Git+Github Advances FAQ

Git & GitHub Questions + Answers

Question 1:

You cloned a repository but realize you need to **work on a new feature without affecting the main branch**. What would you do?

Answer:

- Create a new branch:

```
git checkout -b feature/new-feature
```

- Work on your changes in this branch.
- Commit changes locally:

```
git add .
git commit -m "Implemented new feature"
```

- Push the branch to GitHub:

```
git push origin feature/new-feature
```

- Create a **Pull Request (PR)** on GitHub to merge into the main branch.

Question 2:

You committed changes directly to the main branch and want to **remove the last commit**. How would you do it?

Answer:

- If you haven't pushed:

```
git reset --soft HEAD~1   # keeps changes staged
git reset --hard HEAD~1   # discards changes
```

- If already pushed:

```
git revert HEAD   # creates a new commit undoing the last commit
```

- Always use `git revert` for pushed commits to maintain history integrity.

Question 3:

Explain **difference between git merge and git rebase** with scenario.

Answer:

- **Merge:** Combines branches, preserves commit history, creates merge commit.
- **Rebase:** Moves commits to the tip of another branch, creating linear history.
- **Scenario:**
    - Use **merge** for team collaboration to maintain history.
    - Use **rebase** for cleaning up local commits before pushing to remote.

Question 4:

How do you **resolve merge conflicts** when merging two branches?

Answer:

- Identify conflicting files: Git will mark them during merge.
- Open files and resolve conflicts manually. Conflict markers:

```
<<<<<<< HEAD
Current branch code
=======
Incoming branch code
>>>>>>> branch-name
```

- Add resolved files:

```
git add <file>
```

- Complete merge:

```
git commit
```

- Push changes to remote.

Question 5:

Your GitHub PR is failing due to **CI/CD checks**. How do you handle it?

Answer:

- Review CI/CD error logs.
- Fix **code/style/tests** locally.
- Commit fixes and push to the same branch.
- GitHub automatically re-runs the workflow.
- Ensure **all checks pass** before merging.

Question 6:

Explain **difference between fork and clone**.

Answer:

- **Clone:** Copies a repository to your local machine for development.
- **Fork:** Creates a personal copy of someone else's repository on GitHub.
- **Scenario:** Fork → clone → make changes → PR to original repo.

Question 7:

How do you **undo a pushed commit without rewriting history**?

Answer:

- Use `git revert`:

```
git revert <commit-hash>
```

- Creates a new commit that undoes the previous commit.
- Safe for collaborative projects.

Question 8:

How would you **check the history of commits** for a file?

Answer:

```
git log -- <file>
```

- Shows all commits affecting that file with author, date, and message.
- Add `-p` to see diffs:

```
git log -p -- <file>
```

Question 9:

How do you **squash multiple commits** into one before merging to main?

Answer:

- Interactive rebase:

```
git rebase -i HEAD~3  # last 3 commits
```

- Change `pick` to `squash` for commits to combine.
- Edit commit message.
- Push changes using force if branch already exists:

```
git push origin branch-name --force
```

Question 10:

Explain **Git branching strategies** commonly used in projects.

Answer:

- **Git Flow:** Feature, develop, release, hotfix branches.
- **GitHub Flow:** Main + short-lived feature branches, simple PR-based workflow.
- **GitLab Flow:** Combines environment branches with feature branches.
- **Scenario:** Use **GitHub Flow** for CI/CD pipelines in DevOps projects.

Question 11:

Your repository has diverged from remote, and you need to **pull latest changes without losing local commits**. How?

Answer:

- Use **rebase**:

```
git fetch origin
git rebase origin/main
```

- Resolve conflicts if any.
- Push rebased branch:

```
git push origin branch-name --force-with-lease
```

- Keeps a **linear commit history**.

Question 12:

How do you **delete a remote branch** after merging?

Answer:

```
git push origin --delete branch-name
```

- Locally, delete branch:

```
git branch -d branch-name
```

Question 13:

How would you **check which files have changed** between two commits?

Answer:

```
git diff <commit1> <commit2>
```

- Shows line-by-line differences.
- Use `--name-only` to list changed files:

```
git diff --name-only <commit1> <commit2>
```

Question 14:

Explain **stashing in Git** with a scenario.

