GitHub Actions: The Complete Interview Guide (Beginner to Advanced)

A comprehensive guide to mastering GitHub Actions for DevOps interviews and real-world projects

---

Introduction

GitHub Actions has revolutionized the way we approach CI/CD in modern software development. Whether you're preparing for a DevOps interview or looking to level up your automation skills, understanding GitHub Actions is no longer optional—it's essential.

This guide takes you from absolute basics to advanced enterprise-level concepts, structured exactly how interview questions progress: from fundamentals to complex scenarios.

Let's dive in!

Part 1: Fundamentals (Must Know)

What is GitHub Actions?

GitHub Actions is a CI/CD platform that allows you to automate your build, test, and deployment pipeline directly within GitHub. Think of it as your personal automation assistant that lives in your repository.

Why GitHub Actions? - Integrated directly into GitHub - No need for external CI/CD tools - Free for public repositories - Massive marketplace of pre-built actions - Native support for Docker and Kubernetes

CI vs CD Concepts

Continuous Integration (CI) - Automatically build and test code when changes are pushed - Catch bugs early - Ensure code quality - Run tests, linting, and security checks

Continuous Deployment (CD) - Automatically deploy code to production/staging - Reduce manual deployment errors - Faster release cycles - Rollback capabilities

Example: When you push code → CI runs tests → If tests pass → CD deploys to staging → Manual approval → Deploy to production

Core Components Explained

1. Workflow A workflow is an automated process defined in a YAML file. It's like a recipe that tells GitHub Actions what to do.

Location: `.github/workflows/your-workflow.yml`

```yaml
name: My First Workflow
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Say Hello
        run: echo "Hello, GitHub Actions!"
```

2. Event Events are triggers that start your workflow. Common events: - `push` - When code is pushed - `pull_request` - When a PR is opened/updated - `schedule` - Run on a schedule (cron) - `workflow_dispatch` - Manual trigger

3. Job A job is a set of steps that execute on the same runner. Multiple jobs can run in parallel or sequentially.

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Build application
        run: npm run build

  test:
    runs-on: ubuntu-latest
    needs: build  # This job waits for 'build' to complete
    steps:
      - name: Run tests
        run: npm test
```

4. Step A step is an individual task within a job. It can either: - Run a command (`run:`) - Use an action (`uses:`)

```yaml
steps:
  - name: Checkout code
    uses: actions/checkout@v3

  - name: Install dependencies
    run: npm install
```

5. Runner A runner is a server that executes your workflows.

GitHub-hosted Runners - Managed by GitHub - Fresh VM for each job - Pre-installed software - Available: Ubuntu, Windows, macOS

Self-hosted Runners - You manage the infrastructure - Can customize environment - Use your own hardware - Better for enterprise needs

6. YAML Syntax Basics YAML is all about indentation (use 2 spaces, not tabs!):

```yaml
# This is a comment
name: My Workflow          # String
on: [push, pull_request]   # Array
jobs:                      # Object
  build:                   # Nested object
    runs-on: ubuntu-latest
    steps:
      - name: Step 1
        run: echo "Hello"
```

7. `on:` Triggers The `on` keyword defines what events trigger your workflow:

```yaml
# Single event
on: push

# Multiple events
on: [push, pull_request]

# Detailed configuration
on:
  push:
    branches:
      - main
      - develop
  pull_request:
    branches:
      - main
```

8. `uses:` vs `run:` **`uses:`** - Use a pre-built action from the marketplace

```yaml
- uses: actions/checkout@v3
- uses: actions/setup-node@v3
  with:
    node-version: '18'
```

`run:` - Execute shell commands

```yaml
- run: npm install
- run: |
    echo "Multi-line command"
    npm test
    npm run build
```

9. Marketplace Actions The GitHub Marketplace has thousands of pre-built actions:

Popular Actions: - `actions/checkout@v3` - Clone your repository - `actions/setup-node@v3` - Setup Node.js - `docker/build-push-action@v3` - Build and push Docker images - `aws-actions/configure-aws-credentials@v1` - Configure AWS credentials

```yaml
steps:
  - uses: actions/checkout@v3

  - uses: actions/setup-python@v4
    with:
      python-version: '3.11'

  - uses: aws-actions/configure-aws-credentials@v1
    with:
      aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
      aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      aws-region: us-east-1
```

Part 2: Workflow & Triggers (Deep Dive)

Push Trigger

Triggered when code is pushed to specific branches:

```yaml
on:
  push:
    branches:
      - main
      - 'release/**'  # Matches release/v1, release/v2, etc.
    tags:
      - 'v*'  # Matches v1.0, v2.0, etc.
```

Pull Request Trigger

Triggered on PR events:

```yaml
on:
  pull_request:
```

```yaml
    types:
      - opened
      - synchronize  # New commits pushed
      - reopened
    branches:
      - main
```

Schedule (Cron Jobs)

Run workflows on a schedule:

```yaml
on:
  schedule:
    # Run every day at 2:30 AM UTC
    - cron: '30 2 * * *'
    # Run every Monday at 8:00 AM UTC
    - cron: '0 8 * * 1'
```

Cron syntax: `minute hour day month weekday`

Manual Trigger (workflow_dispatch)

Allow manual workflow runs with custom inputs:

```yaml
on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment to deploy to'
        required: true
        type: choice
        options:
          - staging
          - production
      debug:
        description: 'Enable debug mode'
        required: false
        type: boolean
        default: false

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to ${{ inputs.environment }}
        run: echo "Deploying to ${{ inputs.environment }}"
```

Path Filters

Only trigger when specific files change:

```yaml
on:
  push:
    paths:
      - 'src/**'           # Any file in src directory
      - '**.js'            # Any JavaScript file
      - '!docs/**'         # Exclude docs directory
    paths-ignore:
      - '**.md'            # Ignore markdown files
```

Multiple Event Triggers

Combine multiple triggers:

