PassQueen
a.k.a dumpsqueen.com

GitHub Actions Exam

Microsoft GH-200

Version Demo

Total Demo Questions: 14

Total Premium Questions: 144

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Author and maintain workflows	73
Topic 2, Consume workflows	6
Topic 3, Author and maintain actions	31
Topic 4, Manage GitHub Actions for a repository	24
Topic 5, Manage GitHub Actions for an enterprise	10
Total	144

QUESTION NO: 1

You are a DevOps engineer configuring a GitHub Actions deployment workflow. Ensure that the `deploy` job runs only when the workflow is triggered from the branch named `feature-branch`. Which YAML snippet correctly implements this job-level condition?

A.

```
jobs:
  deploy:
    if: github.ref_name == 'feature-branch'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```

B.

```
jobs:
  deploy:
    if: github.ref.name == 'feature-branch'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```

C.

```
jobs:
  deploy:
    if: github.branch_name == 'feature-branch'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```

D.

```
jobs:
  deploy:
    if: github.branch.name == 'feature-branch'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```

ANSWER: A

Explanation:

The `github.ref_name` context property is the appropriate value to use when a workflow needs the short name of the Git ref that triggered the run. For a branch-triggered workflow, it contains the branch name without the `refs/heads/` prefix. Therefore, when the workflow runs from `feature-branch`, the expression `github.ref_name == 'feature-branch'` evaluates to true and GitHub Actions schedules the `deploy` job. When the triggering branch has any other name, the expression evaluates to false and the job is skipped. Job-level `if` conditions are evaluated before the job runs, making this a suitable way to prevent deployment infrastructure and steps from executing for branches that are not intended to deploy. GitHub documents `github.ref_name` as the short ref name, and documents that the `jobs.<job_id>.if` key can use expressions to control whether a job executes. See the GitHub context reference and the workflow syntax reference for job conditions.

QUESTION NO: 2

As a developer using a self-hosted Linux runner, you need Redis available during a workflow job. Which approach lets the job use Redis without leaving Redis installed or configured in a way that affects subsequent workflow runs?

A. Add a run step to your workflow, which dynamically installs and configures Redis as part of your job.

B. Specify `services:` in your job definition to leverage a Redis service container.

C. Set up Redis on a separate machine and reference that instance from your job.

D. Install Redis on the hosted runner image and place it in a runner group. Specify label: in your job to target the runner group.

ANSWER: B

Explanation:

Specify `services:` in the job definition to run Redis as a service container. A service container supplies a temporary dependency for the lifetime of a job, which is well suited to integration tests and workflows that require a cache, database, or message broker. The workflow can configure a Redis image, expose the Redis port, and optionally define health checks so that job steps begin only after the service is ready. When the job finishes, GitHub Actions stops and removes the service container, avoiding a persistent Redis installation or configuration on the self-hosted runner that could influence a later run.

For jobs that run directly on the runner, the Redis service is reachable through `localhost` on its mapped port. A job-level `container:` is optional; if the job itself runs in a container, it can reach the Redis service by using the service label as the hostname. Self-hosted runners must have Docker installed and operational to use container-based services. GitHub documents service-container configuration under the `jobs.<job_id>.services` key and describes the networking behavior for both runner and container jobs. See About service containers and Workflow syntax for services.

QUESTION NO: 3

In which two situations is it most useful to temporarily disable a GitHub Actions workflow so that it no longer triggers, while keeping its workflow file in the repository? (Choose two.)

A. A workflow sends requests to a service that is down.

B. A workflow error produces too many, or wrong, requests, impacting external services negatively.

C. A workflow is configured to run on self-hosted runners

D. A workflow needs to be changed from running on a schedule to a manual trigger

E. A runner needs to have diagnostic logging enabled.

ANSWER: A B

Explanation:

Temporarily disabling a workflow is appropriate when continued automated runs could create unnecessary failures, traffic, or operational impact, but the workflow configuration should be retained for later use. A workflow sends requests to a service that is down is a clear example: disabling the workflow stops future event-driven or scheduled executions from repeatedly calling an unavailable dependency. This avoids noisy failed runs and prevents avoidable load while the service is being restored.

