GitHub Actions OIDC Federation Across Clouds: AWS, GCP, and Azure Without Long-Lived CI Secrets

Machine Identity / Last updated 2026-06-11 / https://www.scrambleid.com/learn/github-actions-oidc-federation-across-clouds

In one sentence: GitHub Actions OIDC federation eliminates long-lived cloud credentials from CI/CD workflows by issuing short-lived job-scoped OIDC tokens that AWS, GCP, and Azure can exchange for short-lived cloud credentials, removing the entire class of "leaked CI secret" breaches that has driven multiple high-profile incidents in 2022-2024.

TL;DR (canonical)

- **The problem:** GitHub Secrets containing long-lived AWS/GCP/Azure keys leak (compromised dependencies, malicious workflows, insider misuse, GitHub-side incidents). Once leaked, they're valid until manually rotated.
- **The replacement:** GitHub Actions OIDC. Each workflow run can request a job-scoped OIDC token from GitHub's identity provider. Clouds trust GitHub's issuer and exchange the token for short-lived cloud credentials.
- **Three configurations to know:** AWS (IAM with web-identity federation, configured trust policy with subject-claim conditions), GCP (Workload Identity Federation provider for GitHub), Azure (federated identity credentials on an Entra app registration).
- **The single most important security control:** subject-claim conditions in the trust policy. Restrict to specific branches and environments, never allow pull-request workflows to assume production roles.
- **Production posture:** environment protection rules in GitHub require approval for workflows targeting protected environments. Combined with subject-claim restrictions, this gives reviewable, scoped, short-lived cloud access from CI.
- **ScrambleID composes:** for workflows that need to call services beyond the cloud's native IAM, exchange the cloud credential for a ScrambleID-issued sender-constrained token.

Why long-lived CI secrets are the wrong default

GitHub Secrets are convenient. Drop an AWS access key in. Reference it from your workflow. Done. The pattern works.

Until it doesn't:

- **Compromised dependencies** in workflow actions (Marketplace actions, custom actions in third-party repos) can read secrets in scope.
- **Malicious pull-requests** in some configurations can exfiltrate secrets.
- **Insider misuse** by anyone with admin access to the repository.
- **GitHub-side incidents** (rare but documented).
- **Backup compromise** of organization-level secrets.

Once the key is leaked, it's valid until manually rotated, and it works from anywhere on the internet. The CircleCI 2023 incident is one well-known example of this attack class; many others have not been publicly disclosed.

The replacement architecture eliminates the long-lived credential entirely. The workflow gets credentials only when it runs, scoped only to what that run needs.

How GitHub Actions OIDC works

The flow:

1. The workflow declares `permissions: id-token: write` at the workflow or job level.
2. When the workflow runs, GitHub's identity provider issues an OIDC token to the runner.
3. The token contains claims about the workflow context:
    - `iss`: GitHub's OIDC issuer URL (`https://token.actions.githubusercontent.com`)
    - `sub`: a subject claim with format like `repo:owner/repo-name:ref:refs/heads/main` or `repo:owner/repo-name:environment:production`
    - `aud`: the audience (configurable per cloud)
    - Various other claims: `repository`, `repository_owner`, `workflow`, `job_workflow_ref`, `actor`, `event_name`, `head_ref`, `base_ref`, etc.
4. The workflow uses this token to authenticate to the cloud, exchanging the token for short-lived cloud credentials.
5. The credentials expire when the job ends (or sooner, depending on cloud configuration).

The cloud-side trust policy validates:

- The token's `iss` matches GitHub's OIDC issuer.
- The token's `sub` matches a configured pattern (specific branches, environments, etc.).

- The token's `aud` matches the configured audience.
- The token's signature verifies against GitHub's published JWKS.

If all checks pass, the cloud issues short-lived credentials.

Configuration: AWS

AWS uses IAM identity providers and roles configured for web-identity federation.

Step 1: register GitHub's OIDC issuer in AWS IAM

```
aws iam create-open-id-connect-provider \
  --url https://token.actions.githubusercontent.com \
  --client-id-list sts.amazonaws.com \
  --thumbprint-list <github-thumbprint>
```

The thumbprint can be retrieved from GitHub's documentation; AWS-side tooling can validate.