```yaml
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
  schedule:
    - cron: '0 0 * * 0'  # Weekly on Sunday
  workflow_dispatch:
```

Part 3: Jobs & Steps (Orchestration)

Multiple Jobs

Jobs run in parallel by default:

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Building..."

  test:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Testing..."

  lint:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Linting..."
```

Job Dependencies (needs)

Control job execution order:

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - run: npm run build

  test:
    runs-on: ubuntu-latest
    needs: build  # Wait for build to complete
    steps:
      - run: npm test

  deploy:
    runs-on: ubuntu-latest
    needs: [build, test]  # Wait for both
    steps:
      - run: npm run deploy
```

Flow: build → test → deploy

Matrix Strategy

Run the same job with different configurations:

```yaml
jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        node-version: [14, 16, 18]
        # This creates 9 jobs (3 OS × 3 Node versions)
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
      - run: npm test
```

Advanced Matrix:

```yaml
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest]
    node: [14, 16, 18]
```

```yaml
    include:
      # Add specific combinations
      - os: ubuntu-latest
        node: 18
        experimental: true
    exclude:
      # Remove specific combinations
      - os: windows-latest
        node: 14
```

Fail-Fast Behavior

```yaml
strategy:
  fail-fast: false  # Continue running other matrix jobs even if one fails
  matrix:
    os: [ubuntu-latest, windows-latest, macos-latest]
```

Continue-on-Error

Allow a step to fail without failing the entire job:

```yaml
steps:
  - name: Run tests
    run: npm test
    continue-on-error: true  # Job continues even if tests fail

  - name: Upload results
    run: upload-results.sh
```

Timeout Settings

Prevent jobs from running too long:

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    timeout-minutes: 30  # Job fails if it runs longer than 30 minutes
    steps:
      - name: Build application
        run: npm run build
        timeout-minutes: 10  # Step-level timeout
```

Part 4: Runners (Infrastructure)

GitHub-hosted Runners

Available Images: - `ubuntu-latest` (Ubuntu 22.04) - `ubuntu-20.04` - `windows-latest` (Windows Server 2022) - `windows-2019` - `macos-latest` (macOS 12) - `macos-11`

Pre-installed Software: - Git, Docker, Node.js, Python, Java - Cloud CLIs (AWS, Azure, GCP) - Build tools

```yaml
jobs:
  build:
    runs-on: ubuntu-latest  # Most common choice
    steps:
      - run: docker --version
      - run: node --version
```

Self-hosted Runners

When to use: - Need specific hardware (GPU, specialized CPU) - Access to internal network resources - Compliance requirements - Cost optimization for heavy usage

Setup: 1. Go to Settings → Actions → Runners 2. Click "New self-hosted runner" 3. Follow installation instructions 4. Runner appears as "self-hosted"

```yaml
jobs:
  build:
    runs-on: self-hosted  # Use your own runner
    steps:
      - run: echo "Running on my infrastructure"
```

Runner Labels

Organize and target specific runners:

```yaml
jobs:
  build:
    runs-on: [self-hosted, linux, x64, gpu]  # Multiple labels
    steps:
      - run: nvidia-smi  # GPU command
```

Scaling Self-hosted Runners

Options: - Manual: Add more runners manually - Auto-scaling: Use tools like Terraform, Kubernetes - Runner groups: Organize runners for different teams

Security Considerations

Best Practices: - Don't use self-hosted runners for public repositories (security risk) - Isolate runners (use ephemeral runners) - Limit runner access to specific repositories - Use separate runners for different environments - Regular security updates

---

Part 5: Secrets & Security (Critical!)

Repository Secrets

Store sensitive data like API keys, passwords:

Creating Secrets: 1. Go to Settings → Secrets and variables → Actions 2. Click "New repository secret" 3. Add name and value

Using Secrets:

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to server
        env:
          API_KEY: ${{ secrets.API_KEY }}
          DB_PASSWORD: ${{ secrets.DB_PASSWORD }}
        run: |
          echo "API_KEY is safely stored"
          deploy.sh
```

Organization Secrets

Share secrets across multiple repositories:

```yaml
steps:
  - name: Use org secret
    env:
      ORG_TOKEN: ${{ secrets.ORG_TOKEN }}
    run: echo "Using organization-level secret"
```

Environment Secrets

Scope secrets to specific environments (staging, production):

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production  # Use 'production' environment
```

```yaml
    steps:
      - name: Deploy
        env:
          PROD_API_KEY: ${{ secrets.PROD_API_KEY }}
        run: deploy.sh
```

Masking Secrets

GitHub automatically masks secrets in logs:

```yaml
steps:
  - run: echo "${{ secrets.API_KEY }}"  # Will show *** in logs
```

Add custom masking:

```yaml
steps:
  - name: Mask custom value
    run: echo "::add-mask::my-secret-value"
  - run: echo "my-secret-value"  # Will show *** in logs
```

OIDC Authentication (Modern Approach)

OpenID Connect eliminates the need for long-lived credentials:

Why OIDC? - No static credentials stored - Temporary, short-lived tokens - Better security posture - Native AWS/Azure/GCP support

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    permissions:
      id-token: write  # Required for OIDC
      contents: read
    steps:
      - uses: aws-actions/configure-aws-credentials@v1
        with:
          role-to-assume: arn:aws:iam::123456789012:role/GitHubActionsRole
          aws-region: us-east-1

      - run: aws s3 ls  # No access keys needed!
```

GitHub → AWS IAM Integration

Setup:

1. **Create OIDC Identity Provider in AWS**:
   - Provider URL: `https://token.actions.githubusercontent.com`
   - Audience: `sts.amazonaws.com`
2. **Create IAM Role**:

- Trust policy allows GitHub Actions to assume role
- Attach permissions policy

3. **Use in Workflow**:

```
- uses: aws-actions/configure-aws-credentials@v1
  with:
    role-to-assume: arn:aws:iam::123456789012:role/GitHubActionsRole
    aws-region: us-east-1