Answer:

- Use `git stash` to save unfinished changes temporarily.
- Scenario: You're on a feature branch, need to switch to hotfix branch:

```
git stash
git checkout hotfix-branch
# fix hotfix
git checkout feature-branch
git stash pop
```

- Preserves work without committing incomplete code.

Question 15:

How would you **compare your branch with main** before creating a PR?

Answer:

```
git fetch origin
git diff origin/main..my-branch
```

- Shows all code changes.
- Use GitHub **Pull Request UI** for visual comparison.

Question 16:

Explain **GitHub Actions**. How would you use it to automate testing and deployment?

Answer:

- GitHub Actions is **CI/CD automation** within GitHub.
- Workflow YAML defines **events (push, pull request), jobs, and steps**.
- Example scenario:
    - On **push to main branch**, trigger a workflow:
        1. Checkout code (`actions/checkout@v3`)
        2. Install dependencies

3. Run tests
4. Build Docker image
5. Deploy to AWS ECS or EKS

- You can also **cache dependencies** to speed up workflows.

Question 17:

Your GitHub Actions workflow fails due to **permissions to push Docker images**. How do you fix it?

Answer:

- Configure **GitHub secrets** for Docker credentials (DOCKER_USERNAME, DOCKER_PASSWORD).
- Use `docker/login-action` to login:

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

- Use these credentials in **build and push steps**.
- Ensure proper **permissions in repository settings** for Actions.

Question 18:

How would you **trigger a workflow only when specific files change**?

Answer:

- Use `paths` filter in workflow YAML:

```
on:
  push:
    paths:
      - 'app/**'
      - 'Dockerfile'
```

- Workflow triggers only when files in the specified path are modified.
- Useful for **monorepo projects**.

Question 19:

Explain **GitHub Fork & PR workflow for open-source contributions**.

Answer:

1. Fork the repo to your account.
2. Clone the fork locally:

```
git clone <fork-url>
```

3. Create a **feature branch**, make changes, commit, and push.
4. Open a **Pull Request** to the original repo.
5. Sync fork with upstream if repo updates:

```
git remote add upstream <original-repo>
git fetch upstream
git rebase upstream/main
```

Question 20:

How do you **resolve conflicts in a GitHub PR** before merging?

Answer:

- Checkout the PR branch locally.

```
git fetch origin pull/<PR-number>/head:pr-branch
git checkout pr-branch
```

- Merge main branch to PR:

```
git merge main
```

- Resolve conflicts manually, add and commit changes:

```
git add .
git commit
git push origin pr-branch
```

- GitHub will update the PR automatically.

Question 21:

How do you **protect the main branch** in GitHub to enforce quality?

Answer:

- Enable **Branch Protection Rules**:
    - Require PR reviews before merge.
    - Require passing **CI/CD checks**.
    - Restrict who can push or merge.
    - Enable **status checks** and enforce **signed commits**.
- Prevents accidental direct pushes and ensures quality control.

Question 22:

Explain **Git rebase vs merge in collaborative projects** with example.

Answer:

- **Merge:** Preserves history, creates merge commit. Good for teamwork.
- **Rebase:** Linear history, cleaner commit log, rewrites history.
- Scenario:
  - Feature branch diverges from main.
  - Use `git rebase main` before pushing to avoid multiple merge commits.
- Always **avoid rebasing public branches** already pushed.

Question 23:

You accidentally committed **sensitive data**. How do you remove it completely from GitHub?

Answer:

- Use **git filter-branch** or **BFG Repo-Cleaner**:

```
git filter-branch --force --index-filter 'git rm --cached --ignore-unmatch
<file>' --prune-empty --tag-name-filter cat -- --all
```

- Push forcefully to overwrite history:

```
git push origin --force --all
git push origin --force --tags
```

- Rotate secrets/credentials if they were exposed.

Question 24:

How would you **automate semantic versioning and releases** using GitHub Actions?

