Here's the complete self-hosted runner and interview questions content formatted cleanly for Word:

SELF-HOSTED RUNNERS IN GITHUB ACTIONS

WHAT IS A SELF-HOSTED RUNNER?

Example with Jenkins: In Jenkins, we can create another EC2 instance or a Docker container which will act as a runner.

What is a runner: Runner is a place where our job gets run.

In Jenkins terminology: Runner is an agent or worker node. That's where your application gets run.

In GitHub Actions terminology: Runner is where the workflow executes.

TWO TYPES OF RUNNERS

In GitHub Actions and GitLab: Runners are categorized into:

1. GitHub-hosted runners
2. Self-hosted runners

GITHUB-HOSTED RUNNERS

WHAT ARE GITHUB-HOSTED RUNNERS?

Definition: When you run a GitHub project or when you build a CI project on the GitHub system using runners provided by GitHub.

Examples of projects using GitHub-hosted runners: Open source projects like Kubernetes, Argo CD, etc.

How it works: Their CI system is directly built on GitHub because their source code is also hosted on GitHub.

Cost for open source:

Using GitHub-hosted runners is completely free for:

- Open source projects
- Public repositories

Characteristics of GitHub-hosted runners:

When you take a runner from GitHub and execute your project on GitHub runner:

- You don't own that runner
- It's a public runner
- GitHub gives it to you for the execution of CI pipeline
- Once the execution is done, runner gets terminated
- You don't have any ownership of the runner

This is the GitHub-hosted runner.

SELF-HOSTED RUNNERS

WHAT ARE SELF-HOSTED RUNNERS?

Definition: Similar to what we have done in Jenkins.

In Jenkins:

- We had Jenkins master
- Using this we created Jenkins workers or Jenkins agents
- These can be EC2 instances or Docker containers

In GitHub Actions: We can do the same. Create our own runners.

WHY USE SELF-HOSTED RUNNERS?

Question: What is the purpose of using self-hosted runner when you already have access to GitHub-hosted runners which is free for public repos?

Answer: There are specific scenarios where self-hosted runners are necessary.

SCENARIO 1: ENTERPRISE COMPANY WITH PRIVATE REPOS

Situation: You are an enterprise company and not an open source project.

Details:

- Using private repositories
- Or using public repos where you require huge servers to execute your application

Cost issue: GitHub-hosted runners are:

- Free for public repos
- Limited free minutes for private repos (2000 minutes/month)
- After free tier, you pay per minute

For large private projects: Costs can become very high.

Solution: Use self-hosted runners on your own infrastructure.

SCENARIO 2: SPECIAL HARDWARE REQUIREMENTS

Requirement: You require a runner that has 32GB RAM for some reason.

Or: You have very special requirements of packages that are not coming with GitHub-hosted runners.

GitHub-hosted runner specs:

- 2 CPU cores
- 7GB RAM
- 14GB SSD

Your requirement:

- 16 CPU cores
- 64GB RAM
- 500GB SSD
- GPU support
- Specialized software

Solution: Use self-hosted runner with your custom specifications.

SCENARIO 3: SECURITY AND COMPLIANCE

Example: Banking application

Security concern: Why would you put your code on a runner where you don't have any information about it?

Problems with GitHub-hosted runners:

- Technically it is a GitHub-hosted runner
- You don't know where it is hosted
- It can be hosted in Antarctica or it can be hosted in US
- We don't know where in the world
- You don't know if this server is sharing data with any other company
- No control over infrastructure

When security is the key concern: Even in that case, you should not be using GitHub-hosted runners.

Compliance requirements:

- Data must stay in specific region
- Data sovereignty laws
- Industry regulations (banking, healthcare)
- Company policies

Solution: Use self-hosted runners in your own secure infrastructure.

SETTING UP SELF-HOSTED RUNNER

STEP 1: LAUNCH OR USE EXISTING EC2 INSTANCE

Options:

- Launch a new instance on AWS
- Use existing instance

For this example: We will use the existing instance.

Instance details:

- Operating System: Ubuntu
- Instance type: t2.medium or larger
- Region: Your choice

STEP 2: CONFIGURE SECURITY GROUP

Why needed: Your AWS account and GitHub account may or may not be in the same network.

Example:

- You may be using public hosted GitHub
- Your AWS may be in a VPC

Security concern: We have to be very careful in opening the ports. If there is any misconfiguration in ports, there is a very good chance of running into security concerns.

Requirement: Make sure to open right inbound and outbound rules.

