GitHub Actions

Agenda - Day 1

1

Git Basics

- Introduction to Git
- Pull Requests
- Cloning Project Source Code

2

Simple Workflow

- Overview of GitHub Actions workflows
- Creating a basic workflow

3

Workflow Automations

- Automating common tasks with GitHub Actions
- Exploring built-in actions and custom scripts

4

Event Triggers and Workflow Runners

- Understanding event triggers and when workflows run
- Configuring workflow runners

5

Filter Activity Types

- Filtering activities to run specific workflows

6

Contexts, Expressions, Variables, Functions

- Understanding and using contexts
- Working with expressions, variables, and functions in workflows

Execution Flow

- Managing workflow execution flow

Agenda - Day 2

7

Inputs and Outputs

- Managing inputs and outputs in workflow

8

Caching and Artifacts

- Using caching to speed up workflows
- Storing and retrieving artifacts

9

Releases and Matrix Builds

- Managing releases and tags
- Configuring matrix builds

10

Environments

- Setting up and using environments

11

Shell Scripts

- Writing and executing shell scripts

Agenda - Day 2

13

Fetch Data via API

- Fetching and using data from APIs in workflow

14

Custom Actions

- Composite Actions
- JavaScript (JS) Actions
- Docker Actions

15

Reusable Workflows

- Creating reusable workflows to streamline processes

16

Concurrency and Workflow Security

- Managing concurrent jobs and workflows
- Implementing security best practices

17

Docker

- Introduction to Docker and containerization
- Building Docker images for CI/CD workflows
- Integrating Docker with GitHub Actions

18

CI/CD Pipeline

- Overview of CI/CD pipelines
- Setting up CI/CD workflows in GitHub Actions
- Automating the CI/CD process using GitHub Actions

Agenda - Day 3

16

Concurrency and Workflow Security

- Managing concurrent jobs and workflows
- Implementing security best practices

17

Docker

- Introduction to Docker and containerization
- Building Docker images for CI/CD workflows
- Integrating Docker with GitHub Actions

18

CI/CD Pipeline

- Overview of CI/CD pipelines
- Setting up CI/CD workflows in GitHub Actions
- Automating the CI/CD process using GitHub Actions

19

Jenkins Migration

- Overview of migrating from Jenkins to GitHub Actions
- Mapping Jenkins jobs to GitHub Actions workflows

20

Monitoring and Observability

- Setting up monitoring for workflows
- Using logs and metrics to track workflow performance

21

Security and Governance

- Managing permissions and access control in GitHub Actions
- Implementing governance policies for CI/CD pipelines

Agenda - Day 3

22

Introduction to GitHub Copilot

- Overview of GitHub Copilot

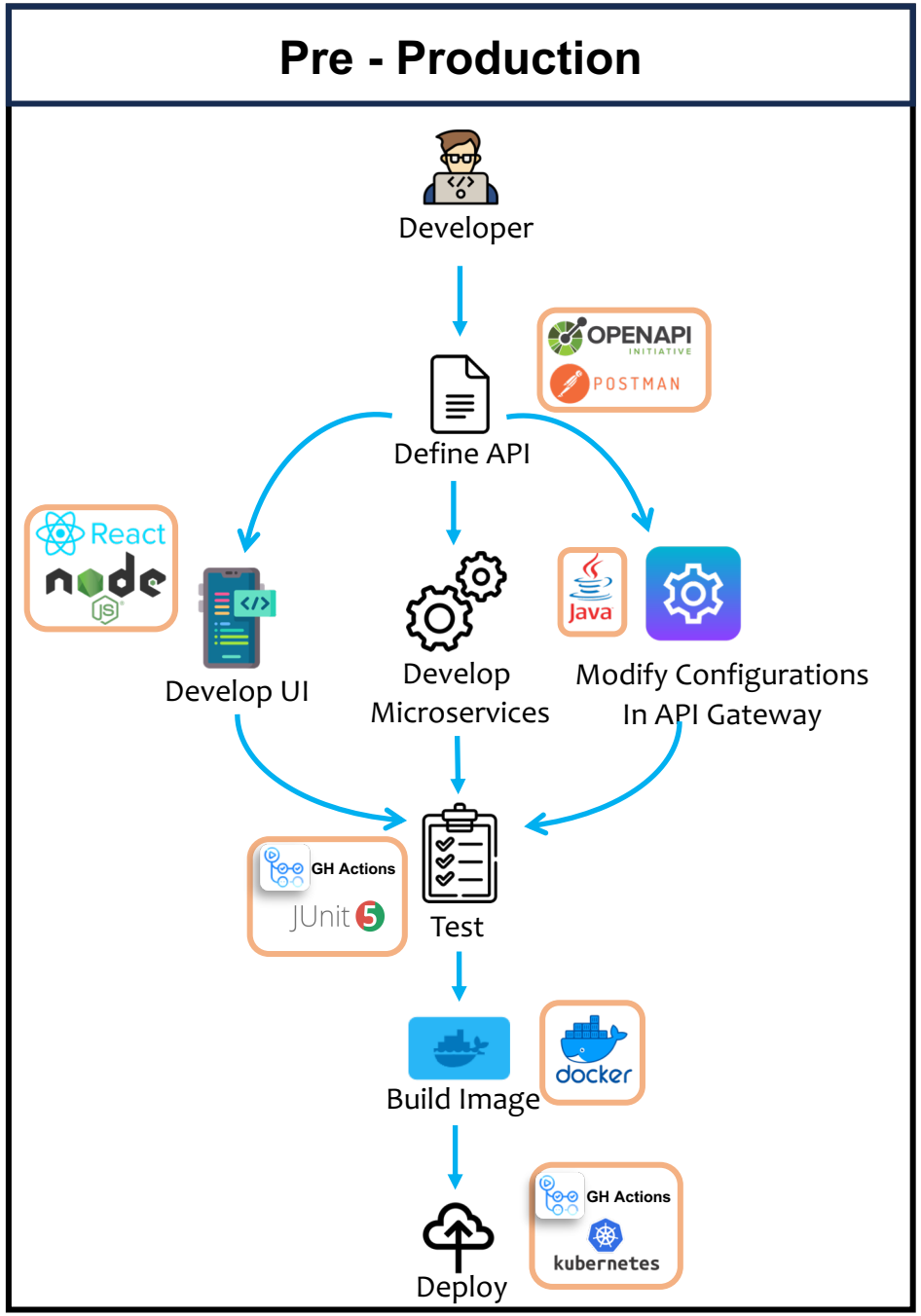
23

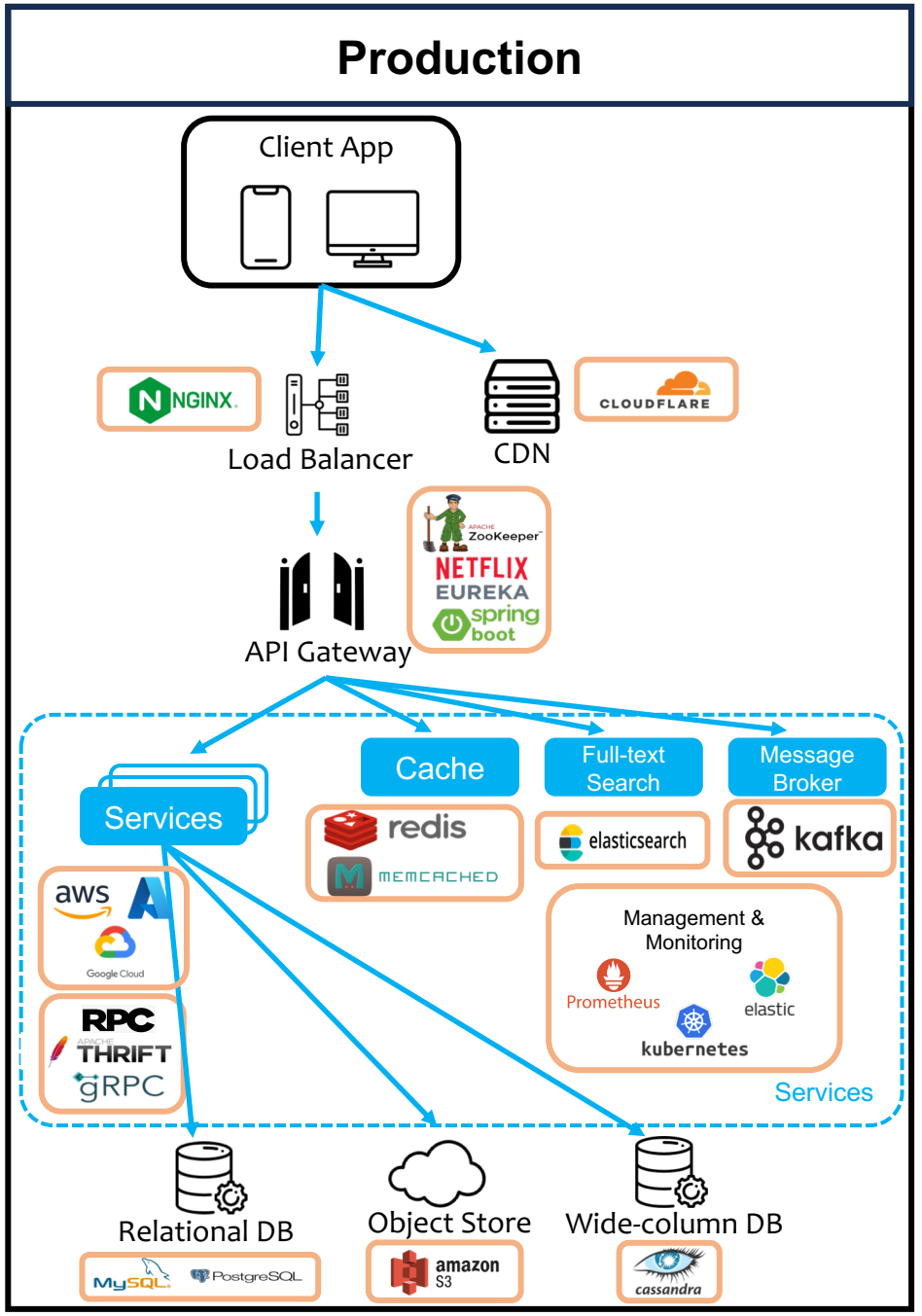
CI/CD Workflow for Android:

- Setting up GitHub Secrets
- Creating the GitHub Actions workflow file for APK build and upload
- Configuring Google Play Service Account and JSON Key
- Running the workflow to build and publish APK to Google Play Store

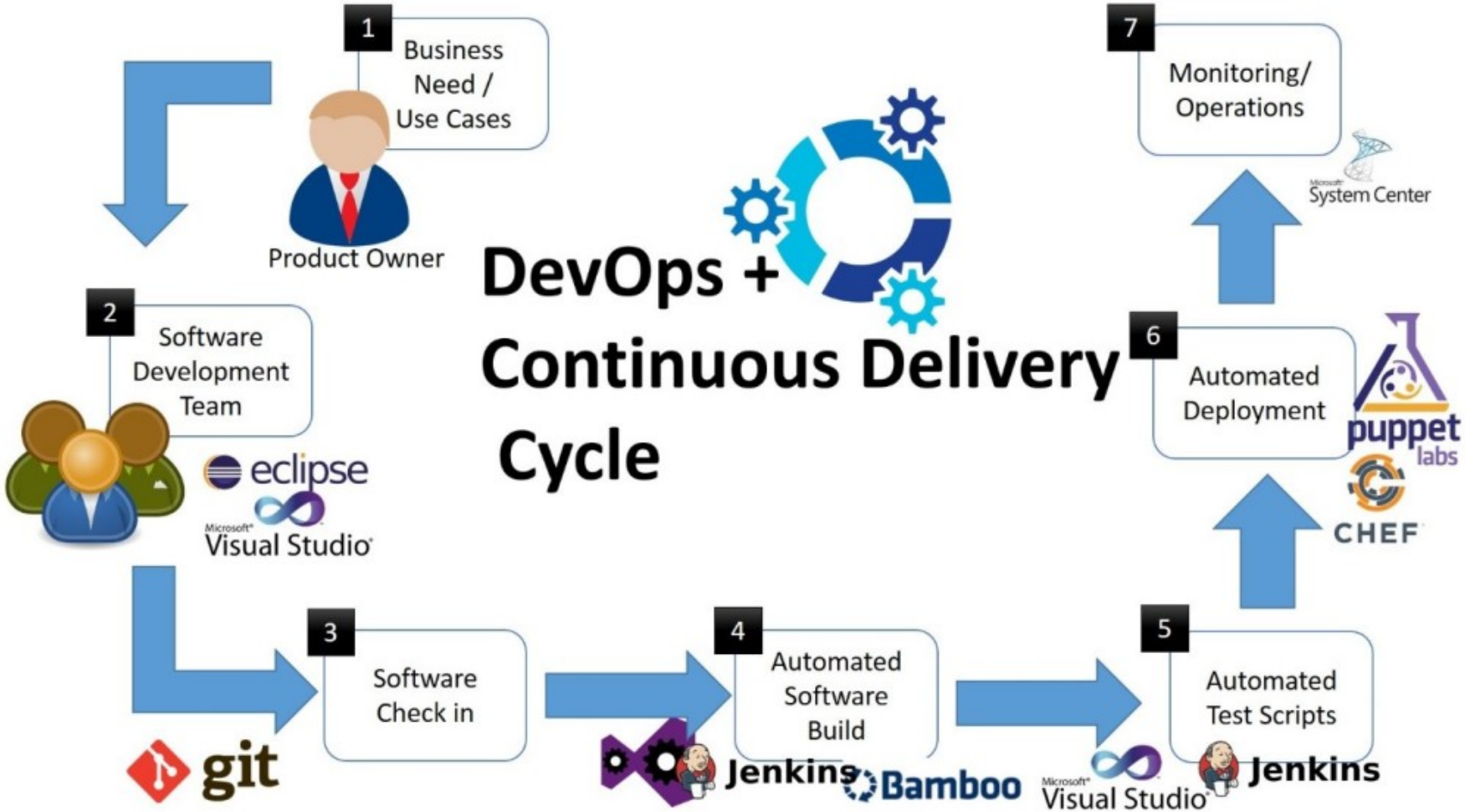
GitHub Introduction

DevOps CI/CD Implementation

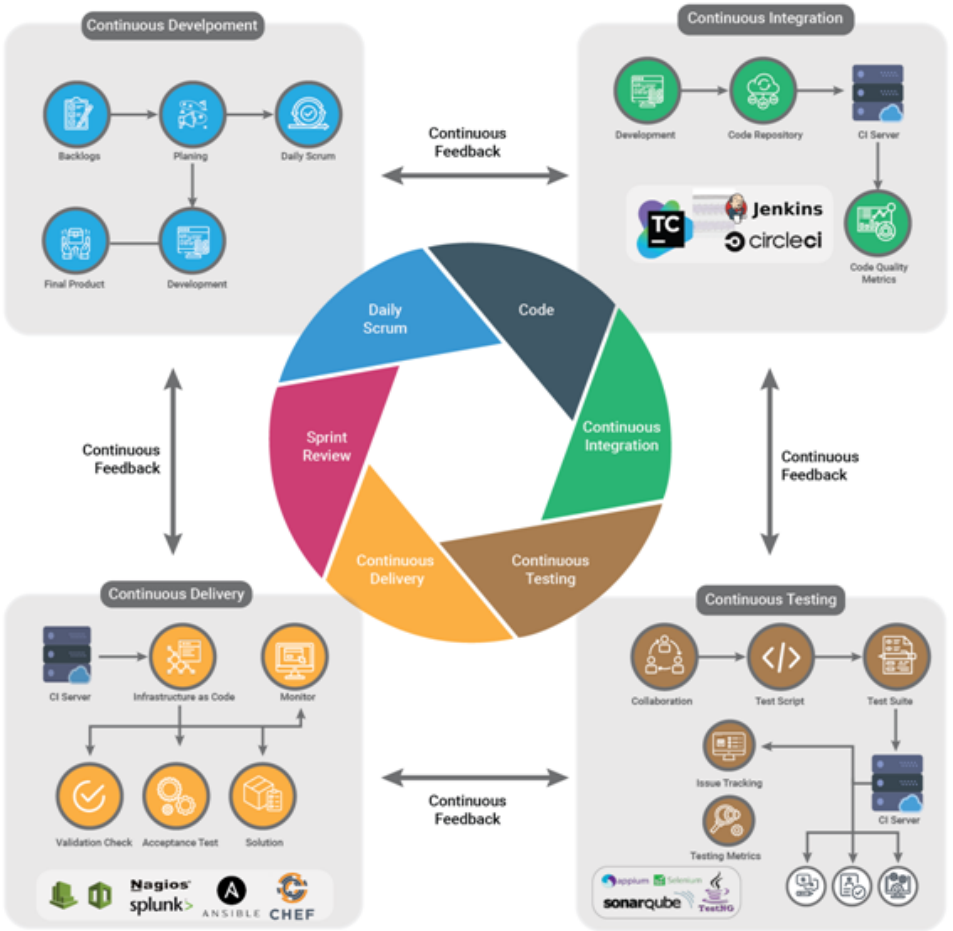

Pre - Production
Developer
OPENAPI
INITIATIVE
POSTMAN
Define API
React
node
Develop UI
Develop
Microservices
Java
Modify Configurations
In API Gateway
GH Actions
JUnit 5
Test
Build Image
docker
Deploy
GH Actions
kubernetes


Production
Client App
NGINX
Load Balancer
CDN
CLOUDFLARE
API Gateway
ZooKeeper
NETFLIX
EUREKA
spring
boot
Services
Cache
Full-text
Search
Message
Broker
redis
MEMCACHED
elasticsearch
kafka
aws
Google Cloud
RPC
THRIFT
gRPC
Management &
Monitoring
Prometheus
elastic
kubernetes
Services
Relational DB
MySQL
PostgreSQL
Object Store
amazon
S3
Wide-column DB
cassandra

DevOps Workflow



DevOps Workflow



GitHub Actions

What is GitHub Actions

- GitHub Actions will support continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline.

- You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production.

- GitHub Actions goes beyond just DevOps and lets you run workflows when other events happen in your repository.

