

# DevOps Foundations: Version Control and CI/CD with Jenkins



## CI/CD with GitHub Actions



# Learning Objectives

By the end of this lesson, you will be able to:

- Identify different working components of GitHub Actions to handle the workflow execution and resource management
- Create a basic CI/CD pipeline using GitHub Actions workflow for automating the software deployment process
- Design and implement conditional workflows within GitHub Actions to respond to different triggers and conditions
- Evaluate the effectiveness of a GitHub Actions workflow for automating build, test, and deployment task
- Formulate strategies to optimize CI/CD pipelines using advanced GitHub Actions features like manual triggers, scheduled events, and dependency controls

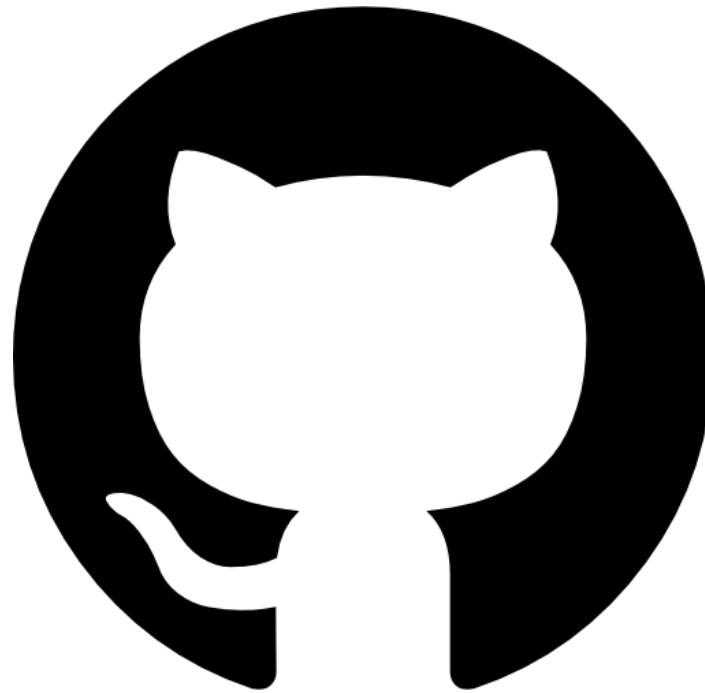




# Getting Started with GitHub Actions

# History of GitHub Actions

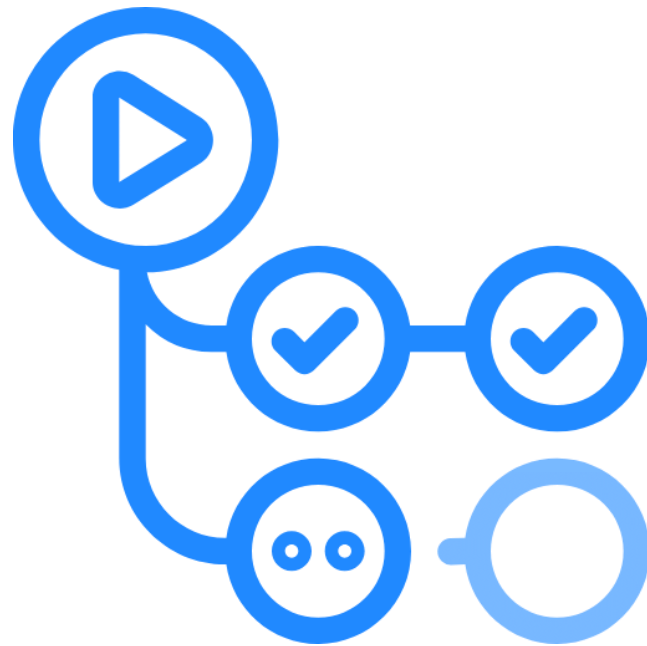
GitHub is one of the most popular Git hosting platforms that hosts more than 200 million repositories, and it was launched in 2008.



Developers were only able to perform CI/CD workflows with third-party tools such as Jenkins, Bamboo, Travis CI, and CircleCI.

# History of GitHub Actions

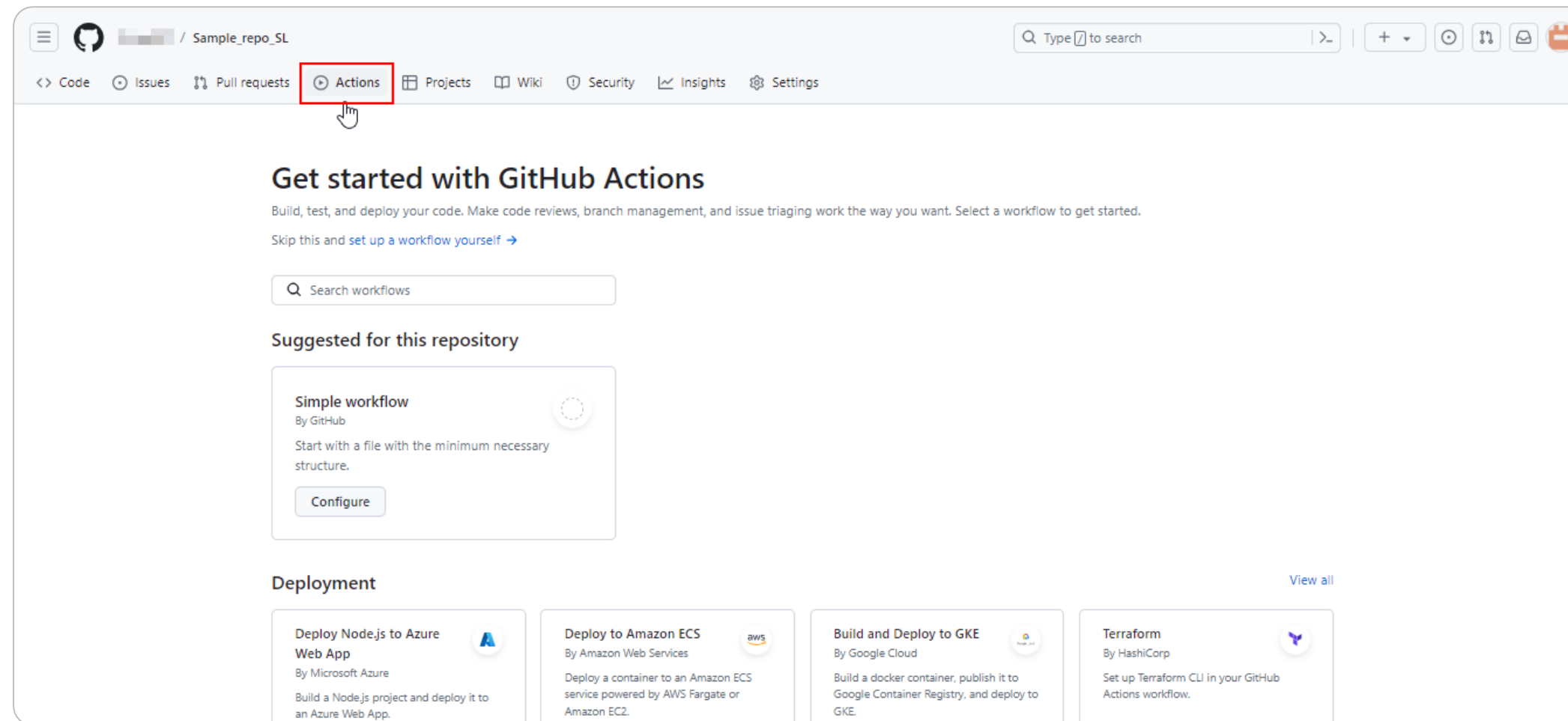
In November 2019, GitHub changed the whole process when it announced GitHub Actions, an integrated CI/CD pipeline platform.



