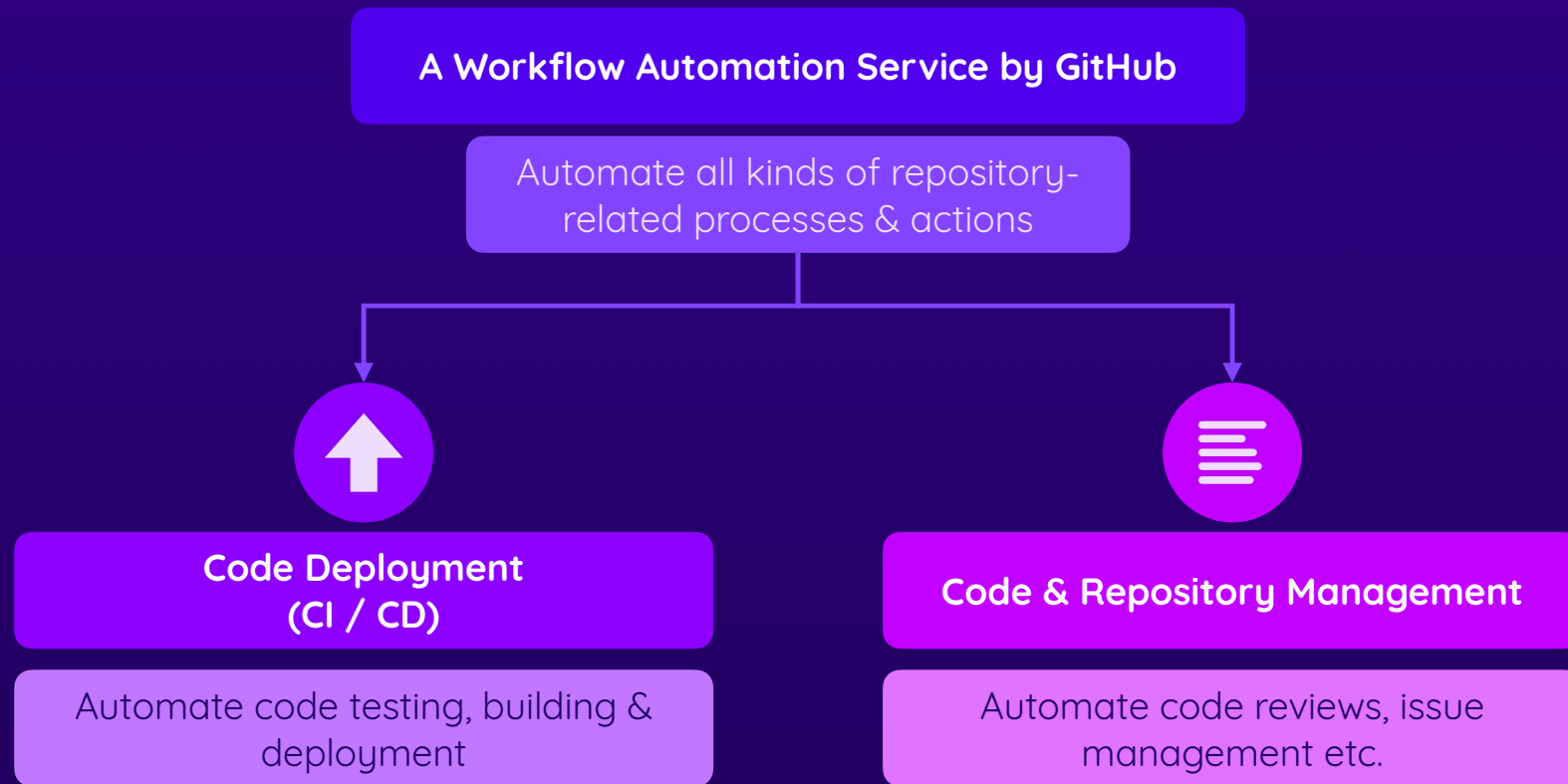


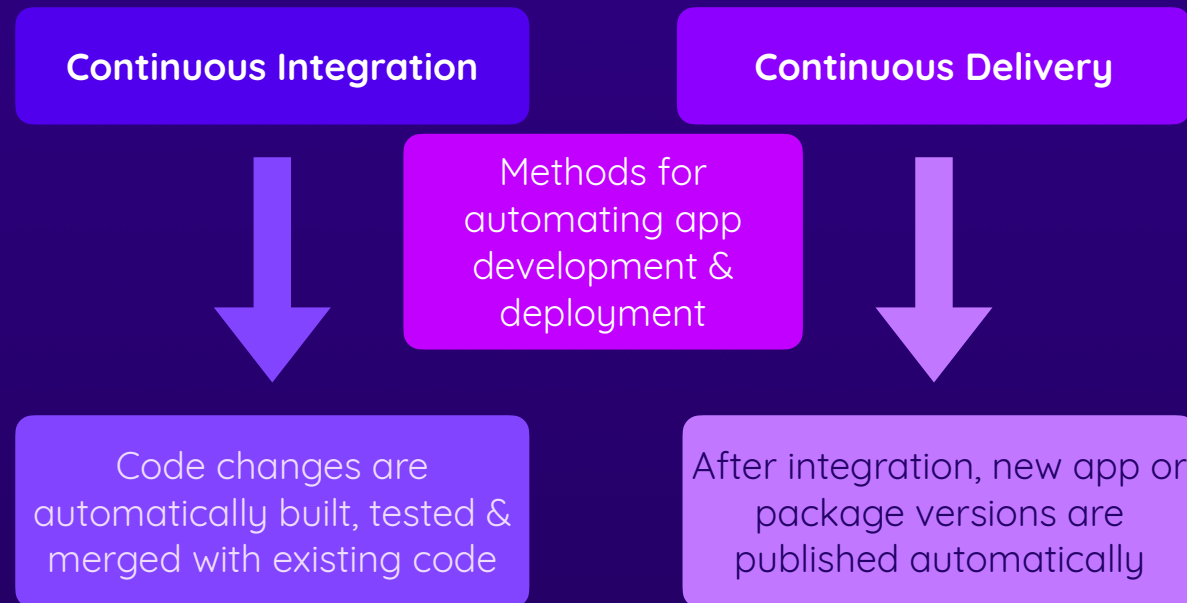
# What Is “GitHub Actions”?



# What is GitHub?

And what are “repositories”?

# What's CI / CD?



# A Typical CI / CD Workflow

**Code was changed**  
(e.g, new feature added)



**New Git Commit**



**Pushed to GitHub**  
Repository

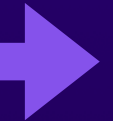
Can be configured & executed  
via GitHub Actions

CI / CD Workflow

**App is built**

**App is tested**  
(e.g., unit tests)

**App is published**  
(e.g., on AWS EC2)



# GitHub Actions Alternatives

GitHub Actions

For CI / CD



Alternatives

Jenkins

GitLab CI/CD

Azure Pipelines

AWS CodePipeline

and many more ...

# What is Git?

# What Is GitHub?



A cloud Git repository & services  
provider

Store & manage Git repositories

# What Is Git?

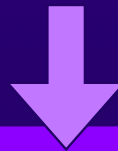


A (free) version control system

A tool for managing source code changes



Save code snapshots  
("commits")



Work with alternative code  
versions ("branches")



Move between branches &  
commits

With Git, you can easily roll back to older code snapshots or  
develop new features without breaking production code.



# What Is GitHub?



A cloud Git repository & services provider

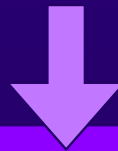
Store & manage Git repositories



Cloud Git repository storage  
("push" & "pull")

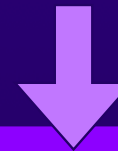
Backup, work across  
machines & work in teams

Public & private, team  
management & more



Code management &  
collaborative development

Via "Issues", "Projects", "Pull  
Requests" & more



Automation & CI / CD

Via **GitHub Actions**, GitHub  
Apps & more



# About This Course

Learn GitHub Actions From The Ground Up



## Video-based Explanations

Watch the videos—at your pace

Recommendation: Watch all videos in the provided order

Repeat videos as needed



## Practice & Experiment

Pause videos and practice on your own

Build up on course examples & feel free to experiment

Build your own demo projects & workflow examples



## Learn & Grow Together

Help each other in the course Q&A section

Dive into our (free!) community

# Git & GitHub Crash Course

## The Very Basics

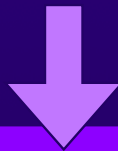
- ▶ Working with Git: Setup & Key Commands
- ▶ Working with GitHub: Creating & Using Repositories
- ▶ Using Git with GitHub

# What Is Git?

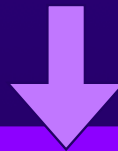


A (free) version control system

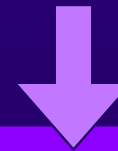
A tool for managing source code changes



Save code snapshots  
(**“commits”**)



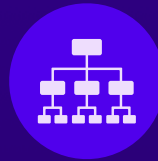
Work with alternative code  
versions (**“branches”**)



Move between branches &  
commits

With Git, you can easily roll back to older code snapshots or  
develop new features without breaking production code.

# Git Repositories



Git features can be used in projects  
with Git repositories



A repository is a folder used by Git to  
track all changes of a given project

Git commands require a  
repository in a project

Create Git repositories via `git init`

Only required once per  
folder / project

Some projects initialize Git for you

e.g., React projects

# Working with Commits (Code Snapshots)

## Create Commits

```
git add <file(s)>
```

Stage changes for next  
commit



```
git commit
```

Create a commit that  
includes all staged changes

## Move between Commits

```
git checkout <id>
```

Temporarily move to  
another commit



# Working with Commits (Code Snapshots)

## Create Commits

```
git add <file(s)>
```

Stage changes for next commit



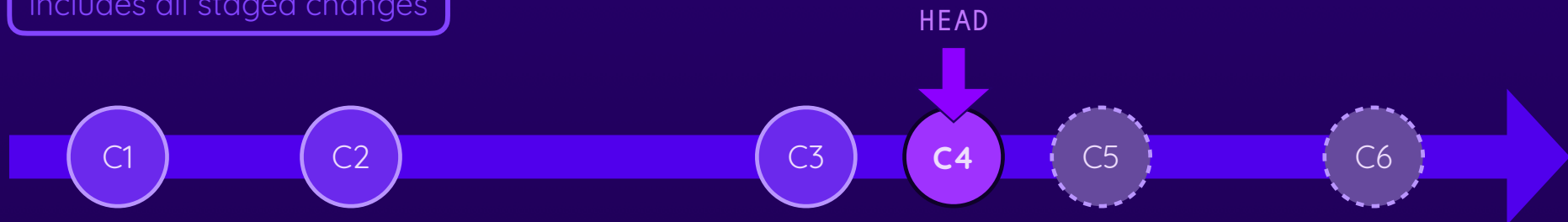
```
git commit
```

Create a commit that includes all staged changes

## Move between Commits

```
git checkout <id>
```

Temporarily move to another commit



# Understanding Staging

Staging controls which changes are part of a commit



With staging, you can make sure that not all code changes made are added to a snapshot

If all changes should be included, you can use `git add .` to include all files in a Git repository



