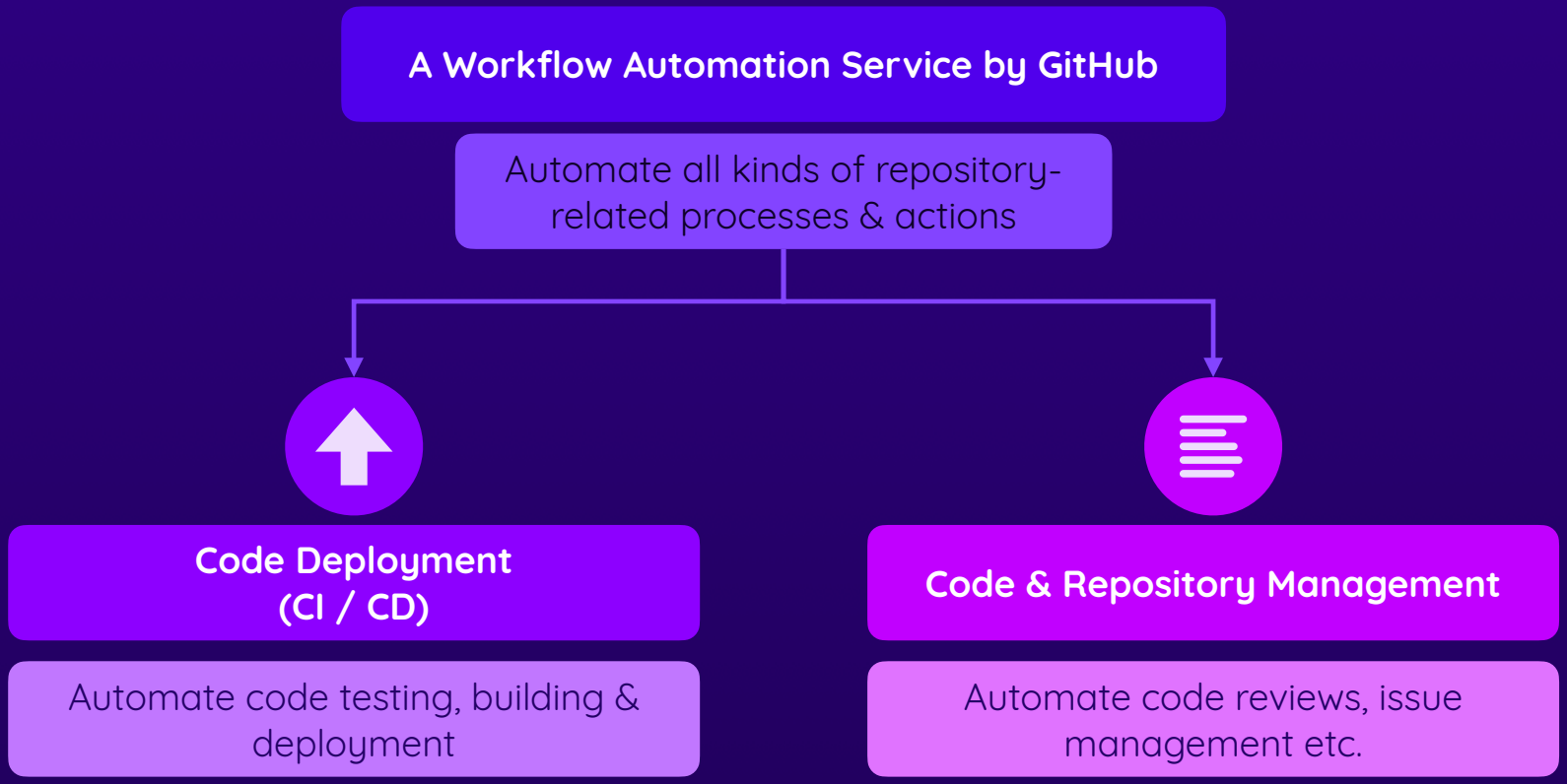
What Is “GitHub Actions”?

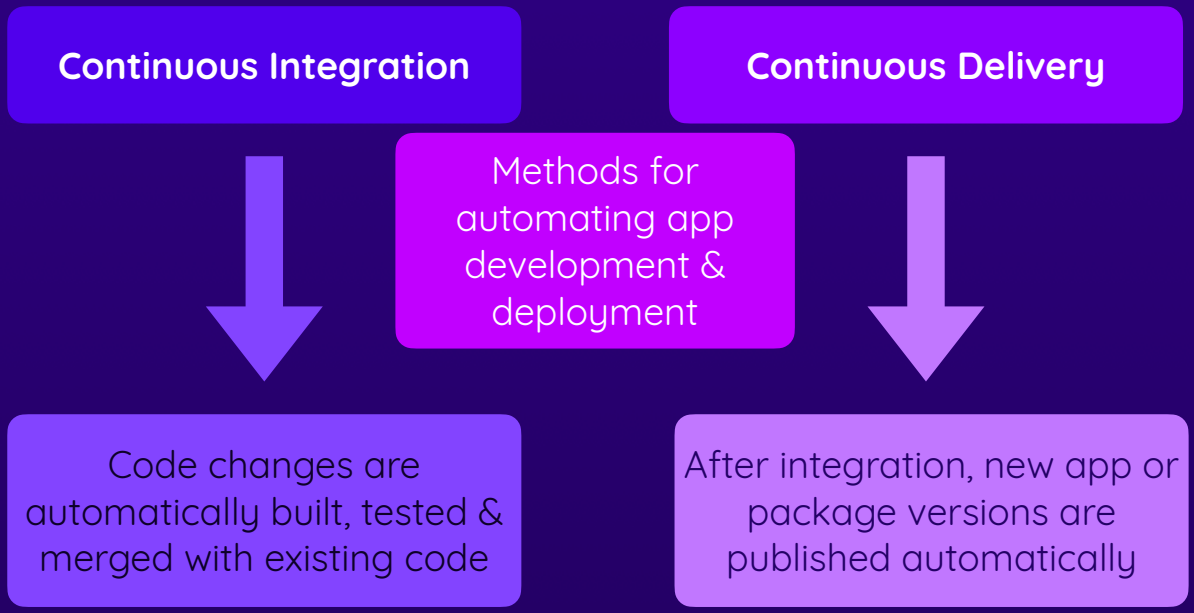


ACADEMIND

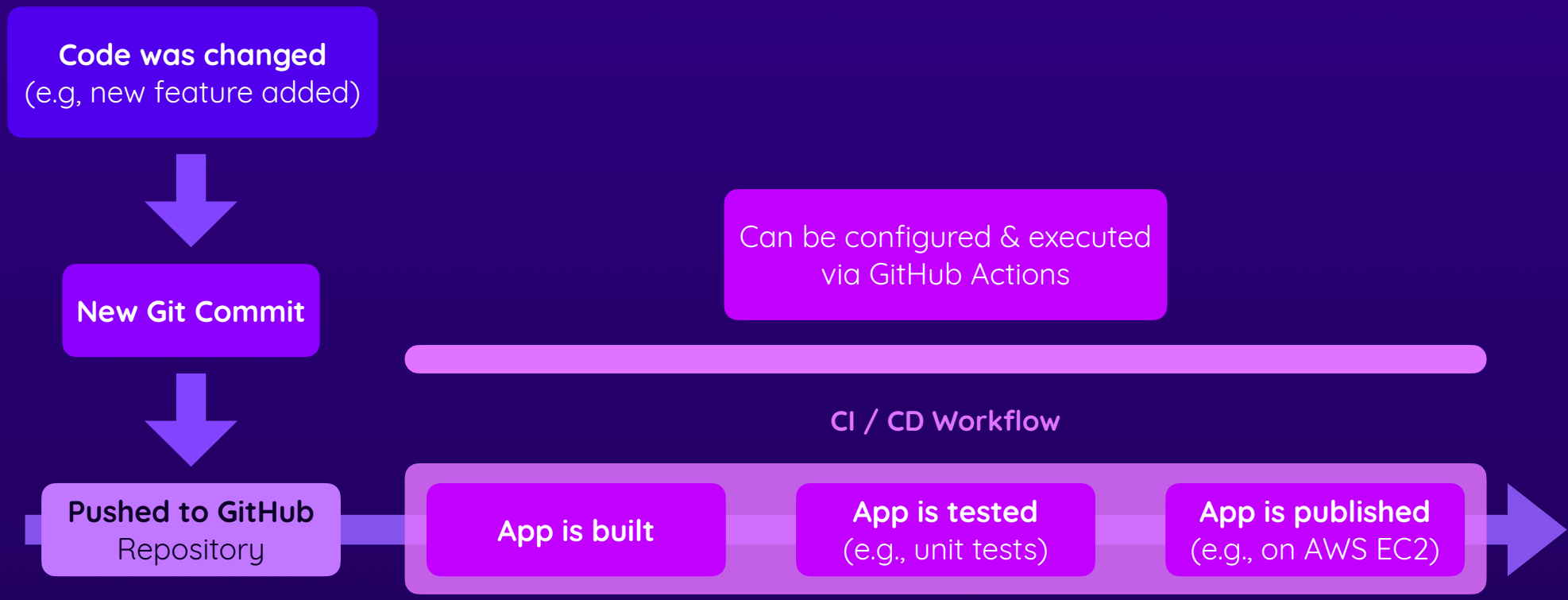
What is GitHub?

And what are “repositories”?