- GitHub Actions implement text-based configurations present in the Git repository.
- It helps developers automate builds, test execution, and deployment.
- Developers can create automated workflows that will be invoked every time they perform a new change or pull request to their repository.

# Introduction to GitHub Actions

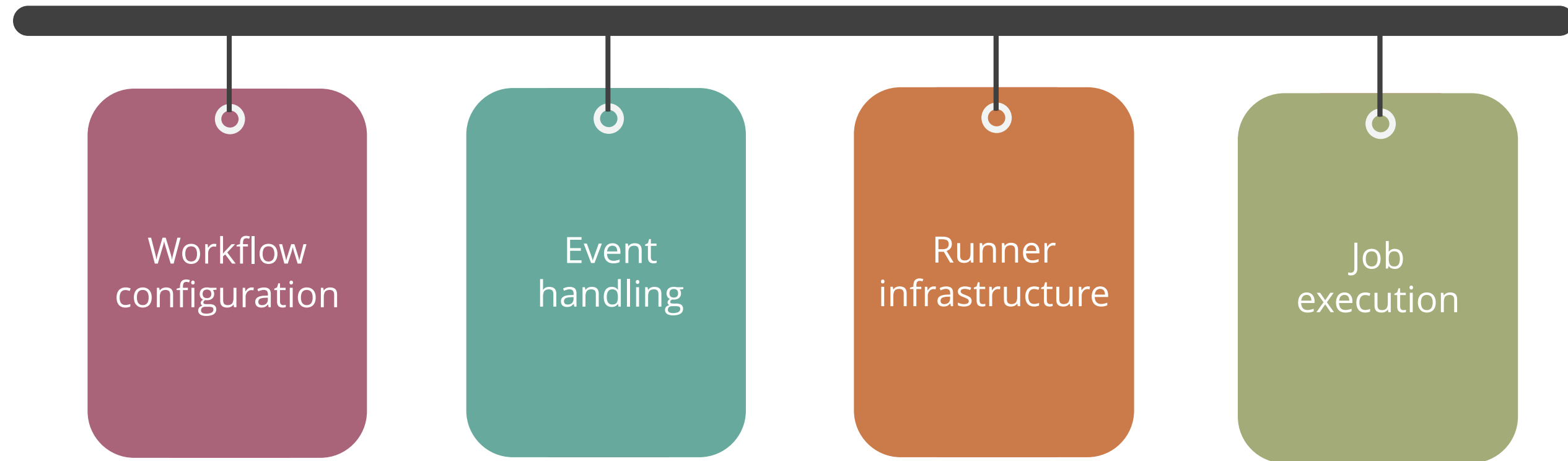
GitHub Actions is a built-in continuous integration and continuous delivery (CI/CD) platform in GitHub. It supports different DevOps and build tools including Maven, Docker, Ansible, and Kubernetes.



It automates the software development lifecycle, specifically focusing on automating tasks like building, testing, and deploying code.

# Backend Architecture of GitHub Actions

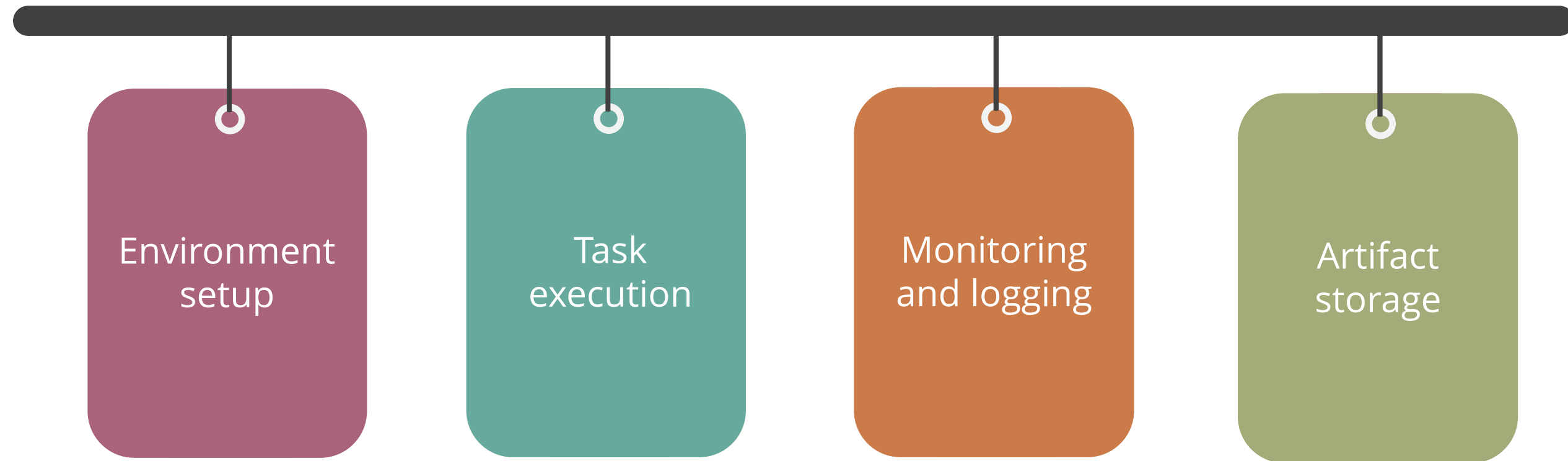
The backend architecture of GitHub Actions handles the workflow execution and manages resources to effectively run the actions. Following are the key working elements:





# Backend Architecture of GitHub Actions

Following are the key working elements:



# Use of GitHub Actions

## Continuous integration and delivery

- Build and test code
- Automate code deployments

## Improved code quality

- Perform static code analysis
- Integrate security scanning tools

## Automated workflows

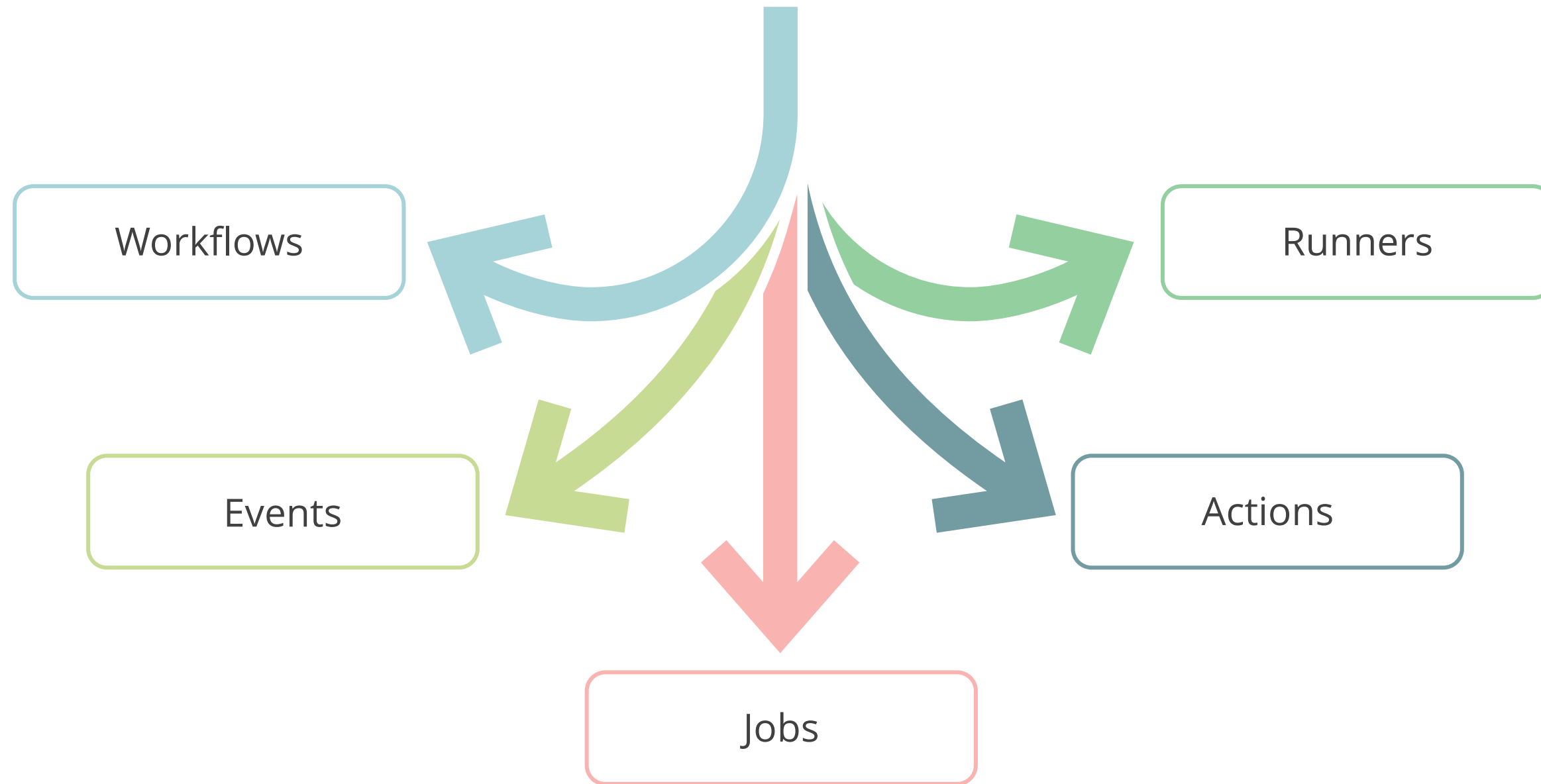
- Label and manage issues
- Generate code documentation

## Collaboration and communication

- Set up notifications for various actions

# Components of GitHub Actions

GitHub Actions supports multiple components that can help automate the development lifecycle from development to deployment in a production environment. Some of the major components are:



# Workflows

Workflows are defined in the form of a YAML file that contains configurable automated process to run one or more jobs. These workflows can be triggered manually or at a defined schedule.

Following are the ways to trigger a workflow:

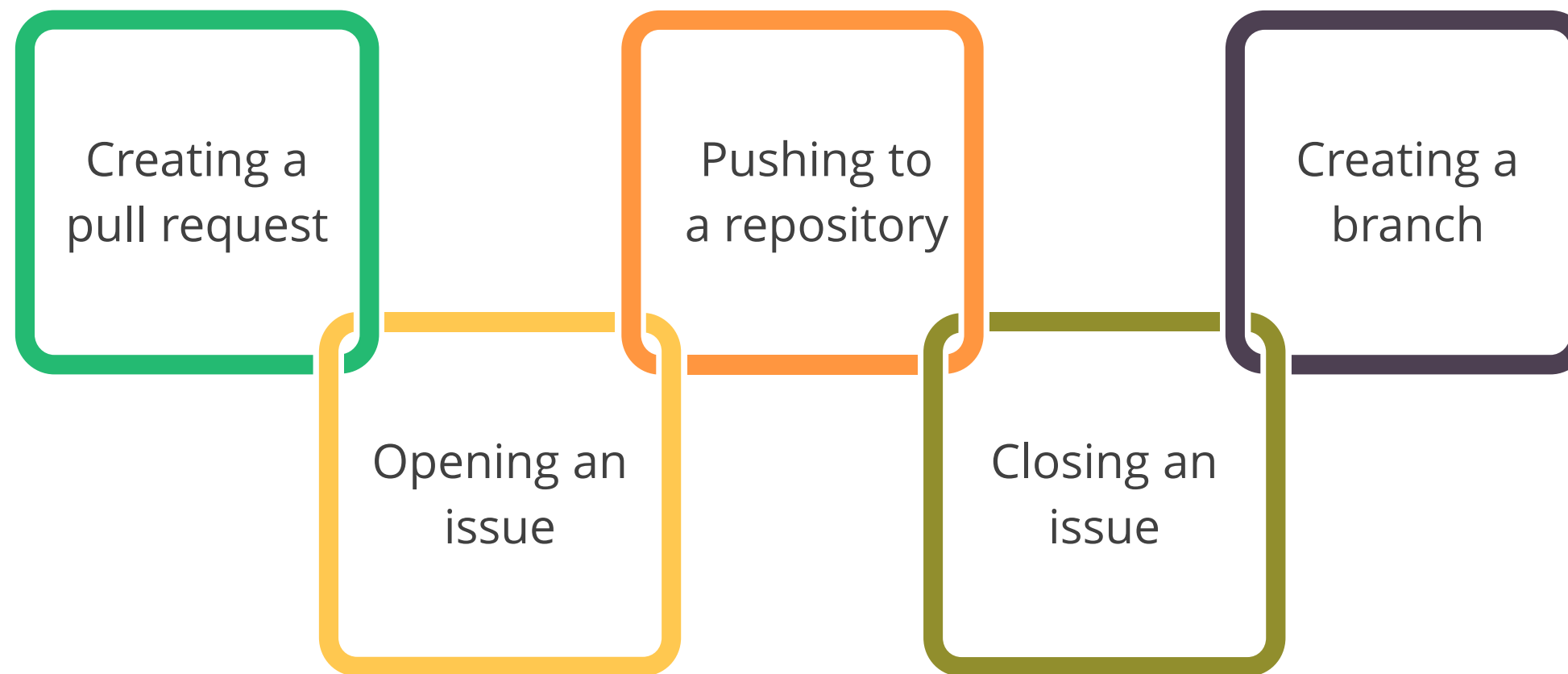
Perform a pull request or code push or modify the source code

