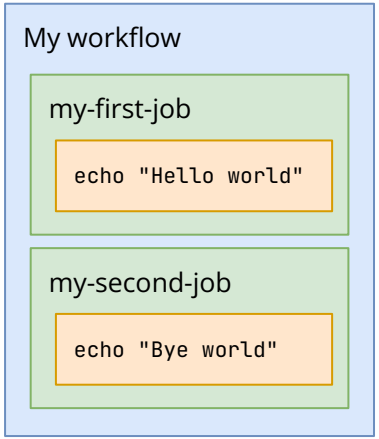
GitHub Actions

Building Blocks

Workflows, Jobs & Steps

Learn the building blocks of GitHub Actions

- 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: My 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"
```

GitHub Actions

Workflow Events

Workflow Events

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`					

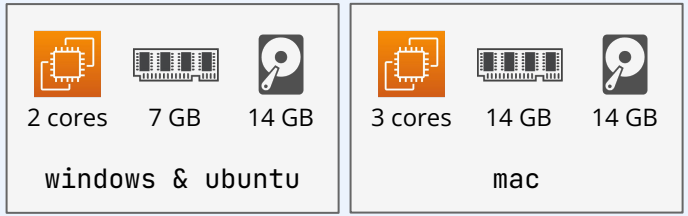
GitHub Actions

Workflow Runners

Workflow 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!

GitHub Actions

Actions

Actions

Custom applications to perform complex, frequently repeated tasks

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

jobs:
 node-20-release:
   runs-on: ubuntu-latest
   steps:
     - uses: actions/setup-node@v3
       with:
         node-version: 20
     - run: npm ci
     - run: npm test
     - run: npm publish ...
```

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

GitHub Actions

Event Filters

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 NPM package workflow
on:
  push:
    branches:
      - main
      - 'releases/**'
    paths-ignore:
      - 'docs/**'

jobs:
  node-16-tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/setup-node@v3
        with:
          node-version: 16
      - run: npm ci
      - run: npm test
```

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

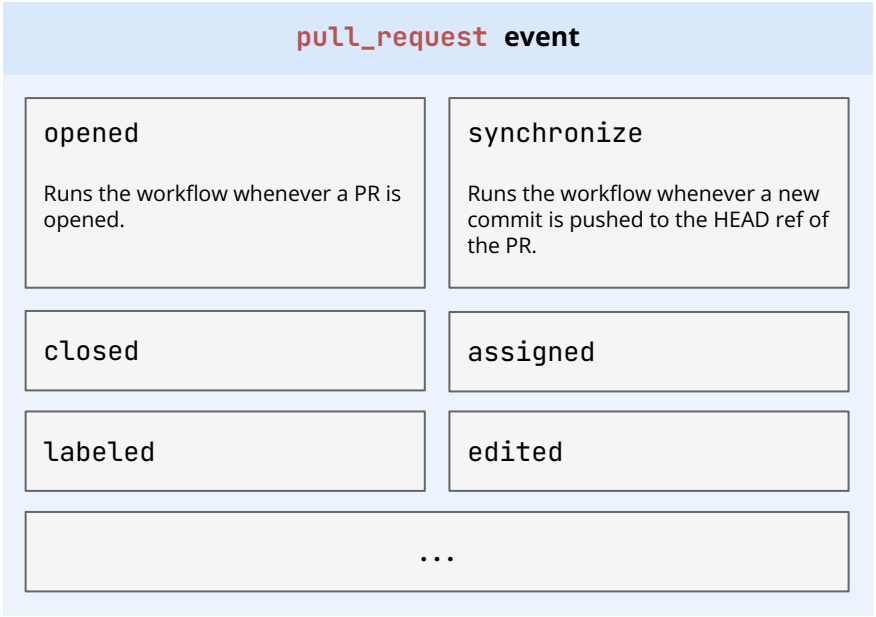
GitHub Actions

Activity Types

Activity Types

Specify which types of certain triggers execute our workflow

Many triggers have multiple activity types we can leverage.




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

jobs:
  node-20-tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/setup-node@v3
        with:
          node-version: 20
      - run: npm ci
      - run: npm run test
```

We can specify one or more activity types

Activity types and event filters can be combined when needed

GitHub Actions

Workflow Contexts

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` `...`

GitHub Actions

Expressions

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.