What are GitHub Actions?

- Packaged Pipeline/Workflows
- Automates common build, test, and deployment actions
- Highly configurable and extendable
- Supports an open marketplace of free and paid tools
- Includes a rich user interface system native in GitHub

Common DevOps Tasks

- Add new contributors
- Close stale issues & PRs
- Pull request
- Organizational tasks: Issue created by users
- Label the issue { minor, major, reproducible, priority etc. }
- Fix the issue and raise a PR
- Verify PR and merge the code to master
- New release with a version number
- Build pipeline to test, and build code

Basic Building Blocks GH

- Workflow
- Event Triggers
- Workflow Runners
- Using Actions
- Filter Activity Types
- Contexts
- Expressions
- Variables
- Function
- Execution Flow
- Inputs
- Outputs
- Caching
- Artifacts
- Releases
- Matrices

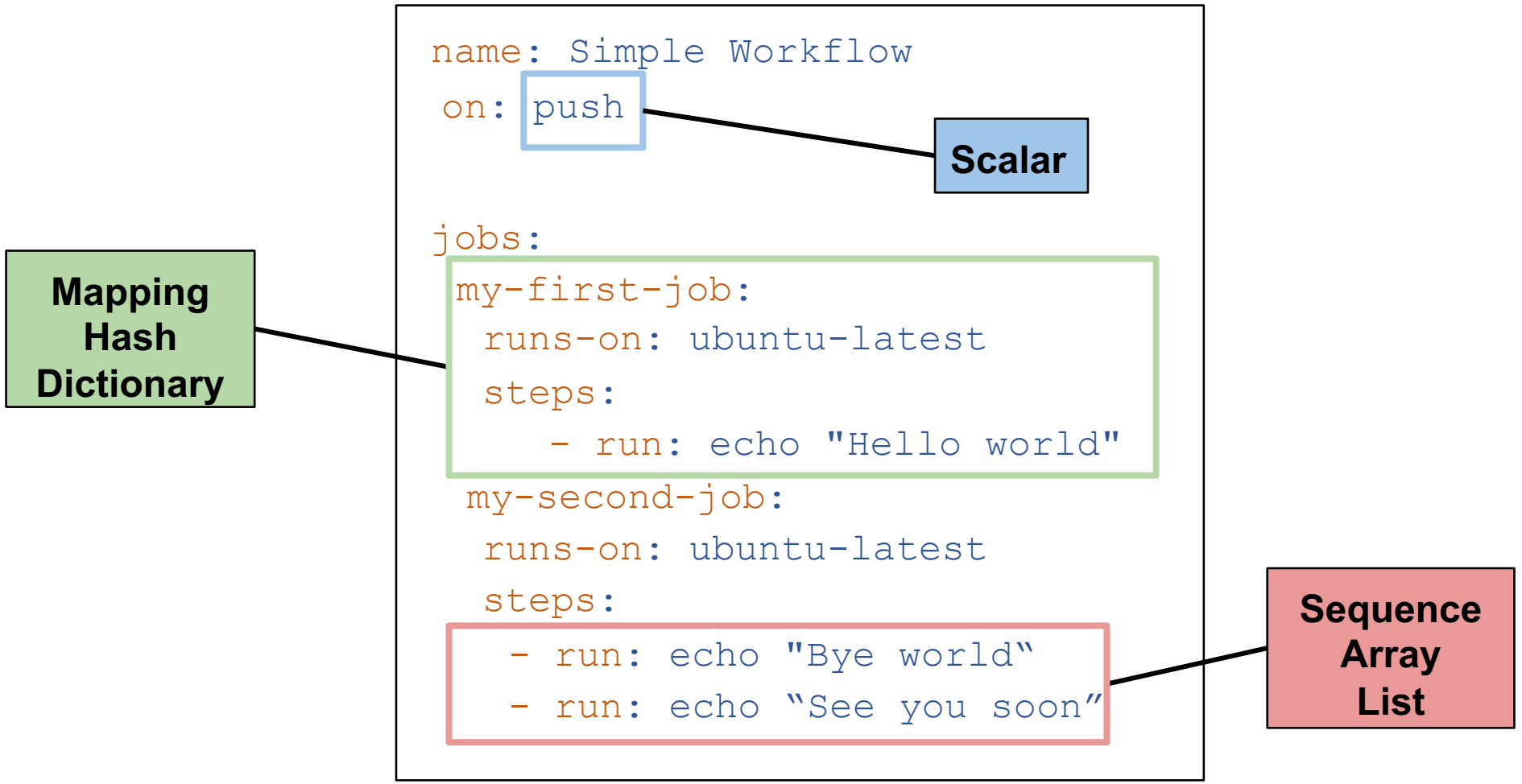
Basic Building Blocks GH

- Environments
- Custom Actions
- Reusable Workflows
- Concurrency
- Workflow Security
- Docker
- CI/CD Pipeline
- Jenkins Migration
- Monitoring and Observability
- Security and Governance
- CI/CD Workflow for Android

Object Expression - YAML

- Files or other representations of Kubernetes Objects are generally represented in YAML.
- A “Human Friendly” data serialization standard.
- Uses white space (specifically spaces) alignment to denote ownership.
- Three basic data types:
  - mappings - hash or dictionary,
  - sequences - array or list
  - scalars - string, number, boolean etc

Object Expression - YAML



YAML vs JSON

```yaml
name: Simple Workflow
on: push


jobs:
 my-first-job:
  runs-on: ubuntu-latest
  steps:
     - run: echo "Hello world"
 my-second-job:
  runs-on: ubuntu-latest
  steps:
   - run: echo "Bye world"
   - run: echo "See you soon"
```

```json
{
  "name": "Simple Workflow",
  "on": "push",
  "jobs": {
    "my-first-job": {
      "runs-on": "ubuntu-latest",
      "steps": [
        {
          "run": "echo \"Hello
World\""
        }
      ]
    },
    "my-second-job": {
      "runs-on": "ubuntu-latest",
      "steps": [
        {
          "run": "echo \"Bye World\""
        }
      ]
    }
  }
}
```

Workflows

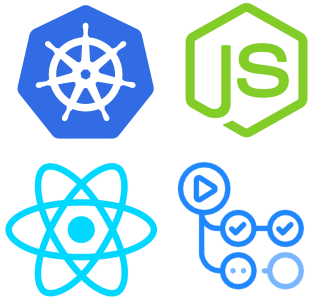
- A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked in to your repository and will run when triggered by an event in your repository, or they can be triggered manually, or at a defined schedule.
- In other platform this is called as Pipelines.

What are workflows?

GitHub

Platform for open-source projects

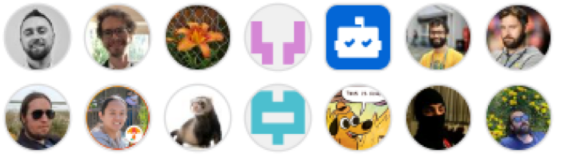
- Publicly available to use and contribute to the project



Used by 420k

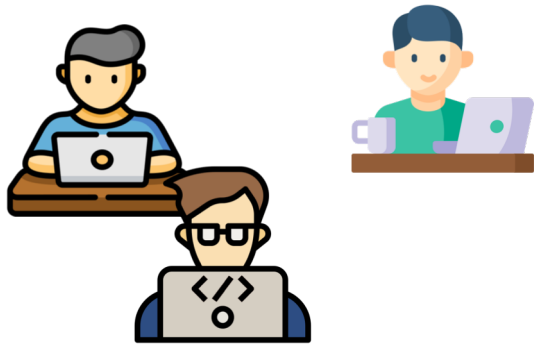
+ 419,650

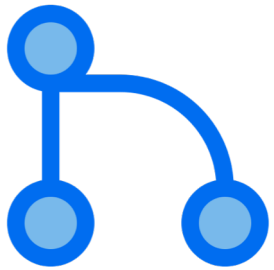
Contributors 435



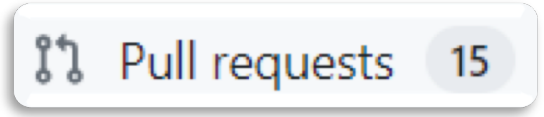
+ 421 contributors

What are workflows?





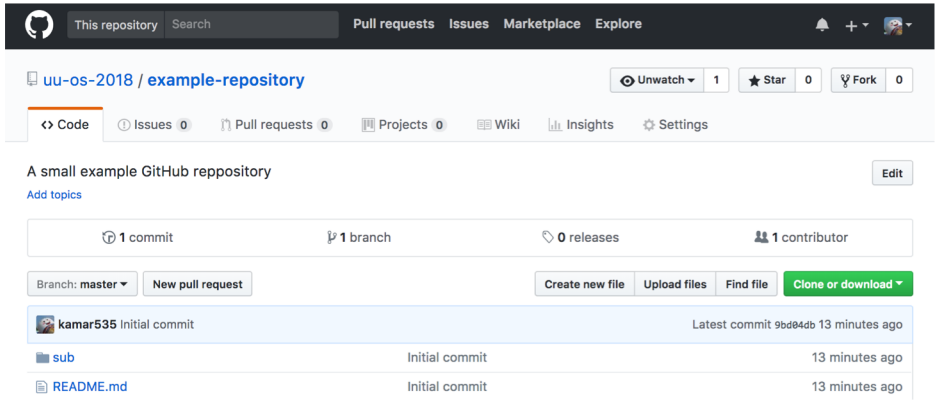
- add new contributors
- pull requests are created
- ....



Organizational Task

What are workflows?

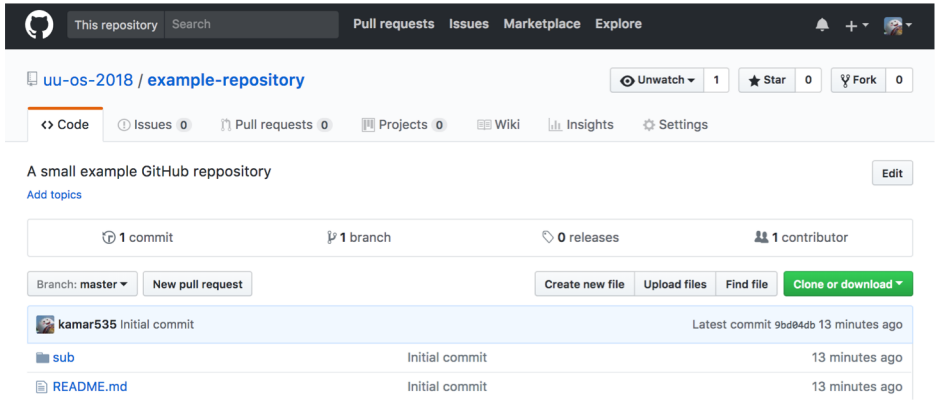
Contributors



What are workflows?

Contributors

Users

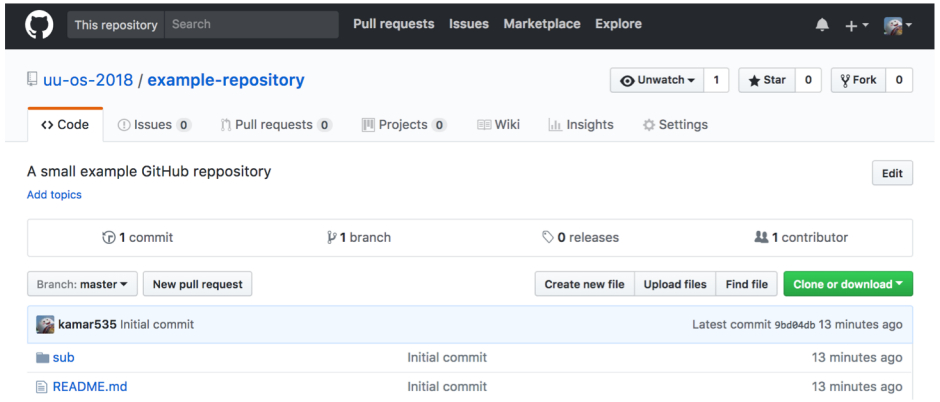


What are workflows?

Organizational Task

Contributors

Users



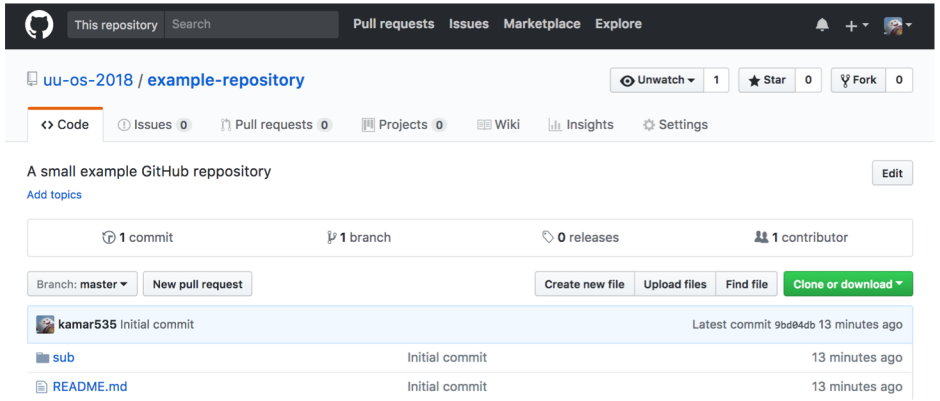
What are workflows?

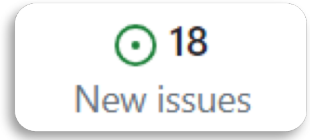
Organizational Task

Contributors

Users

New Issue



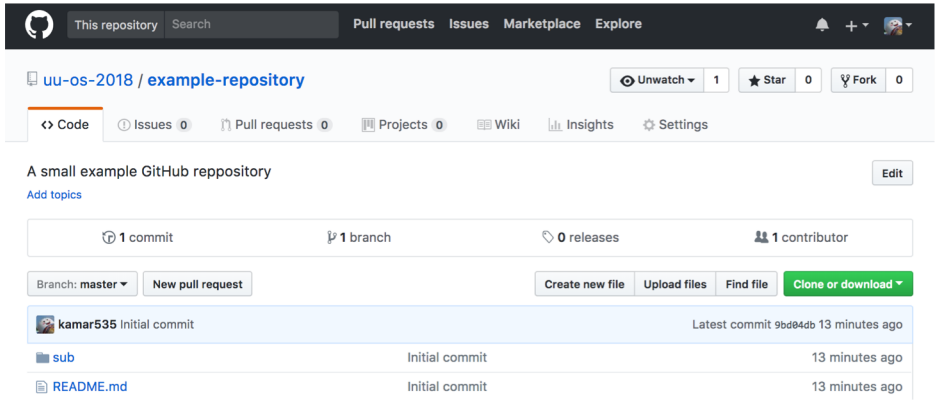


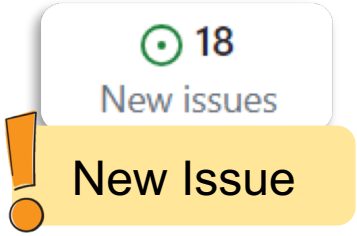
What are workflows?

Organizational Task

Contributors

Users

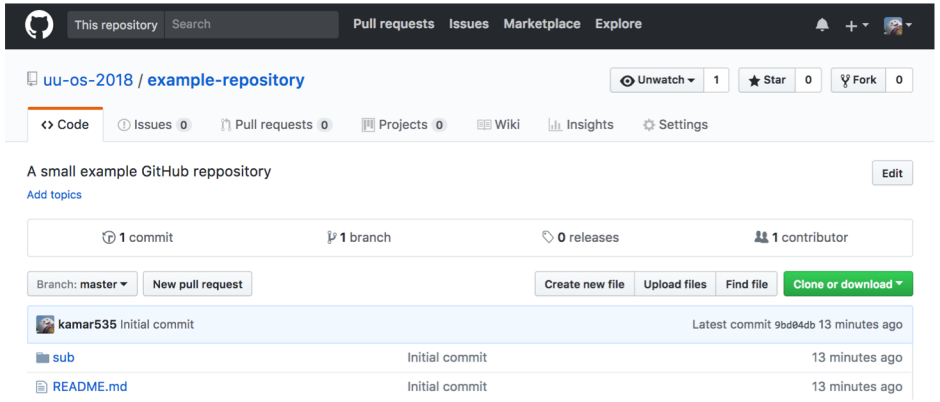


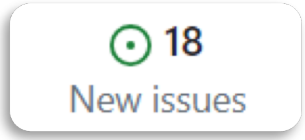


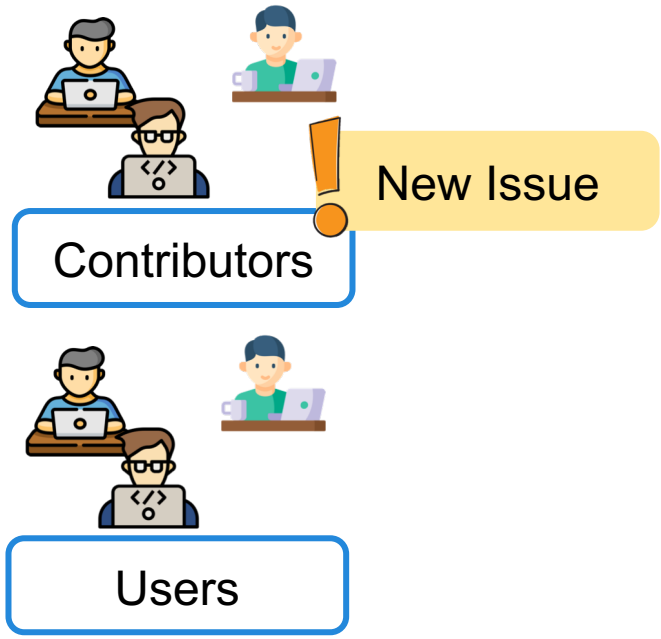
- is it major / minor?
- is it reproduceable?
- assign to a contributor

What are workflows?