Step 2: create an IAM role with a trust policy

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": {
      "Federated": "arn:aws:iam::123456789012:oidc-provider/token.actions.githubusercontent.com"
    },
    "Action": "sts:AssumeRoleWithWebIdentity",
    "Condition": {
      "StringEquals": {
        "token.actions.githubusercontent.com:aud": "sts.amazonaws.com"
      },
      "StringLike": {
        "token.actions.githubusercontent.com:sub": "repo:my-org/my-repo:ref:refs/heads/main"
      }
    }
  }]
}
```

The `sub` condition is the critical security control. The example above allows only the `main` branch in `my-org/my-repo` to assume this role.

For environment-scoped access:

```
"StringLike": {
  "token.actions.githubusercontent.com:sub": "repo:my-org/my-repo:environment:production"
}
```

For pull-request workflows (rarely appropriate for production roles):

```
"StringLike": {
  "token.actions.githubusercontent.com:sub": "repo:my-org/my-repo:pull_request"
}
```

Step 3: configure the workflow (AWS)

```yaml
name: Deploy
on:
  push:
    branches: [main]

permissions:
  id-token: write
  contents: read

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/MyDeployRole
          aws-region: us-east-1

      - name: Deploy
        run: |
          aws s3 sync ./build/ s3://my-bucket/
```

The `aws-actions/configure-aws-credentials` action handles the OIDC token exchange transparently.

Configuration: GCP

GCP uses Workload Identity Federation with a workload identity pool.

Step 1: create a Workload Identity Pool and provider

```
# Create the pool
gcloud iam workload-identity-pools create github-pool \
  --location global \
  --display-name "GitHub Actions Pool"

# Add a provider for GitHub
gcloud iam workload-identity-pools providers create-oidc github-provider \
  --location global \
  --workload-identity-pool github-pool \
  --display-name "GitHub Actions Provider" \
  --attribute-mapping
"google.subject=assertion.sub,attribute.repository=assertion.repository,attribute.repository_owner=a
ssertion.repository_owner" \
  --attribute-condition "assertion.repository_owner=='my-org'" \
  --issuer-uri https://token.actions.githubusercontent.com
```

The `attribute-condition` is the GCP equivalent of AWS's subject condition. The example above allows only repositories owned by `my-org`. For tighter scoping:

```
--attribute-condition "assertion.repository=='my-org/my-repo' && assertion.ref=='refs/heads/main'"
```

Step 2: bind a service account to the pool

```
# Allow the workflow to impersonate a GCP service account
gcloud iam service-accounts add-iam-policy-binding \
  my-sa@my-project.iam.gserviceaccount.com \
  --role roles/iam.workloadIdentityUser \
  --member "principalSet://iam.googleapis.com/projects/<project-
number>/locations/global/workloadIdentityPools/github-pool/attribute.repository/my-org/my-repo"
```

Step 3: configure the workflow (GCP)

```
name: Deploy to GCP
on:
  push:
    branches: [main]

permissions:
  id-token: write
  contents: read

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Authenticate to Google Cloud
        uses: google-github-actions/auth@v2
        with:
          workload_identity_provider: 'projects/<project-
number>/locations/global/workloadIdentityPools/github-pool/providers/github-provider'
          service_account: 'my-sa@my-project.iam.gserviceaccount.com'

      - name: Deploy
        run: |
          gcloud storage cp ./build/ gs://my-bucket/ --recursive
```

Configuration: Azure

Azure uses federated identity credentials on a Microsoft Entra app registration or managed identity.

Step 1: create an app registration (or use a managed identity)

```
az ad app create --display-name github-deploy
APP_ID=$(az ad app list --display-name github-deploy --query [0].appId -o tsv)
az ad sp create --id $APP_ID
SP_ID=$(az ad sp show --id $APP_ID --query id -o tsv)
```

Step 2: add federated identity credentials

```
# Federated credential for main-branch deployments
az ad app federated-credential create \
  --id $APP_ID \
  --parameters '{
    "name": "main-branch",
    "issuer": "https://token.actions.githubusercontent.com",
    "subject": "repo:my-org/my-repo:ref:refs/heads/main",
    "audiences": ["api://AzureADTokenExchange"]
  }'