Answer:

- Use **commit message conventions** (e.g., `feat`, `fix`, `chore`).
- GitHub Action reads commit messages and updates version (major/minor/patch).
- Create **release tags automatically**:

```
on:
  push:
    branches: [main]
jobs:
  release:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: some/semantic-version-action@v1
```

- Automates versioning and release notes generation.

Question 25:

Explain **GitHub Environments and Secrets** for CI/CD security.

Answer:

- **Environments**: Dev, Staging, Production with protection rules.
- **Secrets**: Store sensitive data like API keys, tokens, credentials.
- Workflows access secrets securely:

```
env:
  AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
```

- Environments can require **manual approvals** before deployment.

Question 26:

How would you **rollback a deployment** triggered by GitHub Actions if a bug is found in production?

Answer:

- Use **previous Git tag** or commit hash to rollback.
- Update workflow to deploy the previous stable version:

```
git checkout <stable-tag>
git push origin main
```

- If using **Docker**, redeploy previous image tag to ECS/EKS.
- Optionally, use **manual approvals** in workflow environments to prevent faulty deployment.

Question 27:

Explain **fork + upstream sync workflow** for an open-source contributor.

Answer:

1. Fork the repo.
2. Clone fork locally.
3. Add original repo as upstream:

```
git remote add upstream <original-repo-url>
git fetch upstream
git rebase upstream/main
```

4. Resolve conflicts if any.

5. Push to your fork and create PR.

Question 28:

Your **feature branch has diverged** significantly from main. How do you integrate changes safely?

Answer:

- Option 1: **Merge main into feature branch**

```
git checkout feature-branch
git merge main
```

- Option 2: **Rebase feature branch onto main** (cleaner history)

```
git checkout feature-branch
git rebase main
```

- Resolve conflicts carefully. Push updated branch:

```
git push origin feature-branch --force-with-lease
```

Question 29:

Explain **how GitHub Actions caching works** and its benefits.

Answer:

- Cache dependencies to speed up workflows.
- Example: Node modules or Maven packages.

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

- Reduces build time and resource usage.

Question 30:

Your CI workflow fails intermittently due to **race conditions in tests**. How do you fix it?

Answer:

- Identify flaky tests and make them **idempotent**.
- Use **sequential job execution** in Actions if needed.
- Isolate tests into separate jobs to **avoid shared state**.
- Add **retries** for critical steps:

```
jobs:
```

```
  test:
    runs-on: ubuntu-latest
    steps:
      - run: npm test
        continue-on-error: false
```

Question 31:

How would you **enforce code style and quality checks** before PR merge?

Answer:

- Use **GitHub Actions** for linters and static analysis.
- Example: ESLint for JavaScript:

```
- uses: actions/setup-node@v3
- run: npm install
- run: npm run lint
```

- Enable **branch protection rules** to require passing checks before merge.

Question 32:

Explain **monorepo management** using Git and GitHub Actions.

Answer:

- Use **directory-specific workflows** with `paths` filter.
- Build and test only affected projects:

```
on:
  push:
    paths:
      - 'project-a/**'
      - 'project-b/**'
```

- Use **workspace caching** to avoid rebuilding unaffected projects.
- Version each project independently if needed.

Question 33:

How would you **debug a GitHub Actions workflow** that fails on the runner?

Answer:

- Check **workflow logs** in GitHub Actions UI.
- Use **`set -x` or `echo` statements** for debugging shell commands.
- Run the workflow **locally** using `act` CLI for testing.
- Verify **runner environment** and installed dependencies.

Question 34:

Explain **GitHub Packages integration** in CI/CD.

Answer:

- GitHub Packages allows storing **Docker images, NPM packages, Maven packages**.
- Push packages in CI workflow:

```
- name: Build Docker image
  run: docker build -t ghcr.io/${{ github.repository }}/app:latest .
- name: Push image
  run: docker push ghcr.io/${{ github.repository }}/app:latest
```

- Use **GitHub token** for authentication.

Question 35:

How do you **handle multiple remote repositories** in Git?

Answer:

- Add multiple remotes:

```
git remote add origin <main-repo>
git remote add backup <backup-repo>
```

- Push to both remotes:

```
git push origin main
git push backup main
```

- Fetch/pull separately as needed.

Question 36:

How would you **secure sensitive data** in GitHub Actions workflows?