Configure inbound rules:

Step 1: Go to EC2 instance page

Step 2: Click on Security tab

Step 3: Click on security group link

Step 4: Click "Inbound rules" tab

Step 5: Click "Edit inbound rules"

Step 6: Add 2 rules

Rule 1:

- Type: HTTP
- Port: 80
- Source type: Anywhere-IPv4 (0.0.0.0/0)

Rule 2:

- Type: HTTPS
- Port: 443
- Source type: Anywhere-IPv4 (0.0.0.0/0)

Step 7: Save rules

Configure outbound rules:

Step 1: Click "Outbound rules" tab

Step 2: Click "Edit outbound rules"

Step 3: Add 2 rules

Rule 1:

- Type: HTTP
- Port: 80
- Destination type: Anywhere-IPv4 (0.0.0.0/0)

Rule 2:

- Type: HTTPS
- Port: 443
- Destination type: Anywhere-IPv4 (0.0.0.0/0)

Step 4: Save rules

Why these rules:

Inbound HTTP and HTTPS: Your EC2 instance has to receive requests from GitHub.

Outbound HTTP and HTTPS: Your EC2 instance has to send notifications back to GitHub.

Flow:

- GitHub sends job to runner: Inbound traffic
- Runner executes job
- Runner sends results to GitHub: Outbound traffic

That's the reason why we configured both inbound and outbound traffic.

STEP 3: REGISTER RUNNER ON GITHUB

Navigate to GitHub:

Path: Repository → Settings → Actions → Runners

Click: "New self-hosted runner"

Choose operating system: Linux

Instructions appear:

Download section:

```
# Create a folder
mkdir actions-runner && cd actions-runner


# Download the latest runner package
curl -o actions-runner-linux-x64-2.311.0.tar.gz -L
https://github.com/actions/runner/releases/download/v2.311.0/actions-runner-linux-
x64-2.311.0.tar.gz


# Extract the installer
tar xzf ./actions-runner-linux-x64-2.311.0.tar.gz
```

Configure section:

```
# Create the runner and start the configuration
./config.sh --url https://github.com/yourusername/yourrepo --token YOUR_TOKEN


# Run it
./run.sh
```

STEP 4: EXECUTE COMMANDS ON EC2 INSTANCE

Open terminal on EC2 instance (via SSH or MobaXterm).

Copy and paste all commands from GitHub.

Command 1: Create folder

```
mkdir actions-runner && cd actions-runner
```

Creates actions-runner folder and enters it.

Command 2: Download runner

```
curl -o actions-runner-linux-x64-2.311.0.tar.gz -L
https://github.com/actions/runner/releases/download/v2.311.0/actions-runner-linux-
x64-2.311.0.tar.gz
```

What this does: Downloads the runner configuration package from GitHub.

Command 3: Extract

```
tar xzf ./actions-runner-linux-x64-2.311.0.tar.gz
```

What this does: Extracts the downloaded archive.

Command 4: Configure runner

```
./config.sh --url https://github.com/yourusername/yourrepo --token YOUR_TOKEN
```

What this does: Installs the runner configuration and connects it to your repository.

During configuration, you'll be asked:

- Runner name: (press Enter for default or type custom name)
- Runner group: (press Enter for default)
- Labels: (press Enter for default)
- Work folder: (press Enter for default)

Output:

```
Settings Saved.
```

Command 5: Start runner

```
./run.sh
```

What this does: This script will start the runner and it will start listening to your GitHub.

Output:

```
√ Connected to GitHub
```

Current runner version: '2.311.0'

Listening for Jobs

Your EC2 instance is now configured as a runner.

STEP 5: VERIFY RUNNER ON GITHUB

Go back to GitHub: Repository → Settings → Actions → Runners

You will see your runner listed:

- Name: (your EC2 hostname or custom name)
- Status: Idle (green dot)
- Labels: self-hosted, Linux, X64

Runner is ready to accept jobs.

STEP 6: UPDATE WORKFLOW TO USE SELF-HOSTED RUNNER

Previously in first-actions.yml:

yaml

```
runs-on: ubuntu-latest
```

This was using GitHub-hosted runner with Ubuntu.