Schedule a trigger at a particular time

Run the workflows manually

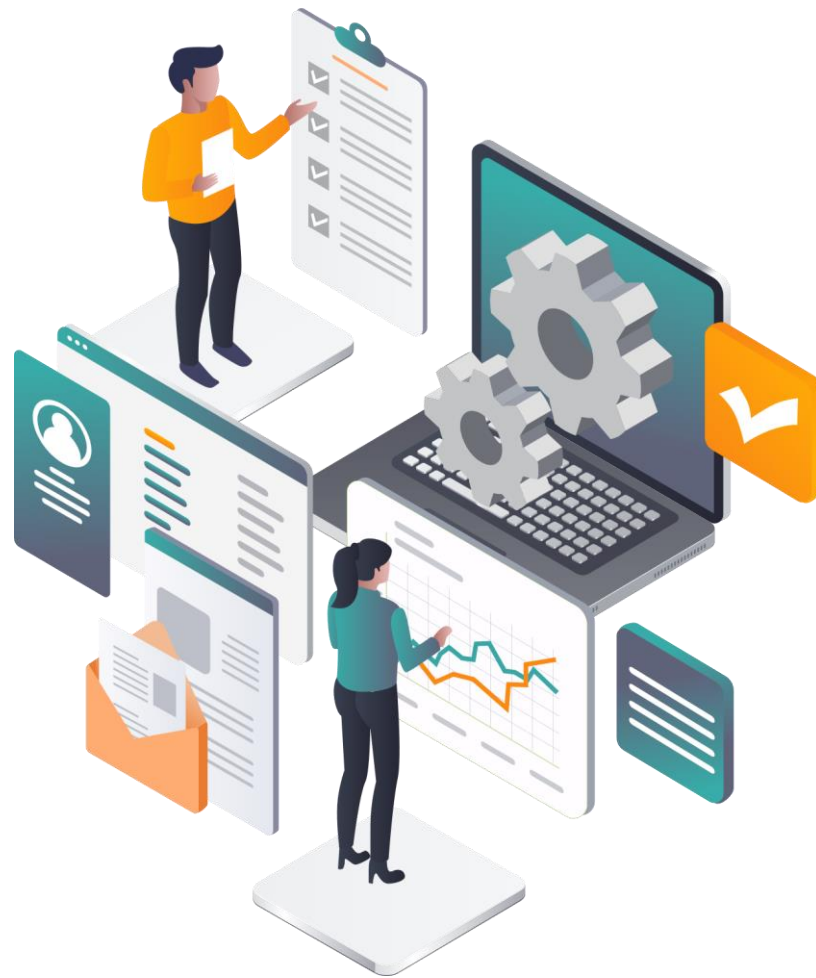
# Events

Events are the triggers that initiate a workflow run. They define how users want a workflow to run for various repository activities, such as:



# Jobs

A job represents a specific stage or task within a workflow. It groups a series of steps that need to be executed sequentially to achieve a particular goal.



## Purpose:

- In a workflow, users can configure multiple jobs with steps to perform a process such as code checkout, build automation, and test execution.
- Users can configure interdependent jobs per the developer's requirements, which will hold the execution of the second job in case the first job execution is under process.

# Runners

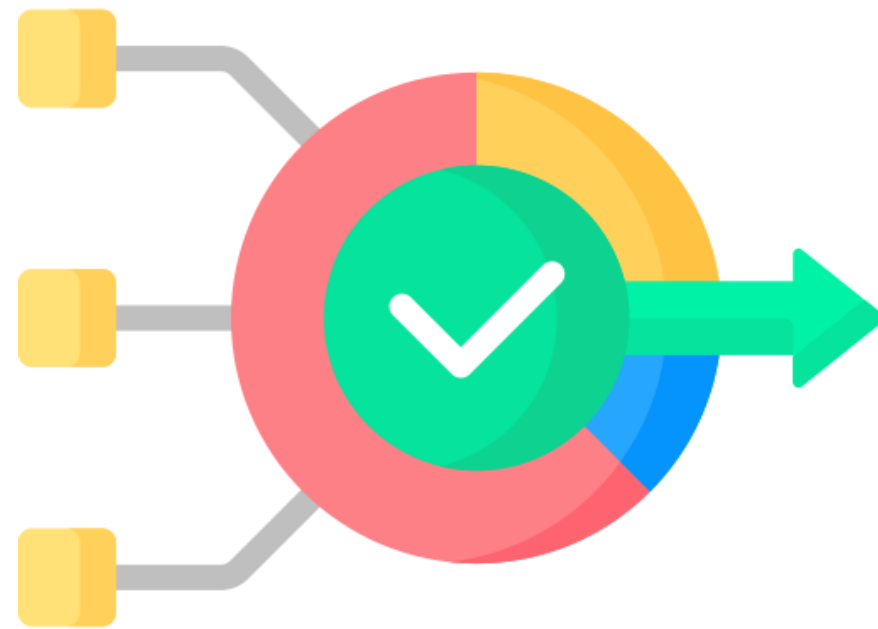
Runners are the machines that execute the jobs and steps within a workflow. Whenever a workflow is triggered, it requires a build server to run it, and these servers are called runners.



These runners produce all the required build tools and runtime so that you don't have to go through the pain of installing them manually.

# Actions

Actions define the exact operation users want to execute as part of their CI/CD workflow.



They can include multiple actions in their workflow such as checking out code, creating Maven builds, and building some Docker images and deploying them to a cloud platform.



## Quick Check



In setting up a GitHub workflow for a new project, you're tasked with ensuring that the workflow can access the code in the repository. Among the components listed below, which one is responsible for checking out the code from the repository so that the workflow can interact with it?

- A. Event trigger
- B. Runner
- C. Job
- D. Action



# **Working with Workflows in GitHub Actions**

# Workflow Components

GitHub workflows are written in YAML format and are stored in **.github/workflows** directory to perform different sets of tasks.

GitHub workflow must contain the following basic components while designing:

## Event

It triggers the workflow.