A workflow error produces too many, or wrong, requests, impacting external services negatively is also an important use case. Disabling the workflow acts as an immediate safeguard against further unintended requests, invalid updates, or excessive consumption of an external service. Maintainers can then investigate and correct the workflow without deleting its YAML definition or losing its configuration. Once the dependency is available again or the workflow issue has been fixed and validated, the workflow can be re-enabled. GitHub documents that disabling a workflow prevents it from running while preserving the workflow file and its history/configuration for subsequent re-enablement. See Disable and enable a workflow and Triggering a workflow.

QUESTION NO: 4

As a developer, which three steps must a GitHub Actions workflow perform to publish a container image to the GitHub Container Registry?

A. Use the actions/setup-docker action

B. Authenticate to the GitHub Container Registry.

C. Build the container image.

D. Push the image to the GitHub Container Registry

E. Pull the image from the GitHub Container Registry.

ANSWER: B C D

Explanation:

Publishing a container image from GitHub Actions to GitHub Container Registry involves authenticating the workflow, building the image, and uploading it under a valid GHCR image name. The workflow authenticates to `ghcr.io`, commonly through `docker/login-action` with `github.actor` and `secrets.GITHUB_TOKEN`. The workflow token must be granted package write access, typically through `permissions: packages: write`, so it can create or update the container package. It then builds the container image from the repository's Dockerfile and build context. Docker's build tooling or `docker/build-push-action` can be used for this operation. Finally, it pushes the tagged image to a GHCR destination such as `ghcr.io/OWNER/IMAGE:TAG`. A build-and-push action may combine image construction and upload into one YAML step, but both are required publication operations. GitHub's documented workflow for publishing Docker images follows this same sequence: log in to the registry, generate appropriate metadata or tags, and build and push the image. For implementation details, see GitHub's Publishing Docker images documentation and the GitHub Container Registry documentation.

QUESTION NO: 5

Which GitHub Actions custom action type lets you package multiple workflow steps into one reusable action?

A. Composite action

B. Bash script action

C. Docker container action

D. JavaScript action

ANSWER: A

Explanation:

Composite action is the correct choice because GitHub Actions provides composite actions specifically to combine multiple workflow steps into a single reusable custom action. A composite action is described in an `action.yml` metadata file, where `runs.using` is set to `composite` and the action defines a sequence of `steps`. Each bundled step can run shell commands or call another action, enabling teams to place common automation—such as tool installation, environment setup, builds, validation, or standardized deployment preparation—behind one reusable `uses` reference in a workflow.

Composite actions can also accept inputs and expose outputs. This means the same grouped logic can be parameterized by a calling workflow and can provide values for downstream workflow steps. Keeping the repeated sequence in a versioned action directory or repository improves consistency across workflows while reducing duplicated YAML. GitHub's documentation explicitly describes composite actions as a way to combine several workflow steps within an action. For implementation details, see Creating a composite action and About custom actions.

QUESTION NO: 6

A DevOps engineer references an organization secret from a repository workflow, but the value available to the workflow differs from the value configured for the organization secret. What is the most likely cause?

A. There is a different value specified at the repository level.

B. There is a different value specified at the workflow level.

C. The Codespace secret doesn ' t match the expected value.

D. The Encrypt Secret setting was not configured for the secret.

E. There is a different value specified at the enterprise level.

ANSWER: A

Explanation:

There is a different value specified at the repository level. GitHub Actions supports secrets at several scopes, including enterprise, organization, repository, and environment. When secrets at multiple scopes have the same name, GitHub applies a defined precedence order. A repository secret takes precedence over an organization secret with an identical name, so a workflow expression such as `${{ secrets.SECRET_NAME }}` resolves to the repository-scoped secret value. This is useful when an organization provides a shared default credential while a particular repository needs a separate credential for its own service, deployment target, or integration.