Organizational Task



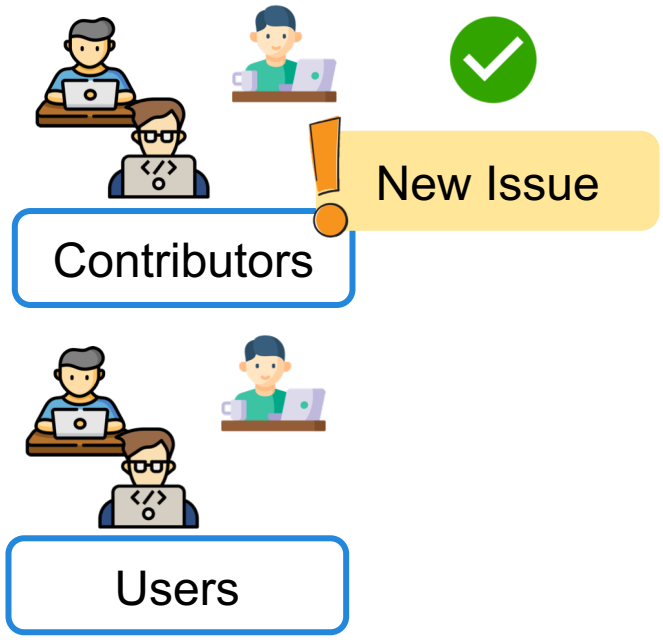


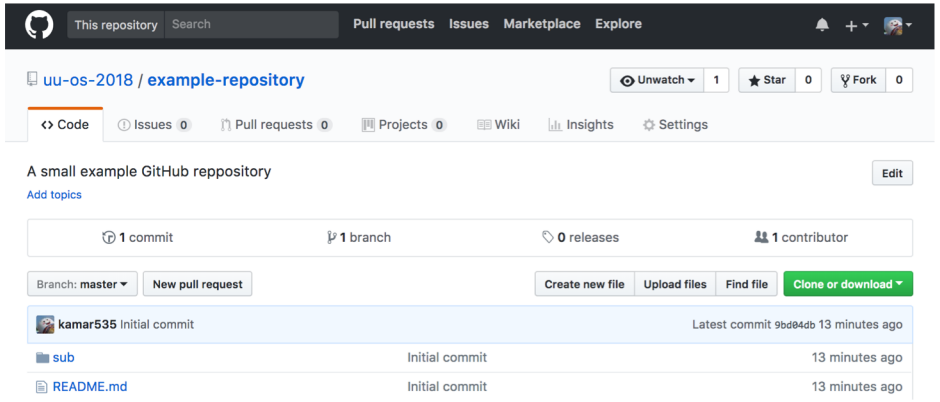


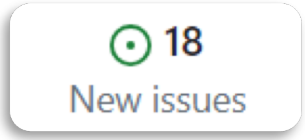
- is it major / minor?
- is it reproducable?
- assign to a contributer

What are workflows?

Organizational Task

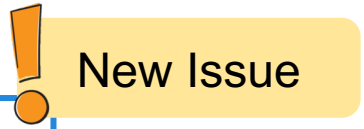






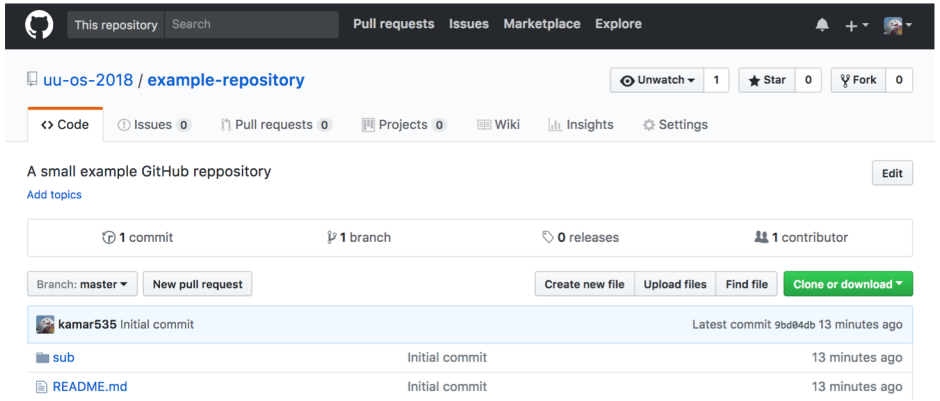
What are workflows?

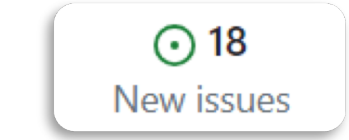
Organizational Task

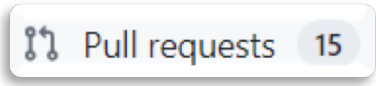


Contributors

Users

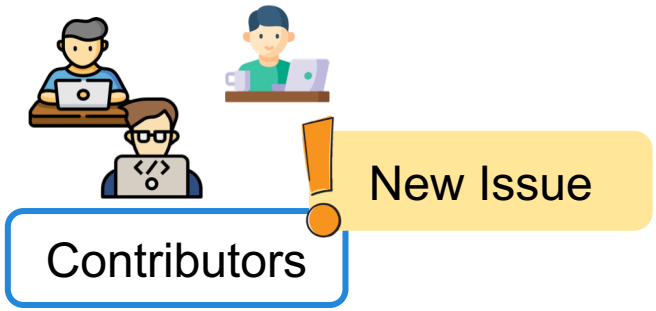


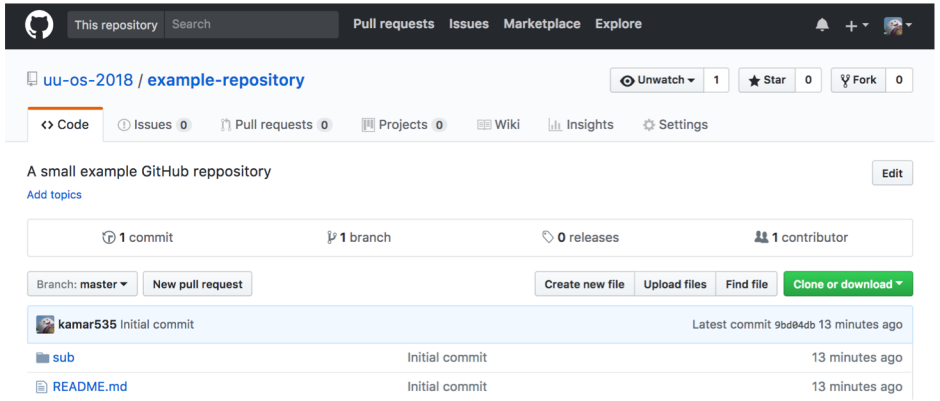


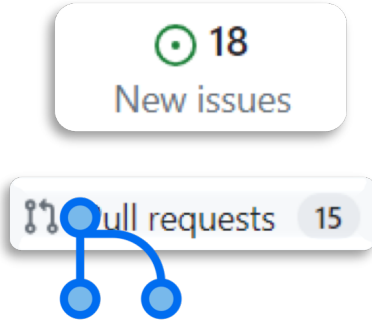


What are workflows?

Organizational Task







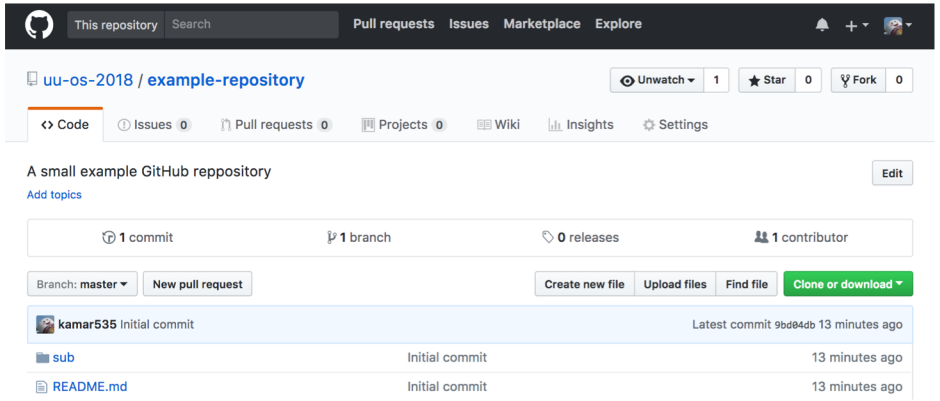
- review pull request
- is the bug fixed?
- merge to master branch

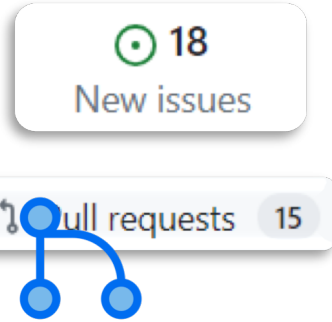
What are workflows?

Organizational Task

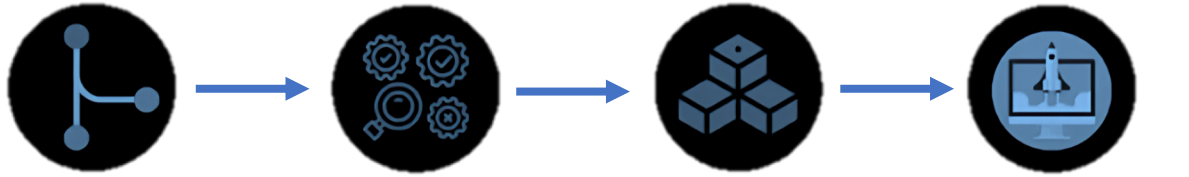
Contributors

- Prepare release notes
- Update version number





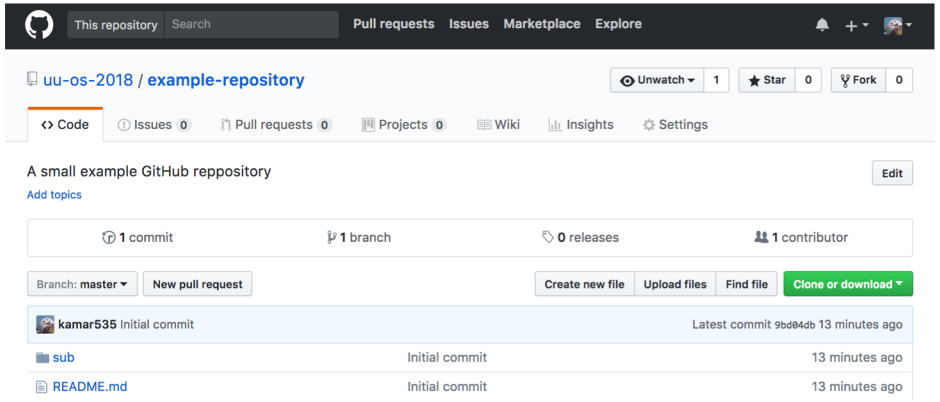
CI/CD Pipeline

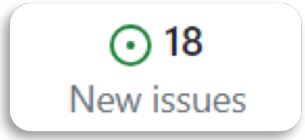


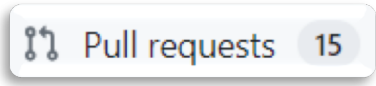
Merged Code → Test → Build → Deployment

What are workflows?

Organizational Task







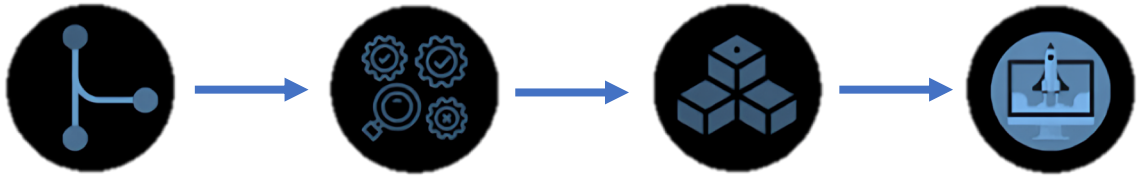
Contributors

- is it major / minor?
- is it reproducable?
- assign to a contributer

- review pull request
- is the bug fixed?
- merge to master branch

- Prepare release notes
- Update version number

WORKFLOWS



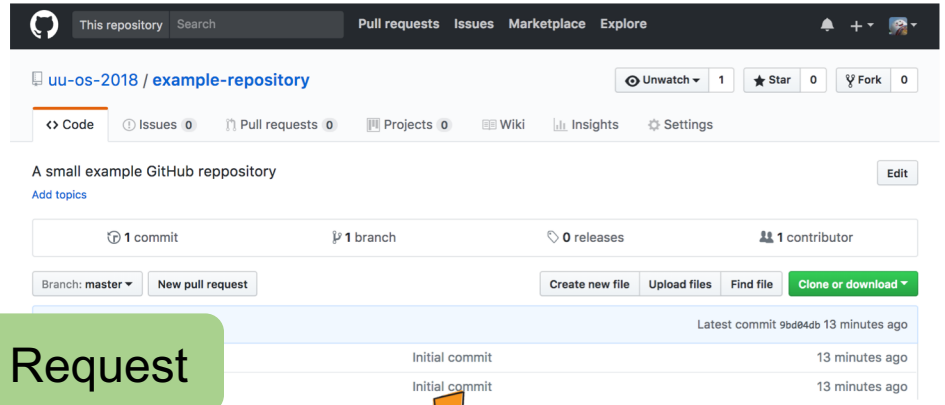
Merged Code → Test → Build → Deployment

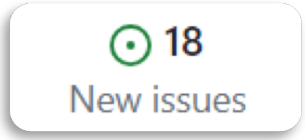
GitHub Actions

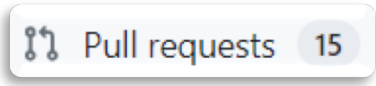
What are workflows?

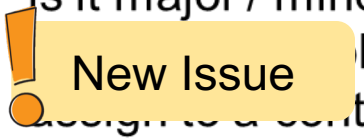
Organizational Task

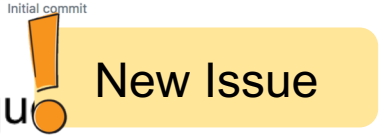
Contributors

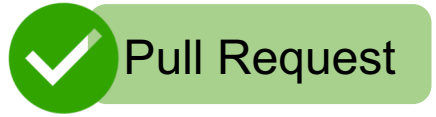


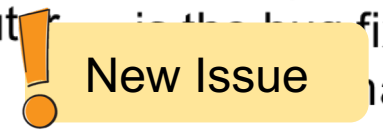












- is it major / minor?
- le?
- review pull requ
- tribut
- fixed?
- master br
- release notes
- Update version number

WORKFLOWS

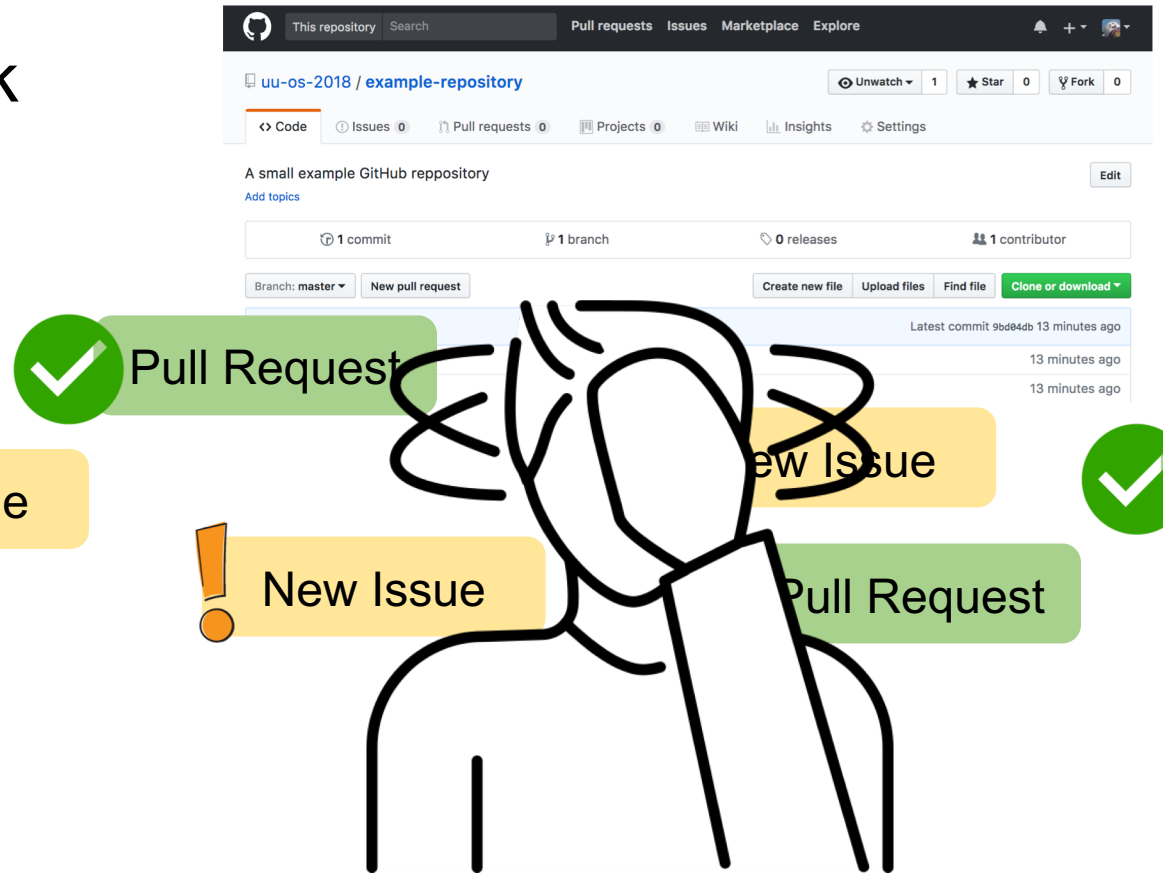
Merged Code | Test | Build | Deployment

GitHub Actions

What are workflows?