Answer:

- Use **GitHub Secrets** instead of hardcoding credentials.
- Use **environment-specific secrets** (dev, prod).
- Access secrets in workflow:

```
env:
  AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
```

- Rotate secrets regularly.

Question 37:

Explain **how to implement canary deployments** using GitHub Actions and ECS/EKS.

Answer:

- Use **workflow jobs** to deploy a small percentage of traffic first.
- Monitor logs and metrics.
- If healthy, update workflow to deploy 100% traffic.
- Example: Use **AWS CodeDeploy or App Mesh** for traffic shifting.

Question 38:

Your workflow needs **matrix builds** for multiple environments. How would you configure it?

Answer:

- Use `strategy.matrix` in GitHub Actions:

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

- Runs **all combinations** of matrix for testing.
- Useful for cross-platform compatibility.

Question 39:

How do you **safely force-push changes** to a shared branch?

Answer:

- Use `--force-with-lease` instead of `--force`:

```
git push origin branch-name --force-with-lease
```

- Ensures **you don't overwrite others' changes** unintentionally.

Question 40:

Explain **how to integrate GitHub Actions with Slack/Teams** for notifications.

Answer:

- Add a **step in workflow** using `slackapi/slack-github-action`:

```
- name: Notify Slack
  uses: slackapi/slack-github-action@v1.24.0
  with:
    channel-id: 'C12345678'
```

```
slack-message: 'Deployment succeeded'
github-token: ${{ secrets.GITHUB_TOKEN }}
```

- Can send **success/failure notifications** for PRs, builds, or deployments.

Question 41:

How would you automatically label GitHub PRs based on file changes?
Answer:
By configuring GitHub Actions with a labeling workflow. The workflow maps file paths to labels, and whenever a PR is opened or updated, the correct labels are applied automatically. This helps triage PRs quickly (e.g., “frontend”, “backend”, “docs”).

Question 42:

How do you use GitHub Actions to implement Infrastructure as Code (IaC) with Terraform?
Answer:
By setting up a workflow that validates Terraform code (format, init, plan) on pull requests and applies changes on merges to main. This ensures infrastructure changes are peer-reviewed, tested, and safely applied using automation.

Question 43:

How would you enforce signed commits in a GitHub repository?
Answer:
By enabling branch protection rules that require signed commits. Developers must configure GPG keys locally, and unsigned commits will be rejected during PR merges. This ensures commit authenticity and integrity.

Question 44:

How do you manage secrets across multiple environments (dev, staging, prod) in GitHub Actions?
Answer:
By creating environment-specific secrets in GitHub. Each workflow can be tied to an environment, and secrets are scoped accordingly. This prevents dev secrets from being used in prod and adds optional approval gates.

Question 45:

How would you roll back a failed GitHub Actions deployment safely?
Answer:
By redeploying the last known stable release (using a previous tag or commit). The workflow can be designed with a rollback job that redeploys automatically if monitoring detects failures, ensuring minimal downtime.

Question 46:

What’s the difference between Git submodules and Git subtrees?
Answer:

- **Submodules**: link to an external repo at a specific commit.
- **Subtrees**: embed another repo directly inside as a subdirectory.
  Submodules require explicit updating, while subtrees are easier to manage but duplicate history.

Question 47:

How would you integrate GitHub with Jira or other issue trackers?
Answer:
By enabling integrations where PR titles, commit messages, or branches reference Jira issue keys. This allows automatic linking of GitHub activity to Jira tickets for better traceability.

Question 48:

How do you handle a situation where a developer force-pushed and rewrote shared branch history?
Answer:

- Identify the overwritten commits from reflog.
- Restore missing commits if needed.
- Educate the team to use `--force-with-lease` instead of `--force`.
- Optionally, enforce branch protection to prevent direct force-pushes.

Question 49:

How do you manage GitHub Actions workflows for a monorepo with multiple services?
Answer:
By scoping workflows to specific paths, caching dependencies separately, and running jobs only for the service that changed. This avoids unnecessary builds and optimizes CI/CD.