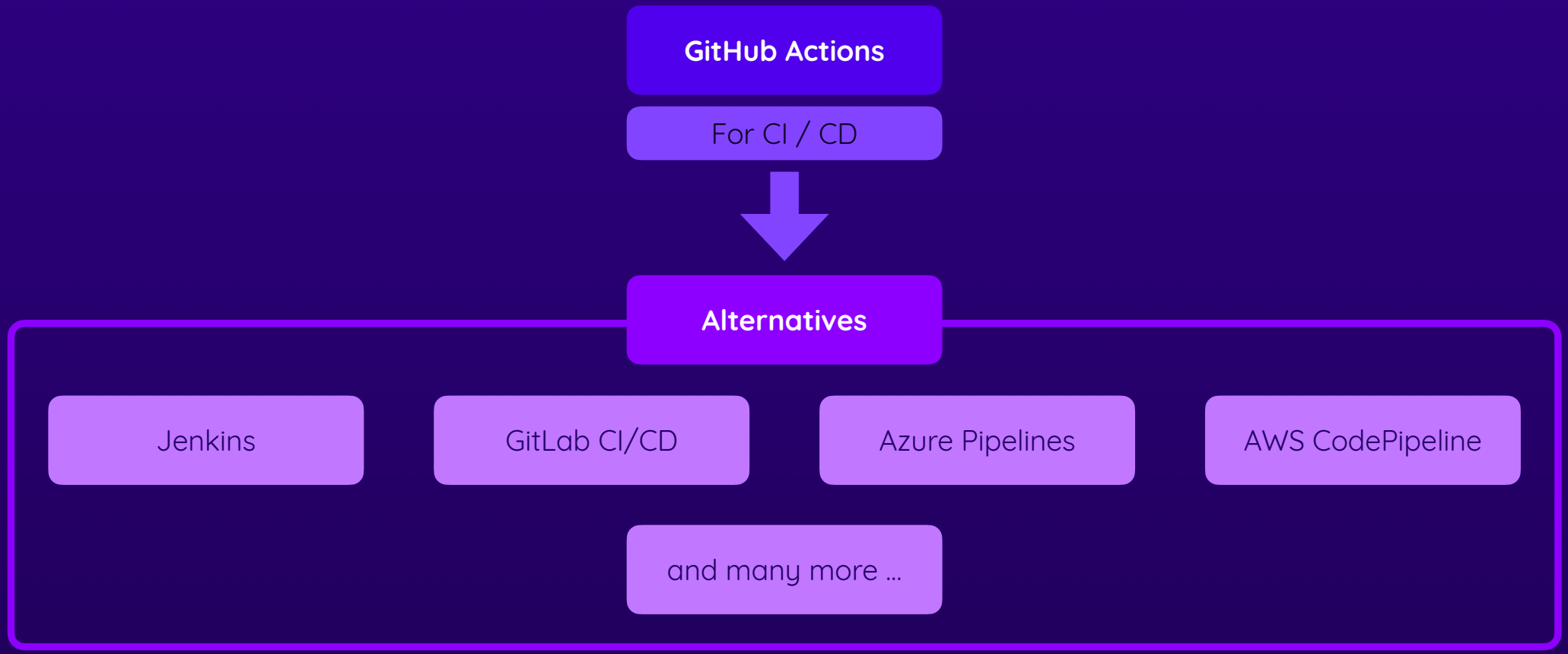
What's CI / CD?


Continuous Integration
Continuous Delivery
Methods for automating app development & deployment
Code changes are automatically built, tested & merged with existing code
After integration, new app or package versions are published automatically

A Typical CI / CD Workflow



GitHub Actions Alternatives



ACADEMIND

What is Git?

What Is GitHub?

A cloud Git repository & services provider

Store & manage Git repositories

ACADEMIND

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

ACADEMIND

Git & GitHub Crash Course

The Very Basics

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

ACADEMIND

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

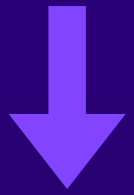
Move between Commits

`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>`

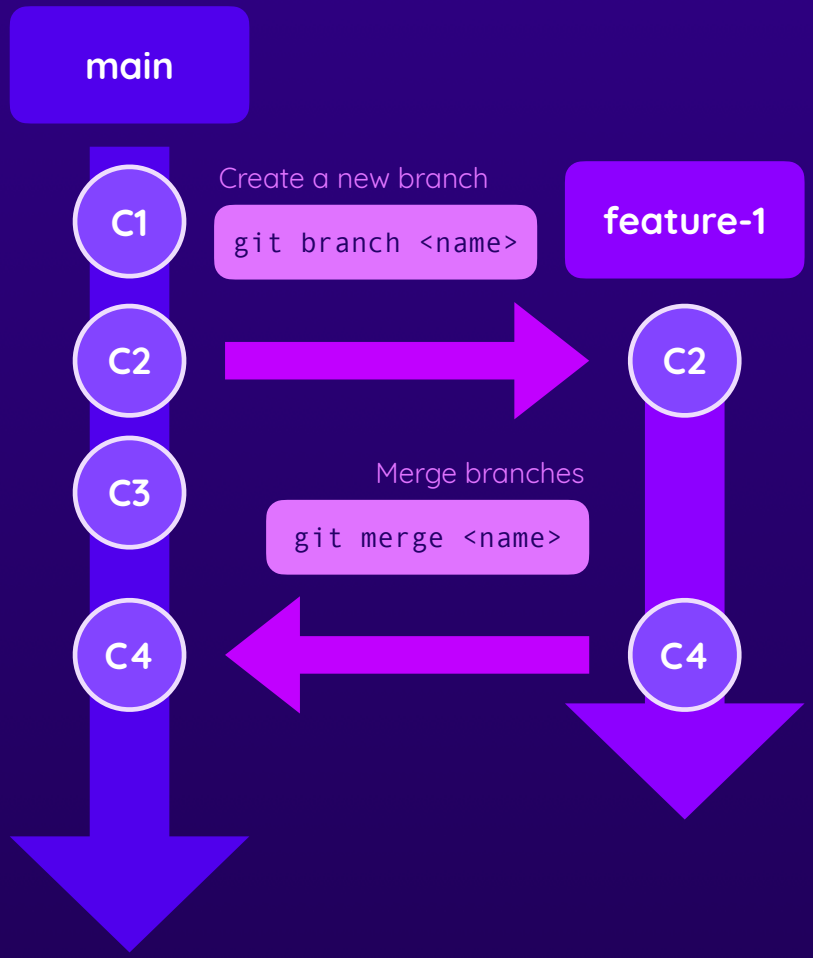
Undo changes by deleting all commits since <id>

ACADEMIND

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

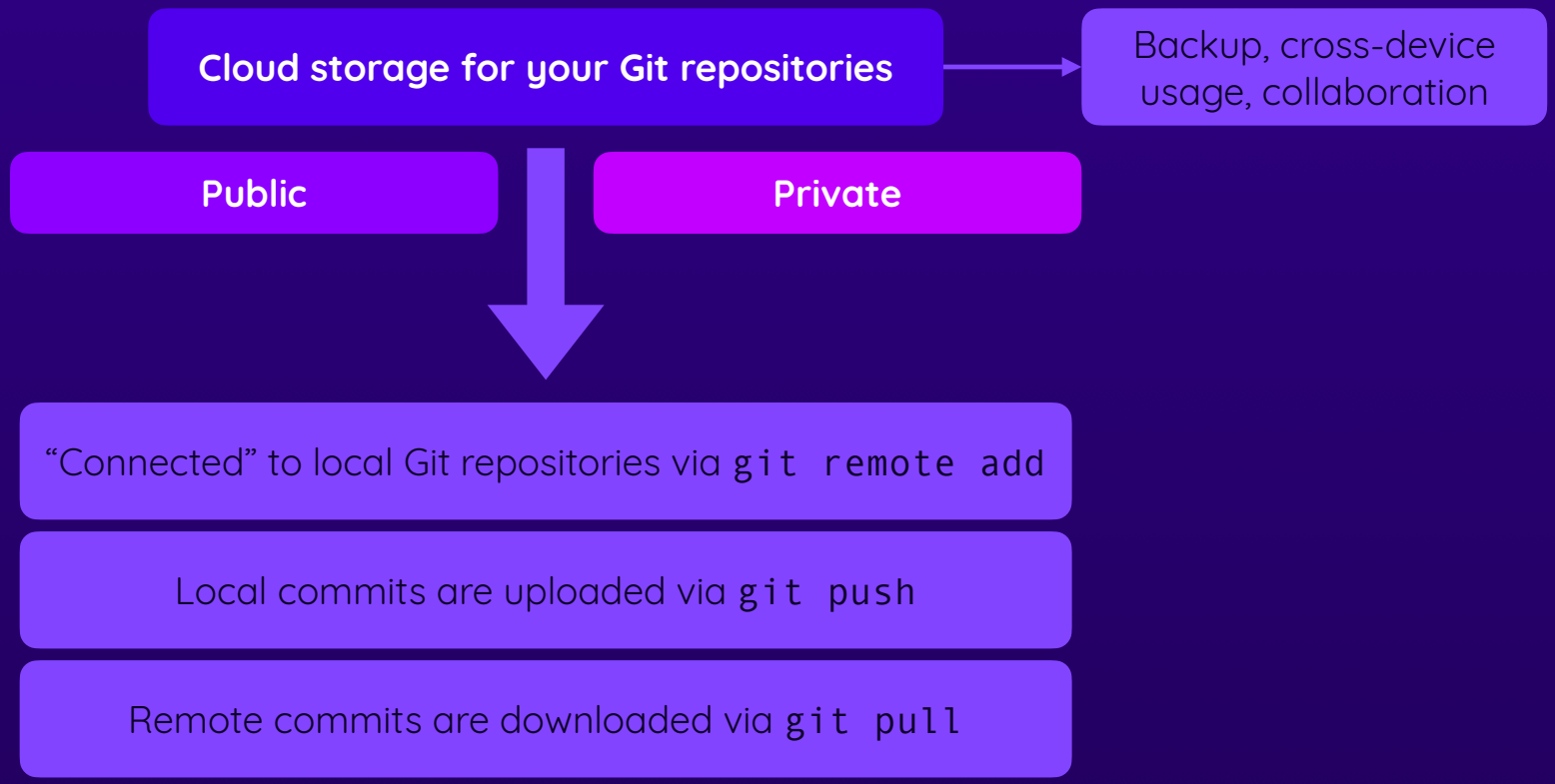
Code management & collaborative development

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

ACADEMIND

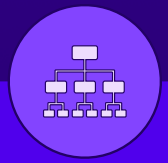
GitHub Actions: Fundamentals

Key Building Blocks & Usage

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

ACADEMIND

Key Elements



Workflows

- Attached to a GitHub repository
- Contain one or more **Jobs**
- Triggered upon **Events**