- run: |
    aws s3 cp ./build s3://my-bucket/
    aws ecs update-service --cluster my-cluster --service my-service
```

Preventing Secret Leaks

Best Practices:

1. Never print secrets:

```
- run: echo "$API_KEY"  # BAD! Even with masking
- run: curl -H "Authorization: $API_KEY" api.example.com  # GOOD
```

2. Use environment variables:

```
env:
  API_KEY: ${{ secrets.API_KEY }}
steps:
  - run: ./script.sh  # Access via $API_KEY in script
```

3. Rotate secrets regularly
4. Use minimal permissions
5. Audit secret usage

Part 6: Artifacts & Caching (Performance Optimization)

Upload Artifacts

Save files from your workflow (build outputs, test results):

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: npm run build

      - name: Upload build artifacts
```

```yaml
        uses: actions/upload-artifact@v3
        with:
          name: build-output
          path: dist/
          retention-days: 30  # Keep for 30 days
```

Download Artifacts

Use artifacts in subsequent jobs:

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - run: npm run build
      - uses: actions/upload-artifact@v3
        with:
          name: dist-files
          path: dist/

  deploy:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - uses: actions/download-artifact@v3
        with:
          name: dist-files
          path: ./dist

      - run: ls -la ./dist
      - run: deploy.sh
```

Cache Dependencies

Speed up workflows by caching dependencies:

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Cache node modules
        uses: actions/cache@v3
        with:
          path: ~/.npm
          key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
```

```
          restore-keys: |
            ${{ runner.os }}-node-

      - run: npm ci
      - run: npm test
```

Cache Key Strategy

Good Cache Keys:

```
# Include OS and hash of dependency file
key: ${{ runner.os }}-npm-${{ hashFiles('**/package-lock.json') }}

# For multiple dependencies
key: ${{ runner.os }}-${{ hashFiles('**/pom.xml', '**/build.gradle') }}

# Include version
key: v1-${{ runner.os }}-npm-${{ hashFiles('**/package-lock.json') }}
```

Cache Restore Keys

Fallback when exact match not found:

```
- uses: actions/cache@v3
  with:
    path: ~/.npm
    key: ${{ runner.os }}-npm-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      ${{ runner.os }}-npm-
      ${{ runner.os }}-
```

How it works: 1. Try exact match: `ubuntu-npm-abc123` 2. If not found, try: `ubuntu-npm-` 3. If not found, try: `ubuntu-` 4. If not found, no cache

Retention Policies

Default Retention: - Artifacts: 90 days (configurable) - Caches: 7 days or 10 GB limit

Configure retention:

```
- uses: actions/upload-artifact@v3
  with:
    name: my-artifact
    path: ./output
    retention-days: 7  # Override default
```

Part 7: CI Pipeline Concepts

Build Automation

Automate compilation and bundling:

```yaml
name: Build Pipeline

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - uses: actions/setup-node@v3
        with:
          node-version: '18'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Build application
        run: npm run build

      - name: Upload build
        uses: actions/upload-artifact@v3
        with:
          name: production-build
          path: dist/
```

Running Unit Tests

```yaml
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '18'

      - run: npm ci
      - run: npm test
```

```yaml
      - name: Upload test results
        if: always()  # Upload even if tests fail
        uses: actions/upload-artifact@v3
        with:
          name: test-results
          path: test-results/
```

Code Coverage

Track test coverage:

```yaml
jobs:
  coverage:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '18'

      - run: npm ci
      - run: npm run test:coverage

      - name: Upload coverage to Codecov
        uses: codecov/codecov-action@v3
        with:
          files: ./coverage/lcov.info
          fail_ci_if_error: true
```

Linting

Enforce code quality standards:

```yaml
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '18'

      - run: npm ci

      - name: Run ESLint
        run: npm run lint
```

```
      - name: Run Prettier check
        run: npx prettier --check .
```

Static Code Analysis

```
jobs:
  analyze:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: SonarCloud Scan
        uses: SonarSource/sonarcloud-github-action@master
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
```

Docker Build in GitHub Actions

```
jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and push
        uses: docker/build-push-action@v4
        with:
          context: .
          push: true
          tags: myapp:${{ github.sha }},myapp:latest
          cache-from: type=registry,ref=myapp:buildcache
          cache-to: type=registry,ref=myapp:buildcache,mode=max
```

Multi-stage Pipeline

Complete CI pipeline:

```yaml
name: Complete CI Pipeline

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '18'
      - run: npm ci
      - run: npm run lint

  test:
    runs-on: ubuntu-latest
    needs: lint
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '18'
      - run: npm ci
      - run: npm test

  build:
    runs-on: ubuntu-latest
    needs: test
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '18'
      - run: npm ci
      - run: npm run build
      - uses: actions/upload-artifact@v3
        with:
          name: dist
          path: dist/

  security:
```

```yaml
    runs-on: ubuntu-latest
    needs: build
    steps:
      - uses: actions/checkout@v3
      - run: npm audit
      - uses: snyk/actions/node@master
        env:
          SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}
```

Part 8: Docker & Containers

Using Docker in Workflows

```yaml
jobs:
  docker-job:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Build Docker image
        run: docker build -t myapp:test .

      - name: Run tests in container
        run: docker run myapp:test npm test
```

Container Jobs

Run entire job in a container:

```yaml
jobs:
  container-job:
    runs-on: ubuntu-latest
    container:
      image: node:18
      env:
        NODE_ENV: development
      options: --cpus 2 --memory 2g

    steps:
      - uses: actions/checkout@v3
      - run: npm ci
      - run: npm test
```

Service Containers

Add databases and services:

```yaml
jobs:
  integration-test:
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres:14
        env:
          POSTGRES_PASSWORD: postgres
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
        ports:
          - 5432:5432

      redis:
        image: redis:7
        options: >-
          --health-cmd "redis-cli ping"
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
        ports:
          - 6379:6379

    steps:
      - uses: actions/checkout@v3

      - name: Run integration tests
        env:
          DATABASE_URL: postgresql://postgres:postgres@localhost:5432/test
          REDIS_URL: redis://localhost:6379
        run: npm run test:integration
```