# Federated credential for production environment
az ad app federated-credential create \
  --id $APP_ID \
  --parameters '{
    "name": "production-env",
    "issuer": "https://token.actions.githubusercontent.com",
    "subject": "repo:my-org/my-repo:environment:production",
    "audiences": ["api://AzureADTokenExchange"]
  }'
```

Step 3: grant the app registration appropriate Azure RBAC

```
az role assignment create \
  --assignee $SP_ID \
  --role "Contributor" \
  --scope /subscriptions/<sub-id>/resourceGroups/my-rg
```

Step 4: configure the workflow

```
name: Deploy to Azure
on:
  push:
    branches: [main]

permissions:
  id-token: write
  contents: read

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Azure Login
        uses: azure/login@v2
        with:
          client-id: ${{ vars.AZURE_CLIENT_ID }}
          tenant-id: ${{ vars.AZURE_TENANT_ID }}
          subscription-id: ${{ vars.AZURE_SUBSCRIPTION_ID }}

      - name: Deploy
        run: |
          az webapp deployment source config-zip \
            --resource-group my-rg \
            --name my-app \
            --src ./build.zip
```

Security pitfalls and how to avoid them

Pitfall 1: overly permissive subject conditions

The most common misconfiguration. A trust policy that allows any branch, any pull request, or any workflow in the repo to assume the role gives the cloud access to anyone who can run a workflow.

Mitigation: restrict to specific branches (typically `main` for production), specific environments (combined with environment protection rules), and explicitly exclude pull-request workflows from

production-grade roles.

Pitfall 2: pull-request workflows accessing production

In many orgs, pull requests automatically run CI workflows. If those workflows can assume production-grade cloud roles, anyone who can open a pull request (often: anyone with read access to the repo, or the public, for open-source projects) can execute against production.

Mitigation: never allow pull-request workflows to assume production-grade roles. Use environment protection rules with required approvers.

Pitfall 3: missing environment protection rules

GitHub environments (e.g., `production`, `staging`) can have protection rules requiring approval before deployment workflows can run. Without these, any push to `main` deploys to production.

Mitigation: configure environment protection rules with required reviewers for sensitive environments.

Pitfall 4: re-usable workflow injection

A workflow that can be called from another repository can inherit the calling repo's permissions. If the called workflow assumes a powerful role, the calling repo can use it.

Mitigation: scope `job_workflow_ref` claims in trust policies to specific workflow versions; require called workflows to come from a specific organization.

Pitfall 5: actor-based trust without considering bots

Some configurations trust based on `actor` (the user who triggered the workflow). GitHub bots (Dependabot, GitHub-app-driven workflows) have their own actor names and may have different security properties.

Mitigation: explicitly enumerate which actors can trigger production-grade workflows; don't use catch-all conditions.

Pitfall 6: long token TTLs

The OIDC token itself is short-lived, but the credentials it's exchanged for can be configured with various TTLs. Default 1-hour AWS credentials are fine for most workflows; longer TTLs (extending into post-job life) are unnecessary and increase exposure.

Mitigation: configure cloud credential TTL to match the workflow's actual needs (typically minutes), not the maximum allowed.

Pitfall 7: forgetting to rotate the OIDC thumbprint

GitHub's OIDC issuer rotates its signing keys periodically. AWS-stored thumbprints become stale. Workflows fail.

Mitigation: automate thumbprint rotation, or use AWS's automatic thumbprint validation features (newer AWS configurations don't require manual thumbprints).

Cross-cloud workflows

A single workflow can authenticate to multiple clouds:

```yaml
permissions:
  id-token: write
  contents: read

jobs:
  multi-cloud-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Authenticate to AWS
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/MyAWSRole
          aws-region: us-east-1

      - name: Authenticate to GCP
        uses: google-github-actions/auth@v2
        with:
          workload_identity_provider: 'projects/.../providers/github-provider'
          service_account: 'my-sa@my-project.iam.gserviceaccount.com'

      - name: Authenticate to Azure
        uses: azure/login@v2
        with:
          client-id: ${{ vars.AZURE_CLIENT_ID }}
          tenant-id: ${{ vars.AZURE_TENANT_ID }}
          subscription-id: ${{ vars.AZURE_SUBSCRIPTION_ID }}