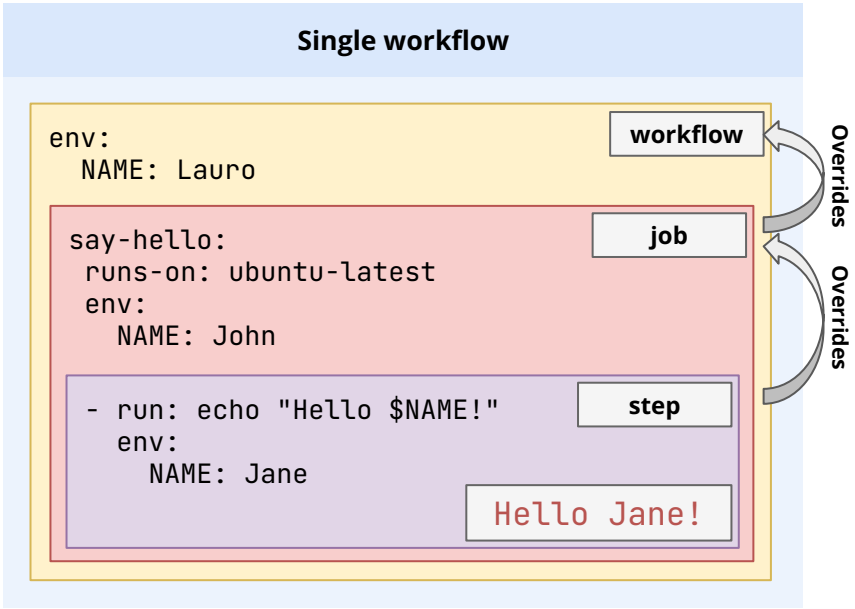
```yaml
name: My NPM workflow
on: [push, pull_request]

jobs:
 node-16-tests:
   runs-on: ubuntu-latest
   steps:
     - if: ${{ github.event_name }} == "push"
       run: |
         echo "Triggered by a push event"
         echo "Running on ${{ github.ref_name }}"
     - uses: actions/setup-node@v3
       with:
         node-version: 16
     - run: npm ci
     - run: npm test
```

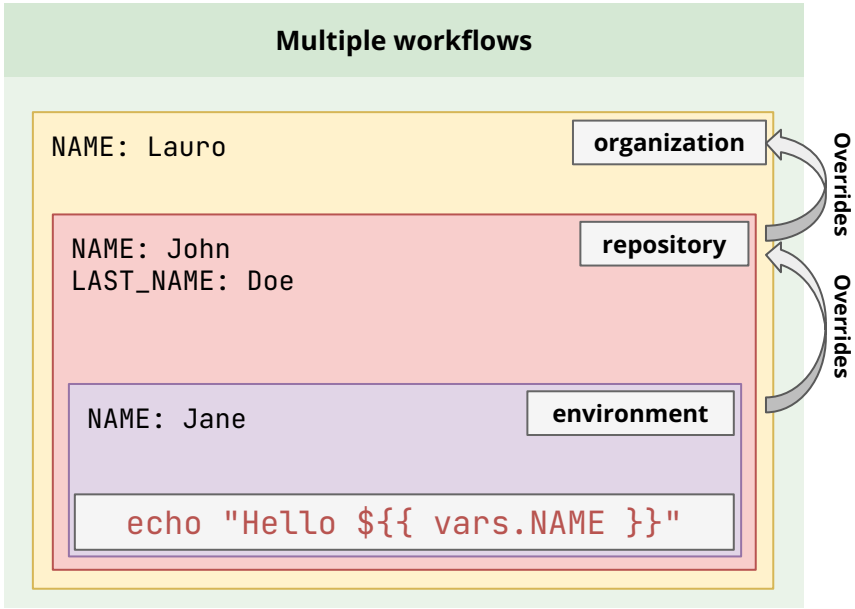

GitHub Actions

Variables

Variables

Set and reuse non-sensitive configuration information





GitHub Actions

Functions

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.