To confirm the source of the unexpected value, review the repository's **Settings** area under **Secrets and variables** for Actions and look for a secret whose name matches the organization secret. If the organization-wide secret should be used, remove the repository-level override or rename the secrets so that their intended scope is unambiguous. GitHub documents the precedence rule as repository secrets over organization secrets, with organization secrets taking precedence over enterprise secrets. See GitHub Docs: Naming your secrets and GitHub Docs: Creating secrets for an organization.

QUESTION NO: 7

Which metadata file for a custom GitHub Action defines how the action runs, including its main entry point?

A. action.js

B. index.js

C. action.yml

D. main.yml

ANSWER: C

Explanation:

`action.yml` is the standard metadata file that GitHub Actions uses to describe a custom action and define how it is executed. The file is placed in the root directory of the action repository or in the directory that contains the action. Its required `runs` section specifies the execution model and operational entry point. For example, a JavaScript action uses `runs.using` to select a supported Node.js runtime and `runs.main` to identify the JavaScript file GitHub should execute. This lets a workflow invoke the action through a `uses` reference without needing to know the action's internal implementation details. The same metadata file also provides the action's display name, description, inputs, outputs, branding, and optional pre- or post-execution behavior. GitHub also accepts the equivalent filename `action.yaml`; however, among the listed filenames, `action.yml` is the correct metadata file. The official metadata syntax reference describes the `runs` section and its `main` property at GitHub Actions metadata syntax. A practical implementation example is available in GitHub's Creating a JavaScript action documentation.

QUESTION NO: 8

How should an encrypted GitHub Actions secret be used in an `if:` conditional for a step within a workflow job?

A. Set the encrypted secret as a job-level environment variable and then reference the environment variable within the conditional statement.

B. Create a job dependency that exposes the encrypted secret as a job output, which can then be leveraged in a subsequent dependent job.

C. Use the secrets context within the conditional statement, e.g. ${{ secrets.MySuperSecret }}.

D. Use a workflow command to expose the encrypted secret via a step ' s output parameter and then use the step output in the job ' s if: conditional.

ANSWER: A

Explanation:

Set the encrypted secret as a job-level environment variable and then reference the environment variable within the conditional. GitHub Actions does not evaluate the `secrets` context directly in `if:` conditionals. Instead, assign the secret to an environment variable at the job level, for example `env: SUPER_SECRET: ${{ secrets.MySuperSecret }}`, and test the environment variable in a step condition, such as `if: ${{ env.SUPER_SECRET != '' }}`. This allows the workflow to determine whether the secret was provided without placing the secret value directly in the condition. It is particularly useful when a workflow can run in contexts where a secret may be unavailable, including certain pull-request workflows originating from forks; unavailable secrets resolve to an empty string. The secret remains handled as an environment variable by the runner, and GitHub masks configured secret values in logs. Workflows should still avoid printing the variable or passing it through commands in ways that could expose transformed values. GitHub documents this environment-variable pattern specifically for conditional logic involving secrets in its Using secrets in GitHub Actions guidance. For expression and conditional syntax, see the GitHub Actions expression documentation.

QUESTION NO: 9

An organization has an organization-level configuration variable named `DEPLOY_REGION`, which is overridden by a repository-level variable of the same name. The `production` environment also defines `DEPLOY_REGION` with a production-specific value.

A deployment job must use the production-specific value. Which two statements are true? Select two.

A. The environment-level value takes precedence over the repository-level and organization-level values.

B. The job must reference the production environment for its environment-level variable to be available.

C. The organization-level value takes precedence because it has the broadest scope.

D. The workflow must access the value only through `secrets.DEPLOY_REGION`.

E. A repository-level variable always overrides an environment-level variable with the same name.

ANSWER: A B

Explanation:

GitHub Actions configuration variables have a defined precedence hierarchy when the same variable name is set at multiple scopes. An environment-level configuration variable has the highest precedence, followed by a repository-level variable and then an organization-level variable. Therefore, for a job targeting the production environment, the production value of `DEPLOY_REGION` is the value available through the `vars` context, such as `${{ vars.DEPLOY_REGION }}`. This allows one workflow to use a common variable name while receiving the appropriate value for each deployment target.