      - name: Deploy across clouds
        run: |
          # Use AWS, GCP, and Azure CLI tools
```

Each cloud's trust relationship is configured independently. The workflow obtains credentials from each.

ScrambleID and GitHub Actions OIDC

The cloud-native pattern handles workflow-to-cloud-IAM. For workflows that need to call services beyond the cloud's native IAM (cross-cloud, cross-tenant SaaS APIs, AI agent platforms, internal services using ScrambleID for authentication), the workflow exchanges its cloud credential for a ScrambleID-issued token:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      # Get cloud credentials via OIDC
      - name: AWS auth
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/MyDeployRole

      # Use cloud credentials to obtain ScrambleID token via JWT client assertion
      - name: Get ScrambleID token
        id: scrambleid-token
        run: |
          ASSERTION=$(./scripts/build-jwt-assertion.sh)  # signs with workload-bound key
          ACCESS_TOKEN=$(curl -X POST https://id.scrambleid.com/oauth2/token \
            -d "grant_type=client_credentials" \
            -d "client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer" \
            -d "client_assertion=$ASSERTION" \
            -d "scope=deploy:write" | jq -r .access_token)
          echo "::add-mask::$ACCESS_TOKEN"
          echo "access_token=$ACCESS_TOKEN" >> $GITHUB_OUTPUT

      # Use the ScrambleID token to call internal services
      - name: Call internal API
        run: |
          curl -X POST https://internal.example.com/deploy \
            -H "Authorization: Bearer ${{ steps.scrambleid-token.outputs.access_token }}" \
            --cert path/to/client.crt --key path/to/client.key
```

The workflow has cloud-native short-lived cloud credentials and ScrambleID-issued sender-constrained tokens for everything else. No long-lived secrets anywhere.

Operational considerations

Audit and monitoring. Cloud-side audit logs (CloudTrail, GCP Audit Logs, Azure Activity Log) capture every credential issuance via OIDC. Monitor for unexpected `AssumeRoleWithWebIdentity` calls, unusual subject patterns, and token issuance rate anomalies.

Rotation. OIDC federation eliminates rotation for the cloud credentials. The trust relationship itself (the trust policy, attribute conditions, federated credentials) needs occasional review. Schedule quarterly review of trust policies.

Migration sequence. Existing workflows with long-lived secrets should migrate one at a time. Stand up the OIDC trust relationship in parallel, switch the workflow to use OIDC, validate, then revoke the old credential. Don't cut over without validation.

Testing. OIDC federation is harder to test in isolation than long-lived credentials. Consider a staging cloud account or sandbox role for testing changes to trust policies before applying to production.

Multi-org governance. Organizations with multiple GitHub orgs and clouds need a policy framework for which orgs can federate to which clouds. Centralize the trust-policy management where possible.

Standards alignment

Standard	Relevance
OIDC	The federation protocol
OAuth 2.0	Token exchange semantics
RFC 7519	JWT format for the OIDC token
AWS IAM web identity federation	AWS-specific implementation
GCP Workload Identity Federation	GCP-specific implementation
Microsoft Entra federated credentials	Azure-specific implementation
RFC 7523	JWT client assertion (used in ScrambleID composition)
RFC 9700	OAuth 2.0 Security BCP
NIST SP 800-207	Zero Trust applied to CI/CD

Key Takeaway

GitHub Actions OIDC federation eliminates long-lived cloud credentials from CI/CD workflows by issuing short-lived, job-scoped OIDC tokens that AWS, GCP, and Azure can exchange for short-lived cloud credentials. Configuration is per-cloud: AWS uses IAM web-identity federation with a configured trust policy and subject-claim conditions; GCP uses Workload Identity Federation with a workload identity pool and provider; Azure uses federated identity credentials on an Entra app registration. The single most important security control is subject-claim conditions in the trust policy: restrict to specific branches and environments, never allow pull-request workflows to assume production roles. Combined with environment protection rules in GitHub (required reviewers for sensitive deployments), this gives reviewable, scoped, short-lived cloud access from CI. ScrambleID composes with the cloud-native pattern: workflows that need to call services beyond the cloud's native IAM exchange the cloud credential for a ScrambleID-issued sender-constrained token via RFC 7523 JWT client assertion. The result is no persistent secrets anywhere in the CI/CD path.

FAQ

What is GitHub Actions OIDC?