# Working with Commits (Code Snapshots)

## Create Commits

```
git add <file(s)>
```

Stage changes for next commit



```
git commit
```

Create a commit that includes all staged changes

## Move between Commits

```
git checkout <id>
```

Temporarily move to another commit



# Working with Commits (Code Snapshots)

## Create Commits

```
git add <file(s)>
```

Stage changes for next commit



```
git commit
```

Create a commit that includes all staged changes

## Move between Commits

```
git checkout <id>
```

Temporarily move to another commit

## Undo Commits

```
git revert <id>
```

Revert changes of commit by creating a new commit



# Working with Commits (Code Snapshots)

## Create Commits

```
git add <file(s)>
```

Stage changes for next commit



```
git commit
```

Create a commit that includes all staged changes

## Move between Commits

```
git checkout <id>
```

Temporarily move to another commit

## Undo Commits

```
git revert <id>
```

Revert changes of commit by creating a new commit



# Working with Commits (Code Snapshots)

## Create Commits

```
git add <file(s)>
```

Stage changes for next commit



```
git commit
```

Create a commit that includes all staged changes

## Move between Commits

```
git checkout <id>
```

Temporarily move to another commit

## Undo Commits

```
git revert <id>
```

Revert changes of commit by creating a new commit

```
git reset --hard <id>
```

Undo changes by deleting all commits since <id>



# Working with Commits (Code Snapshots)

## Create Commits

```
git add <file(s)>
```

Stage changes for next commit



```
git commit
```

Create a commit that includes all staged changes

## Move between Commits

```
git checkout <id>
```

Temporarily move to another commit

## Undo Commits

```
git revert <id>
```

Revert changes of commit by creating a new commit

```
git reset --hard <id>
```

Undo changes by deleting all commits since <id>



# Key Commands

```
git init
```

**Initialize a Git repository** (only required once per project)

```
git add <file(s)>
```

**Stage code changes** (for the next commit)

```
git commit -m "..."
```

**Create a commit for the staged changes** (with a message)

```
git status
```

**Get the current repository status** (e.g., which changes are staged)

```
git log
```

**Output a chronologically ordered list of commits**

```
git checkout <id>
```

**Temporarily move back to commit <id>**

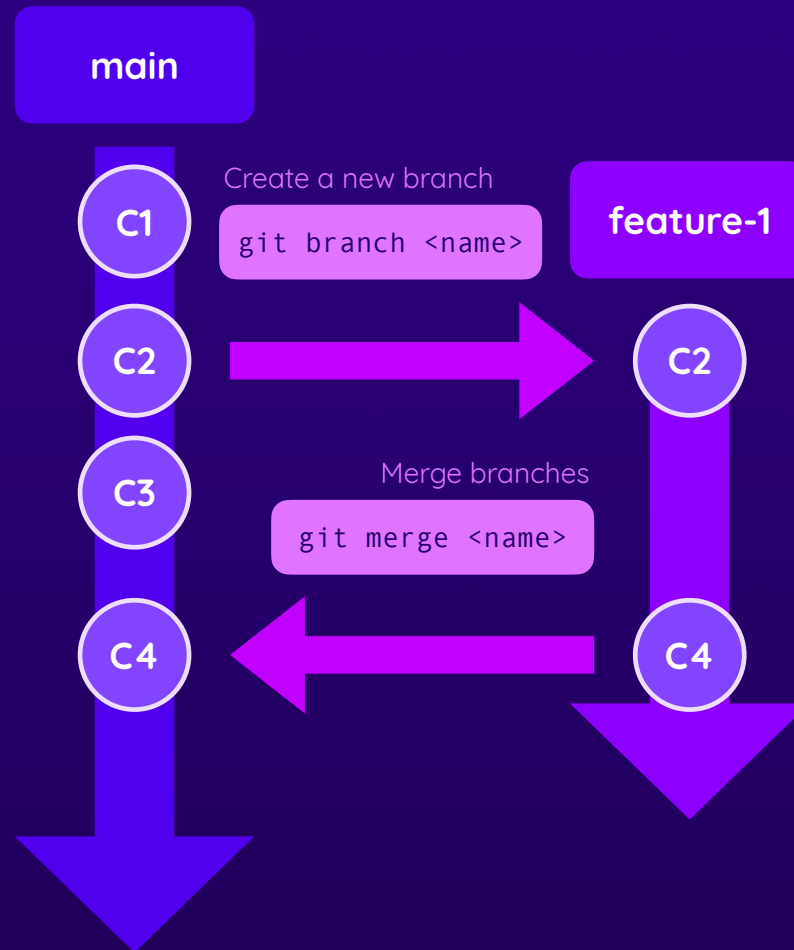
```
git revert <id>
```

**Revert the changes of commit <id>** (by creating a new commit)

```
git reset <id>
```

**Undo commit(s) up to commit <id> by deleting commits**

# Understanding Git Branches



# What Is GitHub?



A cloud Git repository & services provider

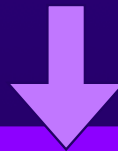
Store & manage Git repositories



Cloud Git repository storage  
("push" & "pull")

Backup, work across  
machines & work in teams

Public & private, team  
management & more



Code management &  
collaborative development

Via "Issues", "Projects", "Pull  
Requests" & more

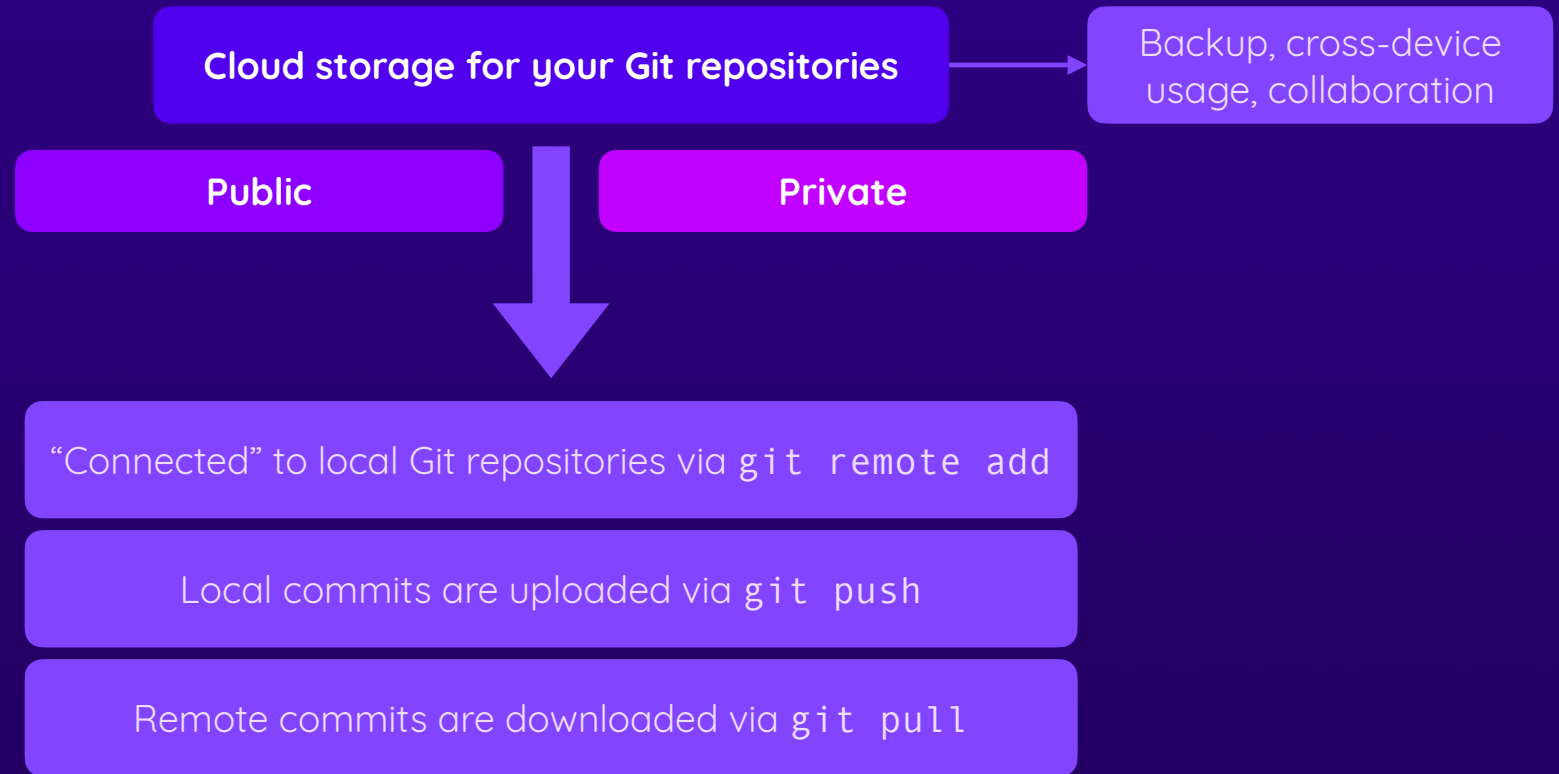


Automation & CI / CD

Via **GitHub Actions**, GitHub  
Apps & more



# GitHub Repositories



# Forking & Pull Requests



## Repository Forking

Creates a standalone copy of a repository

Can be used to work on code without affecting the original repository



## Pull Requests

Requests merging code changes into a branch

Can be based on a forked repository or another branch from the same repository

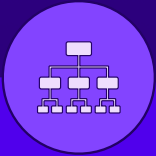
Pull requests allow for code reviews before merging changes

# GitHub Actions: Fundamentals

## Key Building Blocks & Usage

- ▶ Understanding the Key Elements
- ▶ Working with Workflows, Jobs & Steps
- ▶ Building an Example Workflow

# Key Elements



## Workflows

Attached to a GitHub repository

Contain one or more **Jobs**

Triggered upon **Events**



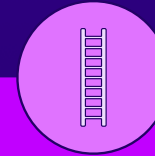
## Jobs

Define a **Runner** (execution environment)

Contain one or more **Steps**

Run in parallel (default) or sequential

Can be conditional



## Steps

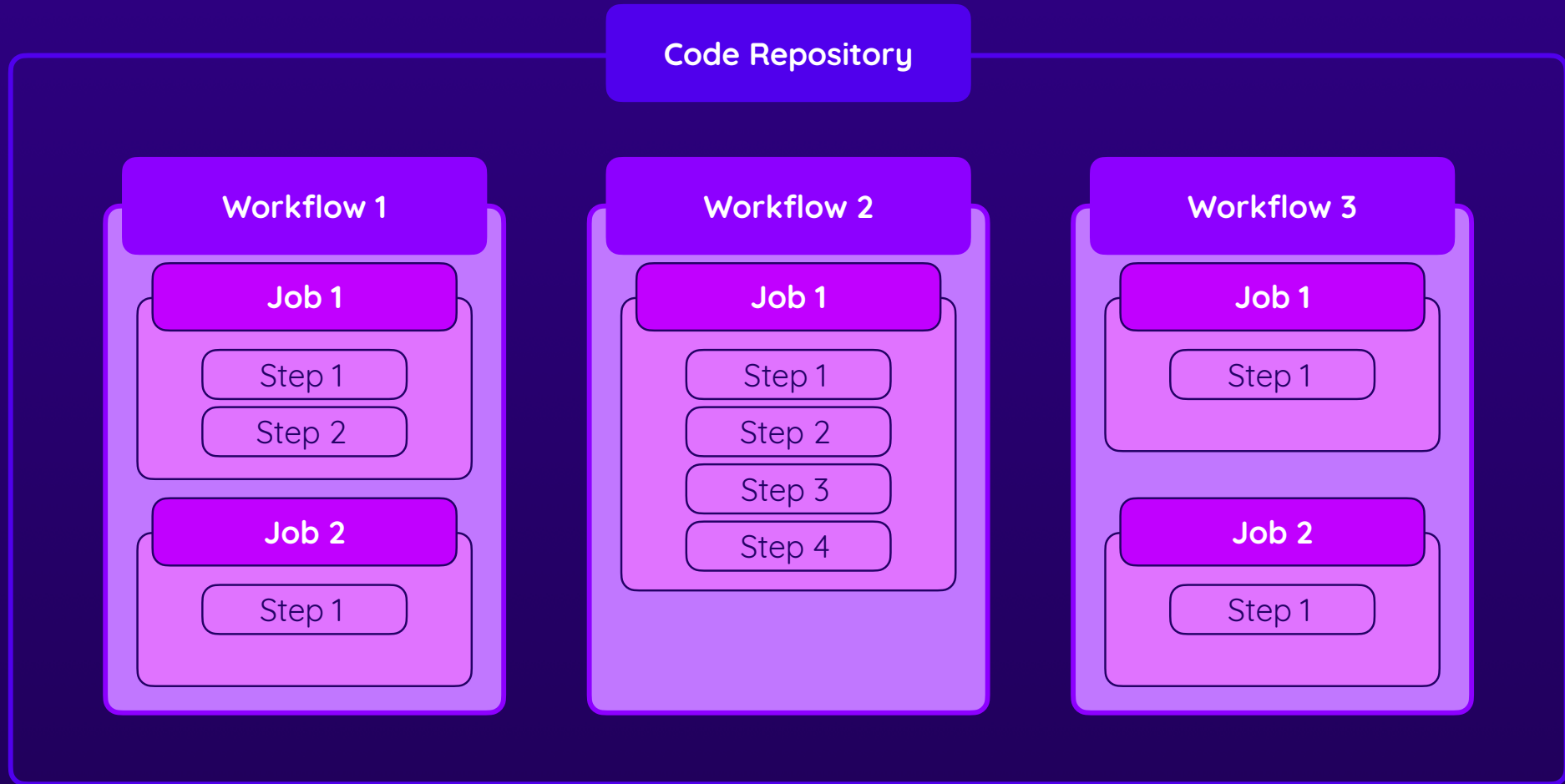
Execute a **shell script** or an **Action**