The job must explicitly reference the production environment, for example with `environment: production`. Environment configuration variables are available only to jobs that reference that environment; declaring a variable at the environment scope alone does not expose it to unrelated jobs. Referencing the environment also enables the environment's deployment protections and scoped configuration before the job proceeds. GitHub documents both the configuration-variable precedence order and the requirement that environment variables are available only after the job references the environment. See GitHub Docs: configuration variable precedence and GitHub Docs: defining configuration variables.

QUESTION NO: 10

You are configuring the `Node CI/CD` GitHub Actions workflow shown below. Which two code blocks can be added beginning at line 5 to configure the workflow to run in response to multiple event types? Each correct answer is a complete solution.

4 name: Node CI/CD

5

6

7

8 jobs:

9 build:

10 runs-on:

11 steps:

12 - uses: actions/checkout@v2

13 - uses: actions/setup-node@v1

14 with:

15 node-version: 12

16 - run: npm ci

17 - run: npm test

18

A. on:env:- ' prod '- ' qa '- ' test '

B. on: [push, commit]

C. on: [push, pull_request]

D. on:branches:- ' main '- ' dev '

E. on:push:branches:- mainrelease:types:- created

F. on:schedule:- cron: ' */15 * * * * 'initiate:- ' main '

ANSWER: C E

Explanation:

`on: [push, pull_request]` is valid GitHub Actions workflow syntax for declaring multiple triggering events in a compact sequence. It causes the workflow to run for pushes and for pull-request activity using each event's default activity types. This form is appropriate when neither trigger requires additional filtering or event-specific configuration, and it belongs at the workflow's top level before `jobs`.

The block beginning with `on:`, containing a configured `push` event and a configured `release` event, is also valid. GitHub Actions supports a mapping under `on` in which each event can have its own configuration. In this case, the workflow runs for pushes to the `main` branch and when a release is created. Branch filters are configured beneath the relevant `push` event, while release activity filters are configured with `types` beneath `release`. This independently configured mapping is the required structure when workflow triggers need conditions such as branch filters or selected event activity types. GitHub documents both the sequence form and the multi-event mapping form in the workflow syntax reference. The available release activity types, including `created`, are documented in the release event reference.

QUESTION NO: 11

Which locations can a GitHub Actions workflow reference when using an action? Choose three.

[IMAGE_1]

A. a separate public repository

B. an .action extension file in the repository

C. the same repository as the workflow

D. a published Docker container image on Docker Hub

E. the runs-on: keyword of a workflow file

F. the repository's Secrets settings page

G. a public NPM registry

ANSWER: A C D

Explanation:

GitHub Actions workflows can reference reusable actions from a separate public repository by using the `OWNER/REPOSITORY@REF` syntax. This lets a workflow consume an action maintained independently, with a tag, branch, or commit SHA identifying the version to use. A workflow can also use an action stored in the same repository as the workflow. For a local action, the workflow supplies a relative path, such as `./.github/actions/my-action`; the repository must be checked out before that local path can be used. In addition, an action may be supplied as a Docker container image published on Docker Hub. The workflow references this image with the `docker://` prefix, for example `docker://alpine:3.8`. These supported locations enable teams to choose between sharing actions across repositories, keeping repository-specific actions alongside their workflows, and packaging action runtime dependencies into a container image. GitHub documents these action-reference forms in its workflow syntax and action creation guidance. See GitHub Actions workflow syntax for uses and About custom actions.

QUESTION NO: 12

You have configured a self-hosted GitHub Actions runner with the custom label `runner1`.