Update to use self-hosted runner:

Edit file: .github/workflows/first-actions.yml

New content:

yaml

```
name: My First GitHub Actions

on: [push]

jobs:
  build:
```

```
  runs-on: self-hosted

  strategy:
    matrix:
      python-version: [3.8, 3.9]

  steps:
  - uses: actions/checkout@v3

  - name: Set up Python
    uses: actions/setup-python@v2
    with:
      python-version: ${{ matrix.python-version }}

  - name: Install dependencies
    run: |
      python -m pip install --upgrade pip
      pip install pytest

  - name: Run tests
    run: |
      cd src
      python -m pytest addition.py
```

Key change:

yaml

```
runs-on: self-hosted
```

Instead of:

yaml

```
runs-on: ubuntu-latest
```

STEP 7: COMMIT AND PUSH

Commands:

```
git add .
git commit -m "Update to use self-hosted runner"
git push origin main
```

STEP 8: OBSERVE EXECUTION

What happens after push:

GitHub side:

- Orange dot appears in front of commit
- Workflow triggered
- Jobs queued

EC2 instance terminal (where run.sh is running):

You can see jobs are running.

Terminal output:

```
Listening for Jobs
Running job: build (3.8)
Running: Set up Python
Running: Install dependencies
Running: Run tests
Job completed successfully
```

Real-time execution visible on your EC2 terminal.

Step-by-step process:

Step 1: GitHub receives push event

Push detected on main branch.

Step 2: Workflow triggered

first-actions.yml workflow starts.

Step 3: Job assigned to self-hosted runner

GitHub looks for runner with label "self-hosted".

Step 4: Runner accepts job

Your EC2 instance accepts the job.

Step 5: Job executes on EC2

All steps run on your EC2 instance.

Step 6: Results sent back to GitHub

Once job build is successful, your EC2 instance or runner will send the notification back to GitHub.

That's the reason why we configured the outbound traffic.

Step 7: GitHub updates status

Green checkmark appears on GitHub.

Traffic flow explanation:

Inbound traffic (HTTP/HTTPS):

EC2 instance has to receive requests from GitHub.

- GitHub sends: "Here's a job to run"

- EC2 receives: Job details

Outbound traffic (HTTP/HTTPS):

EC2 instance has to send notifications back to GitHub.

- EC2 sends: "Job started"

- EC2 sends: "Job completed successfully"

- EC2 sends: Logs and results

This is why both inbound and outbound rules are needed.

STEP 9: CHECK RESULTS ON GITHUB

Once all jobs are executed successfully:

We can see green tick mark in our GitHub commit list.

Click on green checkmark:

Shows all jobs passed.

Click "Details":

Shows execution logs from your self-hosted runner.

We are done now.

Runner is working perfectly.

RUNNING RUNNER AS A SERVICE

Current limitation:

If you close terminal, runner stops.

Solution: Run as a service

Stop current runner:

Press Ctrl+C in terminal where run.sh is running.

Install as service:

```
sudo ./svc.sh install
sudo ./svc.sh start
```

Check status:

```
sudo ./svc.sh status
```

Output:

```
● actions.runner.yourusername-yourrepo.hostname.service - GitHub Actions Runner
  Loaded: loaded
  Active: active (running)
```

Benefit: Runner runs in background. Survives terminal closure and system reboots.

GITHUB ACTIONS INTERVIEW QUESTIONS

QUESTION 1: WHY GITHUB ACTIONS AND NOT JENKINS?

Answer:

In our organization, we do everything on GitHub.

Reasons we prefer GitHub Actions:

Reason 1: Integration

- All code on GitHub

- GitHub Actions natively integrated
- Seamless workflow
- No separate tool to manage

Reason 2: No infrastructure management

- Don't need to install Jenkins
- Don't need to maintain Jenkins server
- GitHub provides runners
- Free for public repos

Reason 3: Ease of use

- Simpler configuration
- YAML files in repository
- Version controlled automatically
- Easier for team to understand

Reason 4: Cost-effective

- Free for public repos
- Free tier for private repos (2000 minutes/month)
- No server costs
- Pay only for usage

Reason 5: Built-in features

- No plugin installation needed
- Actions marketplace available
- Pre-built actions for common tasks
- Quick to set up

Why GitHub Actions is better than Jenkins (summary):

Hosting: Jenkins is self-hosted (requires own server), while GitHub Actions is hosted by GitHub and runs directly in your repository.

User interface: Jenkins has complex and sophisticated user interface, while GitHub Actions has more streamlined and user-friendly interface that is better suited for simple to moderate automation tasks.

Cost: Jenkins can be expensive to run and maintain, especially for organizations with large and complex automation needs. GitHub Actions is free for open-source projects and has tiered pricing model for private repositories, making it more accessible to smaller organizations and individual developers.