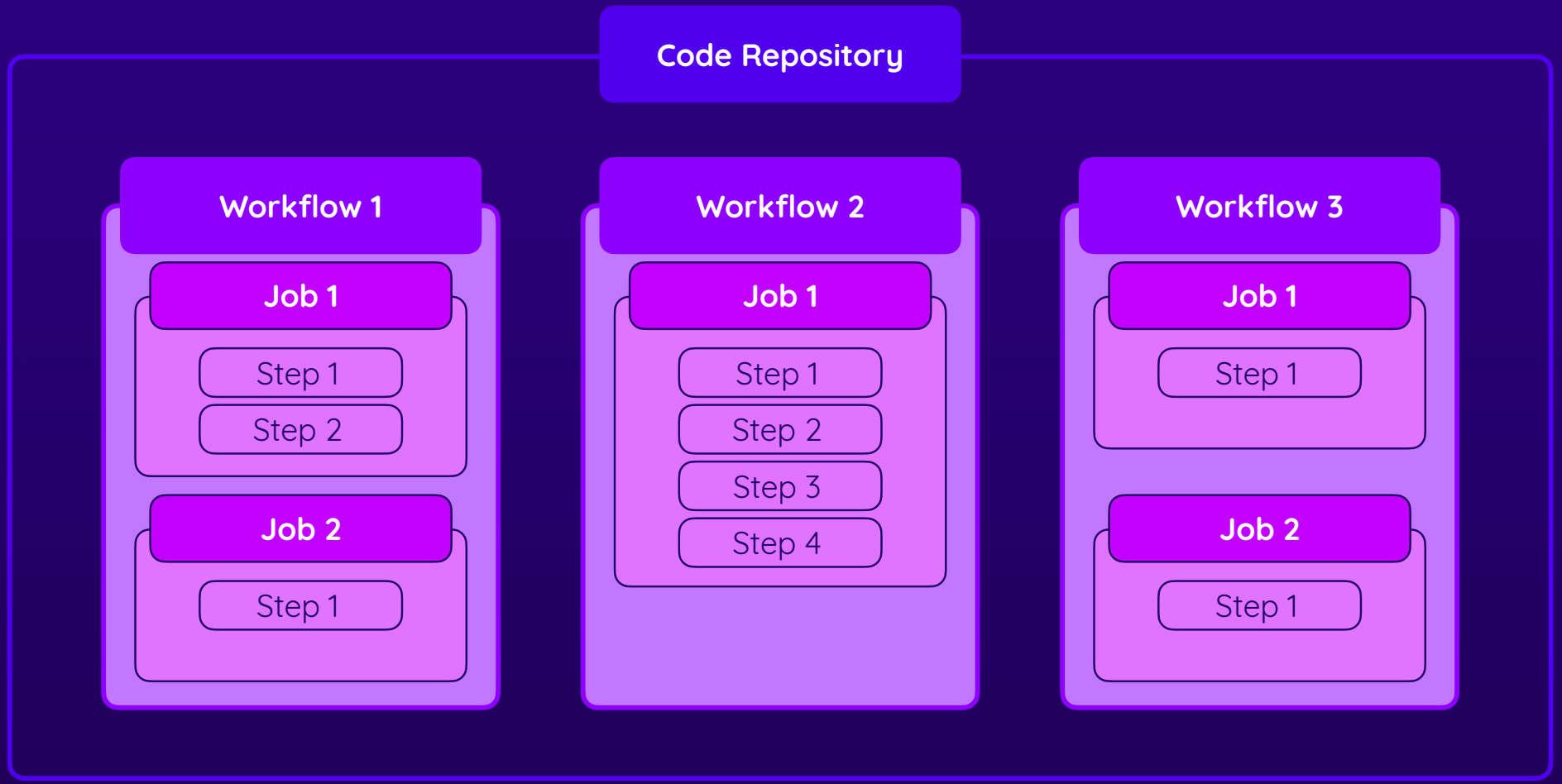
Jobs

- 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

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

What Are Actions?

Command ("run")

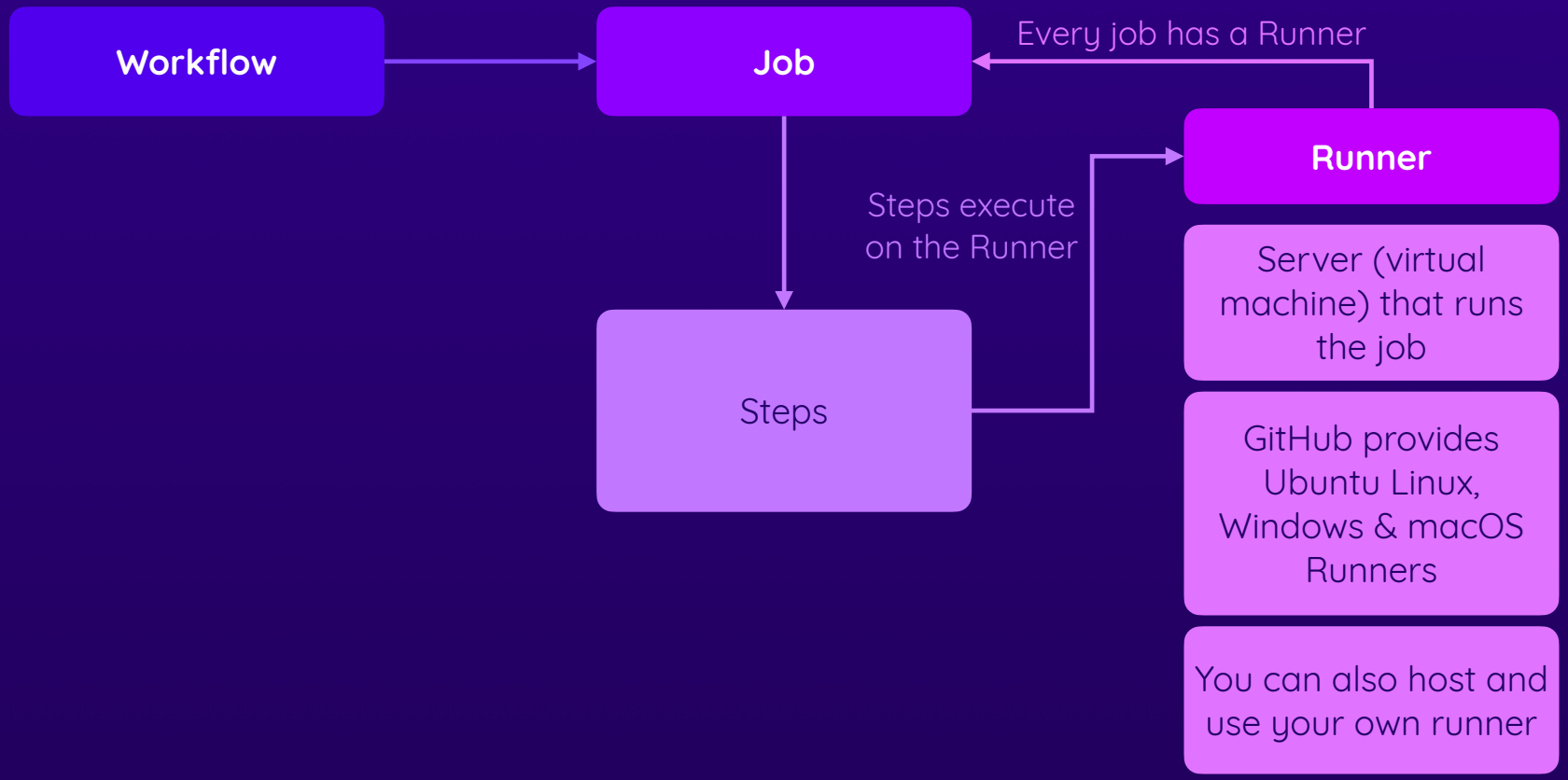
A (typically simple) shell command that's defined by you

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


Workflow
Job
Every job has a Runner
Runner
Steps execute on the Runner
Steps
Server (virtual machine) that runs the job
GitHub provides Ubuntu Linux, Windows & macOS Runners
You can also host and use your own runner

Module Summary

Core Components

- Workflows: Define Events + Jobs
- Jobs: Define Runner + Steps
- Steps: Do the actual work

Defining Workflows

- .github/workflows/<file>.yml (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

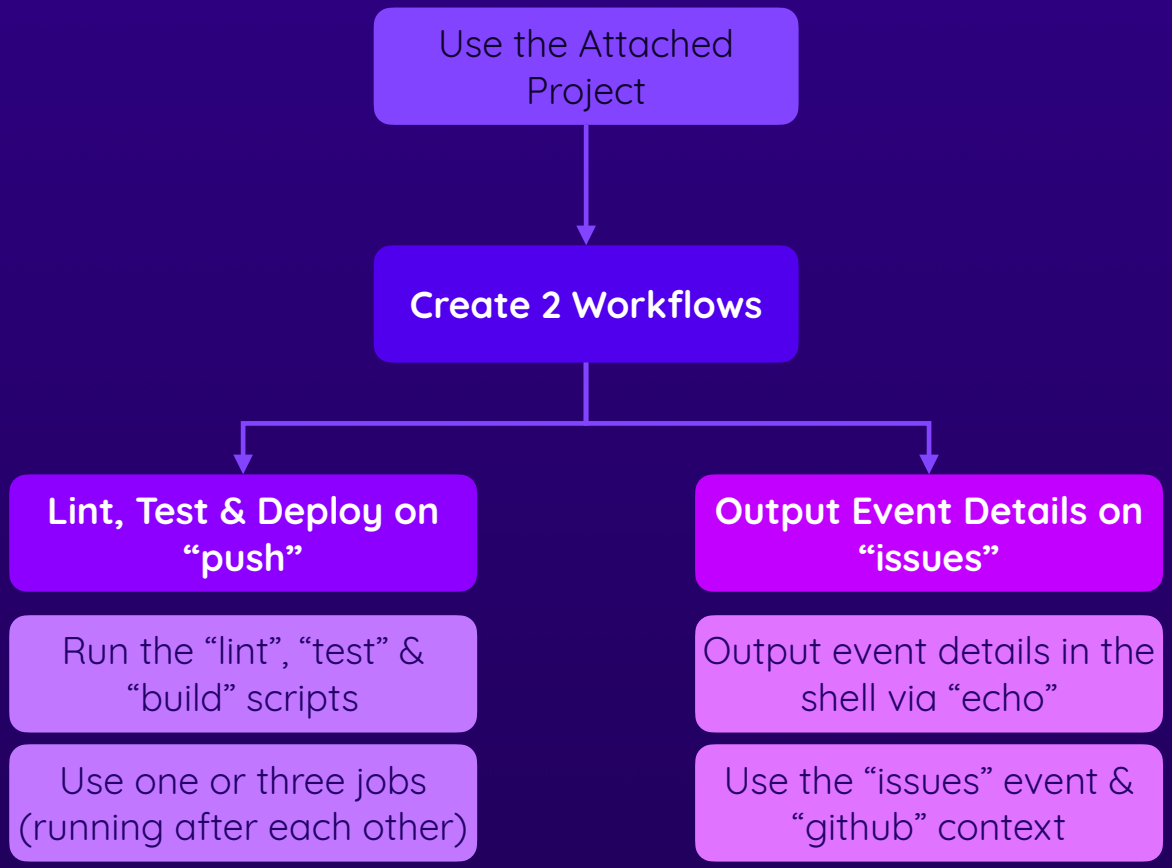
Workflow Execution

- 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!