Docker Build & Push

Multi-platform builds:

```yaml
jobs:
  docker-build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```

```yaml
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v2

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and push multi-platform
        uses: docker/build-push-action@v4
        with:
          context: .
          platforms: linux/amd64,linux/arm64
          push: true
          tags: |
            myapp:latest
            myapp:${{ github.sha }}
            myapp:${{ github.ref_name }}
          build-args: |
            VERSION=${{ github.sha }}
            BUILD_DATE=${{ github.event.head_commit.timestamp }}
```

Login to Docker Hub / ECR

Docker Hub:

```yaml
- uses: docker/login-action@v2
  with:
    username: ${{ secrets.DOCKER_USERNAME }}
    password: ${{ secrets.DOCKER_PASSWORD }}
```

Amazon ECR:

```yaml
- name: Login to Amazon ECR
  uses: aws-actions/amazon-ecr-login@v1

- name: Build and push to ECR
  env:
    ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
    ECR_REPOSITORY: my-app
    IMAGE_TAG: ${{ github.sha }}
  run: |
    docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
```

```
docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
```

Part 9: Cloud Deployments (DevOps Essential)

Deploy to AWS EC2

Using SSH:

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Deploy to EC2
        env:
          PRIVATE_KEY: ${{ secrets.EC2_SSH_KEY }}
          HOST: ${{ secrets.EC2_HOST }}
          USER: ubuntu
        run: |
          echo "$PRIVATE_KEY" > private_key.pem
          chmod 600 private_key.pem
          ssh -o StrictHostKeyChecking=no -i private_key.pem ${USER}@${HOST} '
            cd /var/www/myapp
            git pull origin main
            npm install
            pm2 restart myapp
          '
```

Deploy to ECS / EKS

ECS Deployment:

```yaml
jobs:
  deploy-ecs:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1

      - name: Login to Amazon ECR
```

```
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@v1

      - name: Build and push image
        env:
          ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
          ECR_REPOSITORY: my-app
          IMAGE_TAG: ${{ github.sha }}
        run: |
          docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
          docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG

      - name: Update ECS service
        run: |
          aws ecs update-service \
            --cluster my-cluster \
            --service my-service \
            --force-new-deployment
```

EKS Deployment:

```
jobs:
  deploy-eks:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - uses: aws-actions/configure-aws-credentials@v1
        with:
          role-to-assume: arn:aws:iam::123456789012:role/GitHubActionsRole
          aws-region: us-east-1

      - name: Update kubeconfig
        run: aws eks update-kubeconfig --name my-cluster --region us-east-1

      - name: Deploy to Kubernetes
        run: |
          kubectl set image deployment/myapp \
            myapp=123456789012.dkr.ecr.us-east-1.amazonaws.com/myapp:${{ github.sha }}
          kubectl rollout status deployment/myapp
```

Deploy to S3

Static website hosting:

```
jobs:
  deploy-s3:
```

```yaml
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - uses: actions/setup-node@v3
        with:
          node-version: '18'

      - run: npm ci
      - run: npm run build

      - uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1

      - name: Sync to S3
        run: |
          aws s3 sync ./dist s3://my-website-bucket --delete
          aws cloudfront create-invalidation \
            --distribution-id ABCDEFGHIJK \
            --paths "/*"
```

Deploy to Kubernetes

```yaml
jobs:
  deploy-k8s:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up kubectl
        uses: azure/setup-kubectl@v3

      - name: Configure kubectl
        env:
          KUBECONFIG_DATA: ${{ secrets.KUBECONFIG }}
        run: |
          echo "$KUBECONFIG_DATA" | base64 -d > kubeconfig
          export KUBECONFIG=kubeconfig

      - name: Deploy application
        run: |
          kubectl apply -f k8s/deployment.yaml
          kubectl apply -f k8s/service.yaml
```

```
          kubectl rollout status deployment/myapp -n production
```

Blue-Green Deployment

```yaml
jobs:
  blue-green-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Deploy to Green environment
        run: |
          # Deploy new version to green
          kubectl apply -f k8s/deployment-green.yaml
          kubectl wait --for=condition=ready pod -l app=myapp,environment=green

      - name: Run smoke tests
        run: |
          ./smoke-tests.sh green-service-url

      - name: Switch traffic to Green
        run: |
          # Update service selector to point to green
          kubectl patch service myapp -p '{"spec":{"selector":{"environment":"green"}}}'

      - name: Cleanup Blue environment
        run: |
          kubectl delete deployment myapp-blue
```

Rolling Deployment

```yaml
jobs:
  rolling-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Update deployment
        run: |
          kubectl set image deployment/myapp \
            myapp=myapp:${{ github.sha }} \
            --record

          kubectl rollout status deployment/myapp

      - name: Verify deployment
```

```
        run: |
          if ! ./health-check.sh; then
            echo "Health check failed, rolling back"
            kubectl rollout undo deployment/myapp
            exit 1
          fi
```

Part 10: Reusable Workflows & Advanced Features

Reusable Workflows

Create a reusable workflow (`.github/workflows/reusable-deploy.yml`):

```yaml
name: Reusable Deploy Workflow

on:
  workflow_call:
    inputs:
      environment:
        required: true
        type: string
      version:
        required: true
        type: string
    secrets:
      deploy-token:
        required: true
    outputs:
      deployment-url:
        description: "URL of the deployment"
        value: ${{ jobs.deploy.outputs.url }}

jobs:
  deploy:
    runs-on: ubuntu-latest
    outputs:
      url: ${{ steps.deploy.outputs.url }}
    steps:
      - name: Deploy to ${{ inputs.environment }}
        id: deploy
        env:
          TOKEN: ${{ secrets.deploy-token }}
        run: |
          echo "Deploying version ${{ inputs.version }} to ${{ inputs.environment }}"
          echo "url=https://${{ inputs.environment }}.myapp.com" >> $GITHUB_OUTPUT
```