## Job

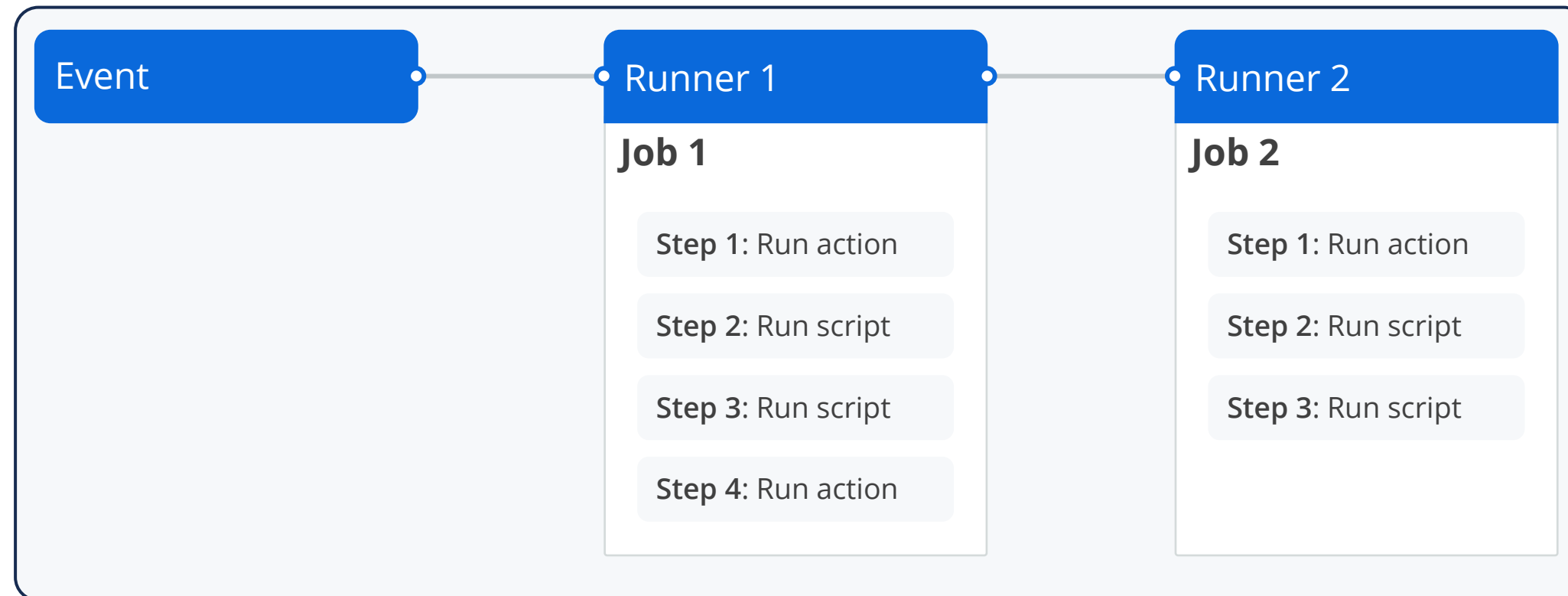
It is a set of steps in a workflow that is executed on the same runner.

## Step

It configures the workflow to perform various operations on the source code.

# Workflow Components

The following depicts the workflow components in the GitHub Actions:



Workflows are made up of jobs, and each job can have multiple steps that involve running scripts or predefined actions from the GitHub Marketplace.

# Workflow Syntax

Below are some of the essential parts of the workflow syntax:

## name

This attribute specifies the workflow name, and GitHub displays it in the **Actions** tab of the repository.

## run-name

This assigns a name to workflow runs generated from the workflow and triggered by a **push** or **pull\_request**; the run name is configured as the commit message.

# Workflow Syntax

**on**

It defines which events can cause the workflow to run.

**on.schedule**

It defines a time schedule for the workflows to be invoked without performing any change to the code repository.

**on.workflow\_dispatch**

It is used to invoke the workflow manually.

# Workflow Example

```
name: First Workflow

on:
  pull_request:
    branches: ["main"]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: Command execution
        run: echo "Executing First GitHub Actions workflow"
```

# Workflow Commands

Workflow commands are used to run shell commands in a workflow or an action within a workflow. These actions are created to interact with the runner machine.

- The **echo** command is used in the following way to send a custom instruction to the GitHub Actions runner.
- The runner interprets the message and performs the specified action with the provided parameters.

## Example

```
echo "::workflow-command  
parameter1={data},parameter2={data}::{command value}"
```



# Workflow Commands

## Setting a debug message

This will print a debug message to the workflow log. The user must create a secret with the name **ACTIONS\_STEP\_DEBUG** with the value true in the repository that contains the workflow.

### Syntax

```
::debug::{message}
```

### Example

```
echo "::debug::Set the Octocat variable"
```

# Workflow Commands

## Setting a notice message

This creates a notice message and prints the messages in the workflow log.

### Syntax

```
::notice  
file={name},line={line},endLine={endLine},title={title}::{message}
```

### Example

```
echo "::notice file=node.js,line=1,col=5,endColumn=7::Missing  
semicolon"
```

# Workflow Commands

## Setting a warning message

This creates a warning message and prints it to the workflow log.

### Syntax

```
::warning  
file={name},line={line},endLine={endLine},title={title}::{message}
```

### Example

```
echo "::warning file=node.js,line=1,col=5,endColumn=7::Missing  
semicolon"
```

# Workflow Commands

## Setting an error message

This creates an error message and prints it to the workflow log.

### Syntax

```
::error  
file={name},line={line},endLine={endLine},title={title}::{message}
```

### Example

```
echo "::error file= node.js,line=1,col=5,endColumn=7::Missing  
semicolon"
```

# Using Jobs in Workflow

Jobs are created in the workflow files to configure what the user wants the workflow to perform on GitHub runners.

The user can:

Configure one or multiple jobs in a workflow; jobs run in parallel by default

Sequence the jobs using **jobs.<job\_id>.needs** keyword

Configure runner details in the workflow file for each job's execution

# Using Jobs in Workflow

The user can configure **jobs.<job\_id>** to provide a unique identifier for every job in the workflow.

## Example: Job configurations with job dependency

```
jobs:
  job_1:
  job_2:
    needs: job_1
  job_3:
    needs: [job_1, job_2]
```

Job id is a string value and must start with a letter or underscore (\_) or any alphanumeric characters like comma (,) or underscore (\_).

## Assisted Practice



### Creating and Executing a Basic GitHub Actions Workflow

Duration: 10 Min.

#### Problem statement:

You have been assigned a task to create and execute a basic GitHub Actions workflow for automated testing and deployment, enhancing the efficiency and reliability of project development processes.

#### Outcome:

By the end of this demo, you will create and execute a GitHub Actions workflow for automated testing and deployment, improving project efficiency and reliability.

**Note:** Refer to the demo document for detailed steps:

# Assisted Practice: Guidelines



Steps to be followed:

1. Create a new GitHub repository
2. Create and execute a new workflow file



## Quick Check

To install dependencies before running tests in a Python project utilizing a **requirements.txt** file, which workflow step is the most effective?

- A. Define an environment variable named `REQUIREMENTS_FILE` pointing to "requirements.txt" and use it within a pip install command
- B. Add a script that copies "requirements.txt" to the runner's working directory and then runs `pip install -r requirements.txt`
- C. Include "requirements.txt" directly within the workflow YAML file and reference it in the pip install command
- D. Leverage a pre-built action from the Actions Marketplace specifically designed for installing Python dependencies





## Essential Features of GitHub Workflow

# Adding Scripts to the Workflow

The user can use a GitHub Actions workflow to run scripts and shell commands, which are then executed on the assigned runner.