ACADEMIND

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

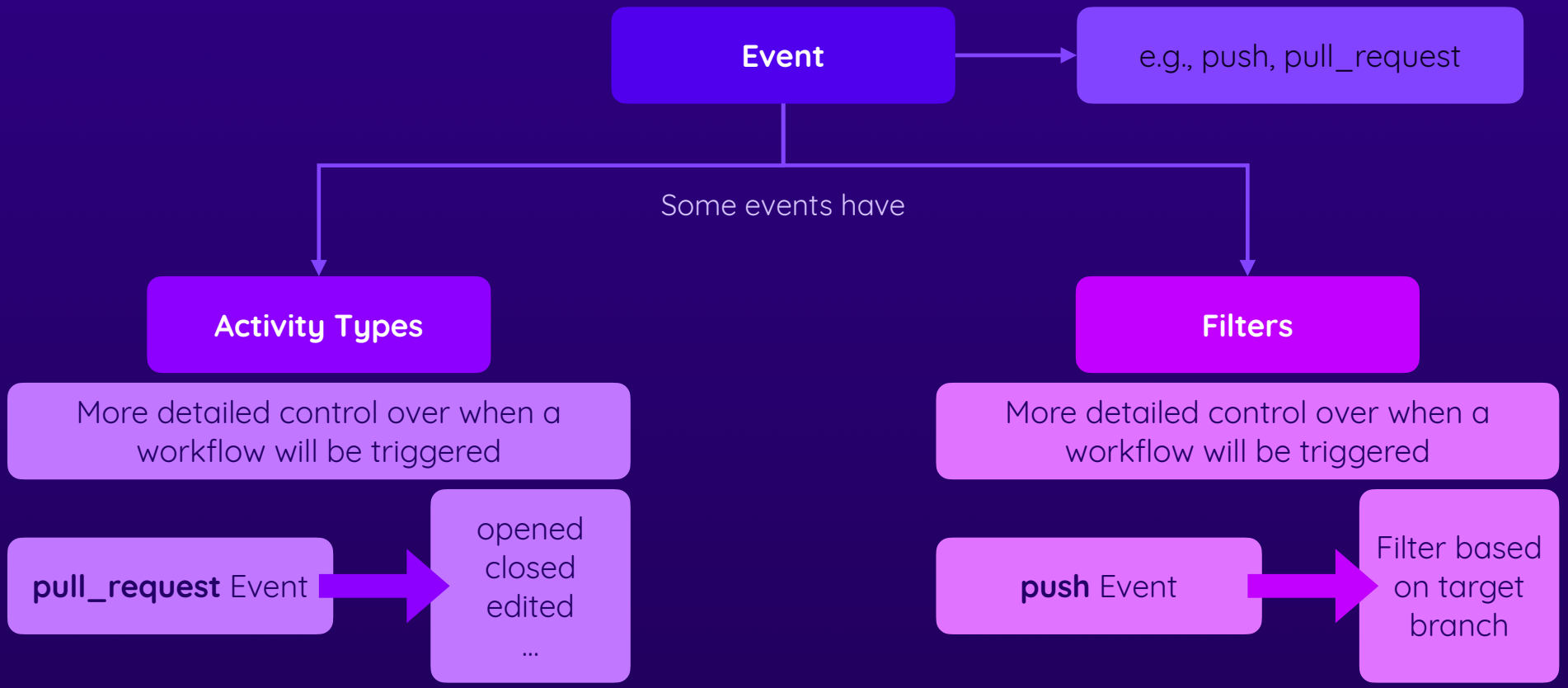
workflow_dispatch
Manually trigger workflow

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

ACADEMIND

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

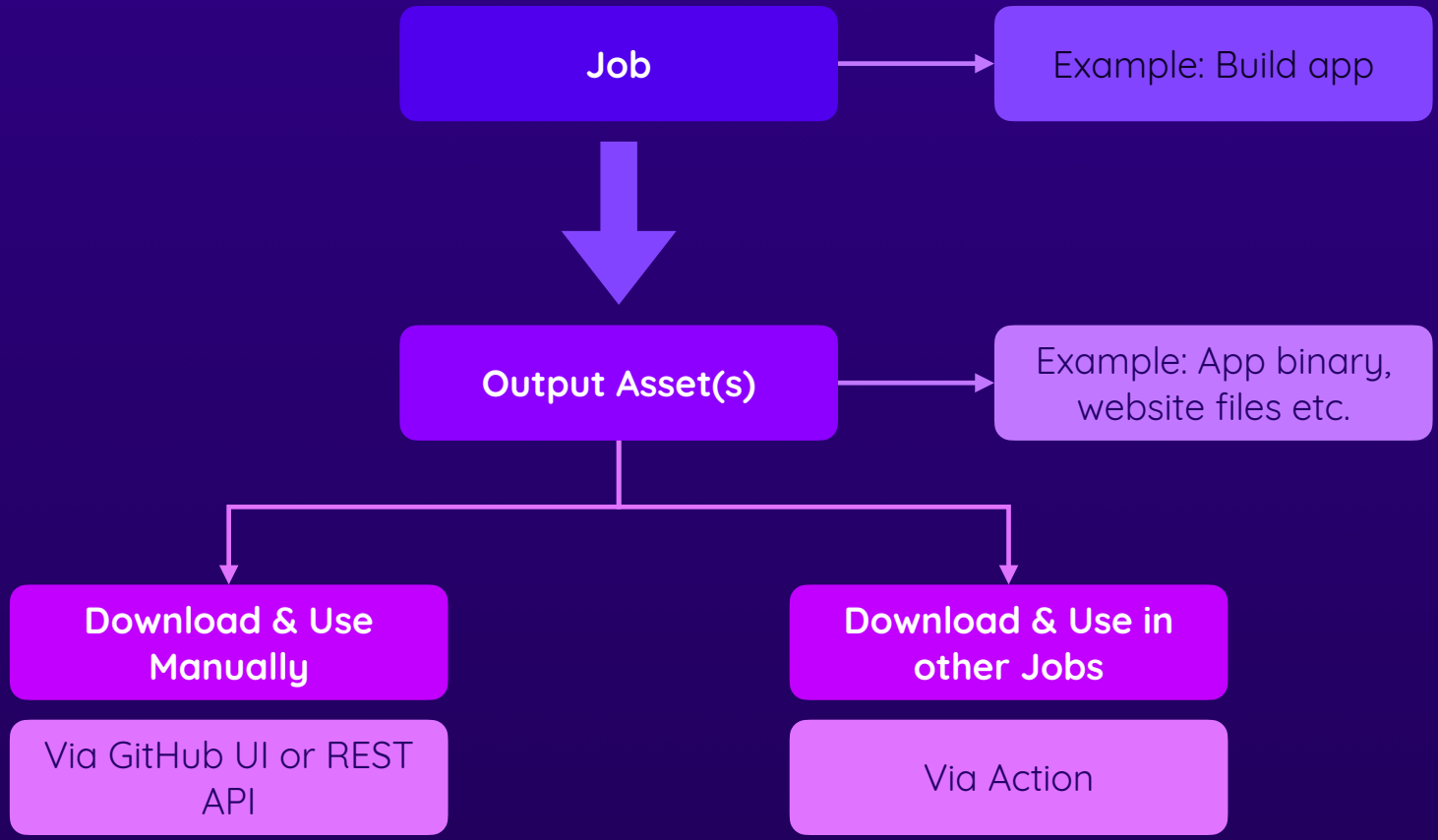
ACADEMIND

Job Data & Outputs

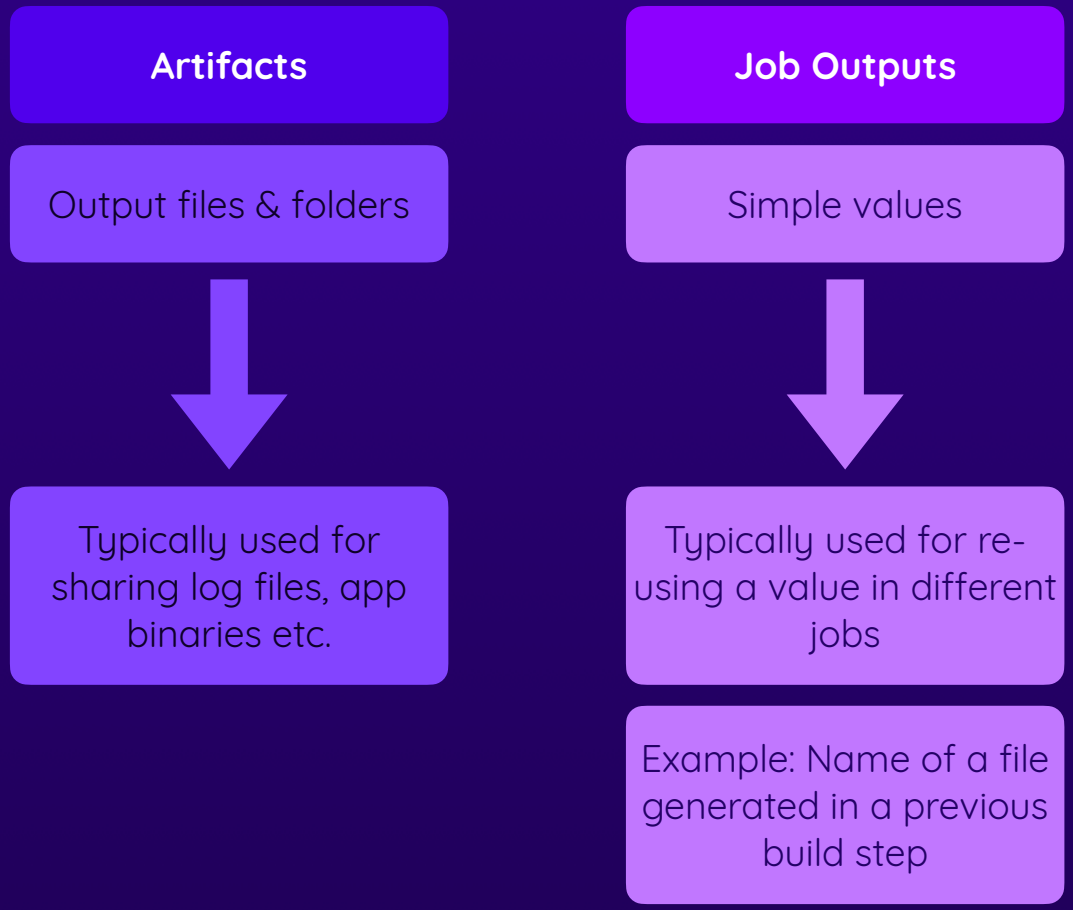
It's All About Data!