Integration: GitHub Actions is tightly integrated with GitHub platform, making it easier to automate tasks related to GitHub workflow.

When to choose GitHub Actions:

- Using GitHub exclusively
- Want simplicity
- Cost-conscious
- Small to medium projects
- Open source projects

QUESTION 2: HOW DO YOU SECURE SENSITIVE INFORMATION IN GITHUB?

Answer:

We go to Settings → Secrets and variables → Actions.

Detailed steps:

Step 1: Navigate to repository on GitHub

Step 2: Click "Settings" tab

Step 3: In left sidebar, click "Secrets and variables"

Step 4: Click "Actions"

Step 5: Click "New repository secret"

Step 6: Add secret:

- Name: DATABASE_PASSWORD (example)
- Secret: your-actual-password
- Click "Add secret"

Secret is now stored securely.

Using secrets in workflow:

Syntax:

yaml

```
steps:
 - name: Deploy application
   run: ./deploy.sh
   env:
     DB_PASSWORD: ${{ secrets.DATABASE_PASSWORD }}
     API_KEY: ${{ secrets.API_KEY }}
```

Benefits:

- Secrets encrypted
- Never visible in logs
- Masked in output
- Secure storage

Security features:

- Secrets not shown in workflow files
- Secrets not shown in logs (masked as ***)
- Only available during workflow execution
- Can be rotated easily

QUESTION 3: HOW DO YOU WRITE A GITHUB CI FILE?

Answer:

Step 1: Create folder structure Create .github/workflows folder in repository root.

Step 2: Create YAML file Write a .yml file inside .github/workflows folder.

Example: .github/workflows/ci.yml

Step 3: Define workflow

Basic structure:

```yaml
name: CI Pipeline

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3

      - name: Set up environment
        uses: actions/setup-node@v3
        with:
          node-version: '18'

      - name: Install dependencies
        run: npm install

      - name: Run tests
        run: npm test

      - name: Build application
        run: npm run build
```

Explanation of code:

name: Workflow name (displays in GitHub UI)

on: [push, pull_request]: Triggers for workflow execution

jobs: Collection of jobs to run

build: Job name

runs-on: ubuntu-latest: Use GitHub-hosted Ubuntu runner

steps: List of actions to perform

uses: actions/checkout@v3: Pre-built action to checkout code

name: Descriptive name for step

run: Shell commands to execute

with: Parameters for action

Step 4: Commit and push

```
git add .github/workflows/ci.yml
git commit -m "Add CI workflow"
git push origin main
```

```

Step 5: Workflow runs automatically

Check "Actions" tab on GitHub to see execution.

---

QUESTION 4: GITHUB ACTIONS vs JENKINS - WHEN TO USE WHAT?

Answer:

Use GitHub Actions when:

Condition 1: Public repository

If public repo, then GitHub Actions is best because it's free.

Condition 2: Using GitHub exclusively

All code, issues, projects on GitHub.

Condition 3: Simple to moderate complexity

Straightforward CI/CD needs.

Condition 4: Small team

Easier onboarding and collaboration.

Condition 5: Want zero infrastructure management

Don't want to manage servers.
```

Use Jenkins when:

Condition 1: Private repositories with heavy usage

GitHub Actions costs add up for private repos with many builds.

Condition 2: Complex automation needs

Very complex pipelines, custom plugins needed.

Condition 3: Multi-VCS environment

Using GitHub, GitLab, Bitbucket together.

Condition 4: On-premises requirements

Need to run CI/CD completely on-premises.

Condition 5: Large organization

Already invested in Jenkins infrastructure and expertise.

Condition 6: Custom integrations

Need to integrate with proprietary tools.

Detailed comparison:

Aspect: Platform dependency

- GitHub Actions: GitHub only

- Jenkins: Any VCS

Aspect: Setup complexity

- GitHub Actions: Very simple (no installation)

- Jenkins: Moderate to complex (install, configure)

Aspect: Infrastructure

- GitHub Actions: Managed by GitHub

- Jenkins: You manage

Aspect: Cost for public repos

- GitHub Actions: Free

- Jenkins: Infrastructure costs

Aspect: Cost for private repos

- GitHub Actions: 2000 free minutes/month, then paid

- Jenkins: Infrastructure costs (can be cheaper at scale)

Aspect: Scalability

- GitHub Actions: Automatic (to thousands)

- Jenkins: Manual (add worker nodes)

Aspect: Plugin ecosystem

- GitHub Actions: Growing (thousands)

- Jenkins: Mature (tens of thousands)

Aspect: Learning curve

- GitHub Actions: Easy

- Jenkins: Steeper

Aspect: Customization

- GitHub Actions: Limited
- Jenkins: Extensive

Aspect: Enterprise features

- GitHub Actions: Basic
- Jenkins: Advanced

QUESTION 5: What happens when you push code with GitHub Actions configured?