## Example

```
jobs:
  example-job:
    runs-on: ubuntu-latest
    steps:
      - run: npm install -g bats
```

The above example demonstrates how to use the run keyword to execute the command **npm install -g bats** on the runner.

# Using Environment Variables

GitHub Actions include default environment variables for each workflow run. If the user needs to use custom environment variables, they can set these in the YAML workflow file.

## Example

```
jobs:
  example-job:
    runs-on: ubuntu-latest
    steps:
      - name: Connect to PostgreSQL
        run: node client.js
        env:
          POSTGRES_HOST: postgres
          POSTGRES_PORT: 5432
```

The above example demonstrates how to create custom variables named **POSTGRES\_HOST** and **POSTGRES\_PORT**.

# Sharing Data between Jobs

If the job generates files that the users want to share with another job in the same workflow, or if the users want to save the files for later reference, they can store them in GitHub as *artifacts*.

## Example

```
jobs:
  example-job:
    name: Save output
    runs-on: ubuntu-latest
    steps:
      - shell: bash
        run: |
          expr 1 + 1 > output.log
      - name: Upload output file
        uses: actions/upload-artifact@v4
        with:
          name: output-log-file
          path: output.log
```

# Expressions in GitHub Workflow

Expressions are combinations of literals (fixed values), context references, functions, and operators. They are used to define values or conditions that can change based on the specific workflow run.

## Syntax

```
${{ <expression> }}
```

Expressions are commonly used with the conditional **if** keyword to validate if a step should be executed or not. Once the **if** condition is **true**, a step will be executed.

# Expressions in GitHub Workflow

## Syntax: Expression in an **if** condition

```
steps:  
  - uses: actions/hello-world-javascript-action@main  
    if: ${{ <expression> }}
```

## Example: Setting an environment variable

```
env:  
  MY_ENV_VAR: ${{ <expression> }}
```

# Contexts in GitHub Workflow

They are a mechanism to access information about workflow runs, variables, runner environments, jobs, and steps. Each context is an object that contains properties, which can be strings or other objects.

Access contexts using the expression syntax:

```
${{ <context> }}
```

Contexts, objects, and properties will vary significantly under different workflow run conditions. For example, the *matrix* context is only populated for jobs in a matrix.



# Contexts in GitHub Workflow

Following are the different contexts in GitHub Actions:

github

job

runner

matrix

env

jobs

secrets

needs

vars

steps

strategy

inputs

# Variables

They are used to store and reuse non-sensitive information. Users can store configuration data such as compiler flags, usernames, or server names as variables.



These variables are stored on runner machines that run the workflow. Using variables improves code readability, maintainability, and security.

# Variables

They can be created, read, and modified by commands in actions or workflow steps.

There are two main ways to define variables in GitHub Actions:

## Environment variables

- Defined to use in a single workflow
- Defined using the **env** keyword

## Configuration variables

- Defined to work across multiple workflows
- Accessed using the **vars** context

# Initializing the Workflow Variables

Following are the ways to assign values to the workflow variables:

Assigning string literals

```
env:  
  BUILD_NUMBER: "123"
```

Assigning context

```
env:  
  BUILD_VERSION: ${ github.sha }
```

Assigning GitHub secrets

```
env:  
  API_KEY: ${ secrets.MY_API_KEY }
```

# Defining Environment Variables for a Single Workflow

Users can define an environment variable in a single workflow, which can be used anywhere within the workflow with commands.

The user can define variables that are scoped for:

The entire workflow, by using **env** at the top level of the workflow file

The contents of a job within a workflow, by using **jobs.<job\_id>.env**

A specific step within a job, by using **jobs.<job\_id>.steps[\*].env.**

# Defining Configuration Variables for Multiple Workflows

The user can create configuration variables for use across multiple workflows.

The user can define variables at either one of the following levels:

Organization level

Repository level

Environment level

# Choosing the Runner for a Job

The runners define the type of machine that will process a job in the workflow.

The user can include a sizable cloud server as a runner in the GitHub Actions.

In a workflow file, you can configure **jobs.<job\_id>.runs-on** to specify where to execute a job.

The **runs-on** parameters can have a single runner value or an array.

GitHub workflows need a machine to perform configured tasks, which can be GitHub-hosted or self-hosted runners.

# GitHub-Hosted Runners

They are preinstalled runner applications and tools on virtual machines (VMs) that are available with Ubuntu Linux, Windows, or macOS operating systems.

The below table shows standard GitHub-hosted runners for public repositories:

Virtual Machine	Processor (CPU)	Memory (RAM)	Storage (SSD)	OS (YAML workflow label)	Notes
Linux	4	16 GB	150 GB	<code>ubuntu-latest</code> , <code>ubuntu-22.04</code> , <code>ubuntu-20.04</code>	The <code>ubuntu-latest</code> label currently uses the Ubuntu 22.04 runner image.
Windows	4	16 GB	150 GB	<code>windows-latest</code> , <code>windows-2022</code> , <code>windows-2019</code>	The <code>windows-latest</code> label currently uses the Windows 2022 runner image.
macOS	3	14 GB	14 GB	<code>macos-latest</code> , <code>macos-12</code> , <code>macos-11</code>	The <code>macos-latest</code> workflow label currently uses the macOS 12 runner image.
macOS	4	14 GB	14 GB	<code>macos-13</code>	N/A
macOS	3 (M1)	7 GB	14 GB	<code>macos-14</code> [Beta]	N/A



# GitHub-Hosted Runners

The below table shows standard GitHub-hosted runners for private repositories:

Virtual Machine	Processor (CPU)	Memory (RAM)	Storage (SSD)	OS (YAML workflow label)	Notes
Linux	2	7 GB	14 GB	<code>ubuntu-latest</code> , <code>ubuntu-22.04</code> , <code>ubuntu-20.04</code>	The <code>ubuntu-latest</code> label currently uses the Ubuntu 22.04 runner image.
Windows	2	7 GB	14 GB	<code>windows-latest</code> , <code>windows-2022</code> , <code>windows-2019</code>	The <code>windows-latest</code> label currently uses the Windows 2022 runner image.
macOS	3	14 GB	14 GB	<code>macos-latest</code> , <code>macos-12</code> , <code>macos-11</code>	The <code>macos-latest</code> workflow label currently uses the macOS 12 runner image.
macOS	4	14 GB	14 GB	<code>macos-13</code>	N/A
macOS	3 (M1)	7 GB	14 GB	<code>macos-14</code> [Beta]	N/A

# Self-Hosted Runners

They can be configured in jobs if the user prefers managing runners using their infrastructure.

The developer needs to configure the **runs-on** keyword with the self-hosted runner labels.

They offer more control of hardware, operating system (OS), and software tools than the GitHub-hosted runners.