Can use custom or third-party actions

Steps are executed in order

Can be conditional

# Workflows, Jobs & Steps



# Events (Workflow Triggers)

Repository-related			Other
<b>push</b> Pushing a commit	<b>pull_request</b> Pull request action (opened, closed, ...)	<b>create</b> A branch or tag was created	<b>workflow_dispatch</b> Manually trigger workflow
<b>fork</b> Repository was forked	<b>issues</b> An issue was opened, deleted, ...	<b>issue_comment</b> Issue or pull request comment action	<b>repository_dispatch</b> REST API request triggers workflow
<b>watch</b> Repository was starred	<b>discussion</b> Discussion action (created, deleted, ...)	<b>Many More!</b>	<b>schedule</b> Workflow is scheduled
			<b>workflow_call</b> Can be called by other workflows

# What Are Actions?

**Command**  
("run")



A (typically simple) shell command  
that's defined by you

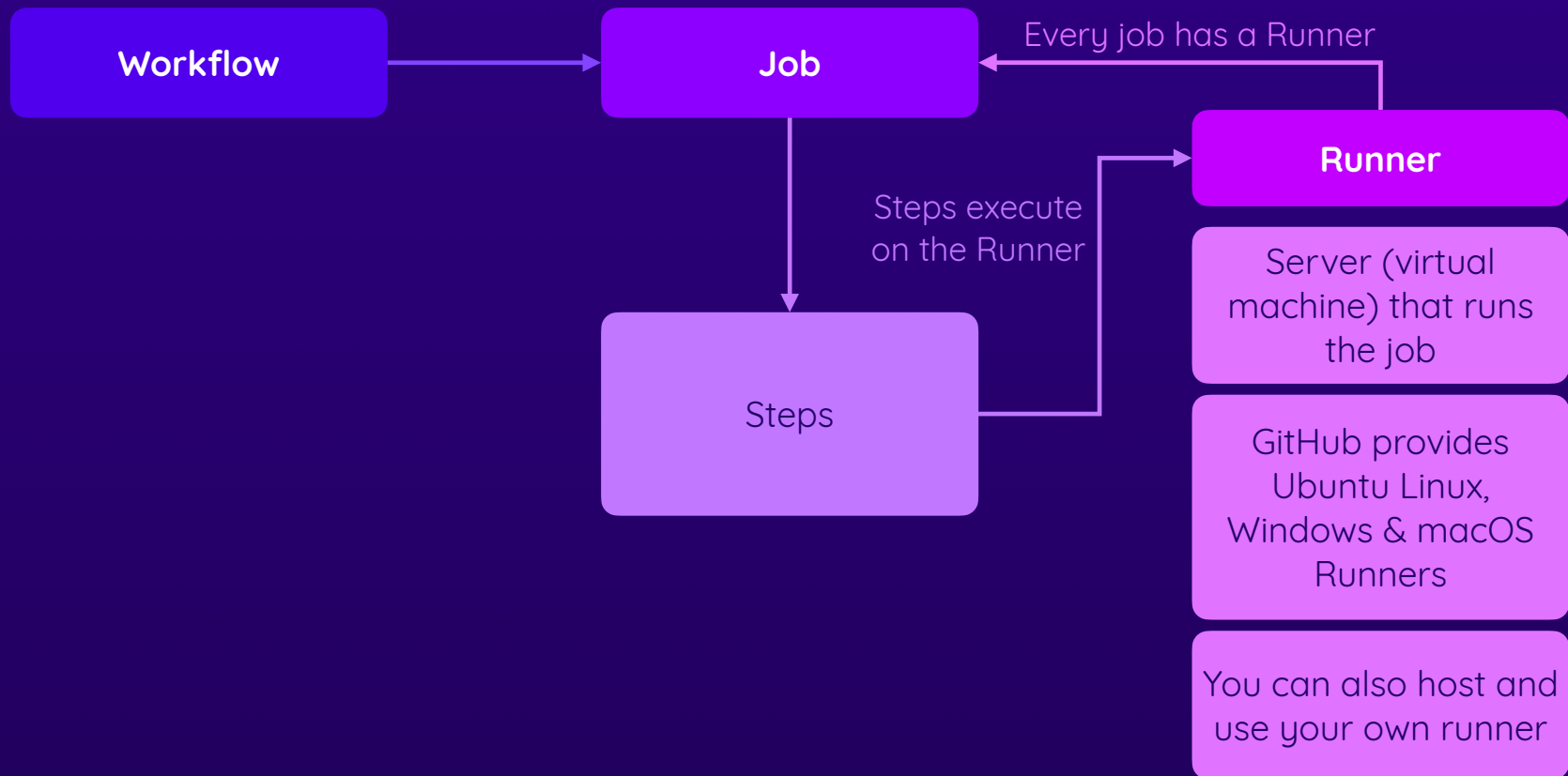
**Action**



A (custom) application that  
performs a (typically complex)  
frequently repeated task

You can build your own Actions but  
you can also use official or  
community Actions

# Job Runners





# Module Summary

## Core Components

Workflows: Define Events + Jobs

Jobs: Define Runner + Steps

Steps: Do the actual work

## Defining Workflows

.github/workflows/<file>.yaml  
(on GitHub or locally)

GitHub Actions syntax must be followed

## Events / Triggers

Broad variety of events  
(repository-related & other)

Workflows have at least one (but possible more) event(s)

## Runners

Servers (machines) that execute the jobs

Pre-defined Runners (with different OS) exist

You can also create custom Runners

## Workflow Execution

Workflows are executed when their events are triggered

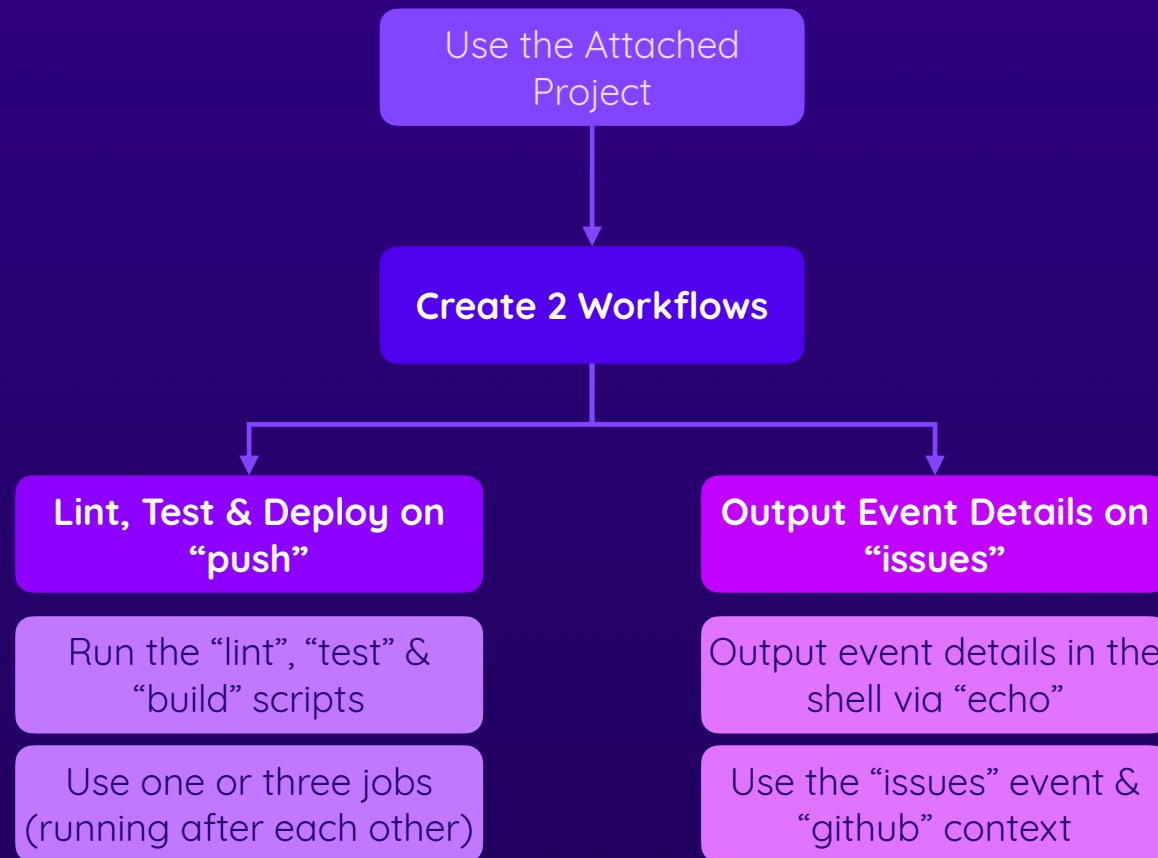
GitHub provides detailed insights into job execution (+ logs)

## Actions

You can run shell commands

But you can also use pre-defined Actions (official, community or custom)

# Exercise Time!



# Events: A Closer Look

## Diving Deeper Into Workflow Triggers

- ▶ Controlling Workflow Execution with Event Filters
- ▶ Detailed Control with Activity Types
- ▶ Examples!

# Available Events

## Repository-related

### **push**

Pushing a commit

### **pull\_request**

Pull request action  
(opened, closed, ...)

### **create**

A branch or tag was  
created

### **fork**

Repository was  
forked

### **issues**

An issue was opened,  
deleted, ...

### **issue\_comment**

Issue or pull request  
comment action

### **watch**

Repository was  
starred

### **discussion**

Discussion action  
(created, deleted, ...)