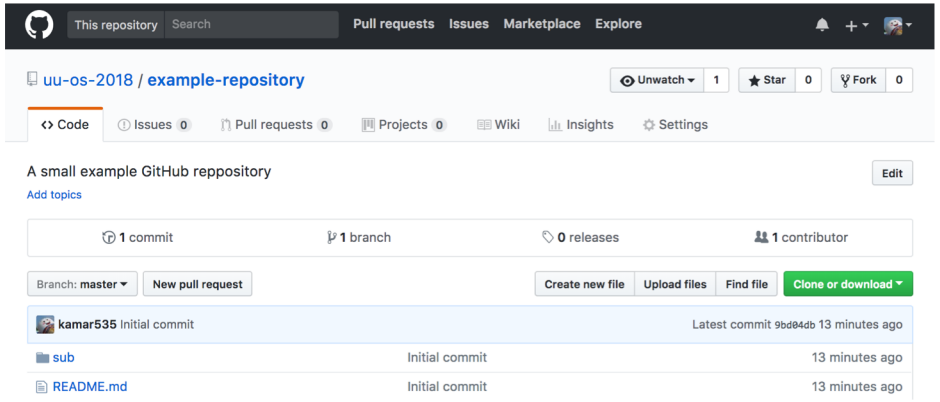
Organizational Task

Contributors



What are workflows?

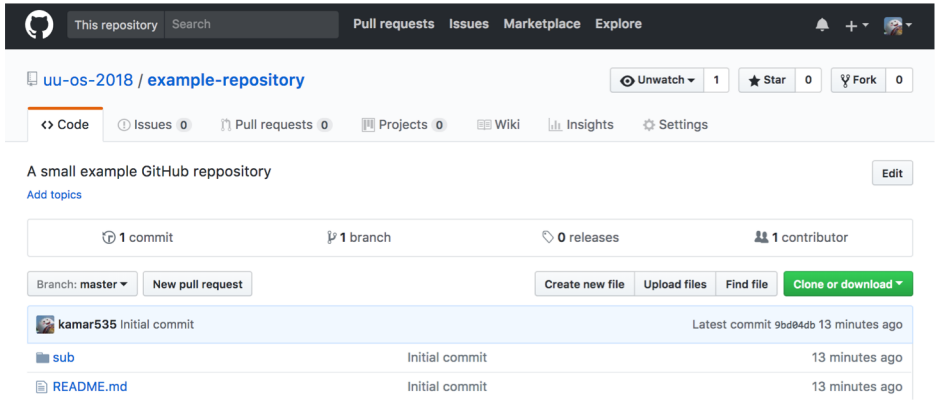
Organizational Task



Automate as much as possible!

What are workflows?

Organizational Task

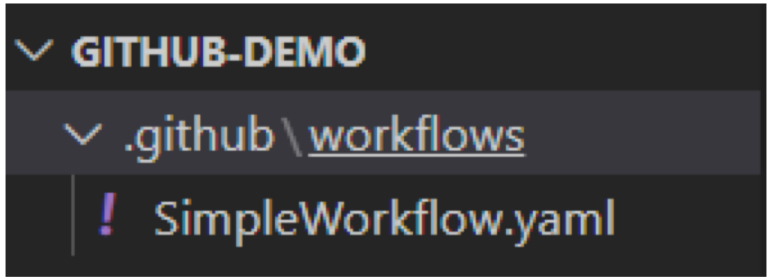


Automate as much as possible!

Creating a Workflow File

- Create a ***.github/workflows*** directory in your repository on GitHub if this directory does not already exist.

- In the .github/workflows directory, create a file with the ***.yml*** or ***.yaml*** extension.

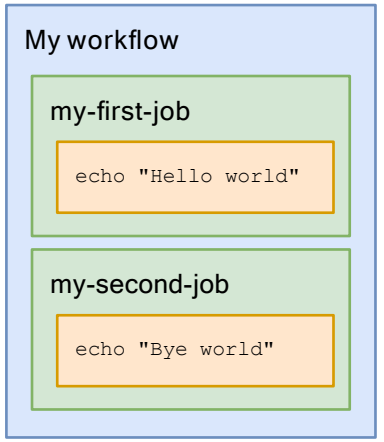


Workflows, Jobs & Steps

- **Workflows:**
  - Are defined at the repository level
  - Define which triggers actually start the workflow
  - Are composed of one or more jobs

  - **Jobs:**
    - Are defined at the workflow level
    - Define in which execution environment they are run
    - Are composed of one or more steps
    - Run in parallel by default

    - **Steps:**
      - Are defined at the job level
      - Define the actual script or GitHub Action that will be executed
      - Run sequentially by default



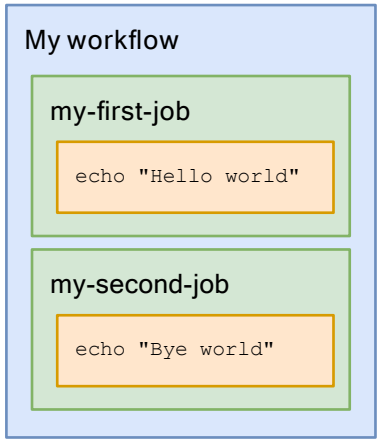

```yaml
name: Simple Workflow
on: push

jobs:
  my-first-job:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Hello world"
  my-second-job:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Bye world"
```

Workflows, Jobs & Steps

- **Workflows:**
  - Are defined at the repository level
  - Define which triggers actually start the workflow
  - Are composed of one or more jobs

  - **Jobs:**
    - Are defined at the workflow level
    - Define in which execution environment they are run
    - Are composed of one or more steps
    - Run in parallel by default

    - **Steps:**
      - Are defined at the job level
      - Define the actual script or GitHub Action that will be executed
      - Run sequentially by default

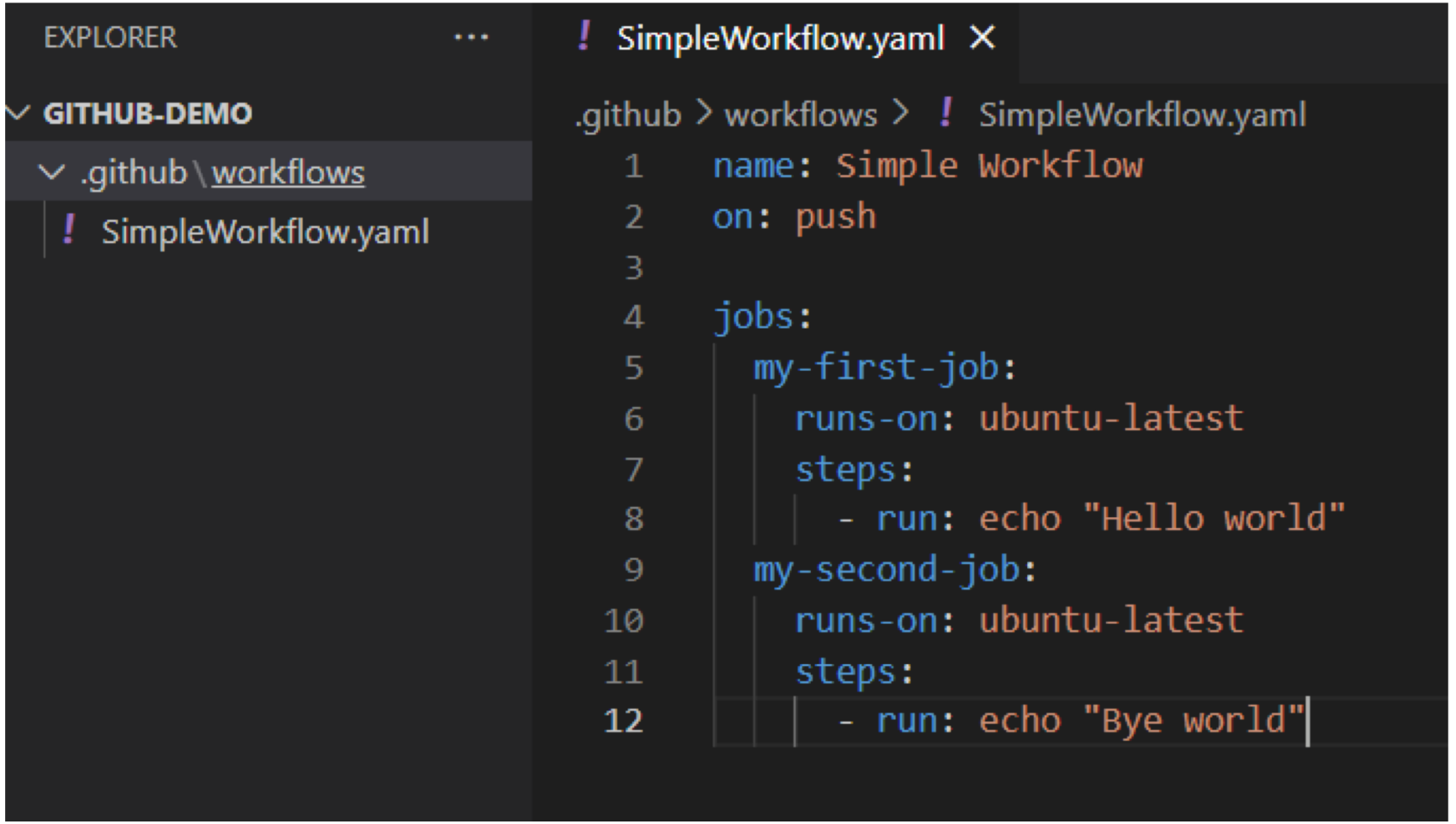



```yaml
name: Simple Workflow
on: push

jobs:
  my-first-job:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Hello world"
  my-second-job:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Bye world"
```

Triggers:

You can configure your workflows to run when specific activity on GitHub happens, at a scheduled time, or when an event outside of GitHub occurs.

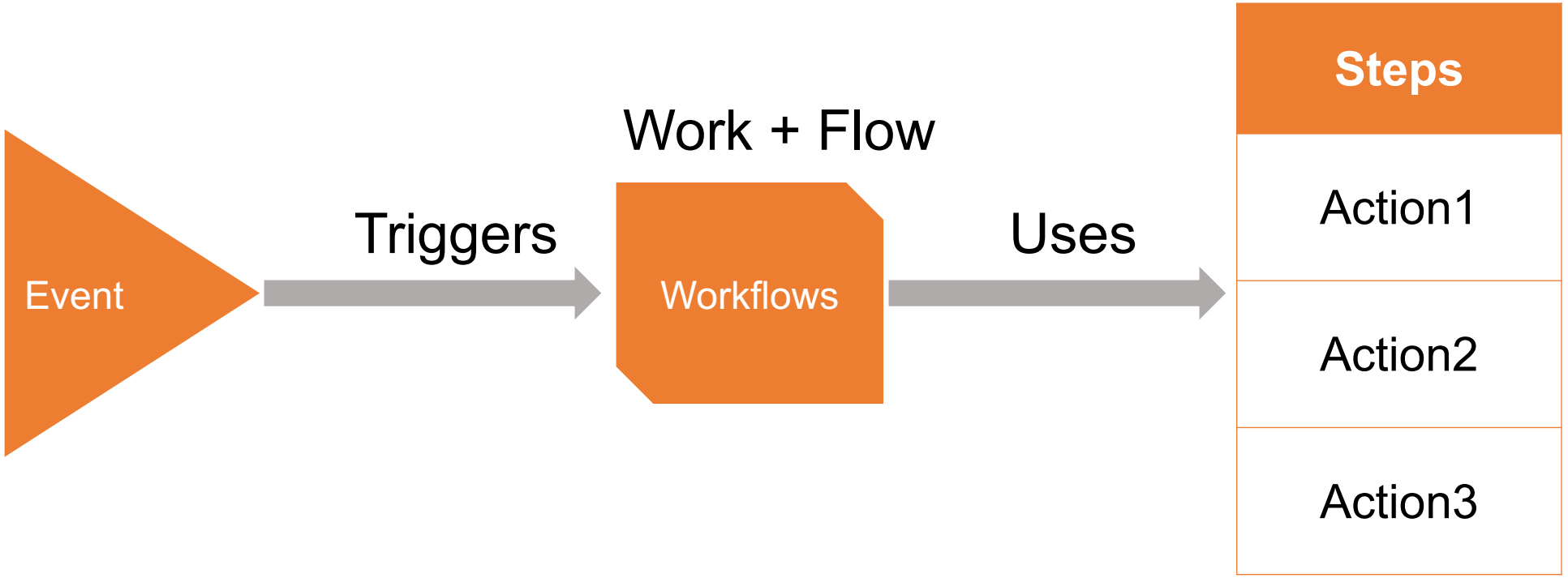
Simple Workflow



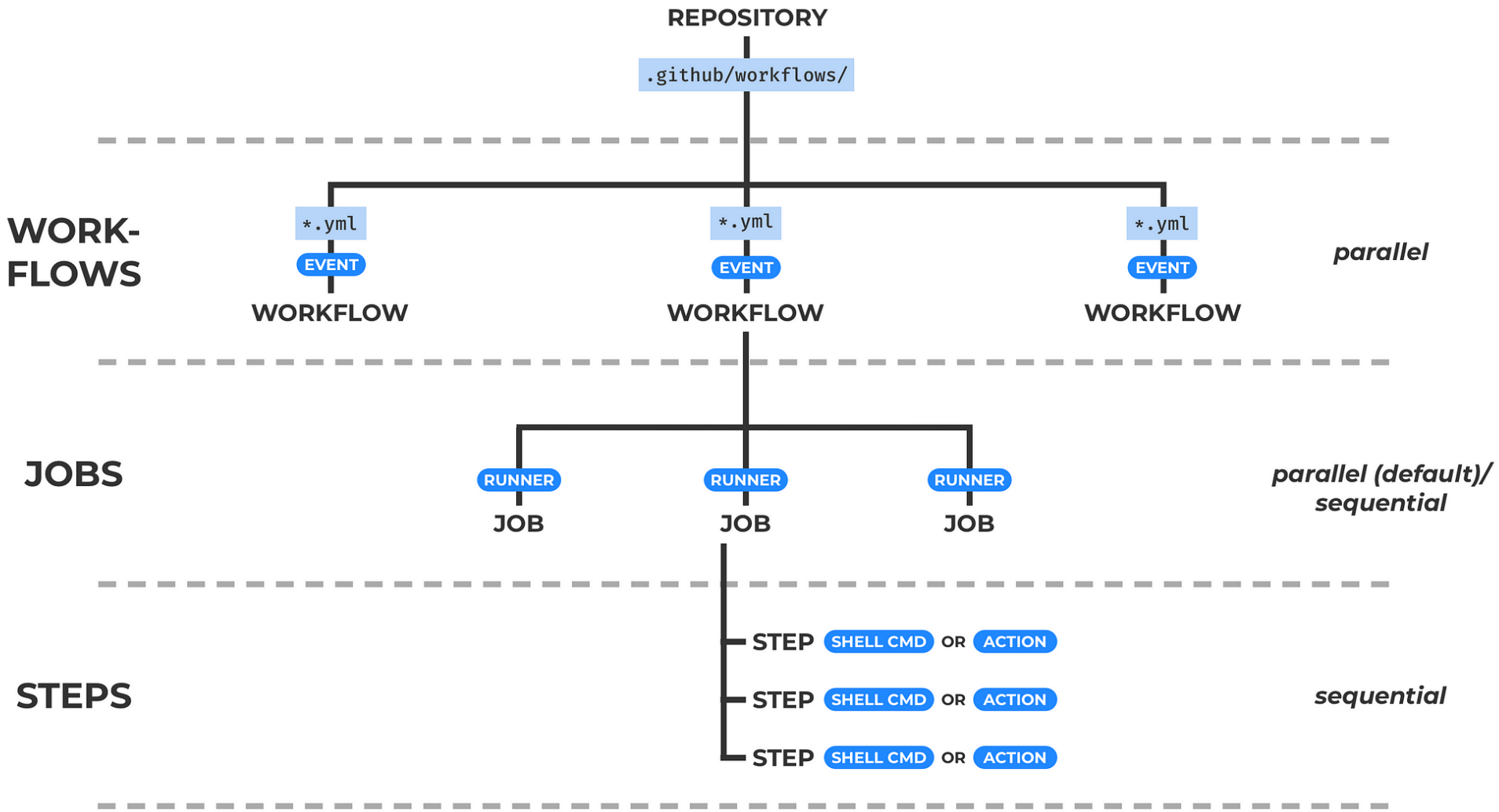
Test Workflow

- Since our workflow's trigger is a push event, push a commit on any repository branch and attach Triggering Push as the commit message.
- Go to the repository's page on GitHub and click the **Actions tab** under the repository's name.
- At the top, the commit message for the push that triggered the workflow will be visible.
- Clicking on the job will show the output from each step in the workflow.

