3,000</b>
<b>GitHub Enterprise Cloud</b>	<b>50 GB</b>	<b>50,000</b>

# git lfs

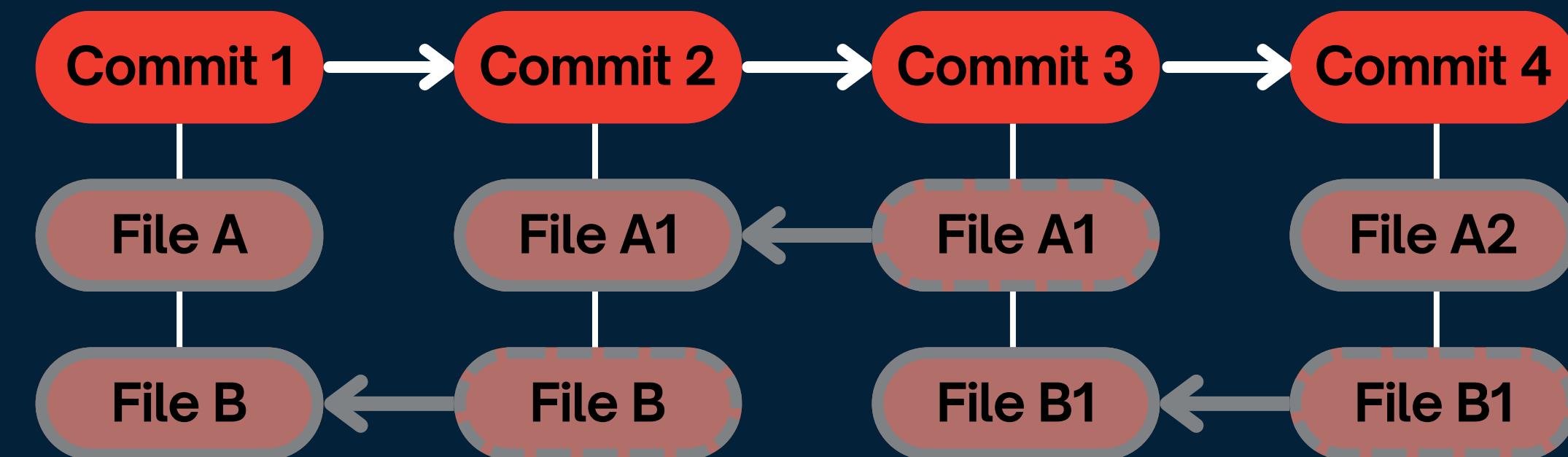


# git lfs

Git è pensato per lavorare solo con file di testo.

Tutti gli algoritmi di diff sono per file di testo.

Con file binari (come pdf, immagini, audio, video... ) non riesce a fare le differenze con la versione precedente: se viene modificato il file, git crea una nuova versione del file da capo.



# git lfs large file storage

Se si volesse sviluppare un videogioco, i file audio, le immagini, ecc... appesantirebbero la history di git.  
Usando **git lfs** risolviamo questo problema.

Invece di salvare direttamente il file nella history di git, viene salvato un “puntatore” in forma testuale.  
I file binari vengono salvati su un server a parte e vengono inclusi nella repository tramite questi  
“puntatori”.

A livello utente non cambia quasi nulla (infatti continueremo a vedere i binari nella repository) ma git  
internamente sarà molto più veloce perché nella history non avrà binari.

# Installazione

Se non avete già installato git lfs, andate sul sito per scaricarlo: <https://git-lfs.com/>

Per verificare se lo avete aprite un terminale e provate

```
$ git-lfs
```

Una volta installato eseguite il comando per inizializzare git

```
$ git lfs install
```

[questo comando va eseguito solo una volta, dura “per sempre”]



# Creiamo la prima repository con lfs

Creiamo una repository da GitHub e cloniamola in locale.

Per usare git lfs su tutti i file .png usiamo

```
$ git lfs track "*.png"
```

Ci creerà un file *.gitattributes* con dentro:

```
*.png filter=lfs diff=lfs merge=lfs -text
```

In questo modo git sa che tutti i file con pattern \*.png devono essere gestiti con lfs.

Ora possiamo usare tutti i soliti comandi git come prima, in automatico sarà git a gestire il tutto.

# git lfs & GitHub

GitHub ci limita nell'uso di git lfs: limiti di storage e di band\_width  
possiamo vedere nelle impostazioni > Billing and Plans