**Many More!**

## Other

### **workflow\_dispatch**

Manually trigger  
workflow

### **repository\_dispatch**

REST API request  
triggers workflow

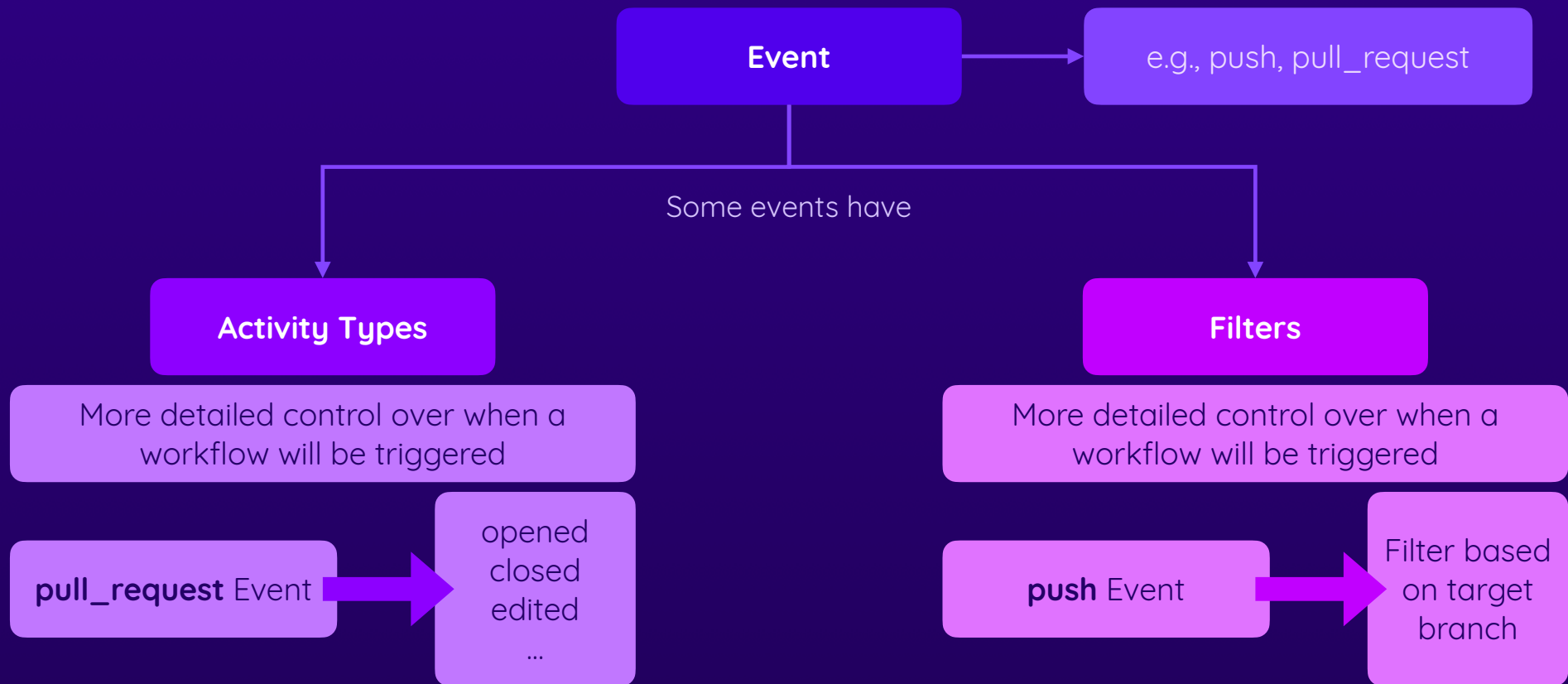
### **schedule**

Workflow is scheduled

### **workflow\_call**

Can be called by other  
workflows

# Event Activity Types & Filters



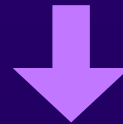
# A Note About Fork Pull Request Workflows

By default, Pull Requests based on Forks do NOT trigger a workflow



Reason: Everyone can fork & open pull requests

Malicious workflow runs & excess cost could be caused



First-time contributors must be approved manually

# Cancelling & Skipping Workflow Runs



## Cancelling

By default, Workflows get cancelled if Jobs fail

By default, a Job fails if at least one Step fails

You can also cancel workflows manually



## Skipping

By default, all matching events start a workflow

Exceptions for “push” & “pull\_request”

Skip with proper commit message

# Module Summary

## Available Events

There are many supported events

Most are repository-related (e.g., push, pull\_request)

But some are more general (e.g., schedule)

## Pull Requests & Forks

Initial approval needed for pull requests from forked repositories

Avoids spam from untrusted contributors

## Activity Types

The exact type of event that should trigger a workflow

Examples: Opening or editing a pull request should trigger the wf

## Cancelling & Skipping

Workflows get cancelled automatically when jobs fail

You can manually cancel workflows

You can skip via [skip ci] etc. in commit message

## Event Filters

For push & pull\_request: Add filters to avoid some executions

Filter based on target branch and / or affected file paths



# Exercise Time!

1

## Test & Deploy For “main” Pushes & Manual Trigger

**Goal:** Run Workflow if a  
commit is pushed to the  
“main” branch

Ignore push events for other  
branches

### Variation

Also run for “dev” branch and  
DON'T run if workflow files  
were edited

2

## Run Tests Upon Pull Requests

**Goal:** Run Workflow if a  
collaborator pull\_request is  
opened

Only run for pull\_requests  
targeting the “main” branch

### Variation

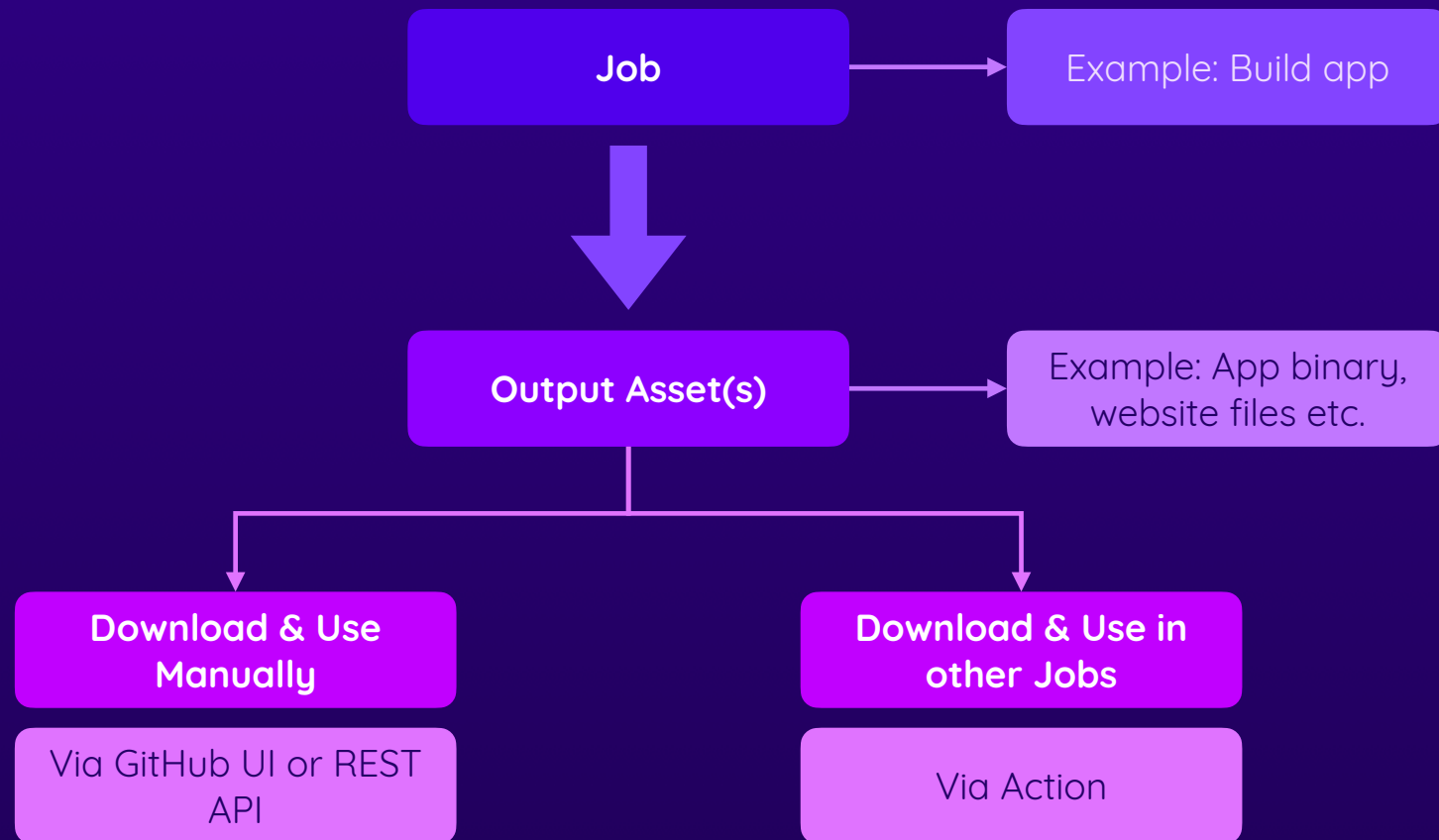
Also run for “dev” branch

# Job Data & Outputs

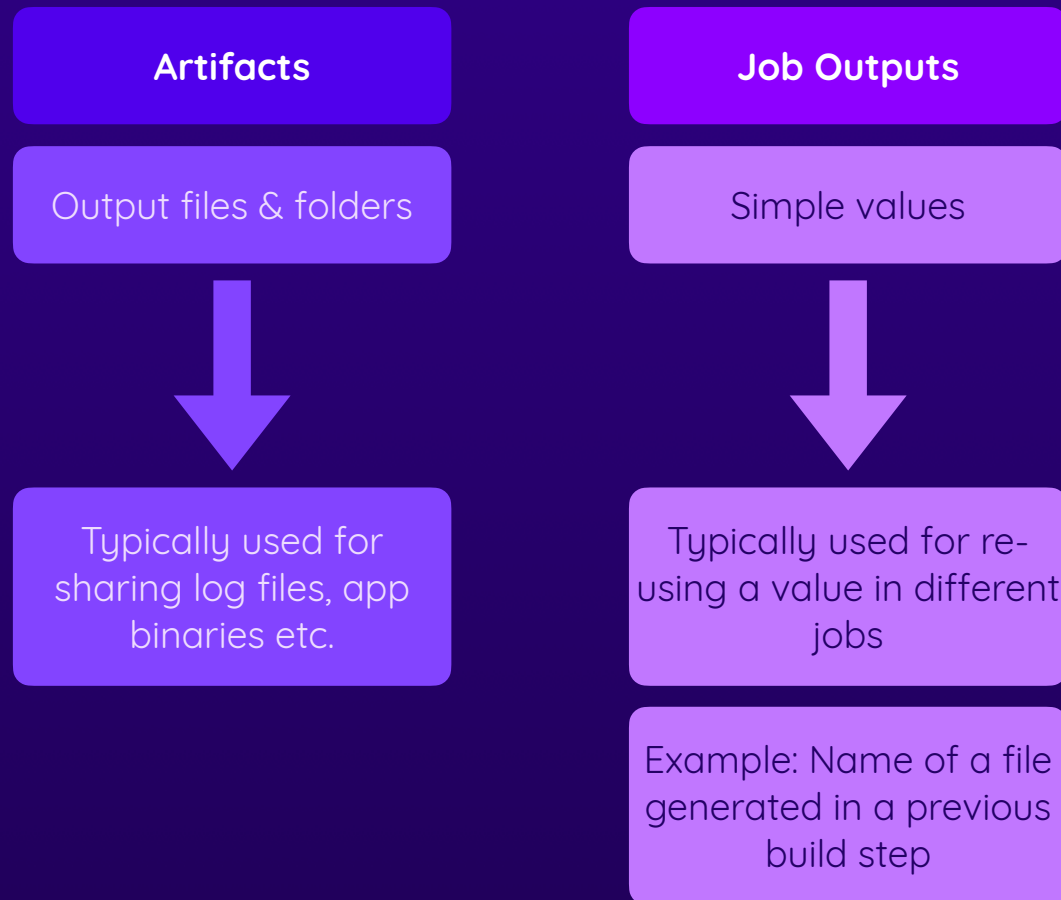
It's All About Data!