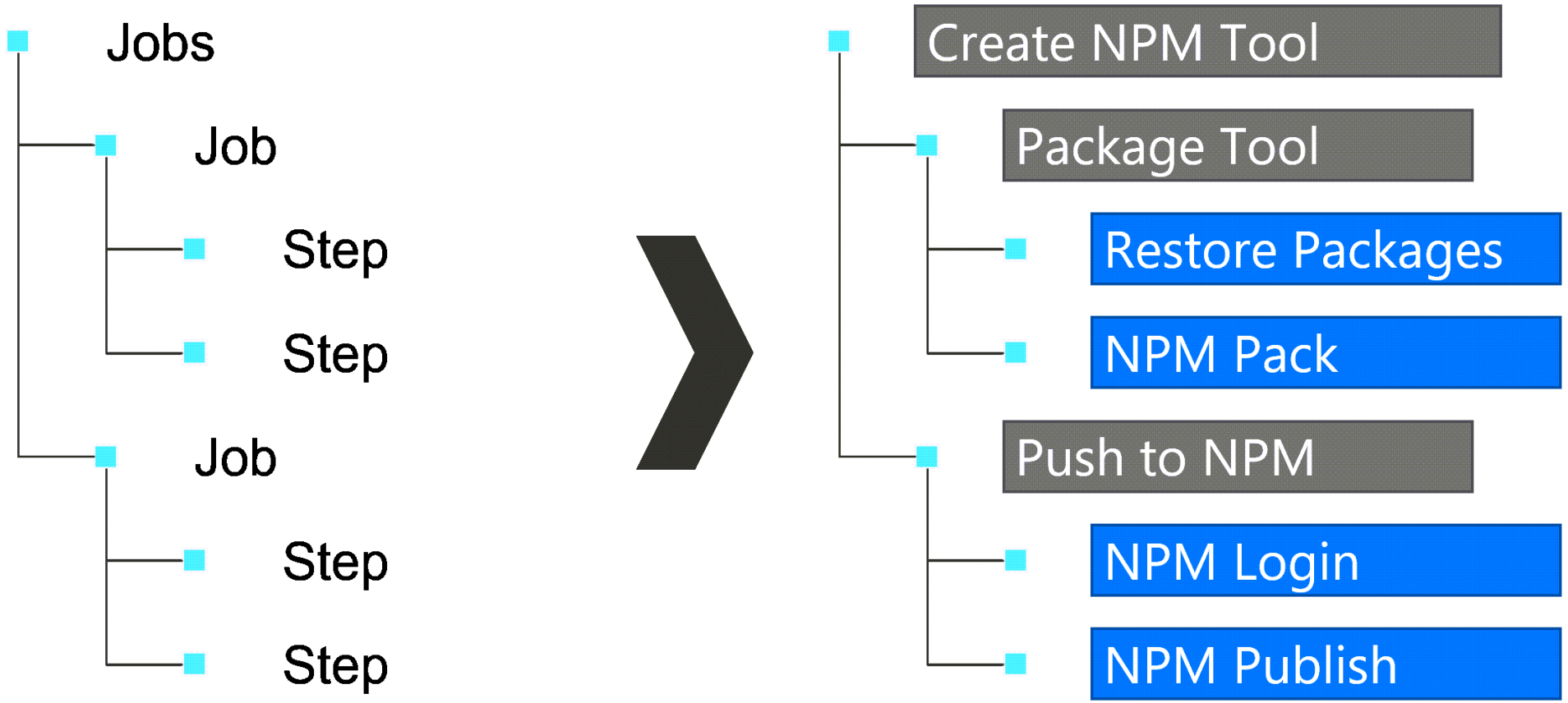
GitHub Action Workflow



GitHub Actions



Example of a workflow



Workflows

- Types of Workflows
  - Automation
  - Pages
  - Continuous integration
  - Deployment
  - Security

Workflows - Automation

- Labeler
- Stale
- Greetings
- Manual workflow

Automation - Labeler

- Automatically label new pull requests based on the paths of files being changed or the branch name.

Automation - Labeler

Label.yml

```yaml
name: Labeler
on: [pull_request_target]

jobs:
  label:

    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: write

    steps:
    - uses: actions/labeler@v4
      with:
        repo-token: "${{
secrets.GITHUB_TOKEN }}"
```

labeler.yml

```yaml
# Add 'Documentation' label to any file
changes within '01-SimpleWorkflow' folder

Documentation:
- changed-files:
  - any-glob-to-any-file:
    - Docs/*
```

Automation - Stale

- The GitHub Stale Action is a community-maintained GitHub Action that helps automate the process of identifying and closing stale issues and pull requests in a GitHub repository.

Automation - Stale

```yaml
name: Mark stale issues and pull requests

on:
  schedule:
  - cron: '18 17 * * *'

jobs:
  stale:

    runs-on: ubuntu-latest
    permissions:
      issues: write
      pull-requests: write

    steps:
    - uses: actions/stale@v5
      with:
        repo-token: ${{ secrets.GITHUB_TOKEN }}
        stale-issue-message: 'Stale issue message'
        stale-pr-message: 'Stale pull request message'
        stale-issue-label: 'no-issue-activity'
        stale-pr-label: 'no-pr-activity'
```

Automation - Greetings

- The "greetings" GitHub Action automatically comments on pull requests from first-time contributors.

Automation - Greetings

```yaml
name: Greetings

on: [pull_request_target, issues]

jobs:
  greeting:
    runs-on: ubuntu-latest
    permissions:
      issues: write
      pull-requests: write
    steps:
    - uses: actions/first-interaction@v1
      with:
        repo-token: ${{ secrets.GITHUB_TOKEN }}
        issue-message: "Message that will be displayed on users' first issue"
        pr-message: "Message that will be displayed on users' first pull request"
```

Automation - Manual workflow

- GitHub Actions has a feature called Manual Workflow Approval that pauses a workflow and requires manual approval before continuing.

Automation - Manual workflow

```yaml
name: Manual workflow

on:
  workflow_dispatch:
inputs:
      name:
        description: 'Person to greet'
        default: 'World'
        required: true
       type: string

jobs:
  greet:
    runs-on: ubuntu-latest
    steps:
    - name: Send greeting
      run: echo "Hello ${{ inputs.name }}"
```

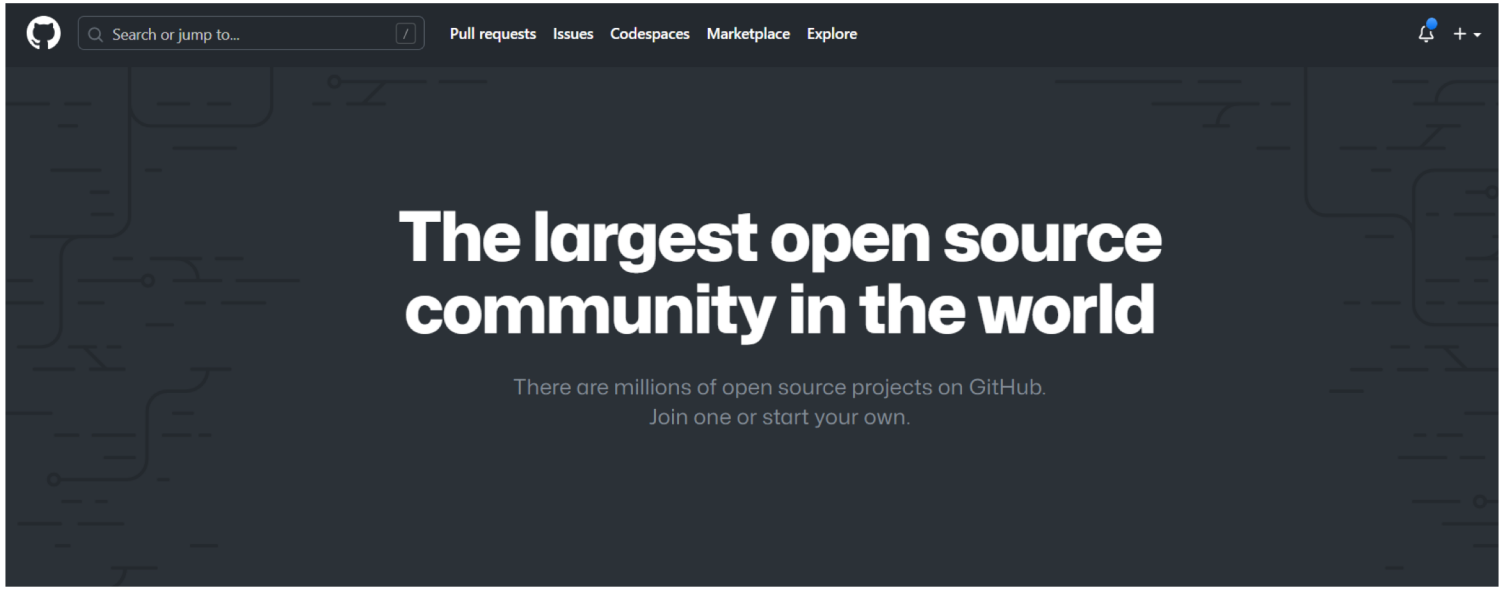
Workflows – Pages

- You can use GitHub Pages to host a website about yourself, your organization, or your project directly from a repository on GitHub.com.

What are those workflows?

GitHub

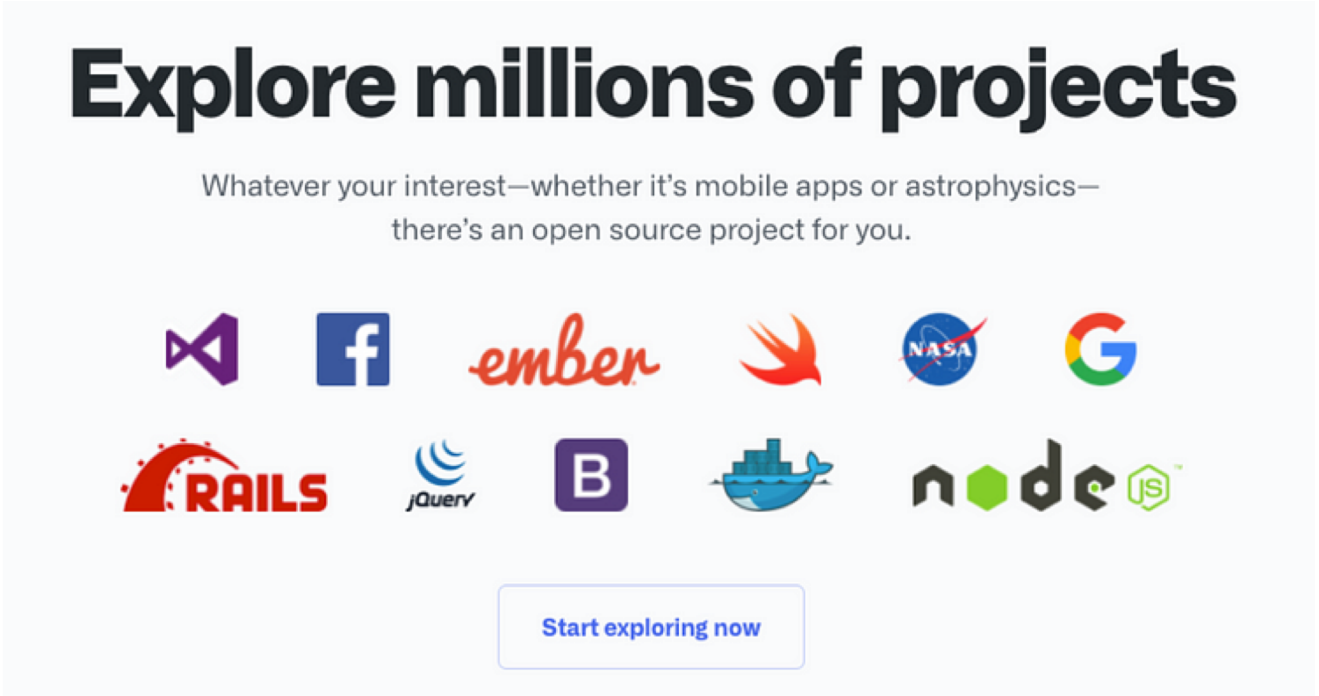
Platform for open-source projects



What are those workflows?

GitHub

Platform for open-source projects



Triggers

- You can configure your workflows to run when specific activity on GitHub happens, at a scheduled time, or when an event outside of GitHub occurs.

Triggering workflows in multiple ways

There are many ways we can trigger GitHub workflows:

Repository event	`push` Triggered when someone pushes to the repo	`issues` Triggered by a variety of events related to issues	`pull_request` Triggered by a variety of events related to PRs	`pull_request_review` Triggered by a variety of events related to PR reviews (submitting, editing, deleting)	`fork` Triggered when your repository is forked	...
Manual trigger	`Triggered via the UI` Triggered from the **Actions** tab in GitHub	`Triggered via an API call` Triggered via GitHub's REST API	`Triggered from another workflow` Triggered from within another workflow			
Schedule	`Runs as a cron job`					

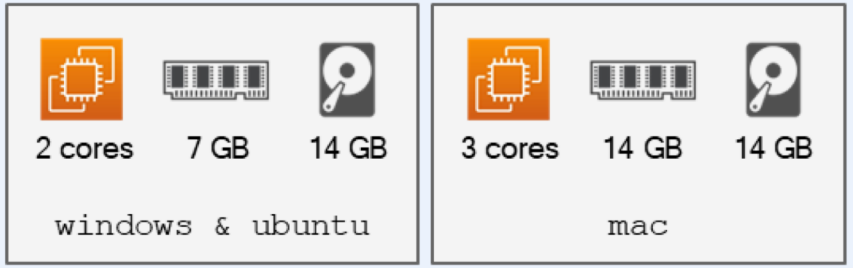
Triggers

```yaml
name: Workflow Events

on:
  # push:
  workflow_dispatch:
  # pull_request:
  # schedule:
  #   - cron: '*/5 0 * * *' # Run every 5 min

jobs:
  echo:
    runs-on: ubuntu-latest
    steps:
      - name: Show the trigger
        run: echo "I've been triggered by a(n) ${{ github.event_name }}
event."
```

Runners

Virtual servers that execute jobs from workflows

GitHub-hosted (standard)



- Managed service
- A VM is scoped to a job: steps share the VM, but jobs don't (by default, each job receives a clean VM instance)

Self-hosted

- Run workflows on (almost) any infrastructure of your choice
- Full control over the VM infrastructure
- It's not managed, meaning we need to take care of OS patching, software updates, among other ops tasks
- Can be added at the repository, organization, or enterprise level
- Jobs do not necessarily have to run on clean instances

Pro-tip

Keep the VM resources in mind, especially when running commands that rely on parallel execution (for example, running parallel jest tests).

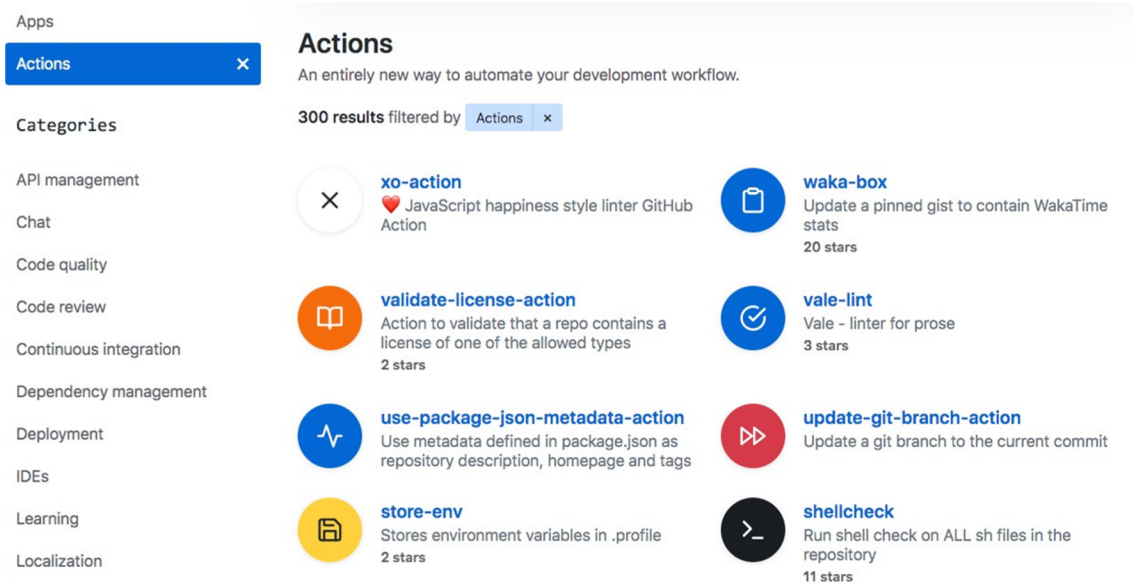
Warning

Do not use self-hosted runners in public repositories!

Runners

```yaml
name: Workflow Runners

on: workflow_dispatch

jobs:
  ubuntu-echo:
    runs-on: ubuntu-latest
    steps:
      - name: Show OS
        run: |
          echo "This job is running on an Ubuntu runner."
          echo "Runner OS: $RUNNER_OS"
  self-echo:
    runs-on: self-hosted
    steps:
      - name: Show OS
        run: |
          echo "This job is running on a Self-Hosted runner."
          echo "Runner OS: $RUNNER_OS"
```

GitHub Actions Marketplace



- The GitHub Actions Marketplace is a curated collection of pre-built actions that enable automation and streamline workflows within GitHub repositories, covering diverse tasks from testing and building to deploying and managing code.

- https://github.com/marketplace?type=actions

Actions

Custom applications to perform complex, frequently repeated tasks

```yaml
name: My Java package
on: push

jobs:
 java-release:
   runs-on: ubuntu-latest
   steps:
     - name: Set up JDK
       uses: actions/setup-java@v2
       with:
         java-version: '11'
         distribution: 'adopt'
         cache: maven
     - name: Run Tests
       run: mvn test
```

Avoids repetitive and extensive commands

Prevents code duplication and reduces the chances of mistakes

Can be configured via the `with` key-value pair

Enables great flexibility and reusability

Can be combined with other steps

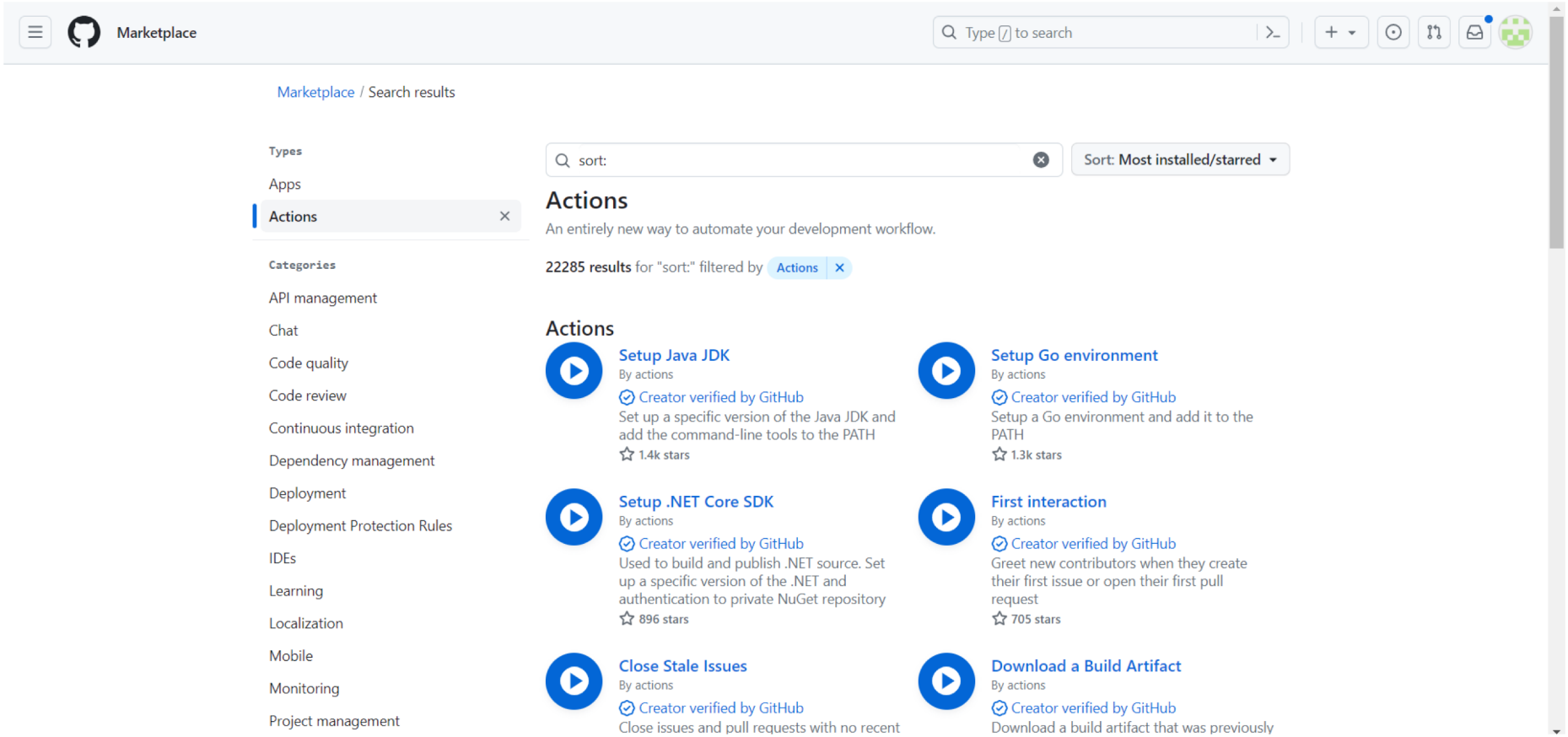
Can be used to encapsulate setup tasks before running other commands

We can create our own actions for public or private use

We are not restricted only to the actions available at the marketplace

Actions

- You can find actions in GitHub Marketplace



Actions

```yaml
name: Actions

on: workflow_dispatch

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v4

      - name: Setup Java
        uses: actions/setup-java@v4
        with:
          distribution: 'adopt'
          java-version: '11'
```

Event Filters

Specify under which conditions a specific event triggers our workflow

Most triggers can be configured to specify when they should run.