Question 50:

How do you scale GitHub Actions workflows for heavy workloads?
Answer:
By using self-hosted runners with higher resources, splitting workflows into parallel jobs, leveraging matrix builds, and caching dependencies to reduce redundant work.

Question 51:

What is the difference between GitHub Packages and external registries (like Docker Hub, npm)?
Answer:
GitHub Packages integrates tightly with repos, access is controlled by repo permissions, and publishing is automated via Actions. External registries are broader but require additional authentication and management.

Question 52:

How do you enforce that all PRs must be reviewed before merging?
Answer:
By enabling branch protection rules requiring PR approvals. You can mandate one or more reviewers, restrict who can approve, and block merges if checks fail.

Question 53:

How do you handle performance issues in very large Git repositories?
Answer:
By using sparse-checkout, shallow clones, splitting into multiple repos, or using Git LFS for large files. This reduces repo size and improves performance.

Question 54:

What's the difference between `git fetch --all` and `git pull --all`?
Answer:

- `fetch --all`: updates local knowledge of all remotes without merging.
- `pull --all`: attempts to fetch and merge from all remotes, which can lead to conflicts.

Question 55:

How would you prevent merge conflicts in a team working on the same files?
Answer:
By establishing coding conventions, breaking down files into modular components, encouraging frequent pulls/rebases, and using feature toggles to reduce overlapping work.

Question 56:

How do you enforce consistent commit messages in GitHub repositories?
Answer:
By using Git hooks (commit-msg) locally and GitHub Actions checks remotely to validate commit messages against a defined regex or convention (like Conventional Commits).

Question 57:

How would you run integration tests across multiple repositories with GitHub Actions?
Answer:
By using a workflow that fetches dependent repositories, builds them, and runs integration tests together. Alternatively, use GitHub Actions' `repository_dispatch` to trigger workflows across repos.

Question 58:

What's the difference between GitHub Actions `workflow_dispatch` and `repository_dispatch`?
Answer:

- `workflow_dispatch`: manual workflow trigger from the GitHub UI.
- `repository_dispatch`: programmatically triggers workflows via API, often used across repositories.

Question 59:

How would you implement feature toggles with Git and GitHub?
Answer:
By merging incomplete features behind feature flags, allowing continuous integration without exposing unfinished code. This avoids long-lived branches and simplifies deployments.

Question 60:

How do you handle GitHub Actions secrets rotation?
Answer:
By periodically updating secrets in repo settings, integrating with secret managers (AWS Secrets Manager, Vault), and ensuring workflows always pull the latest values.

Question 61:

How do you handle multi-repo orchestration in GitHub Actions?
Answer:
By using `repository_dispatch` or scheduled workflows to trigger dependent builds across repositories. This ensures services are built and tested together, avoiding manual coordination.

Question 62:

What’s the difference between GitHub Enterprise Cloud and GitHub Enterprise Server?
Answer:

- **Cloud**: Fully managed by GitHub, scalable, integrates with GitHub.com ecosystem.
- **Server**: Self-hosted, gives more control over data and compliance, requires infrastructure management.

Question 63:

How do you secure GitHub Actions against supply-chain attacks?
Answer:

- Pin action versions (avoid `@main`).
- Use Dependabot to update actions.
- Limit third-party actions or host internal actions.
- Run workflows with least privilege using fine-grained tokens.

Question 64:

How do you integrate GitHub with Kubernetes deployments?
Answer:
By using GitHub Actions to build Docker images, push to a registry, and apply manifests via `kubectl` or GitOps tools like ArgoCD/Flux. This enables continuous delivery to clusters.

Question 65:

How do you deal with long-running workflows in GitHub Actions?
Answer:
Split workflows into smaller jobs, use artifacts to pass outputs, cache results, or offload heavy processing to self-hosted runners. For very long jobs, schedule periodic checkpoints.

Question 66:

How would you enforce dependency scanning on GitHub repositories?
Answer:
Enable **Dependabot alerts and security updates**. Combine with GitHub Advanced Security (CodeQL + secret scanning) or run third-party scanners via GitHub Actions for compliance.

Question 67:

What’s the difference between Git shallow clone and sparse checkout?
Answer:

- **Shallow clone**: downloads limited commit history (e.g., last 10 commits).
- **Sparse checkout**: downloads only specific folders/files, useful for monorepos.