# Using Conditions to Control Job Execution

The workflow supports conditional statements for job execution.



You can configure the **if** condition for jobs using **jobs.<job\_id>.if** to specify when the job should be executed based on the given condition.

# Using Conditions to Control Job Execution

## Example

```
name: build-workflow
on: [push]
jobs:
  production-deploy:
    if: github.repository == 'simplilearn-org/counterapp-prod'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g angular-cli
```

# Assisted Practice



## Creating a conditional workflow

Duration: 10 Min.

### Problem statement:

You have been assigned a task to create a conditional workflow to specify when a particular job or step should run.

### Outcome:

By the end of this demo, you will be able to create a conditional workflow in GitHub Actions, specifying when jobs or steps should run based on defined conditions.

**Note:** Refer to the demo document for detailed steps:

# Assisted Practice: Guidelines



Steps to be followed:

1. Create a new GitHub repository
2. Create and execute a new conditional workflow file

## Quick Check



In a GitHub Actions workflow, you need to set up an environment variable that can be accessed by all jobs within the workflow. Which of the following methods would achieve this?

- A. Add the variable to the **env** key at the workflow level
- B. Use the **set-env** command in a specific job to set the variable
- C. Add the variable to the **env** key at the job level for each job.
- D. Use **GITHUB\_ENV** to set the variable within a specific job

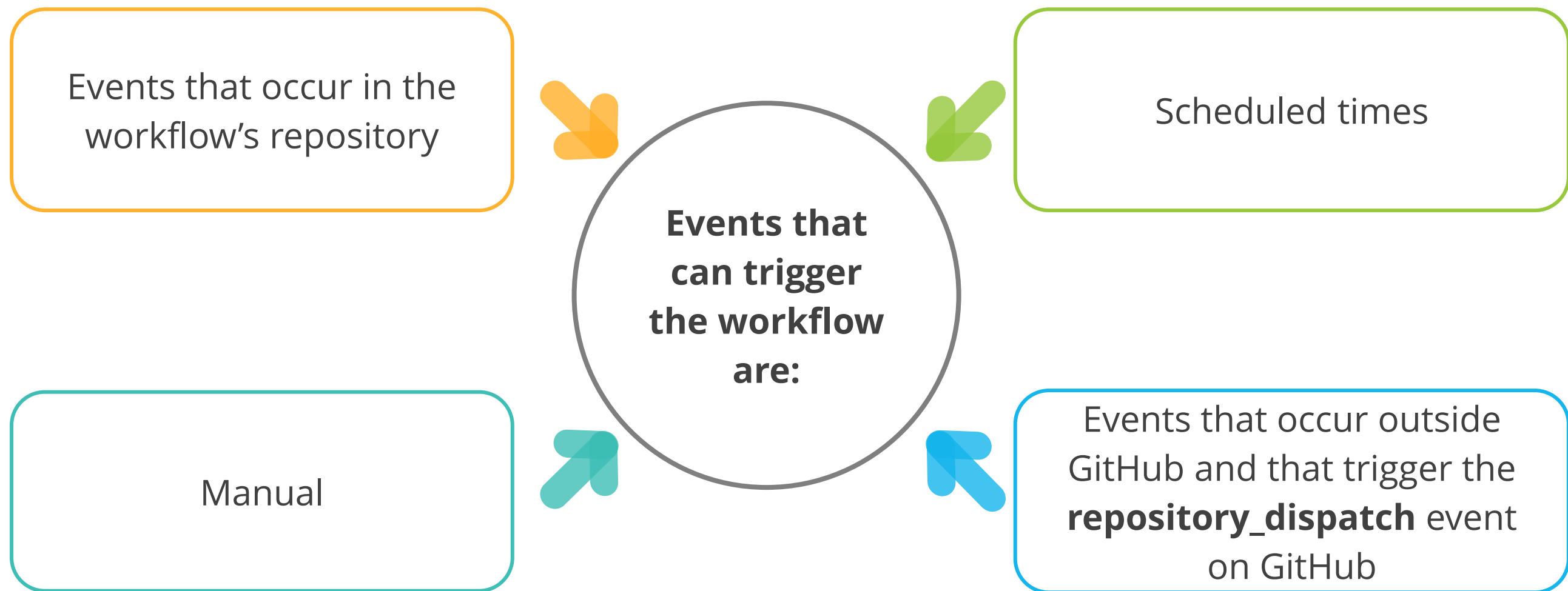


## Triggering and Reusing Workflows



# Triggering a Workflow

It is an event that helps to trigger a workflow and execute jobs configured inside it.



# Trigger Workflows Using Events

Below are the two ways of triggering workflows using events:

## Using single event

The developer can invoke workflow by using a single event.  
Example: **on: push**

## Using multiple events

In the workflow, the developer can configure multiple events to invoke when any specified event occurs.  
Example: **on: [push, fork]**

# Trigger Workflows Using Events

In the below example, there is a push event with a filter on the branches that can invoke the workflow instead of push activity on every branch.

## Example

```
on:
  push:
    branches:
      - main
      - 'feature/**'
```

# Trigger Workflows Using Events

## Example

```
name: First Workflow
on:
  pull_request:
    branches: [ "main" ]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Command execution
        run: echo "Executing First GitHub Actions workflow"
```

## Assisted Practice



### Creating an event-based workflow

**Duration: 10 Min.**

#### Problem statement:

You have been assigned a task to create an event-based workflow using the GitHub Actions trigger to automatically initiate continuous integration processes whenever new code is pushed to the repository.

#### Outcome:

By the end of this demo, you will be able to create an event-based workflow using GitHub Actions triggers to automatically start continuous integration when new code is pushed to the repository.