GitHub Actions

Controlling the Execution Flow

Controlling the Execution Flow

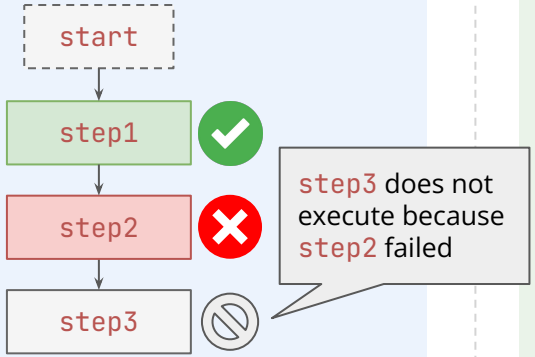
Execute jobs and steps conditionally, and set dependencies between jobs

Conditional job execution via the `if` key:

Standard execution

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

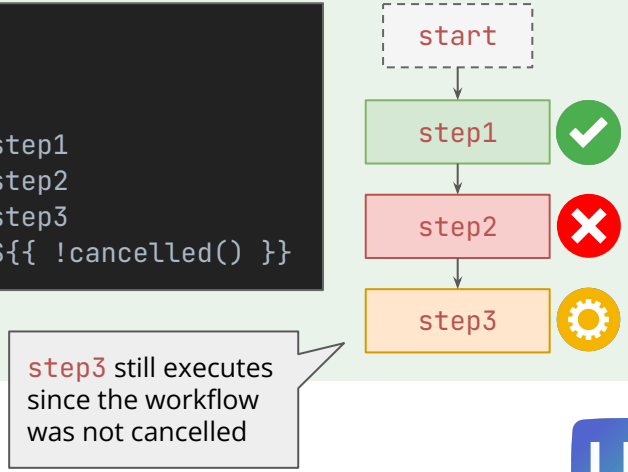
```yaml
jobs:
  job1:
    steps:
      - id: step1
      - id: step2
      - id: step3
```



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

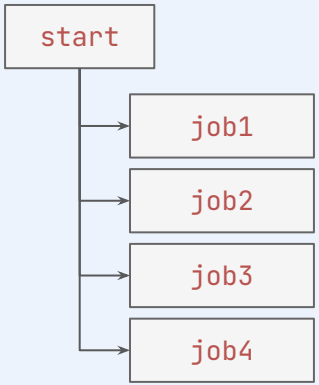
Execute jobs and steps conditionally, and set dependencies between jobs

Sequential job execution via the `needs` key:

Non-dependent execution

All jobs are executed in parallel by default:

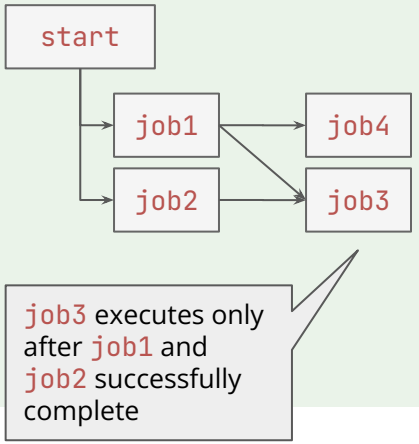
```yaml
jobs:
  job1:
  job2:
  job3:
  job4:
```



Dependent execution

Jobs wait until their dependencies successfully execute:

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



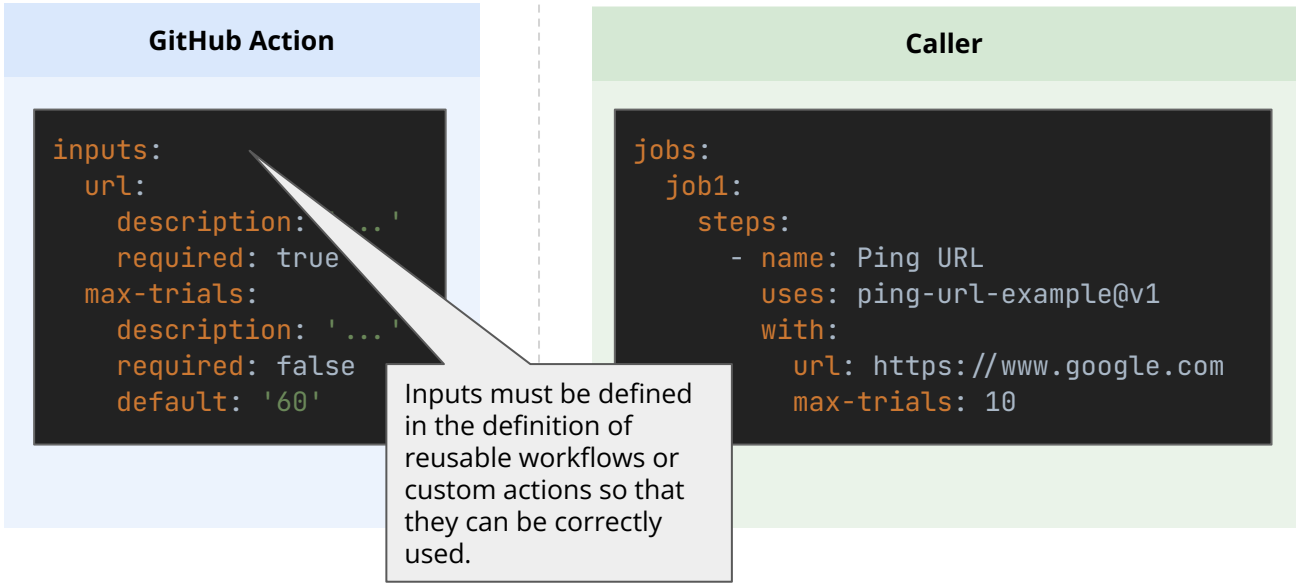
GitHub Actions

Inputs

Inputs

Provide information to customize workflows and actions

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



Use-case

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

GitHub Actions

Outputs

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

```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

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

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

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

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

GitHub Actions

Caching

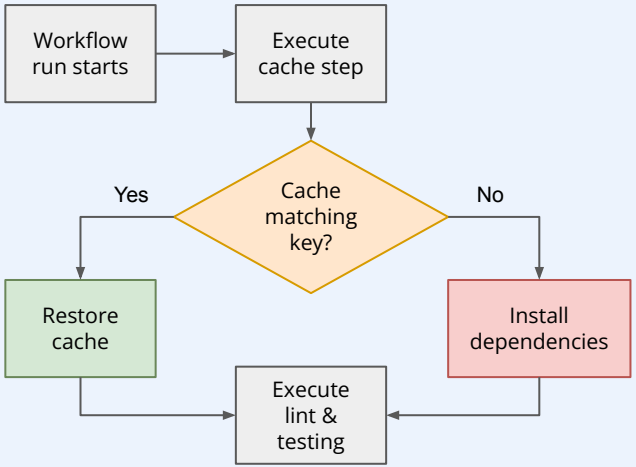
Caching

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:
 - uses: actions/cache@v3
   id: cache
   with:
     path: node_modules
     key: ${{ hashFiles('**/package-lock.json') }}
 - name: Install dependencies
   if: steps.cache.outputs.cache-hit != 'true'
   run: npm ci
 - name: Lint & test
   run: |
     npm run lint
     npm run test
```



Use-case

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

GitHub Actions

Artifacts

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

GitHub Actions

Matrices

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.