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

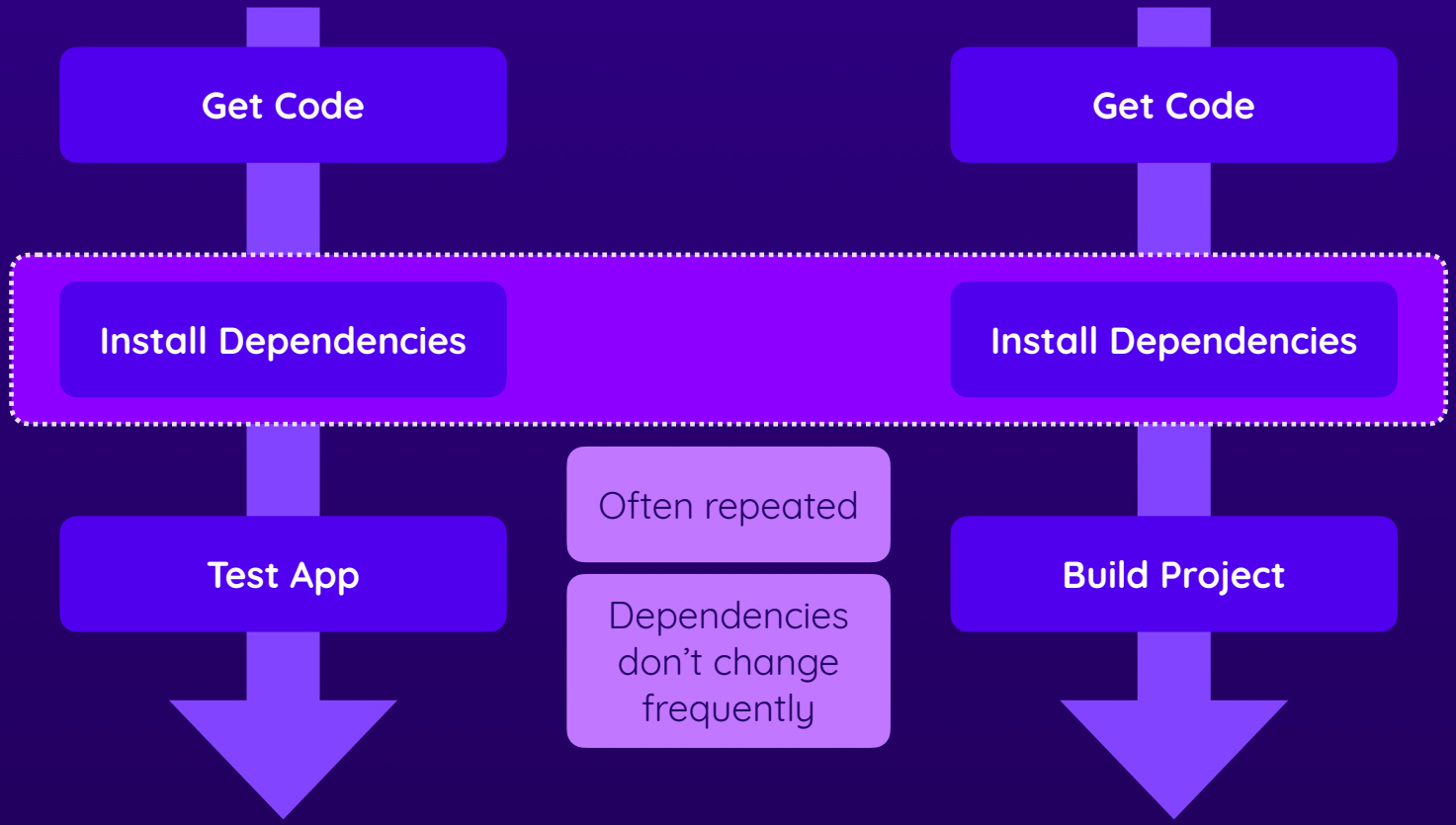
Understanding Job Artifacts



Understanding Job Outputs



Caching Dependencies



Module Summary

Artifacts	Outputs	Caching
Jobs often product assets that should be shared or analyzed	Besides Artifacts, Steps can product and share simple values	Caching can help speed up repeated, slow Steps
Examples: Deployable website files, logs, binaries etc.	These outputs are shared via `::set-output`	Typical use-case: Caching dependencies
These assets are referred to as “Artifacts” (or “Job Artifacts”)	Jobs can pick up & share Step outputs via the `steps` context	But any files & folder can be cached
GitHub Actions provides Actions for uploading & downloading	Other Jobs can use Job outputs via the `needs` context	The cache Action automatically stores & updates cache values (based on the cache key)
		Important: Don’t use caching for artifacts!