`push` event

`branches` Specifies which branches must match in order for the workflow to execute	`branches_ignore` Specifies which branches must not match in order for the workflow to execute
`tags`	`tags_ignore`
`paths`	`paths_ignore`
.	

```yaml
name: My package workflow
on:
  push:
    branches:
      - main
    - 'releases/**'
    paths-ignore:
      - 'docs/**'
```

If multiple filters are specified, all of them must be satisfied for the workflow to run!

Activity Types

Specify which types of certain triggers execute our workflow

Many triggers have multiple activity types we can leverage.

`pull_request` event

`opened` Runs the workflow whenever a PR is opened.	`synchronize` Runs the workflow whenever a new commit is pushed to the HEAD ref of the PR.
`closed`	`assigned`
`labeled`	`edited`
.	

```yaml
name: My package workflow on:
  pull_request:
    types: [opened, synchronize]
    branches:
      - main
      - 'releases/**'
```

We can specify one or more activity types

Activity types and event filters can be combined when needed

Event Filters & Activity Types

```yaml
name: Event Filters and Activity Types

on:
  pull_request:
    types:
      - opened
      - synchronize
    branches:
      - main

jobs:
  echo:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Running whenever a PR is opened or synchronized AND base branch is main"
```

Workflow Contexts

Access information about runs, variables, jobs, and much more

GitHub provides multiple sources of data in different contexts so that we can easily provide all the necessary information to our workloads

`github`

- Commit SHA
- Event name
- Ref of branch or tag triggering the workflow

`env`

Contains variables that have been defined in a workflow, job, or step. Changes based on which part of the workflow is executing.

`inputs`

Contains input properties passed via the keyword `with` to an action, to a reusable workflow, or to a manually triggered workflow.

`vars`

Contains custom configuration variables set at the organization, repository, and environment levels.

`secrets`

`matrix`

`needs`

`.`

Workflow Contexts

```yaml
name: Contexts - github

on:
  workflow_dispatch:

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: Display Event Information
      run: |
        echo "Event Name: ${{ github.event_name }}"
        echo "Ref: ${{ github.ref }}"
        echo "SHA: ${{ github.sha }}"
        echo "Actor: ${{ github.actor }}"
        echo "Workflow: ${{ github.workflow }}"
        echo "Run ID: ${{ github.run_id }}"
        echo "Run number: ${{ github.run_number }}"
```

Expressions

Use dynamic values and expressions in your workflows

- Can be used to reference information from multiple sources within the workflow
- Must use the `${{ <expression> }}` syntax
- Can be any combination of:

Literal values	Context values	Functions
Strings, numbers, booleans, null.	Values passed via the many workflow contexts.	Built-in functions provided by GitHub Actions.

- Support the use of functions and operators such as `!`, `<`, `>`, `!=`, `&`, `|`, and many others.

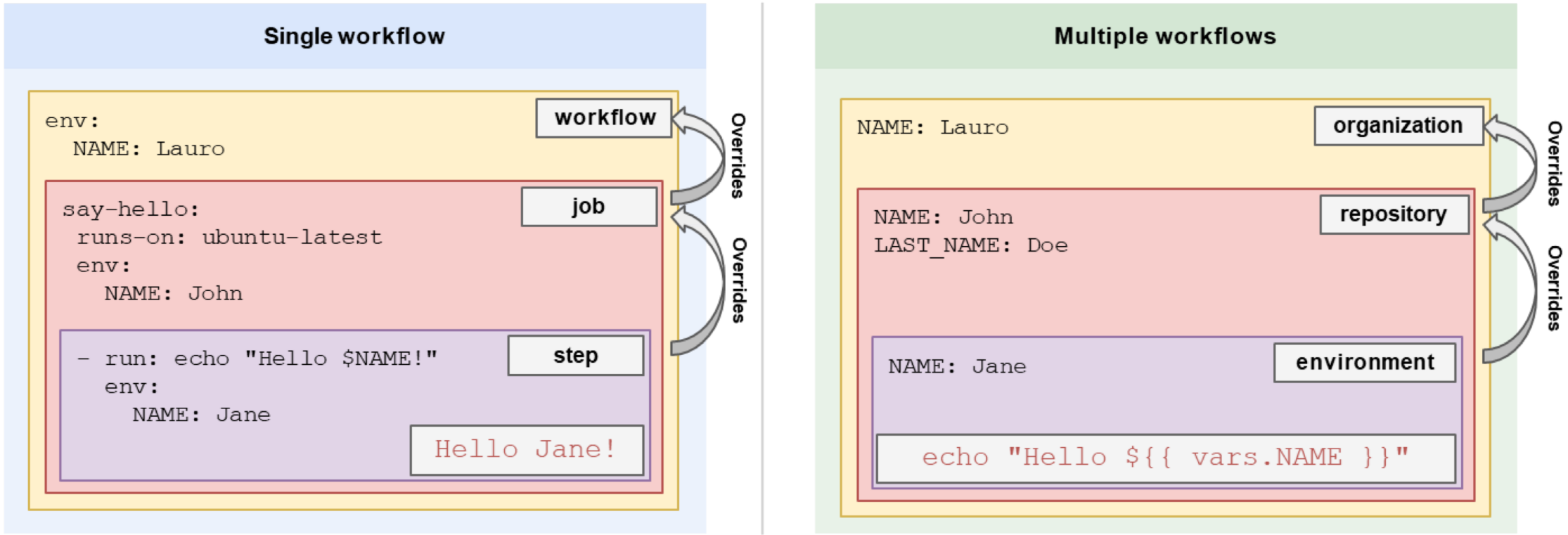
```yaml
name: My workflow
on: [push, pull_request]

jobs:
 tests:
   runs-on: ubuntu-latest
   steps:
     - if: ${{ github.event_name }}  = "push"
       run: |
         echo "Triggered by a push event"
         echo "Running on ${{ github.ref_name }}"
```

Expressions

```yaml
name: Expressions
run-name: Expressions | DEBUG - ${{ inputs.debug && 'ON' || 'OFF' }}

on:
  # push:
  workflow_dispatch:
    inputs:
      debug:
        type: boolean
        default: false

jobs:
  echo:
    runs-on: ubuntu-latest
    steps:
      - name: '[debug] Print start-up data'
        if: inputs.debug
        run: |
          echo "Triggered by: ${{ github.event_name }}"
          echo "Branch: ${{ github.ref }}"
          echo "Commit SHA: ${{ github.sha }}"
          echo "Runner OS: ${{ runner.os }}"
      - name: '[debug] Print when triggered from master'
        if: inputs.debug && github.ref == 'refs/heads/master'
        run: echo "I was triggered from master"
      - name: Greeting
        run: echo "Hello, world"
```

Variables

Set and reuse non-sensitive configuration information



Variables

```yaml
name: Using Variables

on:
  workflow_dispatch:

env:
  WORKFLOW_VAR: 'I am a workflow env var'
  OVERWRITTEN: 'I will be overwritten'
  UNDEFINED_VAR_WITH_DEFAULT: ${{ vars.UNDEFINED_VAR || 'default value' }}

jobs:
  echo:
    runs-on: ubuntu-latest
    env:
      JOB_VAR: 'I am a job env var'
      OVERWRITTEN: 'I have been overwritten at the job level'
    steps:
      - name: Print Env Variables
        env:
          STEP_VAR: 'I am a step env var'
          step_var2: 'I am another step env var'
        run: |
          echo "Step env var: ${{ env.STEP_VAR }}"
          echo "Step env var 2: $step_var2"
          echo "Job env var: ${{ env.JOB_VAR }}"
          echo "Workflow env var: ${{ env.WORKFLOW_VAR }}"
          echo "Overwritten: ${{ env.OVERWRITTEN }}"
      - name: Overwrite job variable
        env:
          OVERWRITTEN: 'I have been overwritten at the step level'
        run: |
          echo "Step env var: ${{ env.OVERWRITTEN }}"
```

Functions

Out-of-the-box functions to model complex behavior

General purpose functions

Provides a set of utility functions to interact with data from multiple contexts and model more complex behavior, such as more advanced control of the workflow and job execution.

`contains()`	`startsWith()`
`endsWith()`	`fromJSON()`
`toJSON()`	`.`

Status check functions

Provides a set of functions that allow using the status of the workflow, previous jobs or steps to define whether a certain job or step should be executed.

`success()`	`failure()`
`always()`	`cancelled()`

Use-case

`fromJSON()` can be used to convert variables from `string` to different data types.

Use-case

`!cancelled()` can be used to execute jobs or steps even if previous jobs or steps failed, but to prevent execution in case the workflow is cancelled.

Functions

```yaml
name: Functions

on:
  pull_request:
  workflow_dispatch:

jobs:
  echo1:
    runs-on: ubuntu-latest
    steps:
      - name: Print PR title
        run: echo "${{ github.event.pull_request.title }}"
      - name: Print PR labels
        run: |
          cat << EOF
          ${{ toJSON(github.event.pull_request.labels) }}
          EOF
      - name: Bug step
        if: ${{ !cancelled() && contains(github.event.pull_request.title, 'fix') }}
        run: echo "I am a bug fix"
      - name: Sleep for 20 seconds
        run: sleep 20
      - name: Failing step
        run: exit 1
      - name: I will be skipped
        if: ${{ success() }}
        run: echo "I will print if previous steps succeed."
      - name: I will execute
        if: ${{ failure() }}
```

Controlling the Execution Flow

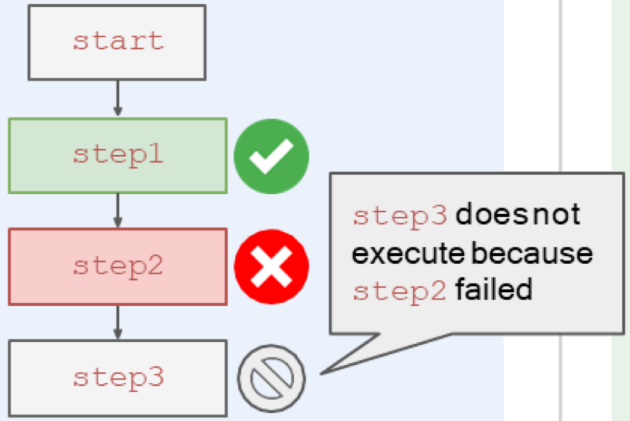
Execute jobs and steps conditionally, and set dependencies between jobs

Conditional job execution via the `if` key:

Standard execution

Downstream jobs and steps execute if and only if upstream jobs and steps succeed:

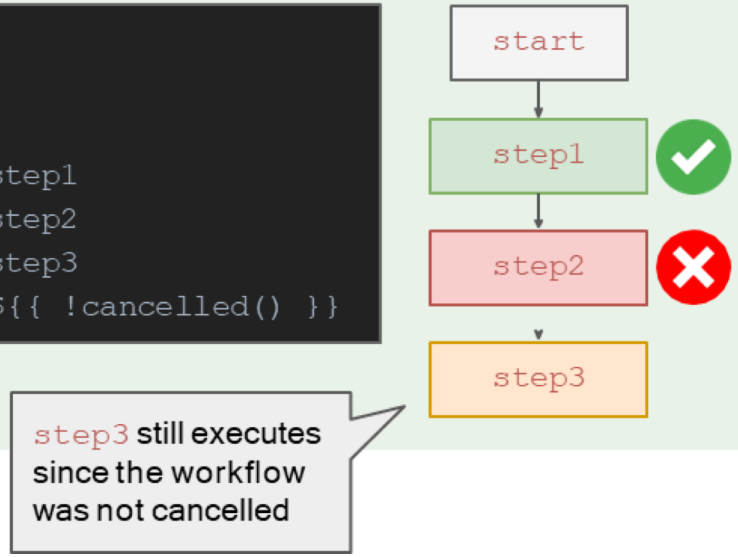
```yaml
jobs:
  job1:
    steps:
      - id: step1
      - id: step2
      - id: step3
```



Conditional execution

Downstream jobs and steps can be executed even if the upstream ones fail:

```yaml
jobs:
  job1:
    steps:
      - id: step1
      - id: step2
      - id: step3
        if: ${{ !cancelled() }}
```



Controlling the Execution Flow

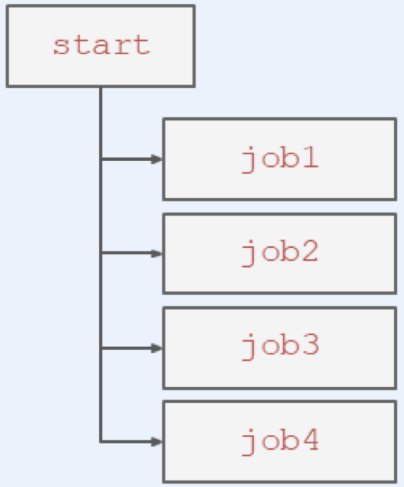
Execute jobs and steps conditionally, and set dependencies between jobs

Sequential job execution via the `needs` key:

Non-dependent execution

All jobs are executed in parallel by default:

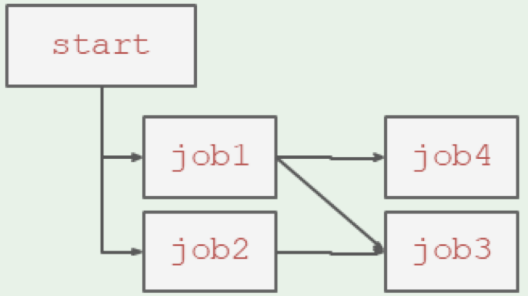
```yaml
jobs:
  job1:
  job2:
  job3:
  job4:
```



Dependent execution

Jobs wait until their dependencies successfully execute:

```yaml
jobs:
  job1:
  job2:
  job3:
    needs:
      - job1
  -   job2
  job4:
    needs: job1
```



`job3` executes only after `job1` and `job2` successfully complete

Inputs

Inputs enable us to request specific information from the workflow or action caller and use this information at runtime.

GitHub Action

```
inputs:
  url:
    description:     .'
    required: true
  max-trials:
    description: '   .'
    required: false
    default: '60'
```

Caller

```
jobs:
  job1:
    steps:
      - name: Ping URL
        uses: ping-url-example@v1
        with:
          url: https: /www.google.com
          max-trials: 10
```

Inputs must be defined in the definition of reusable workflows or custom actions so that they can be correctly used.

Use-case

Provide information such as target runtime version (when setting up node, for example), caching keys, among others.

Outputs

Output data from jobs for later usage

```yaml
jobs:
 welcome:
   runs-on: ubuntu-latest
   outputs:
     name: ${{ steps.step1.outputs.NAME }}
   steps:
     - id: step1
       run: echo "NAME=Lauro"  > "$GITHUB_OUTPUT"
 goodbye:
   runs-on: ubuntu-latest
   needs: welcome
   steps:
     - run: echo "Bye, ${{ needs.welcome.outputs.name }}"
```

Outputs

Output data from jobs for later usage

1. Define a step ID for the steps that produce outputs

```
    - id: step1
```

Outputs

Output data from jobs for later usage

1. Define a step ID for the steps that produce outputs
2. Echo key-value pairs to the `$GITHUB_OUTPUT` variable

```yaml
    - id: step1
      run: echo "NAME=Lauro"  > "$GITHUB_OUTPUT"
```

Outputs

Output data from jobs for later usage

```yaml
outputs:
  name: ${{ steps.step1.outputs.NAME }}
steps:
  - id: step1
    run: echo "NAME=Lauro"  > "$GITHUB_OUTPUT"
```

1. Define a step ID for the steps that produce outputs
2. Echo key-value pairs to the `$GITHUB_OUTPUT` variable
3. Mention the outputs in the `outputs` section of the job

Outputs

Output data from jobs for later usage

```yaml
jobs:
 welcome:
   runs-on: ubuntu-latest
   outputs:
     name: ${{ steps.step1.outputs.NAME }}
   steps:
     - id: step1
       run: echo "NAME=Lauro"  > "$GITHUB_OUTPUT"
 goodbye:
   runs-on: ubuntu-latest
   needs: welcome
```

1. Define a step ID for the steps that produce outputs
2. Echo key-value pairs to the `$GITHUB_OUTPUT` variable
3. Mention the outputs in the `outputs` section of the job
4. List the job as a dependency of jobs that need the output

Outputs

Output data from jobs for later usage

```yaml
jobs:
 welcome:
   runs-on: ubuntu-latest
   outputs:
     name: ${{ steps.step1.outputs.NAME }}
   steps:
     - id: step1
       run: echo "NAME=Lauro"  > "$GITHUB_OUTPUT"
 goodbye:
   runs-on: ubuntu-latest
   needs: welcome
   steps:
     - run: echo "Bye, ${{ needs.welcome.outputs.name }}"
```

1. Define a step ID for the steps that produce outputs
2. Echo key-value pairs to the `$GITHUB_OUTPUT` variable
3. Mention the outputs in the `outputs` section of the job
4. List the job as a dependency of jobs that need the output
5. Access the output via the `needs` context

Managing Workflows

- Workflows are a configurable automated process that will run one or more jobs.
- Workflows are defined by a YAML file checked in to your repository and will run when triggered by an event in your repository.
- A workflow must contain one or more events that will trigger the workflow, and one or more jobs, each of which will execute on a runner machine and run a series of one or more steps.
- Each step can either run a script that you define or run an action, which is a reusable extension that can simplify your workflow.