- ▶ Working with Artifacts
- ▶ Working with Job Outputs
- ▶ Caching Dependencies

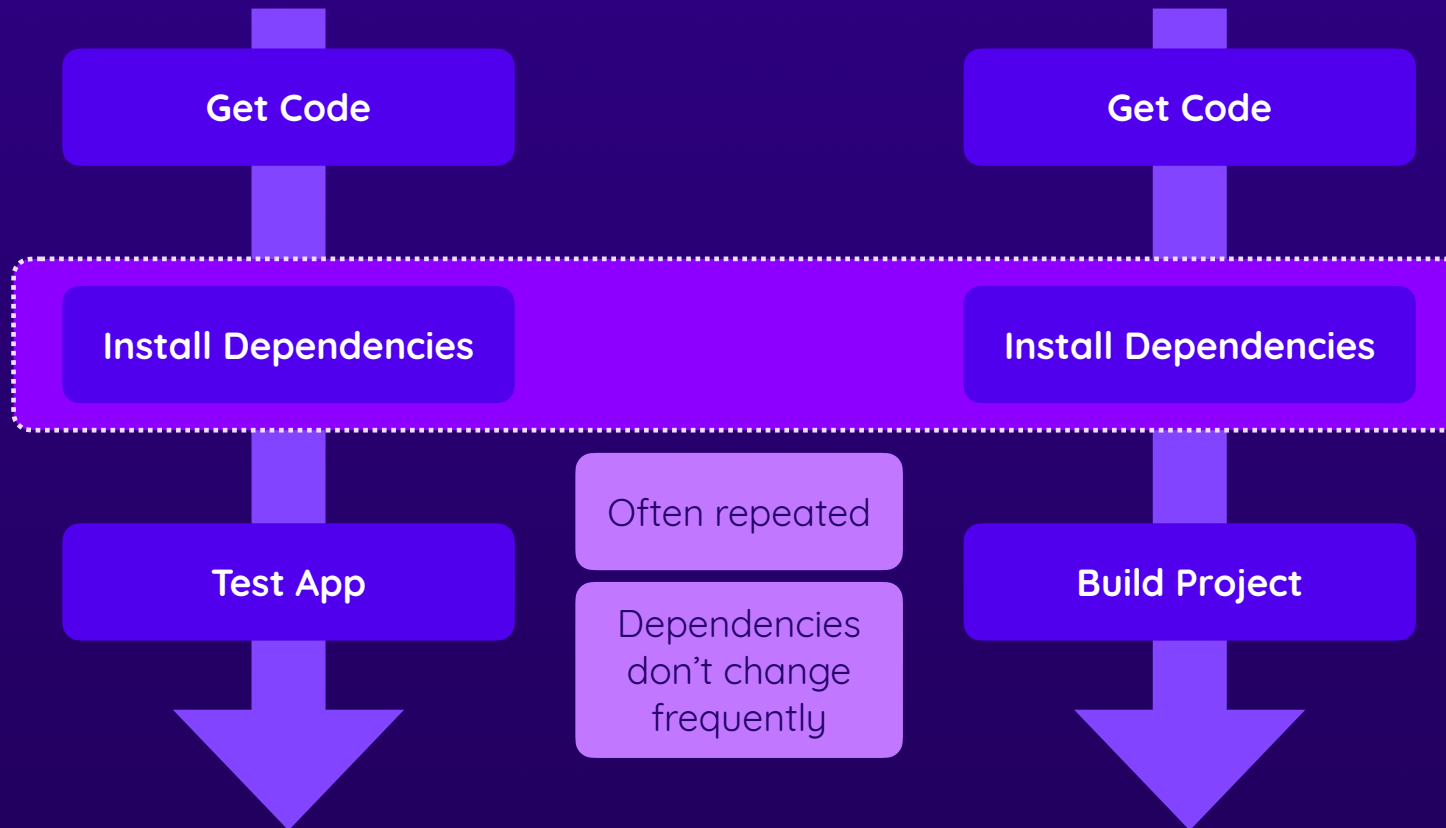
# Understanding Job Artifacts



# Understanding Job Outputs



# Caching Dependencies



# Module Summary

## Artifacts

Jobs often produce assets that should be shared or analyzed

Examples: Deployable website files, logs, binaries etc.

These assets are referred to as “Artifacts” (or “Job Artifacts”)

GitHub Actions provides Actions for uploading & downloading

## Outputs

Besides Artifacts, Steps can produce and share simple values

These outputs are shared via `::set-output`

Jobs can pick up & share Step outputs via the `steps` context

Other Jobs can use Job outputs via the `needs` context

## Caching

Caching can help speed up repeated, slow Steps

Typical use-case: Caching dependencies

But any files & folder can be cached

The `cache` Action automatically stores & updates cache values (based on the cache key)

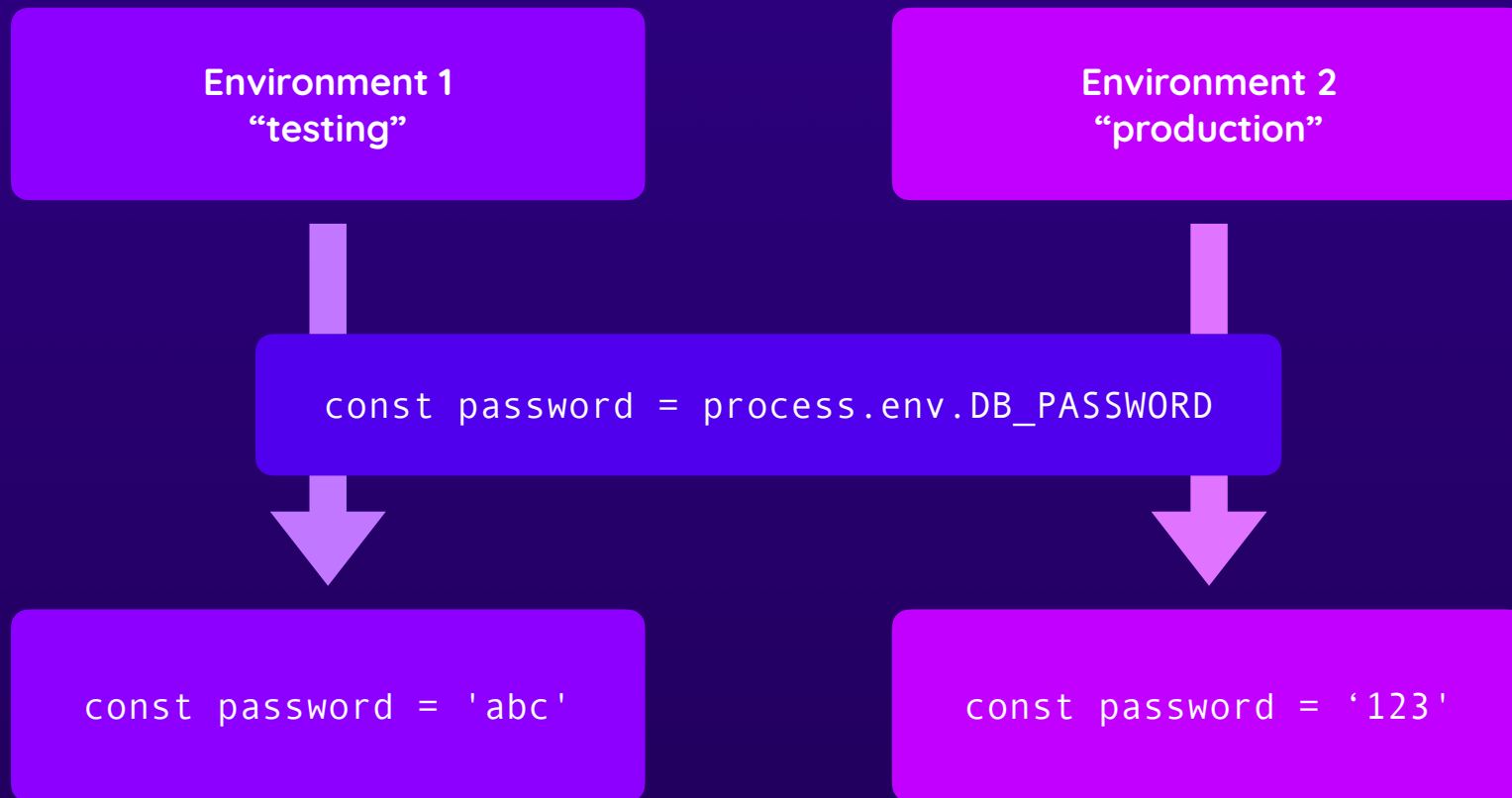
Important: Don't use caching for artifacts!

# Environment Variables & Secrets

Hardcoding Is Not (Often) The Solution

- ▶ Understanding & Using Environment Variables
- ▶ Using Secrets
- ▶ Utilizing Job Environments

# Understanding Environment Variables





# Environment Variables vs Secrets

Some environment variable values  
should never be exposed



Example: Database access password

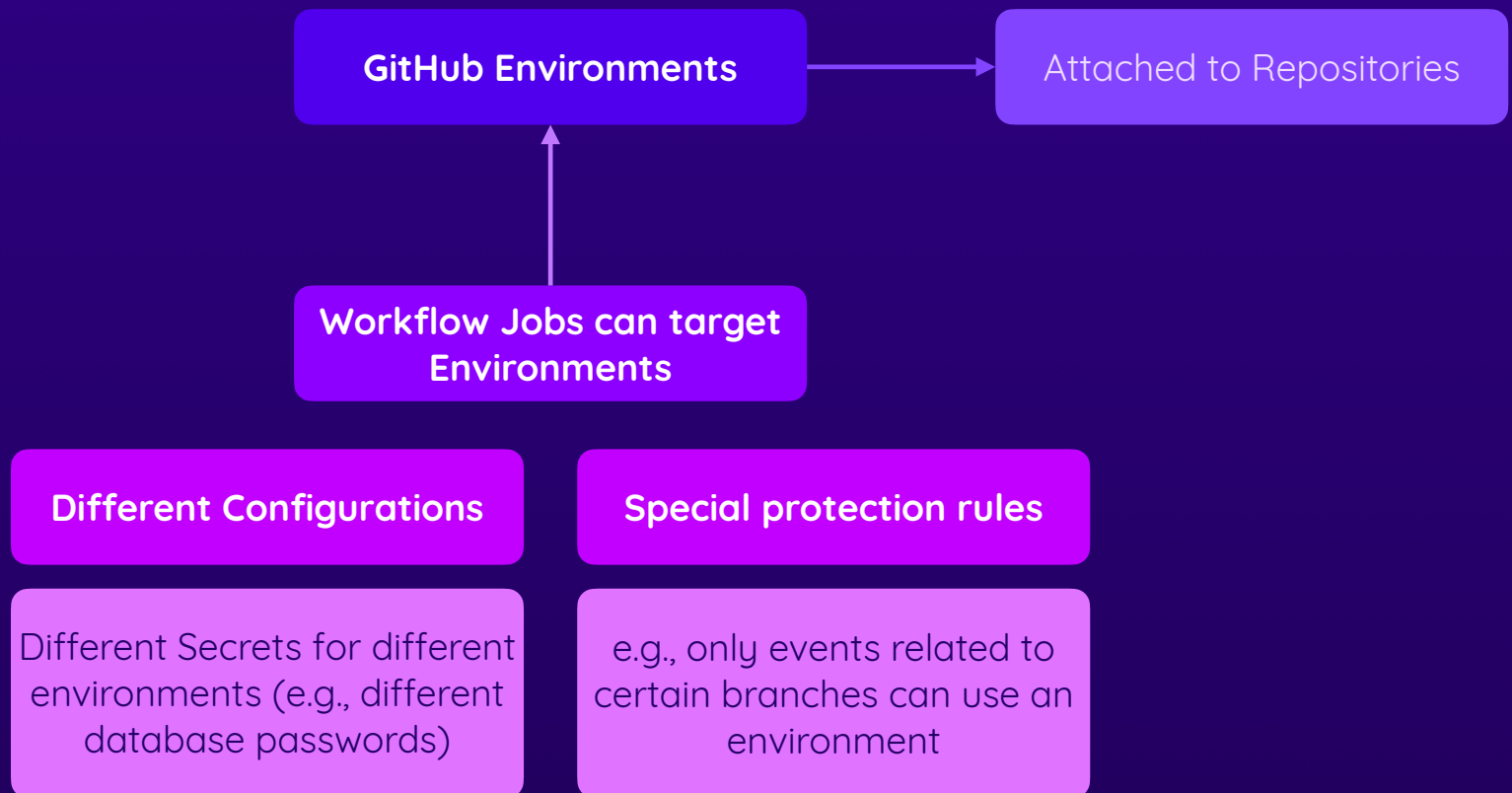


**Use Secrets**



Together with  
environment variables

# GitHub Repository Environments



# Module Summary

## Environment Variables

Dynamic values used in code  
(e.g., database name)

May differ from workflow to  
workflow

Can be defined on Workflow-,  
Job- or Step-level

Can be used in code and in the  
GitHub Actions Workflow

Accessible via interpolation and  
the `env` context object

## Secrets

Some dynamic values should not  
be exposed anywhere

Examples: Database credentials,  
API keys etc.