You must ensure that a workflow job is assigned only to that runner.

Which YAML statement should you specify for the job?

A. runs-on: [self-hosted, linux-xlarge]

B. runs-on: - self-hosted - ubuntu-latest

C. runs-on: [self-hosted, runner1]

D. runs-on: self-hosted

ANSWER: C

Explanation:

`runs-on: [self-hosted, runner1]` is the appropriate statement because the `runs-on` key determines the runner on which a GitHub Actions job executes. When `runs-on` contains an array of labels, GitHub selects a runner only when it matches every label in that array. Self-hosted runners are automatically assigned the `self-hosted` label, while `runner1` is the custom label used to target the required machine. Consequently, this configuration requires an eligible runner to be both self-hosted and labeled `runner1`. To ensure the job truly runs only on the intended machine, assign the custom `runner1` label exclusively to that runner. This targeting pattern is useful when a job needs a particular runner because it has specialized hardware, installed tools, network connectivity, or another machine-specific capability. GitHub documents that self-hosted jobs can be routed using one label or an array of labels, and that a runner must match all labels supplied in the array. For implementation details, see Using self-hosted runners in a workflow and the runs-on workflow syntax reference.

QUESTION NO: 13

Which two advantages does a matrix strategy provide when configured in a GitHub Actions job definition? Choose two.

A. It can test code in multiple versions of a programming language.

B. It can decrease the costs for running multiple combinations of programming language/operating systems.

C. It can run up to 512 jobs per workflow run.

D. It can test code in multiple operating systems.

ANSWER: A D

Explanation:

“It can test code in multiple versions of a programming language.” is an advantage because a matrix defines variables with multiple values, and GitHub Actions automatically creates a separate job for each resulting matrix combination. A workflow can, for example, supply several supported versions of Node.js, Python, Java, or .NET, then execute the same build and test steps for every version. This gives a project repeatable compatibility validation without requiring a separately authored job definition for each runtime version. It also helps identify failures that occur only with a particular supported language or runtime release.

“It can test code in multiple operating systems.” is also an advantage because operating-system runner labels can be a matrix dimension. A matrix can include values such as `ubuntu-latest`, `windows-latest`, and `macos-latest`, and the matrix value can be used in the job’s `runs-on` property. GitHub Actions expands these values into independent jobs, allowing the same workflow logic to validate an application across all supported platforms. Combining an operating-system dimension with a language-version dimension provides broad environment coverage from a concise, maintainable workflow definition. GitHub documents matrix expansion and using matrix values in job configuration in Using a matrix for your jobs and the workflow syntax reference.

QUESTION NO: 14

A team is building a composite action to standardize linting commands across multiple repositories. Which two statements correctly describe composite actions? Select two.

A. A composite action can run multiple shell commands and action steps as one workflow step.

B. A composite action can define inputs and outputs in its metadata file.

C. A composite action can declare its own `runs-on` value to select a runner.

D. A composite action can define job-level `permissions` for its caller.

E. A composite action requires a Dockerfile to define its runtime.

ANSWER: A B

Explanation:

`A composite action can run multiple shell commands and action steps as one workflow step.` This is a core purpose of composite actions: their `action.yml` metadata contains a `runs` section with `using: composite` and a sequence of steps. Those steps can invoke shell commands with `run` or call other actions with `uses`. A consuming workflow then references the composite action from one workflow step, allowing teams to package and consistently reuse a linting sequence without duplicating its individual commands in every repository.

`A composite action can define inputs and outputs in its metadata file.` The action metadata syntax supports an `inputs` map for caller-supplied configuration, such as a language version or linting target. It also supports an `outputs` map, enabling the action to expose values produced by its internal steps to later workflow steps. This makes a composite action configurable while retaining a single, reusable interface for workflows that consume it. GitHub documents composite-action structure and the supported `runs.steps` syntax in its Creating a composite action guide, and documents the metadata fields, including inputs and outputs, in Metadata syntax for GitHub Actions.