DUMPSBOSS.

GitHub Actions Certificate Exam

GitHub GitHub-Actions

Version Demo

Total Demo Questions: 8

Total Premium Questions: 81

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Author and maintain workflows	52
Topic 2, Consume workflows	5
Topic 3, Author and maintain actions	10
Topic 4, Manage GitHub Actions in the enterprise	13
Topic 5, Mix Questions	1
Total	81

QUESTION NO: 1

Which default environment variable specifies the branch or tag that triggered a workflow?

A. GITHUB_TAG

B. GITHUB_REF

C. ENV_BRANCH

D. GITHUB_BRANCH

ANSWER: B

Explanation:

`GITHUB_REF` is the default GitHub Actions environment variable that contains the fully formed Git ref for the branch or tag that triggered the workflow. For example, a workflow run triggered by a push to the `main` branch receives a value such as `refs/heads/main`; a run triggered by a tag named `v1.0.0` receives `refs/tags/v1.0.0`. This makes the variable useful when a workflow needs to identify the triggering source ref, such as when selecting a deployment target, applying branch-specific behavior, or recording the source revision in build metadata. The value is available automatically to every workflow run, so it does not need to be defined in the workflow's `env` section. GitHub also exposes this information through the `github.ref` context, which corresponds to the same triggering ref and can be used directly in expressions. The exact format and availability can vary with the event type; GitHub documents event-specific behavior, including special cases for pull request workflows. See the official default environment variables reference and the GitHub context reference.

QUESTION NO: 2

A deployment workflow uses OIDC to obtain short-lived cloud credentials. Which two measures reduce the risk that another workflow can obtain credentials intended for production? (Choose two.)

A. Grant `id-token: write` only to the job that performs the deployment.

B. Restrict the cloud trust policy to token subjects for the intended repository and production environment or branch.

C. Set `permissions: write-all` at the workflow level.

D. Store a permanent cloud access key as a repository secret for fallback authentication.

E. Use a workflow concurrency group named `production`.

ANSWER: A B

Explanation:

Granting `id-token: write` only to the deployment job limits which job can request an OIDC token. The cloud provider's trust policy should also restrict the token subject to the intended repository and production context, such as a protected environment or an approved branch. These controls reduce the set of workflows and token claims that can satisfy the cloud trust policy.

Granting broad token permissions to every job increases exposure. Storing cloud credentials as repository secrets abandons the short-lived credential model, and a concurrency group controls execution overlap rather than cloud identity authorization.

QUESTION NO: 3

A workflow in `octo-org/app` must check out a second private repository, `octo-org/shared-config`. The checkout step uses `${{ secrets.GITHUB_TOKEN }}` and receives a repository-not-found error, although the workflow token has `contents: read` permission. What is the appropriate solution?

A. Use a GitHub App installation token or other credential with read access to `octo-org/shared-config`.

B. Add `contents: write` to the permissions for `GITHUB_TOKEN`.

C. Set `persist-credentials: true` on the first checkout step.

D. Move the workflow file to the organization's `.github` repository.

ANSWER: A

Explanation:

The automatically generated `GITHUB_TOKEN` is scoped to the repository that contains the workflow. Granting it `contents: read` allows read access to `octo-org/app`, but it does not automatically grant access to a separate private repository such as `octo-org/shared-config`.

Use a GitHub App installation token, a fine-grained personal access token, or another credential that has been explicitly granted read access to the second repository, and provide that credential to the checkout action. Increasing the `GITHUB_TOKEN` permissions does not extend its repository scope.

QUESTION NO: 4

Which default GitHub environment variable indicates the owner and repository name?

A. REPOSITORY NAME

B. GITHUB REPOSITORY

C. ENV REPOSITORY

D. GITHUB WORKFLOW REPO

E. GITHUB_REPOSITORY

ANSWER: E

Explanation:

`GITHUB_REPOSITORY` is the default GitHub Actions environment variable that identifies the repository associated with a workflow run. Its value uses the fully qualified `owner/repository` format, such as `octo-org/octo-repo`. The owner portion is the user or organization that owns the repository, and the repository portion is its name. GitHub automatically sets this variable for every step in a workflow, so it can be referenced directly by commands running in a job, for example with `$GITHUB_REPOSITORY` in a shell step. This makes it useful when a workflow needs to pass the current repository identifier to a GitHub CLI command, an API request, a deployment script, or another action without hard-coding the repository name. GitHub documents `GITHUB_REPOSITORY` among the default environment variables and specifies that it contains the owner and repository name. The variable name must use underscores exactly as shown; environment variable names are not written with spaces. See GitHub's default environment variables reference and its guide to using variables in workflows.

QUESTION NO: 5

Which default GitHub environment variable indicates the name of the person or app that initiated a workflow?