Managing Dependencies

- To cache dependencies for a job, you can use GitHub's cache action.
- The action creates and restores a cache identified by a unique key.
- Caching allows your workflow to store data for future uses, limiting the time spent downloading and making requests.
- If you use a package manager like npm and on every run, you need to get the same dependency, you should cache it.

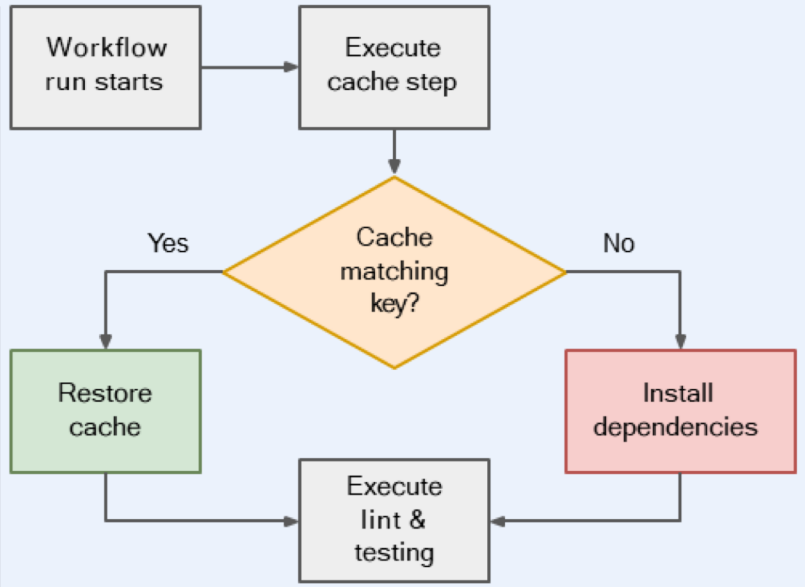
Caching

Speed up workflow runs by caching stable files

Caching allows us to store files and later retrieve them based on a key. Workflows can access the cache from their branch or from the default branch.

Caching process

```yaml
steps:
 - name: Setup Java

 uses:actions/setupjava@v4
   with:
     distribution: 'adopt'
     java-version: '11'
     cache: 'maven'
 - name: Setup Java
   uses:mvn dependency
```



Use-case

Cache dependencies to speed up execution by avoiding always downloading and installing them.

Artifacts

Artifacts

- Stored for up to 90 days
- Managed via two actions
  - upload-artifact
  - download-artifact
- Recommended when the stored files are likely to be accessed outside the workflow, for example:
  - Build outputs
  - Test results
  - Logs

Artifacts

Share data between jobs and store data after workflows have completed

Artifacts

- Stored for up to 90 days
- Managed via two actions
  - upload-artifact
  - download-artifact
- Recommended when the stored files are likely to be accessed outside the workflow, for example:
  - Build outputs
  - Test results
  - Logs

Caching

- Stored for up to 7 days
- Managed via a single action
  - cache
- Recommended when the stored files are likely to be accessed only within the workflow, for example:
  - Build dependencies

Use-case

Uploading and downloading build outputs for deployments

Optimizing Workflows

- You can manage workflow concurrency, implement conditional statements, and augment workflows with in-house scripts to streamline and accelerate your development pipeline.
- You can analyze your current workflow to see if you can divide it into different jobs.
- Jobs as opposed to steps can be run in parallel by different machines (runners).
- You can select a faster runner.
- The strategy matrix is a very interesting, yet powerful syntax in GitHub action. The matrix allows you to test many (up to 256) version configurations for the same job.

Matrices

Run several variations of the same job

```yaml
name: My NPM package workflow
on: push

jobs:
  backwards-compatibility:
    name: ${{ matrix.os }}-${{ matrix.node }}
    strategy:
      matrix:
        node: [14, 16, 18]
        os:
          - ubuntu-latest
          - macos-latest
    -     windows-latest
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node }}
```

- ubuntu-latest-14
- ubuntu-latest-16
- ubuntu-latest-18
- macos-latest-14
- macos-latest-16
- macos-latest-18
- windows-latest-14
- windows-latest-16
- windows-latest-18

Use-case

Run test suits in parallel in multiple Node versions before publishing an NPM package to ensure backward compatibility.

Warning

Each executed job counts towards billing purposes. A matrix generating 9 jobs of 3 minutes each will lead to 27 minutes of billed time.

Environments

Create multiple environments with different rules for multiple deployments

Staging environment

UAT environment

Production environment

Required approvers:

- John Smith
- Jane Doe

Branch restrictions: `main`

Variables:

- `FIRST_NAME`: Lauro

```yaml
on:
  push:
    branches: main

jobs:
  welcome:
    runs-on: ubuntu-latest
    environment: prod
    steps:
      - run: echo "Hi, ${{ vars.FIRST_NAME }}"
```

Environment variables are available in the `vars` context, while secrets are available in the `secrets` context.

Use-case

We can use multiple GitHub Actions Environments to require manual approvals before deploying to higher environments.

GitHub Custom Actions

- Write and reuse any custom logic

- Custom Actions allow us to write, encapsulate, and reuse pieces of custom logic in any programming language.

There are three types of custom actions

	Composite Actions	JavaScript Actions	Docker Actions
Pros / Cons	■ Simplest type of Custom Action. ■ Grouping of other GitHub Actions. ■ May not be enough for complex functionality.	■ Allows writing any type of custom logic. ■ `@actions` packages provide lots of functionality. ■ Requires JavaScript knowledge and Node environment.	■ Allows writing any type of custom logic. ■ Can be written in any programming language. ■ Might be more verbose as only JSsupports `@actions` package.
Common Requirements	■ Requires an `action.yaml` file ■ Must be on its own repository if it is to be reused by other repos.	■ Requires an `action.yaml` file ■ Must be on its own repository if it is to be reused by other repos.	■ Requires an `action.yaml` file ■ Must be on its own repository if it is to be reused by other repos.

Composite Actions

- Composite actions are a straightforward type of custom action in GitHub Actions, allowing you to group multiple steps that can be executed together.

Composite Action - Use Case

- **Use Case:**
  - Simplifying Java and Maven Dependency Setup
- **Scenario:**
  - As a Java developer, you frequently work on projects that utilize Maven for dependency management. However, setting up the Java environment and installing dependencies can be time-consuming and error-prone. You want to streamline this process by creating a reusable GitHub Action that automates the setup of Java and Maven dependencies based on the pom.xml file.
- **Solution:**
  - You create the "ANZ Setup Java" GitHub Action, which allows caching both Java and Maven dependencies. This action simplifies the setup process by providing the necessary tools and configurations to ensure a consistent development environment.

JavaScript Actions

- A JavaScript (JS) custom action in the context of GitHub Actions is a reusable piece of automation logic written in JavaScript that can be executed as part of a GitHub Actions workflow. It allows you to perform specific tasks, interact with GitHub's APIs, or automate actions within your software development workflows.

JavaScript Actions - Use Case

- **Use Case:**
  - Automating NPM Dependency Updates and Creating Pull Requests
- **Scenario:**
  - Your project relies on NPM packages, and you want to ensure that your dependencies are always up to date. However, manually checking for updates and creating pull requests for dependency updates can be time-consuming and error-prone. You want to automate this process to ensure that your project stays current with the latest package versions.
- **Solution:**
  - You can create a custom GitHub Action that checks for updates to NPM packages, creates a pull request with the updated **package*.json** files, and notifies you of any available updates. This action can be triggered on a schedule or manually through the GitHub UI, providing a seamless way to keep your project dependencies up to date.

Docker Container Actions

- A Docker container action is a type of GitHub Action that runs within a Docker container environment. Unlike JavaScript or composite actions, which execute within the GitHub-hosted runners, Docker container actions provide a more flexible and customizable execution environment by running within a Docker container specified by the user.

Docker Container Actions - Use Case

- **Use Case:**
  - (**Continuous URL Monitoring Action**)You need a solution to continuously ping a URL until a specific condition is met, such as receiving a successful response or reaching a maximum number of trials. This functionality is crucial for monitoring services and ensuring their availability.
- **Scenario:**
  - As a DevOps engineer, you want to create a GitHub Action that can repeatedly ping a URL and check for a successful response. If the URL does not return a status code of 200 within a specified number of trials, the action should fail. This action will be useful for monitoring critical endpoints and triggering alerts if there are any issues.
- **Solution:**
  - You develop the "Ping URL" Docker Custom Action, which allows users to specify a URL to ping and the maximum number of trials before the action fails. The action runs in a Docker container, ensuring portability and consistency across different environments. It utilizes Python and the requests library to send HTTP **requests** to the specified URL and checks the response status code.

Reusable Workflows

Create and reuse workflows to avoid duplication of common tasks

Any workflow can be made reusable by adding `workflow_call` to the top-level `on` key of the workflow definition.

Reusable Workflow

```yaml
on:
  workflow_call:
    inputs:
      aws-region:
        type: string
    secrets:
      auth-token:
        required: true
    outputs:
      server-url:
        value: <value>
```

Caller Workflow

- Inputs are passed via the `with` keyword
- Secrets are passed via the `secrets` keyword
- Outputs are accessed via the `outputs` keyword

```yaml
jobs:
 backend-infra-nonprod:
   uses: reusable-deploy@v1
   secrets:
     auth-token: ${{ secrets.GH_PAT }}
   with:
     aws-region: ${{ vars.AWS_REGION }}
```

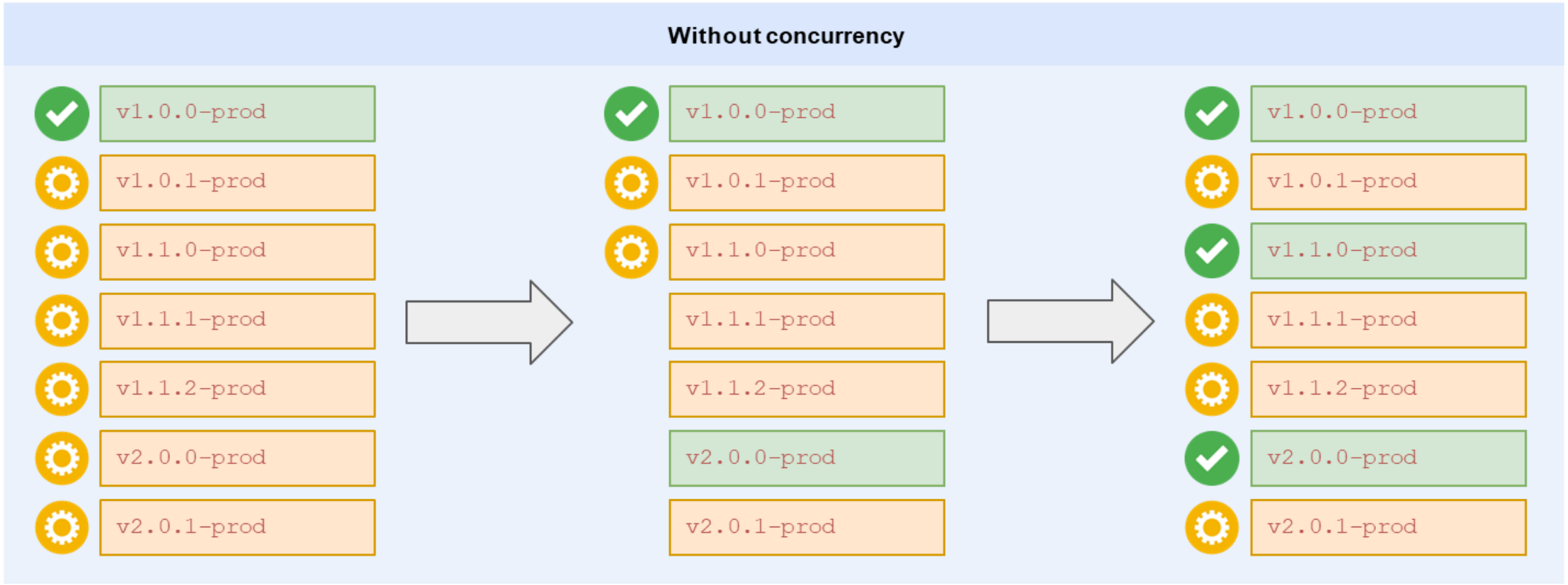
We often need to create and provide a custom Personal Access Token (PAT) to give the reusable workflows access to other repositories.

Use-case

One example of workflow reusability is to deploy to different environments while ensuring that the deployment process is the same.

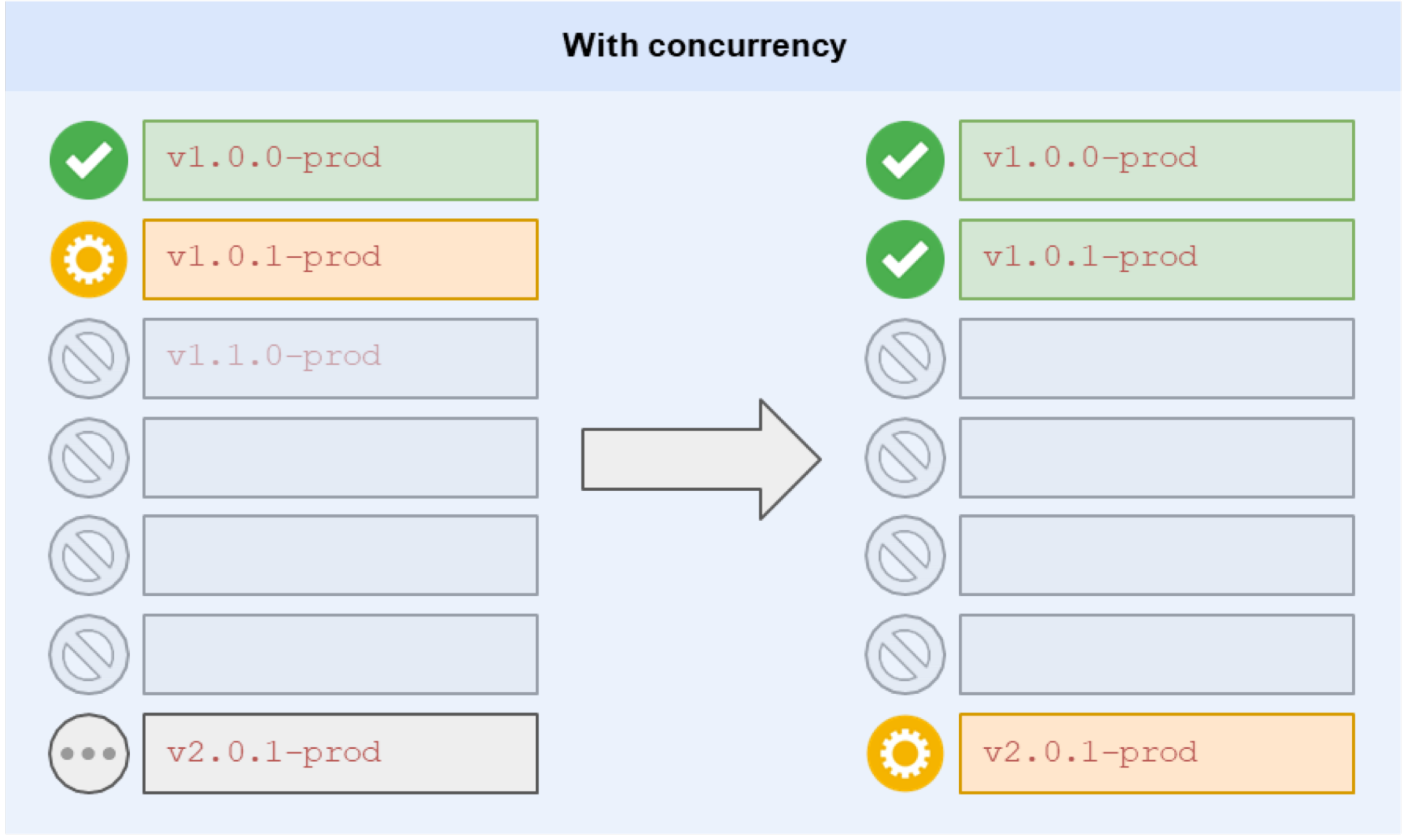
Managing Concurrency

Fine-tune the execution behavior of concurrent workflows



Managing Concurrency

Fine-tune the execution behavior of concurrent workflows



Use-case

Cancel older workflows with pending deployments once a newer version of the application is ready to be deployed to higher environments.

Warning

The concurrency group must be unique across workflows to prevent cancelling runs from other workflows! Also, ensure you turn the cancel-in-progress flag off for deployments.