Call the reusable workflow:

```yaml
jobs:
  deploy-staging:
    uses: ./.github/workflows/reusable-deploy.yml
    with:
      environment: staging
      version: ${{ github.sha }}
    secrets:
      deploy-token: ${{ secrets.STAGING_TOKEN }}

  deploy-production:
    needs: deploy-staging
    uses: ./.github/workflows/reusable-deploy.yml
    with:
      environment: production
      version: ${{ github.sha }}
    secrets:
      deploy-token: ${{ secrets.PROD_TOKEN }}
```

Composite Actions

Create custom action (`.github/actions/setup-app/action.yml`):

```yaml
name: 'Setup Application'
description: 'Install dependencies and configure app'

inputs:
  node-version:
    description: 'Node.js version'
    required: false
    default: '18'
  cache-key:
    description: 'Cache key suffix'
    required: false
    default: 'default'

outputs:
  cache-hit:
    description: 'Whether cache was hit'
    value: ${{ steps.cache.outputs.cache-hit }}

runs:
  using: 'composite'
  steps:
    - uses: actions/setup-node@v3
      with:
```

```
        node-version: ${{ inputs.node-version }}

    - name: Cache dependencies
      id: cache
      uses: actions/cache@v3
      with:
        path: ~/.npm
        key: ${{ runner.os }}-npm-${{ inputs.cache-key }}

    - name: Install dependencies
      shell: bash
      run: npm ci
```

Use composite action:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - uses: ./.github/actions/setup-app
        with:
          node-version: '18'
          cache-key: ${{ hashFiles('**/package-lock.json') }}

      - run: npm run build
```

Custom Actions (JavaScript)

Create JavaScript action (`.github/actions/hello/action.yml`):

```
name: 'Hello Action'
description: 'Greet someone'
inputs:
  who-to-greet:
    description: 'Who to greet'
    required: true
    default: 'World'
outputs:
  time:
    description: 'The time we greeted you'
runs:
  using: 'node16'
  main: 'index.js'
```

JavaScript file (`.github/actions/hello/index.js`):

```javascript
const core = require('@actions/core');
const github = require('@actions/github');

try {
  const nameToGreet = core.getInput('who-to-greet');
  console.log(`Hello ${nameToGreet}!`);

  const time = (new Date()).toTimeString();
  core.setOutput("time", time);

  const payload = JSON.stringify(github.context.payload, undefined, 2);
  console.log(`The event payload: ${payload}`);
} catch (error) {
  core.setFailed(error.message);
}
```

Environment Protection Rules

Configure in Settings → Environments:

```yaml
jobs:
  deploy-production:
    runs-on: ubuntu-latest
    environment:
      name: production
      url: https://myapp.com
    steps:
      - name: Deploy
        run: echo "Deploying to production"
```

Protection rules: - Required reviewers - Wait timer (delay deployment) - Environment secrets - Deployment branches (only from main)

Approval Gates

```yaml
jobs:
  deploy-staging:
    runs-on: ubuntu-latest
    environment: staging
    steps:
      - run: deploy-to-staging.sh

  approval:
    runs-on: ubuntu-latest
    needs: deploy-staging
    environment: production-approval  # Requires manual approval
    steps:
```

```
      - run: echo "Approved for production"

  deploy-production:
    runs-on: ubuntu-latest
    needs: approval
    environment: production
    steps:
      - run: deploy-to-production.sh
```

Concurrency Control

Prevent multiple deployments:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    concurrency:
      group: production-deployment
      cancel-in-progress: false  # Wait for previous to finish
    steps:
      - run: deploy.sh
```

Per-branch concurrency:

```
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true  # Cancel old runs on new push
```

Workflow Permissions

Fine-grained permissions:

```
name: Secure Workflow

permissions:
  contents: read       # Read repository contents
  issues: write        # Write to issues
  pull-requests: write # Write to PRs
  packages: write      # Publish packages

jobs:
  build:
    runs-on: ubuntu-latest
    permissions:
      contents: read  # Job-level override
      packages: write
    steps:
```

```yaml
      - uses: actions/checkout@v3
      - run: npm publish
```

Disable all permissions:

```yaml
permissions: {}
```

Part 11: Monitoring & Debugging

Debug Logging

Enable detailed logs:

Method 1: Set repository secret `ACTIONS_STEP_DEBUG` = `true`

Method 2: Add debug statements:

```yaml
steps:
  - name: Debug information
    run: |
      echo "::debug::This is a debug message"
      echo "::notice::This is a notice"
      echo "::warning::This is a warning"
      echo "::error::This is an error"
```

Step Output Variables

Pass data between steps:

```yaml
jobs:
  example:
    runs-on: ubuntu-latest
    steps:
      - name: Set outputs
        id: vars
        run: |
          echo "sha_short=$(git rev-parse --short HEAD)" >> $GITHUB_OUTPUT
          echo "branch=${GITHUB_REF#refs/heads/}" >> $GITHUB_OUTPUT
          echo "date=$(date +'%Y-%m-%d')" >> $GITHUB_OUTPUT

      - name: Use outputs
        run: |
          echo "Short SHA: ${{ steps.vars.outputs.sha_short }}"
          echo "Branch: ${{ steps.vars.outputs.branch }}"
          echo "Date: ${{ steps.vars.outputs.date }}"
```

Job Outputs

Share data between jobs:

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    outputs:
      version: ${{ steps.get-version.outputs.version }}
      image-tag: ${{ steps.build.outputs.tag }}
    steps:
      - id: get-version
        run: echo "version=1.0.0" >> $GITHUB_OUTPUT

      - id: build
        run: echo "tag=myapp:${{ github.sha }}" >> $GITHUB_OUTPUT

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - run: |
          echo "Deploying version: ${{ needs.build.outputs.version }}"
          echo "Using image: ${{ needs.build.outputs.image-tag }}"
```

Workflow Run Logs

Viewing logs: 1. Go to Actions tab 2. Click on workflow run 3. Click on job 4. Expand steps

Grouping logs:

```yaml
steps:
  - name: Complex operation
    run: |
      echo "::group::Installing dependencies"
      npm install
      echo "::endgroup::"

      echo "::group::Running tests"
      npm test
      echo "::endgroup::"
```

Re-run Failed Jobs

```yaml
jobs:
  flaky-test:
    runs-on: ubuntu-latest
```

```
    steps:
      - run: npm test
        continue-on-error: true

      - name: Retry on failure
        if: failure()
        run: npm test
```

In UI: Click "Re-run failed jobs" button

GitHub Actions Usage Metrics

Track usage in organization settings:

- Workflow run times
- Artifact storage
- Minutes used
- Cost tracking

Get usage via API:

```
curl -H "Authorization: token $GITHUB_TOKEN" \
  https://api.github.com/repos/OWNER/REPO/actions/workflows
```

Part 12: Enterprise-Level Topics

Branch Protection Rules

Configure in Settings → Branches:

Required settings for production: - Require pull request reviews (2+ approvers) - Require status checks to pass - Require conversation resolution - Include administrators - Restrict who can push

Workflow example:

```
name: Required Checks

on:
  pull_request:
    branches: [main]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: npm run lint
```

```yaml
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: npm test

  security:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: npm audit
```

Required Status Checks

In branch protection, mark these as required: - `lint` - `test` - `security`

PRs can't be merged until all pass.

Code Owners

Create `CODEOWNERS` file:

```
# Global owners
* @org/core-team

# Frontend code
/src/frontend/** @org/frontend-team

# Backend code
/src/backend/** @org/backend-team

# Infrastructure
/.github/** @org/devops-team
/terraform/** @org/devops-team

# Documentation
/docs/** @org/docs-team
```

Workflow integration:

```yaml
name: Code Review

on:
  pull_request:
    types: [opened, synchronize]

jobs:
```

```yaml
  check-owners:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Verify CODEOWNERS approval
        run: |
          # Custom script to verify approvals from code owners
          ./scripts/verify-codeowners.sh
```

Workflow Governance

Organization-level policies:

1. **Restrict actions**:
   - Only allow specific actions
   - Block third-party actions
   - Require action approval
2. **Enforce workflows**:
   - Required workflows for all repos
   - Centralized security scanning

Example required workflow (`.github/workflows/security-scan.yml`):

```yaml
name: Required Security Scan

on:
  push:
    branches: [main]
  pull_request:

jobs:
  security:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Run Trivy scanner
        uses: aquasecurity/trivy-action@master
        with:
          scan-type: 'fs'
          scan-ref: '.'
          format: 'sarif'
          output: 'trivy-results.sarif'

      - name: Upload to Security tab
        uses: github/codeql-action/upload-sarif@v2
```

```
        with:
          sarif_file: 'trivy-results.sarif'
```

Fine-grained PAT Tokens

Create fine-grained token: 1. Settings → Developer settings → Personal access tokens 2. Fine-grained tokens → Generate new token 3. Select specific repositories 4. Choose minimal permissions needed

Use in workflow:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
        with:
          token: ${{ secrets.FINE_GRAINED_PAT }}
          fetch-depth: 0

      - run: git push origin --tags
```

GitHub Apps vs PAT

GitHub Apps (Recommended): - More secure - Fine-grained permissions - Better audit trail - App installation tokens expire - Can act as different entities

PAT (Personal Access Token): - Tied to user account - Broader permissions - Longer-lived - Easier to set up

Using GitHub App:

```
jobs:
  app-auth:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/create-github-app-token@v1
        id: app-token
        with:
          app-id: ${{ secrets.APP_ID }}
          private-key: ${{ secrets.APP_PRIVATE_KEY }}

      - uses: actions/checkout@v3
        with:
          token: ${{ steps.app-token.outputs.token }}
```

Rate Limits

GitHub API rate limits: - Authenticated: 5,000 requests/hour - GitHub Actions: Higher limits (varies)

Handle rate limits:

```yaml
steps:
  - name: Check rate limit
    run: |
      RATE_LIMIT=$(curl -s -H "Authorization: token ${{ secrets.GITHUB_TOKEN }}" \
        https://api.github.com/rate_limit | jq -r '.rate.remaining')
      echo "Remaining API calls: $RATE_LIMIT"

      if [ "$RATE_LIMIT" -lt 100 ]; then
        echo "::warning::Approaching rate limit"
      fi
```

Best practices: - Cache API responses - Use conditional requests (ETags) - Batch operations - Use GraphQL instead of REST

Interview Important Questions & Answers

Q1: How does GitHub Actions work internally?

Answer:

GitHub Actions architecture consists of several layers:

1. **Event System**:
   - Webhooks trigger workflows
   - Events are queued in GitHub's event system
   - Filters (branches, paths) are applied
2. **Workflow Dispatcher**:
   - Parses YAML workflow files
   - Validates syntax and permissions
   - Creates job execution plan
3. **Job Queue**:
   - Jobs are queued based on availability
   - Matrix jobs are expanded
   - Dependencies (`needs`) are resolved
4. **Runner Assignment**:
   - Available runners are matched to jobs
   - Labels are used for self-hosted runners
   - VM/container is provisioned
5. **Job Execution**:
   - Each job runs in isolated environment
   - Steps execute sequentially

- Actions are downloaded from marketplace or repository
- Outputs and artifacts are collected

6. **Result Collection**:
   - Logs are streamed to GitHub
   - Status checks are updated
   - Artifacts are stored

Key Points: - Each job runs on a fresh runner (stateless) - Jobs in parallel = faster execution - Secrets are injected at runtime, never logged

---

Q2: How do you secure secrets in GitHub Actions?