A. ENV_ACTOR

B. GITHUB_WORKFLOW_ACTOR

C. GITHUB_ACTOR

D. GITHUB_USER

ANSWER: C

Explanation:

`GITHUB_ACTOR` is the default environment variable that contains the name of the person or app that initiated a workflow run. GitHub automatically sets it for every workflow, so steps can use it without defining the variable in the workflow YAML. Its value is the account login associated with the triggering actor, such as a GitHub username or the name of a GitHub App. This is useful for logging, conditional workflow behavior, audit-related automation, or passing the initiating actor to scripts and external systems. For example, a shell step can access the value as `$GITHUB_ACTOR`, while an expression can use the related `github.actor` context property. GitHub distinguishes the initiating actor from the actor that may re-run a workflow: `GITHUB_ACTOR` retains the original triggering actor for a re-run, whereas `GITHUB_TRIGGERING_ACTOR` identifies the user who initiated that re-run. The documented default-variable reference explicitly defines `GITHUB_ACTOR` as the name of the person or app that initiated the workflow. See GitHub's default environment variables reference and its github context documentation.

QUESTION NO: 6

As a developer, you need to use GitHub Actions to deploy a microservice that requires runtime access to a secure token. This token is used by a variety of other microservices managed by different teams in different repos. To minimize management overhead and ensure the token is secure, which mechanisms should you use to store and access the token? (Choose two.)

A. Store the token in a configuration file in a private repository. Use GitHub Actions to deploy the configuration file to the runtime environment.

B. Store the token as a GitHub encrypted secret in the same repo as the code. Create a reusable custom GitHub Action to access the token by the microservice at runtime.

C. Use a corporate non-GitHub secret store (e.g., HashiCorp Vault) to store the token. During deployment, use GitHub Actions to store the secret in an environment variable that can be accessed at runtime.

D. Store the token as a GitHub encrypted secret in the same repo as the code. During deployment, use GitHub Actions to store the secret in an environment variable that can be accessed at runtime.

E. Store the token as an organizational-level encrypted secret in GitHub. During deployment, use GitHub Actions to store the secret in an environment variable that can be accessed at runtime.

ANSWER: C E

Explanation:

Using a corporate non-GitHub secret store (e.g., HashiCorp Vault) to store the token provides centralized ownership, rotation, auditing, and access controls for a credential used by services across teams and repositories. A deployment workflow can retrieve the value only when needed and pass it to the deployed workload as a runtime environment variable. This approach avoids maintaining duplicate copies of the token in individual source repositories while keeping secret-management responsibilities in the organization's dedicated secret-management system. GitHub Actions supports securely consuming secrets in workflows, and GitHub recommends masking and avoiding unnecessary exposure of secret values in logs; see Using secrets in GitHub Actions.

Storing the token as an organizational-level encrypted secret in GitHub also fits a shared-token scenario. Organization secrets can be made available to selected repositories, enabling teams to use one centrally managed secret instead of separately defining the same value in every repository. The workflow can supply that secret during deployment as an environment variable for the service's runtime configuration. Repository access policies on the organization secret allow administrators to limit which repositories can use it, supporting both lower administrative overhead and least-privilege access. GitHub documents organization-secret creation and repository access policies at Creating secrets for an organization.

QUESTION NO: 7

As a developer, which workflow steps should you perform to publish an image to the GitHub Container Registry? (Choose three.)

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 to GitHub Container Registry requires authenticating the workflow, building the image, and pushing its tagged image reference to the registry. “Authenticate to the GitHub Container Registry.” is required so Docker is authorized to write packages; in GitHub Actions, this is commonly done with `docker/login-action` using `GITHUB_TOKEN` and appropriate package-write permissions. “Build the container image.” creates the Docker or OCI image artifact that will be published, typically through `docker build` or Docker’s build-push action. “Push the image to the GitHub Container Registry” uploads the built, appropriately tagged image to the `ghcr.io` registry namespace. A workflow can use Docker setup or build tooling where needed, but GitHub’s publishing process fundamentally requires credentials with package write access, an image build, and a registry push. GitHub’s package publishing documentation shows this sequence and explains the required `packages: write` permission for workflows that publish container images. See GitHub’s Container registry documentation and Publishing Docker images with GitHub Actions.

QUESTION NO: 8

Which statements about `defaults.run` in a GitHub Actions workflow are true? (Choose two.)

A. Defaults can be defined at the workflow level for all `run` steps.

B. Defaults can provide a common `working-directory` for `run` steps in one job.

C. Expressions such as `${{ github.workspace }}` can be used in `defaults.run.working-directory`.

D. Defaults automatically apply to inputs passed to actions with the `uses` keyword.

E. A step-level `defaults.run` mapping can override the workflow default for one step.

ANSWER: A B

Explanation:

A workflow can define default `shell` and `working-directory` values for all `run` steps, or define them more narrowly for the `run` steps in a particular job. Job-level defaults take precedence over workflow-level defaults when both are present.

Expressions and contexts cannot be used in `defaults.run`. In addition, these defaults apply only to steps that use `run`; they do not supply inputs or execution settings for a step that uses an action through `uses`.