Workflow Security - Handling Secrets

Best practices for managing and using secrets in workflows

- Do not use complex data types for storing secrets.

```
{
 "sensitiveValue1": "abdc",
 "sensitiveValue2": "123"
}
```

```
SENSITIVE_VALUE1 = "abcd"
```

```
SENSITIVE_VALUE2 = "123"
```

Workflow Security - Handling Secrets

Best practices for managing and using secrets in workflows

- Do not use complex data types for storing secrets.
- Register generated sensitive values within actions.

```
{
 "sensitiveValue1": "abdc",
 "sensitiveValue2": "123"
}
```

```
SENSITIVE_VALUE1 = "abcd"
```

```
SENSITIVE_VALUE2 = "123"
```

```
echo " :add-mask :$GENERATED_SENSITIVE_VALUE"
```

```
const generatedSecret = jwt.sign(   .);
core.setSecret(generatedSecret);
```

Workflow Security - Handling Secrets

Best practices for managing and using secrets in workflows

- Do not use complex data types for storing secrets.
- Register generated sensitive values within actions.
- Make sure that any third-party or external action your workflows depend on does not expose sensitive values in logs or to other external services.
  - This can be done by auditing the source code of these actions.
- Regularly rotate secrets.
- Delete unused secrets.
- Limit those with access to create and update secrets.
- Always use credentials with as few permissions as necessary

```
{
 "sensitiveValue1": "abdc",
 "sensitiveValue2": "123"
}
```

```
SENSITIVE_VALUE1 = "abcd"
```

```
SENSITIVE_VALUE2 = "123"
```

```
echo " :add-mask :$GENERATED_SENSITIVE_VALUE"
```

```
const generatedSecret = jwt.sign(   .);
core.setSecret(generatedSecret);
```

Workflow Security - Handling Tokens

Best practices for managing and using tokens and permissions

- Never use a classic PAT (Personal Access Token) to grant a workflow access to code from another repo.
  - Ideally, create a GitHub App and use its short-term credentials.
  - If needed, use a fine-grained PAT and give as few permissions as necessary for the workflow to do its job (i.e. only read access, only to the necessary repos).
  - When using a fine-grained PAT, rotate it regularly.
  - When using a fine-grained PAT, remember that it is bound to a specific user.
- When extending the permissions of $GITHUB_TOKEN, use only the minimum set of permissions required by the workflow.
- Do not pass the workflow's token stored in $GITHUB_TOKEN to untrusted third-party software (for example, custom actions from untrusted sources).

Workflow Security - Preventing Script Injection

Avoid malicious code execution in Workflows

- Script injection happens when attackers inject malicious code into the workflow's context in the hope that it will be executed.
  - Examples include contexts ending in body, default_branch, email, head_ref, label, message, name, among others.
- How to avoid script injection:
  - Create custom actions instead of executing inline shell scripts.
  - Use intermediary environment variables.
  - Reduce as much as possible the dependency of custom actions on inputs from external users.
  - Setup code scanning.

Workflow Security - Using OpenID Connect

Use GitHub Actions' authentication instead of long-term access credentials

- For providers and external services that support OpenID Connect, it is possible to set up authentication so that we obtain short-term credentials from cloud providers instead of having to store
- long-term access credentials.
- Downsides of long-term credentials:
  - Are hard-coded in the secrets used by the workflow.
  - Need to be rotated regularly.
  - Are valid beyond the execution of the workflow.
  - Any misstep can expose these credentials, which normally have many permissions since they allow managing cloud resources.

PLAINTEXT SECRETS

Never store any API key, token or password in plain text!

GitHub Secrets allows you to store and use your secrets in a safe way for your workflows. Running a CI workflow is also a good place to **implement secrets detection and remediation.**
You can leverage the `ggshield-action` for that.

GitGuardian [gitguardian-shield]

MINIMALLY SCOPED CREDENTIALS

Every credential used in the workflow should have the minimum required permissions to execute the job.

In particular, use the '**permissions' key** to make sure the **GITHUB_TOKEN** is configured with the least privileges for each job.

To limit their scope, **environnement variables** should be declared at the step level when possible.

SELF-HOSTED RUNNERS

It is strongly recommended to not use self-hosted runners for open-source repositories!

```
runs-on: self-hosted
```

With self-hosted runners, you are responsible for hardening your virtual machines:

- configure a dedicated low-privileged user
- preferably use isolated and ephemeral workloads to execute the jobs
- use logging and security monitoring to ensure visibility

For public repositories, you should also configure GitHub to require approval for any workflow runs.

PREFER USING OPENID CONNECT

As a best practice, use **OpenID Connect (OIDC) instead of long-lived secrets** to allow your workflows to interact with your cloud provider.

```yaml
# .github/workflows/fake-build.yaml
name: Create an issue before building

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  create_issue:
    name: Create an issue
    runs-on: ubuntu-latest

    permissions:
      issues: write

    steps:

      - name: Create issue using REST API
        uses: someperson/post-issue@f054a8b539a109f9f41c372932f1ae047eff08c9
        with:
          token: ${{ secrets.GITHUB_TOKEN }}

  build:
    name: Install dependencies
    runs-on: ubuntu-latest
    steps:

    - uses: actions/checkout@v3
      with:
        ref: ${{ github.event.pull_request.head.sha }}

    - name: Setup Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: '3.10'

    - name: Install dependencies
    - run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

####                          Rest of the workflow ...
```

```yaml
# .github/workflows/fake-deployment.yaml

on: deployment

jobs:
  deploy:
    name: Build and test
    runs-on: ubuntu-latest
    steps:

      - name: Deploy cloud resource service
      - uses: cloudprovider/deploy@v13d241b293754004c80624b5567555c4a39ffbe3
        with:
          token: ${{ secrets.CLOUD_PROVIDER_SECRET }}
```

SPECIFIC VERSION TAGS

When using third-party actions, **pin the version with a commit hash** rather than a tag to shield your workflow from potential supply-chain compromise.

PULL_REQUEST_TARGET

Don't check out external PRs when using the `pull_request_target` event:

```
on: pull_request_target
...
 - uses: actions/checkout@v3
   with:
    ref: ${{ github.event.pull_request.head.sha }}
```

You are giving write permission and secrets access to untrusted code. Any building step, script execution, or even action call could be used to compromise the entire repository.

UNTRUSTED INPUT

Don't directly reference values you don't control, like:

```
echo "${{github.event.pull_request.title}}"
```

It's all too easy for a malicious PRs to be executed in your workflow. Instead:

- use an action with arguments (recommended):

```
uses: fakeaction/printtitle@v3
with:
  title: ${{ github.event.pull_request.title }}
```

- or bind the value to an intermediate environment variable:

```
- name: Print title
  env:
    PR_TITLE: ${{ github.event.pull_request.title }}
  run: |
  echo "$PR_TITLE"
```

Workflow Billing

Understand how workflows are billed

- Free for:
  - GitHub-hosted runners on public repos
  - Self-hosted runners (you still need to pay for the underlying infrastructure hosting the runners, for example in AWS)
- Not free (will count towards your plan's usage):
  - GitHub-hosted runners on private repos
- GitHub hosted runner usage:
  - Linux has a multiplier of 1
  - Windows has a multiplier of 2
  - macOS has a multiplier of 10
- Large runners are always billed, including public repositories.
- Runs are rounded upwards to the nearest minute for billing purposes.
- Storage refers to storage used by Artifacts and Packages.
  - Caching is not included, but it limited to 10GB per repository.

Plan	Usage limits
GitHub Free	**Storage:** 500 MB **Minutes:** 2000 / month
GitHub Pro	**Storage:** 1 GB **Minutes:** 3000 / month
GitHub Free for Organizations	**Storage:** 500 MB **Minutes:** 2000 / month
GitHub Team	**Storage:** 2 GB **Minutes:** 3000 / month
GitHub Enterprise Cloud	**Storage:** 50 GB **Minutes:** 50.000 / month

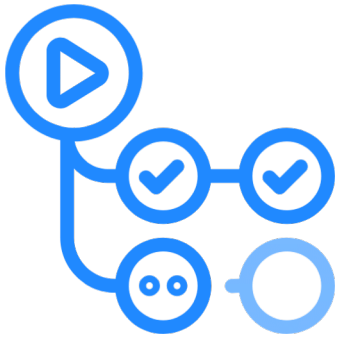
Jenkins to Github Actions Migration

Why migrate from Jenkins to GitHub Actions?

Jenkins	GitHub Actions
Server needs installation	No installation required as it is on the cloud
Tasks or jobs will be synchronous, which will consume more time to deploy a product to market	Asynchronous CI/CD is achieved
Is based on accounts and triggers and centers on builds which does not conform to GitHub events	Provides actions to every Github event and supports many languages and frameworks
Need to run on a Docker image for environment compatibility	Compatible with any environment
Plugins are available to support caching mechanisms	Have to write your own caching mechanism if you require caching
Does not have the capability to be shared	Can be shared via the GitHub Marketplace.

Jenkins Migration

Action Importer



What is GitHub Actions Importer

- Helps plan and automate migrations to GitHub Actions
- “Best effort” migrations
  - Goal of at least 80% of existing workflow can be converted automatically
  - Human review required

Travis CI

Bamboo

Azure DevOps

circleci

GitLab

Organization

- Creating an organization on GitHub without a clear purpose is not uncommon, especially if you're exploring the platform or its features. GitHub organizations are typically used for collaborative software development, project management, and team collaboration. Here are some common use cases for GitHub organizations:
  - Open Source Projects
  - Team Collaboration
  - Company or Group Projects
  - Community or Interest Groups
  - Educational Institutions
  - Experimentation

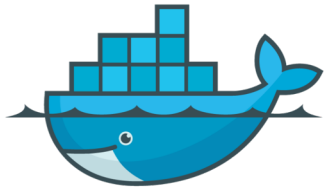
GitHub organizations

- GitHub organizations are shared accounts where an unlimited number of people can collaborate across many projects at once. They offer a number of benefits over using personal accounts, including:
  - Fine-grained access control
  - Centralized management
  - Branding and identity

Project

Java App with Maven

Build Artifact



Docker Image

Build Image

Private Repository

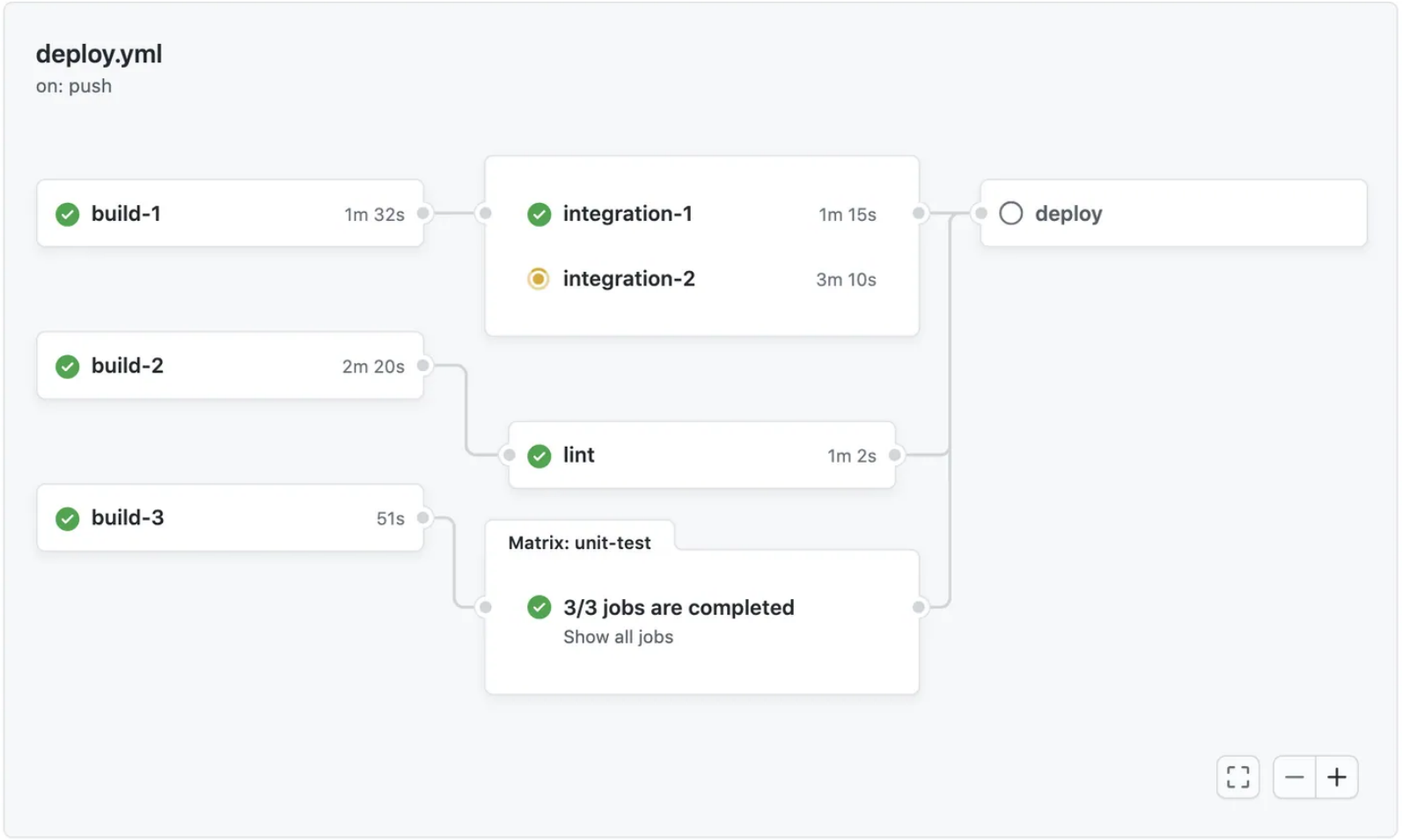
Push Docker Repo

Governance and compliance considerations

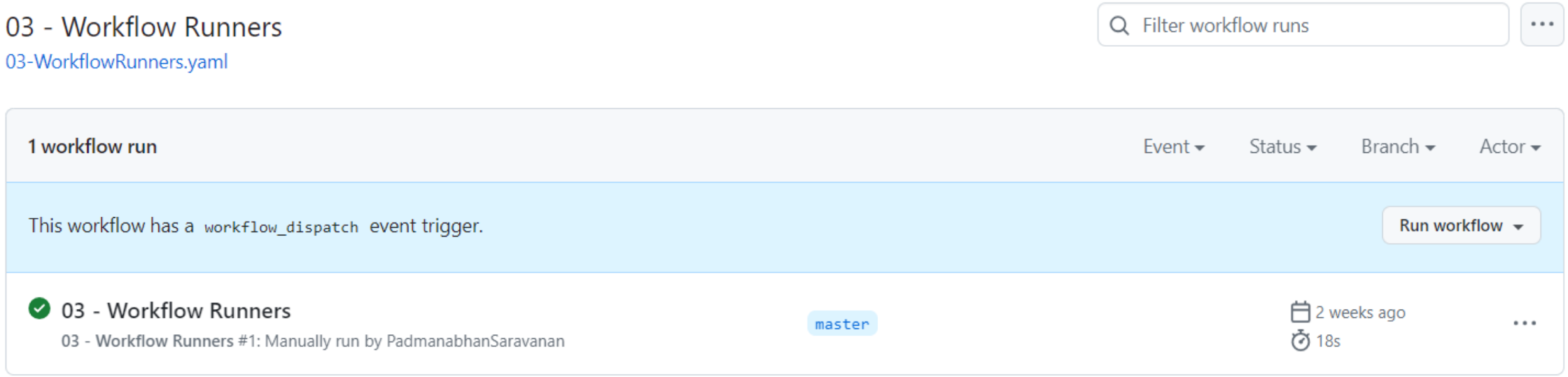
- Determine Allowed Actions
- Configure Actions at Various Levels
- Use OIDC with Reusable Workflows
- Audit and Log Management
- Continuous Evaluation and Improvement

Monitoring and observability

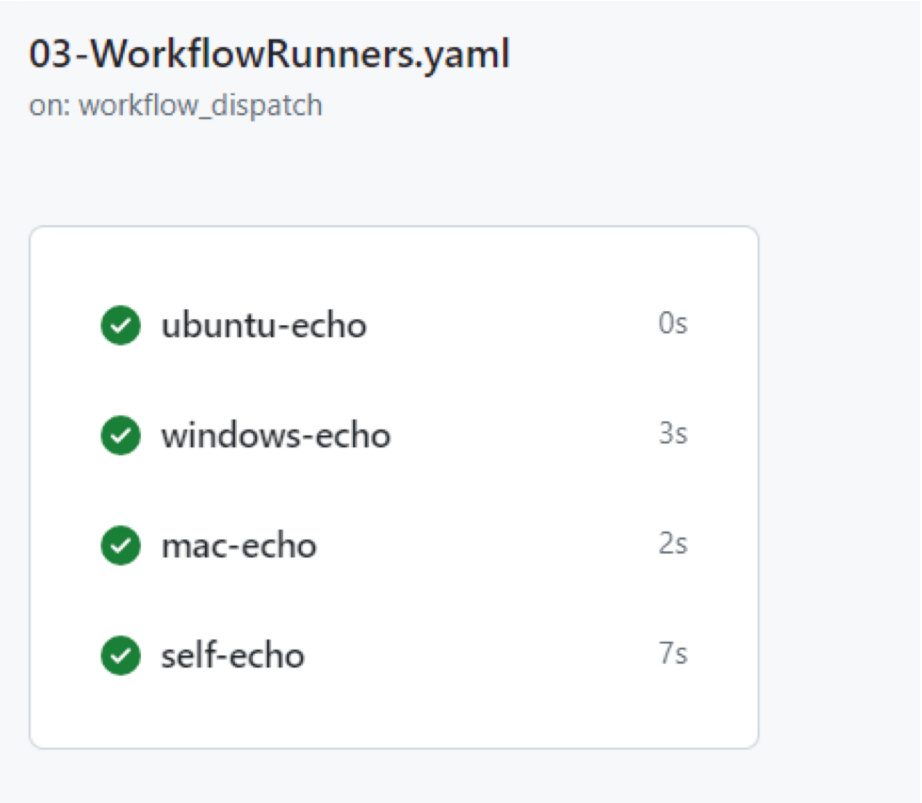
Using the visualization graph



Viewing workflow run history



Viewing job execution time



Using workflow run logs

Viewing logs to diagnose failures