Secrets can be stored on  
Repository-level or via  
Environments

Secrets can be referenced via the  
`secrets` context object

## GitHub Actions Environments

Jobs can reference different  
GitHub Actions Environments

Environments allow you to set up  
extra protection rules

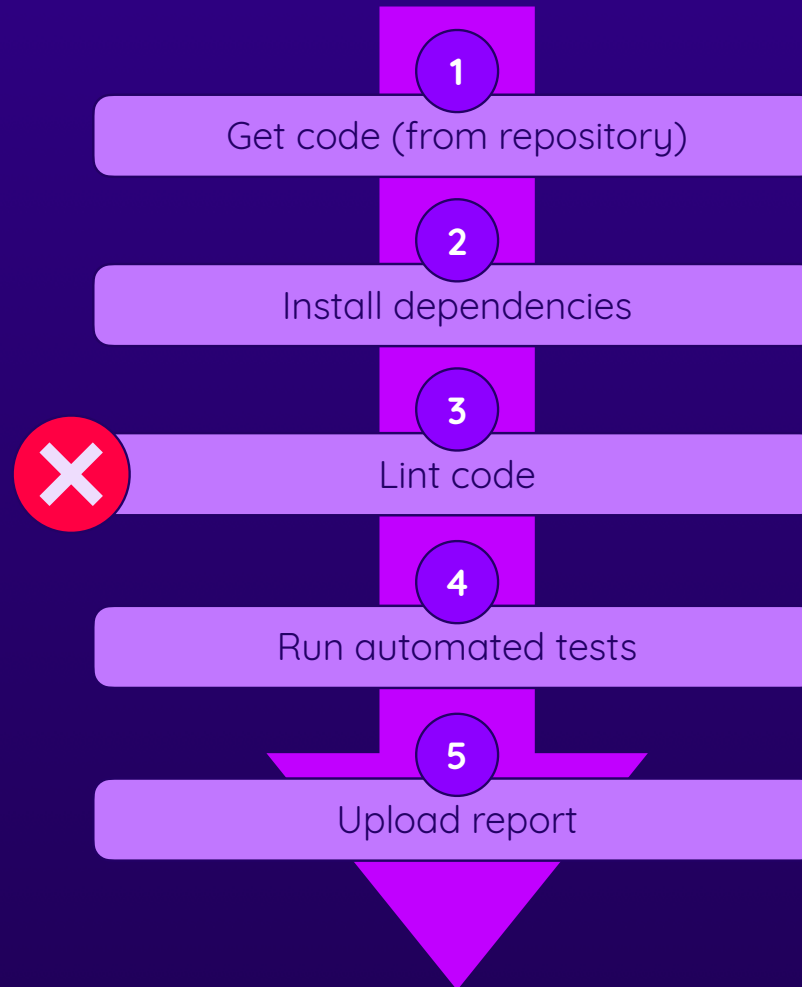
You can also store Secrets on  
Environment-level

# Controlling Execution Flow

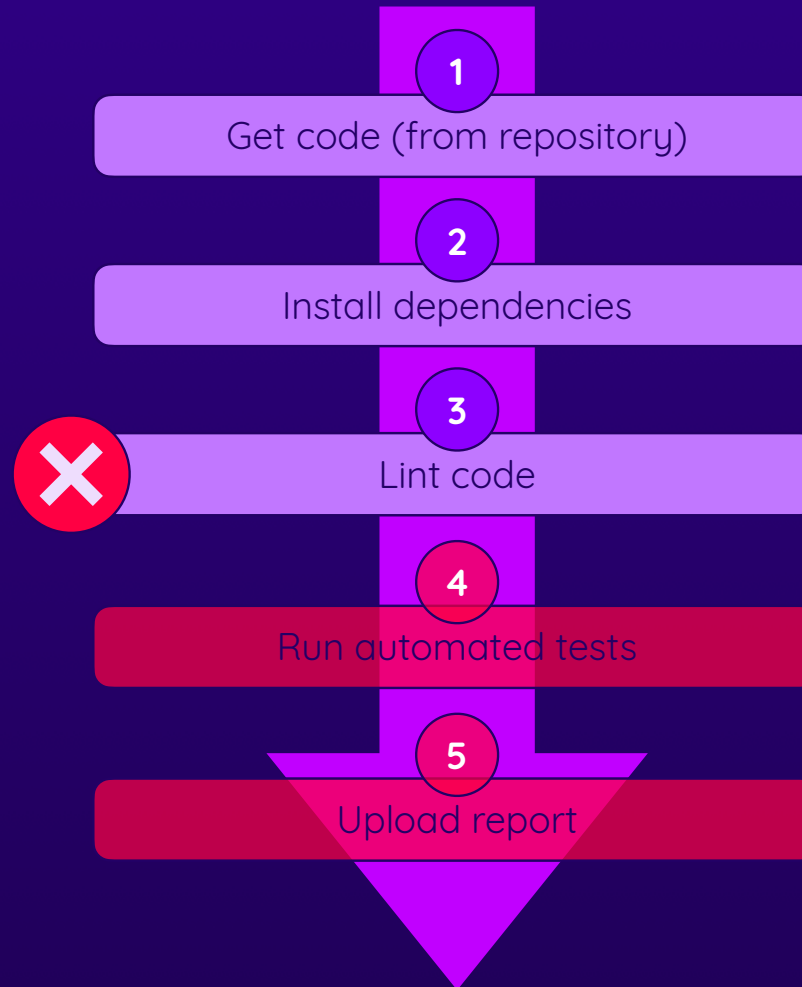
## Beyond Step-By-Step Flows

- ▶ Running Jobs & Steps Conditionally
- ▶ Running Jobs with a Matrix
- ▶ Re-Using Workflows