Answer:

Multiple layers of security:

1. **Storage**:
   - Encrypted at rest using AES-256
   - Decrypted only at runtime
   - Never exposed in logs (automatically masked)
2. **Access Control**:
   - Repository secrets: repo-level access
   - Organization secrets: org-level access
   - Environment secrets: environment-specific + protection rules
3. **Best Practices**:

```
#  BAD - Don't do this
- run: echo "My secret is ${{ secrets.API_KEY }}"

#  GOOD - Use secrets properly
- name: Deploy
  env:
    API_KEY: ${{ secrets.API_KEY }}
  run: ./deploy.sh

#  BETTER - Use OIDC (no static secrets)
- uses: aws-actions/configure-aws-credentials@v1
  with:
    role-to-assume: arn:aws:iam::123456789012:role/MyRole
    aws-region: us-east-1
```

4. **OIDC Approach** (Modern):
   - No long-lived credentials
   - Temporary tokens
   - Based on OIDC claims (repo, branch, etc.)
5. **Additional Security**:
   - Rotate secrets regularly

- Use minimal scope secrets
- Different secrets for different environments
- Audit secret usage

Q3: How to deploy to AWS using GitHub Actions?

Answer:

Method 1: Using Access Keys (Traditional):

```yaml
name: Deploy to AWS

on:
  push:
    branches: [main]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1

      - name: Deploy to S3
        run: aws s3 sync ./build s3://my-bucket/

      - name: Deploy to EC2
        run: |
          aws ssm send-command \
            --instance-ids i-1234567890abcdef0 \
            --document-name "AWS-RunShellScript" \
            --parameters 'commands=["cd /var/www/app","git pull","npm install","pm2 restart
```

Method 2: Using OIDC (Recommended):

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    permissions:
      id-token: write  # Required for OIDC
      contents: read
```

```yaml
    steps:
      - uses: actions/checkout@v3

      - uses: aws-actions/configure-aws-credentials@v1
        with:
          role-to-assume: arn:aws:iam::123456789012:role/GitHubActionsRole
          aws-region: us-east-1

      - name: Deploy to ECS
        run: |
          aws ecs update-service \
            --cluster my-cluster \
            --service my-service \
            --force-new-deployment
```

AWS Setup for OIDC: 1. Create OIDC provider in IAM 2. Create IAM role with trust policy for GitHub 3. Attach permissions to role

---

Q4: Difference between Composite Action and Reusable Workflow?

Answer:

Feature	Composite Action	Reusable Workflow
Scope	Single steps	Complete jobs
Location	`.github/actions/`	`.github/workflows/`
Reusability	Within steps	Entire workflow
Secrets	Not directly supported	Full secret support
Outputs	Step-level outputs	Job-level outputs
Runner	Uses parent job's runner	Can specify own runner

Composite Action (for reusable steps):

```yaml
# .github/actions/setup/action.yml
name: 'Setup App'
runs:
  using: 'composite'
  steps:
    - uses: actions/setup-node@v3
      with:
        node-version: '18'
    - run: npm ci
      shell: bash

# Usage in workflow
```

```yaml
steps:
  - uses: ./.github/actions/setup
  - run: npm test
```

Reusable Workflow (for complete workflows):

```yaml
# .github/workflows/deploy.yml
on:
  workflow_call:
    inputs:
      environment:
        required: true
        type: string

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Deploying to ${{ inputs.environment }}"

# Usage in another workflow
jobs:
  call-deploy:
    uses: ./.github/workflows/deploy.yml
    with:
      environment: production
```

When to use what: - Composite Action: Reuse common setup steps - Reusable Workflow: Standardize entire deployment process

---

Q5: What is Matrix Strategy?

Answer:

Matrix strategy runs the same job with different configurations in parallel.

Basic Example:

```yaml
jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        node: [14, 16, 18]
        # Creates 9 jobs: 3 OS × 3 Node versions
    steps:
```

```
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node }}
      - run: npm test
```

Advanced Matrix:

```
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest]
    node: [16, 18]
    include:
      # Add specific combination
      - os: ubuntu-latest
        node: 20
        experimental: true
    exclude:
      # Remove specific combination
      - os: windows-latest
        node: 16
  fail-fast: false  # Don't stop other jobs if one fails
```

Real-world use: - Test across multiple OS/versions - Deploy to multiple regions - Run different database versions

Benefits: - Parallel execution = faster - Comprehensive testing - Single workflow definition

---

Q6: How does caching improve performance?

Answer:

Without Caching:

```
steps:
  - run: npm install  # Downloads all packages (~2-3 minutes)
  - run: npm test
```

With Caching:

```
steps:
  - uses: actions/cache@v3
    with:
      path: ~/.npm
      key: ${{ runner.os }}-npm-${{ hashFiles('**/package-lock.json') }}
      restore-keys: |
        ${{ runner.os }}-npm-
```

```
  - run: npm ci  # Uses cache (~10-30 seconds)
  - run: npm test
```

How it works:

1. **First run**:
   - No cache exists
   - Install dependencies (slow)
   - Cache is saved with key
2. **Subsequent runs**:
   - Cache key matches
   - Dependencies restored from cache (fast)
   - Skip installation

Performance Gains: - **npm/yarn**: 2-3 minutes → 10-30 seconds (80-90% faster) - **pip**: 1-2 minutes → 5-10 seconds - **Docker layers**: 5-10 minutes → 30 seconds

Cache Strategy:

```
# Good key - changes when dependencies change
key: ${{ runner.os }}-npm-${{ hashFiles('**/package-lock.json') }}

# Fallback keys - use older cache if exact match not found
restore-keys: |
  ${{ runner.os }}-npm-
  ${{ runner.os }}-
```

Best Practices: - Cache only dependencies, not build outputs - Use specific keys with hash of dependency files - Set restore-keys for partial matches - Clean cache periodically (auto-deletes after 7 days)

Q7: How to integrate GitHub Actions with Kubernetes?

Answer:

Complete K8s Deployment:

```
name: Deploy to Kubernetes

on:
  push:
    branches: [main]