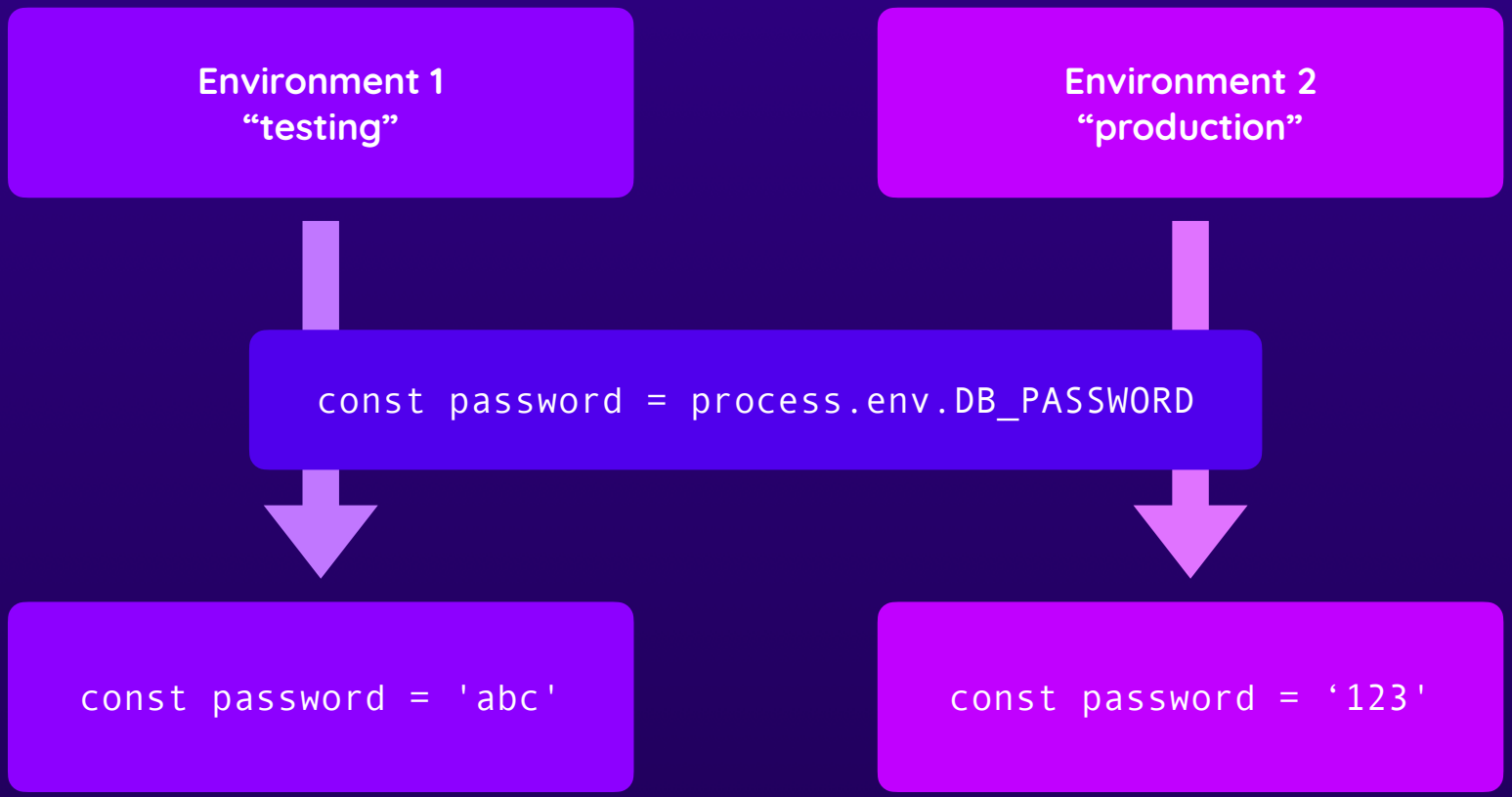
ACADEMIND

Environment Variables & Secrets

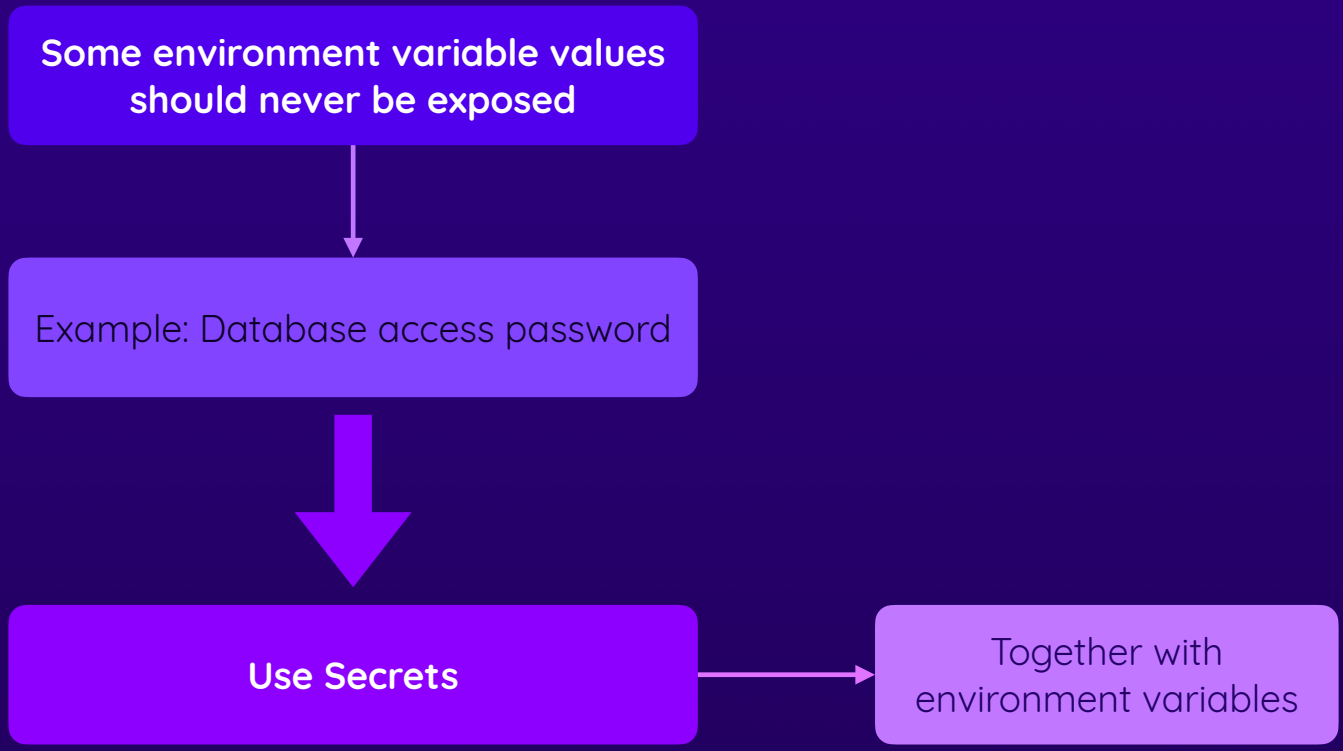
Hardcoding Is Not (Often) The Solution

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

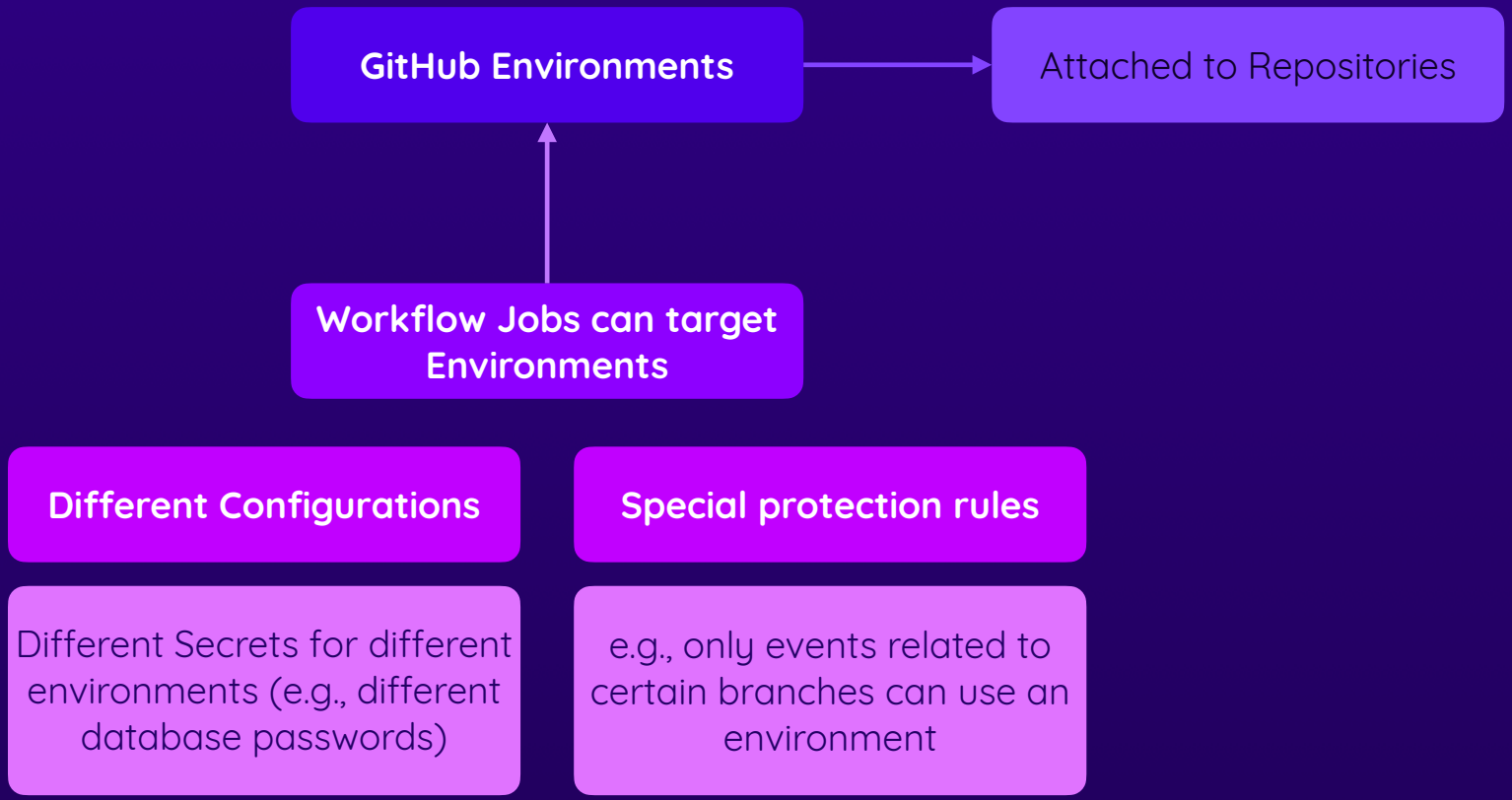
Understanding Environment Variables



Environment Variables vs Secrets



GitHub Repository Environments



Module Summary

Environment Variables	Secrets	GitHub Actions Environments
Dynamic values used in code (e.g., database name)	Some dynamic values should not be exposed anywhere	Jobs can reference different GitHub Actions Environments
May differ from workflow to workflow	Examples: Database credentials, API keys etc.	Environments allow you to set up extra protection rules
Can be defined on Workflow-, Job- or Step-level	Secrets can be stored on Repository-level or via Environments	You can also store Secrets on Environment-level
Can be used in code and in the GitHub Actions Workflow	Secrets can be referenced via the `secrets` context object	
Accessible via interpolation and the `env` context object		