GitHub Actions OIDC is a feature where every workflow run can request a short-lived OIDC token from GitHub's identity provider. The token contains claims about the workflow context: the repository, the branch, the environment, the workflow file, the actor who triggered the run. Cloud providers (AWS, GCP, Azure) can be configured to trust GitHub's OIDC issuer and accept these tokens for short-lived cloud credentials, eliminating the need to store long-lived cloud keys as GitHub secrets.

Why is GitHub Actions OIDC better than storing AWS keys as GitHub Secrets?

GitHub Secrets containing long-lived AWS access keys have several failure modes: secrets can be exfiltrated by malicious workflows in compromised dependencies, by insider misuse, or by GitHub-side incidents (rare but possible). Once exfiltrated, the keys are valid until manually rotated and can be replayed from anywhere. OIDC federation issues credentials at job execution that are scoped to the specific job and expire when the job ends. There is no persistent secret to leak.

How do I scope GitHub Actions OIDC tokens to specific branches or environments?

Subject-claim conditions in the cloud's trust policy. The OIDC token includes a `sub` claim with the format `repo:owner/name:ref:refs/heads/branch-name`, `repo:owner/name:environment:env-name`, or `repo:owner/name:pull_request`. The cloud's trust policy can restrict which subject patterns are allowed to assume the role. For example, AWS IAM trust policies can require `sub` matches a specific branch, environment, or pull-request pattern. This is the most important security control: a permissive subject condition gives any workflow in the repo access to the cloud role.

Can a malicious pull request use my OIDC trust to access AWS?

If your trust policy allows pull-request workflows to assume the role, yes. This is the most common GitHub Actions OIDC misconfiguration. The mitigation: restrict subject conditions to specific branches (typically `main`) or environments (with deployment approval gates), not pull-request workflows. Use the environment-protection-rules feature in GitHub to require approval before workflows targeting protected environments can run. Pull-request workflows should not be allowed to assume production-grade cloud roles.

Does this work with self-hosted runners?

Yes. The OIDC token is issued by GitHub's identity provider regardless of where the runner is hosted. Self-hosted runners receive the same OIDC tokens as GitHub-hosted runners. The trust policy at the cloud doesn't distinguish between runner hosting; it validates the OIDC token's issuer and claims. Self-hosted runners do introduce additional considerations (the runner host's compromise can replay tokens during the job window), but the federation mechanism itself is unchanged.

How does ScrambleID compose with GitHub Actions OIDC?

GitHub Actions OIDC handles workflow-to-cloud-IAM authentication. For workflows that need to call services beyond the cloud's native IAM (cross-cloud, cross-tenant SaaS APIs, AI agent platforms, internal services using ScrambleID for authentication), the workflow uses its cloud-native credential to obtain a ScrambleID-issued token via JWT client assertion (RFC 7523), bound via mTLS or DPoP. The pattern preserves the no-persistent-secrets property end-to-end while extending it across the boundaries that cloud-native IAM doesn't cover.

References (public)

- GitHub: Configuring OpenID Connect in GitHub Actions: https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/about-security-hardening-with-openid-connect
- GitHub: OIDC in AWS: https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/configuring-openid-connect-in-amazon-web-services
- GitHub: OIDC in GCP: https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/configuring-openid-connect-in-google-cloud-platform
- GitHub: OIDC in Azure: https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/configuring-openid-connect-in-azure
- AWS: Web identity federation: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_create_for-idp_oidc.html
- GCP: Workload Identity Federation with GitHub Actions: https://cloud.google.com/iam/docs/workload-identity-federation-with-other-providers

- Microsoft Entra: Federated Identity Credentials: https://learn.microsoft.com/en-us/entra/workload-id/workload-identity-federation
- RFC 7523 (JWT Profile for OAuth 2.0): https://datatracker.ietf.org/doc/html/rfc7523
- NIST SP 800-207 (Zero Trust): https://csrc.nist.gov/publications/detail/sp/800-207/final

---

Related reading

- Service Account Replacement
- Cloud Workload Identity Compared
- M2M Authentication Without Secrets
- Sender-Constrained Tokens (mTLS, DPoP)
- client_secret vs JWT Client Assertion vs mTLS
- What Is Non-Human Identity (NHI)?