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
```

```yaml
steps:
  - uses: actions/checkout@v3

  # Build Docker image
  - name: Build Docker image
    run: |
      docker build -t myapp:${{ github.sha }} .

  # Push to registry
  - name: Login to Docker Hub
    uses: docker/login-action@v2
    with:
      username: ${{ secrets.DOCKER_USERNAME }}
      password: ${{ secrets.DOCKER_PASSWORD }}

  - name: Push image
    run: |
      docker tag myapp:${{ github.sha }} username/myapp:${{ github.sha }}
      docker push username/myapp:${{ github.sha }}

  # Configure kubectl
  - name: Set up kubectl
    uses: azure/setup-kubectl@v3
    with:
      version: 'v1.28.0'

  - name: Configure kubectl
    env:
      KUBECONFIG_DATA: ${{ secrets.KUBECONFIG }}
    run: |
      echo "$KUBECONFIG_DATA" | base64 -d > kubeconfig
      export KUBECONFIG=kubeconfig
      kubectl config use-context my-cluster

  # Deploy to K8s
  - name: Update deployment
    run: |
      kubectl set image deployment/myapp \
        myapp=username/myapp:${{ github.sha }} \
        -n production

      kubectl rollout status deployment/myapp -n production

  # Verify deployment
  - name: Verify deployment
    run: |
```

```
          kubectl get pods -n production
          kubectl get svc -n production
```

Using Helm:

```yaml
- name: Deploy with Helm
  run: |
    helm upgrade --install myapp ./helm/myapp \
      --set image.tag=${{ github.sha }} \
      --set environment=production \
      --namespace production \
      --wait \
      --timeout 5m
```

With ArgoCD (GitOps):

```yaml
- name: Update ArgoCD manifest
  run: |
    # Update image tag in K8s manifests
    sed -i 's|image: myapp:.*|image: myapp:${{ github.sha }}|' k8s/deployment.yaml

    # Commit and push
    git config user.name github-actions
    git config user.email github-actions@github.com
    git add k8s/deployment.yaml
    git commit -m "Update image to ${{ github.sha }}"
    git push

    # ArgoCD auto-syncs the changes
```

Q8: How to handle failures in workflows?

Answer:

1. Continue on Error:

```yaml
steps:
  - name: Run linter
    run: npm run lint
    continue-on-error: true  # Job continues even if this fails

  - name: Run tests
    run: npm test  # This still runs even if linter failed
```

2. Conditional Execution:

```yaml
steps:
  - name: Run tests
```

```
    id: test
    run: npm test

  - name: Upload coverage (only on success)
    if: success()
    run: upload-coverage.sh

  - name: Notify on failure
    if: failure()
    run: send-slack-notification.sh

  - name: Cleanup (always runs)
    if: always()
    run: cleanup.sh
```

3. Retry on Failure:

```
steps:
  - name: Flaky test with retry
    uses: nick-fields/retry-action@v2
    with:
      timeout_minutes: 10
      max_attempts: 3
      retry_wait_seconds: 30
      command: npm test
```

4. Fallback Strategy:

```
steps:
  - name: Try primary deployment
    id: primary
    run: deploy-to-primary.sh
    continue-on-error: true

  - name: Fallback deployment
    if: steps.primary.outcome == 'failure'
    run: deploy-to-secondary.sh
```

5. Failure Notifications:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - run: npm test

  notify:
    runs-on: ubuntu-latest
    needs: build
```

```
    if: failure()
    steps:
      - name: Send Slack notification
        uses: slackapi/slack-github-action@v1
        with:
          payload: |
            {
              "text": "Build failed: ${{ github.repository }}",
              "workflow": "${{ github.workflow }}",
              "commit": "${{ github.sha }}"
            }
        env:
          SLACK_WEBHOOK_URL: ${{ secrets.SLACK_WEBHOOK }}
```

6. Rollback on Failure:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Deploy new version
        id: deploy
        run: kubectl set image deployment/myapp myapp:${{ github.sha }}

      - name: Health check
        id: health
        run: ./health-check.sh
        timeout-minutes: 5

      - name: Rollback on failure
        if: failure() && steps.deploy.outcome == 'success'
        run: |
          echo "Health check failed, rolling back"
          kubectl rollout undo deployment/myapp
```

7. Manual Approval After Failure:

```
jobs:
  deploy-staging:
    runs-on: ubuntu-latest
    steps:
      - run: deploy-staging.sh

  manual-approval:
    runs-on: ubuntu-latest
    needs: deploy-staging
    if: failure()
    environment: production-override  # Requires manual approval
```

```
    steps:
      - run: echo "Manual override approved"

  deploy-production:
    runs-on: ubuntu-latest
    needs: [deploy-staging, manual-approval]
    if: |
      success() ||
      (needs.deploy-staging.result == 'failure' && needs.manual-approval.result == 'success'
    steps:
      - run: deploy-production.sh
```

Conclusion

GitHub Actions is a powerful automation platform that's become essential for modern DevOps workflows. From simple CI pipelines to complex enterprise deployments, mastering these concepts will make you a valuable asset to any development team.

Key Takeaways:

1. **Start Simple**: Master fundamentals before jumping to advanced topics
2. **Security First**: Always use secrets properly, prefer OIDC over static credentials
3. **Optimize Performance**: Use caching, matrix strategies, and parallel jobs
4. **Think Reusability**: Create composite actions and reusable workflows
5. **Monitor & Debug**: Use proper logging and error handling
6. **Stay Updated**: GitHub Actions evolves rapidly; keep learning

Next Steps: - Build sample projects using different workflows - Contribute to open-source projects using GitHub Actions - Practice deploying to cloud platforms (AWS, Azure, GCP) - Create your own custom actions - Experiment with advanced features

Resources: - Official GitHub Actions Documentation - GitHub Actions Marketplace - Awesome Actions Repository

Remember: The best way to learn is by doing. Start automating your projects today!

Found this helpful? Give it a and follow for more DevOps content!

Questions? Drop them in the comments below.

Connect with me: - GitHub: [Your GitHub] - LinkedIn: [Your LinkedIn] - Twitter: [Your Twitter]

Happy Automating!