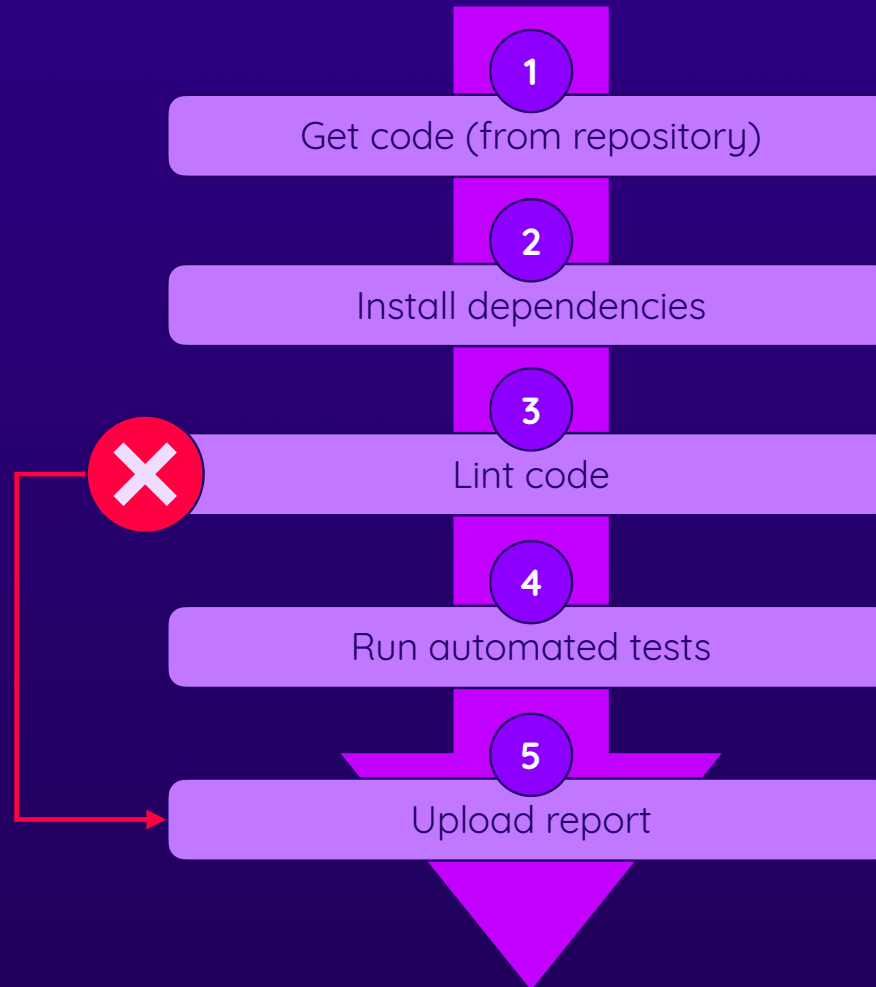
# Controlling Execution Flow



# Controlling Execution Flow



# Controlling Execution Flow



# Conditional Jobs & Steps

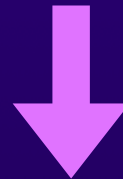
**Jobs**

Conditional execution via  
`if` field

**Steps**

Conditional execution via  
`if` field

Ignore errors via  
`continue-on-error` field



**Evaluate conditions via Expressions**



# Special Conditional Functions

**failure()**

Returns `true` when any previous Step or Job failed

**success()**

Returns `true` when none of the previous steps have failed

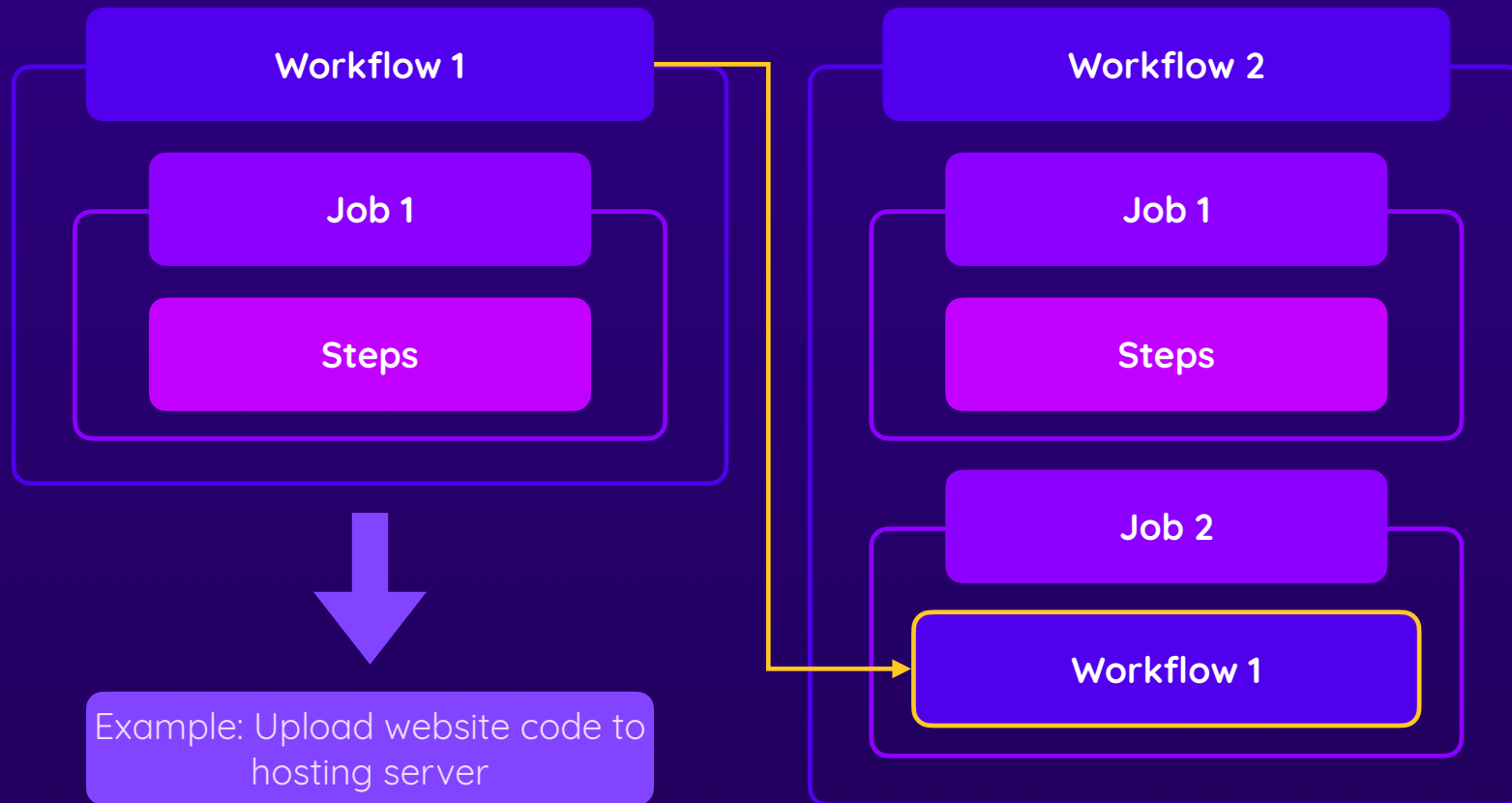
**always()**

Causes the step to always execute, even when cancelled

**cancelled()**

Returns `true` if the workflow has been cancelled

# Reusable Workflows



# Module Summary

## Conditional Jobs & Steps

Control Step or Job execution with `if` & dynamic expressions

Change default behavior with `failure()`, `success()`, `cancelled()` or `always()`

Use `continue-on-error` to ignore Step failure

## Matrix Jobs

Run multiple Job configurations in parallel

Add or remove individual combinations

Control whether a single failing Job should cancel all other Matrix Jobs via `continue-on-error`

## Reusable Workflows

Workflows can be reused via the `workflow_call` event

Reuse any logic (as many Jobs & Steps as needed)

Work with `inputs`, `outputs` and `secrets` as required

# Using Containers

## Utilizing Docker Containers

- ▶ Containers - A Re-Introduction
- ▶ Running Jobs in Containers
- ▶ Using Service Containers

# What Are Containers?



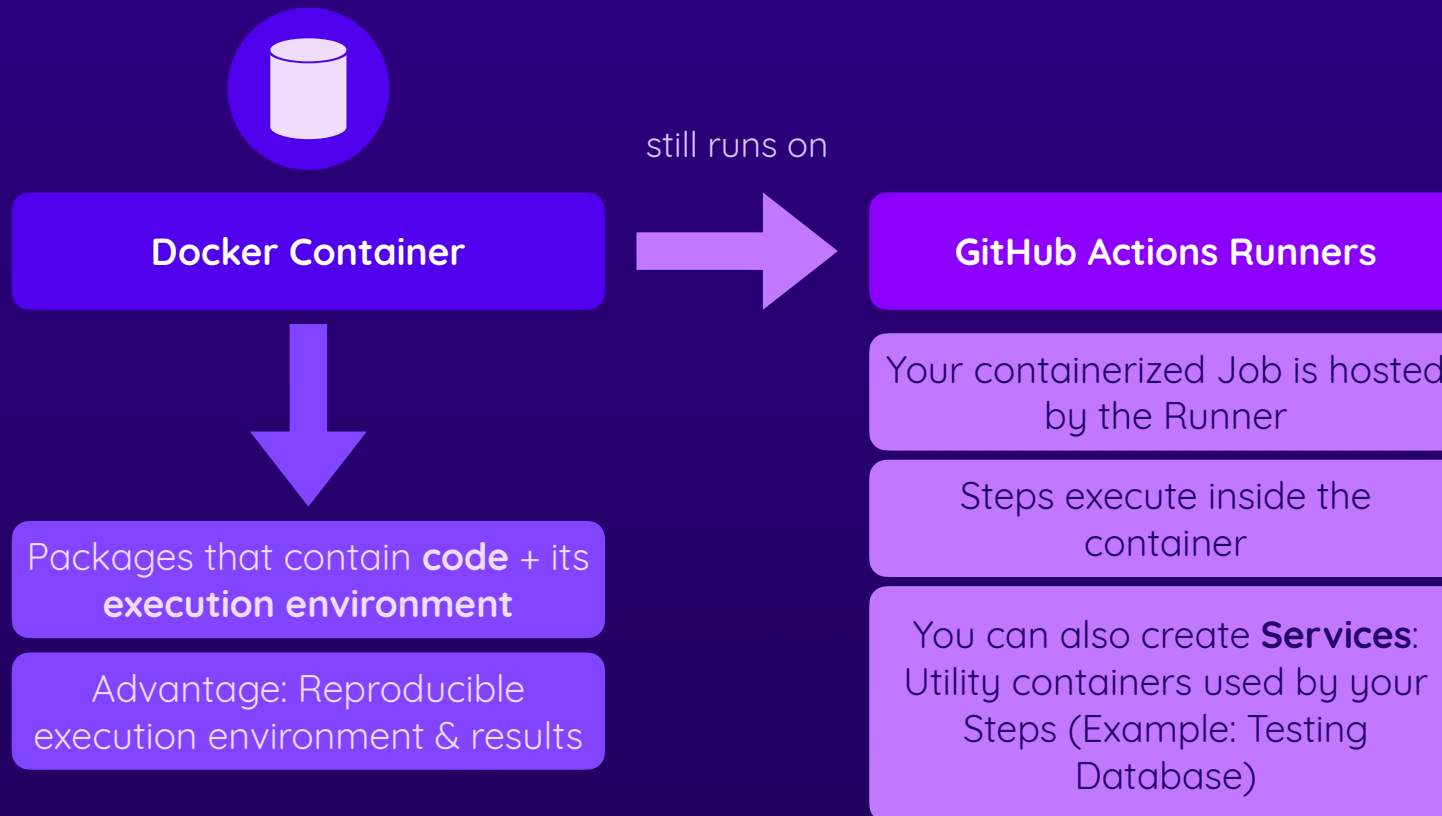
Docker Container



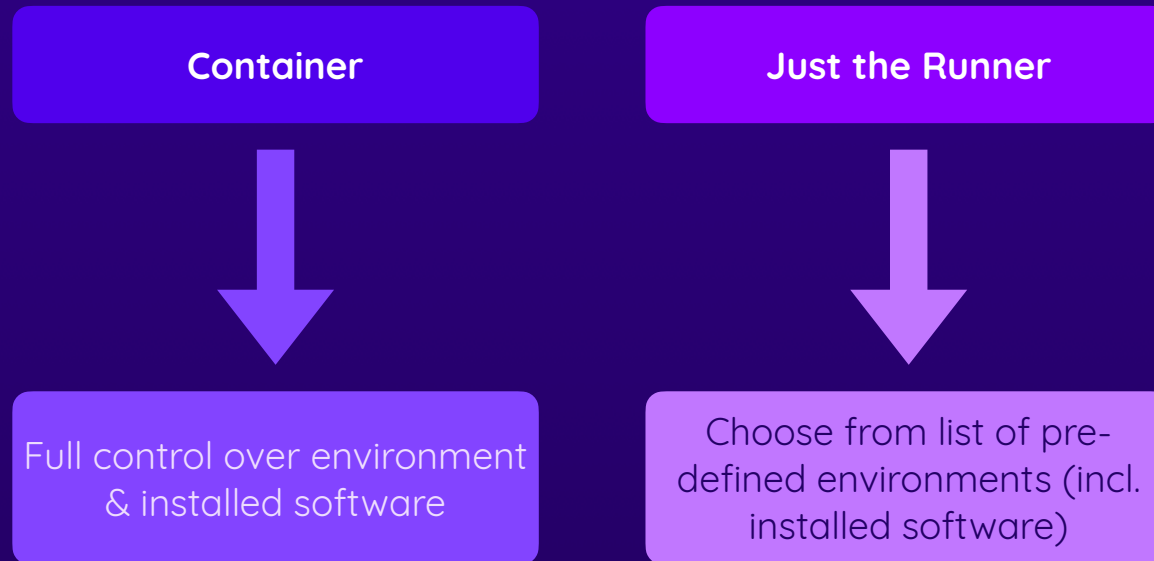
Packages that contain **code** + its  
**execution environment**

Advantage: Reproducible  
execution environment & results

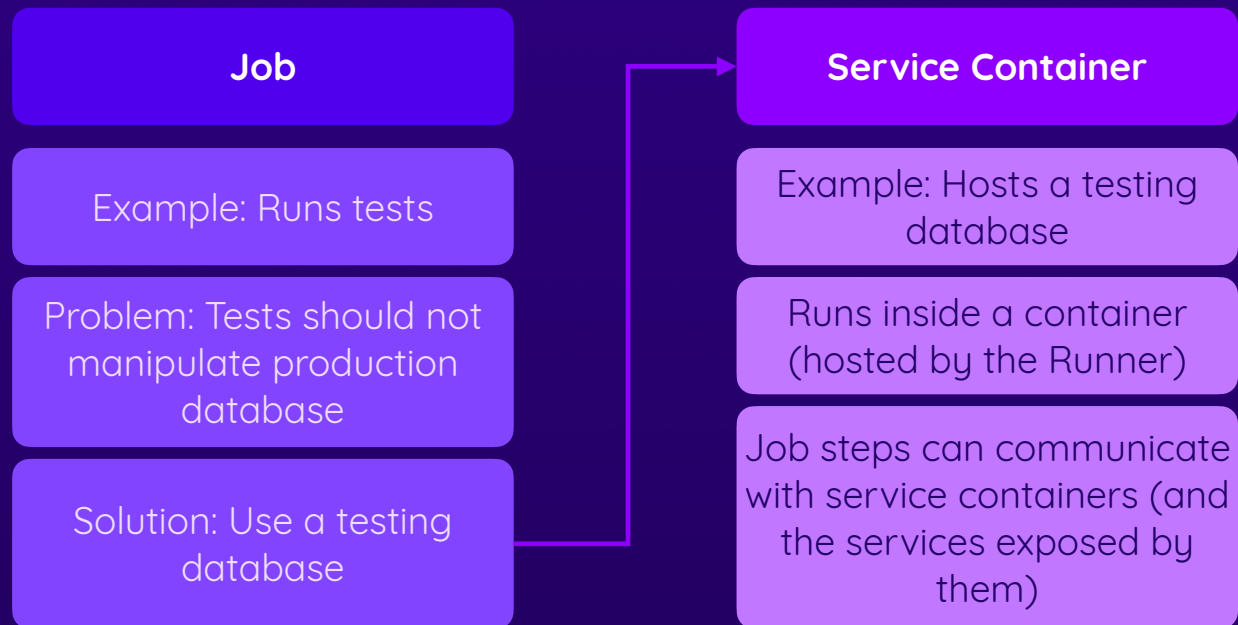
# Containers & GitHub Actions



# Why Use Containers?



# Using Service Containers (“Services”)





# Module Summary

## Containers

Packages of code + execution environment

Great for creating re-usable execution packages & ensuring consistency

Example: Same environment for testing + production

## Containers for Jobs

You can run Jobs in pre-defined environments

Build your own container images or use public images

Great for Jobs that need extra tools or lots of customization

## Service Containers

Extra services can be used by Steps in Jobs

Example: Locally running, isolated testing database

Based on custom images or public / community images

# Building Custom Actions

Beyond Shell Commands & The Marketplace

- ▶ What & Why?
- ▶ Different Types of Custom Actions
- ▶ Building & Using Custom Actions

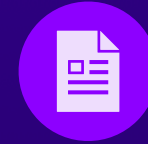
# Why Custom Actions?