**Note:** Refer to the demo document for detailed steps:

# Assisted Practice: Guidelines

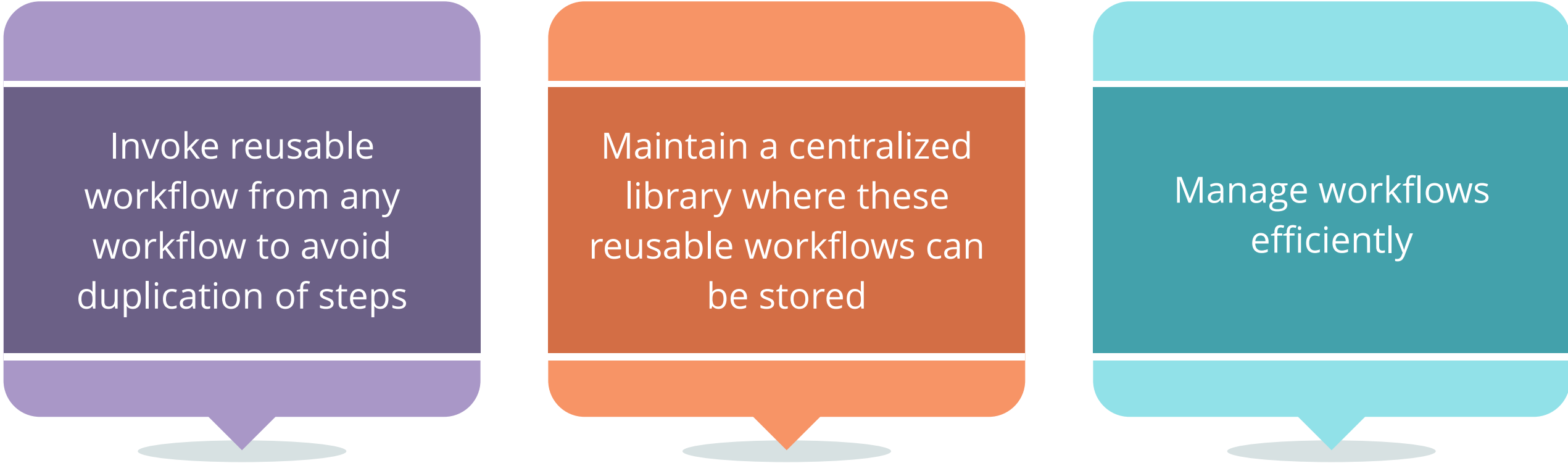


Steps to be followed:

1. Create a new GitHub repository
2. Create and execute a new workflow file using the GitHub Actions trigger

# Reusing Workflows

The following are the ways in which developers are benefited by reusing workflows:



Invoke reusable workflow from any workflow to avoid duplication of steps

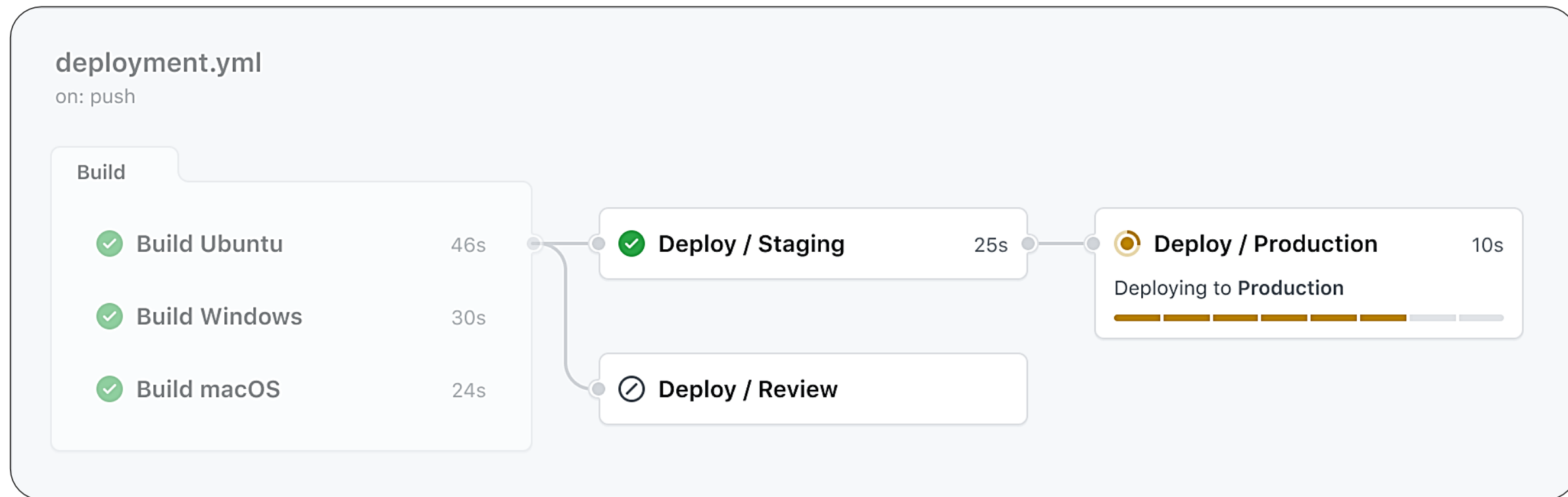
Maintain a centralized library where these reusable workflows can be stored

Manage workflows efficiently

# Reusing Workflows

A workflow that implements another workflow is called a **caller** workflow, and a reusable workflow is known as a **called** workflow.

The diagram below shows an in-progress workflow run that uses a reusable workflow:





# Reusing Workflows

After executing three build jobs for different environments, a dependent job called **Deploy** is executed on a successful build.

The **Deploy** job calls a reusable workflow that contains three jobs: **Staging**, **Review**, and **Production**.

Once the **Staging** job is completed, then only the **Production** deployment job would be invoked.

Using **Deploy** as a reusable workflow allows you to execute those jobs without repeating them multiple times in the workflow.

# Reusing Workflows

Basic implementation of reusable workflow:

## Example

```
name: Reusable Workflow
on:
  workflow_dispatch:

jobs:
  code_checkout:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
```

# Reusing Workflows

Implementation of reusing a reusable workflow:

## Example

```
name: Main Workflow

on:
  push:
    branches:
      - main
jobs:
  use_reusable_workflow:
    runs-on: ubuntu-latest
    steps:
      - name: Use Reusable Workflow
        uses: ../github/workflows/my-reusable-workflow.yml
```

# Key Takeaways

- GitHub Actions is a built-in continuous integration and continuous delivery (CI/CD) platform in GitHub. It supports different DevOps and build tools including Maven, Docker, Ansible, and Kubernetes.
- GitHub workflows are defined in the **.github/workflows** directory in a repository, and a repository can have multiple workflows, each of which can perform a different set of tasks.
- Workflows are made up of one or more jobs, and each job can have multiple steps that involve running scripts or predefined actions from the GitHub Marketplace.
- A job represents a specific stage or task within a workflow. It groups a series of steps that need to be executed sequentially to achieve a particular goal.



# Key Takeaways

- GitHub Actions include default environment variables for each workflow run. If the user needs to use custom environment variables, they can set these in the YAML workflow file.
- Contexts in GitHub Actions workflows provide access to information about the current workflow run, the runner environment, and various other aspects of the execution.
- Events are the triggers that initiate a workflow run. They define how users want a workflow to run for various repository activities.
- Expressions are combinations of literals (fixed values), context references, functions, and operators.



# Executing CI/CD with GitHub Actions

**Duration: 20 Min.**

**Project agenda:** To create an event-based workflow using Git and GitHub Actions for efficient project automation and version control

**Description:** As a DevOps engineer at a tech company embarking on a significant project, the success of the team depends on seamless collaboration and rapid delivery of high-quality code. To supercharge the development process and maintain code integrity, the focus is on diving into continuous integration (CI) and continuous deployment (CD) workflows using Java CI with Maven. The aim is to revolutionize the workflow and propel the project to new heights of efficiency and excellence.



# Executing CI/CD with GitHub Actions

Duration: 20 Min.

## Perform the following:

1. Log in to GitHub.com and fork the repository
2. Create a new workflow file
3. Execute the GitHub Actions workflow

**Expected deliverables:** A GitHub Actions CI/CD workflow to create automated Maven builds





**Thank You**