GitHub Actions

Environments

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 }}"
```

Use-case

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

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

GitHub Actions

Custom Actions

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 JS supports `@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.

GitHub Actions

Reusable Workflows

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

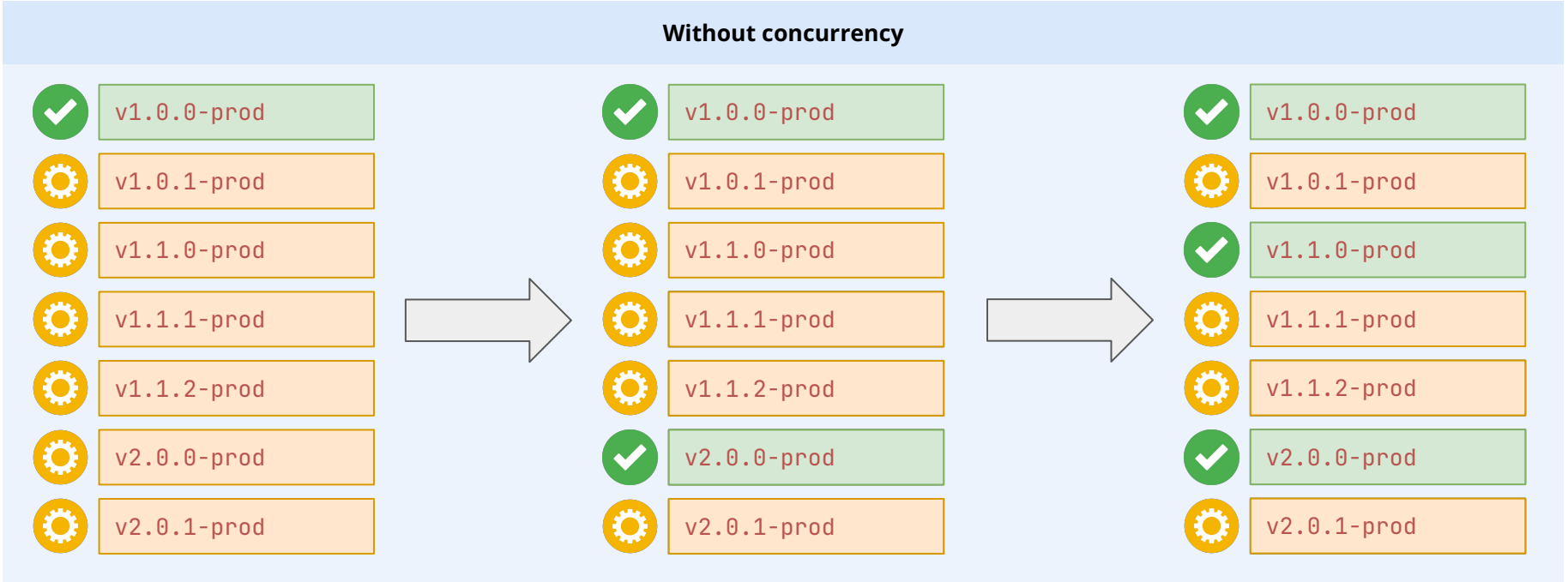
One example of workflow reusability is to deploy to different environments while ensuring that the deployment process is the same.

GitHub Actions

Managing Concurrency

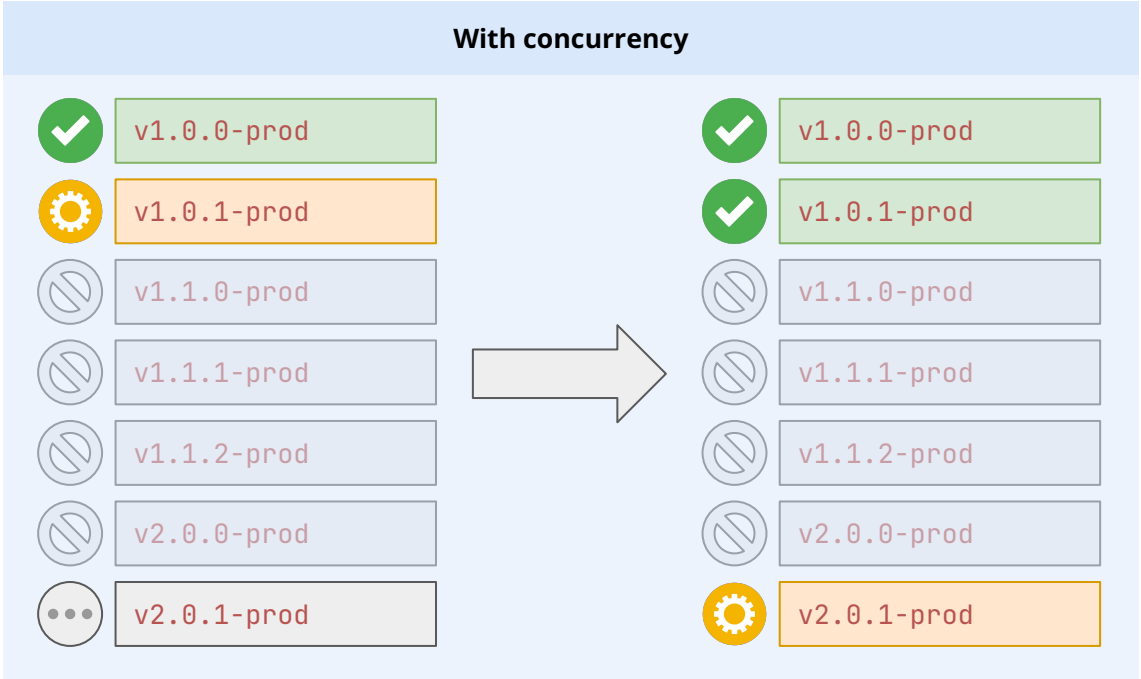
Managing Concurrency

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.

GitHub Actions

Workflow Security

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.

1 **Create a role to be used by workflows**

The role should contain the minimum set of permissions for the workflows to accomplish their tasks.

2 **Create an OIDC trust in the cloud provider**

The trust should specify which repositories are allowed to obtain tokens, as well as any additional information necessary to increase security.

Done in the cloud provider

3 **Exchange GitHub's OIDC token for credentials**

There are several actions from trusted providers that implement this exchange process, we can simply use them.

4 **Use the short-lived credential to manage resources**

The short-lived credentials are valid only for a single job, and expire after that. If we need to use credentials in other jobs, we can simply reuse the third-party actions to obtain new short-lived credentials.

Done in the GitHub Workflow

GitHub Actions

Workflow Billing

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