Question 68:

How do you optimize CI/CD pipelines for monorepos in GitHub?
Answer:

- Use **path filters** to trigger workflows only for changed services.
- Cache dependencies separately for each service.
- Parallelize builds/tests per project.
- Apply independent versioning and tagging.

Question 69:

How do you audit GitHub repository activities?
Answer:
By enabling **audit logs** in GitHub Enterprise. You can track repo access, pushes, PR merges, and workflow executions. Logs can be exported to SIEM systems for monitoring.

Question 70:

How would you implement Blue-Green deployments using GitHub Actions?
Answer:
By deploying a new version to a “green” environment while “blue” serves traffic. Once validated, switch traffic to green. Rollback is simple — just switch traffic back to blue.

Question 71:

How do you manage large binary artifacts in GitHub?
Answer:
By using **Git LFS** for source control, and GitHub Packages or external artifact repositories (Artifactory, S3) for large binaries. This avoids bloating Git history.

Question 72:

How do you integrate GitHub Actions with AWS/GCP/Azure securely?
Answer:
By using OIDC federation (OpenID Connect). This avoids long-lived cloud credentials by issuing temporary tokens for workflows. More secure than storing keys in secrets.

Question 73:

What is Git bisect, and how would you use it in debugging?
Answer:
`git bisect` helps identify the commit that introduced a bug using binary search. You mark commits as good or bad, and Git automatically checks out intermediate commits until the culprit is found.

Question 74:

How do you implement release pipelines with GitHub Actions?
Answer:
By tagging versions in Git. A workflow listens for new tags, builds release artifacts, generates release notes, and publishes them to GitHub Releases or package registries.

Question 75:

How do you enforce DORA metrics tracking with GitHub?
Answer:
By collecting deployment frequency, lead time, change failure rate, and MTTR from GitHub Actions and Git logs. Integrate with dashboards (Grafana, Datadog) for continuous measurement.

Question 76:

How do you secure GitHub repositories in regulated industries (finance, healthcare)?
Answer:

- Enforce branch protection rules.
- Require code reviews + CI checks.
- Enable secret scanning and Dependabot.
- Use GitHub Enterprise with SAML/SSO.
- Regular audits of access and permissions.

Question 77:

What's the difference between GitHub Codespaces and local development?
Answer:

- **Codespaces**: Cloud-hosted dev environments, preconfigured, accessible anywhere.
- **Local**: Developer manages tools/configs manually.
  Codespaces improve consistency but may have cost implications.

Question 78:

How do you manage GitHub Actions concurrency to avoid duplicate runs?
Answer:
By defining concurrency groups in workflows. This ensures only the latest run for a branch is active, canceling previous in-progress runs to save resources.

Question 79:

How do you handle cross-organization contributions in GitHub?
Answer:
By using forks, PR reviews, and branch protection. For enterprises, GitHub Enterprise Cloud allows **internal repositories** accessible only to org members, with secure collaboration policies.

Question 80:

How would you implement canary testing for microservices deployed via GitHub Actions?
Answer:
By deploying the new version to a small subset of users or traffic, monitoring behavior, and progressively increasing rollout. If issues appear, rollback happens before full deployment.

Question 81:

How do you implement **GitOps with GitHub**?
Answer:
By treating Git as the single source of truth. Any infra/app changes are pushed to GitHub, and GitOps tools (ArgoCD, Flux) automatically sync clusters/environments from the repo.

Question 82:

How do you handle **multi-cloud deployments** with GitHub Actions?
Answer:

- Use matrix workflows to deploy in parallel to AWS, GCP, and Azure.
- Authenticate via OIDC federation.

- Centralize secrets and configs using Vault or cloud-specific secret managers.

Question 83:

How do you enforce **governance policies** across GitHub repos?
Answer:

- Use GitHub Organization rulesets.
- Enforce mandatory code reviews, security scans, and branch protection.
- Run policy-as-code (OPA/Conftest) checks in CI before merging.

Question 84:

How would you design a **disaster recovery strategy** for GitHub Enterprise Server?
Answer:

- Enable geo-replication.
- Take scheduled backups of repos, configs, and actions runners.
- Implement failover between primary and DR instances.