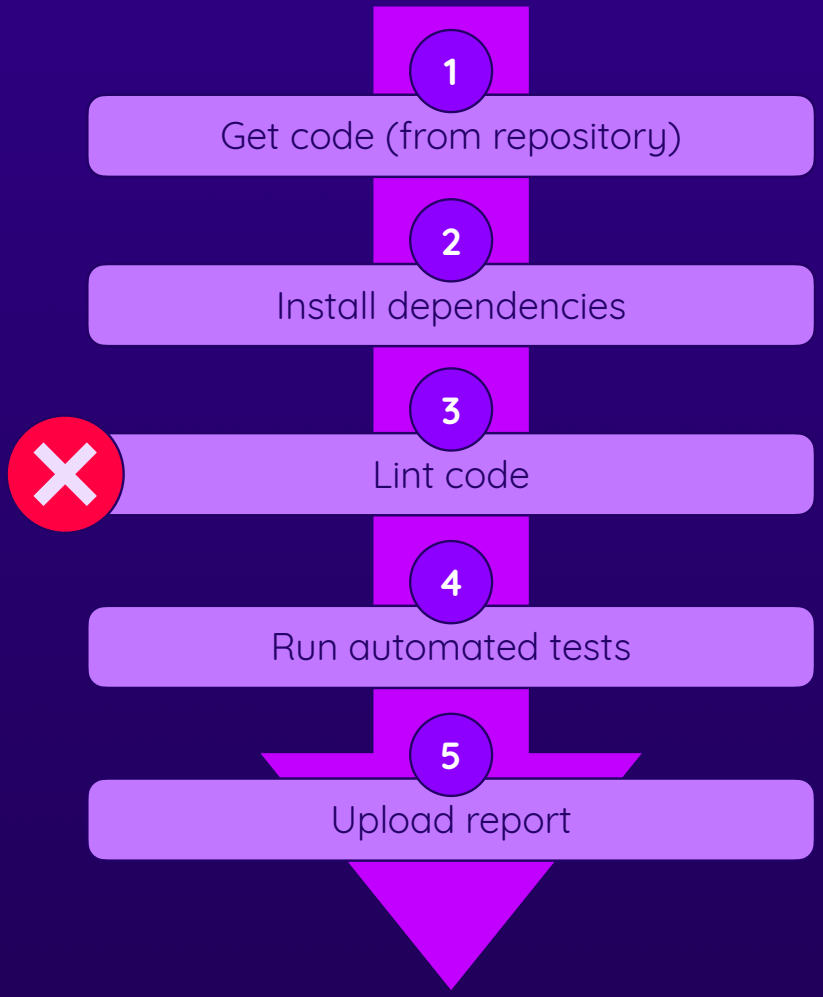
ACADEMIND

Controlling Execution Flow

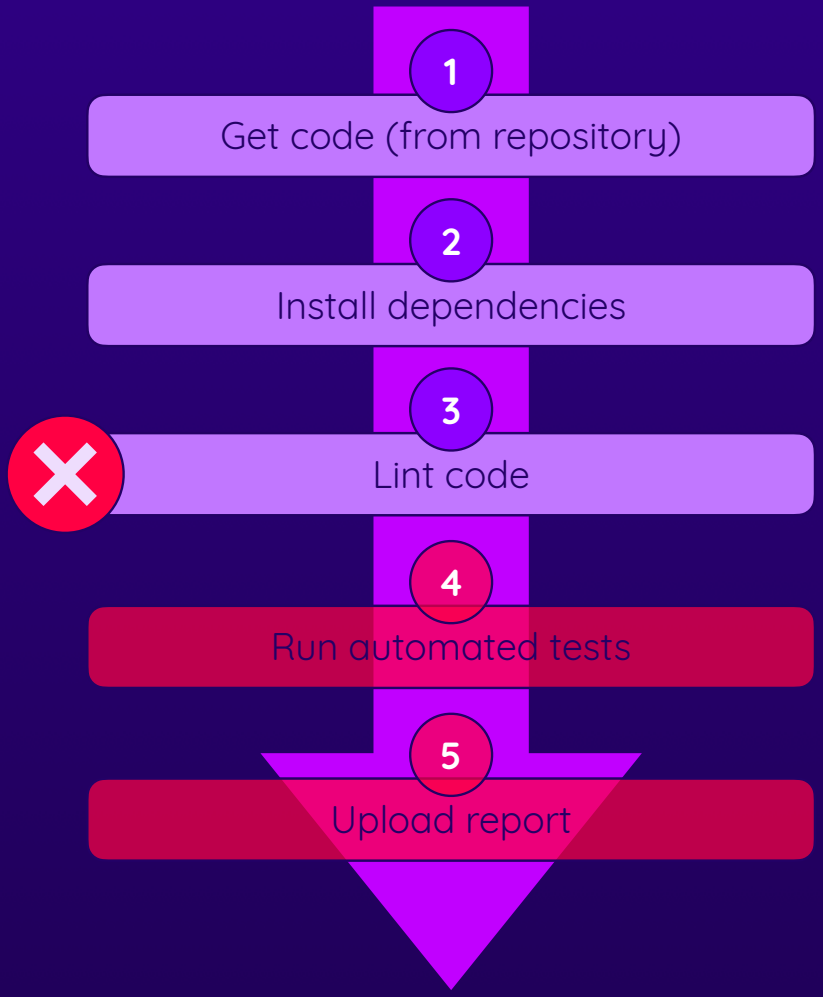
Beyond Step-By-Step Flows

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

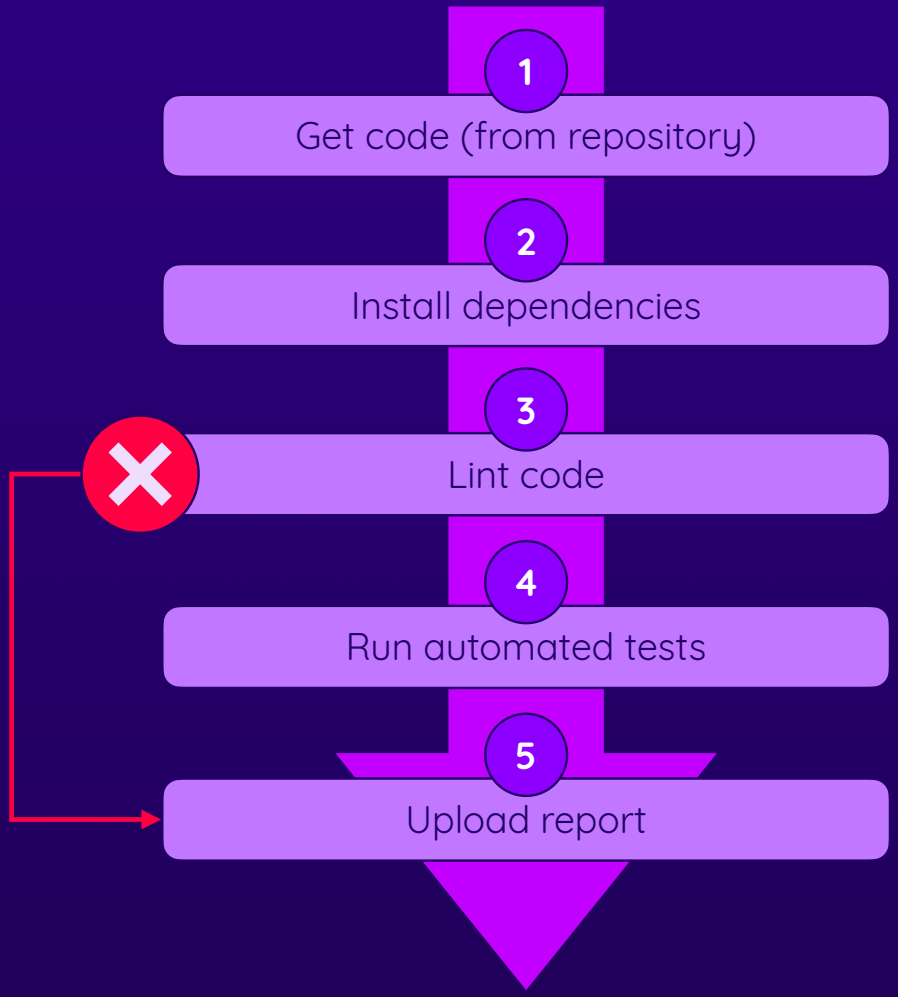
Controlling Execution Flow



Controlling Execution Flow



Controlling Execution Flow



Conditional Jobs & Steps

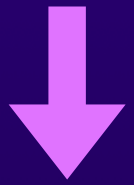
Jobs

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

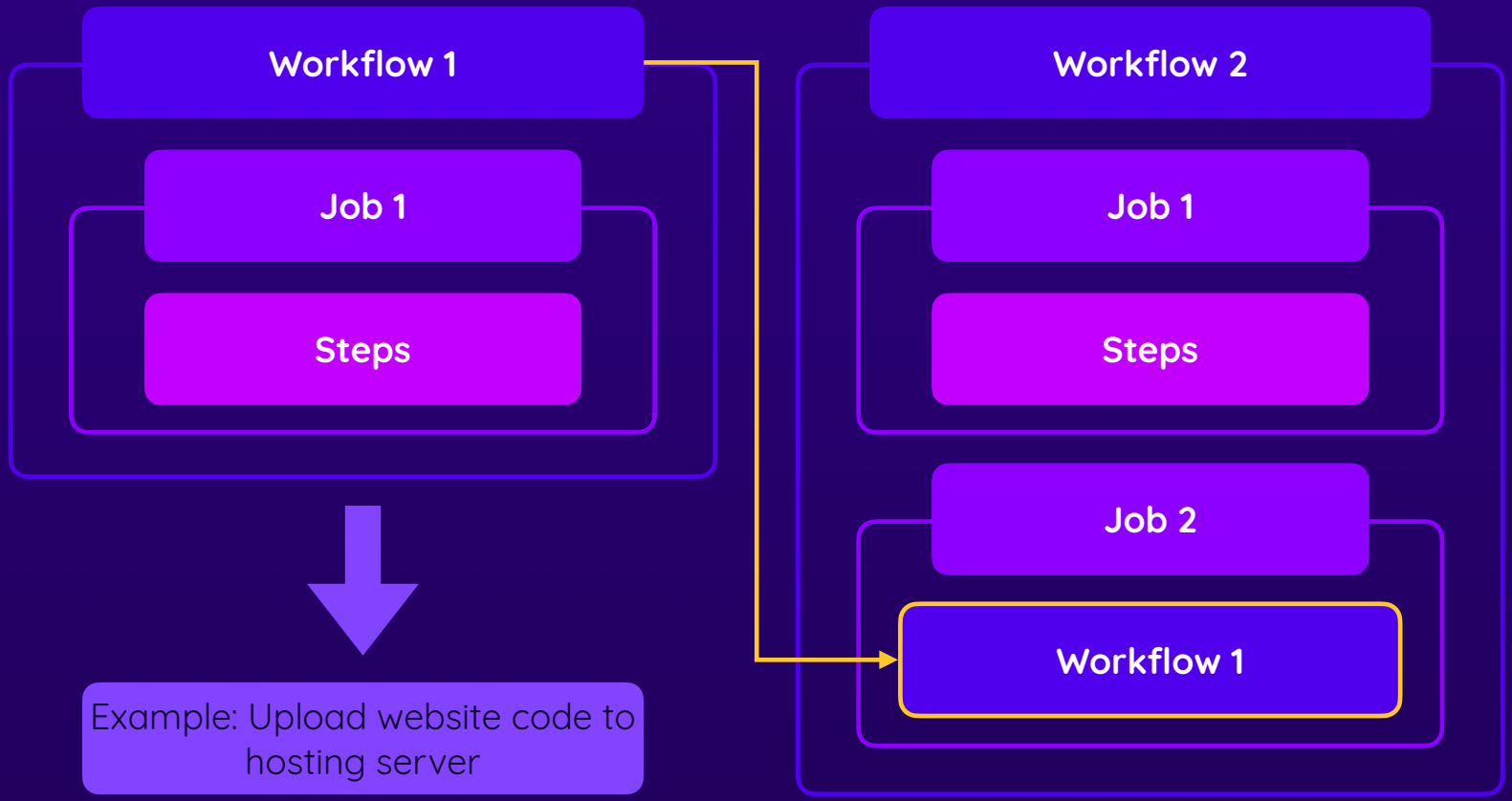
always()

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	Matrix Jobs	Reusable Workflows
Control Step or Job execution with `if` & dynamic expressions	Run multiple Job configurations in parallel	Workflows can be reused via the `workflow_call` event
Change default behavior with `failure()`, `success()`, `cancelled()` or `always()`	Add or remove individual combinations	Reuse any logic (as many Jobs & Steps as needed)
Use `continue-on-error` to ignore Step failure	Control whether a single failing Job should cancel all other Matrix Jobs via `continue-on-error`	Work with `inputs`, `outputs` and `secrets` as required

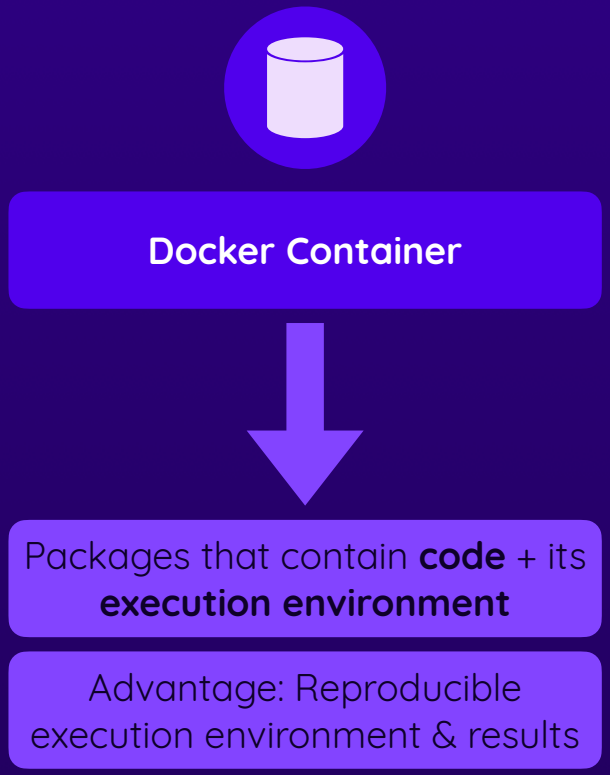
ACADEMIND

Using Containers

Utilizing Docker Containers

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

What Are Containers?