## Simplify Workflow Steps

Instead of writing multiple (possibly very complex) Step definitions, you can build and use a single custom Action

Multiple Steps can be grouped into a single custom Action



## No Existing (Community) Action

Existing, public Actions might not solve the specific problem you have in your Workflow

Custom Actions can contain any logic you need to solve your specific Workflow problems

# Different Types of Custom Actions



## JavaScript Actions

Execute a JavaScript file

Use JavaScript (NodeJS) + any packages of your choice

Pretty straightforward (if you know JavaScript)



## Docker Actions

Create a Dockerfile with your required configuration

Perform any task(s) of your choice with any language

Lots of flexibility but requires Docker knowledge



## Composite Actions

Combine multiple Workflow Steps in one single Action

Combine run (commands) and uses (Actions)

Allows for reusing shared Steps (without extra skills)

# Module Summary

## What & Why?

Simplify Workflows & avoid repeated Steps

Implement logic that solves a problem not solved by any publicly available Action

Create & share Actions with the Community

## Composite Actions

Create custom Actions by combining multiple Steps

Composite Actions are like “Workflow Excerpts”

Use Actions (via `uses`) and Commands (via `run`) as needed

## JavaScript & Docker Actions

Write Action logic in JavaScript (NodeJS) with `@actions/toolkit`

Alternatively: Create your own Action environment with Docker

Either way: Use inputs, set outputs and perform any logic

# Permissions & Security

Keep Things Secure

- ▶ Securing Your Workflows
- ▶ Working with GitHub Tokens & Permissions
- ▶ Third-Party Permissions

# Security Concerns



## Script Injection

A value, set outside a Workflow, is used in a Workflow

Example: Issue title used in a Workflow shell command

Workflow / command behavior could be changed



## Malicious Third-Party Actions

Actions can perform any logic, including potentially malicious logic

Example: A third-party Action that reads and exports your secrets

Only use trusted Actions and inspect code of unknown / untrusted authors



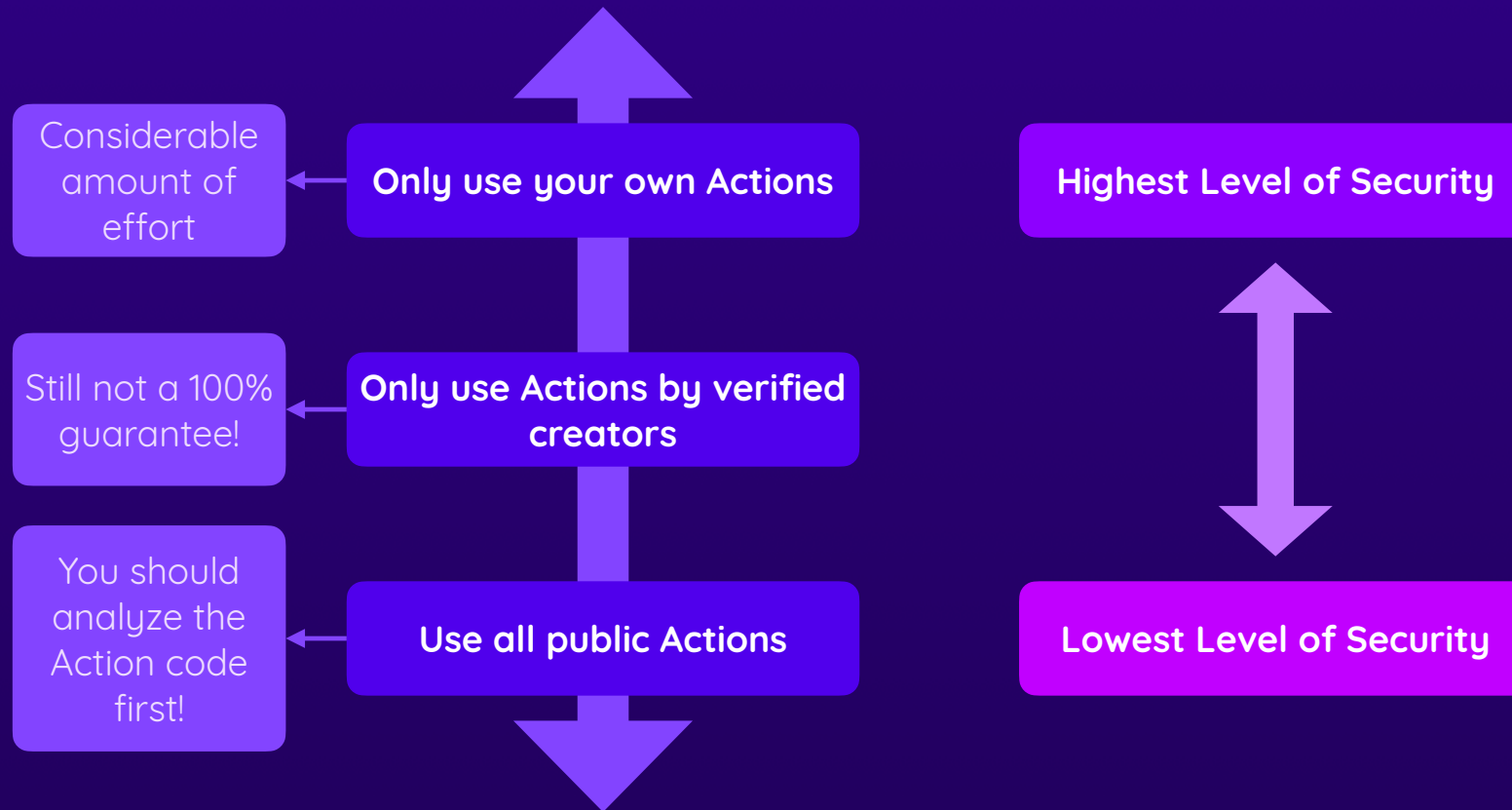
## Permission Issues

Consider avoiding overly permissive permissions

Example: Only allow checking out code ("read-only")

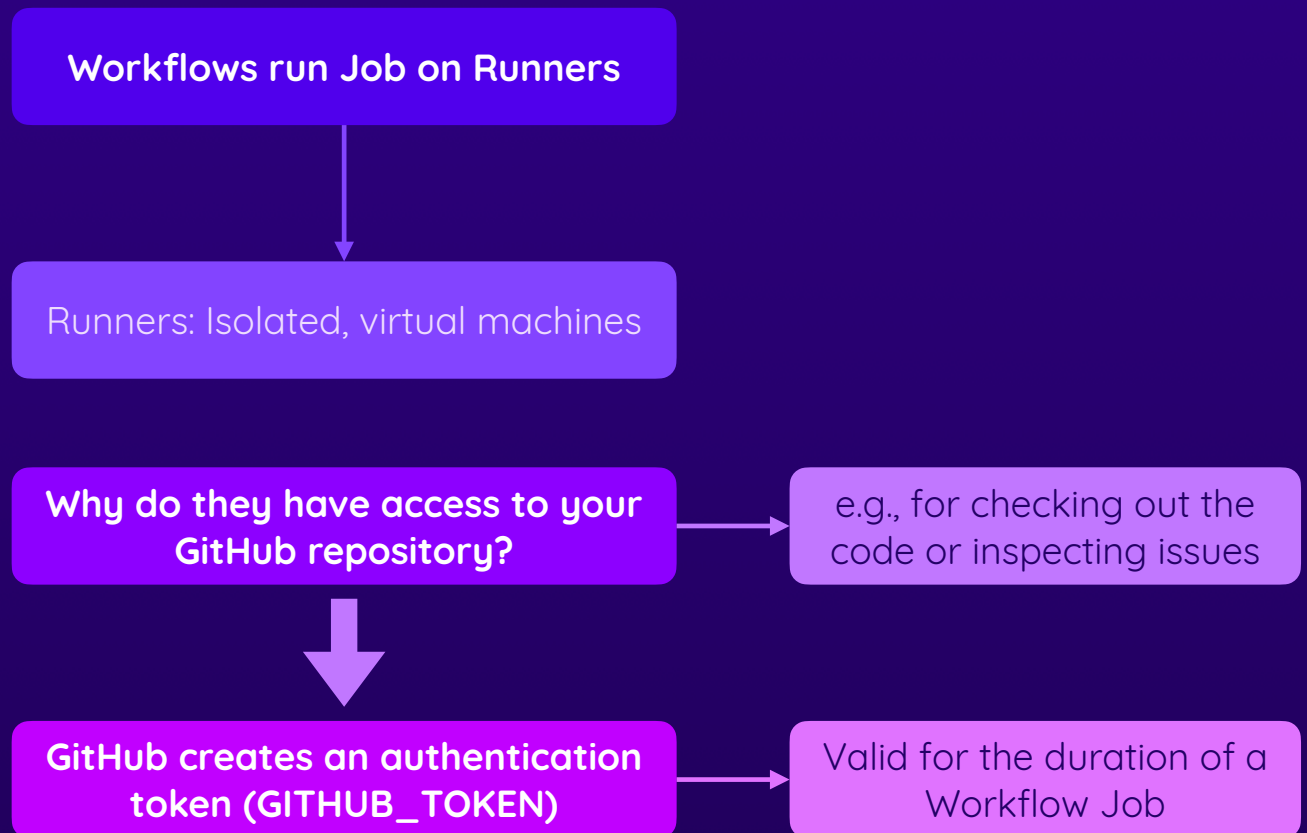
GitHub Actions supports fine-grained permissions control

# Using Actions Securely





# GitHub Permissions & GITHUB\_TOKEN



# Third-Party Permissions

Especially for deployment tasks, GitHub Workflows regularly communicate with third-party cloud / hosting providers



**Credentials / Authentication required**

**Option 1: API or Access Keys**

via secrets

**Option 2: Open ID Connect**

Workflow assumes a temporary, provider-managed role

# A Complete, Realistic Example

## Applying What You Learned

- ▶ Build Two Complete, Realistic Example Workflows
- ▶ Workflow 1: Complete Deployment Workflow A - Z
- ▶ Workflow 2: Manage Repository Issues

# An Example Workflow

## Demo Website

### Goal

Deploy (“publish”) the website

### Trigger

Whenever a new commit is pushed to the “main” branch