Question 85:

How do you implement **progressive delivery** with GitHub?
Answer:
By integrating GitHub Actions with feature flag tools (LaunchDarkly, Unleash) to gradually enable features for subsets of users. Rollout/rollback happens without redeploying code.

Question 86:

What's the role of **GitHub Advanced Security (GHAS)** in DevSecOps?
Answer:
It provides CodeQL scanning, secret scanning, and dependency scanning at the repo level. Helps shift security left by detecting vulnerabilities during development.

Question 87:

How do you handle **compliance requirements (ISO, SOC2, HIPAA)** with GitHub?
Answer:

- Use GitHub Enterprise with audit logs.
- Enforce MFA/SSO.
- Automate evidence collection (code reviews, test results).

- Integrate with GRC tools for compliance reporting.

Question 88:

How do you deal with **secrets rotation** in GitHub Actions?
Answer:

- Use external secrets managers (AWS Secrets Manager, Vault, Azure Key Vault).
- Rotate secrets automatically and update GitHub Actions via API.
- Avoid long-lived credentials by using OIDC.

Question 89:

How would you **scale GitHub Actions runners** for thousands of jobs?
Answer:

- Use self-hosted runners in Kubernetes or autoscaling VMs.
- Group runners by workload type (build, test, deploy).
- Use caching/artifacts to reduce redundant work.

Question 90:

How do you ensure **data residency** compliance in GitHub?
Answer:
With GitHub Enterprise Server, you can control where data is stored (on-prem/cloud region). For Enterprise Cloud, GitHub provides regional hosting options (US/EU).

Question 91:

How do you optimize **merge strategies** in GitHub for large teams?
Answer:

- **Squash merges** → cleaner history.
- **Rebase merges** → linear history.
- **Merge commits** → retain branch context.
  Choice depends on auditing, debugging, and team preference.

Question 92:

How would you integrate **SAST + DAST** with GitHub workflows?
Answer:

- **SAST**: Run CodeQL, SonarQube, or Bandit in GitHub Actions.
- **DAST**: Trigger dynamic scans using tools like OWASP ZAP post-deployment.
- Block merges if high-severity issues are found.

Question 93:

How do you implement **multi-tenancy** in GitHub Enterprise?
Answer:
By using organizations for business units, teams for sub-groups, and repository access controls. Combine with SSO for identity federation.

Question 94:

How do you perform **cost optimization** for GitHub Actions?
Answer:

- Cancel duplicate workflows with concurrency.
- Use self-hosted runners for heavy jobs.
- Cache dependencies.
- Trigger workflows only on relevant changes.

Question 95:

How do you **migrate from Bitbucket/GitLab to GitHub**?
Answer:

- Use GitHub's **importer tool** or `git clone --mirror` + `git push --mirror`.
- Migrate CI/CD pipelines to GitHub Actions.
- Recreate repo permissions and branch policies.

Question 96:

How would you implement **zero-downtime deployments** with GitHub?
Answer:
By using rolling updates, canary releases, or blue-green deployments via Actions workflows integrated with Kubernetes/ALB/NGINX.

Question 97:

What's the difference between **GitHub Packages** and external artifact registries?
Answer:

- **GitHub Packages**: tightly integrated, supports npm, Maven, Docker, NuGet.
- **External**: (Artifactory, Nexus) offer broader enterprise features, caching, and multi-cloud distribution.

Question 98:

How do you enable **audit logging for GitHub Actions workflows**?
Answer:
By capturing workflow execution logs, publishing them to SIEM systems, and correlating with GitHub's native audit logs for full visibility of deployments and changes.

Question 99:

How do you implement **multi-region failover** for GitHub-hosted apps?
Answer:
By deploying apps in multiple regions via Actions workflows, using DNS-based failover (Route53, Cloudflare), and keeping database replication in sync.

Question 100:

How do you integrate **machine learning workflows** into GitHub Actions?
Answer:
By setting up pipelines that handle data prep, model training, testing, and deployment. Use caching for datasets, GPU-enabled runners for training, and automate model promotion after validation.