Containers & GitHub Actions

Docker Container

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

Advantage: Reproducible execution environment & results

still runs on

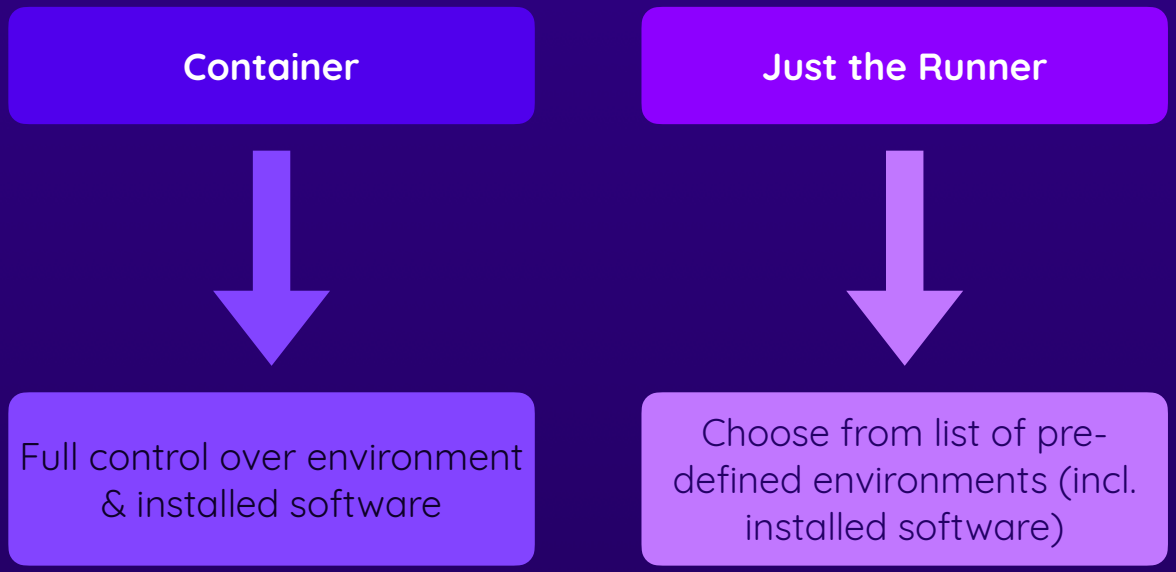
GitHub Actions Runners

Your containerized Job is hosted by the Runner

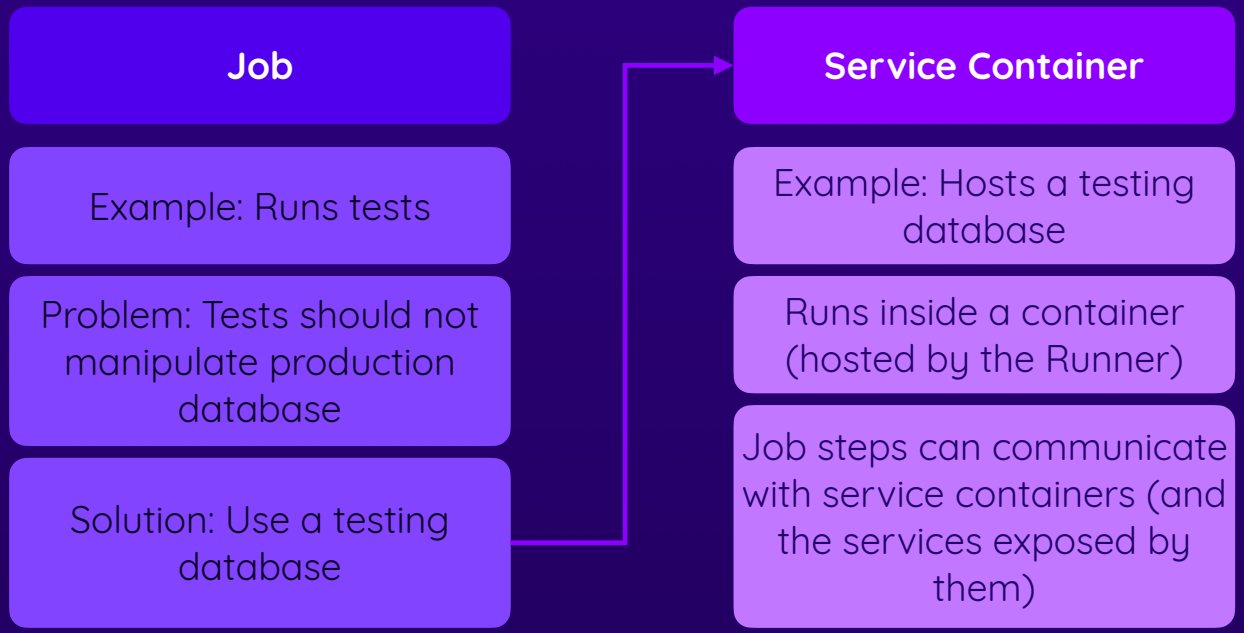
Steps execute inside the container

You can also create **Services**: Utility containers used by your Steps (Example: Testing Database)

Why Use Containers?



Using Service Containers ("Services")



Module Summary

Containers	Containers for Jobs	Service Containers
Packages of code + execution environment	You can run Jobs in pre-defined environments	Extra services can be used by Steps in Jobs
Great for creating re-usable execution packages & ensuring consistency	Build your own container images or use public images	Example: Locally running, isolated testing database
Example: Same environment for testing + production	Great for Jobs that need extra tools or lots of customization	Based on custom images or public / community images

ACADEMIND

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	Docker Actions	Composite Actions
Execute a JavaScript file	Create a Dockerfile with your required configuration	Combine multiple Workflow Steps in one single Action
Use JavaScript (NodeJS) + any packages of your choice	Perform any task(s) of your choice with any language	Combine `run` (commands) and `uses` (Actions)
Pretty straightforward (if you know JavaScript)	Lots of flexibility but requires Docker knowledge	Allows for reusing shared Steps (without extra skills)

ACADEMIND

Module Summary

What & Why?	Composite Actions	JavaScript & Docker Actions
Simplify Workflows & avoid repeated Steps	Create custom Actions by combining multiple Steps	Write Action logic in JavaScript (NodeJS) with @actions/toolkit
Implement logic that solves a problem not solved by any publicly available Action	Composite Actions are like “Workflow Excerpts”	Alternatively: Create your own Action environment with Docker
Create & share Actions with the Community	Use Actions (via `uses`) and Commands (via `run`) as needed	Either way: Use inputs, set outputs and perform any logic

ACADEMIND

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

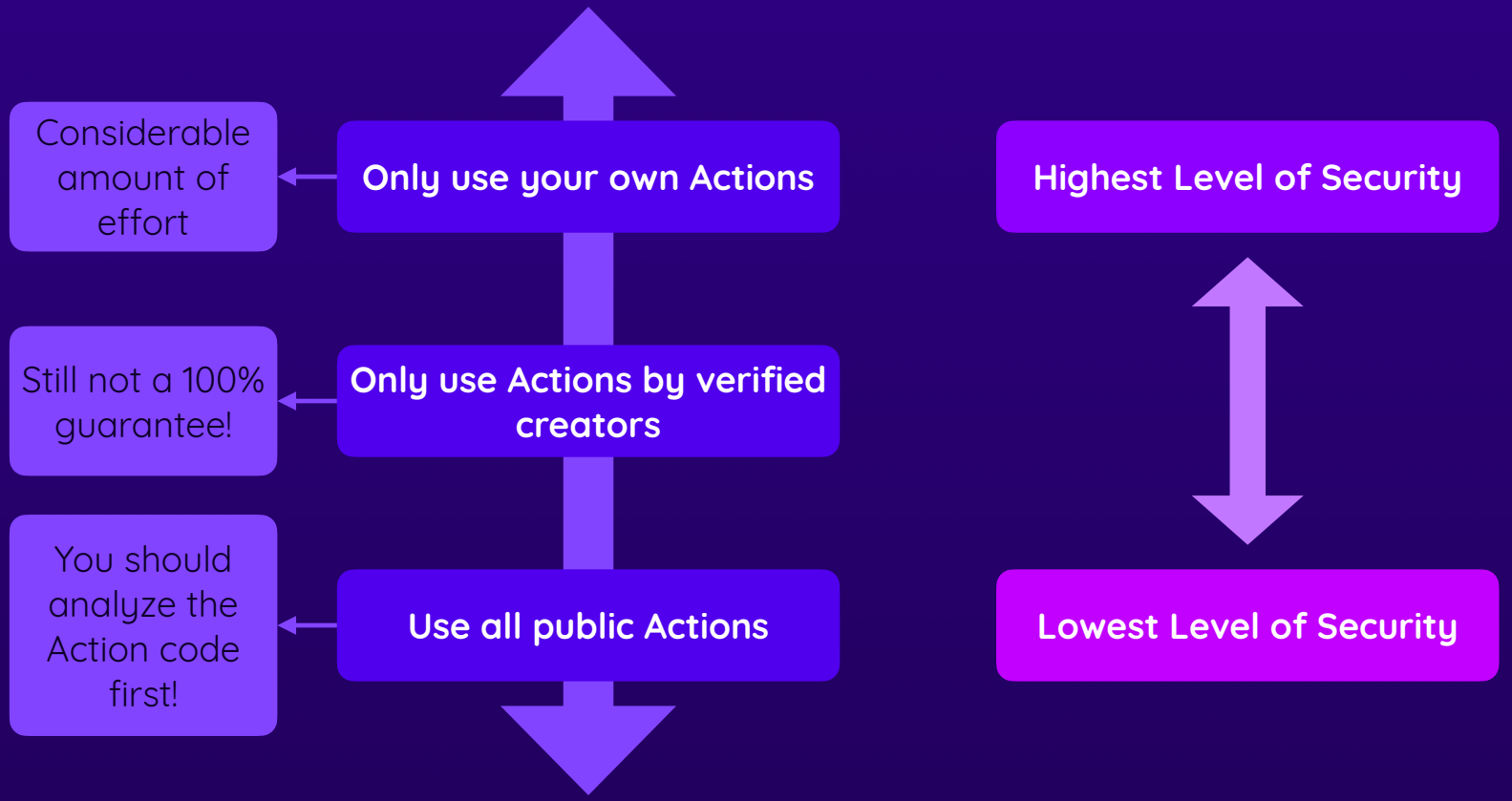
Permission Issues

Consider avoiding overly permissive permissions

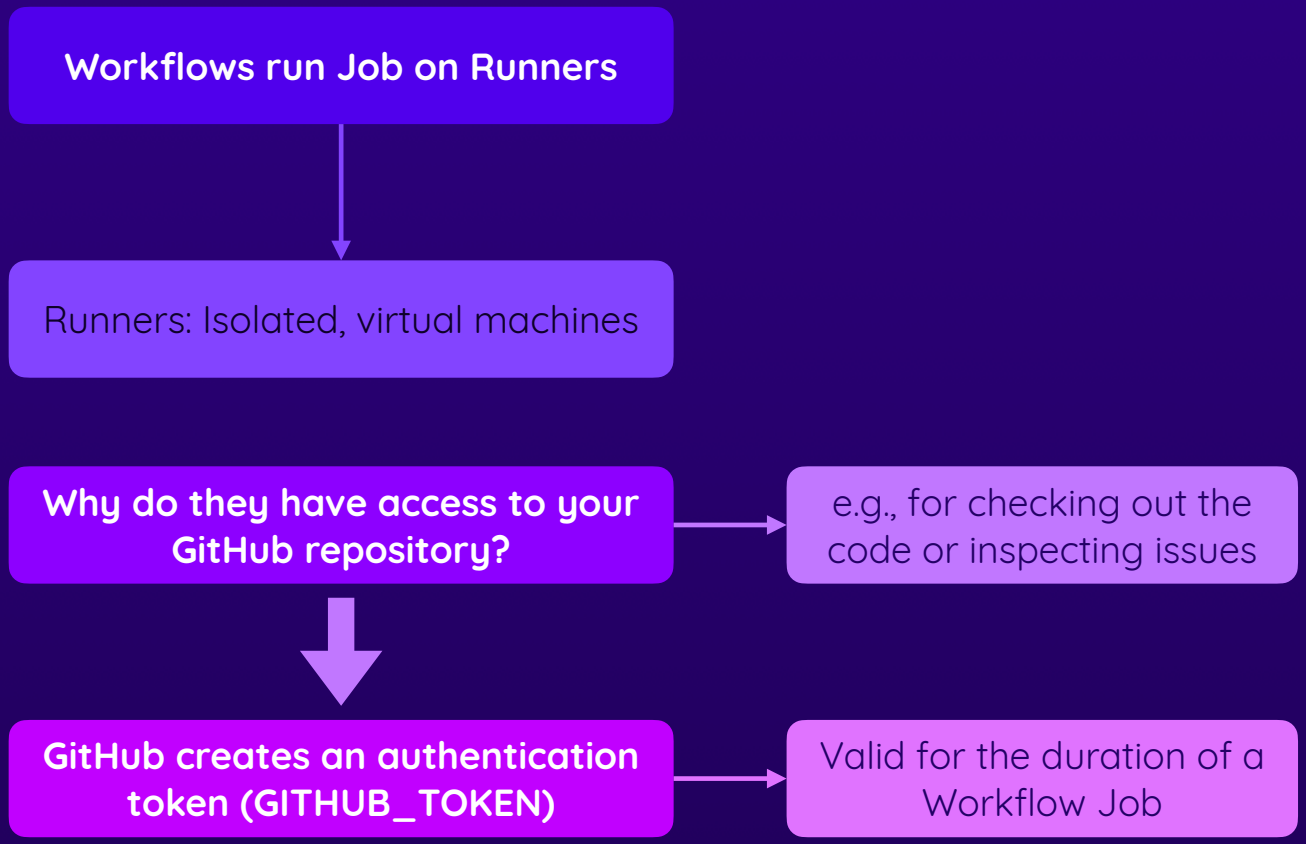
Example: Only allow checking out code (“read-only”)

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