Answer:

Step 1: Developer pushes code

```
git push origin main
```

Step 2: GitHub detects push Webhook triggered.

Step 3: Workflow triggered Workflows with on: [push] trigger execute.

Step 4: Runner provisioned

- GitHub-hosted: GitHub creates VM/container
- Self-hosted: Job sent to your runner

Step 5: Workflow executes All steps run in order.

Step 6: Results reported

- Success: Green checkmark
- Failure: Red X

- Running: Orange dot

Step 7: Notifications

- Email sent (if configured)
- PR status updated
- Commit status updated

Step 8: Cleanup

- GitHub-hosted: Runner terminated
- Self-hosted: Job completes, waits for next

QUESTION 6: What is matrix strategy in GitHub Actions?

Answer:

Matrix strategy allows testing across multiple versions/configurations in parallel.

Example:

yaml

```
strategy:
  matrix:
    python-version: [3.8, 3.9, 3.10]
    os: [ubuntu-latest, windows-latest]
```

Creates 6 jobs:

- Python 3.8 on Ubuntu
- Python 3.9 on Ubuntu
- Python 3.10 on Ubuntu
- Python 3.8 on Windows
- Python 3.9 on Windows
- Python 3.10 on Windows

All run in parallel.

Benefit: Ensure code works across all combinations.

QUESTION 7: How do you debug failed GitHub Actions workflows?

Answer:

Method 1: Check logs

- Go to Actions tab
- Click on failed workflow
- Click on failed job
- Expand failed step
- Read error message

Method 2: Enable debug logging Add secrets to repository:

- ACTIONS_RUNNER_DEBUG: true
- ACTIONS_STEP_DEBUG: true

Provides verbose logging.

Method 3: Use tmate action Add step to SSH into runner:

```yaml
- name: Setup tmate session
  uses: mxschmitt/action-tmate@v3
```

Allows interactive debugging.

Method 4: Run locally

Use act tool to run GitHub Actions locally:

```
brew install act
act -l  # List workflows
act push  # Run push event
```

QUESTION 8: Can you use both GitHub-hosted and self-hosted runners in same workflow?

Answer:

Yes, you can.

Example:

```yaml
jobs:
  test:
    runs-on: ubuntu-latest  # GitHub-hosted
    steps:
      - run: npm test

  deploy:
    runs-on: self-hosted    # Self-hosted
    needs: test
    steps:
      - run: ./deploy.sh
```

Use case:

- Test on GitHub-hosted (free, consistent environment)
- Deploy on self-hosted (access to internal network)

SUMMARY

Self-hosted runners:

What they are: Your own machines that run GitHub Actions workflows.

Why use:

- Private repositories (cost savings)
- Special hardware requirements
- Security and compliance

- Custom software requirements
- Network access to internal resources

Setup:

1. Launch EC2 instance
2. Configure security groups (HTTP/HTTPS in/out)
3. Download runner from GitHub
4. Configure runner with ./config.sh
5. Start runner with ./run.sh
6. Update workflow: runs-on: self-hosted

Traffic requirements:

- Inbound: Receive jobs from GitHub
- Outbound: Send results to GitHub

GitHub-hosted vs self-hosted:

GitHub-hosted:

- Free for public repos
- No management
- Limited specs
- Unknown location
- Limited customization

Self-hosted:

- Cost of infrastructure
- You manage
- Custom specs
- Your location/region
- Full customization

Interview answers:

Q1: Why GitHub Actions:

- Using GitHub exclusively

- No infrastructure management
- Cost-effective
- Easy to use

Q2: Secure secrets:

- Settings → Secrets and variables
- Add repository secrets
- Use ${{ secrets.NAME }}

Q3: Write CI file:

- Create .github/workflows/file.yml
- Define triggers, jobs, steps
- Commit and push

Q4: GitHub vs Jenkins:

- Public repo: GitHub Actions (free)
- Private repo with heavy use: Jenkins (cheaper at scale)
- Multi-VCS: Jenkins
- GitHub only: GitHub Actions

Orange dot: Workflow running Green checkmark: Workflow passed Red X: Workflow failed

All workflows version controlled, reviewable, and automatically executed on every push.